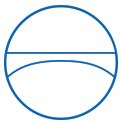
1 ECTS Study Project of the Module Parallel Computing (BV440004) SS21'

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 Report -



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

Supervisor: Christoph Ertl

Contents

1 Introduction

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.

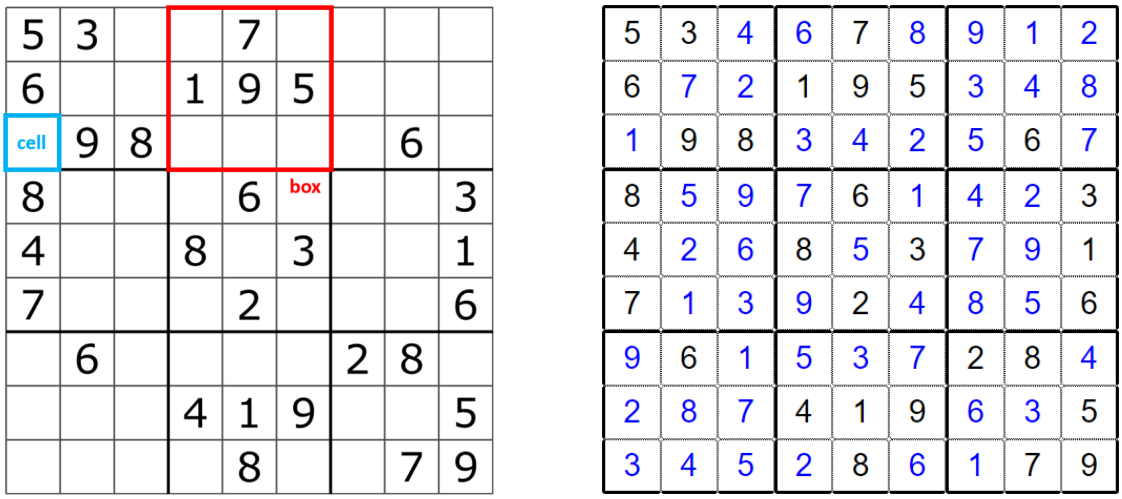


Figure 1: A typical Sudoku puzzle (*left*) and its solution (*right*)

The goal of the puzzle game is to fill in the empty cells on the board such that each column, row, and box (or called "subgrid", "region", "block") 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 Solving Algorithms). Solutions might not be unique, or might not even exist. However, a properly formulated Sudoku puzzle should have an unique solution that can be reached logically.

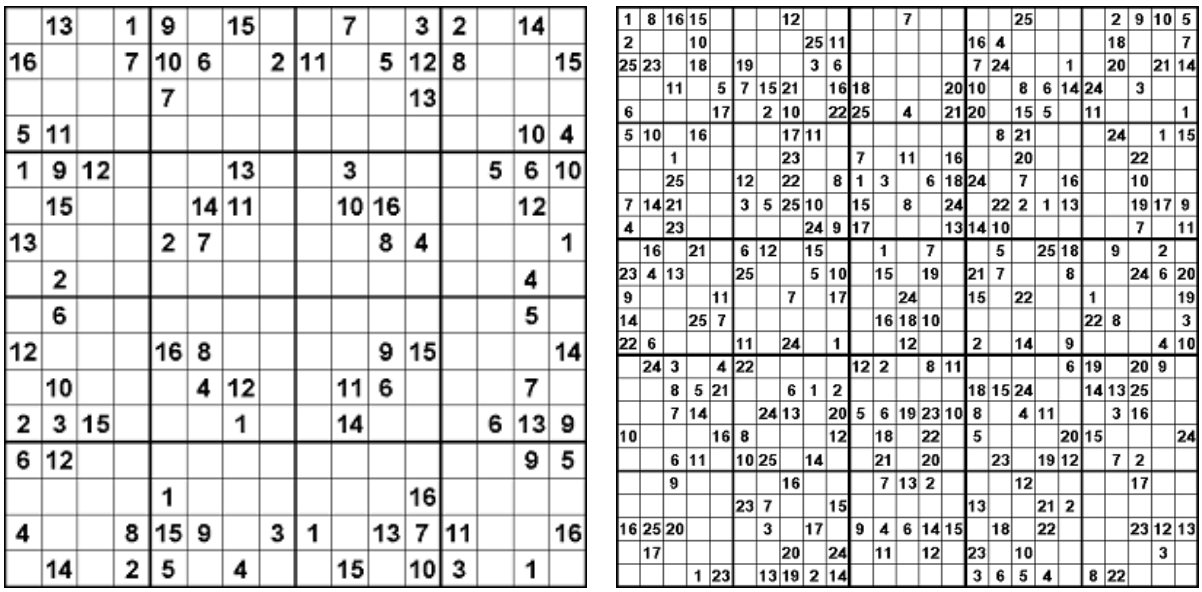
1.2 Rules of Sudoku

- 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.

2 Motivation

Thanks to the computational power provided by modern computers, there are already several algorithms (See Section Solving Algorithms) that will solve 9×9 Sudoku puzzles in fractions of a second. However, as the size of Sudoku puzzle n gets larger, e.g., 16×16, 25×25 (Figure 2), the *combinatorial explosion* occurs and thus leads to *exponential* growth of overall solving time. Combinatorial explosion creates limits to the properties of Sudokus that can be constructed, analyzed, and solved [2]. This also directly eliminates the possibility of solving Sudoku in a reasonable amount of time.



(a) A 16×16 Sudoku puzzle

(b) A 25×25 Sudoku puzzle

Figure 2: Large Sudoku puzzles

Therefore, this project aims to implement a Sudoku solver that could solve **large** Sudoku puzzles as efficiently as possible by means of various parallelization techniques and algorithms.

3 Solving Algorithms

[7]

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 intensive, but guarantees to find a solution eventually and the solving time is mostly unrelated to degree of difficulty.

3.2 Backtracking algorithm

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: [5]

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

3.3 Dancing Links algorithm

The Dancing Links algorithm developed by Dr. Donald Knuth is an algorithm for solving the *exact cover problem*, where some constraints need to be applied. The exact cover problem can be represented by a matrix of 0s and 1s and the Dancing Links algorithm will find a set of rows such that the number 1 appears in each column exactly once. [3][1][4]

As Sudoku is a special case of the exact cover problem, we can model a Sudoku puzzle in the form of a (sparse) cover matrix:

- The *columns* of cover matrix represent the 4 constraints of the Sudoku puzzle:
  - **Cell constraint:** Each cell contains only an integer between 1 and n which corresponds to the size of the baord.
  - **Row constraint:** Each row contains numbers in the set $\{1, ..., n\}$ exactly once.
  - **Column constraint:** Each column contains numbers in the set $\{1, ..., n\}$ exactly once.
  - **Box constraint:** Each box contains numbers in the set $\{1, ..., n\}$ exactly once.

  Each number comes with its own set of constraints. Therefore, there are $n \times n \times 4$ columns.
- The *rows* of cover matrix represent every single possible position for every number. Therefore, there are $n \times n \times n$ rows.

The initial state of the board can be expressed by selecting which rows are in the exact coverage. The Dancing Links algorithm will then give us a subset of rows where each constraint is filled exactly once and thus the solution to the Sudoku board.

4 Implementation Details & Performance Results

Please accomany this section with the code and comments in the corresponding `*.hpp` and `*.cpp` files. Only some key aspects will be listed below.

4.1 Sequential backtracking algorithm

- A backtracking algorithm visits the empty cells in some order, filling in possible digits sequentially, or backtracking when the number is found to be not valid. [7]
- There are no parallelizations done for this algorithm because of the following reasons:
  - Since backtracking is a depth-first search (DFS), it is not directly parallelizable - because this algorithm depends on the *stack*, which is implicit in the function call stack. Stacks are hard to parallelize because threads cannot all work on the same stack and efficiently move forward in the algorithm together without causing very high contention for accessing to the stack. Therefore, any newly computed result cannot be directly used by another thread.
  - With the `boolean` return type of backtracking algorithm (since the recursion *only* continues solutions that currently work, a `boolean` checking is required), it's also hard to parallelize. Instead, functions with `void` return type is much easier to parallelize.

4.2 Sequential brute-force algorithm

As stated in 3.1, brute-force algorithm is guaranteed to find a solution since it tries every possible number in each empty cell. This algorithm is very effective for 9×9 puzzles. Unfortunately for 16×16 (or 25×25) puzzles there are 16 (or 25) possibilities for each empty cell. This means that there are roughly 16^{16^2-n} (or 25^{25^2-n}) possible states that might need to be searched, where n is the number of filled-in cells in the given Sudoku puzzle. It turns out that the brute-force algorithm scales very badly (Figure 3).

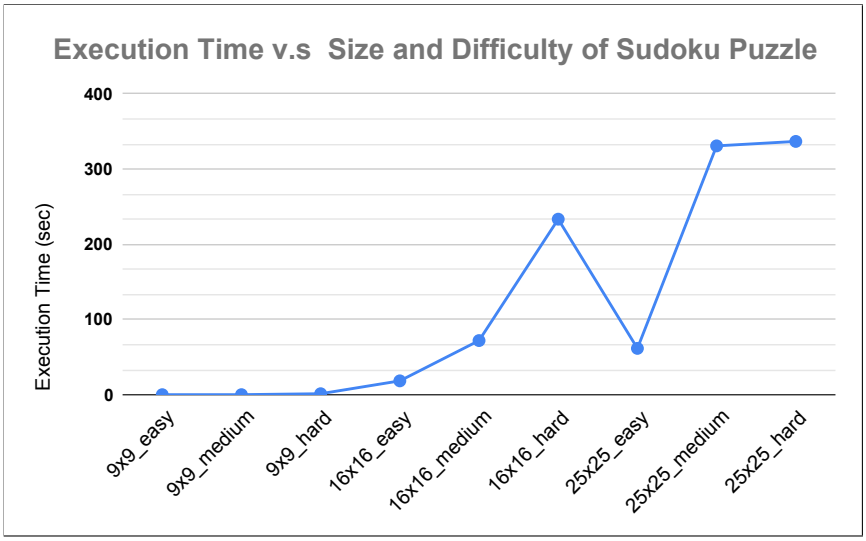


Figure 3

4.3 Parallel brute-force algorithm

There are 3 versions of this implementation:
`solve_bruteforce_par`, `solve_kernel_1` and `solve_kernel_2`

- `solve_bruteforce_par`: Exploit OpenMP tasks for parallelism on multiple recursion levels and prevent overhead of task creation by setting a threshold for the recursion depth. If the recursion depth is larger than the threshold, switch to `solve_bruteforce_seq`.
- The ideas of `solve_kernel_1` and `solve_kernel_2` are the same: Similar to backtracking algorithm, brute-force algorithm is also a depth-first search (DFS) and doesn't appear to be very parallelizable. In contrast, breadth-first search (BFS) is easy to parallelize. So we can modify the depth-first search (DFS) approach such that threads can work independently without reading from the same stack. Namely, we can parallelize over different branches of the depth-first search (DFS). This approach is summarized as follows:
  1. Pushes the input board onto the board deque to initialize a list of incomplete (or intermediate) solutions. (The deque scheme for parallization is inspired by this GitHub repo: [6])
  2. **Bootstrapping**
     (a) Pops a new Sudoku board from the front of the board deque.
     (b) Serially fills in empty cells with all possible numbers on this board, and pushes the newly created grids into the deque accordingly. This step expands the search

tree in a BFS manner. Whenever an empty cell has no possible numbers, prune the DFS tree, i.e., ignore the branch.

(c) Repeats step (a) and (b) till the specified level of bootstrapping is reached. This may give us something like thousands of possible boards on the deque to be solved by each thread. An example of bootstrapping to level 5 with `Test_Cases/9x9_easy.txt` as input grid is illustrated in Figure 4.

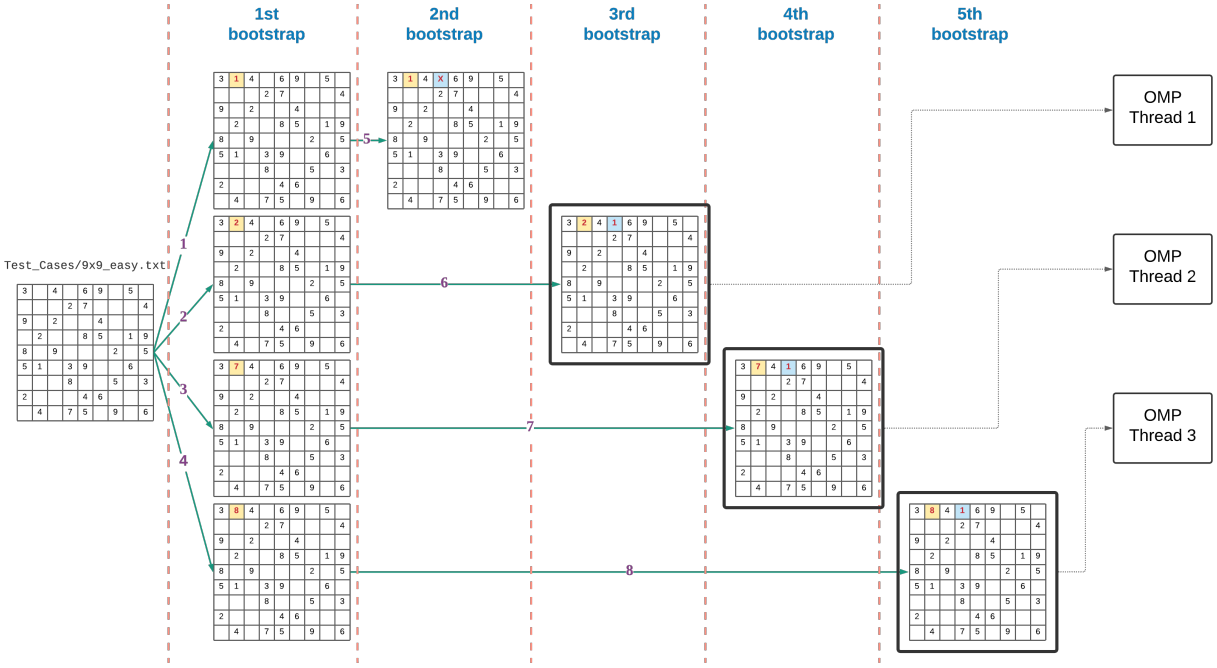


Figure 4

3. Each OpenMP thread gets a Sudoku board from the board deque as starting points respectively and applies sequential brute-force solving algorithm (DFS) on its own board in parallel. (Use static scheduling to balance the workload among all threads in round-robin fashion because each thread solves exactly one board.)
4. Whenever the solution is found in one of the spawned threads, a flag variable is set to `true` and the remaining iterations will be skipped by the `continue` statement.

❋ **Discussion:** As seen from the performance result in Figure 5, this competitive parallelization approach is not scalable with respect to the number of OMP threads, which is set equal to the level of bootstrapping in the current version of implementation. I found the main overhead of this parallelization approach comes from the *level of bootstrapping*, whcih is critical and should be obtained by some optimization metrics. If a non-optimal

value of bootstrapping level is chosen, the limitation of scalibility will be created and no visible speed-up will be measured.

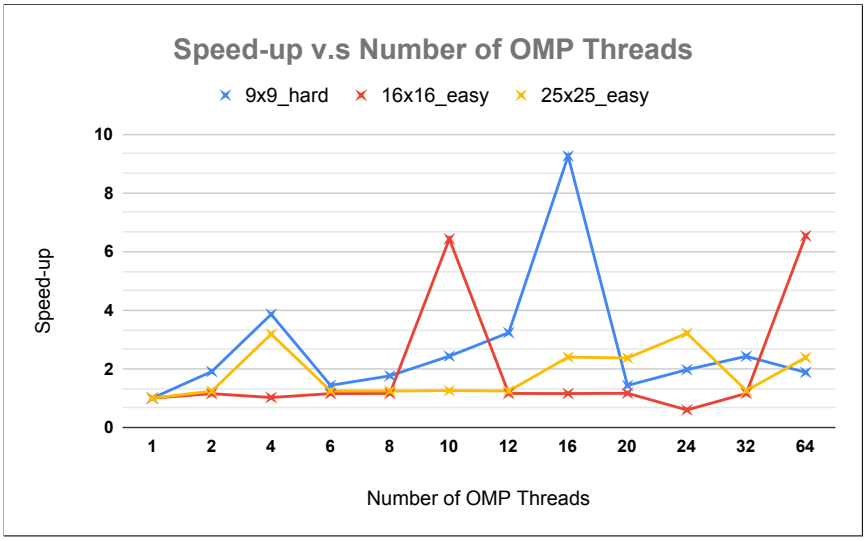


Figure 5

4.4 Sequential DLX algorithm

With the Dancing Links algorithm, even complex Sudoku puzzles can be solved very quickly and efficiently as indicated in Figure 6. This performance result can be compared to Figure 3.

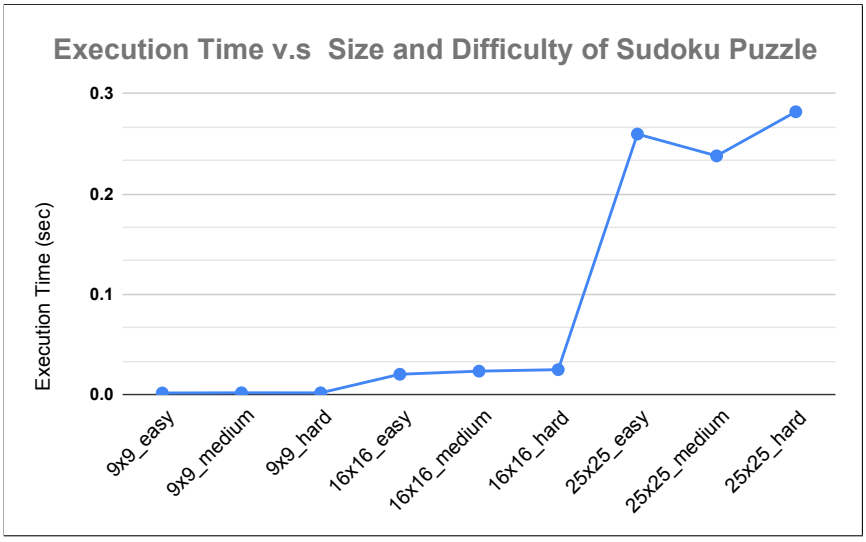


Figure 6

4.5 Parallel DLX algorithm

The Dancing Links algorithm does not appear to be very parallelizable. At its core, it is similar to the depth-first search (DFS) solving algorithms (i.e., backtracking, brute force) and will face similar parallelization challenges. Therefore, in this implementation I simply put the `solve` function into a parallel region and modify the `selectColumnNodeHeuristic` function to start with satisfying a different constraint in each thread. As expected, the performance doesn't benefit from parallelization because the sequential version performs well enough already.

References

[1] *A Sudoku Solver in Java implementing Knuth's Dancing Links Algorithm*. URL: https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/sudoku.paper.html.

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

[3] Donald E Knuth. "Dancing links". In: *arXiv preprint cs/0011047* (2000).

[4] *Knuth's Algorithm X — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Knuth%27s_Algorithm_X.

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

[6] Chris Rycroft. *am225_solutions*. 2021. URL: https://github.com/chr1shr/am225_solutions/tree/master/hw1/problem4.

[7] *Sudoku solving algorithms — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Sudoku_solving_algorithms.