Sudoku Game Solver Using Parallel Computing

Harshit Mishra	Kartik Goel	Alokam Nikhitha
19BCE0799	19BCE2002	19BCE2555
harshit.mishra2019@vitstudent.ac.in	kartik.goel2019@vitstudent.ac.in	alokam.nikhitha2019@vitstudent.ac.in

Course Code: CSE4001

Course Title: Parallel and Distributed Computing

Prof In-Charge: Dr. Hiteshwar Kumar Azad
Associate Professor, SCOPE, VIT, Vellore

School Of Computer Science and Engineering

December, 2021

INDEX

1. INTRODUCTION

Sudoku is one of the most well-known puzzles of all time. This game aims at filling a grid with numbers from a given range such that every row, column and mini grid does not have any repeating numbers. The grid comes in varying sizes such as 4x4, 9x9, 16x16 etc. Computing the solution to a sudoku puzzle falls under the category of NP-complete and so the algorithms that have been proposed for solving it are computationally complex. Executing such an algorithm serially would be slow and inefficient. So, solving the sudoku puzzle has been directed towards parallel computing. Computing technology has experienced a drastic transition from serial computing to parallel and distributed computing. With multi-processing computers, distributed and faster networks and tech giants shifting towards parallelization, it has become more essential to delve into parallel programming to enhance computation. In light of this, our project aims to implement the sudoku solver using C and OpenMP serially and in parallel. We also aim to identify the limitations and advantages in both and draw a comparative study.

Keywords - Sudoku, OpenMP, C Programming, NP-Complete, Parallel programming.

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

Fig 1.1 Sudoku grid

2. LITERATURE SURVEY

S. No	Title	Author	Summary	Drawbacks
1.	Sudoku Game Solving Approach Through Parallel Processing	Saxena, R., Jain, M. and Yaqub, S.M.	The paper examines different answers for the issue alongside their advantages and disadvantages. The paper additionally examines the exhibition of a sequential variant arrangement as for execution time taken to explain Sudoku. With certain modifications and suspicions to the sequential form code, we have assessed the conceivable arrangement and bottleneck in the equal methodology. The outcomes have been assessed for the parallelized calculation on an ordinary client machine first, and afterward on a supercomputing arrangement box PARAM SHAVAK[1]	Because of combinatorial nature of the issue, it is intriguing to explore it with certain heuristics or options in contrast to the sequential methodology as the sequential code will significantly back off as the size of the Sudoku board increments or will most likely be unable to create the arrangement because of computational impediments of the machine.
2.	A search based sudoku solver	Cazenave, T. and Labo, I.A.	Two search algorithms Forward Checking (FC) and Limited Discrepancy Search (LDS) are evaluated. There is an evident phase transition for 25x25 sudoku puzzles. The min-domain-sum ordering of values is a better approach than the ordering of values lexicographically. LDS is a better alternative for FC. The quasi-group completion problem (QCP) is solving an nxn latin square by unassigning variables. It follows an easy-hard-easy phase transition depending on the number of unassigned values. Sudoku can be modelled as a QCP with additional constraints. Sudoku problem generation is done using LDS by unassigning randomly selected variables till a certain percentage of variables is reached. For a given size and percentage around 50 problemas are	LDS computes the solution for 1000 sudoku problems in less time compared to FC in both lexicographic and vdom+ value ordering heuristic. For FC the phase transition starts at 61% holes whereas it is almost unaffected in LDS. The limitation projected by this paper is the inability to generate puzzles with unique solutions. The time taken by LDS for a

			possible. For QCP redundant modelling uses n^2 constraints which may be computationally infeasible. In FC every time a variable is assigned, one from the set of free variables from the same row, column and mini grid is removed. LDS traverses leaves in the search tree in increasing order of discrepancy. A discrepancy occurs when the algorithm does not follow the heuristics. [8]	25x25 puzzle in lex and vdom+ modes are 782 and 465 minutes respectively which can be reduced by solving it in parallel. [8]
3.	Sudoku Solver by Q'tron Neural Networks	Yue, T.W. and Lee, Z.C.	Q'tron neural network is an energy driven model with its basic processing node being called a q'tron (quantum neuron). The rules of a sudoku puzzle are developed as a function of energy. This neural network can be used for solving sudoku puzzles as well as for generating them. Solving sudoku puzzles is an np-complete problem which means it is at least as hard as any np problem. Usually sudoku puzzles are solved algorithmically by using graph theories. In the q'tron neural network a solution is found when the energy is least in the energy landscape. This paper describes how to build a q'tron neural network to solve and generate sudoku puzzles. [9]	This sudoku solver does not function correctly in the noise-free mode. Solving the same puzzle with varying initial conditions on free q'trons did not arrive at a solution for 1000 tests. In contrast when the solver is run in full mode, it always arrives at a solution. [9]
4.	A SAT-based Sudoku Solver	Weber, T.	Since there are 6x10^21+ sudoku grids, a naive backtracking algorithm would be computationally infeasible. Here a sudoku puzzle is transformed into a propositional formula that can be satisfied if and only if the puzzle has a solution. This methodology is implemented using the Isabelle/HOL theorem prover by specifying its constraints. This propositional formula is then fed to a standard SAT solver after which the assignment is transformed into a solution.The propositional formula is a polynomial the size of the grid. The translation is a simple process as an existing framework is made use of. The puzzle is encoded by assigning 9 boolean variables for each of the cells in a 9x9 grid (total = 729 variables). These boolean variables are used	Since solving a sudoku puzzle is an np-complete problem, no other algorithm offers better complexity. A 9x9 sudoku puzzle is solved in milliseconds. This method can extend to enumerate all possible solutions to any given sudoku problem. [10]

			to represent a boolean equation for a given constraint in the puzzle. [10]	
5.	A Comparative Study On The Performance CharacteristicsOf Sudoku Solving Algorithms	Chatterjee, Sankhadeep, Saubhik Paladhi, and Raktim Chakraborty.	Comprehending Sudoku, a NP-Complete combinatorial advancement issue has been done utilizing the enhanced Graph Referencing Algorithm (GRA),Genetic Algorithm (GA), Simulated Annealing (SA), Harmony Search (HS) and Brute Force calculation.The present investigation is fundamentally planned for discovering the quickest calculation as far as least time utilization in unraveling Sudoku. The presentation attributes of calculations of intrigue are considered by sending haphazardly chosen riddles with various trouble levels. The similar exhibition qualities study uncovers the prevalence of the Graph Referencing calculation over different calculations in setting aside least conceivable effort to comprehend Sudoku.[4]	Genetic Algorithm may have discovered broad application in taking care of other streamlining issues however neglects to give good outcomes in fathoming Sudoku. Harmony Search calculations pondered genuine concern for its exhibition and can be finished up to be the most fragile endeavor at any point made to take care of Sudoku issues.
6.	Can traditional programming bridge the Ninja performance gap for parallel computing applications?	Satish, N., Kim, C.,Chhugani, J.,Saito, H	Author first measures the degree of the "Ninja hole", which is the presentation hole between gullibly composed C/C++ code that is parallelism unaware(often serial)and best-streamlined code on current multi-/many-center processors. Utilizing a lot of delegate throughput processing benchmarks,he shows that there is a normal Ninja hole of 24X (up to 53X) for an ongoing 6-center Intel® Core™ i7 X980 Westmere CPU,and that this hole whenever left unaddressed will definitely increment. Alse he shows how a lot of notable algorithmic changes combined with headways in present day compiler innovation can cut down the Ninja hole to a normal of just1.3X.[2]	These progressions commonly require low programming exertion, when contrasted with the extremely high exertion in delivering Ninja code
7.	N×N Sudoku Solver	Abdulaziz Aljohani, William	Solving the Sudoku problem is an NP-Complete problem. The paper consists of a sequential program which uses a	The solution in this research paper has a backtracking

	Sequential and Parallel Computing	Smith	backtracking algorithm with an elimination-based technique. Firstly, the program will receive a file that contains all of the needed information to construct a Sudoku board. Next the rows and columns are iterated and a list is produced with all possible values for all the cells. Elimination will let the program avoid unnecessary computation. Finally, we find the row and column of each unassigned cell and assign a number from a possible values list that belongs to the same unassigned cell. In the parallel algorithm, BFS examines all the branches of the problem tree until it hits a certain level which we call threshold, given with a command line. When the program reaches the threshold, it switches the strategy from BFS to DFS. Finally, the current level of execution is tracked in the search tree. This is done by increasing the state's level value each time BFS or DFS is run on a state [5]	algorithm with an elimination-based technique and a parallel work queue from the PJ2 library. The solutions are not really satisfactory and a parallelized CUDA algorithm can efficiently solve such puzzles using the Backtracking algorithm much effectively.
8.	Sudoku Using Parallel Simulated Annealing	Zahra Karimi-Dehkordi, Kamran Zamanifar, Ahmad Baraani-Dastjerdi, and Nasser Ghasem-Aghaee	The combination of Simulated Annealing and Genetic algorithm is the best one. This is because of the parallel nature of the genetic algorithm with the flexibility of Simulated Annealing. Simulated Annealing (SA) is an optimization technique which finds the optimal point by running a series of moves under different thermodynamic conditions. In this paper 3 different parallel SA are implemented in JADE and compared them. The 3 approaches are-domain decomposition, parallel search with jumps, independent parallel search. The results show that parallel search with periodic jumps gets better efficiency and success rate.[6]	Parallel simulated annealing means searching all or different part of search space with a single or multi, improved or produced start point. Domain decomposition makes search space smaller which obviously results in shorter markov chains, but it could not solve Sudoku efficiently. The best one is parallel search through whole space with

				jumps.
9.	Efficient Parallel Sudoku Solver via Thread Management & Data Sharing Methods	Shawn Lee	In this work, we construct parallel implementations of the Sudoku puzzle-solving algorithm using constraint propagation, storing already-explored grids, and storing to-be-explored grids. In the first phase of reducing execution time via multi-threading, we propose to implement a global data structure to store already explored grids, a thread-local data structure to store to-be-explored grids. In the second phase of reducing execution time, we propose to implement a global data structure to store to-be-explored grids along with already-explored grids, and have threads work in the same branch until it is exhausted.[3]	There is a significant Difference in the performance between different thread management and data sharing methods. Particularly, it is optimal to quickly search the depth of the tree to find the solution to the Sudoku puzzle, rather than widely searching shallow branches. Thus, changes to the algorithm may be a necessary part to ensure speedups.
10.	GPU acceleration for Sudoku solution with genetic operations	Yuji Sato, Naohiro Hasegawa, Mikiko Sato	In this paper, we use the problem of solving Sudoku puzzles to demonstrate the possibility of achieving practical processing time through the use of GPUs for parallel processing in the application of genetic computation to problems for which the use of genetic computing has not been investigated before because of the processing time problem. [7]	Evaluation results showed that execution acceleration factors of from 16 to 25 relative to execution of a C program on a CPU are attained but there is no solution for GPUs of high integration and more complex and difficult sudoku problems Also, there is no comparison with the backtracking algorithm.

Table 2.1 Literature Survey

2.1 PROBLEM DEFINITION

Sudoku is a combinatorial, login-based number placement game which originated in Switzerland and was popularized in Japan. Its rules are as follows:

For an NxN grid:

- Only numbers from 1 to N can be used to fill the grid.
- Every mini grid (for example 3x3 mini grid in a 9x9 grid) should contain all numbers from 1 to N with a frequency of 1 each.
- Every row should contain all numbers from 1 to N with a frequency of 1 each.
- Every column should contain all numbers from 1 to N with a frequency of 1 each.

The sudoku solver has been commonly implemented serially. But it proves to be very slow and inefficient as it takes a linear trajectory. Our project aims at speeding up this algorithm through means of parallel computing.

2.2 EXISTING SOLUTION

The researchers using algorithmic procedures and heuristics have worked upon game playing and course of action frameworks truly. Sudoku is one such game which has been engaged toward this way with different approaches and abnormally various alternatives isolated from the standard chart glancing through frameworks has been suggested in the state of workmanship in light of the NP-complete nature of the issue.

BruteForce with Backtracking: In this technique, numbers are filled first and then checked for correctness. If the cell is filled with the wrong value, backtracking is done.

Naked Singles: In this technique, the cells that can have only one possibly value is filled first.

Hidden Singles: From all the values that can be filled in a cell, the values that cannot be filled are found by breaking down the row, column and mini grid.

Naked Pair: In a void cell, two basic potential values can be repeated in two other void cells in the same row/column/mini grid. This common pair is removed from all other void cells.

Hidden Pairs: Hidden sets are discovered and in small groups with a pair that can be filled in a vacant cell.

Naked Triple: 3 common qualities among void cells of a similar line, row or mini grid is used. Common value from a void cell is removed from other void cells.

2.3 ISSUES IN THE EXISTING SOLUTION

The Brute Force algorithm has its own limitations. The main one being increased execution time with increase in input size. In this algorithm, the time needed to find the solution depends upon the number of empty cells in the Sudoku board. In various other algorithms, filling the naked singles can eliminate the choices in another empty cell because the number is already used in either the same row, column or mini grid. The serial code guarantees to find the solution to the Sudoku board if a solution exists because of elimination of empty cells by rule-based methods and then applying brute force to the remaining cells. The serial code will significantly slow down as the size of the Sudoku board increases or may not be able to produce the solution due to computational limitations of the machine.

3. OVERVIEW OF THE WORK

3.1 OBJECTIVES OF THE PROJECT

Our project aims at implementing sudoku solver serially and in parallel by using a combination of the existing algorithms. In the parallel approach, several threads would be spawned to compute parallely. C programming and OpenMP will be used to perform elimination, lone ranger and twins in the given sequence to get an optimized and faster performance. When the grid sizes reach higher numbers like 25, the existing serial algorithm can take a lot of time which will be overcome by implementing it parallely in this project. We would also be evaluating the improvement projected by parallelizing the code.

3.2 SOFTWARE REQUIREMENTS

The software tools used to implement the sudoku solver are:

- Oracle Virtual Machine VirtualBox Manager
- Ubuntu 20.04 Operating System
- GNU compiler
- Text editor

3.3 HARDWARE REQUIREMENTS

In our project, since VMWare workstation is used to create a virtual environment. The hardware requirements are met by using virtual resources. The following virtual resources are used:

- 4GB RAM
- 4 Cores
- 8 Processors
- 20GB Hard disk space

4. SYSTEM DESIGN

4.1 ALGORITHM

Step 1: Start
Step 2: Input the number of threads, size of the grid and the partially filled sudoku grid.
Step 3: Apply heuristics on the input grid. This gives the initial grid for creation of the allotment list.
Step 4: Create a list of grids for allotment among the threads by doing BFS level-by-level on a tree of intermediate grids that has the initial grid at the root. This is done till there are less than thread count many grids in the list.
Step 5: Each thread gets a grid from the allotment list and uses a local stack to execute brute force DFS on it. It then repeats the following till either the solution is found or the stack is empty in which case it gets the next grid.

Step 5.1: Pop a grid from the search stack.
Step 5.2: Apply heuristics on it.
Step 5.3: Expand the tree by selecting the cell with least number and pushing the newly created grids into the stack.
Step 5.4: Whenever the solution is found the thread sets a shared variable indicating this and all the threads exit the parallel section.

Step 6: After the parallel section, the value of the shared variable is checked to figure out if the solution was found. A solution is found if the output grid follows all the rules of sudoku and there are no zeros.
Step 7: Stop

4.2 HEURISTICS FOR THE ALGORITHM

1. Possible values for each cell in the grid store as a bit mask of length 64.
    a. O(1) addition, deletion and searching etc. of possible values and hence is much faster than using an array.
    b. Other operations such as getting a number of possible values etc. are also done in O(1) time using gcc's builtin functions.
2. The application of heuristics is done as follows till none of the heuristics make any change:
    a. Sequence in which the heuristics are applied: Elimination -> Loneranger -> Twins.
    b. If the application of a heuristics causes some change, then the sequence is repeated from the beginning. This ensures that elimination and lone rangers being more useful are applied more frequently.

3. Static allocation of workload among the threads as dynamic allocation didn’t seem to be useful.
    a. Grids assigned in round-robin fashion for similar workloads on all threads.
4. Prune the DFS tree i.e. ignore the branch whenever:
    a. A cell has no possible values
    b. A number doesn’t occur in the possible values of any cell in a row/column/box.
5. Stacks store any grid that gets freed so that a new grid doesn’t need to be allocated from scratch.
6. Parallelized only the DFS section, as each application of the heuristics doesn't take much time and so the overheads of parallelizing them would have been too much.
7. Triplets is not used as it isn’t efficient enough.

4.2 BLOCK DIAGRAM

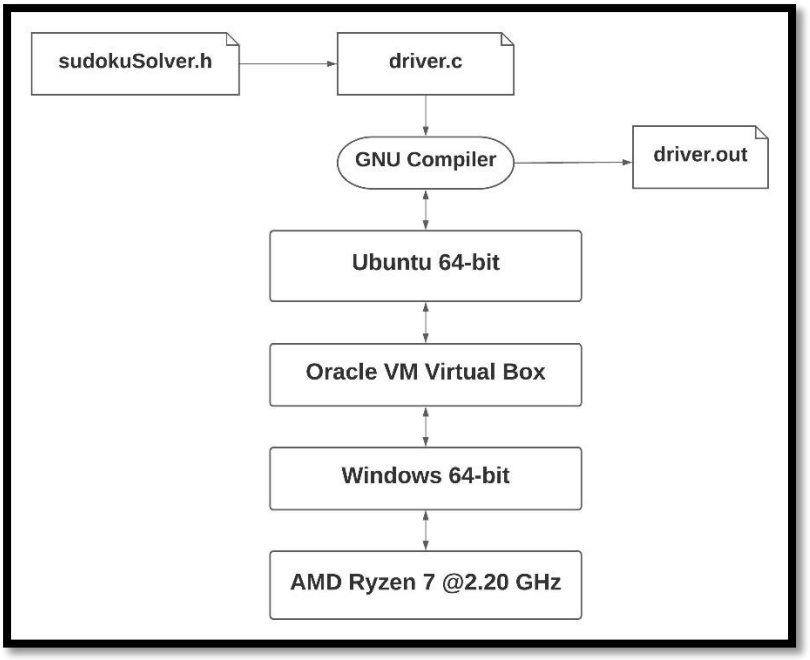

Fig 4.2.1 Block Diagram

4.3 FLOW CHART

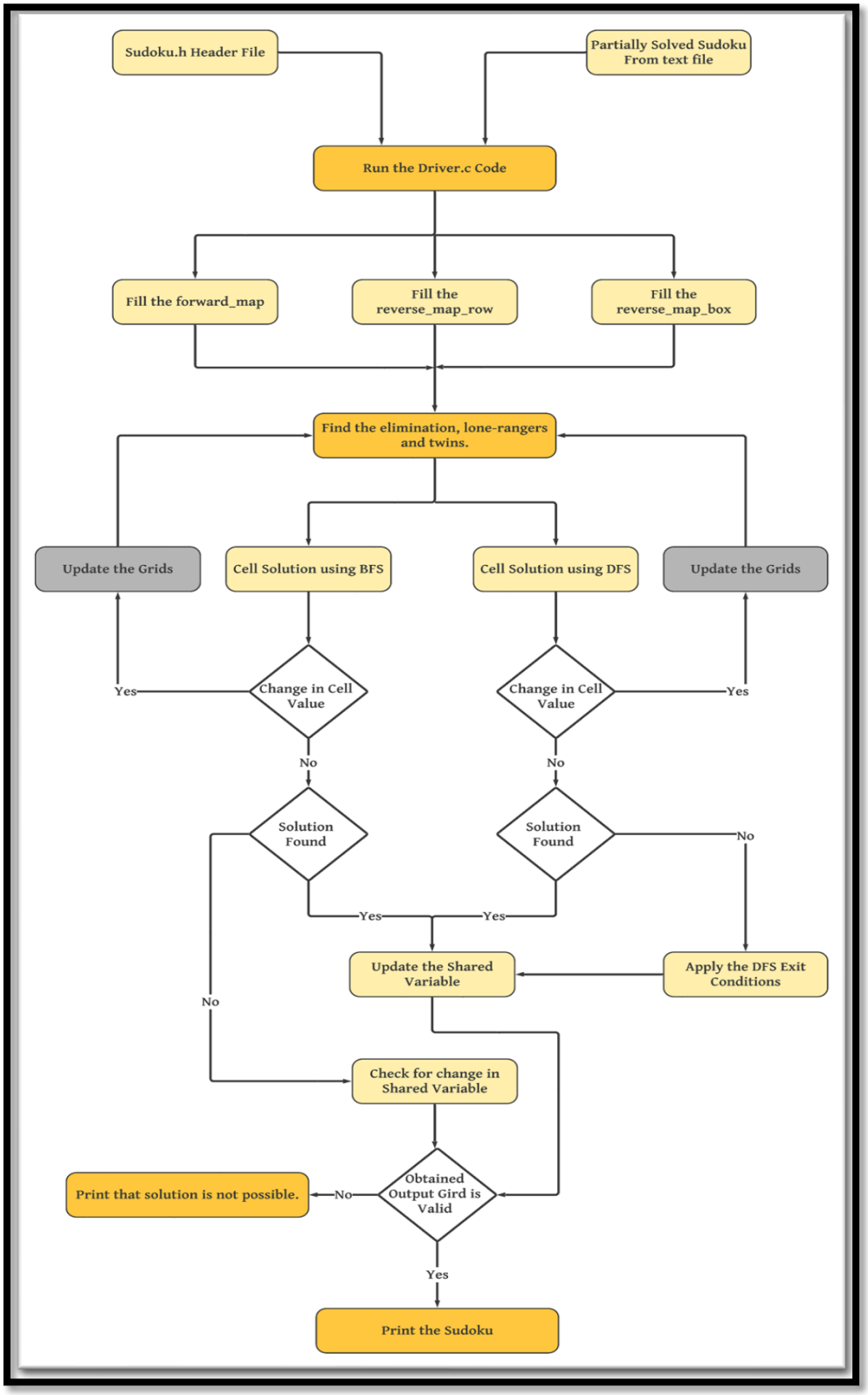

Fig 4.3 Flowchart

5. IMPLEMENTATION

5.1 IDENTIFICATION OF SUDOKU DIFFICULTY

We took some different sudokus with different difficulty levels. Based on the number of null values we are differentiating the difficulty of the sudoku grid.

5.2 DESCRIPTION OF MODULES

driver.c:

driver.c file is the driver file that takes input. The number of threads to be used for execution, input file which contains the partially filled sudoku grid, size of the grid are all taken as input by this module. It also verifies if the sudoku grid and the rest of the inputs are valid. It calls the function solveSudoku from the "sudokuSolver.h" file and also records the execution time. Finally, it verifies the obtained output and verifies it. If the output produced by sudokuSolver is valid it displays the output. Otherwise displays "solution not found".

sudokuSolver.h:

It is a user defined header file that contains the code to solve the sudoku after the input is validated. It first solves the elimination, then, loneranger followed by twins. If the application of a heuristics causes some change, then the sequence is repeated from the beginning. This ensures that elimination and lone rangers being more useful are applied more frequently. The execution carried out in this module is as per the algorithm given in section 4.1. After solving the sudoku it is then sent back to "driver.c".

5.3 SOURCE CODE

5.3.1 SERIAL IMPLEMENTATION

main-serial.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include "sudoku-serial.h"
#include <string.h>
#include <sys/time.h>

int ** readInput(char *filename){
        FILE *infile;
        infile = fopen(filename, "r");
        int i, j;
        char dummyline[SIZE+1];
        char dummy;
        int value;
        int **sudokuGrid = (int**)malloc(sizeof(int*)*SIZE);
        for (i=0;i<SIZE;i++){
                sudokuGrid[i] = (int*)malloc(sizeof(int)*SIZE);
                for (j=0;j<SIZE;j++)
                        sudokuGrid[i][j] = 0;
        }

        for (i = 0; i < SIZE; i++){
                for (j = 0; j < SIZE; j++){
                        if (feof(infile)){
                                if (i != SIZE){
                                        printf("The input puzzle has less number of rows than %d. Exiting.\n",
SIZE);
                                        exit(-1);
                                }
                        }

                        fscanf(infile, "%d", &value);
                        if(value >= 0 && value <= SIZE)
                                sudokuGrid[i][j] = value;
                        else{
                                printf("The input puzzle is not a grid of numbers (0 <= n <= %d) of size %dx%d.
Exiting.\n", SIZE, SIZE, SIZE);
                                exit(-1);
                        }
                }
                fscanf(infile, "%c", &dummy);

                if (j > SIZE){
                        printf("Row %d has more number of elements than %d. Exiting.\n", i+1, SIZE);
                        exit(-1);
                }
        }
        return sudokuGrid;
}


int isValid(int **original, int **solution){
        int valuesSeen[SIZE],i,j,k;

        for (i=0;i<SIZE;i++){
                for (k=0;k<SIZE;k++) valuesSeen[k] = 0;
                for (j=0;j<SIZE;j++){
                        if (solution[i][j]==0) return 0;
                        if ((original[i][j])&&(solution[i][j] != original[i][j])) return 0;
                        int v = solution[i][j];
                        if (valuesSeen[v-1]==1){
                                return 0;
                        }
                        valuesSeen[v-1] = 1;
                }
        }

        for (i=0;i<SIZE;i++){
                for (k=0;k<SIZE;k++) valuesSeen[k] = 0;
                for (j=0;j<SIZE;j++){
                        int v = solution[j][i];
                        if (valuesSeen[v-1]==1){
                                return 0;
                        }
                        valuesSeen[v-1] = 1;
                }
        }

        for (i=0;i<SIZE;i=i+MINIGRIDSIZE){
                for (j=0;j<SIZE;j=j+MINIGRIDSIZE){
                        for (k=0;k<SIZE;k++) valuesSeen[k] = 0;
                        int r,c;
                        for (r=i;r<i+MINIGRIDSIZE;r++)
                                for (c=j;c<j+MINIGRIDSIZE;c++){
                                        int v = solution[r][c];
                                        if (valuesSeen[v-1]==1) {
                                                return 0;
                                        }
                                        valuesSeen[v-1] = 1;
                                }
                }
```

```
        }
        return 1;
}

int main(){
        printf("Enter path to the input file relative to ~/Desktop/PDC: ");
        char path[50];
        scanf("%s",path);

        int **originalGrid = readInput(path);
        int **gridToSolve = readInput(path);
        int i,j;
        printf("INPUT GRID\n");
        for (i=0;i<SIZE;i++){
                for (j=0;j<SIZE;j++){
                        printf("%d ",originalGrid[i][j]);
                }
                printf("\n");
        }
        printf("\n");

        struct timeval start, end;
        gettimeofday(&start, 0);

        int **outputGrid = solveSudoku(originalGrid);
        gettimeofday(&end, 0);
        int sec = end.tv_sec-start.tv_sec;
        int usec = end.tv_usec-start.tv_usec;

        printf("\nOUTPUT GRID\n");
        for (i=0;i<SIZE;i++){
                for (j=0;j<SIZE;j++)
                        printf("%d ",outputGrid[i][j]);
                printf("\n");
        }
        printf("\n");
        if (isValid(originalGrid,outputGrid)){
                printf("SOLUTION FOUND\n");
                printf("Elapsed Time: %lf sec\n", (sec*1000+(usec/1000.0))/1000);
        }
        else{
                printf("NO SOLUTION FOUND\n");
        }
}
```

sudoku-serial.h:

```
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include<sys/time.h>

#define SIZE 9
#define MINIGRIDSIZE 3

int empty_cells=0;

struct pair_t{
    int x,y;
};
typedef struct pair_t pair;
pair empty_cells_list[SIZE*SIZE];

struct queue_element{
    int board[SIZE][SIZE];
    int forward_map[SIZE][SIZE][SIZE];
};

struct queue
{
    struct queue_element **list;
    int size,length;
    int head,tail;
};

int forward_map[SIZE][SIZE][SIZE];
int found = 0;
int **final_board;
int reverse_map_row[SIZE][SIZE][SIZE],reverse_map_column[SIZE][SIZE][SIZE];
int reverse_map_box[SIZE][SIZE][MINIGRIDSIZE][MINIGRIDSIZE];

void init_queue(struct queue *q, int length)
{
    q->list = malloc(sizeof(struct queue_element*)*length);
    q->head=0;
    q->tail=0;
    q->length=0;
    q->size=length;
}
```

```
void push(struct queue *q, struct queue_element *elem)
{
    if((q->tail+1)%q->size==(q->head))
    {
        printf("Queue full!!! PAAANIIIC\n");
        exit(0);
    }
    q->list[q->tail] = elem;
    q->tail = ((q->tail)+1)%(q->size);
    q->length++;
}

struct queue_element* pop(struct queue *q)
{
    if(q->head == q->tail)
    {
        printf("Queue empty!!! PAAANIIIC\n");
        exit(0);
    }
    struct queue_element* ret = q->list[q->head];
    q->head = ((q->head)+1)%(q->size);
    q->length--;
}

void populate_f(int x, int y, int val, int map[SIZE][SIZE][SIZE])
{
    if(val!=0)
    {
        int j,k;
        for(j=0;j<SIZE;j++)
        {
            if(j!=x)
                map[j][y][val-1]=1;
            if(j!=y)
                map[x][j][val-1]=1;
            if(j!=val-1)
                map[x][y][j]=1;
        }
        j=x-x%MINIGRIDSIZE;
        k=y-y%MINIGRIDSIZE;

        int jj,kk;
        for(jj=j;jj<j+MINIGRIDSIZE;jj++)
            for(kk=k;kk<k+MINIGRIDSIZE;kk++)
                if(jj!=x || kk!=y)
                    map[jj][kk][val-1]=1;

    }
}

void dfs_populate_f(int x, int y, int val, int map[SIZE][SIZE][SIZE])
{
    if(val!=0)
    {
        int j,k;
        for(j=0;j<SIZE;j++)
        {
            if(j!=x)
                map[j][y][val-1]=1;
            if(j!=y)
                map[x][j][val-1]=1;
        }
        j=x-x%MINIGRIDSIZE;
        k=y-y%MINIGRIDSIZE;

        int jj,kk;
        for(jj=j;jj<j+MINIGRIDSIZE;jj++)
            for(kk=k;kk<k+MINIGRIDSIZE;kk++)
                if(jj!=x || kk!=y)
                    map[jj][kk][val-1]=1;

    }
}

int findPosition(int x, int y)
{
    int i,single=0,idx=-1;
    for(i=0; i<SIZE; i++)
    {
        if(single)
        {
            if(forward_map[x][y][i] == 0)
                return -1;
        }
        else
        {
```

```
            if(forward_map[x][y][i] == 0)
            {
                single = 1;
                idx = i+1;
            }
        }
    }
    return idx;
}

int findCol(int x, int y, int **inp)
{
    int i,single=0,idx=-1;
    for(i=0; i<SIZE; i++)
    {
        if(single)
        {
            if(forward_map[x][i][y] == 0 && inp[x][i]==0)
                return -1;
        }
        else
        {
            if(forward_map[x][i][y] == 0 && inp[x][i]==0)
            {
                single = 1;
                idx = i;
            }
        }
    }
    return idx;
}

int findRow(int x, int y, int **inp)
{
    int i,single=0,idx=-1;
    for(i=0; i<SIZE; i++)
    {
        if(single)
        {
            if(forward_map[i][x][y] == 0 && inp[i][x]==0)
                return -1;
        }
        else

        {
            if(forward_map[i][x][y] == 0 && inp[i][x]==0)
            {
                single = 1;
                idx = i;
            }
        }
    }
    return idx;
}

pair findCell(int x, int y, int val, int **inp)
{
    int i,j,single=0;
    pair idx;
    idx.x=-1;
    for(i=x; i<x+MINIGRIDSIZE; i++)
    {
        for(j=y; j<y+MINIGRIDSIZE; j++)
        {
            if(single)
            {
                if(forward_map[i][j][val] == 0 && inp[i][j]==0)
                {
                    idx.x=-1;
                    idx.y=-1;
                    return idx;
                }
            }
            else
            {
                if(forward_map[i][j][val] == 0 && inp[i][j]==0)
                {
                    single = 1;
                    idx.x = i;
                    idx.y = j;
                }
            }
        }
    }
    return idx;
}
```

```
void dfs(int board[SIZE][SIZE], int forward_map[SIZE][SIZE][SIZE], int idx)
{
    if(idx==empty_cells)
    {
        found = 1;
        int i,j;
        for(i=0;i<SIZE;i++)
            for(j=0;j<SIZE;j++)
                final_board[i][j]=board[i][j];
        return;
    }
    else if(found)
        return;
    else
    {
        int val;
        for(val=0; val<SIZE; val++)
        {
            int x = empty_cells_list[idx].x;
            int y = empty_cells_list[idx].y;
            if(forward_map[x][y][val]==0)
            {
                int row_bkp[SIZE], col_bkp[SIZE], box_bkp[MINIGRIDSIZE][MINIGRIDSIZE], val_bkp[SIZE];

                int j,k;
                for(j=0;j<SIZE;j++)
                {
                    if(j!=x)
                        col_bkp[j]=forward_map[j][y][val];
                    if(j!=y)
                        row_bkp[j]=forward_map[x][j][val];
                }
                j=x-x%MINIGRIDSIZE;
                k=y-y%MINIGRIDSIZE;

                int jj,kk;
                for(jj=j;jj<j+MINIGRIDSIZE;jj++)
                    for(kk=k;kk<k+MINIGRIDSIZE;kk++)
                        if(jj!=x || kk!=y)
                            box_bkp[jj-j][kk-k]=forward_map[jj][kk][val];

                board[x][y]=val+1;
                dfs_populate_f(x,y,val+1,forward_map);
                dfs(board,forward_map,idx+1);

                for(j=0;j<SIZE;j++)
                {
                    if(j!=x)
                        forward_map[j][y][val]=col_bkp[j];
                    if(j!=y)
                        forward_map[x][j][val]=row_bkp[j];
                }
                j=x-x%MINIGRIDSIZE;
                k=y-y%MINIGRIDSIZE;

                for(jj=j;jj<j+MINIGRIDSIZE;jj++)
                    for(kk=k;kk<k+MINIGRIDSIZE;kk++)
                        if(jj!=x || kk!=y)
                            forward_map[jj][kk][val]=box_bkp[jj-j][kk-k];

                board[x][y]=0;
            }
        }
    }
}

int **solveSudoku(int ** inp)
{
    final_board=malloc(sizeof(int*)*SIZE);
    int i,j;
    for(i=0;i<SIZE;i++)
        final_board[i]=malloc(sizeof(int)*SIZE);
    memset(forward_map,0,sizeof(forward_map));
    memset(reverse_map_row,0,sizeof(reverse_map_row));
    memset(reverse_map_column,0,sizeof(reverse_map_column));
    memset(reverse_map_box,0,sizeof(reverse_map_box));


    for(i=0;i<SIZE*SIZE;i++)
    {
        int x=i%SIZE;
        int y=i/SIZE;
        int val=inp[x][y];
        if(val>0)
            populate_f(x,y,val,forward_map);
    }
```

```
int changed_outer=1;
while(changed_outer)
{
    int changed=1;
    changed_outer=0;
    while(changed)
    {
        changed=0;
        {
            for(j=0;j<SIZE*SIZE;j++)
            {
                int x=j%SIZE, y=j/SIZE;
                if(inp[x][y]!=0)
                {
                    continue;
                }
                int pos = findPosition(x,y);
                if(pos>0)
                {
                    inp[x][y]=pos;
                    populate_f(x,y,pos,forward_map);
                    changed=1;
                }
            }
            for(j=0;j<SIZE;j++)
            {
                int p;
                for(p=0;p<SIZE;p++)
                {
                    int pos = findCol(j,p,inp);
                    if(pos>=0)
                    {
                        inp[j][pos]=p+1;
                        populate_f(j,pos,p+1,forward_map);
                        changed=1;
                    }

                    pos = findRow(j,p,inp);
                    if(pos>=0)
                    {
                        inp[pos][j]=p+1;
                        populate_f(pos,j,p+1,forward_map);
                        changed=1;


                    }

                    int x=(j%MINIGRIDSIZE)*MINIGRIDSIZE, y=(j/MINIGRIDSIZE)*MINIGRIDSIZE;
                    pair p1 = findCell(x,y,p,inp);
                    if(p1.x>=0)
                    {
                        inp[p1.x][p1.y]=p+1;
                        populate_f(p1.x,p1.y,p+1,forward_map);
                        changed=1;
                    }
                }
            }

        }
    }
    int k;
    for(k=0;k<SIZE;k++)
    {
        for(i=0;i<SIZE;i++)
        {
            for(j=i+1;j<SIZE;j++)
            {
                int l,cnt=0,first=0,second=0,position[2];
                for(l=0;l<SIZE;l++)
                {
                    if(forward_map[k][l][i]==0){
                        first|=1<<l;
                        if(cnt<=1)
                            position[cnt]=l;
                        else
                        {
                            cnt=0;
                            break;
                        }
                        cnt++;

                    }
                    if(forward_map[k][l][j]==0)
                        second|=1<<l;
                }
                if(cnt==2 && second==first)
                {
                    for(l=0;l<SIZE;l++)
```

```
                {
                    forward_map[k][position[0]][l]=1;
                    forward_map[k][position[1]][l]=1;
                }
                forward_map[k][position[0]][j]=0;
                forward_map[k][position[0]][i]=0;
                forward_map[k][position[1]][j]=0;
                forward_map[k][position[1]][i]=0;
            }
        }
    }
}
for(k=0;k<SIZE;k++)
{
    for(i=0;i<SIZE;i++)
    {
        for(j=i+1;j<SIZE;j++)
        {
            int l,cnt=0,first=0,second=0,position[2];
            for(l=0;l<SIZE;l++)
            {
                if(forward_map[l][k][i]==0){
                    first|=1<<l;
                    if(cnt<=1)
                        position[cnt]=l;
                    else
                    {
                        cnt=0;
                        break;
                    }
                    cnt++;

                }
                if(forward_map[l][k][j]==0)
                    second|=1<<l;
            }
            if(cnt==2 && second==first)
            {
                for(l=0;l<SIZE;l++)
                {
                    forward_map[position[0]][k][l]=1;
                    forward_map[position[1]][k][l]=1;
                 }
                 forward_map[position[0]][k][j]=0;
                 forward_map[position[0]][k][i]=0;
                 forward_map[position[1]][k][j]=0;
                 forward_map[position[1]][k][i]=0;
             }
        }
    }
}

for(k=0;k<SIZE;k++)
{
    int x=(k%MINIGRIDSIZE)*MINIGRIDSIZE, y=(k/MINIGRIDSIZE)*MINIGRIDSIZE;
    for(i=0;i<SIZE;i++)
    {
        for(j=i+1;j<SIZE;j++)
        {
            int l1,l2,cnt=0,first=0,second=0,positionx[2],positiony[2];
            for(l1=x;l1<x+MINIGRIDSIZE;l1++)
                for(l2=y;l2<y+MINIGRIDSIZE;l2++)
                {
                    if(forward_map[l1][l2][i]==0){
                        first|=1<<(l1*MINIGRIDSIZE+l2);
                        if(cnt<=1)
                        {
                            positionx[cnt]=l1;
                            positiony[cnt]=l2;
                        }
                        else
                        {
                            cnt=0;
                            break;
                        }
                        cnt++;

                    }
                    if(forward_map[l1][l2][j]==0)
                        second|=1<<(l1*MINIGRIDSIZE+l2);
                }
            if(cnt==2 && second==first)
            {
                for(l1=0;l1<SIZE;l1++)
                {
```

```
                        forward_map[positionx[0]][positiony[0]][l1]=1;
                        forward_map[positionx[1]][positiony[1]][l1]=1;
                    }
                    forward_map[positionx[0]][positiony[0]][j]=0;
                    forward_map[positionx[0]][positiony[0]][i]=0;
                    forward_map[positionx[1]][positiony[1]][j]=0;
                    forward_map[positionx[1]][positiony[1]][i]=0;
                }
            }
        }
    }
    final_board=inp;
    for(i=0; i<SIZE*SIZE; i++)
    {
        int x=i%SIZE, y=i/SIZE;
        if(inp[x][y]==0)
        {
            int ind;
            ind=empty_cells++;
            empty_cells_list[ind].x=x;
            empty_cells_list[ind].y=y;
        }
    }
    struct queue *q = malloc(sizeof(struct queue));
    int num_threads=1;
    int idx=0;
    {

        int k;
        {
            init_queue(q, num_threads*SIZE);
            struct queue_element *elem;
            elem = malloc(sizeof(struct queue_element));
            for(i=0;i<SIZE;i++)
                for(j=0;j<SIZE;j++)
                    elem->board[i][j]=inp[i][j];

            for(i=0;i<SIZE;i++)
                for(j=0;j<SIZE;j++)
                    for(k=0;k<SIZE;k++)
                        elem->forward_map[i][j][k]=forward_map[i][j][k];


            push(q,elem);
            while(q->length < num_threads && idx<empty_cells)
            {
                int l=q->length;
                int i;
                struct queue *tmp=malloc(sizeof(struct queue));
                init_queue(tmp,l*SIZE);
                for(i=0;i<l;i++)
                {
                    int j;
                    for(j=0;j<SIZE;j++)
                    {
                        if(forward_map[empty_cells_list[idx].x][empty_cells_list[idx].y][j]==0)
                        {
                            int a,b,c;
                            struct queue_element * temp=malloc(sizeof(struct queue_element));
                            for(a=0;a<SIZE;a++)
                                for(b=0;b<SIZE;b++)
                                    temp->board[a][b]=q->list[i]->board[a][b];


                            for(a=0;a<SIZE;a++)
                                for(b=0;b<SIZE;b++)
                                    for(c=0;c<SIZE;c++)
                                        temp->forward_map[a][b][c]=q->list[i]->forward_map[a][b][c];

                            temp->board[empty_cells_list[idx].x][empty_cells_list[idx].y]=j+1;
                            populate_f(empty_cells_list[idx].x,empty_cells_list[idx].y,j+1,temp->forward_map);
                            push(tmp,temp);
                        }
                    }
                }
                idx++;
                free(q);
                q=tmp;
            }

        }
        for(i=0; i<q->length;i++)
        {
            dfs(q->list[i]->board,q->list[i]->forward_map,idx);
        }
    }
    return final_board;
}
```

5.3.2 PARALLEL IMPLEMENTATION

driver.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include "sudokuSolver.h"
#include <string.h>
#include <omp.h>
#include <sys/time.h>

int thread_count = 4;

int ** readInput(char *filename){
        FILE *infile;
        infile = fopen(filename, "r");
        int i, j;
        char dummyline[SIZE+1];
        char dummy;
        int value;
        int **sudokuGrid = (int**)malloc(sizeof(int*)*SIZE);
        for (i=0;i<SIZE;i++){
                sudokuGrid[i] = (int*)malloc(sizeof(int)*SIZE);
                for (j=0;j<SIZE;j++)
                        sudokuGrid[i][j] = 0;
        }

        for (i = 0; i < SIZE; i++){
                for (j = 0; j < SIZE; j++){
                        if (feof(infile)){
                                if (i != SIZE){
                                        printf("The input puzzle has less number of rows than %d. Exiting.\n", SIZE);
                                        exit(-1);
                                }
                        }

                        fscanf(infile, "%d", &value);
                        if(value >= 0 && value <= SIZE)
                                sudokuGrid[i][j] = value;
                        else{
                                printf("The input puzzle is not a grid of numbers (0 <= n <= %d) of size %dx%d. Exiting.\n", SIZE, SIZE, SIZE);
                                exit(-1);
                        }
                }
                fscanf(infile, "%c", &dummy);

                if (j > SIZE){
                        printf("Row %d has more number of elements than %d. Exiting.\n", i+1, SIZE);
                        exit(-1);
                 }
        }
        return sudokuGrid;
}

int isValid(int **original, int **solution){
        int valuesSeen[SIZE],i,j,k;

        for (i=0;i<SIZE;i++){
                for (k=0;k<SIZE;k++) valuesSeen[k] = 0;
                for (j=0;j<SIZE;j++){
                        if (solution[i][j]==0) return 0;
                        if ((original[i][j])&&(solution[i][j] != original[i][j])) return 0;
                        int v = solution[i][j];
                        if (valuesSeen[v-1]==1){
                                return 0;
                        }
                        valuesSeen[v-1] = 1;
                }
        }

        for (i=0;i<SIZE;i++){
                for (k=0;k<SIZE;k++) valuesSeen[k] = 0;
                for (j=0;j<SIZE;j++){
                        int v = solution[j][i];
                        if (valuesSeen[v-1]==1){
                                return 0;
                        }
                        valuesSeen[v-1] = 1;
                }
        }

        for (i=0;i<SIZE;i=i+MINIGRIDSIZE){
                for (j=0;j<SIZE;j=j+MINIGRIDSIZE){
                        for (k=0;k<SIZE;k++) valuesSeen[k] = 0;
                        int r,c;
                        for (r=i;r<i+MINIGRIDSIZE;r++)
                                for (c=j;c<j+MINIGRIDSIZE;c++){
                                        int v = solution[r][c];
                                        if (valuesSeen[v-1]==1) {
                                                return 0;
                                        }
                                        valuesSeen[v-1] = 1;
                                }
                }
```

```
        }
        return 1;
}

int main(){
        printf("Enter path to the input file relative to ~/Desktop/PDC: ");
        char path[50];
        scanf("%s",path);

        int **originalGrid = readInput(path);
        int **gridToSolve = readInput(path);
        printf("Enter number of threads: ");
        scanf("%d", &thread_count);
        if (thread_count<=0){
                printf("Usage: Thread Count should be positive\n");
        }
        omp_set_num_threads(thread_count);

        int i,j;
        printf("\nINPUT GRID\n");
        for (i=0;i<SIZE;i++){
                for (j=0;j<SIZE;j++){
                        printf("%d ",originalGrid[i][j]);
                }
                printf("\n");
        }
        printf("\n");

        struct timeval start, end;
        gettimeofday(&start, 0);

        int **outputGrid = solveSudoku(originalGrid);
        gettimeofday(&end, 0);
        int sec = end.tv_sec-start.tv_sec;
        int usec = end.tv_usec-start.tv_usec;

        printf("\nOUTPUT GRID\n");
        for (i=0;i<SIZE;i++){
                for (j=0;j<SIZE;j++)
                        printf("%d ",outputGrid[i][j]);
                printf("\n");
        }
        printf("\n");
        if (isValid(originalGrid,outputGrid)){
                printf("SOLUTION FOUND\n");
                printf("Elapsed Time: %lf sec\n", (sec*1000+(usec/1000.0))/1000);
        }
        else{
                printf("NO SOLUTION FOUND\n");
        }
}
```

sudoku.h:

```
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

#define SIZE 16
#define MINIGRIDSIZE 4

int empty_cells=0;

struct pair_t{
    int x,y;
};
typedef struct pair_t pair;
pair empty_cells_list[SIZE*SIZE];

struct queue_element{
    int board[SIZE][SIZE];
    int forward_map[SIZE][SIZE][SIZE];
};

struct queue
{
    struct queue_element **list;
    int size,length;
    int head,tail;
};

int forward_map[SIZE][SIZE][SIZE];
int found = 0;
int **final_board;
int reverse_map_row[SIZE][SIZE][SIZE],reverse_map_column[SIZE][SIZE][SIZE];
int reverse_map_box[SIZE][SIZE][MINIGRIDSIZE][MINIGRIDSIZE];

void init_queue(struct queue *q, int length)
{
    q->list = malloc(sizeof(struct queue_element*)*length);
    q->head=0;
    q->tail=0;
    q->length=0;
    q->size=length;
}

void push(struct queue *q, struct queue_element *elem)
```

```
{
    if((q->tail+1)%q->size==(q->head))
    {
        exit(0);
    }
    q->list[q->tail] = elem;
    q->tail = ((q->tail)+1)%(q->size);
    q->length++;
}

struct queue_element* pop(struct queue *q)
{
    if(q->head == q->tail)
    {
        exit(0);
    }
    struct queue_element* ret = q->list[q->head];
    q->head = ((q->head)+1)%(q->size);
    q->length--;
}

void populate_f(int x, int y, int val, int map[SIZE][SIZE][SIZE])
{
    if(val!=0)
    {
        int j,k;
        for(j=0;j<SIZE;j++)
        {
            if(j!=x)
                map[j][y][val-1]=1;
            if(j!=y)
                map[x][j][val-1]=1;
            if(j!=val-1)
                map[x][y][j]=1;
        }
        j=x-x%MINIGRIDSIZE;
        k=y-y%MINIGRIDSIZE;

        int jj,kk;
        for(jj=j;jj<j+MINIGRIDSIZE;jj++)
            for(kk=k;kk<k+MINIGRIDSIZE;kk++)
                if(jj!=x || kk!=y)
                    map[jj][kk][val-1]=1;

    }
}

void dfs_populate_f(int x, int y, int val, int map[SIZE][SIZE][SIZE])
{
    if(val!=0)
    {
        int j,k;
        for(j=0;j<SIZE;j++)
        {
            if(j!=x)
                map[j][y][val-1]=1;
            if(j!=y)
                map[x][j][val-1]=1;
        }
        j=x-x%MINIGRIDSIZE;
        k=y-y%MINIGRIDSIZE;

        int jj,kk;
        for(jj=j;jj<j+MINIGRIDSIZE;jj++)
            for(kk=k;kk<k+MINIGRIDSIZE;kk++)
                if(jj!=x || kk!=y)
                    map[jj][kk][val-1]=1;

    }
}

int findPosition(int x, int y)
{
    int i,single=0,idx=-1;
    for(i=0; i<SIZE; i++)
    {
        if(single)
        {
            if(forward_map[x][y][i] == 0)
                return -1;
        }
        else
        {
            if(forward_map[x][y][i] == 0)
            {
                single = 1;
                idx = i+1;
            }
        }
    }
    return idx;
```

```
}

int findCol(int x, int y, int **inp)
{
    int i,single=0,idx=-1;
    for(i=0; i<SIZE; i++)
    {
        if(single)
        {
            if(forward_map[x][i][y] == 0 && inp[x][i]==0)
                return -1;
        }
        else
        {
            if(forward_map[x][i][y] == 0 && inp[x][i]==0)
            {
                single = 1;
                idx = i;
            }
        }
    }
    return idx;
}

int findRow(int x, int y, int **inp)
{
    int i,single=0,idx=-1;
    for(i=0; i<SIZE; i++)
    {
        if(single)
        {
            if(forward_map[i][x][y] == 0 && inp[i][x]==0)
                return -1;
        }
        else
        {
            if(forward_map[i][x][y] == 0 && inp[i][x]==0)
            {
                single = 1;
                idx = i;
            }
        }
    }
    return idx;
}

pair findCell(int x, int y, int val, int **inp)
{
    int i,j,single=0;
    pair idx;
    idx.x=-1;
    for(i=x; i<x+MINIGRIDSIZE; i++)
    {
        for(j=y; j<y+MINIGRIDSIZE; j++)
        {
            if(single)
            {
                if(forward_map[i][j][val] == 0 && inp[i][j]==0)
                {
                    idx.x=-1;
                    idx.y=-1;
                    return idx;
                }
            }
            else
            {
                if(forward_map[i][j][val] == 0 && inp[i][j]==0)
                {
                    single = 1;
                    idx.x = i;
                    idx.y = j;
                }
            }
        }
    }
    return idx;
}

void dfs(int board[SIZE][SIZE], int forward_map[SIZE][SIZE][SIZE], int idx)
{
    if(idx==empty_cells)
    {
        found = 1;
        int i,j;
        for(i=0;i<SIZE;i++)
            for(j=0;j<SIZE;j++)
                final_board[i][j]=board[i][j];
        return;
    }
    else if(found)
        return;
     -
```

```
    else
    {
        int val;
        for(val=0; val<SIZE; val++)
        {
            int x = empty_cells_list[idx].x;
            int y = empty_cells_list[idx].y;
            if(forward_map[x][y][val]==0)
            {
                int row_bkp[SIZE], col_bkp[SIZE], box_bkp[MINIGRIDSIZE][MINIGRIDSIZE], val_bkp[SIZE];

                int j,k;
                for(j=0;j<SIZE;j++)
                {
                    if(j!=x)
                        col_bkp[j]=forward_map[j][y][val];
                    if(j!=y)
                        row_bkp[j]=forward_map[x][j][val];
                }
                j=x-x%MINIGRIDSIZE;
                k=y-y%MINIGRIDSIZE;

                int jj,kk;
                for(jj=j;jj<j+MINIGRIDSIZE;jj++)
                    for(kk=k;kk<k+MINIGRIDSIZE;kk++)
                        if(jj!=x || kk!=y)
                            box_bkp[jj-j][kk-k]=forward_map[jj][kk][val];

                board[x][y]=val+1;
                dfs_populate_f(x,y,val+1,forward_map);
                dfs(board,forward_map,idx+1);

                for(j=0;j<SIZE;j++)
                {
                    if(j!=x)
                        forward_map[j][y][val]=col_bkp[j];
                    if(j!=y)
                        forward_map[x][j][val]=row_bkp[j];
                }
                j=x-x%MINIGRIDSIZE;
                k=y-y%MINIGRIDSIZE;

                for(jj=j;jj<j+MINIGRIDSIZE;jj++)
                    for(kk=k;kk<k+MINIGRIDSIZE;kk++)
                        if(jj!=x || kk!=y)
                            forward_map[jj][kk][val]=box_bkp[jj-j][kk-k];

                board[x][y]=0;
            }
        }
    }
}

int **solveSudoku(int ** inp)
{
    final_board=malloc(sizeof(int*)*SIZE);
    int i,j;
    for(i=0;i<SIZE;i++)
        final_board[i]=malloc(sizeof(int)*SIZE);
    memset(forward_map,0,sizeof(forward_map));
    memset(reverse_map_row,0,sizeof(reverse_map_row));
    memset(reverse_map_column,0,sizeof(reverse_map_column));
    memset(reverse_map_box,0,sizeof(reverse_map_box));


    for(i=0;i<SIZE*SIZE;i++)
    {
        int x=i%SIZE;
        int y=i/SIZE;
        int val=inp[x][y];
        if(val>0)
            populate_f(x,y,val,forward_map);
    }
    int changed_outer=1;
    while(changed_outer)
    {
        int changed=1;
        changed_outer=0;
        while(changed)
        {
            changed=0;
            #pragma omp parallel
            {
                #pragma omp for
                for(j=0;j<SIZE*SIZE;j++)
                {
                    int x=j%SIZE, y=j/SIZE;
                    if(inp[x][y]!=0)
                    {
                        continue;
```

```
            }
            int pos = findPosition(x,y);
            if(pos>0)
            {
                inp[x][y]=pos;
                populate_f(x,y,pos,forward_map);
                changed=1;
            }
        }
        #pragma omp for
        for(j=0;j<SIZE;j++)
        {
            int p;
            for(p=0;p<SIZE;p++)
            {
                int pos = findCol(j,p,inp);
                if(pos>=0)
                {
                    inp[j][pos]=p+1;
                    populate_f(j,pos,p+1,forward_map);
                    changed=1;
                }
                pos = findRow(j,p,inp);
                if(pos>=0)
                {
                    inp[pos][j]=p+1;
                    populate_f(pos,j,p+1,forward_map);
                    changed=1;
                }

                int x=(j%MINIGRIDSIZE)*MINIGRIDSIZE, y=(j/MINIGRIDSIZE)*MINIGRIDSIZE;
                pair p1 = findCell(x,y,p,inp);
                if(p1.x>=0)
                {
                    inp[p1.x][p1.y]=p+1;
                    populate_f(p1.x,p1.y,p+1,forward_map);
                    changed=1;
                }
            }
        }

    }
}
int k;
for(k=0;k<SIZE;k++)
{
    for(i=0;i<SIZE;i++)
    {
        for(j=i+1;j<SIZE;j++)
        {
            int l,cnt=0,first=0,second=0,position[2];
            for(l=0;l<SIZE;l++)
            {
                if(forward_map[k][l][i]==0){
                    first|=1<<l;
                    if(cnt<=1)
                        position[cnt]=l;
                    else
                    {
                        cnt=0;
                        break;
                    }
                    cnt++;

                }
                if(forward_map[k][l][j]==0)
                    second|=1<<l;
            }
            if(cnt==2 && second==first)
            {
                for(l=0;l<SIZE;l++)
                {
                    forward_map[k][position[0]][l]=1;
                    forward_map[k][position[1]][l]=1;
                }
                forward_map[k][position[0]][j]=0;
                forward_map[k][position[0]][i]=0;
                forward_map[k][position[1]][j]=0;
                forward_map[k][position[1]][i]=0;
            }
        }
    }
}
for(k=0;k<SIZE;k++)
{
    for(i=0;i<SIZE;i++)
    {
        for(j=i+1;j<SIZE;j++)
        {
            int l,cnt=0,first=0,second=0,position[2];
```

```
                for(l=0;l<SIZE;l++)
                {
                    if(forward_map[l][k][i]==0){
                        first|=1<<l;
                        if(cnt<=1)
                            position[cnt]=l;
                        else
                        {
                            cnt=0;
                            break;
                        }
                        cnt++;

                    }
                    if(forward_map[l][k][j]==0)
                        second|=1<<l;
                }
                if(cnt==2 && second==first)
                {
                    for(l=0;l<SIZE;l++)
                    {
                        forward_map[position[0]][k][l]=1;
                        forward_map[position[1]][k][l]=1;
                    }
                    forward_map[position[0]][k][j]=0;
                    forward_map[position[0]][k][i]=0;
                    forward_map[position[1]][k][j]=0;
                    forward_map[position[1]][k][i]=0;
                }
            }
        }
    }

    for(k=0;k<SIZE;k++)
    {
        int x=(k%MINIGRIDSIZE)*MINIGRIDSIZE, y=(k/MINIGRIDSIZE)*MINIGRIDSIZE;
        for(i=0;i<SIZE;i++)
        {
            for(j=i+1;j<SIZE;j++)
            {
                int l1,l2,cnt=0,first=0,second=0,positionx[2],positiony[2];
                for(l1=x;l1<x+MINIGRIDSIZE;l1++)
                    for(l2=y;l2<y+MINIGRIDSIZE;l2++)
                    {
                        if(forward_map[l1][l2][i]==0){
                        if(forward_map[l1][l2][j]==0){
                            first|=1<<(l1*MINIGRIDSIZE+l2);
                            if(cnt<=1)
                            {
                                positionx[cnt]=l1;
                                positiony[cnt]=l2;
                            }
                            else
                            {
                                cnt=0;
                                break;
                            }
                            cnt++;

                        }
                        if(forward_map[l1][l2][j]==0)
                            second|=1<<(l1*MINIGRIDSIZE+l2);
                    }
                if(cnt==2 && second==first)
                {
                    for(l1=0;l1<SIZE;l1++)
                    {
                        forward_map[positionx[0]][positiony[0]][l1]=1;
                        forward_map[positionx[1]][positiony[1]][l1]=1;
                    }
                    forward_map[positionx[0]][positiony[0]][j]=0;
                    forward_map[positionx[0]][positiony[0]][i]=0;
                    forward_map[positionx[1]][positiony[1]][j]=0;
                    forward_map[positionx[1]][positiony[1]][i]=0;
                }
            }
        }
    }
}
final_board=inp;

#pragma omp parallel for
for(i=0; i<SIZE*SIZE; i++)
{
    int x=i%SIZE, y=i/SIZE;
    if(inp[x][y]==0)
    {
        int ind;
            #pragma omp critical
            ind=empty_cells++;
```

```
            empty_cells_list[ind].y=y;
        }
    }
    struct queue *q = malloc(sizeof(struct queue));
    int num_threads;
    int idx=0;
    #pragma omp parallel shared(q)
    {

        int k;
        #pragma omp single
        {
            num_threads = omp_get_num_threads();
            init_queue(q, num_threads*SIZE);
            struct queue_element *elem;
            elem = malloc(sizeof(struct queue_element));
            for(i=0;i<SIZE;i++)
                for(j=0;j<SIZE;j++)
                    elem->board[i][j]=inp[i][j];

            for(i=0;i<SIZE;i++)
                for(j=0;j<SIZE;j++)
                    for(k=0;k<SIZE;k++)
                        elem->forward_map[i][j][k]=forward_map[i][j][k];

            push(q,elem);
            while(q->length < num_threads && idx<empty_cells)
            {
                int l=q->length;
                int i;
                struct queue *tmp=malloc(sizeof(struct queue));
                init_queue(tmp,l*SIZE);
                for(i=0;i<l;i++)
                {
                    int j;
                    for(j=0;j<SIZE;j++)
                    {
                        if(forward_map[empty_cells_list[idx].x][empty_cells_list[idx].y][j]==0)
                        {
                            int a,b,c;
                            struct queue_element * temp=malloc(sizeof(struct queue_element));
                            for(a=0;a<SIZE;a++)
                                for(b=0;b<SIZE;b++)
                                    temp->board[a][b]=q->list[i]->board[a][b];


                            for(a=0;a<SIZE;a++)
                                for(b=0;b<SIZE;b++)
                                    for(c=0;c<SIZE;c++)
                                        temp->forward_map[a][b][c]=q->list[i]->forward_map[a][b][c];

                            temp->board[empty_cells_list[idx].x][empty_cells_list[idx].y]=j+1;
                            populate_f(empty_cells_list[idx].x,empty_cells_list[idx].y,j+1,temp->forward_map);
                            push(tmp,temp);
                        }
                    }
                }
                idx++;
                free(q);
                q=tmp;
            }

        }
        #pragma omp for schedule(dynamic,1)
        for(i=0; i<q->length;i++)
        {
            dfs(q->list[i]->board,q->list[i]->forward_map,idx);
        }
    }
    return final_board;
}
```

5.4 TEST CASES

5 test cases from each difficulty for grid sizes 16x16 and 25x25 were used to obtain average execution time for each difficulty. A sample of the test cases is given below.

Testcase no.	Difficulty	Test case value
1	Easiest	0 2 6 0 0 0 8 1 0 3 0 0 7 0 8 0 0 6 4 0 0 0 5 0 0 0 7 0 5 0 1 0 7 0 9 0 0 0 3 9 0 5 1 0 0 0 4 0 3 0 2 0 5 0 1 0 0 0 3 0 0 0 2 5 0 0 2 0 4 0 0 9 0 3 8 0 0 0 4 6 0
2	Easy	0 0 0 0 9 0 0 8 0 0 0 4 0 0 0 7 0 0 5 0 6 7 0 0 0 0 0 0 0 0 0 0 0 0 6 7 1 4 0 0 0 6 9 0 0 0 0 9 3 0 0 0 4 0 0 0 0 0 0 5 4 0 8 0 0 0 0 0 1 0 0 5 8 1 0 0 2 0 0 0 0
3	Medium	0 0 0 0 0 0 6 8 0 0 0 0 0 7 3 0 0 9 3 0 9 0 0 0 0 4 5 4 9 0 0 0 0 0 0 0 8 0 3 0 5 0 9 0 2 0 0 0 0 0 0 0 3 6 9 6 0 0 0 0 3 0 8 7 0 0 6 8 0 0 0 0 0 2 8 0 0 0 0 0 0
4	Hard	0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 8 5 0 0 1 0 2 0 0 0 0 0 0 0 5 0 7 0 0 0 0 0 4 0 0 0 1 0 0 0 9 0 0 0 0 0 0 0 5 0 0 0 0 0 0 7 3 0 0 2 0 1 0 0 0 0 0 0 0 0 4 0 0 0 9

Table 5.3.1 Sample test cases

6. OUTPUT AND PERFORMANCE ANALYSIS

6.1 EXECUTION SNAPSHOTS

```
forkbomb@nikhitha:~/Documents/PDC Project$ gcc -o output main_serial.c
forkbomb@nikhitha:~/Documents/PDC Project$ ./output
Enter path to the input file relative to ~/Document/PDC Project: ./xyz.txt
INPUT GRID
0 2 6 0 0 0 8 1 0
3 0 0 7 0 8 0 0 6
4 0 0 0 5 0 0 0 7
0 5 0 1 0 7 0 9 0
0 0 3 9 0 5 1 0 0
0 4 0 3 0 2 0 5 0
1 0 0 0 3 0 0 0 2
5 0 0 2 0 4 0 0 9
0 3 8 0 0 0 4 6 0


OUTPUT GRID
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

Solution Found
Elapsed Time: 0.000099 sec
```

Fig 6.1.1 Serial Execution of 9X9 grid

```
forkbomb@nikhitha:~/Documents/PDC Project$ gcc -o output main_serial.c
forkbomb@nikhitha:~/Documents/PDC Project$ ./output
Enter path to the input file relative to ~/Document/PDC Project: ./pqr.txt
INPUT GRID
8 7 0 0 0 0 0 0 0 3 0 0 13 0 4 0
0 5 14 0 0 0 3 10 15 9 1 0 0 6 0 0
16 0 0 0 5 8 7 0 0 14 0 0 9 0 11 12
0 0 4 0 0 14 6 13 0 11 10 12 0 7 0 3
14 0 0 8 0 0 1 0 0 0 0 3 7 4 12 0
9 0 0 0 0 6 15 12 0 0 13 14 0 3 1 0
11 0 10 3 0 0 13 0 0 8 0 1 0 0 6 0
6 0 0 1 14 0 4 0 0 5 0 9 11 0 0 13
0 0 0 0 15 0 0 0 0 0 9 0 5 0 2 10
10 1 0 0 6 0 5 0 13 15 7 16 0 0 0 0
0 0 16 11 0 4 0 8 2 0 0 0 0 13 0 7
0 9 0 7 1 3 0 2 6 0 8 10 16 15 14 4
7 0 13 0 9 16 0 5 0 0 14 4 3 8 0 2
0 0 3 0 10 0 0 0 0 0 0 0 0 16 15 0
1 0 9 0 0 0 14 4 0 0 0 0 0 0 7 0
0 6 8 0 3 0 0 0 10 7 0 0 0 0 0 0


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

Solution Found
Elapsed Time: 0.001360 sec
```

Fig 6.1.2 Serial Execution of 16X16 grid

```
navya@ubuntu: ~/Desktop/PDC
File  Edit  View  Search  Terminal  Help
navya@ubuntu:~/Desktop/PDC$ gcc driver.c -fopenmp -o driver
navya@ubuntu:~/Desktop/PDC$ ./driver
Enter path to the input file relative to ~/Desktop/PDC: easiest.txt
Enter number of threads: 10

INPUT GRID
0 2 6 0 0 0 8 1 0
3 0 0 7 0 8 0 0 6
4 0 0 0 5 0 0 0 7
0 5 0 1 0 7 0 9 0
0 0 3 9 0 5 1 0 0
0 4 0 3 0 2 0 5 0
1 0 0 0 3 0 0 0 2
5 0 0 2 0 4 0 0 9
0 3 8 0 0 0 4 6 0


OUTPUT GRID
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

SOLUTION FOUND
Elapsed Time: 0.002737 sec
navya@ubuntu:~/Desktop/PDC$
```

Fig 6.1.3: Parallel execution of easiest test case

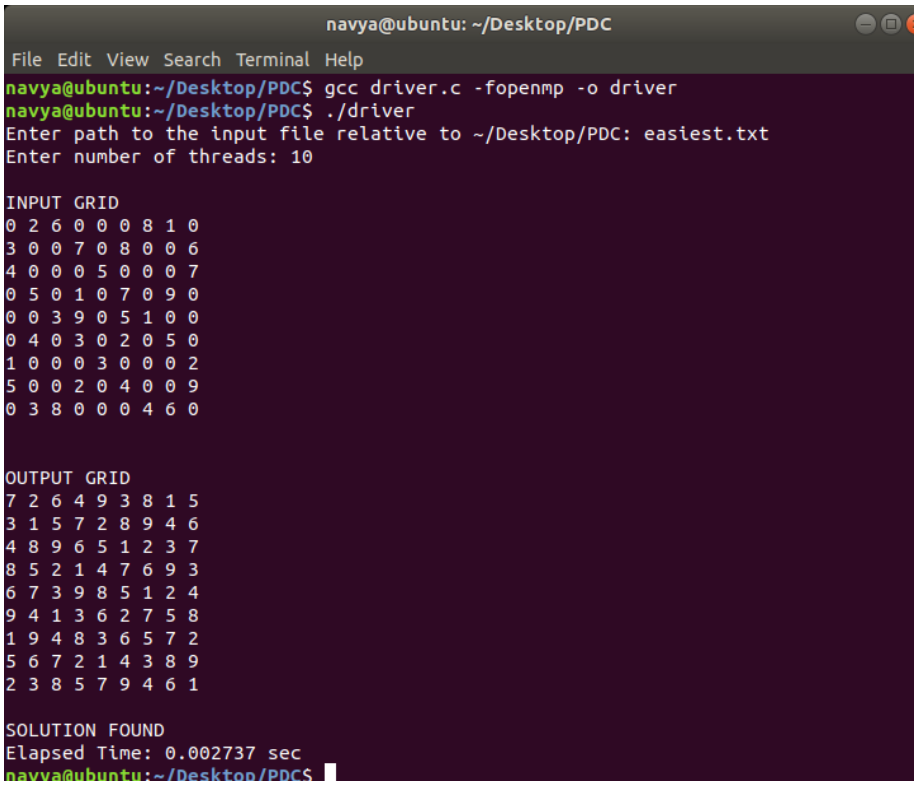

Fig 6.1.4 Paralell 9X9 Grid Easy case

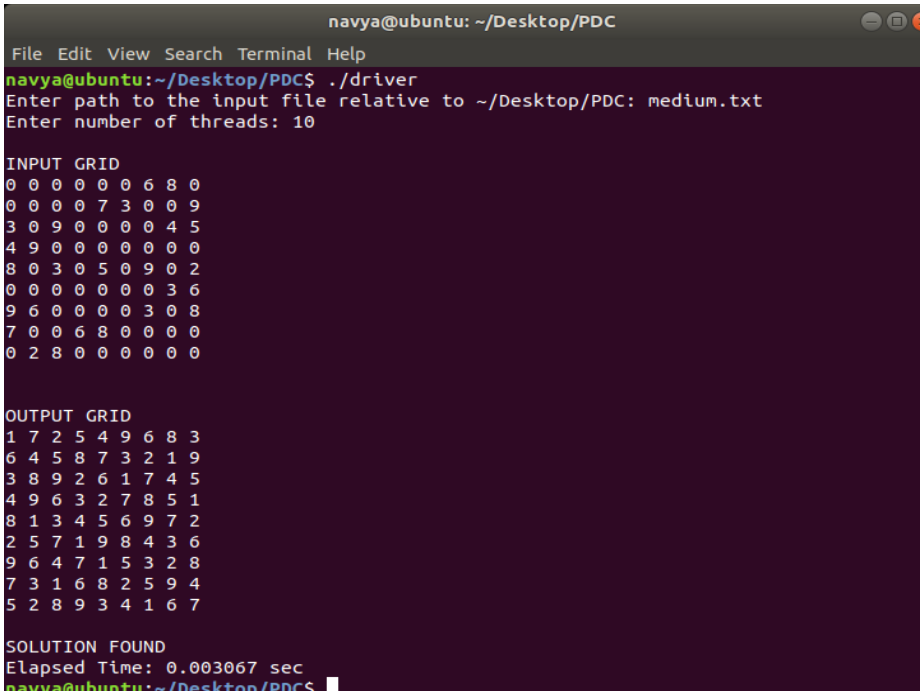

Fig 6.1.5: Parallel execution of medium test case

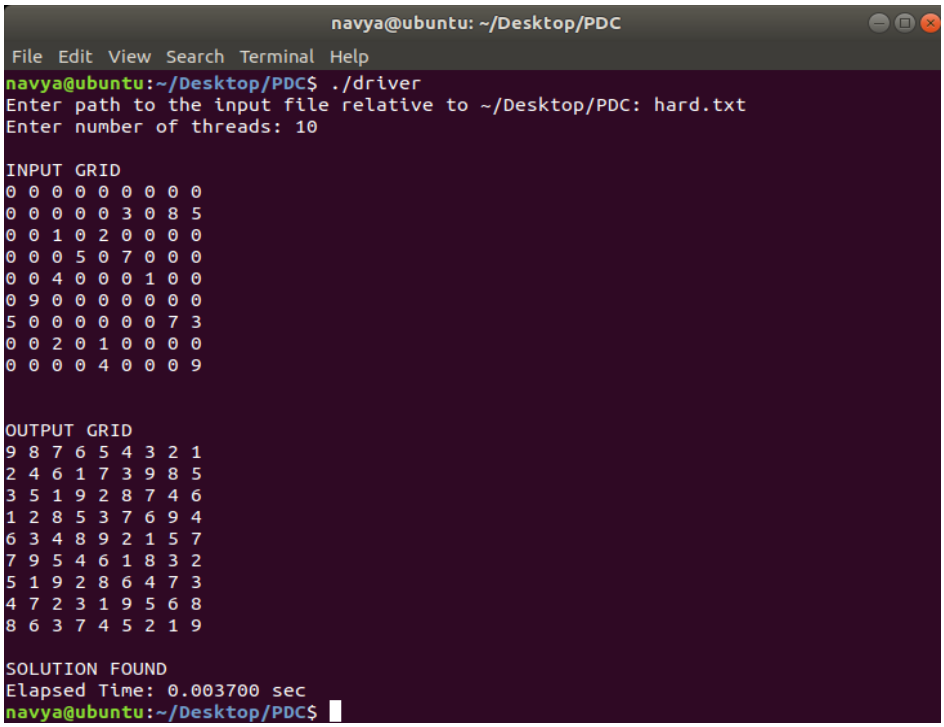

Fig 6.1.6 Parallel execution for 9x9 Hard case

6.2 Results and Discussions

6.2.1 OUTPUT IN TERMS OF PERFORMANCE METRICS

Size	Difficulty	Serial execution time (ms)	Parallel execution time (ms)
16x16	Easiest	0.00352	0.002650
	Easy	0.00407	0.00399
	Medium	0.39592	0.02401
	Hard	0.0677	0.02958
25x25	Easiest	3.582398	3.39847
	Easy	20.22872	9.28479
	Medium	Infinity	64.40987
	Hard	Infinity	160.22982

Table 6.2: Execution times of different sudoku puzzles

The above table of execution times was calculated by averaging the execution times of 5 sudoku puzzles of each category. This was done because of the varying times for a single sudoku puzzle itself. As shown in the table and output screenshots above, it can be easily observed that the execution times of smaller grids is much faster serially than parallelly. But for grids of higher dimensions, the serial code does not arrive at a solution whereas the parallel code does.

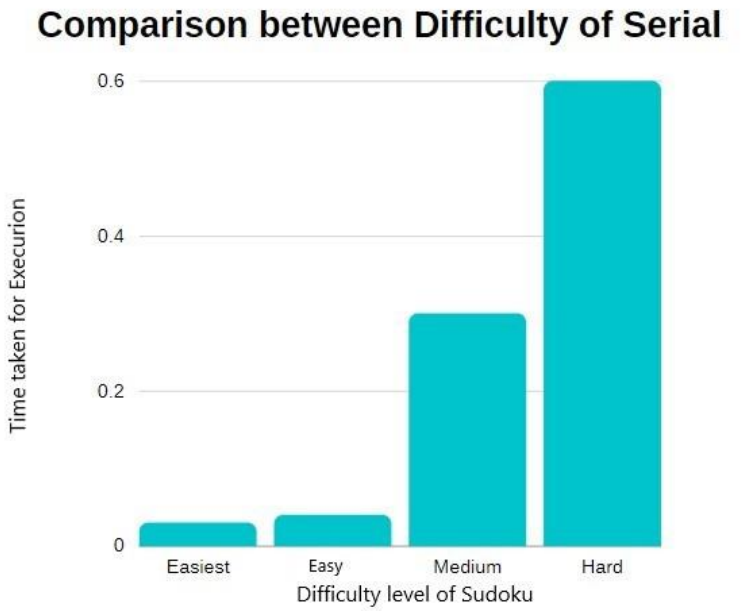

Fig 6.2.1 Comparison between Difficulty of Serial Approach

Here we have visualized the time taken by our serial code to find the solution of sudokus of different difficulty levels. We can clearly interpret that as the difficulty of our sudoku increases, the time taken by our code increases exponentially. For instance, the time taken to solve the easiest of all the sudokus is in a minimal range of 0.002ms and that of hardest has a large solve time of 0.6ms.

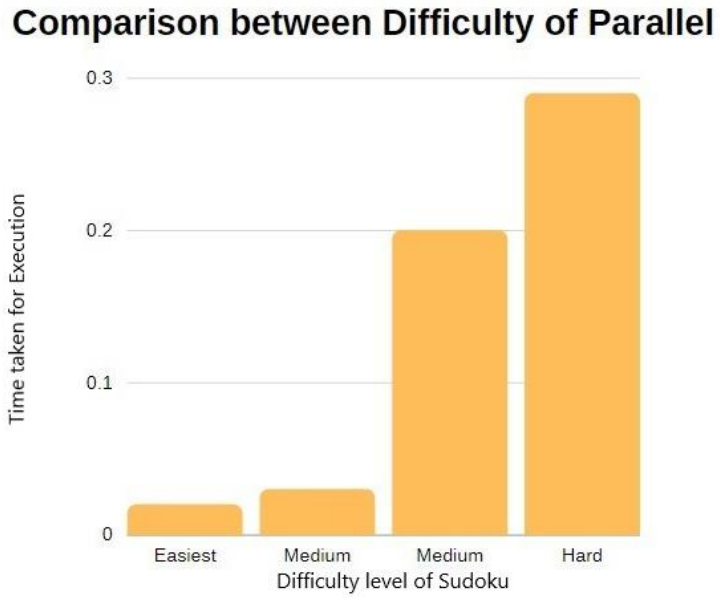

Fig 6.2.2 Comparison between Difficulty of Parallel Approach

Here we have visualized the time taken by our parallel code to find the solution of sudokus of different difficulty levels. We can clearly interpret that as the difficulty of our sudoku increases, the time taken by our code increases exponentially. For instance, the time taken to solve the easiest of all the sudokus is in a minimal range of 0.0004ms and that of hardest has a large solve time of 0.3ms.

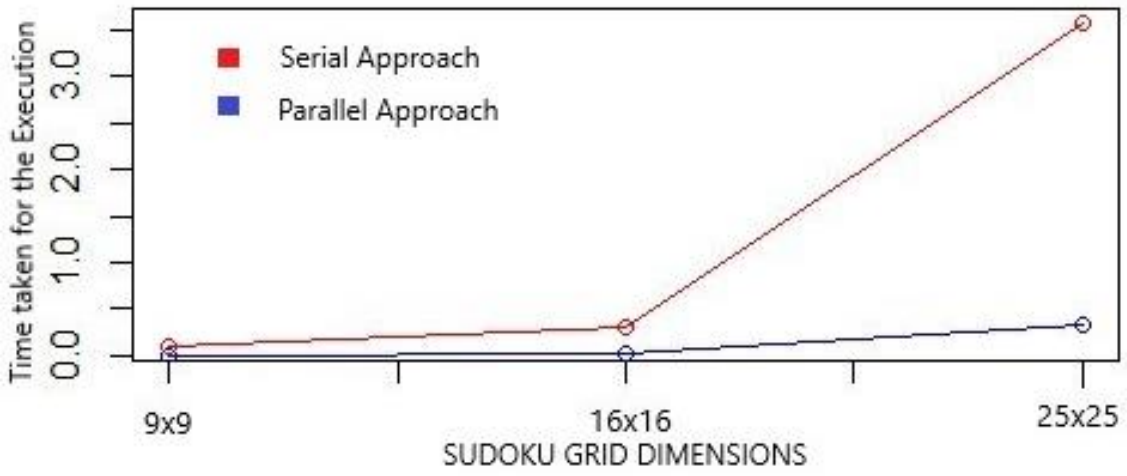

Fig 6.2.3 Comparison between Serial and Parallel Codes

Here we have compared the run time of our code with the increase in grid size of our sudoku. We have also compared the time it takes to solve these sudokus using serial and parallel code. Coming upon the interpretation part of the above graph, the parallel code is able to find the solution of all the sizes of sudokus with fairly equal time. The time limit ranges from 0.04ms to 0.3ms. But when it comes to serial sudoku, the time it takes to solve our sudokus increases exponentially with the size of the input grid. The time it takes to solve a 9X9 grid starts at a bare minimum of 0.2ms but as the size increases to 25X25, the timing increases to 3ms for the same.
All these interpretations indicate that our parallel code is far better than the execution of serial code

7. CONCLUSION AND FUTURE DIRECTIONS

Sudoku is one of the most popular number puzzles and solving it can be really challenging. Our aim in this project was to solve sudoku using parallel programming to improve the execution time. But the results observed shows that for sudoku puzzles of smaller dimensions, the serial code runs faster than the parallel code. This is because of the use of depth first search which is difficult to parallelize. Another observation was that for sudoku puzzles of large dimensions such as 25x25, the serial code runs for a very long time which can almost be considered as a never-ending program. Whereas the parallel code arrived at a solution. Improvement of parallelization of DFS could be a promising direction of research which could not only improve the execution time of sudoku solver but also all other applications involving the use of DFS.

REFERENCES

[1] Saxena, R., Jain, M. and Yaqub, S.M., 2018. Sudoku Game Solving Approach Through Parallel Processing. In Proceedings of the Second International Conference on Computational Intelligence and Informatics (Springer), Singapore.

[2] Satish, N., Kim, C.,Chhugani, J.,Saito, H. (2012) Can traditional programming bridge the Ninja performance gap for parallel computing applications? (IEEE)

[3] Shawn Lee, Efficient Parallel Sudoku Solver via Thread Management & Data Sharing Methods, Department of Computer Science & Engineering, University of California, Riverside

[4] Chatterjee, Sankhadeep, Saubhik Paladhi, and Raktim Chakraborty.(2016) "A Comparative Study On The Performance Characteristics Of Sudoku Solving Algorithms." IOSR Journals (IOSR Journal of Computer Engineering)

[5] Abdulaziz Aljohani, William Smith (2015). N×N Sudoku Solver Sequential and Parallel Computing

[6]Karimi-Dehkordi, Z., Zamanifar, K., Baraani-Dastjerdi, A. and Ghasem-Aghaee, N., 2010, June. Sudoku using parallel simulated annealing. In International Conference in Swarm Intelligence (pp. 461-467). Springer, Berlin, Heidelberg.

[7]Sato, Y., Hasegawa, N., & Sato, M. (2011, June). GPU acceleration for Sudoku solution with genetic operations. In 2011 IEEE Congress of Evolutionary Computation (CEC) (pp. 296-303). IEEE.

[8] Cazenave, T. and Labo, I.A., 2006. A search based sudoku solver. Labo IA Dept. Informatique Universite Paris, 8, p.93526.

[9] Yue, T.W. and Lee, Z.C., 2006, August. Sudoku solver by q'tron neural networks. In International Conference on Intelligent Computing (pp. 943-952). Springer, Berlin, Heidelberg.

[10] Weber, T., 2005, December. A SAT-based Sudoku solver. In LPAR-12, Short paper proc.