1 ECTS Study Project of the Module Parallel Computing (BV440004) SS21'

Chair of Computational Modeling and Simulation
Department of Civil, Geo and Environmental Engineering
Technical University of Munich
Arcisstraße 21
D-80333 München

Parallelization of a Sudoku Solver

- Project Proposal -

Hua-Ming Huang
huaming.huang.tw@gmail.com

Supervisor	Christoph Ertl
Start	26.07.2021
End	20.08.2021

Contents

1 Background

1.1 What is Sudoku?

A Sudoku puzzle is a n-by-n grid that contains numbers from 1 to n, with box size $\sqrt{n} \times \sqrt{n}$. A standard Sudoku contains 81 cells, in a 9×9 grid, and has 9 boxes (3×3 grid), as shown in Figure 1.

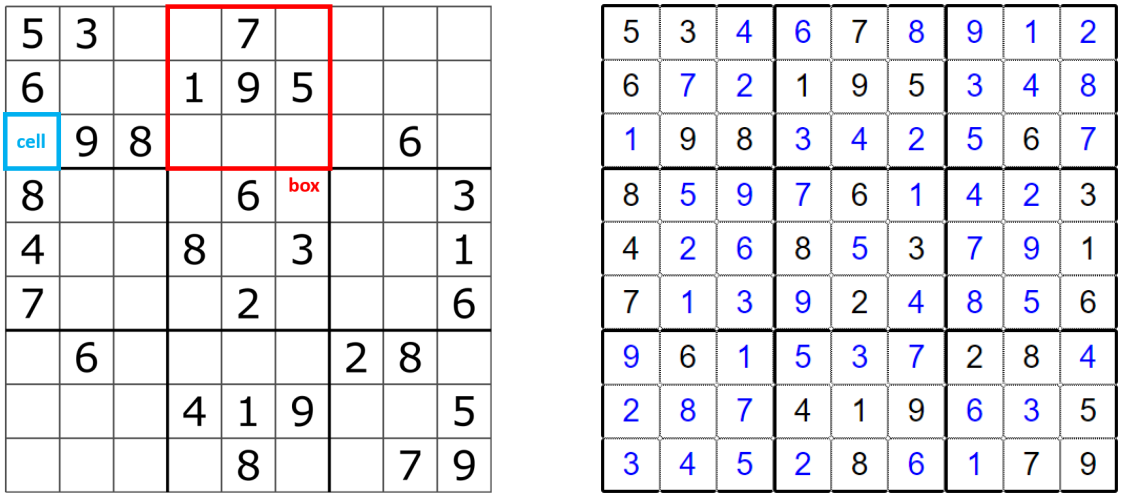

Figure 1: A typical Sudoku puzzle (left) and its solution (right)

The goal of the puzzle game is to fill in the empty cells on the board such that each column, row, and box (subgrid) contains every number in the set $\{1, ..., 9\}$ exactly once. Sudoku puzzles usually come with a partially filled-in board (*clues*). The difficulty of the puzzle varies depending on how many numbers are given, as well as the location of the given numbers. There are many strategies to solve Sudoku puzzles (See Section 1.3); solutions might not be unique, or might not even exist. - However, A properly formulated Sudoku puzzle has an unique solution that can be reached logically.

1.2 Rules of Sudoku

- Each number must appear exactly once in each row.
- Each number must appear exactly once in each column.
- Each number must appear exactly once in each box.

The above rules imply no *duplicate* numbers in any row, column and box.

1.3 Solving Algorithms

1.3.1 Naive brute-force algorithm

A brute-force algorithm would enumerate *all* possible combinations of numbers for each empty cell. This algorithm has an incredibly large search space to wade through. For instance, a blank n-by-n grid has a total of $n^{n \times n}$ different possible combinations of solutions. Solving Sudoku puzzles in this way is extremely slow and computationally expensive, but guarantees to find a solution eventually.

1.3.2 Backtracking algorithm

[2]

A common algorithm to solve Sudoku boards is called *backtracking*, which is a type of brute-force search. This algorithm is essentially a depth-first search (DFS) by completely exploring one branch to a possible solution before moving to another branch.

Given a partially filled-in (incomplete) board, the following illustrates the backtracking algorithm step by step:

1. Finds the first empty cell on the given board.
2. Attempts to place the smallest possible number (namely 1) in that cell.
3. Checks if the inserted number is valid.
    - If the number is valid (i.e., does not violate the rules stated in Section 1.2), proceed to the next empty cell and **recursively** repeat steps 1-3.
    - If the number is not valid, increment its number by 1 and repeat step 3. If a cell is discovered where none of the possible numbers is allowed, then reset the cell you just filled to zero/blank and backtrack to the most recently filled cell. The value in THAT cell is then incremented by 1 and repeat step 3. If the number cannot be incremented, then we backtrack again.
4. Repeats the procedure until either there are no more empty cells on the board, which means we have found the solution. Or we backtracked to the first unfilled cell. In this case, no solution exists for the given board.

Rather than trying to continuing a solution that can never possibly work which we do with naive brute-force algorithm, we *only* continue solutions that currently work and if they don't work we backtrack to the previous step and try other numbers again. This is going to be a lot faster than trying every single possible combination of solutions as naive brute-force algorithm did.

2 Motivation

When I was a little boy, it was much interest for me to play Sudoku games with a pen and a sheet of paper. Even though it sometimes took me hours to find a solution, I always found myself enjoying in the process of solving Sudoku. Sudoku is a "brain game" that gives me a chance to stretch my brain and polish my thinking skills. After taking the lecture "Parallel Programming (IN2147)" in the summer term 2021, I see the possibility of parallelizing the solving algorithms of Sudoku. This inspires me to do a case study for Sudoku solver and apply the parallelization techniques I have learned throughout the lecture to this topic.

It has been proven that the total number of valid solutions to a standard 9×9 Sudoku puzzle is approximately 6.671×10^{21} [1]. Trying to populate all possible valid solutions is itself a difficult problem because of the huge number. Assuming each solution takes 1 millisecond to be found, then with a simple calculation we can determine that it takes about 2.115×10^{11} years to find all possible solutions.

There are already several computer algorithms that will solve 9×9 Sudoku puzzles *in fractions of a second*. However, as the size of Sudoku puzzle gets larger, e.g., 16×16 (Figure 2), 25×25 (Figure 3), the *combinatorial explosion* occurs and thus leading to the overall solving time grows *exponentially* [3]. This also directly eliminates the possibility of solving Sudoku in a reasonable amount of time.

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

Figure 2: A 16-by-16 Sudoku puzzle

1	8	16	15				12					7					25				2	9	10	5
2			10					25	11						16	4					18			7
25	23		18		19			3	6						7	24			1		20		21	14
		11		5	7	15	21		16	18				20	10		8	6	14	24		3		
6				17		2	10		22	25		4		21	20		15	5		11				1
5	10		16				17	11								8	21				24		1	15
		1					23			7		11		16			20					22		
		25			12		22		8	1	3		6	18	24		7		16			10		
7	14	21			3	5	25	10		15		8		24		22	2	1	13			19	17	9
4		23						24	9	17				13	14	10						7		11
	16		21		6	12		15			1		7			5		25	18		9		2	
23	4	13			25			5	10		15		19		21	7			8			24	6	20
9				11			7		17			24			15		22			1				19
14			25	7							16	18	10							22	8			3
22	6				11		24		1			12			2		14		9				4	10
	24	3		4	22					12	2		8	11					6	19		20	9	
		8	5	21			6	1	2						18	15	24			14	13	25		
		7	14			24	13		20	5	6	19	23	10	8		4	11			3	16		
10				16	8				12		18		22		5				20	15				24
		6	11		10	25		14			21		20			23		19	12		7	2		
		9					16				7	13	2				12					17		
					23	7			15						13			21	2					
16	25	20				3		17		9	4	6	14	15		18		22				23	12	13
	17						20		24		11		12		23		10						3	
			1	23		13	19	2	14						3	6	5	4		8	22			

Figure 3: A 25-by-25 Sudoku puzzle

Therefore, I aim to parallelize a Sudoku solver, so that large Sudoku puzzles can be solved more efficiently.

3 Proposed Approach

I will start with the sequential implementation of a standard 9×9 Sudoku solver primarily from scratch using C++. The sequential code will be based on the backtracking algorithm as introduced in Wikipedia. After that, it will be extended to accept a $N \times N$ Sudoku grid. In order to solve the Sudoku puzzle in the shortest amount of time, I plan to have a parallel implementation of the Sudoku solver using OpenMP. With the built-in dynamic scheduling policy in OpenMP, I anticipate there will be a fair amount of work distribution among workers and therefore reducing the computation time. Hopefully the speedup will scale as the number of threads increases in my parallel implementation. Finally, I will write up a project report to document and summarize the performance results of both my sequential and parallel implementation.

4 Schedule

This study project will consist of 30 hours of workload to compensate the missing 1 ECTS in the module "Parallel Computing (BV440004)". Considering the aforementioned time commitment, I organize a proposed schedule and define some tasks in order to complete this project in the following table:

Week	**Date**	**ToDo Plan**
1	July 26 - July 30	• Do research about Sudoku problem in depth. • Implement sequential version of a 9×9 Sudoku solver based on backtracking algorithm in C++. • Prepare test cases with different puzzle sizes and difficulties. • Build a Makefile to compile the sequential code.
2	August 2 - August 6	• Extend sequential implementation to work on Sudoku puzzles bigger than 9×9. • Profile sequentail code and identify bottlenecks.
3	August 9 - August 13	• Implement a parallel Sudoku solver using OpenMP. • Build a Makefile to compile the parallel code.
4	August 16 - August 20	• Write up a final project report for speedup results analysis and also create a README for code usage. • Clean up code, write down comments for readability, and submit solutions to remote repository.

References

[1] *Combinatorial explosion of Sudoku puzzle — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Combinatorial_explosion#Sudoku.

[2] *Python Sudoku Solver Backtracking — tech with tim*. URL: https://www.techwithtim.net/tutorials/python-programming/sudoku-solver-backtracking/.

[3] *Sudoku solving algorithms — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Sudoku_solving_algorithms#Example_of_a_brute_force_Sudoku_solver_.28in_C.29.