UNIVERSITY OF TRENTO

PROJECT REPORT

Sudoku Puzzle

Supervisor
Prof. Sandro Luigi Fiore

Wamiq Raza
224824

A report submitted in fulfillment of the requirements
for the course of HPC4DS

in the

High Performance Computing for Data Science
Department of Information Engineering and Computer Science

August 24, 2021

Declaration of Authorship

I, declare that this project titled, “Sudoku Puzzle” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a semester project at this University.
- Where any part of this report has previously been submitted for a course or any other qualification at this University or any other institution, this has been clearly stated.
- Where we, have consulted the published work of others, this is always clearly attributed.
- Where we, have quoted from the work of others, the source is always given. With the exception of such quotations, this report is entirely our own work.
- We have acknowledged all main sources of help.
- Where the report is based on work done by our-self jointly with others team-mate.

Date:

Contents

Chapter 1

Introduction

1.1 Introduction

This document explains what we have done for the project for the course on High Performance Computing for Data Science. The steps that we have taken resemble the outline that we defined on the Project Road-map document, with a few modifications. This time we are going to dive deep on the implementation details in order to explain the choices that we have made and how we decided to implement the Sudoku puzzles solver. The entire code of our project is hosted on GitHub at the following link: https://github.com/razacode/hpc4ds. The code of our project is in the /src folder, and it is thoroughly documented. We generated a PDF with the documentation of the code via LateX which is in the main folder.

Sudoku has become increasingly popular in the past few years. The puzzle appears in newspapers and magazines, in books and scientific papers and there even exist TV Shows about it. There are many reasons for its international popularity. The rules are simple and easy and it is based on numbers instead of words or letters. Nevertheless there are quite complex mathematical ideas behind it and solving such a puzzle can prove quite difficult and time consuming.

1.2 Acknowledgements

In this section we want to make clear what we have reused from other projects and what is original work which we referenced them in their respective sections.

1.3 Background

Sudoku is a deceptively, simple logic puzzles which has captured majority of public interest and is a logic-based combinatorial number placement puzzle. The puzzle likely originated from the 18th-century Swiss mathematician Leonhard Euler's game called "Latin Squares". (The Economist, 2005) It first gained popularity in Japan, which is also where the name 'Sudoku' was given to it.

Consisting of a 9 × 9 grid and further sub-divided into 'mini-grids' of size 3 × 3 with each of the 81 cells of the grid to be filled with the digits 1 to 9. Such that each digit appears exactly once in each row, column and mini-grid as shown in Figure 1.1. Puzzles are sufficiently simple in concept for wide sections of the population to attempt their solution but it still retaining a sufficient challenge for most through the necessity of applying several methods of reasoning. Since the puzzle is based on mathematical logic rather than guessing, a well set puzzle should have only one solution.

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

FIGURE 1.1: Unsolved Sudoku Puzzle .

1.4 Problem Statement

Solving Sudoku has been a challenging problem, in the last decade. The purpose has been to develop more effective algorithm in order to reduce the computing time and utilize lower memory space. This essay develops an algorithm for solving Sudoku puzzle by using a method different method. Many algorithm resembles human methods, i.e. it describes how a person tries to solve the puzzle by using certain techniques.

1.5 Project Proposal and Objectives

The project proposal was to create a tool using openmpi and cluster in which uses propositional logic to solve Sudoku puzzles. The expectation was that since Sudoku is a logic based puzzle it could be broken down into a set of propositional constraints and these constraints could be used to find its solution by utilizing the power of server processors; thus helping players to check their solution to a given Sudoku puzzle. We have created separate c program where user can put the puzzle into the solver as in .txt file extension and that **C** file can read and send to main program for solving.

The aim of this project is it would give the users an opportunity to solve different puzzles within the application and improve their solving skills. From the above proposal the objectives of the project could be described as follows:

1. The game should include of all difficulty puzzles levels and displaying the solution to certain cells or the entire puzzle, as requested by the user.
2. The solution should be presented in a manner that is understandable easily by both new and experienced users.

1.6 Source Code

The source code of the entire project is available at the following URL for GitHub repository. https://github.com/razacode/hpc4ds

1.7 Libraries

In the project we used several libraries that helped us reduce the overall amount of work and complexity.
The following is the list of libraries used for source code 1.6.

1.7.1 argparse

This library provides high-level arguments parsing solutions. A command line arguments parsing library in C inspired by parse-options.c (git) and python's argparse module. Arguments parsing is common task in cli program, but traditional getopt libraries are not easy to use. The program specifies the parameters it requires, and argparse will find out how to parse those from argc and argv. It also creates help and use messages automatically, and gives errors when users provide the program with incorrect arguments. The repository link is given below.
Repository: https://github.com/cofyc/argparse

1.7.2 cJSON

cJSON is an ultralightweight JSON parser written in ANSI C. Its goal is to be the absolute dumbest parser that will get the job done. It consists of a single C file and a single header file. As a library, cJSON exists to remove as much effort as possible while being unobtrusive. This library parses and generates JSON strings. The master node sends a string representation of a JSON simulation read from the simulations.json file to each worker node. The data is then extrapolated from the received string. When the simulation is finished, the resultant data is written to a JSON file.
Repository: https://github.com/DaveGamble/cJSON

1.7.3 omp.h

An OpenMP program is divided into sequential and parallel parts. In general, an OpenMP application begins with a sequential section that configures the environment, initializes variables, and so on. Why OpenMP? More efficient, and lower-level parallel code is possible, however OpenMP hides the low-level details and allows the programmer to describe the parallel code with high-level constructs, which is as simple as it can get.

1.7.4 mpi.h

The Message Passing Interface (MPI) Standard based on the consensus of the MPI Forum. The MPI Forum consists over 40 organizations including vendors, researchers, developers and users. The goal of MPI is to provide a portable, efficient and flexible standard for message passing parallel programming.

Note with linking these libraries to an executable is not only easy, but CMake 1.8 this procedure is much easier, portable and maintainable.

1.8 Build

Cmake, a common de-facto program that simplifies overall operations, was used to generate the executable. CMake is used to establish the project structure and to collect all the information required during the compilation process, such as dependencies/libraries to include and Debug or Release mode, as well as to generate directory hierarchies and a Makefile.

1.8.1 CMake

CMake is cross-platform available as an open software that uses a compiler independent technique to automate build, testing, packaging, and installation of software. CMake is not a build system; rather, it creates build files for other systems. It supports hierarchical structures as well as applications that use multiple libraries. It works with native build environments including Make, Qt Creator, Ninja, Android Studio, Apple's Xcode, and Microsoft Visual Studio. It has few dependencies, just requiring a C++ compiler on its own build system.

1.8.2 Procedure Steps

The building phase is separated by four different steps:

1. Create build directory. In the project root, create a build directory that will be used to store the generated files.
   => **$ mkdir build**

2. Change directory context Change the current working directory to the newly created build directory.
   => **$ cd build**

3. Launch cmake and build the project in Release mode to optimize the executable.
   => **cmake ..**

4. Launch make and debug the program.
   => **make**

1.8.3 Compilation

The compilation phase is very straightforward, since it requires a single command to start it which is (make). To compile the executable GNU Make is used, another standard defacto software to make the overall procedures easier to perform. GNU Make is used to build the final executable running various recipes, generated by CMake, defined in the Makefile 1.8.2. Launching make will run the default recipe, that is defined by default as compiling the executable using the mpicc compiler.

Make is a build automation system that supports executable programs and dependencies from code base by processing Makefiles that define how to generate the target program. Though integrated development platforms and language specific compiler capabilities can also be utilized to manage a build process. Make continues to be extensively used, particularly in Unix and Unix-like operating systems. Make, in addition to constructing programs, may be used to handle any project in which certain files must be automatically updated from others if the others change.

1.9 Execution

To execute, the program mpirun (or equivalent) is required. The program requires at least 2 processes since is based on a the requirement of task to solve the problem. Below is an example on how to test Sudoku executable directly using mpirun.

=> **mpirun -np 4 ./sudoku –input=./inputs/0.txt**
=> **mpirun**, is wrapper script to compile.
=> **-np 4**, mean assigning 4 processor to task where minimum require is 2.
=> **sudoku**, read main program.
=> **–input=./inputs**, take input file from the inputs directory.
=> **0.txt**, is the file that consist un-solved sudoku.

1.9.1 PBS

=> **#!/bin/bash**
=> **#PBS -l select=1:ncpus=4:mem=2gb**
=> **# set max execution time**
=> **#PBS -l walltime=00:00:45**
=> **# imposta la coda di esecuzione**
=> **#PBS -q short_cpuQ**
=> **module load mpich-3.2**
=> **mpirun.actual -n 4 ./sudoku-mpi ./for_sudoku_test.txt**

The program is designed to be executed the code on unitn (Uniersity Cluster) cluster using the PBS scheduler. Portable Batch System (PBS) is the name of the computer software that performs job scheduling. Its primary task is to allocate computational tasks, i.e. batch jobs, among the available computing resources. It is often used in conjunction with UNIX cluster environments. Above is an example on how to test Sudoku executable on the unitn cluster.

Chapter 2

Technical Background

This chapter we provides the technical background required to understand the concepts and development process described in the report.

2.1 Problem Analysis

To solve the puzzle, each row, column and box must contain each of the numbers 1 to 9. Throughout this document we refer to the whole puzzle as the grid, a small 3x3 grid as a box and the cell that contains the number as a square as shown in as shown in Figure 1.1. Rows and columns are referred to with row number first, followed by the column number: 3,4 is row 3, column 4 and so on. Boxes are numbered 19 in reading order i.e. 123, 456, 789.

2.1.1 Making a start

To solve Sudoku puzzles you need to use logic. You have to ask yourself questions like, "if a 1 is in this box, will it go in this column ?" or "if a 9 is already in this row, can a 9 go in this square? " To make a start you have to look at each of the boxes and see which squares are empty, at the same time checking that square's column and row for a missing number.

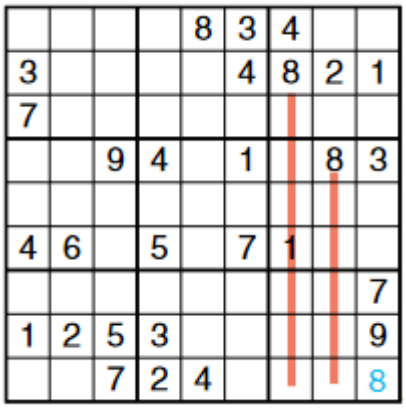

FIGURE 2.1: Sudoku Puzzle solving Explanation.

In Figure 2.1, if we look at box 9. There is no 8 in the box, but there is an 8 in column 7 and in column 8. The only place for an 8, is in column 9, and in this box the only square available is in the row 9. So we can put an 8 in that square. This is how we have solved first number.

2.2 Computational Complexity

From a mathematical perspective, it has been proven that the total number of valid Sudoku grids is 6,670,903,752,021,072,936,960 or approximately 6.671×10^{21}.

Trying to populate all these grids is itself a difficult problem because of the huge number, Assuming each solution takes 1 micro second to be found, then with a simple calculation we can determine that it takes 211,532,970,320.3 years to find all possible solutions.

If that number was small, suppose 1000, it would, be very easy to write a Sudoku solver application that can solve a given puzzle in short time. The program would simply enumerate all the 1000 puzzles and compares the given puzzle with the enumerated ones. Unfortunately, this is not the case here since the actual number of possible valid grids is extremely large so it's not possible to enumerate all the possible solutions. This large number also directly eliminates the possibility of solving the puzzle with brute-force technique in a reasonable amount of time. Therefore, a method for solving the puzzle quickly will be derived that takes advantage of some "logical" properties of Sudoku to reduce the search space and optimize the running time of the algorithm. Our goal off course, is to allow for that algorithm to take advantage of the many-core architecture to further reduce its running time. Because we can extend the 9×9 Sudoku grid to an NxN grid and as you would expect the complexity of the puzzle goes up by a big notch. Hence, the parallel algorithm must also scale as the grid size increases.

2.3 Approaches

Sudoku algorithm describes a few methods, for deterministic-ally solving cells in Sudoku puzzles. Solving Sudoku is proven, to be an NP-complete problem. Their is no serial algorithms, that can solve Sudoku in polynomial time. Usage of humanistic algorithms to fill up as many empty cells as possible but this algorithm doesn't guarantee a solution. If necessary the brute force algorithm solves the remaining of the puzzle. The four of which are used for humanistic solver:

1. Elimination: This occurs when there is only one valid value left for a cell. Therefore, that value must belong in the cell.

2. Lone Ranger: This is a number that is valid for only one cell in a column, row, or box. There may be other numbers that are valid for that particular cell. But since that number cannot go anywhere else in the column, row, or box, it must be the value for that cell.

3. Twins: These are a pair of numbers, that appear together twice in the same two cells and only in those two cells. When twins are found all other numbers can be eliminated as possible values for that cell. The top row depicts a row where the highlighted cells contain twins. The bottom row shows the possible values left for those cells after the twins rule has been applied.

4. Triplets: Triplets follow the same rules as twins, except with three values over three cells. The top row depicts a row of a Sudoku board before triplets is applied. The bottom row is the resulting valid values after triplets has been applied.

Sudoku has a lot of potential for parallelism, as the grid layout allow different regions to be computed independently. However, the brute force method definitely

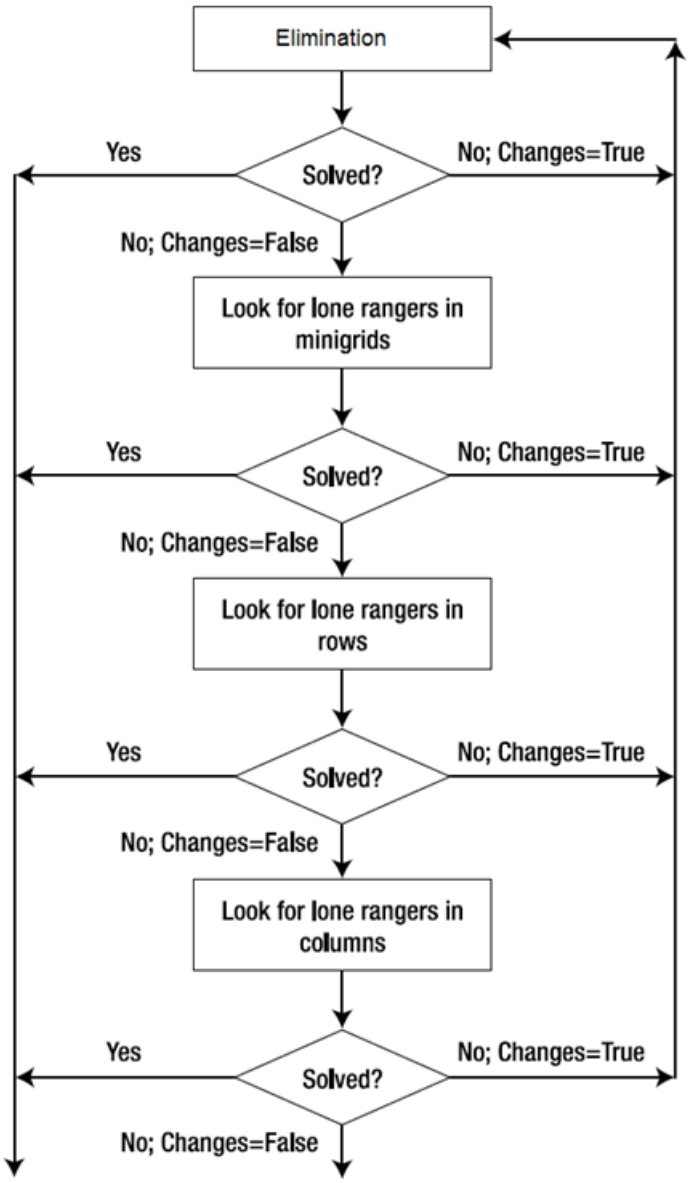

FIGURE 2.2: Flow chart

has more potential for parallelism. Since Sudoku is NP-complete, solving the board using brute force could result in substantial speedup.

In Figure 2.2,is a flow chart to visualize this process solver tries the four methods in the order above. If any method makes changes, then the solver starts over from elimination. This is useful since most changes are made by elimination and lone rangers, and also twins and triplets take more time to find.

Chapter 3

Results

In this chapter about the result and application functionality is discussed.

Here presented a Smart Sudoku Solver that can solve unsolved Sudoku images with small amount of per-spective taken from input. The algorithm can also give solution in cases of severe rotation such as when the Sudoku Puzzle is completely inverted. Since the scale of the problem also varies from input to input, this algorithm efficiently manages these problems. Since the time and memory requirements differs for various algorithms, this algorithm was tested on various difficulty levels figure is attached below, ranging from easy level to hard level. Moreover the different level of Sudoku were given and assign multiple processor for reference is shown in Figure 3.4 and the most efficient one was used to solve the Sudoku puzzle. In general,for the many testing, found our algorithm capable of solving the puzzles efficiently.

Although each traditional Sudoku has only one solution there are several ways to find it. Wikipedia suggests that there are three processes to solve a puzzle. Scanning, marking up and analyzing. These techniques can be in any order and repeated if necessary. Scanning and analysing consist of several different techniques to identify possible solutions, or eliminate invalid solutions, to each field in the grid. They take into account the constraints given and any contingencies that might arise. Marking up is very useful for solving puzzles of higher difficulty. It consists of either making small notes of which numbers could be in the relevant cell of the grid or which numbers cannot be in it and can be done using subscripts or small dots.

3.1 Input

For the input program will take **.txt** extension file as input as shown in Figure 3.1 printed input file in terminal.

```
[wamiq.raza@hpc-head-n2 ~]$ module load mpich-3.2
[wamiq.raza@hpc-head-n2 ~]$ mpicc -std=c11 -pedantic -o sudoku-mpi sudoku-mpi.c
list.h list.c
[wamiq.raza@hpc-head-n2 ~]$ qsub sudoko.sh
7831856.hpc-head-n1.unitn.it
[wamiq.raza@hpc-head-n2 ~]$ qstat 7831856
qstat: 7831856.hpc-head-n1.unitn.it Job has finished, use -x or -H to obtain his
torical job information
[wamiq.raza@hpc-head-n2 ~]$ cat sudoko.sh.o7831856
 0  4  0  0
 1  0  2  0
 0  1  0  2
 0  0  4  0
```

FIGURE 3.1: Print for Sudoku Text file on terminal

One of the things I should discuss beforehand is the 0 problem. When starting the program all fields in the grid are filled with a 0. This has two reasons. First, if those fields are left empty and want to perform a check or solve the puzzle, such that the program will trigger the action, the program will crash since it can't pass any value into a list. I could have worked around this by adding some if statements, saying that if a field has no value inside it then add a 0 to the list but decided to leave this for the future development of the program. Some sort of error catching can be added in case a value is deleted by mistake and no 0 is written back into the field, but since it is closely connected to the problem above so, decided to leave this for the future development.

Additionally in Figure 3.1 the first command mean loading library to the cluster, then submitting job to cluster. **qsub** mean submitting queue to cluster then **qstat** is for checking the queue status in cluster. For this project local cluster is build as discussed in section 1.8.

3.2 Output

The unique missing method and the naked single method are able to solve all puzzles with easy and medium level of difficulties. In order to solve puzzles with even more difficult levels such as hard and evil the different method has been used to complete the algorithm. If we take backtracking method it find empty square and assign the lowest valid number in the square once the content of other squares in the same row, column and box are considered. However, if none of the numbers from 1 to 9 are valid in a certain square, the algorithm backtracks to the previous square, which was filled recently.

3.2.1 Sudoku 4 x 4

In Figure 3.2 the out for the input Sudoku of **4 x 4** assigning 2 processors is shown.

```
wamiqraza@DESKTOP-DJL7EOI:/mnt/c/Users/User/Desktop/hpc4ds/build$ mpirun -np 2 ./sudoku --input=./inputs/0.txt
 1  0  2  3
 2  3  4  1
 3  2  1  4
 4  1  3  2
```

FIGURE 3.2: Solution Sudoku 4 x 4 with 2 processors

3.2.2 Sudoku 9 x 9

In Figure 3.3 the out for the input Sudoku of **9 x 9** assigning 4 processors is shown.

```
wamiqraza@DESKTOP-DJL7EOI:/mnt/c/Users/User/Desktop/hpc4ds/build$ mpirun -np 4 ./sudoku --input=./inputs/0.txt
 1  5  3  2  6  7  4  8  9
 2  4  6  1  8  9  3  5  7
 7  8  9  3  4  5  1  2  6
 3  1  2  4  5  6  7  9  8
 4  6  7  8  9  1  2  3  5
 5  9  8  7  2  3  6  1  4
 6  2  1  5  7  8  9  4  3
 8  7  4  9  3  2  5  6  1
 9  3  5  6  1  4  8  7  2
```

FIGURE 3.3: Solution Sudoku 9 x 9 with 4 processors

3.2.3 Sudoku 16 x 16

In Figure 3.4 the out for the input Sudoku of **16 x 16** assigning 2 processors is shown in the first part of figure while in below solution is for assigning 4 processors is shown. The reason for showing the different grids Sudoku and solving with multiple and different processor mean that program is capable to solve all kind of tasks.

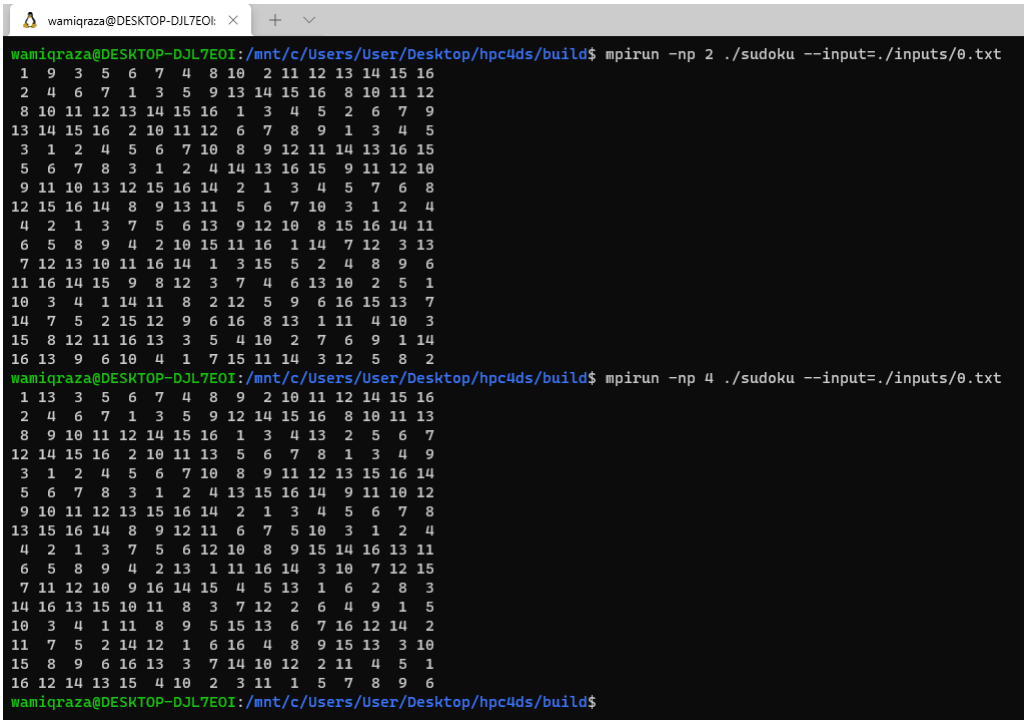

FIGURE 3.4: Solution Sudoku 9 x 9 with 2 and 4 processors

3.3 Future Development

In this chapter want to discuss some of the things that could be implement, change or add in any future development of my program. This project has much potential. There are many topics, that could be deepened and extended. This includes some of the theoretical aspects as well as some practical things. I will divide all these ideas into two areas; bugs and improvements.

3.3.1 Bugs

1. **The 0 problem**

   This is obviously one of the more serious issues, that should have been addressed already. Ideas of how to solve this problem or indeed work around it could be to add some if statements and in case an entry field is empty we mechanically add a 0 to the corresponding list.

3.3.2 Improvements

1. **Lock given numbers in grid**

2. **Nicer GUI, bigger font**

3. **Pencil function**

4. **Improve Brute Force Solver with heuristics**

These are some general improvements, for the game. Being able to lock the initial, set of numbers is a convenient feature and can probably be implemented by making better use of the Glade programs and the attributes for each text entry field. Similarly, making the GUI version look a bit nicer and increase the font size for example should be doable with Glade.

Some more advanced improvements could be to add the possibility of writing small notes in the corner of each text entry field. Not sure how could do this, One way would be to add a button which activates the 'pencil' function which then just simply puts some label on top of the text field might be an option. It is defiantly an interesting feature to add, both for the puzzle experience and also as a challenge for the coder.

3.4 Time complexity

Time complexity of an algorithm, describes the time that is needed to run on a computer and it's commonly expressed using with **O** notation. One way to estimate time complexity, is to count the number of operation's achieved by the algorithm. Since the performance time can be changed with different inputs of the same size we use the worst-case time of complexity, denoted as **T(n)**. A recent study by Kovacs showed that the worst-case complexity is related to the difficulty of the hardest puzzle.

We shall examine the time complexity of Sudoku solver. The input to a Sudoku solver is a Sudoku board. The standard board is a **n x n** grid and both smaller and larger boards are used. Theses types of puzzles with small sizes can be solved quickly and faster by computer, but only because this is small for a computer. Solving Sudoku is one of NP-complete problems and it says that Sudoku algorithms do not scale well to larger boards and puzzles, for example **100000 X 100000** grids is not feasible. If the size of input to Sudoku solver goes to infinity the time of complexity will increase exponentially. However, when solving the puzzles with limited input size such as **9 x 9** grids it is feasible because they can be solved in polynomial time.

3.5 Summary of Completed Work

Have managed to write a Sudoku program that can solve puzzles entered by the user using separate file having text format. All code is written in **C** language along with **MPI** programming built in function to send and receive message from server . The program also gives the opportunity to solve same input example puzzles with assigning multiple processors for demonstration purposes refer to section 3.2.