Parallel Sudoku Solver

CSE 691. Parallel Programming and Multithreading

Homework 3 Report

Anagha Girish Dhekne

603845786

1. Introduction

Sudoku is a logic-based, combinatorial number-placement puzzle that has gained immense popularity worldwide due to its simple rules and challenging gameplay. It typically comprises a 9x9 grid divided into nine 3x3 sub grids. The objective of the puzzle is to fill each cell in the grid with a digit from 1 to 9, such that each row, column, and 3x3 sub grid contains all the digits from 1 to 9 without repetition. The puzzle starts with some cells already filled, known as "givens" or "clues," which serve as the initial constraints for solving the puzzle.

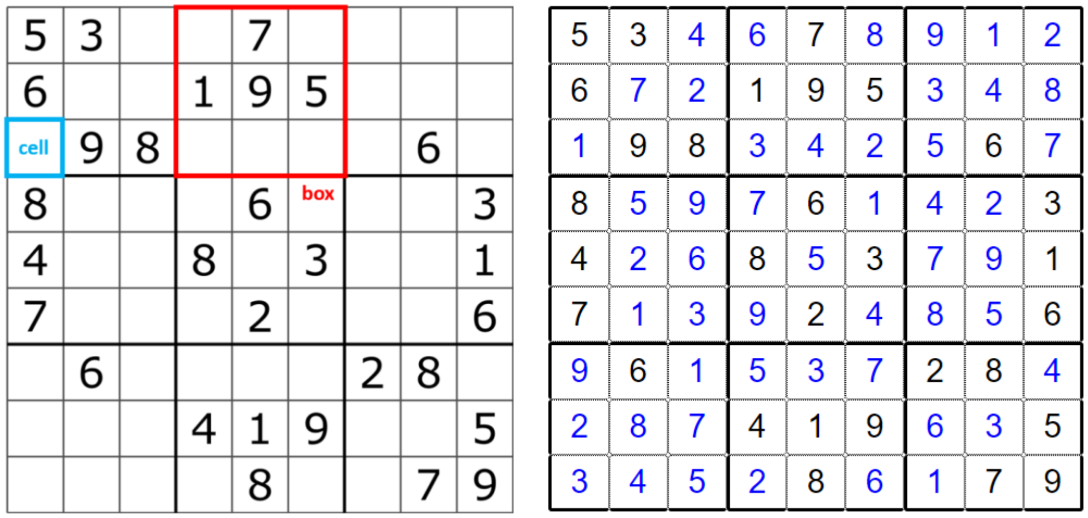


Figure 1: A typical Sudoku puzzle (left) and its solution (right)

Rules of Sudoku

- Single Occurrence Rule: Each digit from 1 to 9 must appear exactly once in each row, column, and 3x3 sub grid.
- Initial Constraints: Some cells in the grid are pre-filled with digits, and these initial values cannot be changed.
- Logical Deduction: Players use deduction and elimination strategies to fill in the remaining empty cells, ensuring that the puzzle adheres to the single occurrence rule.

Characteristics of Sudoku

- Symmetry: Sudoku puzzles often exhibit symmetrical patterns, both in terms of the initial clues provided and the solution itself. Symmetry enhances the aesthetic appeal of puzzles and adds to the challenge of solving them.
- Variable Difficulty: Sudoku puzzles come in varying levels of difficulty, ranging from easy to fiendish. The difficulty depends on factors such as the number of initial clues provided, the complexity of the logical deductions required, and the presence of multiple solution paths.
- Uniqueness: A well-constructed Sudoku puzzle has a unique solution, meaning there is only one valid way to fill in the grid while satisfying all the rules. Ensuring puzzle uniqueness is essential for maintaining the integrity of the puzzle-solving experience.

Computational Complexity of Sudoku

Despite its simple rules, Sudoku is an NP-complete problem, which means that solving it optimally is computationally intractable for large grids. The computational complexity of Sudoku stems from the exponential growth in the number of possible configurations as the grid size increases. While traditional solving techniques rely on exhaustive search or backtracking algorithms, these methods become increasingly inefficient for large puzzles or puzzles with limited initial clues.

Efforts to analyze the computational complexity of Sudoku have led to the development of various solving techniques and algorithmic optimizations. These include constraint propagation methods, heuristic search algorithms, and advanced logical deduction strategies aimed at reducing the search space and improving solving efficiency.

With the exponential growth of computational power and the advent of parallel computing architectures, there exists a compelling opportunity to explore parallel algorithms for Sudoku solving, aiming to expedite the resolution of puzzles and enhance solving scalability.

This project endeavors to design, implement, and evaluate a parallel Sudoku solver leveraging the computational capabilities of modern parallel computing platforms. By harnessing the power of parallelism, the solver seeks to reduce solving time for Sudoku puzzles of varying complexities.

2. Objective

The project aims to investigate sequential and parallel algorithms for solving Sudoku puzzles, with the overall goal of improving solving efficiency and scalability. Sequential solvers, such as backtracking and brute-force techniques, will be used to construct a benchmark for performance evaluation. Furthermore, parallel solution algorithms based on parallel computing techniques such as multithreading will be developed to take use of modern hardware architectures' computational capabilities. The project's goal is to quantify the efficiency benefits made possible by parallel algorithms, particularly when tackling large and complex Sudoku puzzles. Algorithmic optimizations will be researched to improve both sequential and parallel solving strategies, with a particular emphasis on lowering search space and improving constraint propagation.

3. Problem Statement

Develop a parallel Sudoku solver in C++ to efficiently solve Sudoku puzzles by distributing the workload across multiple processing units simultaneously.

4. Implementation

The Sudoku solver project was implemented in C++ utilizing object-oriented programming principles. The implementation consists of several classes and functionalities aimed at solving Sudoku puzzles efficiently. Below is a breakdown of the key components and their functionalities:

- **Main Functionality:** The main function serves as the entry point of the program, responsible for parsing command-line arguments, reading input files, selecting the solving mode, and initiating the solving process. It supports four modes:
    1. Sequential mode with backtracking algorithm.

2. Sequential mode with brute force algorithm.
3. Parallel mode utilizing multithreading.
4. All modes combined, providing solutions for comparison.

- **Brute Force:** Inherits from the SudokuSolver class and implements a brute-force approach to solve Sudoku puzzles. Utilizes a recursive algorithm to systematically try all possible combinations until a solution is found. For each empty cell, it iterates through all possible values (1 to N, where N is the size of the Sudoku board) and recursively tries each value, backtracking if a solution is not valid. For each number, it checks if placing that number at the current cell violates Sudoku rules using the isValid() function. If the number is valid, it sets the number in the current cell and recursively calls solve() to proceed to the next empty cell. If the recursive call returns true (indicating that a solution has been found), it sets the solved flag to true and returns true. If the recursive call returns false (indicating that the current assignment does not lead to a solution), it resets the cell value and tries the next number. If no valid number is found for the current cell, it returns false, indicating that no solution could be found for the current puzzle configuration.
- **Back Tracking:** Inherits from the SudokuSolver class and implements a backtracking algorithm to solve Sudoku puzzles. Utilizes a recursive approach that incrementally fills cells and backtracks when reaching an invalid state. It first checks if the puzzle has already been solved by checking the solved flag. If not, it checks if all cells in the Sudoku board are filled using the checkIfAllFilled() function. If all cells are filled, it sets the solved flag to true and returns true. If not, it finds the next empty cell using the find_empty() function. It then iterates through all possible numbers (1 to N) and recursively tries each number at the empty cell. For each number, it checks if placing that number at the current cell violates Sudoku rules using the isValid() function. If the number is valid, it sets the number in the current cell and recursively calls solve() to proceed to the next empty cell. If the recursive call returns true (indicating that a solution has been found), it sets the solved flag to true and returns true. If the recursive call returns false (indicating that the current assignment does not lead to a solution), it resets the cell value and tries the next number. If no valid number is found for the current cell, it returns false, indicating that no solution could be found for the current puzzle configuration.
- **Parallel:** Inherits from the SudokuSolver class and implements a parallel solving approach using multithreading. Divides the puzzle-solving task into multiple threads, each responsible for solving a subset of the puzzle. Utilizes mutexes to synchronize access to shared data structures to avoid race conditions. It calculates the number of threads to use based on available hardware concurrency and divides the Sudoku board into equal parts for each thread to solve. Each thread independently solves its assigned subset of the Sudoku board using a recursive approach like the backtracking algorithm. Each thread checks if the assigned subset of cells is filled. If not, it finds the next empty cell and recursively tries all possible numbers, like the backtracking algorithm. If a thread finds a solution, it sets a flag indicating success and terminates. Other threads continue their search until a solution is found or all possibilities are exhausted. Once a solution is found or all threads finish execution without finding a solution,

the main thread checks the status of each thread and returns true if any thread found a solution, otherwise returns false.

Various helper functions are implemented within each solver class to validate cell assignments, check for the presence of empty cells, and determine the validity of the puzzle state. The program dynamically selects the appropriate solver based on the chosen mode and executes it. Upon successful solution or termination, the program generates output files containing the solved Sudoku puzzle along with the time taken for solving. In parallel mode, the number of threads is determined based on the hardware concurrency to optimize performance. Proper synchronization mechanisms, such as mutexes, are employed to ensure thread safety and prevent data races.

5. Result

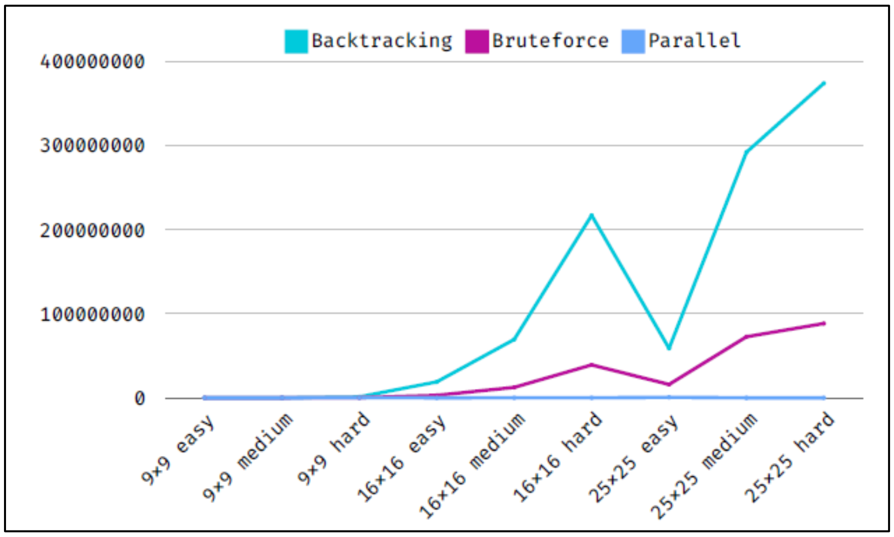


Figure 2: Summary of execution time in microseconds

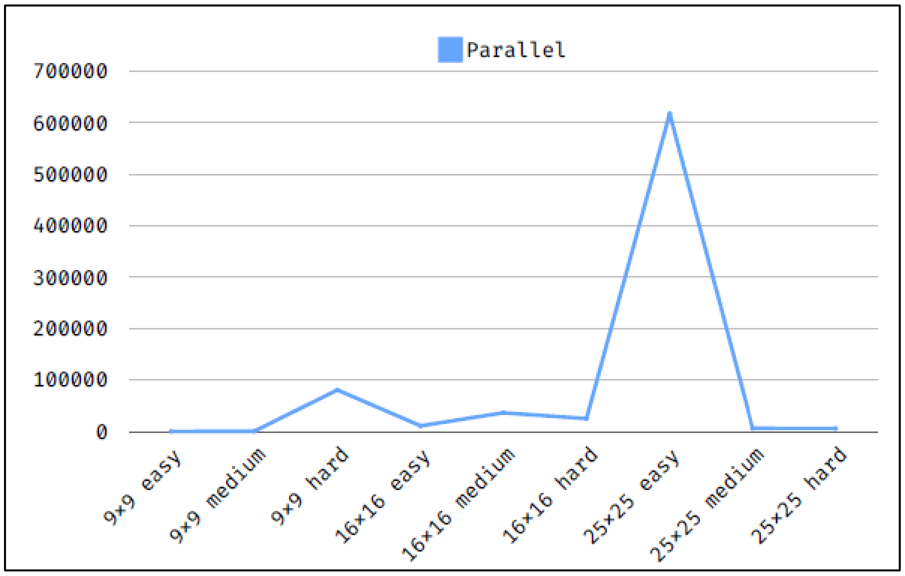


Figure 3: Summary of execution time in microseconds for parallel approach

Size	Difficulty	Back Tracking	Brute force	Parallel
9x9	Easy	331	33	344
9x9	Medium	766	194	541
9x9	Hard	1149724	199288	80769
16x16	Easy	19008462	2992406	11157
16x16	Medium	69479044	12447560	36432
16x16	Hard	217103753	39186530	25222
25x25	Easy	58947119	15920735	617783
25x25	Medium	292268481	72665737	6304
25x25	Hard	374281434	88452821	5984

Table 1: Summary of execution time in microseconds

These results demonstrate the efficiency of each algorithm in solving Sudoku puzzles of varying sizes and difficulty levels. The parallel algorithm shows promising results, especially for larger puzzles, leveraging parallel processing to significantly reduce solving times compared to sequential approaches.

6. Conclusion

In this project, we investigated the performance of three different algorithms, namely Backtracking, Brute Force, and Parallel, for solving Sudoku puzzles of various sizes and difficulty levels. The algorithms were evaluated based on their execution times measured in microseconds across multiple rounds of testing.

The results indicate that the Parallel algorithm consistently outperforms the sequential algorithms, particularly for larger puzzle sizes. This can be attributed to the parallelization of the solving process, which allows for concurrent exploration of solution paths, leveraging the capabilities of multi-core processors. As a result, the Parallel algorithm demonstrates significant reductions in solving times, especially for puzzles with higher complexity and larger dimensions.

7. ReadMe

Run: To run the solver, supply the executable with command-line arguments for input sudoku file, and modes. I have all the input files in the submitted zip file under ‘input’ folder. It supports four modes:

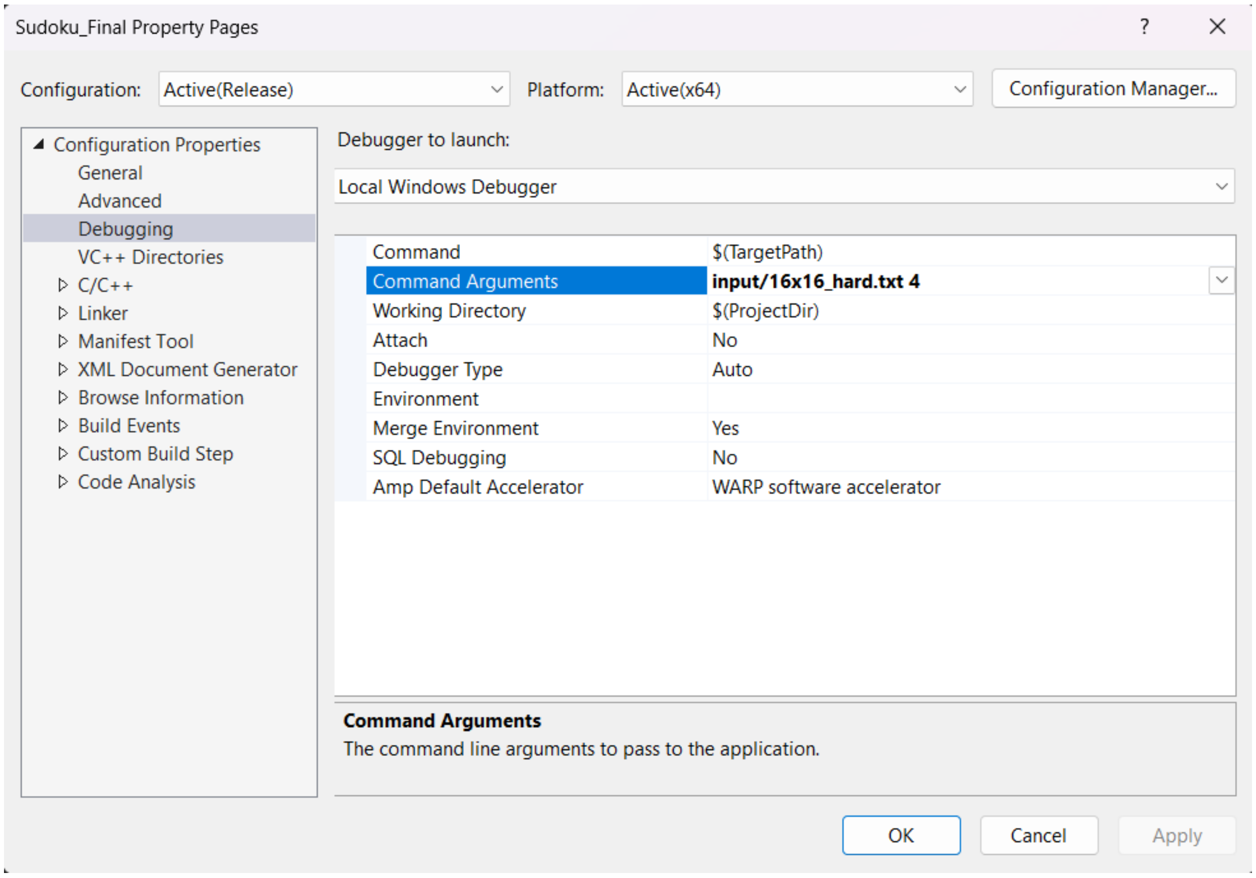
- Sequential mode with backtracking algorithm.
- Sequential mode with brute force algorithm.
- Parallel mode utilizing multithreading.
- All modes combined, providing solutions for comparison.

Example of command line arguments:

```
./sudoku_file <PATH_TO_INPUT_FILE> <MODE>
```

```
./sudoku_file input/16x16_hard.txt 4
```

Example of how I passed arguments in Visual Studio: ‘Select in VS - Project -> Properties’



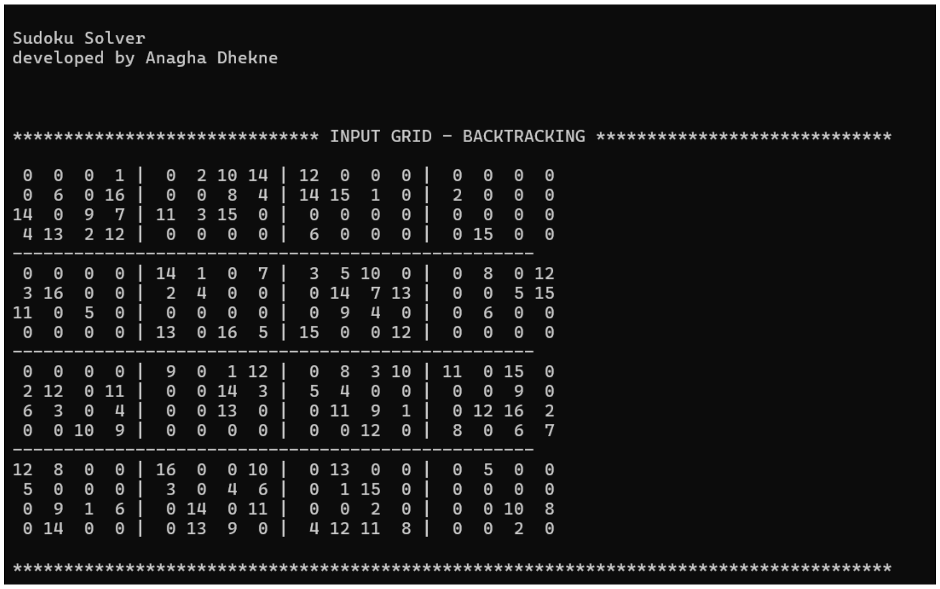
Configuration: Active(Release) Platform: Active(x64)

Debugger to launch: Local Windows Debugger

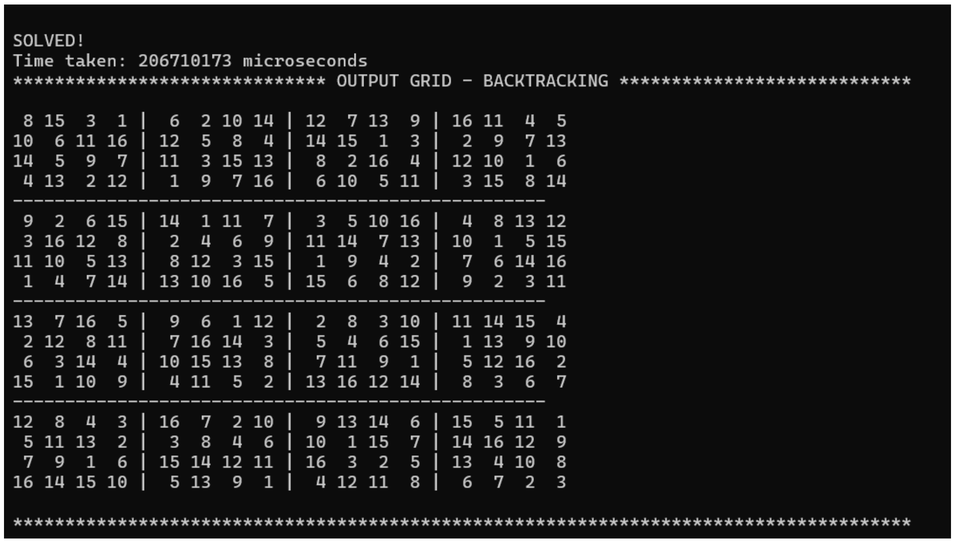
Command	$(TargetPath)
Command Arguments	**input/16x16_hard.txt 4**
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No
Amp Default Accelerator	WARP software accelerator

Command Arguments
The command line arguments to pass to the application.

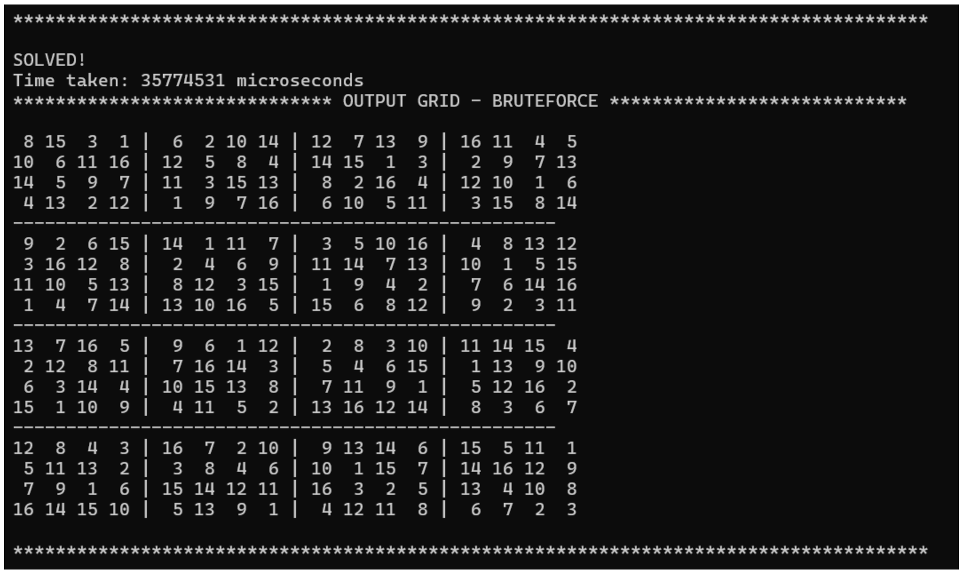
Output



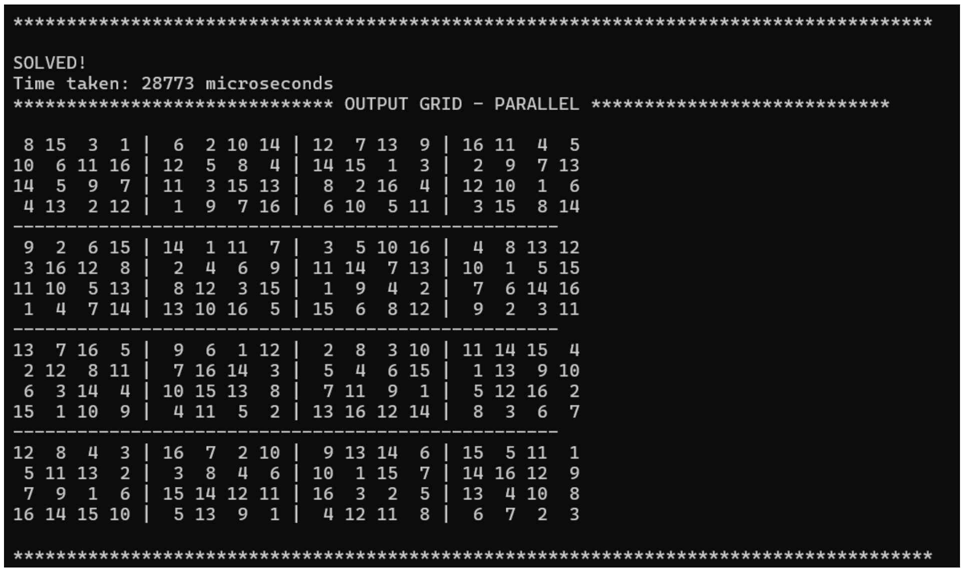
```
Sudoku Solver
developed by Anagha Dhekne



****************************** INPUT GRID - BACKTRACKING ******************************

  0  0  0  1 |  0  2 10 14 | 12  0  0  0 |  0  0  0  0
  0  6  0 16 |  0  0  8  4 | 14 15  1  0 |  2  0  0  0
 14  0  9  7 | 11  3 15  0 |  0  0  0  0 |  0  0  0  0
  4 13  2 12 |  0  0  0  0 |  6  0  0  0 |  0 15  0  0
 ---------------------------------------------------
  0  0  0  0 | 14  1  0  7 |  3  5 10  0 |  0  8  0 12
  3 16  0  0 |  2  4  0  0 |  0 14  7 13 |  0  0  5 15
 11  0  5  0 |  0  0  0  0 |  0  9  4  0 |  0  6  0  0
  0  0  0  0 | 13  0 16  5 | 15  0  0 12 |  0  0  0  0
 ---------------------------------------------------
  0  0  0  0 |  9  0  1 12 |  0  8  3 10 | 11  0 15  0
  2 12  0 11 |  0  0 14  3 |  5  4  0  0 |  0  0  9  0
  6  3  0  4 |  0  0 13  0 |  0 11  9  1 |  0 12 16  2
  0  0 10  9 |  0  0  0  0 |  0  0 12  0 |  8  0  6  7
 ---------------------------------------------------
 12  8  0  0 | 16  0  0 10 |  0 13  0  0 |  0  5  0  0
  5  0  0  0 |  3  0  4  6 |  0  1 15  0 |  0  0  0  0
  0  9  1  6 |  0 14  0 11 |  0  0  2  0 |  0  0 10  8
  0 14  0  0 |  0 13  9  0 |  4 12 11  8 |  0  0  2  0

***************************************************************************************
```



```
SOLVED!
Time taken: 206710173 microseconds
****************************** OUTPUT GRID - BACKTRACKING ****************************

 8 15  3  1 |  6  2 10 14 | 12  7 13  9 | 16 11  4  5
10  6 11 16 | 12  5  8  4 | 14 15  1  3 |  2  9  7 13
14  5  9  7 | 11  3 15 13 |  8  2 16  4 | 12 10  1  6
 4 13  2 12 |  1  9  7 16 |  6 10  5 11 |  3 15  8 14
-------------------------------------------------------
 9  2  6 15 | 14  1 11  7 |  3  5 10 16 |  4  8 13 12
 3 16 12  8 |  2  4  6  9 | 11 14  7 13 | 10  1  5 15
11 10  5 13 |  8 12  3 15 |  1  9  4  2 |  7  6 14 16
 1  4  7 14 | 13 10 16  5 | 15  6  8 12 |  9  2  3 11
-------------------------------------------------------
13  7 16  5 |  9  6  1 12 |  2  8  3 10 | 11 14 15  4
 2 12  8 11 |  7 16 14  3 |  5  4  6 15 |  1 13  9 10
 6  3 14  4 | 10 15 13  8 |  7 11  9  1 |  5 12 16  2
15  1 10  9 |  4 11  5  2 | 13 16 12 14 |  8  3  6  7
-------------------------------------------------------
12  8  4  3 | 16  7  2 10 |  9 13 14  6 | 15  5 11  1
 5 11 13  2 |  3  8  4  6 | 10  1 15  7 | 14 16 12  9
 7  9  1  6 | 15 14 12 11 | 16  3  2  5 | 13  4 10  8
16 14 15 10 |  5 13  9  1 |  4 12 11  8 |  6  7  2  3

***************************************************************************************
```



```
***************************************************************************************

SOLVED!
Time taken: 35774531 microseconds
****************************** OUTPUT GRID - BRUTEFORCE ****************************

 8 15  3  1 |  6  2 10 14 | 12  7 13  9 | 16 11  4  5
10  6 11 16 | 12  5  8  4 | 14 15  1  3 |  2  9  7 13
14  5  9  7 | 11  3 15 13 |  8  2 16  4 | 12 10  1  6
 4 13  2 12 |  1  9  7 16 |  6 10  5 11 |  3 15  8 14
-------------------------------------------------------
 9  2  6 15 | 14  1 11  7 |  3  5 10 16 |  4  8 13 12
 3 16 12  8 |  2  4  6  9 | 11 14  7 13 | 10  1  5 15
11 10  5 13 |  8 12  3 15 |  1  9  4  2 |  7  6 14 16
 1  4  7 14 | 13 10 16  5 | 15  6  8 12 |  9  2  3 11
-------------------------------------------------------
13  7 16  5 |  9  6  1 12 |  2  8  3 10 | 11 14 15  4
 2 12  8 11 |  7 16 14  3 |  5  4  6 15 |  1 13  9 10
 6  3 14  4 | 10 15 13  8 |  7 11  9  1 |  5 12 16  2
15  1 10  9 |  4 11  5  2 | 13 16 12 14 |  8  3  6  7
-------------------------------------------------------
12  8  4  3 | 16  7  2 10 |  9 13 14  6 | 15  5 11  1
 5 11 13  2 |  3  8  4  6 | 10  1 15  7 | 14 16 12  9
 7  9  1  6 | 15 14 12 11 | 16  3  2  5 | 13  4 10  8
16 14 15 10 |  5 13  9  1 |  4 12 11  8 |  6  7  2  3

***************************************************************************************
```



```
***************************************************************************************

SOLVED!
Time taken: 28773 microseconds
****************************** OUTPUT GRID - PARALLEL ****************************

 8 15  3  1 |  6  2 10 14 | 12  7 13  9 | 16 11  4  5
10  6 11 16 | 12  5  8  4 | 14 15  1  3 |  2  9  7 13
14  5  9  7 | 11  3 15 13 |  8  2 16  4 | 12 10  1  6
 4 13  2 12 |  1  9  7 16 |  6 10  5 11 |  3 15  8 14
-------------------------------------------------------
 9  2  6 15 | 14  1 11  7 |  3  5 10 16 |  4  8 13 12
 3 16 12  8 |  2  4  6  9 | 11 14  7 13 | 10  1  5 15
11 10  5 13 |  8 12  3 15 |  1  9  4  2 |  7  6 14 16
 1  4  7 14 | 13 10 16  5 | 15  6  8 12 |  9  2  3 11
-------------------------------------------------------
13  7 16  5 |  9  6  1 12 |  2  8  3 10 | 11 14 15  4
 2 12  8 11 |  7 16 14  3 |  5  4  6 15 |  1 13  9 10
 6  3 14  4 | 10 15 13  8 |  7 11  9  1 |  5 12 16  2
15  1 10  9 |  4 11  5  2 | 13 16 12 14 |  8  3  6  7
-------------------------------------------------------
12  8  4  3 | 16  7  2 10 |  9 13 14  6 | 15  5 11  1
 5 11 13  2 |  3  8  4  6 | 10  1 15  7 | 14 16 12  9
 7  9  1  6 | 15 14 12 11 | 16  3  2  5 | 13  4 10  8
16 14 15 10 |  5 13  9  1 |  4 12 11  8 |  6  7  2  3

***************************************************************************************
```

Code

```
#include <iostream>
#include <chrono>
#include <vector>
#include <string>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <algorithm>
#include <thread>
#include <future>
#include <mutex>

using namespace std;

using Position = pair<int, int>;

// Abstract base class for Sudoku solvers
class SudokuSolver {
protected:
        vector<vector<int>> board;  // Sudoku board
        int boardSize;              // Size of the board (e.g., 9x9)
        int boxSize;                // Size of each box (e.g., 3x3)
        bool solved;                // Flag to indicate if the puzzle is solved
        int emptyCells;             // Number of empty cells in the puzzle
        int emptyCellsValue;        // Value used to represent empty cells
        int minValue;               // Minimum value allowed in the puzzle
        int maxValue;               // Maximum value allowed in the puzzle
```

```
public:
    // Constructor to initialize the Sudoku solver with input from a file
    SudokuSolver(const string& filename) {
        // Open input file
        ifstream inputFile(filename);
        if (!inputFile) {
            cerr << "Error opening file " << filename << endl;
            exit(1);
        }

        // Read board size and initialize board
        inputFile >> boardSize;
        boxSize = sqrt(boardSize);
        board.resize(boardSize, vector<int>(boardSize));

        // Read board values and count empty cells
        int numEmptyCells = 0;
        for (int row = 0; row < boardSize; ++row) {
            for (int col = 0; col < boardSize; ++col) {
                int value;
                inputFile >> value;
                board[row][col] = value;
                numEmptyCells += (value == 0);
            }
        }
        emptyCells = numEmptyCells;
        emptyCellsValue = 0;
        minValue = 1;
        maxValue = boardSize;
        solved = false;

        inputFile.close();
}

// Pure virtual function to solve the Sudoku puzzle (implemented by derived classes)
virtual bool solve() = 0;

void printSolution() const {
        // Print the solved Sudoku board
        for (int i = 0; i < boardSize; ++i) {
                if (i % boxSize == 0 && i != 0) {
                        cout << string(boardSize * 3 + boxSize - 1, '-') << endl;
                }

                for (int j = 0; j < boardSize; ++j) {
                        if (j % boxSize == 0 && j != 0) {
                                cout << "| ";
                        }

                        cout << setw(2) << board[i][j] << " ";
                }
                cout << endl;
        }
}
```

```
void writeOutput(const string& filename, long long duration) const {
    // Write the solved Sudoku board and time taken to an output file
    ofstream outputFile(filename);

    int digit = log10(boardSize) + 1;

    for (int r = 0; r < boardSize; ++r) {
        for (int c = 0; c < boardSize; ++c) {
            outputFile << setw(digit) << board[r][c];

            if (c != boardSize - 1) {
                outputFile << " ";
            }

            if (c % boxSize == (boxSize - 1) && c != boardSize - 1) {
                outputFile << "  ";
            }
        }

        if (r != boardSize - 1) {
            outputFile << "\n";
            if (r % boxSize == (boxSize - 1)) {
                outputFile << "\n";
            }
        }
    }

    outputFile << "\n\n";
```

```
                outputFile << "Time taken: " << duration << " microseconds";

                outputFile.close();
        }
};

// Derived class for solving Sudoku using brute force approach
class SudokuSolverBruteForce : public SudokuSolver {
private:
        bool isValidRow(int num, Position pos) const
        {
                // Check if the number is valid in the given row
                for (int i = 0; i < boardSize; ++i)
                {
                        if ((i != pos.second) && (board[pos.first][i] == num)) { return false; }
                }

                return true;
        }

        bool isValidColumn(int num, Position pos) const
        {
                // Check if the number is valid in the given column
                for (int i = 0; i < boardSize; ++i)
                {
                        if ((i != pos.first) && (board[i][pos.second] == num)) { return false; }
                }
        return true;
    }

    bool isValidBox(int num, Position pos) const
    {
        // Check if the number is valid in the given box
        int box_x = pos.first - pos.first % boxSize;
        int box_y = pos.second - pos.second % boxSize;

        for (int i = box_x; i < box_x + boxSize; ++i)
        {
            for (int j = box_y; j < box_y + boxSize; ++j)
            {
                if ((i != pos.first || j != pos.second) && (board[i][j] == num)) { return
false; }
            }
        }

        return true;
    }

    bool isValid(int num, Position pos) const
    {
        // Check if the number is valid in the given position
        return isValidRow(num, pos) && isValidColumn(num, pos) && isValidBox(num,
pos);
    }

    // Set a number at a given position on the board
```

```
        void set_board_data(int row, int col, int num) { board[row][col] = num; }

    public:
        // Constructor to initialize the Sudoku solver with input from a file
        SudokuSolverBruteForce(const string& filename) : SudokuSolver(filename) {}

        // Solve the Sudoku puzzle using brute force approach
        bool solve() override {
            for (int row = 0; row < boardSize; ++row) {
                for (int col = 0; col < boardSize; ++col) {
                    if (board[row][col] == emptyCellsValue) {
                        for (int num = minValue; num <= maxValue; ++num) {
                            Position pos = make_pair(row, col);

                            if (isValid(num, pos)) {
                                set_board_data(row, col, num);

                                if (solve()) {
                                    solved = true;
                                    return true;
                                }
                                else {
                                    set_board_data(row,                    col,
    emptyCellsValue);
                                }
                            }
                        }
                        return false;
                    }
                }
            }

            solved = true;
            return true;
        }
};

// Derived class for solving Sudoku using backtracking approach
class SudokuSolverBackTracking : public SudokuSolver {
private:
        bool isValidRow(int num, Position pos) const
        {
            // Check if the number is valid in the given row
            for (int i = 0; i < boardSize; ++i)
            {
                if ((i != pos.second) && (board[pos.first][i] == num)) { return false; }
            }

            return true;
        }

        bool isValidColumn(int num, Position pos) const
        {
            // Check if the number is valid in the given column
            for (int i = 0; i < boardSize; ++i)
            {
                if ((i != pos.first) && (board[i][pos.second] == num)) { return false; }
            }

            return true;
        }

        bool isValidBox(int num, Position pos) const
        {
            // Check if the number is valid in the given box
            int box_x = pos.first - pos.first % boxSize;
            int box_y = pos.second - pos.second % boxSize;

            for (int i = box_x; i < box_x + boxSize; ++i)
            {
                for (int j = box_y; j < box_y + boxSize; ++j)
                {
                    if ((i != pos.first || j != pos.second) && (board[i][j] == num)) { return
false; }
                }
            }

            return true;
        }

        bool isValid(int num, Position pos) const
        {
            // Check if the number is valid in the given position
            return isValidRow(num, pos) && isValidColumn(num, pos) && isValidBox(num,
pos);
        }
```

```
// Set a number at a given position on the board
void set_board_data(int row, int col, int num) { board[row][col] = num; }

bool checkIfAllFilled() const
{
    // Check if all cells in the board are filled
    for (int i = 0; i < boardSize; ++i)
    {
        for (int j = 0; j < boardSize; ++j)
        {
            if (board[i][j] == emptyCellsValue)
            {
                return false;
            }
        }
    }
    return true;
}

Position find_empty () const
{
    // Find the next empty cell in the board
    Position empty_cell;
    bool stop = false;

    for (int i = 0; i < boardSize; ++i)
    {
```

```
                for (int j = 0; j < boardSize; ++j)
                {
                        if (board[i][j] == emptyCellsValue)
                        {
                                empty_cell = make_pair(i, j);
                                stop = true;
                                break;
                        }
                }
                if (stop) { break; }
        }

        return empty_cell;  // (row, col)
}

vector<int> getNumbersInRow(int indexOfRows) const
{
        // Get all the numbers present in the given row
        vector<int> numbersInRow;

        for (int col = 0; col < boardSize; ++col)
        {
                int num = board[indexOfRows][col];
                if (num == emptyCellsValue) continue;
                numbersInRow.push_back(num);
        }

        return numbersInRow;
```

```
}

vector<int> getNumbersInCol(int indexOfColumns) const
{
    // Get all the numbers present in the given column
    vector<int> numbersInCol;

    for (int row = 0; row < boardSize; ++row)
    {
        int num = board[row][indexOfColumns];
        if (num == emptyCellsValue) continue;
        numbersInCol.push_back(num);
    }

    return numbersInCol;
}

bool isUnique(int num, Position pos) const
{
    // Check if the number is unique in its row, column, and box
    int local_row = pos.first % boxSize;
    int local_col = pos.second % boxSize;

    int box_x = floor(pos.first / boxSize);
    int box_y = floor(pos.second / boxSize);

    for (int i = ((local_row == 0) ? 1 : 0); i < boxSize; ++i)
    {
            if (i == local_row) { continue; }
            vector<int> numbersInRow = getNumbersInRow(box_x * boxSize + i);
            if (find(numbersInRow.begin(), numbersInRow.end(), num) ==
numbersInRow.end())
            {
                return false;
            }
        }

        for (int j = ((local_col == 0) ? 1 : 0); j < boxSize; ++j)
        {
            if (j == local_col) { continue; }
            vector<int> numbersInCol = getNumbersInCol(box_y * boxSize + j);
            if (find(numbersInCol.begin(), numbersInCol.end(), num) ==
numbersInCol.end())
            {
                return false;
            }
        }

        return true;
    }

    public:
        // Constructor to initialize the Sudoku solver with input from a file
        SudokuSolverBackTracking(const string& filename) : SudokuSolver(filename) {}

        bool solve() override {
            // Solve the Sudoku using backtracking approach
```

```
if (solved) {
        return true;
}

if (checkIfAllFilled()) {
        solved = true;
        return true;
}

Position emptyPos = find_empty();
int row = emptyPos.first;
int col = emptyPos.second;

for (int num = minValue; num <= maxValue; ++num) {
        if (isValid(num, emptyPos)) {
                set_board_data(row, col, num);

                if (isUnique(num, emptyPos)) {
                        num = maxValue + 1;
                }

                if (solve()) {
                        solved = true;
                        return true;
                }
                else {
                        set_board_data(row, col, emptyCellsValue);
                }
```

```
                    }
                }

                solved = false;
                return false;
            }
};

//class SudokuSolverParallel : public SudokuSolver {
//private:
//        mutex mtx;
//
//        bool is_valid(int row, int col, int num) {
//                int box_row = (row / boxSize) * boxSize;
//                int box_col = (col / boxSize) * boxSize;
//
//                for (int i = 0; i < boardSize; i++) {
//                        unique_lock<mutex> lock(mtx);
//                        if (board[row][i] == num || board[i][col] == num ||
//                                board[box_row + i / boxSize][box_col + i % boxSize] == num)
//                                return false;
//                }
//                return true;
//        }
//
//        pair<int, int> find_empty() {
//                for (int i = 0; i < boardSize; i++) {
//                        for (int j = 0; j < boardSize; j++) {
```

```
//                              unique_lock<mutex> lock(mtx);
//                              if (board[i][j] == 0)
//                                      return { i, j };
//                      }
//              }
//              return { -1, -1 };
//      }
//
//      bool solve_sudoku(int& empty_count) {
//              auto find = find_empty();
//              if (find.first == -1)
//                      return true;
//
//              int row = find.first;
//              int col = find.second;
//
//              for (int i = 1; i <= boardSize; i++) {
//                      if (is_valid(row, col, i)) {
//                              unique_lock<mutex> lock(mtx);
//                              board[row][col] = i;
//                              empty_count--;
//                              lock.unlock();
//
//                              if (empty_count == 0) {
//                                      return true; // Puzzle solved, terminate recursion
//                              }
//
//                              if (solve_sudoku(empty_count))
//                                  return true;
//
//                          lock.lock();
//                          board[row][col] = 0;
//                          empty_count++;
//                          lock.unlock();
//
//                  }
//          }
//          return false;
//      }
//
//public:
//      SudokuSolverParallel(const string& filename) : SudokuSolver(filename) {}
//bool solve() override {
//
//          //int num_threads = 3;
//          //if (emptyCells > 50) {
//                  int       num_threads       =       min(emptyCells       /       10+1,
static_cast<int>(thread::hardware_concurrency()));
//          //}
//
//          //int          num_threads          =          min(empty_count/10+1,
static_cast<int>(thread::hardware_concurrency()));
//          vector<future<bool>> futures;
//
//          for (int i = 0; i < num_threads; i++) {
//                  futures.push_back(async(launch::async, [this]() {
//                          return solve_sudoku(emptyCells);
//                              }));
//              }
//
//              for (auto& f : futures) {
//                      if (f.get())
//                              return true;
//              }
//              return false;
//      }
//};

// Derived class for solving Sudoku using parallel approach
class SudokuSolverParallel : public SudokuSolver {
private:
        vector<Position> empty_cells;  // List of empty cells on the board
        vector<vector<bool>> row_used;  // Array to track numbers used in each row
        vector<vector<bool>> col_used;  // Array to track numbers used in each column
        vector<vector<vector<bool>>> box_used;  // Array to track numbers used in each box
        mutex board_mutex;  // Mutex for synchronizing access to the board

        bool is_valid(int row, int col, int num) {
                // Check if the number is valid in the given position
                return !row_used[row][num] && !col_used[col][num] && !box_used[row /
boxSize][col / boxSize][num];
        }

        bool solve_sudoku(int& empty_count) {
                // Solve the Sudoku using parallel approach
                if (empty_count == 0)
        return true;

    int row = empty_cells[empty_count - 1].first;
    int col = empty_cells[empty_count - 1].second;

    for (int i = 1; i <= boardSize; i++) {
        {
            unique_lock<mutex> lock(board_mutex);
            if (!is_valid(row, col, i))
                continue;
        }

        {
            unique_lock<mutex> lock(board_mutex);
            board[row][col] = i;
            row_used[row][i] = true;
            col_used[col][i] = true;
            box_used[row / boxSize][col / boxSize][i] = true;
            empty_count--;
        }

        if (solve_sudoku(empty_count))
            return true;

        {
            unique_lock<mutex> lock(board_mutex);
            board[row][col] = 0;
            row_used[row][i] = false;
```

```
                        col_used[col][i] = false;
                        box_used[row / boxSize][col / boxSize][i] = false;
                        empty_count++;
                    }
                }

                return false;
            }

public:
            SudokuSolverParallel(const string& filename) : SudokuSolver(filename) {
                // Initialize data structures for parallel solving
                row_used.resize(boardSize, vector<bool>(boardSize + 1, false));
                col_used.resize(boardSize, vector<bool>(boardSize + 1, false));
                box_used.resize(boxSize,                                 vector<vector<bool>>(boxSize,
vector<bool>(boardSize + 1, false)));

                for (int i = 0; i < boardSize; i++) {
                    for (int j = 0; j < boardSize; j++) {
                        if (board[i][j] == 0) {
                            empty_cells.emplace_back(i, j);
                        }
                        else {
                            row_used[i][board[i][j]] = true;
                            col_used[j][board[i][j]] = true;
                            box_used[i / boxSize][j / boxSize][board[i][j]] = true;
                        }
                    }
                }
```

```
    }

    bool solve() override {
        int num_threads = min(emptyCells / 10 + 1,
static_cast<int>(thread::hardware_concurrency()));
        vector<future<bool>> futures;

        for (int i = 0; i < num_threads; i++) {
            futures.push_back(async(launch::async, [this]() {
                int empty_count = emptyCells;
                return solve_sudoku(empty_count);
                }));
        }

        for (auto& f : futures) {
            if (f.get())
                return true;
        }

        return false;
    }
};

void runSolver(SudokuSolver& solver, const string& algorithmName, const string& outputFile) {
    cout << "\n" << "****************************** INPUT GRID - " <<
algorithmName << " ******************************" << "\n\n";
    solver.printSolution();
    cout << "\n" <<
"******************************************************************************
*********" << "\n";
```

```
    auto start = chrono::high_resolution_clock::now();
    bool solved = solver.solve();
    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);

    if (solved) {
        cout << "\n" << "SOLVED!" << "\n";
        cout << "Time taken: " << duration.count() << " microseconds" << endl;
        cout << "****************************** OUTPUT GRID - " <<
algorithmName << " ****************************" << "\n\n";
        solver.printSolution();
        solver.writeOutput(outputFile, duration.count());
        cout << "\n" <<
"****************************************************************************
*********" << "\n";
    }
    else {
        cout << "\n" << "NO SOLUTION FOUND!" << "\n";
    }
}

int main(int argc, char** argv) {
    cout << "\n" << "Sudoku Solver" << "\n" << "developed by Anagha Dhekne" << "\n\n\n";

    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " <PATH_TO_INPUT_FILE>" << "\n";
        cerr << "Please try again." << "\n";
        exit(-1);
```

```
	}

	if (argc != 3) {
		cerr << "Usage: " << argv[0] << " <PATH_TO_INPUT_FILE> <MODE>" << "\n";
		cerr << "        1. <MODE>: " << "\n";
		cerr << "            - 1: sequential mode with backtracking algorithm" << "\n";
		cerr << "            - 2: sequential mode with brute force algorithm" << "\n";
		cerr << "            - 3: parallel mode" << "\n";
		cerr << "            - 4: All" << "\n";
		cerr << "Please try again." << "\n";
		exit(-1);
	}

	int mode = stoi(argv[2]);
	string inputFile = argv[1];
	string outputDirectory = "output/";

	switch (mode) {
	case 1: {
		string outputFile = outputDirectory + "solution_backtracking_" +
inputFile.substr(inputFile.rfind('/') + 1);
		SudokuSolverBackTracking solver(inputFile);
		runSolver(solver, "BACKTRACKING", outputFile);
		break;
	}
	case 2: {
		string outputFile = outputDirectory + "solution_bruteforce_" +
inputFile.substr(inputFile.rfind('/') + 1);;
		SudokuSolverBruteForce solver(inputFile);
```

```
            runSolver(solver, "BRUTEFORCE", outputFile);
            break;
        }
        case 3: {
            string outputFile = outputDirectory + "solution_parallel_" + inputFile.substr(inputFile.rfind('/') + 1);;
            // Parallel mode implementation
            SudokuSolverParallel solver(inputFile);
            runSolver(solver, "PARALLEL", outputFile);
            break;
        }
        case 4: {
            string outputFileBacktracking = outputDirectory + "solution_backtracking_" + inputFile.substr(inputFile.rfind('/') + 1);;
            string outputFileBruteForce = outputDirectory + "solution_bruteforce_" + inputFile.substr(inputFile.rfind('/') + 1);;
            string outputFileParallel = outputDirectory + "solution_parallel_" + inputFile.substr(inputFile.rfind('/') + 1);;

            SudokuSolverBackTracking solverBacktracking(inputFile);
            SudokuSolverBruteForce solverBruteForce(inputFile);
            SudokuSolverParallel solver(inputFile);

            runSolver(solverBacktracking, "BACKTRACKING", outputFileBacktracking);
            runSolver(solverBruteForce, "BRUTEFORCE", outputFileBruteForce);
            runSolver(solver, "PARALLEL", outputFileParallel);

            break;
        }
        default: {
                cerr << "Available options for <MODE>: " << "\n";
                cerr << "           - 1: sequential mode with backtracking algorithm" << "\n";
                cerr << "           - 2: sequential mode with brute force algorithm" << "\n";
                cerr << "           - 3: parallel mode" << "\n";
                cerr << "           - 4: All" << "\n";
                cerr << "Please try again." << "\n";
                exit(-1);
        }
        }

        return 0;
}
```