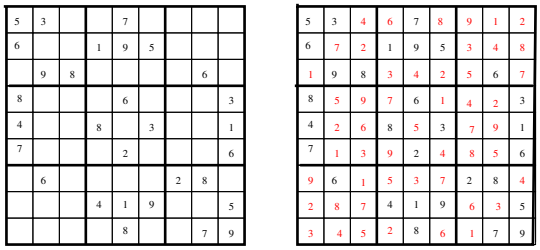
COL 380	March 28, 2016
Lab 2	
Instructor: Subodh Sharma	**Due: April 8, 23:55 hrs**

NOTE:
a) You can either choose to perform the assignment in group of size 2 or choose to perform it alone.
b) There are two deadlines for this project. First deadline is by April 5, 23:55 hrs. By this deadline you should submit a working *parallel* version of the Sudoku solver. Only those submissions that are fulfill this deadline will be allowed to compete in the next stage.
c) The submissions in the next stage are supposed to make their code more performant. Resubmissions are allowed until April 8, 23:55 hrs. Grading will be performed according to the standing of the submission in the leaderboard.
d) Submissions that did not make it to the first deadline will be allowed to submit until April 8. 23:55 hrs but will not be allowed to compete. The performance portion of the grading for such submissions will be 5 marks less than the bottom of the leaderboard.

1 Sudoku Puzzle

Sudoku is a number placement puzzle wherein given a NxN grid the objective is to fill the grid with digits from 1 to N such that each row, each column and each of the $\sqrt{N} \times \sqrt{N}$ sub-grid (assuming N is a perfect square) contains all the digits from 1 to N. The puzzle is posed with a partially completed grid. The solution is not always guaranteed and when the solution exists it may not be unique. However, for a well posed puzzle there is a unique solution. Consider a 9x9 partially filled Sudoku and its solution shown below:



1.1 Statement of work

Your task is to create a parallel Sudoku solver using OpenMP. Note that Solving Sudoku is shown to be an NP-complete problem; thus, there aren't any serial algorithms that can solve sudoku in polynomial time. However, relying only on brute-force algorithms is also not advised. There are many heuristics that exist that people use to speed-up Sudoku solving. We provide you some heuristics that can come handy; you are free to device your own heuristics. Additionally, you can add, modify, or improvise the heuristics provided to you.

- **Elimination**: This occurs when there is only one valid value for a cell. Therefore, that value must belong to that cell. In the figure below, assuming $2, 5$ were provided as a part of partially complete grid. Then by identifying the potential values that the rest of the cells can take, we observe that the cell containing 6 has only one option and therefore it must be the solution for that cell.

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

- **Lone ranger**: There is a single value that is valid for a cell in a row, column, sub-grid despite the fact that there are other values that can be possible solutions for that cell. This is because that single value is not present as a solution for any other row, column or sub-grid. Consider the example below. Here the yellow cell has $5, 6, 7$ as solution values but note that 7 is unique to this cell as a possible solution for a row, column or sub-grid. Thus, 7 is assigned to that cell.

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

- **Twins and Triplets**: Twins are pair of values that appear together in two separate cells and in only those two cells. When twins are found the other values in those cells can be removed. Consider the figure shown below. Here in the yellow cells (and in no other cell) $5, 7$ are appearing together. Thus, 3 and 6 can be dropped from those cells as possible solutions.

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

  For triplets, much like twins, three values should appear together in only three separate cells.

Note that these heuristics can help in discovering the solution quickly but offer no guarantee that the solution would be found when a solution does exist. Thus, in such circumstances, one may have to eventually resort to the brute-force technique.

1.2 Input, Compilation and Output

Compilation of the programs will be performed under `-O3`. Tools such as Valgrind, DDD, or GDB should be used to debug the program for data races and memory leaks. One can also use GNU Gprof to identify code regions for optimization that take lot of time to execute. The output of the program is expected to be in the following format:

```
2 3 1 4
1 4 2 3
3 2 4 1
4 1 3 2

SOLUTION FOUND
TIME = 0.2 sec
```

That is the solution grid in a space separated values of cells. If no solution is found then print:

```
NO SOLUTION FOUND
TIME = 0.2 sec
```

Input to the progam will be supplied in the form of a partially filled grid in a file. Empty cells in that grid will have 0 as the value.

1.3 Grading

Your parallel solution would be checked for correctness and performance. For performance, we will create a leader-board and the grading would be performed according to the rank of the program on the leader-board. Additionally, a lab report (as a text file) is expected where you clearly mention your design of algorithm and data/task distribution among threads. You are expected to clearly indicate the rationale behind your decisions for performance. Make sure the lab report does not exceed 400 words and text should be supplied in short bullet-ed points.

- Correctness: 40 marks
- Performance: 40 marks
- Lab report: 20 marks

1.4 Files Provided

- main.c – Implement the function solveSudoku(). Keep the code modular. Don't touch any other code of main.c
- sudoku.h – header file