SJSU SAN JOSÉ STATE UNIVERSITY

Sudoku Solution Validator

CMPE 180C Operating System Design
Spring 2020

Submitted by:
Team 12

Akhil Cherukuri	014525420	akhil.cherukuri@sjsu.edu
Dishant Shah	014615614	dishant.shah@sjsu.edu

Course Instructor
Dr. Hungwen Li

Table of Contents

Executive Summary

Sudoku is a puzzle game consisting of a 9x9 grid filled with several numbers randomly. The goal of the puzzle is to have a filled array where each of the 9 rows and 9 columns, as well as the 9 (3x3) squares, have 1-9 in them without repeating in a single row, column or square. Finding a valid solution can be difficult as there could be multiple solutions to the same puzzle and validating a non-official solution could be challenging. Currently, the common method of validating the sudoku puzzle involves looking at the solution provided in the book and comparing each square of the solution with each square of the filled puzzle manually. Usually, only one solution is provided. The benefit of this method is that it is the most convenient and readily available. However, its flaw is that it wouldn't account for alternate solutions which would require a person to manually traverse the filled puzzle and validate the three criteria for a valid solution.

The solution to this problem that we would like to provide would be the development of a program to help validate the solution. The filled puzzle will be input into the program which would segment the data by the rows, the columns and the 3x3 square which is analyzed by the threads which the program will create. 1 thread will be implemented to validate that each row has all the values between 1-9 without repeat. 1 thread will be implemented to validate that each column has all the values between 1-9 without repeat. A set of 9 threads will be implemented to validate that each of the 9 squares also fulfills the above condition. Each of the threads will return a Boolean value of true to the parent if the condition is valid and false if it is not. If all 11 threads return true, it would indicate that the solution is valid. The implementation of multithreading would allow for efficient utilization of the CPU resources and reduce time and user manual effort required for solution validation. The deliveries would be a program that would allow the user to input a filled sudoku puzzle, row by row, and have it output the validity of the solution.

Chapter 1: Introduction

Sudoku is a puzzle game consisting of a 9x9 square grid consisting of randomly filled numbers that are between 1-9 (1). The goal of the game is to fill the empty squares with a number from 1-9 such that each row and column has every number from 1-9 without repeating. Each of the 9 3x3 squares also needs to be filled with 1-9 as well. There can be multiple solutions to Sudoku puzzles especially the easier ones with fewer randomly filled squares. Normally, to check a solution, the puzzle is compared to the officially provided solution. If they do not match, manually checking the puzzling by hand can take time and effort. It would be more efficient to develop a program that can validate any solution by checking if the board fulfills the three requirements. Developing a program utilizing threads to simultaneously validate the solution of the problem.

Multi-threading is useful in any situation where a single thread has to wait for a resource, and you can run another thread in the meantime. This includes a thread waiting for an I/O request or database access while another thread continues with CPU work. In the sudoku solution validator, if we use only one thread to solve the problem (brute force method), it would take more time than it would if we divide the possible algorithm into several sub-algorithms which are as independent as possible and yet combine to give the solution. You are allowing the CPU to make several calculations simultaneously to solve the problem faster. This would allow for optimal utilization of resources thus reducing the overall time required.

In the sudoku solution validator, we will utilize threads and implement multi-threading to help solve and develop a tool for determining the validity of the solution efficiently. we will be using 1 thread for row calculations, 1 thread for column calculations, and 9 threads for each 3x3 grid calculations for a total of 11 threads which will be synchronized using the fork-join strategy. Each of the parent thread will return a Boolean value depending on whether the column, row or square is valid. The parent thread will wait until the validation of the parts is completed. It will determine if the overall solution is valid or not by determining that all Boolean values are true. In this project, we will implement multi-threading as well as the fork-join synchronization strategy to develop an efficient tool for sudoku solution validation.

Chapter 2: Background

Sudoku is a mathematical puzzle game that was first published in 1979 in Dell Magazines (2). There have been predecessors on other number-based grid games in the past, but this was the first modern implementation of the game. Sudoku has since grown in popularity with a dedicated fan base. Its popularity is so significant that a large number of newspapers feature a sudoku puzzle in their word/ number puzzles section. As previously stated, sudoku consists of a 9x9 array randomly filled with numbers in the 1-9 range which needs to be filled to complete the game. 3 conditions that need to be fulfilled for the solution to the given puzzle are valid.

- Row: Each row needs to contain every number from 1-9 without repeats.
- Column. Each column needs to contain every number from 1-9 without repeat.
- Square: there are 9 3x3 squares in the 9x9 grid of the puzzle. Each of them must contain every number from 1-9 without repeating as well.

Any filled sudoku puzzle that does not match the above requirements cannot be considered a valid solution. However, there can be multiple solutions to the puzzle, and it is possible to stumble across a solution to the puzzle that is not the official one provided in the puzzle book or the newspaper. During such circumstances, having a program that can determine the validity of the solution would be beneficial to save time and effort.

Thread is an execution unit that consists of its program counter, a stack, and a set of registers. Threads are also known as lightweight processes. Creating multiple threads is more economical than creating multiple processes as threads take fewer resources and as well as share resources such as code, data, and files between each thread under that process which is known as Inter-thread communication. Another advantage is that context switches between threads are faster than between processes. Threads are a popular way to improve the application through parallelism. The Central Processing Unit switches rapidly back and forth among the threads giving the illusion that the threads are running in parallel. This allows for threads to be an ideal way to access the computation resources for the needs of this project.

Threads are visible only from within the process, where they share all process resources like address space, open files, and so on. Because threads share data, a change on one thread can be visible by another thread in the operating system. Apart from data, each thread has its unique properties or private variables like Thread ID, Register state, Stack, Priority, Thread-Private Storage.

A system can be utilized either to use a single thread or multi-threads. In a multithreaded process on a single processor, the processor can switch execution resources between threads, resulting in concurrent execution. In a multithreaded process in a shared-memory multiprocessor environment, each thread in the process can run on a separate processor at the same time, resulting in parallel execution. An application can have hundreds of threads and still not consume many kernel resources. The amount of kernel resources the application uses is largely determined by the application requirements. Implementation of multi-threading will allow for algorithm for validation to be split into smaller subsections which can be run in a more efficient manner

The user threads must be mapped to kernel threads either by Many to One Model, One to One Model, or Many to Many Models. There are 3 types of threads i.e., POSIX Pitheads (Linux, UNIX), Win32 threads (windows), Java threads. There are two types of threads: User threads are threads that do not need kernel support. These threads are used inside an application program. Kernel threads are threads that are used inside an Operating System. These threads support the kernel to perform multiple tasks at the same time or to serve multiple kernel system calls simultaneously.

Multithreading has the possibility of resulting in improper manipulation of the shared data through unorganized access to it by the threads. To solve this, a variety of methods have been implemented. An example of the several possible solutions is Mutex, fork-join, Conditional variable, barrier, spinlock as well as a semaphore. The fork-join strategy involves the parent thread waiting until the child threads perform the work concurrently. After the child threads are finished, the parent thread can continue. In this project, none of the child threads should manipulate the shared data that the other will manipulate. The main thread would need to wait till the other threads finish validation of the parts which can be performed concurrently. Both of these facts make the fork-join strategy ideal for implementation in this project.

Chapter 2: Objectives

- To develop a program to validate the solution to the sudoku puzzle.
- To implement threading and subsequently multithreading using the fork-join strategy to the solution to optimize the timing and resource usage.

Chapter 3: Methodology

We have taken 2 approaches to solve the problem. All the code is written in C++ so that we can utilize the Pthreads library.

Method 1:

In the first approach we utilized a single thread and have not created any worker threads for the parent thread.

- This single thread itself will check all the rows and columns and also each 3 by 3 sub-grids.
- The elapsed time for the approach will also be displayed to understand the performance improvement/degradation.

Method 2:

In the second approach we have created a total of 11 threads to achieve the problems criteria.

- 1 thread to check the columns whether each column contains the digits 1 through 9.
- 1 thread to check the rows whether each row contains the digits 1 through 9.
- 9 threads to check that each of the 3 by 3 sub-grids contains the digits 1 through 9.
- The parent thread will create all the worker threads required, passing each worker the location that it must check in the Sudoku grid.
- 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 thread.
- The elapsed time for the approach will also be displayed to understand the performance improvement/degradation.

Chapter 3: Algorithm

The validation algorithm for the row/column/grid validation is stated below. This basic algorithm was used in both the threaded and non-threaded implementation with slight changes. :

```
boolint def checkc(int[] arr){
int [] x=[1,2,3,4,5,6,7,8,9];
int count=0;
for(int i=0;i<arr.size();i++){
      for(int y=0;y<x.size();y++){
            if(arr[i]==x[y]){
                  count++;
                  x[y]=0;
}}}
if (count=9){
return true1;}
else{
return false0;}}
```

The above algorithm was slightly modified to include a 2D array as well as the row or the column of the array to be checked. For the one theadless version, the following algorithm was used.

```
int[][] sud= {{}};
int rc;
int cc;
int colc;

for(int i=0;i<9;i++){
      rc =checkr(sud, i);
      cc=checkc(sud,i);
      if(rc==0 ||cc==0){
            cout<<"Inalid solution";
            break;}
}
int cn=0;
int rn=0;
for(int i=0;i<3;i++){
      cn=0;
      for (int e=0;e<3;e++){
                  rowc=checkcol(sud,rn,cn);
            cn=cn+3;
if(rowc==0){
cout<<"Invalid Solution";}}
```

```
 if(rc==1 && cc==1&&rowc==1){
        cout<<"The following is a valid solution to the sudoku
puzzle"<< endl;
    }
```

For the threaded version, due to the nature of the data sharing between the threads where the threads only view the data and there was no critical region, the wait and join method of synchronization was implemented. all the child threads were allowed to reach a logical point and return their result to the parent thread before the parent thread was allowed to proceed. The basic algorithm is stated below:

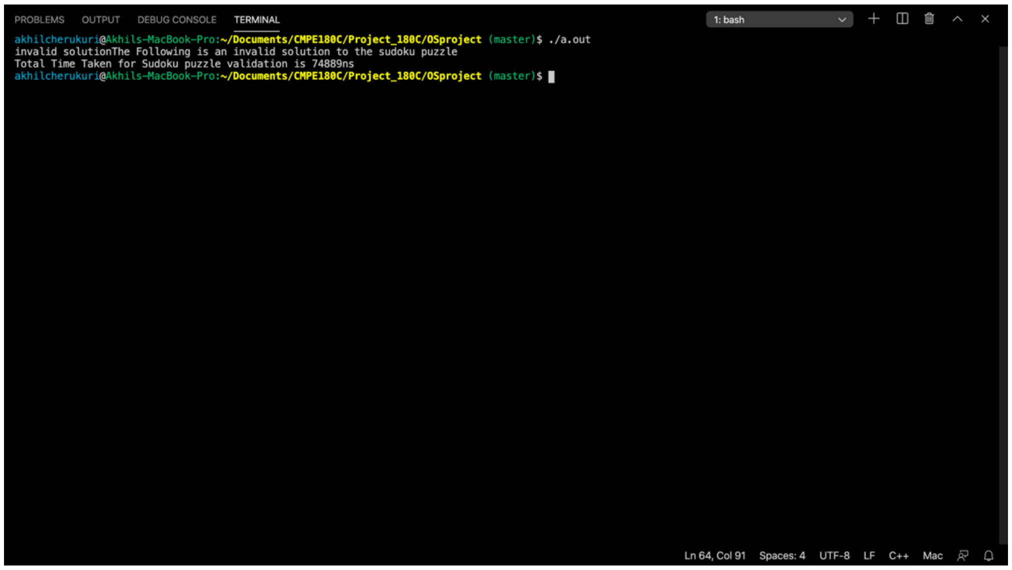
```
int[9][9] sud= {{}};
//variables for the result to be stored
int[9] res;
int colch;
int rowch;
pthread_t threads[11];
// thread for all row and all column
pthread_create( &threads[thread++],NULL,checkr,sudinfo);
pthread_create( &threads[thread++],NULL,checkc,sudinfo);
// thread for the 9 3x3 squares.
for (int m=0;m<3;m++){
     cn=0;
     for (int e=0;e<3;e++){
          pthread_create(&threads[t_index++],NULL, checkcol,
sudinfo2);
}}
// for loop that joins all the child thread back and stores the result
in the above variables.
for(int x=0, x<11,x++){
     pthread_join(threads[x],NULL);
}
int rescount=0;
// count the valid grids
for(int y=0;y<9;y++){
     if(res[y]==1){
     rescount++;
     }
}
// if all rows, columns and girds are valid
if(rescount==9 && colch==1 && rowch==1){
     cout<<”Valid result”;
}
else{
     cout<<”invalid result”;
}
```

Chapter 4: Test Results

Test Case 1:

Invalid Sudoku solution [Single thread]

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



```
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$ ./a.out
invalid solutionThe Following is an invalid solution to the sudoku puzzle
Total Time Taken for Sudoku puzzle validation is 74889ns
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$ 
```

Figure 1: Output of Test Case 1

Result:

Invalid solution.

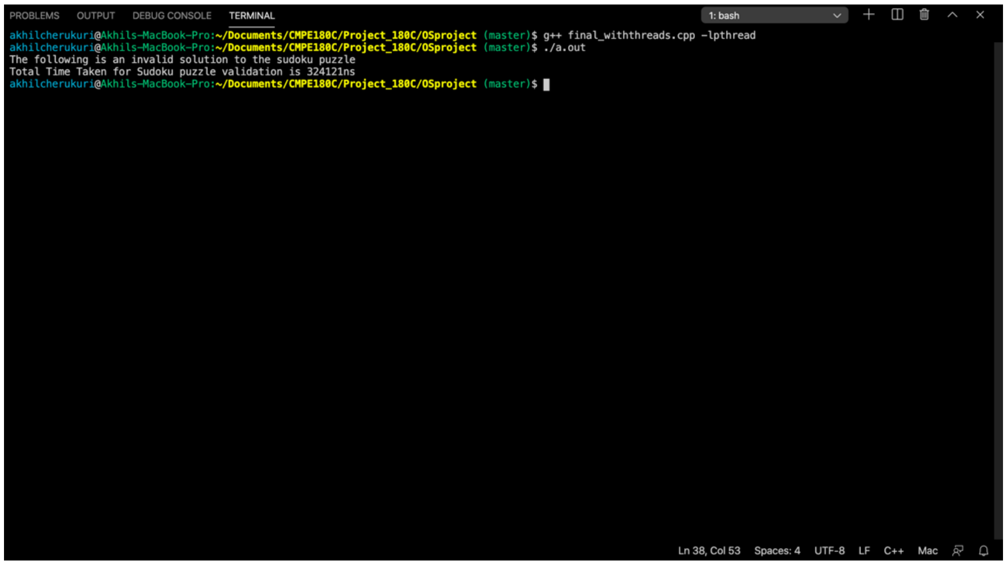
The Following is an invalid solution to the sudoku puzzle

Total Time Taken for Sudoku puzzle validation is 74889ns

<u>Test Case 2</u>:

Invalid Sudoku solution [11 threads]

{0, 4, 8, 3, 9, 5, 7, 1, 6},
{5, 7, 1, 6, 2, 8, 3, 4, 9},
{9, 3, 6, 7, 4, 1, 5, 8, 2},
{6, 8, 2, 5, 3, 9, 1, 7, 4},
{3, 5, 9, 1, 7, 4, 6, 2, 8},
{7, 1, 4, 8, 6, 2, 9, 5, 3},
{8, 6, 3, 4, 1, 7, 2, 9, 5},
{1, 9, 5, 2, 8, 6, 4, 3, 7},
{4, 2, 7, 9, 5, 3, 8, 6, 1}



```
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$ g++ final_withthreads.cpp -lpthread
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$ ./a.out
The following is an invalid solution to the sudoku puzzle
Total Time Taken for Sudoku puzzle validation is 324121ns
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$ 
```

Figure 2: Output of Test Case 2

<u>Result</u>:

Invalid solution.

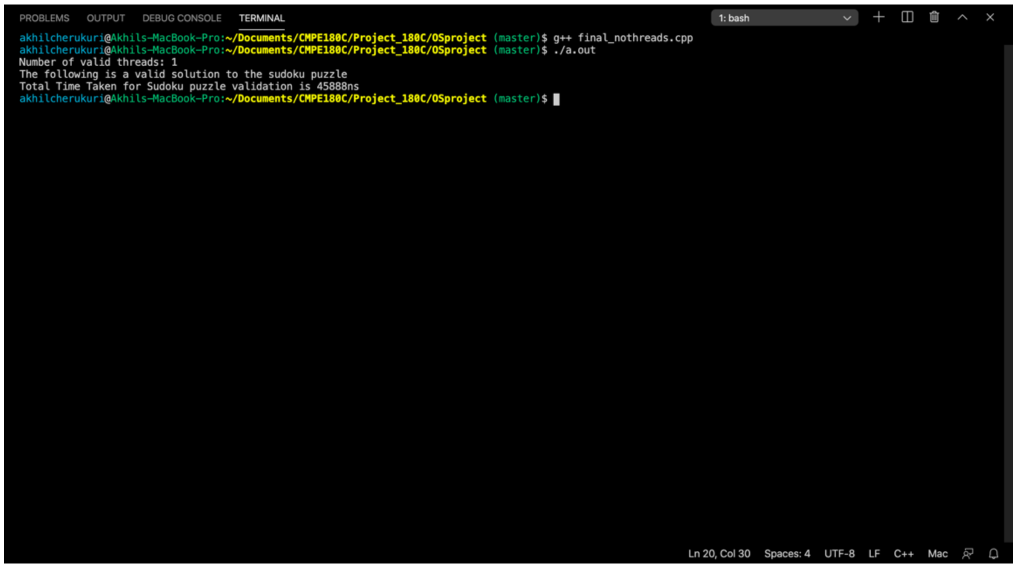
The Following is an invalid solution to the sudoku puzzle

Total Time Taken for Sudoku puzzle validation is 324121ns

Test Case 3:

Valid Sudoku Solution [Single thread]

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



```
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$ g++ final_nothreads.cpp
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$ ./a.out
Number of valid threads: 1
The following is a valid solution to the sudoku puzzle
Total Time Taken for Sudoku puzzle validation is 45888ns
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$ 
```

Figure 3: Output of Test Case 3

Result:

Number of valid threads: 1

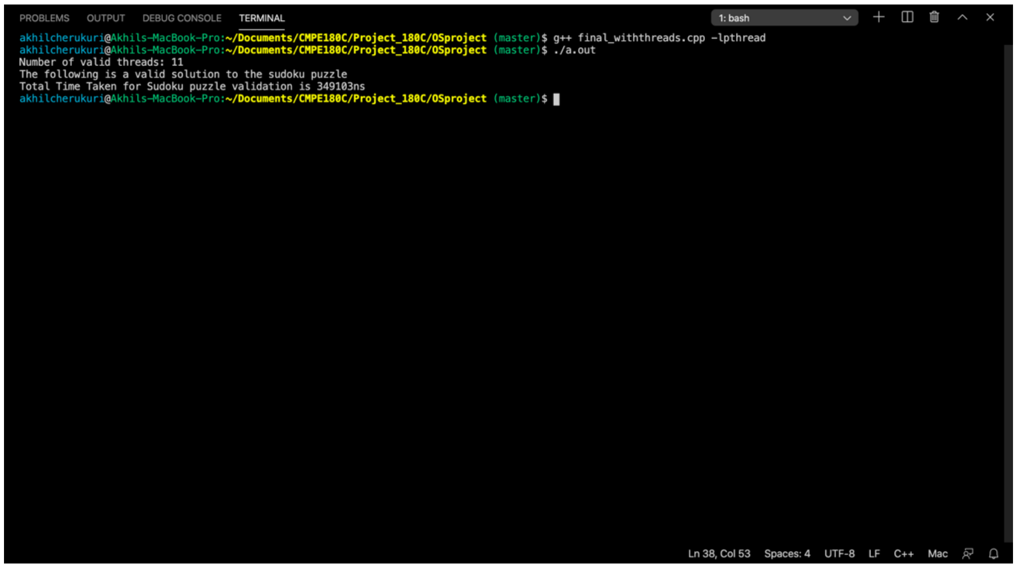
The following is a valid solution to the sudoku puzzle

Total Time Taken for Sudoku puzzle validation is 45888ns

Test Case 4:

Valid Sudoku Solution [11 threads]

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



```
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$ g++ final_withthreads.cpp -lpthread
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$ ./a.out
Number of valid threads: 11
The following is a valid solution to the sudoku puzzle
Total Time Taken for Sudoku puzzle validation is 349103ns
akhilcherukuri@Akhils-MacBook-Pro:~/Documents/CMPE180C/Project_180C/OSproject (master)$
```

Figure 4: Output of Test Case 4

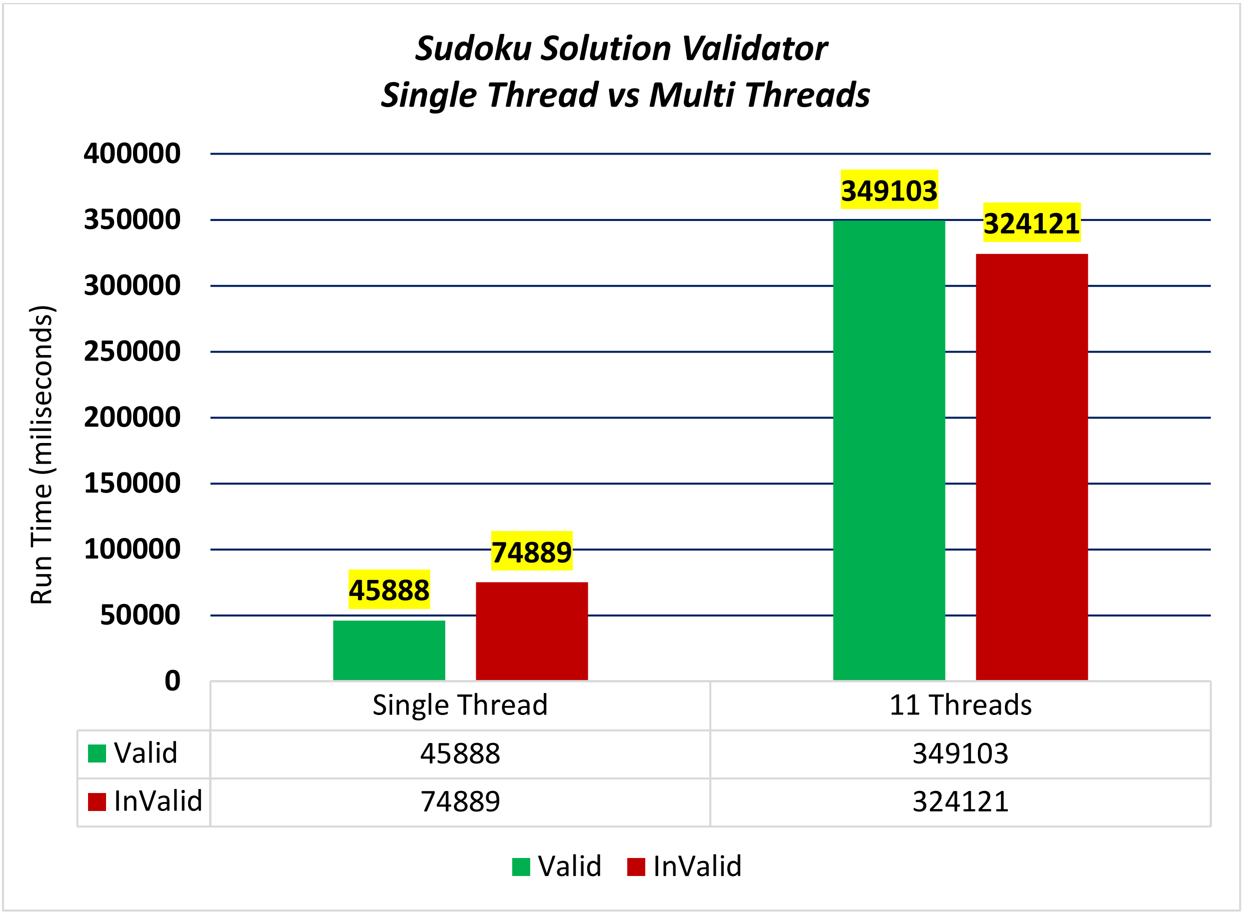
Result:

Number of valid threads: 11

The following is a valid solution to the sudoku puzzle

Total Time Taken for Sudoku puzzle validation is 349103ns

Chapter 4: Performance Analysis



	Single Thread	11 Threads
🟩 Valid	45888	349103
🟥 InValid	74889	324121

Figure 5: Comparison between the runtime of valid and invalid solution in single multithreaded programs.

The following figure shows that the single thread version is much more efficient in the run time required for validation in comparison to the multithreaded version. This is true for the validation of both valid and invalid solutions. Normally, threading is expected to decrease the run time by more efficiently utilizing the resources available. However, the overhead for thread switching still takes time. In this situation, the time required for thread creation and switching exceeds the runtime of the program without any threads. This makes multithreading relatively inefficient than just allowing a single thread to do all the work in the case of validating a single sudoku solution.

Chapter 5: Conclusion

- During our testing, we have observed that the multithread version was taking much more time to execute than the single thread version.

- For small simple programs such a Sudoku (9 by 9 Model), single thread execution is much faster than multithread. Multithreading requires thread creation and handling overhead which increases the time and the resources required.

- Creating a thread is a relatively expensive OS operation

- Context switching (where the CPU stops working on one thread and starts working on another) is again a relatively expensive operation

- Multithreading will definitely have advantages and performance improvement over single thread for complex code structures.

References

Sources:

- Silberschatz, A., Galvin, P. B., &amp; Gagne, G. (2019). Operating system concepts. Hoboken, NJ: Wiley.

- Block, S. S., &amp; Tavares, S. A. (2009). Before Sudoku: the world of magic squares. New York: Oxford University Press.

- For sudoku examples : https://sudoku.com/

Appendix: Code

Part 1: Single Thread

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <iomanip>
using namespace std;
using namespace std::chrono;
struct data
{
    /* data */
    int row;
    int col;
    int** s;

};

bool checkr(int ** m, int rn);
bool checkc(int ** m, int cn);
bool checkcol(int ** m, int rn,int cn);
 int main(){
     int array [9][9]={ {2, 4, 8, 3, 9, 5, 7, 1, 6},
                        {5, 7, 1, 6, 2, 8, 3, 4, 9},
                        {9, 3, 6, 7, 4, 1, 5, 8, 2},
                        {6, 8, 2, 5, 3, 9, 1, 7, 4},
                        {3, 5, 9, 1, 7, 4, 6, 2, 8},
                        {7, 1, 4, 8, 6, 2, 9, 5, 3},
                        {8, 6, 3, 4, 1, 7, 2, 9, 5},
                        {1, 9, 5, 2, 8, 6, 4, 3, 7},
                        {4, 2, 7, 9, 5, 3, 8, 6, 1}};

    int** arr = new int*[9];
    for(int p = 0; p<9;p++){
        arr[p]= new int[9];
        for (int u=0;u<9;u++){
            arr[p][u]=array[p][u];
        }
    }
    steady_clock::time_point start_time = steady_clock::now();
    // for loop to crete array and comulm data into arrays to be
tested.
    bool rc;
    bool cc;
    bool rowc;
    int rn=0;
    int cn=0;
     for (int i = 0; i < 9; i++)
     {

        rc =checkr(arr, i);
        cc=checkc(arr,i);

        if (rc==0 || cc==0){
            cout<<"invalid solution";
            break;
        }
     }
     for (int m=0;m<3;m++){
         cn=0;


         for (int e=0;e<3;e++){
             rowc=checkcol(arr,rn,cn);
             cn=cn+3;
             //cout<<rn<<"  "<<cn<<"  "<<endl;
             if(rowc==0){
                  cout<<"The Following is an invalid solution to the 
sudoku puzzle"<< endl;
                    break;
             }
         }
         rn=rn+3;}
     if(rc==1 && cc==1&&rowc==1){
         cout<<"Number of valid threads: 1"<<endl;
         cout<<"The following is a valid solution to the sudoku 
puzzle"<< endl;
     }
    steady_clock::time_point end_time = steady_clock::now();
    long time_taken=duration_cast<nanoseconds>(end_time - 
start_time).count();
    cout<<"Total Time Taken for Sudoku puzzle validation is 
"<<time_taken<<"ns"<<endl;
 }
 /** fucntion to check if input array has all the variables from 1-9
  * @params: int*m int array pointer to the array to be checked
  * @returns: bool true or false based if the check is valid or not.
  * */

  bool checkr(int** m,int rn){
      int* n = new int [9];
      int q=1;
      // Initialze an array to compare 1-9
```

```
        for (int p=0;p<9;p++){
            n[p]=q;
            q++;
        }
        int count=0;
        // for loop 1 loops through the input array m
        for (int i = 0; i < 9; i++)
        {
            // for loop 2 loops though the initialized array n
          for (int r = 0; r < 9; r++)
          {
              // check if they are equal and add to the count
              if (m[rn][i] ==n[r] &&n[r]!=0){
                n[r]=0;
                count=count+1;
            }
          }
        }
        // if count is 9 then the row contains all from 1-9 and return
  true or return false
        if (count==9){
            return true;
        }
        return false;
    }
     bool checkc(int** m,int cn){
        int* n = new int [9];
        int q=1;
        // Initialze an array to compare 1-9
        for (int p=0;p<9;p++){
            n[p]=q;
            q++;
        }
        int count=0;
        // for loop 1 loops through the input array m
        //cout<<"newrow"<<endl;
        for (int i = 0; i < 9; i++)
        {
            // for loop 2 loops though the initialized array n
          for (int r = 0; r < 9; r++)
          {
              //cout<<m[i][cn]<<"  "<<n[r]<<endl;
              // check if they are equal and add to the count
              if (m[i][cn] ==n[r] &&n[r]!=0){
                n[r]=0;
                count=count+1;
                //cout<<count<<endl;
            }
```

```
         }
       }
       // if count is 9 then the row contains all from 1-9 and return
true or return false
       if (count==9){
           return true;
       }
       return false;
   }
   bool checkcol(int** m,int rn,int cn){
         int* n = new int [9];
       int q=1;
       // Initialze an array to compare 1-9
       for (int p=0;p<9;p++){
           n[p]=q;
           q++;
       }
       int count=0;
       // for loop 1 loops through the input array m
       //cout<<"newrow"<<endl;
        for (int i = rn; i < rn+3; i++)
       {
           for (int o = cn; o < cn+3; o++)
       {
            for (int r = 0; r < 9; r++)
         {
               //cout<<m[i][o]<<"  "<<n[r]<<endl;
                if (m[i][o] ==n[r] &&n[r]!=0){
                 n[r]=0;
                 count=count+1;
                 //cout<<count<<endl;
                 }}}}
          if (count==9){
           return true;
       }
       return false;
   }
```

Part 2: 11 Threads

```
#include <iostream>
#include <fstream>
#include <pthread.h>
#include <iomanip>
#include <chrono>
```

```
#define no_thread 11;
using namespace std;
using namespace std::chrono;
int res [9];
int cvalid;
int rvalid;
struct sudokuinfo
{
    /* data */
    //column number to start for the 3x3
    int row;
    //row number to start for the 3x3
    int col;
    int** grids;

} ;
// one function to check all rows
void * checkr(void* m);
// one function to check all columns
void * checkc(void* m);
// one function to check each of the 3x3 grids
void *  checkcol(void* m);
 int main(){


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

    int** arr = new int*[9];
    for(int p = 0; p<9;p++){
        arr[p]= new int[9];
        for (int u=0;u<9;u++){
            arr[p][u]=puzzle[p][u];
        }
    }
    steady_clock::time_point start_time = steady_clock::now();
    sudokuinfo info;
    bool rowc;
    int rn=0;
    int cn=0;
    info.col=0;
    info.row=0;
    info.grids=arr;
    pthread_t threads[11];
    int t_index=0;
    // thread to check all rows
    pthread_create(&threads[t_index++],NULL, checkr, (void*)&info);
    // pthread to check all columns
    pthread_create(&threads[t_index++],NULL, checkc, (void*)&info);
    //for loop to check for all 3x3 square and create threads for them
    for (int m=0;m<3;m++){
         cn=0;


         for (int e=0;e<3;e++){


             sudokuinfo * info2=(sudokuinfo
*)malloc(sizeof(sudokuinfo));

             info2->col=cn;
             info2->row=rn;
             info2->grids=arr;
             pthread_create(&threads[t_index++],NULL, checkcol,
info2);

             cn=cn+3;

         }
         rn=rn+3;}
    int r_index=t_index;
    t_index=0;
    int* results=new int[r_index];


    // for loop to return the result of all the threads;
    for (int x=0;x<r_index;x++){


        pthread_join(threads[t_index++],NULL);


    }

    // checking that each of the 3x3 squares produces a valid result
    int rescountfin=0;
    for(int z=0;z<9;z++){
        if(res[z]==1){
            rescountfin++;
        }
    }
    // confirming that parts of the sudoku puzlle are valid and thus
confirming the validity of the solution.
    if(rescountfin==9 && rvalid==1 && cvalid==1){
        cout<<"Number of valid threads:
"<<rescountfin+rvalid+cvalid<<endl;
        cout<<"The following is a valid solution to the sudoku
puzzle"<<endl;

    }
    else{
        cout<<"The following is an invalid solution to the sudoku
puzzle "<<endl;
    }
    steady_clock::time_point end_time = steady_clock::now();
    long time_taken=duration_cast<nanoseconds>(end_time -
start_time).count();
    cout<<"Total Time Taken for Sudoku puzzle validation is
"<<time_taken<<"ns"<<endl;




 }
 /** fucntion to check if input array has all the variables from 1-9
in all of the rows
  * @params: void*m void pointer to the struct
  * @returns: bool true or false based if the check is valid or not.
  * */

  void* checkr(void* m){
       // cast the data into the struct
      struct sudokuinfo * sudata;
      sudata=(struct sudokuinfo *)m;

      int* n = new int [9];
      int q=1;
      // Initialze an array to compare 1-9
      for (int p=0;p<9;p++){
          n[p]=q;

          q++;
      }
      int count=0;
      // for loop that loops through the input array m
      for (int u = 0; u < 9; u++)
     {
      q=1;
      //reset the compare array
      for (int p=0;p<9;p++){
          n[p]=q;

          q++;
      }
      // for loop to loop the row
      for (int i = 0; i < 9; i++)
      {
          // for loop 2 loops though the initialized array n
        for (int r = 0; r < 9; r++)
        {
            // check if they are equal and add to the count
            if (sudata->grids[u][i] ==n[r] &&n[r]!=0){
              n[r]=0;
              count=count+1;

          }
        }
      }}
      // if count is 81 then all the row contains all from 1-9 and
return true or return false
     if (count==81){
          rvalid=1;
      }
      else{
          rvalid=0;
      }
      pthread_exit(NULL);
  }
 /** fucntion to check if input array has all the variables from 1-9
in all of the columns
  * @params: void*m void pointer to the struct
  * @returns: bool true or false based if the check is valid or not.
  * */
   void* checkc(void* m){
       // cast the data into the struct
       struct sudokuinfo * sudata;
      sudata=(struct sudokuinfo *)m;
      int* n = new int [9];

      int q=1;
      // Initialze an array to compare 1-9
      for (int p=0;p<9;p++){
          n[p]=q;

          q++;
      }
      int count=0;
      // for loop 1 loops through the input array m
      for (int u = 0; u < 9; u++)
     {
        q=1;
      //reset the compare array
      for (int p=0;p<9;p++){
          n[p]=q;

          q++;
      }
      // for loop to loop the column
      for (int i = 0; i < 9; i++)
      {
          // for loop 2 loops though the initialized array n
        for (int r = 0; r < 9; r++)
        {
            // check if they are equal and add to the count
            if (sudata->grids[i][u] ==n[r] &&n[r]!=0){
              n[r]=0;
              count=count+1;

          }
        }
      }}
      // if count is 81 then all the column contains all from 1-9 and
return true or return false

      if (count==81){
          cvalid=1;
      }
      else{
          cvalid=0;
      }
      pthread_exit(NULL);
  }
   /** fucntion to check if input array has all the variables from 1-9
in a 3x3 grid
  * @params: void*m void pointer to the struct
  * @returns: bool true or false based if the check is valid or not.
  * */
  void* checkcol(void* m){
      // take the pointer and cast the data into the struct
      struct sudokuinfo * sudata;
      sudata=(struct sudokuinfo *)m;
        int* n = new int [9];
      int q=1;
      // Initialze an array to compare 1-9
      for (int p=0;p<9;p++){
          n[p]=q;
          q++;
      }
      int count=0;
      // for loop 1 loops through the 3x3 based on the data in the
struct
       for (int i = sudata->row; i < sudata->row+3; i++)
      {
          for (int o = sudata->col; o < sudata->col+3; o++)
      {
           for (int r = 0; r < 9; r++)
        {
            // if the data matches the compare grid, then the compare
grid index is set to 0 and count is increased
             if (sudata->grids[i][o] ==n[r] &&n[r]!=0){
              n[r]=0;
              count=count+1;
              }}}}
        // if count is 9, all 1-9 values are present in the thread.
         if (count==9){
          res[sudata->row+sudata->col/3]=1;
      }
      else{
          res[sudata->row+sudata->col/3]=0;

      }
      pthread_exit(NULL);
  }
```