

Project Report

Sudoku Solver using

Exploratory

Decomposition

Group Members:

Wahaj Javed Alam (20K-0208)

Hatif Mujahid (20K-0218)

2nd December, 2022

Introduction

Sudoku is a logic-based number-placement puzzle game where the player's goal is to complete a $N \times N$ table such that each row, column and box contains every number in the set $\{1, \dots, n\}$ exactly once.

Inspiration

A sudoku board can be solved by placing a number in an empty space and then checking if the board is in the correct state, if yes then proceed to find the next empty space and place another number else backtrack to the first one and change it. The process is really time consuming and sometimes becomes impossible to achieve since considering a 9×9 board having around 20 empty spaces, the game tree can have 20^9 possible boards in the [game tree](#) which would be space and time expensive to explore. So, in order to utilize the concepts of Parallel Computing and demonstrate our understanding of the course we took a [research paper](#) written over a parallel solution of a sudoku board and converted the mentioned algorithm into our own code (till the extent of our knowledge).

Algorithm


We have utilized OpenMP in order to overcome the task.

Concepts:

There are two main concepts behind our algorithm.

1. Constraint Propagation

It is the very first step towards solving the problem. The main idea is that if a number exists in a cell's row, column or box then the possibility of that number appearing in that specific cell is removed. And for a cell if all of its peers have a specific number removed, the cell itself must contain the number. Visually, it can be shown as:



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

Considering the following list of possibilities for a row of 9 x 9 Sudoku Board, the 5th cell can only have 9 since all other numbers are removed from its possibility list because its peers have it, Hence this is Elimination.

2. Search

Once the constraint propagation (elimination) has been performed, the program rechecks if there are any singletons (cells following constraint propagation), if yes then elimination is applied again, else new boards are generated with all possible values for a cell and constraint propagation is applied again.

The Program Execution is applied as:

CP() -> Search() -> CP() -> Search -> ..

Logic:

With the theoretical concepts put aside, let's move on to the actual code and how it is executed step by step.

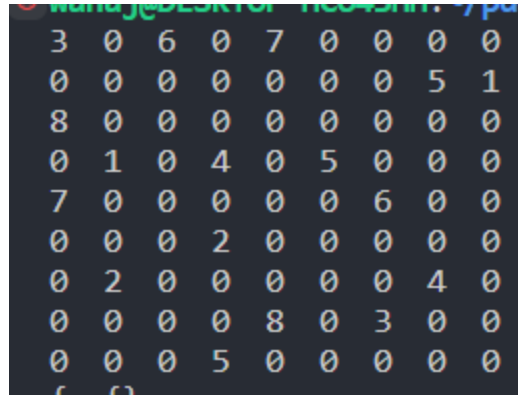
We have four types of matrices:

1. **Sudoku Matrix** (the board itself)
2. **Possibility Matrix** (a list of all possible numbers for each cell generated by performing Constraint Propagation)
3. **Indexes Matrix** (a matrix that denotes how many numbers can be placed in each cell)
4. **Solution Grid** (a matrix that contains the result of the calculations).

Now, getting to the real code,

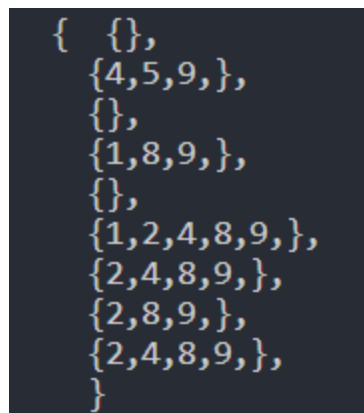
The function solveSudoku receives the sudoku board, the possibility and the indexes matrix, the **first step** is to verify if the board is correct or not, if not then the **solution is not possible**. Now, if the solution is considerably possible, a list of all possibilities is generated by constraint propagation(elimination) and stored in the possibility matrix.

For the Matrix mentioned below,



3	0	6	0	7	0	0	0	0
0	0	0	0	0	0	0	5	1
8	0	0	0	0	0	0	0	0
0	1	0	4	0	5	0	0	0
7	0	0	0	0	0	6	0	0
0	0	0	2	0	0	0	0	0
0	2	0	0	0	0	0	4	0
0	0	0	0	8	0	3	0	0
0	0	0	5	0	0	0	0	0

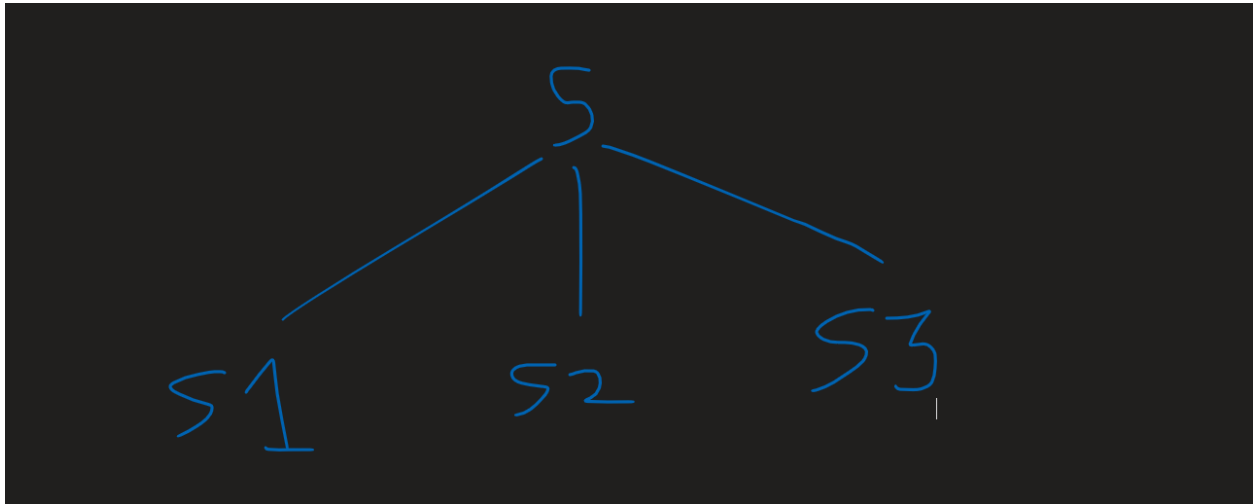
The possibility matrix generated for the first row is mentioned below,



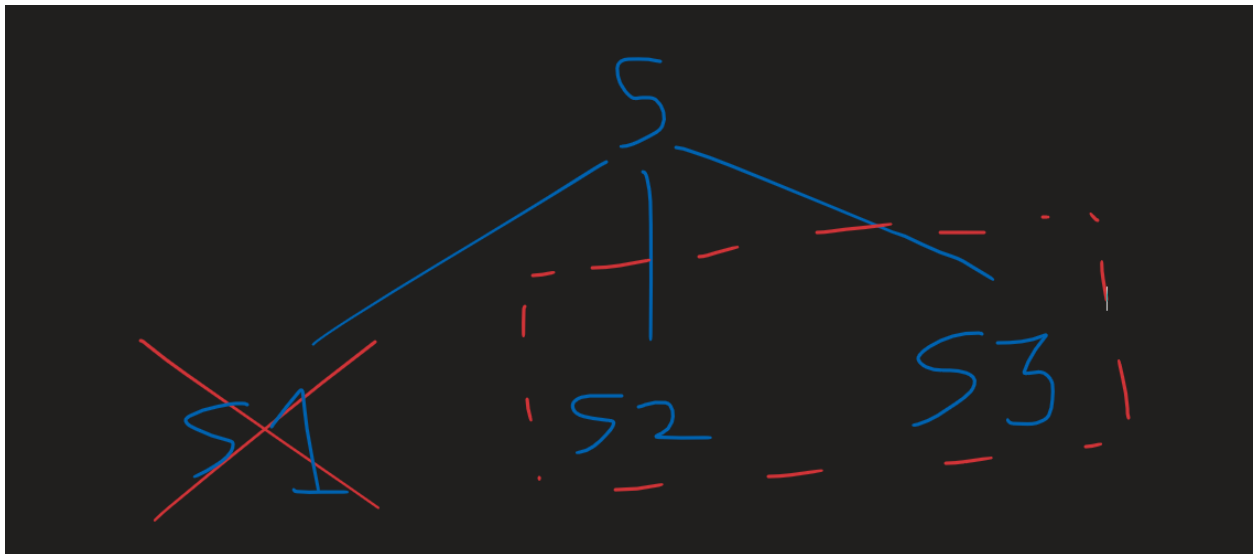
```
{ {},
  {4,5,9,},
  {},
  {1,8,9,},
  {},
  {1,2,4,8,9,},
  {2,4,8,9,},
  {2,8,9,},
  {2,4,8,9,},
}
```

The reason we are creating this possibility matrix is to ensure that we do not go on checking every number since it would be a waste of calculations.

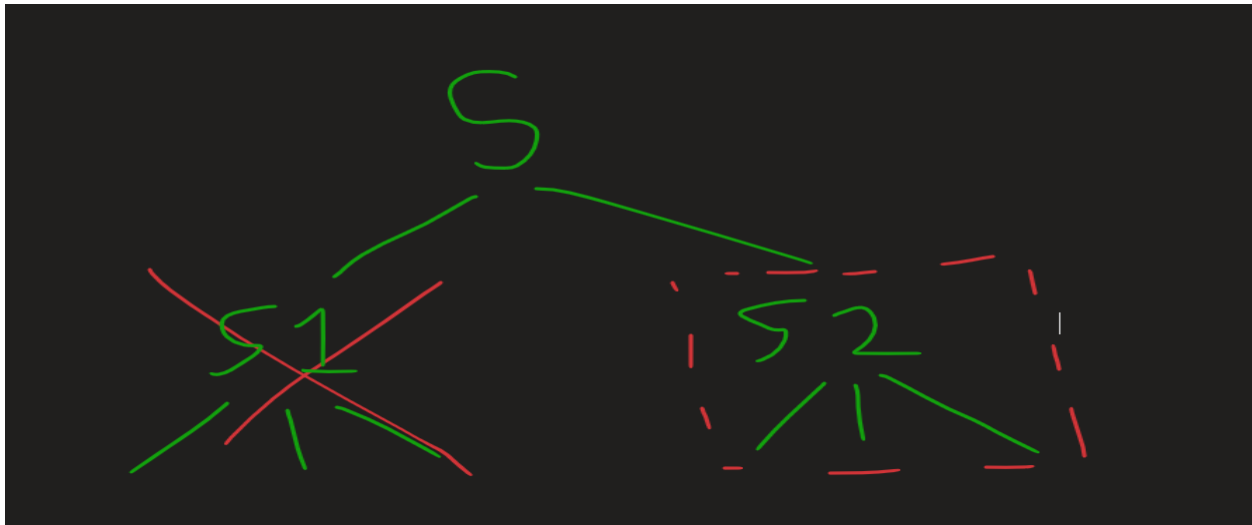
Moving on, the next step is to **find a cell with minimum count of possibilities** since it can help in a significant number of calculation reductions, consider a 9x9 board where one of the cells has 3 possibilities for a number the game tree would like this:



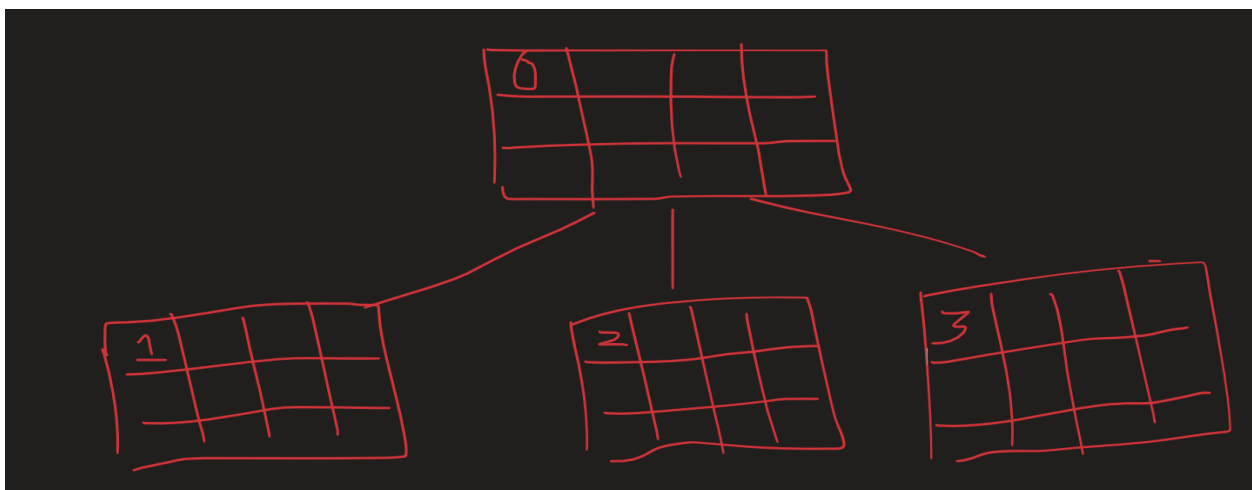
Now, if we were to eliminate S1, we would still have **2/3rd of the game tree left to explore** shown as:



Consequently, the reason to choose a cell with minimum possibilities is now clear, consider the same 9x9 board but this time a cell has 2 possibilities, **if we reject one of them, we ignore half of the game tree** which is significant when it comes to calculations.



Once the minimum possibility row and column are found, we move on to creating multiple boards. The way this is performed is that, we **create a number of new boards equal to the number of possibilities** with **each possibility placed in that board** and **pass each board to a thread** so that each thread calculates a new possibility and indexes matrix for its own board and recursively calls the function to **apply the exploratory decomposition on the newly generated boards until a solution is found**. Consider the 9x9 board again where one cell row number 1 and column number 1 can have three numbers placed (1,2,3). Now the grid handler creates new boards by placing each number into a new board like:



Now Each of these boards are placed onto a stack, which are then popped by threads in a critical section and then the process goes on. **If a board cannot be solved, it is simply deallocated.** The Recursive calls go on till a solution is found. Then, the recorded time to solve is displayed and the log files are updated with the total number of boards explored and the number of times the recursion was done.

The **Result** looks like this:

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

And the **Log File** is written as:

```
=====
EXPLORATORY DECOMPOSITION OF A 9x9 SUDOKU BOARD
=====
Requested Thread Count: 10
INPUT BOARD :
0 9 2 3 0 0 0 0 0
0 0 0 0 8 0 1 0 0
0 0 0 0 0 0 0 0 0
1 0 7 0 4 0 0 0 0
0 0 0 0 0 0 0 6 5
8 0 0 0 0 0 0 0 0
0 6 0 5 0 2 0 0 0
4 0 0 0 0 0 7 0 0
0 0 0 9 0 0 0 0 0
Total Boards Created : 6861791
Total Recursive Calls : 3332454
Output Board:
7 9 2 3 5 1 6 4 8
5 4 3 7 8 6 1 2 9
6 8 1 4 2 9 5 3 7
1 5 7 6 4 8 2 9 3
9 2 4 1 3 7 8 6 5
8 3 6 2 9 5 4 7 1
3 6 8 5 7 2 9 1 4
4 1 9 8 6 3 7 5 2
2 7 5 9 1 4 3 8 6
Total parallel time taken to solve a 9x9 Sudoku Board : 431.306
=====
```

Analysis

The Parallelized and Serial Code all included in the files were run on Core i7 @4.8 Ghz 10th Generation Processor. The CPU had 16 GB of RAM, 8 Cores and 16 Threads by default. The L1 cache of the system is 256KB in each CPU, L2 cache is 2MB and L3 is 16 MBs.

The code has some optimal cases and worst cases, the optimal cases just being a board which can be solved with simple elimination in the case where no recursion is done which can be seen below:

```
=====
EXPLORATORY DECOMPOSITION OF A 9x9 SUDOKU BOARD
=====
Requested Thread Count: 1
INPUT BOARD :
1 0 6 0 0 2 3 0 0
0 5 0 0 0 6 0 9 1
0 0 9 5 0 1 4 6 2
0 3 7 9 0 5 0 0 0
5 8 1 0 2 7 9 0 0
0 0 0 4 0 8 1 5 7
0 0 0 2 6 0 5 4 0
0 0 4 1 5 0 6 0 9
9 0 0 0 0 4 0 1 0
Total Boards Created : 0
Total Recursive Calls : 0
Output Board:
1 4 6 7 9 2 3 8 5
2 5 8 3 4 6 7 9 1
3 7 9 5 8 1 4 6 2
4 3 7 9 1 5 8 2 6
5 8 1 6 2 7 9 3 4
6 9 2 4 3 8 1 5 7
7 1 3 2 6 9 5 4 8
8 2 4 1 5 3 6 7 9
9 6 5 8 7 4 2 1 3
Total parallel time taken to solve a 9x9 Sudoku Board : 0.131506
=====
```

The worst case scenario is when there is a huge board and the number of zeros to be filled is too much, in such case, the code is not crashing neither is it facing any kind of memory leaks, it just needs time to execute we ran a 25x25 scenario for around 8 hours but the code was continuously running , it might have needed 2 more hours

approximately but our university time was over so we could not see it. Although, this is confirmed that the code is not crashing even at such bigger memory allocations.

Finally, compare it by writing a serial version of the same code.

1. For the 16 x 16 Board,

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

The parallel code reached the solution in **195.629 seconds** and explored **390819 boards** whereas the serial version took **390 seconds** and explored **390725 boards** to achieve the same results.

2. For the 9 x 9 board,

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

The parallel code reached the solution in **431.306 seconds** and explored **6861791 boards** whereas the serial version took **1029 seconds** while exploring **2393899 boards** to solve the problem.



Conclusion

The Speedup achieved by writing the parallel code can be clearly seen in the above mentioned results and the logs of the program. There is some difference when it comes to the number of boards explored by the serial and parallel versions of the board and this is because there are some chances that the right boards are scheduled earlier in case of serial code but still the calculation overhead is greater where as the parallel version explores more boards since the exploration is done in parallel and to the extent where a lot of boards are considered and neglected in a lesser amount of time with increased throughput. The number of threads only matter in the parent recursive call since all the upcoming recursive calls are scheduled by openMP even when `OMP_NESTED` is set to true, so the number of threads help with the searching/verifying/eliminating at a great rate but do not affect the overall performance greatly. For that, we would have to implement message passing which is beyond the scope of this course.