**OPERATING SYSTEM PROJECT**

**SUDOKU SOLUTION VALIDATOR WITH**

**MULTITHREADING**



INTRODUCTION

Sudoku puzzles are popular among people all over the world. Today, the game appears in almost every newspaper, magazine and in many websites. Sudoku is a logic-based number-placement puzzle game. The standard Sudoku puzzle is a table made up of 9 rows, 9 columns and 9, 3x3 boxes. 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 the box contains every number from 1 to 9 without any repetition. Below is a sample sudoku problem.

A close up of a device

Description automatically generated

The aim of this project is to develop an application in C/C++ language to validate the Sudoku solution with the help of multi-threading. Multithreaded applications are the ones which uses concept of Concurrency i.e. they are capable of processing more than one tasks in parallel.

BACKGROUND

Multithreading is a specialized form of multitasking and multitasking is the feature that allows computer to run two or more programs concurrently. Multitasking can be achieved by data parallelism and task parallelism. Data parallelism focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. So different threads will be executing the same task on different core with different subset of data. Task parallelism involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

A thread library provides the programmer with an API for creating and managing threads. There are three main thread libraries which are in use today: POSIX Pthreads, Windows, and Java. For this project, we are using POSIX thread (Pthreads) to achieve multithreading.

The POSIX thread (pthread) libraries are a standard based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment.

pthread is a specification for thread behavior, not an implementation. Some commonly used methods of pthread library are:

1. pthread\_attr\_init() - This function initializes the thread attributes object.
2. pthread\_create() -This function starts a new thread in the calling process.
3. pthread\_join() - This function waits for the thread specified by thread to

terminate. If that thread has already terminated, then

pthread\_join () returns immediately.

1. Pthread\_exit() – This function terminates the calling thread and returns a value

OBJECTIVE AND DELIVERABLES

The primary objective of this project is to design the implement multithreading application that validates the sudoku solution using POSIX thread. We are also comparing the performance of multithreaded application using single threaded sudoku solution validator.

APPROACH

There are several methods to implement the multithreaded application. As part of this project are considering three thread to divide the work and make the process faster.

Below are three threads that we are implemented.

1. One thread for validating all the columns.
2. One thread for validating all the rows.
3. We logically divide 9x9 grid to nine 3\*3 sub-grid and implement one thread for each of such sub-grids, so in total we will have nine threads to cover all the 3\*3 sub-grids.

Conditions to be validated by individual threads are as below:

1. The thread validating the columns should iterate over all the columns from one to nine and check for the below validations.
   1. Column should have all the numbers from 1- 9.
   2. There should be no empty cell.
   3. There should not be any duplicate value in the column.
   4. All the numbers should lie between 1-9.
2. The thread validating the rows should iterate over all the rows from one to nine and check for the below validations.
   1. Row should have all the numbers from 1- 9.
   2. There should be no empty cell.
   3. There should not be any duplicate value in the row.
   4. All the numbers should lie between 1-9.
3. The threads validating the sub-grid should check each cell of the sub-grid and check for the below validations.
   1. Sub-grid should have all the numbers from 1- 9.
   2. There should be no empty cell.
   3. There should not be any duplicate value in the sub-grid.
   4. All the numbers should lie between 1-9.

METHODOLOGY

* Main Process creates 3 worker threads one for validation of columns in Sudoku, second for validation of rows in Sudoku, third for validation of 9 3\*3 sub-grid in Sudoku.
* The program waits for each thread to complete their process and then program checks if all validation conditions are satisfied.
* If all the validation conditions are satisfied, then program declares the sudoku as valid.
* If any of the child thread finds invalid column or row or sub matrix, it will stop validating further. After completion of all the threads, the program prints the invalid row, column or grid and declares Sudoku Solution invalid.
* We tested this multithreaded Sudoku Solution Validator on multiple Sudoku Solution sets.
* We compared performance of multithreaded application with single threaded application for the Sudoku Solution Validator.

FINDINGS, ANALYSIS, AND DISCUSSION

We have successfully implemented multithreaded application in C++ language for validating sudoku solution. We have implemented the application using two approaches. Initially we implemented using single threaded application and timed the process. Then we implemented multithreaded application solution using POSIX thread and timed it. As part of multithreading approach, we create 9 threads to validate the sudoku solution. On comparing time for both the approaches, we found that in our scenario single threaded application is faster as compared to the multithreaded application. If a process execution time is less than or equal to thread creation time, then creating a thread will be considerable overhead to that program. This project implementation is example of such a scenario. One negative impact of multithreading is increased code complexity. Because of it, it’s hard to write code and debug it. Even testing it also becomes difficult. Also, multithreaded program achieves better performance in a multiprocessor or multicore platforms.

Validating Sudoku Solutions:

**Case 1:** Evaluating valid Sudoku Solution.

A screenshot of a social media post

Description automatically generated

**Case 2:** Evaluating invalid Sudoku Solution.

A screenshot of a cell phone

Description automatically generated

CONCLUSION

We successfully implemented the multithreaded application. For simple, short programs like 9 \* 9 Sudoku, single thread faster than multi thread, as process execution time is less than to thread creation time in multithreaded application. Multi-threading will definitely have advantages and performance improvement over Single thread for complex programs.

REFERENCE

1. Abraham-Silberschatz-Operating-System-Concepts (Ninth edition)
2. <https://en.wikipedia.org/wiki/Sudoku>
3. <https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>

APPENDIX

Please find below the source code for Sudoku solution validator.

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <unistd.h>

**#include** <pthread.h>

**#include** <iostream>

**#include** <chrono>

**using** **namespace** std;

**using** **namespace** std::chrono;

**#define** num\_threads 11

**int** valid[num\_threads] = {0};

**int** valid\_single\_thread[num\_threads] = {0};

**typedef** **struct** {

**int** row;

**int** column;

} parameters;

//Valid sudoku

**int** arr[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}};

//All Errors

//int arr[9][9] = {

// {6, 2, 4, 5, 3, 9, 1, 8, 7},

// {5, 1, 6, 7, 2, 8, 6, 3, 4},

// {8, 3, 7, 6, 1, 4, 2, 9, 5},

// {1, 4, 3, 8, 9, 5, 7, 2, 6},

// {9, 5, 8, 2, 4, 7, 3, 6, 1},

// {7, 9, 2, 3, 6, 1, 4, 5, 8},

// {3, 7, 1, 9, 5, 6, 8, 4, 2},

// {4, 6, 9, 1, 8, 2, 5, 7, 3},

// {2, 8, 5, 4, 7, 3, 6, 1, 9}

//};

//Validating each subgird

**void** \***isSubGridValid**(**void**\* param) {

parameters \*params = (parameters\*) param;

**int** row = params->row;

**int** col = params->column;

**int** validityArray[9] = {0};

**int** i, j;

**for** (i = row; i < row + 3; i++) {

**for** (j = col; j < col + 3; j++) {

**int** num = arr[i][j];

**if** ( validityArray[num - 1] == 1) {

**pthread\_exit**(NULL);

} **else** {

validityArray[num - 1] = 1;

}

}

}

valid[row + col/3] = 1;

**pthread\_exit**(NULL);

}

//Validating each row

**void** \***isRowValid**(**void**\* param) {

**int** validityArray[9] = {0};

**for** (**int** i = 0; i < 9; i++) {

**for**(**int** j =0; j<9; j++) {

**int** num = arr[i][j];

**if** ( validityArray[num - 1] == 1) {

**pthread\_exit**(NULL);;

} **else** {

validityArray[num - 1] = 1;

}

}

**for**(**int** i= 0 ; i <9; i++){

validityArray[i] = 0;

}

}

valid[9] = 1;

**pthread\_exit**(NULL);

}

//Validating each columns

**void** \***isColumnValid**(**void**\* param) {

**int** validityArray[9] = {0};

**int** i,j;

**for** (j = 0; j < 9; j++) {

**for** ( i = 0; i < 9; i++) {

**int** num = arr[i][j];

**if** (validityArray[num - 1] == 1) {

**pthread\_exit**(NULL);

} **else** {

validityArray[num - 1] = 1;

}

}

**for**(**int** i= 0 ; i <9; i++){

validityArray[i] = 0;

}

}

valid[10] = 1;

**pthread\_exit**(NULL);

}

//Validating all the rows for single threaded application

**int** **checkRow**(**int** row){

**int** validityArray[9] = {0};

**for** (**int** i = 0; i < 9; i++) {

**int** num = arr[row][i];

**if** ( validityArray[num - 1] == 1) {

cout<<"Error in element of row : " << row + 1 << " and column : " <<i + 1<<endl;

**return** 1;

} **else** {

validityArray[num - 1] = 1;

}

}

valid\_single\_thread[9] = 1;

**return** 0;

}

//Validating all the columns for single threaded application

**int** **checkColumn**(**int** col){

**int** validityArray[9] = {0};

**int** i;

**for** (i = 0; i < 9; i++) {

**int** num = arr[i][col];

**if** (validityArray[num - 1] == 1) {

cout<<"Error in element of column : " << col + 1 << " and row : " <<i + 1<<endl;

**return** 1;

} **else** {

validityArray[num - 1] = 1;

}

}

valid\_single\_thread[10] = 1;

**return** 0;

}

//Validating all the subgrids for single threaded application

**int** **checkSubsection**(**int** row, **int** col) {

**int** validityArray[9] = {0};

**int** i, j;

**for** (i = row; i < row + 3; i++) {

**for** (j = col; j < col + 3; j++) {

**int** num = arr[i][j];

**if** ( validityArray[num - 1] == 1) {

cout<<"Error in subgrid, element of row : "<< i+1 << " and column : " <<j+1<<endl;

**return** 0;

} **else** {

validityArray[num - 1] = 1;

}

}

}

valid\_single\_thread[row + col/3] = 1;

**return** 0;

}

//Method to invoke single threaded process for evaluating Sudoku solution validator.

**bool** **SingleThreadValidator**(){

**int** i =0, j=0;

**for** (i = 0; i < 9; i++) {

**for** (j = 0; j < 9; j++) {

**if** (i%3 == 0 && j%3 == 0) {

checkSubsection(i, j);

}

**if** (i == 0) {

checkColumn(j);

}

**if** (j == 0) {

checkRow(i);

}

}

}

**for** (i = 0; i < num\_threads; i++) {

**if** (valid\_single\_thread[i] == 0) {

**return** **false**;

}

}

**return** **true**;

}

//Pritnting the input sudoku solution

**void** **printSuduko**(){

**for**(**int** i=0;i<9;i++)

{

**if**(i==0)

cout<<"\t\t-------------------"<<endl;

**for**(**int** j=0;j<9;j++){

**if**(j==0)

cout<<"\t\t|";

cout<<arr[i][j];

**if**(j!=0 && (j+1)%3 == 0){

cout<<"|";

} **else**{

cout<<" ";

}

}

cout<<endl;

**if**(i!=0 && (i+1)%3 == 0){

cout<<"\t\t-------------------"<<endl;

}

}

cout<<endl;

}

//Main Method

**int** **main**() {

pthread\_t threads[num\_threads];

cout<<"Input Sudoku : "<<endl;

printSuduko();

//Single Threaded Validator

steady\_clock::time\_point start\_time\_single\_thread = steady\_clock::**now**();

**int** result = SingleThreadValidator();

steady\_clock::time\_point end\_time\_single\_thread = steady\_clock::**now**();

duration<**double**> elapsed\_time\_single\_thread = duration\_cast <duration <**double**> >(end\_time\_single\_thread - start\_time\_single\_thread);

cout<<"\nTime taken by using Single threads : " << elapsed\_time\_single\_thread.count() \*1000 <<" milliseconds"<<endl;

**if**(!result) {

cout<<"Sudoku solution is invalid! - checked using single threaded method\n";

} **else** {

cout<<"Sudoku solution is valid! - checked using single threaded method\n";

}

// Multi threaded Validator

**int** threadIndex = 0;

**int** i = 0,j = 0;

steady\_clock::time\_point start\_time = steady\_clock::**now**();

//Checking the Columns of 9\*9 sudoku

parameters \*columnData = (parameters \*) **malloc**(**sizeof**(parameters));

columnData->row = i;

columnData->column = j;

**pthread\_create**(&threads[threadIndex++], NULL, isColumnValid, columnData);

//Checking the Rows of 9\*9 sudoku

parameters \*rowData = (parameters \*) **malloc**(**sizeof**(parameters));

rowData->row = i;

rowData->column = j;

**pthread\_create**(&threads[threadIndex++], NULL, isRowValid, rowData);

//Checking the subgrids of 9\*9 sudoku

**for** (i = 0; i < 9; i++) {

**for** (j = 0; j < 9; j++) {

**if** (i%3 == 0 && j%3 == 0) {

parameters \*data = (parameters \*) **malloc**(**sizeof**(parameters));

data->row = i;

data->column = j;

**pthread\_create**(&threads[threadIndex++], NULL, isSubGridValid, data);

}

}

}

//Waiting for all the threads to complete

**for** (i = 0; i < num\_threads; i++) {

**pthread\_join**(threads[i], NULL);

}

steady\_clock::time\_point end\_time = steady\_clock::**now**();

duration<**double**> elapsed\_time = duration\_cast <duration <**double**> >(end\_time - start\_time);

cout<<"\nTime taken by using threads : " << elapsed\_time.count() \*1000 <<" milliseconds"<<endl;

//Checking for error

**bool** errFlag = **false**;

**for** (i = 0; i < num\_threads; i++) {

**if** (valid[i] == 0) {

errFlag = **true**;

}

}

**if**(!errFlag)

cout<<"Sudoku solution is valid! - checked using multi threaded method\n";

**else**{

cout<<"Sudoku solution is invalid! - checked using multi threaded method\n";

}

**return** EXIT\_SUCCESS;

}