SUDOKU SOLUTION VALIDATOR

Course:

CMPE-180C Sec 01-Operating Systems

Spring 2021

Submitted By:

Group 19

Team Members:

Name	Student Id
Akarsh Chandrashekar	015249819
Rakesh Nagarajappa	015248974

Instructor:

Hungwen Li

Date:

05/29/2021

Contents

Executive Summary

The goal of this project is to verify the solution to a 9x9 sudoku puzzle using two approaches.

First approach is using a single thread and second approach is using multiple threads. A given solution is verified using both the approaches and execution time is also calculated for both the approaches. Based on the execution time recorded, an analysis of performance is made to check which approach was better for this process.

Introduction

Sudoku is an easy to learn logic-based number placement puzzle. The name “Sudoku” is short for a longer expression in Japanese –“Sūjiwadokushinnikagiru” – which means, “the digits are limited to one occurrence. “The standard Sudoku puzzle is a table made up of 9 rows, 9 columns consisting 81 cells. The 9x9 grid itself contains nine 3x3 subgrids. The puzzle starts with given numbers in various positions and the player’s goal is to complete the table such that each row, column, and subgrid contains all the numbers from 1 to 9 exactly once. Although the 9x9 variation of Sudoku is most common, other variations like 16x16(16 rows and 16 columns) exist to increase the puzzles difficulty.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

9x9 Sudoku

Background and Objectives

The aim of the project is to evaluate a given sudoku solution to determine its validity using a single thread and multi-threaded approach.

The objective of this application is to implement a C++ program to validate a sudoku solution. This is implemented in two ways; first approach involves creating multiple threads and assigning these threads the job of validation and second approach would be to perform the same validation without creating multiple threads.

A comparative study between the two approaches involving the time taken for validation is carried out to understand whether the multithreading approach has improved the performance.

Approach and Methodology

Single thread Approach:

A Sudoku will be validated by implementing separate functions which will validate the row, column, and sub grids. Each function will be called sequentially if the previous function check has been executed successfully. The elapsed time for this approach will be stored displayed after the execution is finished.

Multi-thread Approach:

Creating multiple threads that check the following:

- Nine threads to check whether each row contains the digits 1 through 9 without any repetition.
- Nine threads to check whether each column contains the digits 1 through 9 without any repetition.
- Nine threads to check whether each of the 3×3 sub grids contain the digits 1 through 9 without any repetition.

  A total of 27 threads will be created for validating a sudoku puzzle solution.

Passing Parameters to each thread

The parent thread will create the worker threads, passing each worker thread the locations that it must check in the Sudoku puzzle.

Returning Results to the Parent Thread

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. The time taken for the creation and execution of all the 27 threads are also stored to be displayed later.

Passing Parameters to Each Thread

The parent thread will create 10 worker threads and pass each of them with a sudoku board that the thread is responsible to validate .

Returning Results to the Parent Thread

When each thread finishes the assigned task of validating a sudoku board ,it must pass the result back to the parent ,this is done through updating a result vector.

The total time taken by multi thread approach is stored and displayed after the execution is finished. This time is used to compare the execution time with first approach for performance analysis.

Multiple Sudoku validation using a single thread:

Similar to single-threaded approach, we validate 10 different sudoku solutions sequentially in a single thread the elapsed time for this approach will also be displayed to compare the execution time of different approaches.

Multiple Sudoku validation using a multi thread approach: Create multiple threads to check the following criteria :

- Ten threads where each check if a particular sudoku is valid
- Each thread calls function which will validate the rows, columns and subgrids
- The elapsed time for all the threads to complete is computed

Algorithm Design and Implementation

Multiple Thread single solution:

1. A sudoku solution is considered valid if all the rows, columns and sub grid contains only digits from one to nine without any repetition .
2. Initially we Create an integer array of size equal to the number of threads and initialize it to zero ,this array can be accessed by all the threads. Create three functions to check row ,column and grid. These functions take location to be checked that is row ,column and a pointer to the current sudoku as input
3. Each of the functions created in step 2.will check whether the corresponding row, column or grid is valid for the region passed as the parameter. If valid the specific index of null int array created in step2 is updated to 1
4. In the parent thread store the current time and create a variables to store thread ID's using **pthread_t**
5. Create one thread each of the row, column and 3x3 sub grid using the function **pthread_create().** Call the functions created in step2 in each of threads and pass the required parameters encapsulated in a structure.
6. After all the 27 threads are created and executing the control flow should wait for all the threads to complete to determine the result. This is done by calling the function **pthread_join()**
7. Check each value of the integer array created in step 2. If any of the values is found to be not equal to 1 then the Sudoku is invalid else it is valid.
8. Calculate the difference between the current time and the time stored in step 4. Display the elapsed time in microseconds.

Multiple Thread multiple solutions

1. A sudoku solution is considered valid if all the rows, columns and sub grid contains only digits from one to nine without any repetition.
2. Initially we create a Boolean vector of size 10 and set all items to false, this vector can be accessed by all the threads. Create three functions that check row ,column and grid. For each function pass a two-d array representing a sudoku puzzle to be validated.
3. Each of the functions created in step2 will check whether the corresponding row, column or grid is valid for the sudoku solution passed as the parameter. If valid it returns true
4. In the parent thread start clock and store the current time and create a variable to store thread ID's using **pthread_t.**
5. Create thread for each of 10 sudoku solutions to be validated using **pthread_create().**for each thread pass the two-d array as parameters encapsulated in a structure and the function to validate sudoku is called which in turn calls three functions created in step 2 and modifies the vector created in step2 based on what these functions return.
6. After all the 10 threads are created and in execution, we wait for all the threads to complete to determine the result by calling **pthread_join().**
7. Check each value of the Boolean vector created in step 2 the Boolean value of each element indicates if the sudoku executed by the corresponding thread valid or invalid.
8. Capture time and calculate and display elapsed time.

Single Thread single solution:

1. Store the current time in a variable
2. Create 3 functions to validate the row, column and grid and a verify function that calls these 3 functions and returns true only if these 3 functions return true meaning columns grids and rows are valid.
3. Call the function that checks if a sudoku solution is valid this function in turn calls the 3 functions created in step 2 sequentially and if any of the function returns invalid the further execution is stopped and false is returned and the message “sudoku board in not valid” is print.
4. If all the three functions return valid then the function returns true and “sudoku board is valid” is print.
5. Current time is stored in a variable and elapsed time is calculated and time is displayed in microseconds.

Single Thread multiple solutions:

1. Create 3 functions to validate row, column and sub grid and create a function that calls these functions and returns if the sudoku solution is valid or not.
2. These three functions will check each row, column and sub-grid to be valid or not and return true or false based on the validation result and the function that encompasses these functions returns true only if all the three functions return true.
3. Store the current time .iterate over all the 10 sudoku solutions and call the function created in step2 which calls the three functions for column row and grid sequentially.
4. If the validation function returns false print that the “ sudoku (i) is not valid” else print “sudoku (i) is valid” where i is the corresponding sudoku solution.
5. Stire current time in a variable, calculate and print elapsed time in microseconds.

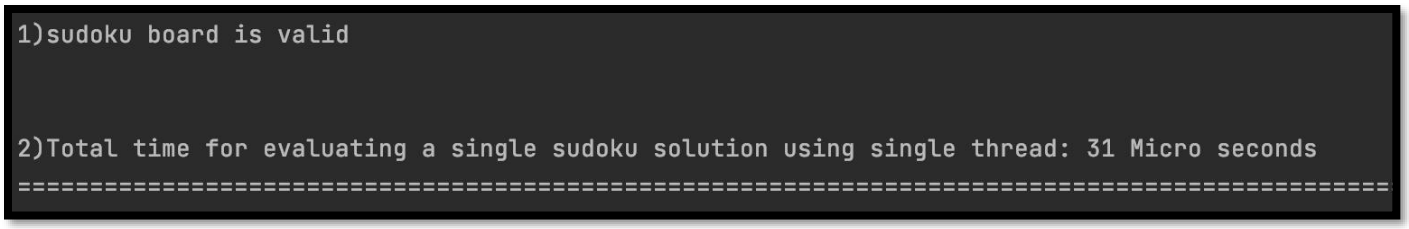
Results, Findings and Analysis

Valid Test Case

Sudoku:

6, 2, 4, 5, 3, 9, 1, 8, 7
5, 1, 9, 7, 2, 8, 6, 3, 4
8, 3, 7, 6, 1, 4, 2, 9, 5
1, 4, 3, 8, 6, 5, 7, 2, 9
9, 5, 8, 2, 4, 7, 3, 6, 1
7, 6, 2, 3, 9, 1, 4, 5, 8
3, 7, 1, 9, 5, 6, 8, 4, 2
4, 9, 6, 1, 8, 2, 5, 7, 3
2, 8, 5, 4, 7, 3, 9, 1, 6

Test Result:



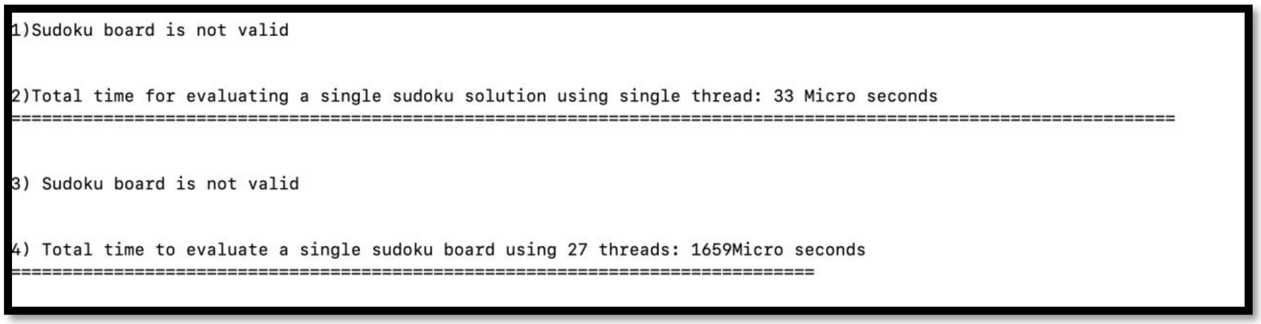
```
1)sudoku board is valid


2)Total time for evaluating a single sudoku solution using single thread: 31 Micro seconds
=====================================================================================
```

Invalid Test Case

Sudoku:

6, 2, 4, 5, 3, 9, 1, 8, 7
5, 1, 9, 7, 2, 8, 6, 3, 4
8, 3, 7, 6, 1, 4, 2, 9, 5
1, 4, 3, 8, 6, 5, 7, 2, 9
9, 5, 8, 2, 4, 7, 3, 6, 1
7, 6, 2, 3, 9, 1, 2, 5, 8
3, 7, 1, 9, 5, 6, 8, 4, 2
4, 9, 6, 1, 8, 2, 5, 7, 3
2, 8, 5, 4, 7, 3, 9, 1, 6

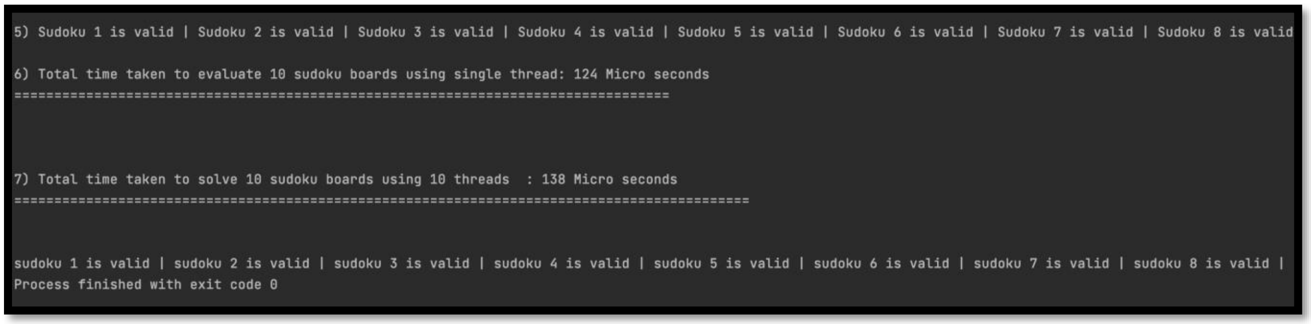
Test Result:



```
1)Sudoku board is not valid


2)Total time for evaluating a single sudoku solution using single thread: 33 Micro seconds
==========================================================================================================


3) Sudoku board is not valid


4) Total time to evaluate a single sudoku board using 27 threads: 1659Micro seconds
============================================================================
```

A single thread execution takes less time compared to multiple threads.

Evaluating 10 sudoku solutions



```
5) Sudoku 1 is valid | Sudoku 2 is valid | Sudoku 3 is valid | Sudoku 4 is valid | Sudoku 5 is valid | Sudoku 6 is valid | Sudoku 7 is valid | Sudoku 8 is valid

6) Total time taken to evaluate 10 sudoku boards using single thread: 124 Micro seconds
==================================================================================



7) Total time taken to solve 10 sudoku boards using 10 threads  : 138 Micro seconds
============================================================================================


sudoku 1 is valid | sudoku 2 is valid | sudoku 3 is valid | sudoku 4 is valid | sudoku 5 is valid | sudoku 6 is valid | sudoku 7 is valid | sudoku 8 is valid |
Process finished with exit code 0
```

The difference between single thread and multithread execution time in multiple solution approach is very less compared to the single solution approach

Conclusion and Recommendation

- For small tasks such as validating regions of a 9*9 sudoku puzzle the single thread is faster than multithread approach. The overhead involved in creating and maintaining the threads in case of multithread approach may cause such performance parity.

- The multithreaded performance of multiple sudoku solution approach has a relatively better performance compare to single solution approach we can draw an inference that the multithreaded approach would have better performance compared to single threaded approach in case of large tasks.

References

- A. Silberschatz, P. Galvin, and G. Gagne, *Operating Systems Concepts*, 9th edition, Wiley, 2013
- Online Resources: www.google.com, www.stackoverflow.com

Appendix: Code

```
//Description:
/*
*   File Name: Sudoku_validator.cpp
* * Authors:
*   Akarsh Chandrashekar
*   Rakesh Nagarajappa
* This program takes a Sudoku puzzle solution as an input and then determines
whether
 * the puzzle solution is valid. This validation is done using single thread a
nd 27 threads.
 * 27 threads are assigned as  follows:
 * 9 for each 3x3 subsection, 9 for the 9 columns, and 9 for the 9 rows.
 * Each thread returns a integer value of 1 indicating that
 * the corresponding region in the puzzle they were responsible for is valid.
 * The program then waits for all threads to complete their execution and
 * checks if the return values of all the threads have been set to 1.
 * If yes, the solution is valid. If not, solution is invalid.
 *
 * This Program also takes multiple sudoku puzzle solution as an input and the
n determines whether
 * all the solutions are valid .This validation is done using single threads a
nd 10 threads
 * where each of the 10  threads are assigned each of 10 boards
 * in the single thread approach the function checks and returns false if any
of the condition is not met
 * in multi thread approach a vector of size 10 is set in the coreesponding in
dex if that solution is invalid
 *
 * This program also displays the total time taken for validation.
 */


#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <iostream>
#include <chrono>
#include <vector>
#include <typeinfo>
#include <string>
```

```
using namespace std;
using namespace std::chrono;
using std::vector;

const int number_of_threads=27;
const int rows=9;
const int columns=9;
int result_final[number_of_threads]={};
vector<bool> res_mul(10,false);

//the sudoku puzzle used for testing validation
int sudoku[9][9]=
            {
                {6, 2, 4, 5, 3, 9, 1, 8, 7},
                {5, 1, 9, 7, 2, 8, 6, 3, 4},
                {8, 3, 7, 6, 1, 4, 2, 9, 5},
                {1, 4, 3, 8, 6, 5, 7, 2, 9},
                {9, 5, 8, 2, 4, 7, 3, 6, 1},
                {7, 6, 2, 3, 9, 1, 4, 5, 8},
                {3, 7, 1, 9, 5, 6, 8, 4, 2},
                {4, 9, 6, 1, 8, 2, 5, 7, 3},
                {2, 8, 5, 4, 7, 3, 9, 1, 6}
        };

struct arguments
{
// initial row
int row;
//initial column
int column;
//the pointer to the sudoku board
int (*sud_ptr)[9];
};

//Prototype of single thread single sudoku validation functions

bool verifyGrid();
bool verifyColumn();
bool verifyRow();
bool verifySudokuBySingleThread();

////Prototype of 27 thread single sudoku validation functions

void *verify_grid(void *parameter);
void *verify_row(void *params);
void *verify_col(void *params);


//Prototype of helper functions to  validate multiple sudoku solutions using m
ultiple threads

void copy2darrays(int params[9][9],int params1[9][9]);
bool verifyGrid_multiple(int sudd[][9]);
bool verifyColumn_multiple(int sudd[][9]);
bool verifyRow_multiple(int sudd[][9]);
bool verifySudokuBySingleThread_multiple(int sudd[][9]);
void *verify_multiple_multithread(void * parameter);


//containers for solving mutiple sudoku solutions using multiple threads

class sud_container
{
    //used to encapsulate  2d arrays so that they can be stored and accessed v
ia a std vector
    public:
    sud_container(int b[][9])
    {
        // copy the 2d array to data member
        for(int i=0;i<9;i++)
        {
            for(int j=0;j<9;j++)
            {
                this->arr[i][j]=b[i][j];
            }
        }

    }
    //2d array public data member
    int arr[9][9];

};


struct multiple_thread_obj
{
    // structure to encapsulate 2d array and iterator info  to be passed to pt
hread create
    multiple_thread_obj(int par[][9],int j)
    {
    copy2darrays(a,par);
    b=j;
    }
    //public data member containing 2d array and an integer
    int a[9][9];
    int b;

};





//Multiple sudoku solutions


int sudoku1[9][9]=
            {
                {6, 2, 4, 5, 3, 9, 1, 8, 7},
                {5, 1, 9, 7, 2, 8, 6, 3, 4},
                {8, 3, 7, 6, 1, 4, 2, 9, 5},
                {1, 4, 3, 8, 6, 5, 7, 2, 9},
                {9, 5, 8, 2, 4, 7, 3, 6, 1},
                {7, 6, 2, 3, 9, 1, 4, 5, 8},
                {3, 7, 1, 9, 5, 6, 8, 4, 2},
                {4, 9, 6, 1, 8, 2, 5, 7, 3},
                {2, 8, 5, 4, 7, 3, 9, 1, 6}

            };

int sudoku2[9][9]=
            {
                {4, 3, 5, 2, 6, 9, 7, 8, 1},
                {6, 8, 2, 5, 7, 1, 4, 9, 3},
                {1, 9, 7, 8, 3, 4, 5, 6, 2},
                {8, 2, 6, 1, 9, 5, 3, 4, 7},
                {3, 7, 4, 6, 8, 2, 9, 1, 5},
                {9, 5, 1, 7, 4, 3, 6, 2, 8},
                {5, 1, 9, 3, 2, 6, 8, 7, 4},
                {2, 4, 8, 9, 5, 7, 1, 3, 6},
                {7, 6, 3, 4, 1, 8, 2, 5, 9}

            };
```

```
int sudoku3[9][9]=
            {
                {1, 5, 2, 4, 8, 9, 3, 7, 6},
                {7, 3, 9, 2, 5, 6, 8, 4, 1},
                {4, 6, 8, 3, 7, 1, 2, 9, 5},
                {3, 8, 7, 1, 2, 4, 6, 5, 9},
                {5, 9, 1, 7, 6, 3, 4, 2, 8},
                {2, 4, 6, 8, 9, 5, 7, 1, 3},
                {9, 1, 4, 6, 3, 7, 5, 8, 2},
                {6, 2, 5, 9, 4, 8, 1, 3, 7},
                {8, 7, 3, 5, 1, 2, 9, 6, 4}

            };

int sudoku4[9][9]=
            {
                {1, 2, 3, 6, 7, 8, 9, 4, 5},
                {5, 8, 4, 2, 3, 9, 7, 6, 1},
                {9, 6, 7, 1, 4, 5, 3, 2, 8},
                {3, 7, 2, 4, 6, 1, 5, 8, 9},
                {6, 9, 1, 5, 8, 3, 2, 7, 4},
                {4, 5, 8, 7, 9, 2, 6, 1, 3},
                {8, 3, 6, 9, 2, 4, 1, 5, 7},
                {2, 1, 9, 8, 5, 7, 4, 3, 6},
                {7, 4, 5, 3, 1, 6, 8, 9, 2}

            };
int sudoku5[9][9]=
            {
                {5, 8, 1, 6, 7, 2, 4, 3, 9},
                {7, 9, 2, 8, 4, 3, 6, 5, 1},
                {3, 6, 4, 5, 9, 1, 7, 8, 2},
                {4, 3, 8, 9, 5, 7, 2, 1, 6},
                {2, 5, 6, 1, 8, 4, 9, 7, 3},
                {1, 7, 9, 3, 2, 6, 8, 4, 5},
                {8, 4, 5, 2, 1, 9, 3, 6, 7},
                {9, 1, 3, 7, 6, 8, 5, 2, 4},
                {6, 2, 7, 4, 3, 5, 1, 9, 8}

            };
int sudoku6[9][9]=
            {
                {2, 7, 6, 3, 1, 4, 9, 5, 8},
                {8, 5, 4, 9, 6, 2, 7, 1, 3},
                    {9, 1, 3, 8, 7, 5, 2, 6, 4},
                    {4, 6, 8, 1, 2, 7, 3, 9, 5},
                    {5, 9, 7, 4, 3, 8, 6, 2, 1},
                    {1, 3, 2, 5, 9, 6, 4, 8, 7},
                    {3, 2, 5, 7, 8, 9, 1, 4, 6},
                    {6, 4, 1, 2, 5, 3, 8, 7, 9},
                    {7, 8, 9, 6, 4, 1, 5, 3, 2}

                };
        int sudoku7[9][9]=
                {
                    {1, 2, 6, 4, 3, 7, 9, 5, 8},
                    {8, 9, 5, 6, 2, 1, 4, 7, 3},
                    {3, 7, 4, 9, 8, 5, 1, 2, 6},
                    {4, 5, 7, 1, 9, 3, 8, 6, 2},
                    {9, 8, 3, 2, 4, 6, 5, 1, 7},
                    {6, 1, 2, 5, 7, 8, 3, 9, 4},
                    {2, 6, 9, 3, 1, 4, 7, 8, 5},
                    {5, 4, 8, 7, 6, 9, 2, 3, 1},
                    {7, 3, 1, 8, 5, 2, 6, 4, 9}

                };
        int sudoku8[9][9]=
                {
                    {1, 7, 2, 5, 4, 9, 6, 8, 3},
                    {6, 4, 5, 8, 7, 3, 2, 1, 9},
                    {3, 8, 9, 2, 6, 1, 7, 4, 5},
                    {4, 9, 6, 3, 2, 7, 8, 5, 1},
                    {8, 1, 3, 4, 5, 6, 9, 7, 2},
                    {2, 5, 7, 1, 9, 8, 4, 3, 6},
                    {9, 6, 4, 7, 1, 5, 3, 2, 8},
                    {7, 3, 1, 6, 8, 2, 5, 9, 4},
                    {5, 2, 8, 9, 3, 4, 1, 6, 7}

                };

        int sudoku9[9][9]=
                {
                    {7, 2, 6, 4, 9, 3, 8, 1, 5},
                    {3, 1, 5, 7, 2, 8, 9, 4, 6},
                    {4, 8, 9, 6, 5, 1, 2, 3, 7},
                    {8, 5, 2, 1, 4, 7, 6, 9, 3},
                    {6, 7, 3, 9, 8, 5, 1, 2, 4},
                    {9, 4, 1, 3, 6, 2, 7, 5, 8},
                    {1, 9, 4, 8, 3, 6, 5, 7, 2},
                {5, 6, 7, 2, 1, 4, 3, 8, 9},
                {2, 3, 8, 5, 7, 9, 4, 6, 1}

            };
int sudoku10[9][9]=
            {
                {6, 5, 9, 3, 1, 4, 2, 8, 7},
                {1, 8, 7, 6, 5, 2, 4, 3, 9},
                {2, 3, 4, 8, 9, 7, 5, 1, 6},
                {4, 2, 6, 1, 3, 5, 9, 7, 8},
                {8, 7, 1, 9, 4, 6, 3, 5, 2},
                {5, 9, 3, 2, 7, 8, 6, 4, 1},
                {3, 1, 2, 5, 8, 9, 7, 6, 4},
                {7, 6, 5, 4, 2, 1, 8, 9, 3},
                {9, 4, 8, 7, 6, 3, 1, 2, 5}

            };



int main(void)
{
    //starting time for single thread execution
    steady_clock::time_point start_time_single_thread = steady_clock::now();
    if(verifySudokuBySingleThread())
    {
        cout<<endl;
        cout<<endl;
        std::cout<<"1)sudoku board is valid"<<std::endl;
    }
    else
    {
        cout<<endl;
        cout<<endl;
        std::cout<<"1)Sudoku board is not valid"<<std::endl;
    }

    // calculating the elapsed time in microseconds.
    steady_clock::time_point end_time_single_thread = steady_clock::now();
    long elapsed_time_single_thread = duration_cast<microseconds>(end_time_sin
gle_thread - start_time_single_thread).count();
    cout<<endl;
    cout << endl << "2)Total time for evaluating a single sudoku solution usin
g single thread: " << elapsed_time_single_thread << " Micro seconds" <<endl;
```

```
    cout<<"===================================================================
==============================================="<<endl;
    cout<<endl;
    cout<<endl;




    // using 27  threads to solve a single sudoku board

    pthread_t threads[number_of_threads];

    //start time for validating the solution with 27 threads

    steady_clock::time_point starttime_threadmethod = steady_clock::now();
    int thread_index = 0;

    // create 27 threads 9 for 3*3 grids ,9 for columns ,9 for rows
    //pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start
_routine)(void *), void * arg);

    for (int k=0;k<9;k++)
    {   //3*3 grid validation
        for(int l=0;l<9;l++)
        {
            if(k%3==0 && l%3==0)
            {
                arguments *sud_grid=new arguments;
                sud_grid->row=k;
                sud_grid->column=l;
                sud_grid->sud_ptr=sudoku;
                pthread_create(&threads[thread_index++], NULL, verify_grid, su
d_grid);
            }
            if(l==0)
            {   // row validation
                arguments *sud_row=new arguments;
                sud_row->row=k;
                sud_row->column=l;
                sud_row->sud_ptr=sudoku;
                pthread_create(&threads[thread_index++], NULL, verify_row, sud
_row);
            }
            if(k==0)
            {   //column validation
```

```
                arguments *sud_col=new arguments;
                sud_col->row=k;
                sud_col->column=l;
                sud_col->sud_ptr=sudoku;
                pthread_create(&threads[thread_index++], NULL, verify_col, sud
_col);
            }
        }
    }

    //waiting for all threads to finish execution
    for(int i=0;i<27;i++)
    {
        pthread_join(threads[i], NULL);
    }

    // if any  elements of  result_final contains 0 then the solution is inval
id
    int cond=0;
    for(int i=0;i<27;i++)
    {
        if(result_final[i]==0)
        {
            cout<<"3) Sudoku board is not valid"<<endl;
            steady_clock::time_point endtime_threadmethod = steady_clock::now(
);
            //calculating time spent in microseconds
            long total_time = duration_cast<microseconds>(endtime_threadmethod
 - starttime_threadmethod).count();
            cout<<endl;
            cout<<endl;
            cout <<"4) Total time to evaluate a single sudoku board using 27 t
hreads: " << total_time << "Micro seconds" << endl;
            cout<<"===========================================================
===================="<<endl;
            cond=1;
            break;

        }
    }
    if(cond==0)
    {
        cout<<"3) Sudoku board is valid"<<endl;
        steady_clock::time_point endtime_threadmethod = steady_clock::now();
        //calculating time spent in microseconds
        long total_time = duration_cast<microseconds>(endtime_threadmethod - s
tarttime_threadmethod).count();
        cout<<endl;
        cout<<endl;
        cout << "4) Total time to evaluate a single sudoku board using 27 thre
ads: " << total_time << "Micro seconds" << endl;
        cout<<"===============================================================
=================="<<endl;
    }


    cout<<endl;
    cout<<endl;
    cout<<"5) ";

    //multiple boards or validating multiple sudoku solutions

    //creating a vector to store multiple sudoku boards
    // creating objects of sud_containers to store in the vector so that we ar
e able to iterate and access each board one after other

    vector<sud_container> v_sud;
    sud_container *s1=new sud_container(sudoku1);
    v_sud.push_back(*s1);
    sud_container *s2=new sud_container(sudoku2);
    v_sud.push_back(*s2);
    sud_container *s3=new sud_container(sudoku3);
    v_sud.push_back(*s3);
    sud_container *s4=new sud_container(sudoku4);
    v_sud.push_back(*s4);
    sud_container *s5=new sud_container(sudoku5);
    v_sud.push_back(*s5);
    sud_container *s6=new sud_container(sudoku6);
    v_sud.push_back(*s6);
    sud_container *s7=new sud_container(sudoku7);
    v_sud.push_back(*s7);
    sud_container *s8=new sud_container(sudoku8);
    v_sud.push_back(*s8);
    sud_container *s9=new sud_container(sudoku9);
    v_sud.push_back(*s9);
    sud_container *s10=new sud_container(sudoku10);
    v_sud.push_back(*s10);
```

```
   //multiple sudoku execution single thread
   // starting time single time multiple sudoku validations
   steady_clock::time_point start_time_single_thread_multiple = steady_clock::
now();

   for(int i=0;i<v_sud.size();i++)
   {
    bool result=verifySudokuBySingleThread_multiple(v_sud[i].arr);
     if(result)
    {
        std::cout<<"Sudoku "<<(i+1)<<" is valid | ";
    }
    else
    {
        std::cout<<"Sudoku "<<(i+1)<<" is not valid | ";
    }

   }
   // calculate total time spent to  validate 10 solutions in a single thread

   steady_clock::time_point end_time_single_thread_multiple = steady_clock::no
w();
   long elapsed_time_single_thread_multiple = duration_cast<microseconds>(end_
time_single_thread_multiple - start_time_single_thread_multiple).count();
   cout<<endl;
   cout<<endl;
   cout<< "6) Total time taken to evaluate 10 sudoku boards using single threa
d: " << elapsed_time_single_thread_multiple << " Micro seconds" << endl;
   cout<<"============================================================
==============="<<endl;
   cout<<endl;
   cout<<endl;

   //validate 10 sudoku solutions using 10 threads

   int thread_index_multiple=0;
   int number_of_thread_multiple=10;
   vector<bool> res(10,false);

   pthread_t threads_multiple[number_of_thread_multiple];

   //start clock 10 sudoku solutions using 10 threads
   steady_clock::time_point starttime_threadmethod_multiple = steady_clock::no
w();

   for(int i=0;i<10;i++)
   {
    multiple_thread_obj *mbo=new multiple_thread_obj(v_sud[i].arr,i);
    pthread_create(&threads_multiple[thread_index_multiple++], NULL, verify_mu
ltiple_multithread,mbo);
   }

    // wait for all 10 threads to complete
    for(int i=0;i<10;i++)
    {
        pthread_join(threads_multiple[i], NULL);
    }

    // calculate time taken to  validate 10 sudoku solutions using 10 threads

    steady_clock::time_point endtime_threadmethod_multiple = steady_clock::now
();
    long total_time_multiple = duration_cast<microseconds>(endtime_threadmetho
d_multiple - starttime_threadmethod_multiple).count();
    cout<<endl;
    cout<< "7) Total time taken to solve 10 sudoku boards using 10 threads  :
" << total_time_multiple << " Micro seconds" << endl;
    cout<<"==============================================================
=========================="<<endl;
    cout<<endl;
    cout<<endl;

    // iterate res_mul vector if any element is 1 the sudoku corresponding to
the index of that element is invalid
    for(int i=0;i<res_mul.size();i++)
    {
        if(res_mul[i]==0)
        {
           cout<<"sudoku "<<(i+1)<<" is valid | ";
        }
        else if(res_mul[i]==1)
        {
            cout<<"sudoku "<<(i+1)<<" is not valid | ";
        }
    }
```

```
}

// function definitions
/*used for single thread solution validation
 * verifies if each grid  contains all the digits 1-9.
 * A Boolean vector numbers is initialized to false.
 * For every value in the grid, the corresponding index's value in numbers is
set to true.
 * If a value in numbers vector is false then that value is either absent or p
resent twice.
 * This false value indicates that the given solution is not valid.
 * If all the values are true then solution is valid
 *
 * @return  bool result of grid validation
 */
bool verifyGrid(){
    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
            int grid_row = 3 * i;
            int grid_column = 3 * j;
            vector<bool> numbers(rows, false);
            for(int r=grid_row; r < grid_row+3; r++){
                for(int c=grid_column; c < grid_column+3; c++){
                    numbers[sudoku[r][c]-1] = !numbers[sudoku[r][c]-1];
                }
            }
            for(auto i : numbers){
                if(!i){
                    return false;
                }
            }
        }
    }
    return true;
}

/*used for single thread solution validation
 * verifies if each column  contains all the digits 1-9.
 * A Boolean vector numbers is initialized to false.
 * For every value in the column, the corresponding index's value in numbers i
s set to true.
 * If a value in numbers vector is false then that value is either absent or p
resent twice.
```

```
 * This false value indicates that the given solution is not valid.
 * If all the values are true then solution is valid
 *
 * @return  bool result of column validation
 */
bool verifyColumn(){
    for(int i=0; i<rows; i++){
        vector<bool> numbers(rows, false);
        for(int j=0; j<columns; j++){
            numbers[sudoku[j][i]-1] = !numbers[sudoku[j][i]-1];
        }
        for(auto i : numbers){
            if(!i){
                return false;
            }
        }
    }
    return true;
};

/*used for single thread solution validation
 * verifies if each row  contains all the digits 1-9.
 * A Boolean vector numbers is initialized to false.
 * For every value in the row, the corresponding index's value in numbers is s
et to true.
 * If a value in numbers vector is false then that value is either absent or p
resent twice.
 * This false value indicates that the given solution is not valid.
 * If all the values are true then solution is valid
 *
 * @return  bool result of row validation
 */
bool verifyRow(){
    for(int i=0; i<rows; i++){
        vector<bool> numbers(rows, false);
        for(int j=0; j<columns; j++){
                numbers[sudoku[i][j]-1] = !numbers[sudoku[i][j]-1];
        }
        for(auto i : numbers){
            if(!i){
                return false;
            }
        }
    }
    return true;
```

```
}

/*For single thread solution validation
 * verifies the solution based on row, column and grid
 *
 * @return  bool result of solution validation
 */
bool verifySudokuBySingleThread(){
    if (!verifyRow()){
        return false;
    }
    if (!verifyColumn()){
        return false;
    }
    if (!verifyGrid()) {
        return false;
    }
    return true;
}




/* used for multi thread single solution validation
 * verifies if each grid  contains all the digits 1-9.
 * A Boolean vector numbers is initialized to false.
 * For every value in the grid, the corresponding index's value in numbers is
set to true.
 * If a value in index corresponding to the current value is already true then
 that value is present twice.
 * This indicates that the given solution is not valid.
 * If all the values are true then solution is valid
 * @param   void * parameters (pointer).
 *
 */

void *verify_grid(void *parameter)
{
    arguments *sud=(arguments*) parameter;
    int row=sud->row;
    int col=sud->column;
    vector<bool> numbers(10,false);
    for (int i = row; i < row + 3; ++i)
    {
        for(int j=col;j<col+3;++j)
        {
            int ind=sud->sud_ptr[i][j];
            if (numbers[ind]==true)
            {
            pthread_exit(NULL);
            }
            else
            {
            numbers[ind]=!numbers[ind];
            }
        }
    }

    // If the execution has reached this point, then the 3x3 sub-
grid is valid.
    result_final[row + col/3] = 1; // Maps the 3X3 sub-
grid to an index in the first 9 indices of the result array
    pthread_exit(NULL);

}



/* used for multi thread single solution validation
 * verifies if each row  contains all the digits 1-9.
 * A Boolean vector numbers is initialized to false.
 * For every value in the grid, the corresponding index's value in numbers is
set to true.
 * If a value in index corresponding to the current value is already true then
 that value is present twice.
 * This indicates that the given solution is not valid.
 * If all the values are true then solution is valid
 * @param   void * params (pointer).
 *
 */


void *verify_row(void *params)
{
    arguments *sud=(arguments*) params;
    int row=sud->row;
    vector<bool> numbers(10,false);
    for (int j=0;j<9;j++)
    {
        int ind=sud->sud_ptr[row][j];
        if (numbers[ind]==true)
        {
            pthread_exit(NULL);
        }
        else
        {
            numbers[ind]=!numbers[ind];
        }
    }

    result_final[9 + row] = 1;
    pthread_exit(NULL);
}

/* used for multi thread single solution validation
 * verifies if each column  contains all the digits 1-9.
 * A Boolean vector numbers is initialized to false.
 * For every value in the grid, the corresponding index's value in numbers is
set to true.
 * If a value in number corresponding to the current value is already true the
n that value is present twice.
 * This indicates that the given solution is not valid.
 * If all the values are true then solution is valid
 * @param   void * params (pointer).
 *
 */

void *verify_col(void *params)
{
    arguments *sud=(arguments*) params;
    int col=sud->column;
    vector<bool> numbers(10,false);
    for (int i = 0; i < 9; i++)
    {
        int ind=sud->sud_ptr[i][col];
        if (numbers[ind]==true)
        {
            pthread_exit(NULL);
        }
        else
        {
            numbers[ind]=!numbers[ind];
        }
    }
    // If the execution has reached this point, then the column is valid.
    result_final[18 + col] = 1; // Maps the column to an index in the third se
t of 9 indices of the result array
    pthread_exit(NULL);

}


//functions to solve multiple sudoku solutions

/*
 * copies 2d array source and destination both passed as parameters
 * nested for loops used for deep copy of each element
 * @param  int 2d array destination, int 2d array of sudoku board source  .
 */

void copy2darrays(int params[9][9],int params1[9][9])
{
    for(int i=0;i<9;i++)
    {
        for(int j=0;j<9;j++)
        {
            params[i][j]=params1[i][j];
        }
    }
}


/* Used for both single and multi thread multi solution validation
 * Checks if a grid of size 3x3 contains all numbers from 1-9.
 * A Boolean vector numbers is initialized to false.
 * For every value in the grid, the corresponding index in numbers is set to t
rue.
 * If the value in numbers is false implies that the value is repeating or the
 value is not preesent.
 * This indicates that the grid is invalid
 * @param   int  2d array of the sudoku solution
 * @return  bool result of grid validation
 */

bool verifyGrid_multiple(int sudd[][9]){
    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
            int grid_row = 3 * i;
            int grid_column = 3 * j;
            vector<bool> numbers(rows, false);
            for(int r=grid_row; r < grid_row+3; r++){
                for(int c=grid_column; c < grid_column+3; c++){
                    numbers[sudd[r][c]-1] = !numbers[sudd[r][c]-1];
                }
            }
            for(auto i : numbers){
                if(!i){
                    return false;
                }
            }
        }
    }
    return true;
}




/* Used for both single and multi thread multi solution validation
 * Checks if each column contains all digits from 1-9.
 * A Boolean vector numbers is initialized to false.
 * For every value in the column, the corresponding index in numbers is set to
true.
 * If the value in numbers is false implies that the value is repeating or the
value is not preesent.
 * This indicates that the column is invalid
 * @param   int  2d array of the sudoku solution
 * @return  bool result of grid validation
 */


bool verifyColumn_multiple(int sudd[][9]){
    for(int i=0; i<9; i++){
        vector<bool> numbers(rows, false);
        for(int j=0; j<9; j++){
            numbers[sudd[j][i]-1] = !numbers[sudd[j][i]-1];
        }
        for(auto i : numbers){
            if(!i){
                return false;
            }
        }
    }
    return true;
};


/* used for both single and multi thread  multiple solution validation
 * Checks if each row  contains all the digits 1-9.
 * A Boolean vector numbers is initialized to false.
 * For every value in the row, the corresponding index in numbers is set to tr
ue.
 * If the value in numbers is false implies that the value is repeating or the
 value is not preesent.
 * This means the row is  not valid
 * @param   int  2d array of the sudoku solution
 * @return  bool result of row validation
 */


bool verifyRow_multiple(int sudd[][9]){
    for(int i=0; i<rows; i++){
        vector<bool> numbers(rows, false);
        for(int j=0; j<columns; j++){
                numbers[sudd[i][j]-1] = !numbers[sudd[i][j]-1];
        }
        for(auto i : numbers){
            if(!i){
                return false;
            }
        }
    }
    return true;
}

/* For single thread multiple solution validation
 * verifies the solution based on row, column and grid
 *
 * @return  bool result of solution validation
 */


bool verifySudokuBySingleThread_multiple(int sudd[][9]){
    if (!verifyRow_multiple(sudd)){
        return false;
    }
    if (!verifyColumn_multiple(sudd)){
        return false;
    }
    if (!verifyGrid_multiple(sudd)) {
        return false;
    }

    return true;
}

/* For multiple thread multiple solution validation
 * verifies the solution based on row, column and grid
 * sets the global vector res_mul based on the  result of solution validation
 */

void *verify_multiple_multithread(void * parameter)
{

    multiple_thread_obj *mto = (multiple_thread_obj*) parameter;
    int k= mto->b;

    if (!verifyRow_multiple(mto->a)){
        res_mul[k]=1;
    }
    if (!verifyColumn_multiple(mto->a)){

        res_mul[k]=1;
    }
    if (!verifyGrid_multiple(mto->a)) {
          res_mul[k]=1;

    }


    pthread_exit(NULL);
}
```