Parallelization of Sudoku Solvers: A Comparative Study of Brute Force and Backtracking

Wei Che Hsu
National Yang Ming Chiao Tung University
Taoyuan, Taiwan
wesleyshi.en10@nycu.edu.tw

Yu Hong Shen
National Yang Ming Chiao Tung University
Taipei, Taiwan
stanley.shen2003.c@nycu.edu.tw

Ting Han Wu
National Yang Ming Chiao Tung University
Taichung, Taiwan
greenleaf.cs10@nycu.edu.tw

Figure 1: A Sudoku puzzle

1 Abstract

We present a parallel Sudoku solver that uses a bootstrap method to achieve effective workload balancing. We analyze the characteristics of depth-first and breadth-first search spaces employed by backtracking and brute-force methods, respectively. Implementing a depth-first search reduces unnecessary exploration, saving time and memory. Both backtracking and brute-force algorithms are parallelized using OpenMP and pthreads. Experimental results show that parallel backtracking achieves a speed-up of 3.39× and parallel brute force attains a speed-up of 3.79× compared to their serial versions. Our approach enhances the efficiency and scalability of Sudoku solving through strategic parallelization and workload balancing.

2 Introduction

Sudoku is a constraint satisfaction problem which is applicable in various domains such as optimization, artificial intelligence, and biological systems.

In this project, our aim is to parallelize Sudoku solvers using pthreads and OpenMP, focusing on two distinct solving methods: brute force and backtracking, utilizing BFS and DFS as their respective search spaces. Our objective is to compare the performance, scalability, and memory utilization of these approaches in solving $16 * 16$ Sudoku puzzles.

Sudoku is a combinatorial puzzle that becomes increasingly complex as the size of the puzzle increases [5]. Solving larger Sudoku puzzles presents significant computational challenges, making it an excellent topic for exploring parallel programming techniques.

The objective of Sudoku is to fill in the empty grids of a n-by-n puzzle. The puzzle is divided into $\sqrt{n}$-by-$\sqrt{n}$ subgrids, and the solution should comply with the following rules:

(1) Every grid in a puzzle should be filled with a number in {1, 2, ..., n}
(2) No grid in a row should be repeated.
(3) No grid in a column should be repeated.
(4) No grid in a subgrid should be repeated.

Initially, some clue digits are pre-filled. The player must use logical reasoning to deduce which digits should be filled in to the empty grids.

Our goal is to identify the strengths and limitations of each method and to evaluate the scenarios that are most suitable for each approach.

Conference: None
ACM ISBN None
https://doi.org/None

3 Proposed Solutions

In this section, we introduce three methods for solving Sudoku and propose our approach to parallelizing the code.

3.1 Serial Methods

3.1.1 Naive Brute Force. The most straightforward way for solving a Sudoku is to try all possible answers until the correct one if found. However, the search space for this approach is extremely large, having size n^e, where n is the size of Sudoku and e is the number of empty spaces. Due to this large search space, we try this method only on $16 * 16$ Sudoku puzzles.

3.1.2 Backtracking. The backtracking algorithm for Sudoku is a systematic method used to solve the puzzle by trying to fill the grid with numbers while adhering to Sudoku's rules. Here's a brief overview of how it works:

(1) Start with an empty cell in the grid.
(2) For that cell, place a number from 1 to 9.
(3) After placing a number, check if it follows the Sudoku rules.
(4) If the number is valid, move on to the next empty cell and repeat the process.
(5) If you reach a cell where no numbers fit, backtrack to the previous cell and try the next number.
(6) Continue until solved.

Compared to naive brute force method, backtracking improves efficiency by pruning the search space. It checks the validity of each number placement before moving forward. If a number doesn't fit the Sudoku rules, backtracking immediately reverts to the previous cell, avoiding the exploration of invalid paths.

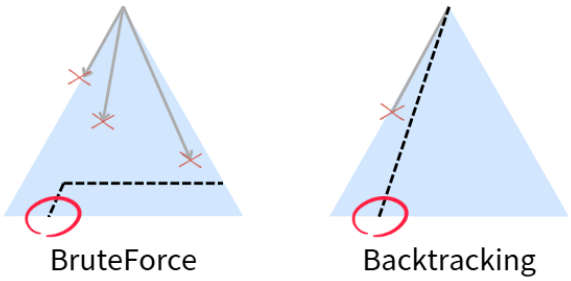

Figure 2: Comparison of bruteforce and backtracking method. The above figure is the search space of the Sudoku. Suppose the solution is at the position of the red circle, the bruteforce method will search the space above the dotted line, while the backtracking method will only search the left part of the dotted line.

3.2 Parallel Methods

3.2.1 Distrubute initail grids. To parallelize the two method mentioned above, we split the original search space into smaller nonoverlapping spaces. Given a Sudoku puzzle, we set the value for the first empty space and treat them as values that was filled in as the question puzzle. Suppose we fill in the first empty space with different values, we can generate several different puzzles and solve them in parallel.

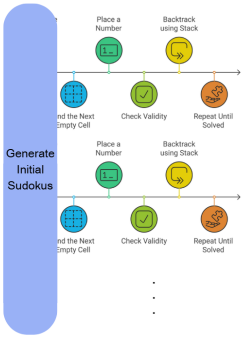

Figure 3: Diagram of parallel methods. The workload is divided at the beginning and given to threads to executed in parallel.

3.2.2 Bootstraping. Simply dividing the workload is not good enough for our task because the workload in each puzzle is not fixed. The execution will end if a thread finds the solution. This means that if the solution is in 51% of the search space and we have two threads that execute in parallel, the second thread will find the solution in 1% of the time, having a speed of approximately 50%. In contrast, if the solution is on 49% of the search space and uses two threads, the second thread will not help, resulting in no speed up and even slower than the serial method due to thread creation overhead. While the expected average speed-up is linear, the actual result is more dependent on the exact solution of the Sudoku. To make the speed up have smaller deviation, the bootstrap method helps by having the execution more similar to the serial method. The method generates many possible puzzles and solves them in parallel, making the speed-up more stable in different Sudoku puzzles.

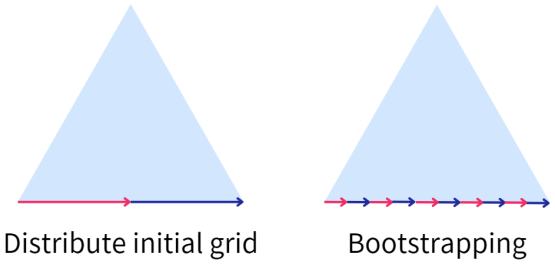

Figure 4: Comparison of two parallel methods. The red and dark blue line represents different threads.

4 Experiment

4.1 Dataset

We employed a recursive backtracking algorithm to generate fully completed 16x16 Sudoku boards using Sudoku puzzles on the internet. This ensures that each board satisfies Sudoku rules.

From each generated Sudoku board, we randomly selected 120 to 140 cells and set their values to 0, creating partially filled Sudoku puzzles of medium difficulty.

In total, we generated 100 distinct Sudoku boards, each of size 16x16.

4.2 Test Method

We evaluated the performance of our Sudoku solvers using six distinct approaches:

(1) **Brute Force - Serial**
(2) **Brute Force - OpenMP**
(3) **Brute Force - Pthreads**
(4) **Backtracking - Serial**
(5) **Backtracking - OpenMP**
(6) **Backtracking - Pthreads**

We ran the above methods on our dataset 5 times and recorded the average execution time.

4.3 Environment

We conduct our experiments on the course's workstations, which is on Debian 12.6.0 with Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz and Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz processors and GeForce GTX 1060 6GB. g++-12, clang++-11, and CUDA 12.5.1.

5 Experimental Results

The experimental results align well with our predictions and analysis. Across all test scenarios, backtracking is consistently faster than naive brute force. This outcome validates the effectiveness of using depth-first search (DFS), which not only reduces unnecessary exploration but also saves memory when solving Sudoku puzzles.

Our proposed parallel methods: brute force with pthreads, brute force with OpenMP, backtracking with pthreads, and backtracking with OpenMP, generally exhibit good scalability across most data points. However, as shown in Figure 6, the performance does not scale proportionally with thread count in all cases. Specifically, when using 5 threads, performance unexpectedly drops compared to 4 threads.

This situation can be attributed to two main factors. First, the workload distribution across Sudoku puzzles is uneven. In the serial version, some puzzles are processed very quickly. Second, the standard deviation of speed-up for individual puzzles is high. By directly dividing the initial Sudoku puzzles and processing, it may result in cases where the speed-up is not proportional to the thread count or even exceeds the theoretical maximum value for the number of threads.

These observations highlight the importance of improving workload balancing in future implementations to achieve more consistent scalability.

6 Related Works

Existing works leverage rule-based methods to fill in empty grids in Sudoku iteratively [3][2][4][7][1]. These rule-based methods are easy to parallelize, but none of them guarantee to find the solution.

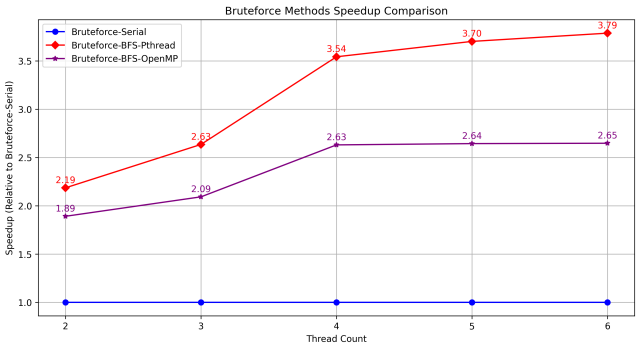

Figure 5: Speedup Comparison of Brute Force Methods: Serial, Pthreads, and OpenMP. The highest speedup of 3.79× was achieved using pthreads with 6 threads.

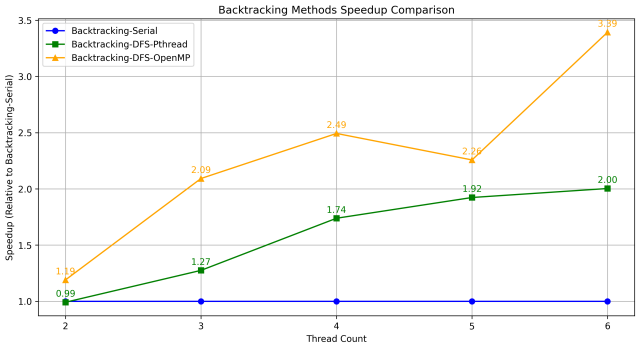

Figure 6: Speedup Comparison of Backtracking Methods: Serial, Pthreads, and OpenMP

Backtracking [8] attempts to solve Sudoku by filling in the puzzle one grid at a time, ensuring that each placement conforms to the constraints of the game. If an invalid number is placed, the algorithm backtracks to previous states and tries a different option. Although it guarantees a solution (if there is one), backtracking can be computationally expensive. To optimize this, we launch multiple processes in the distributed memory model, each working on different puzzle configurations, with validation handled by a GPU. Since solutions are usually found deeper in the game tree, the depth-first search (DFS) is preferred for performance [6]. DFS outperforms the breadth-first search (BFS) because BFS explores multiple possibilities simultaneously, which not only increases computational load but also takes longer to find a solution.

7 Conclusions

In this project, we analyzed the performance of the brute-force (BFS) and backtracking (DFS) methods, demonstrating the better efficiency of the backtracking approach. To further optimize performance, we parallelized our implementation using MPI and PThreads.

Through a comparison of the simple and bootstrap workload distribution methods, we showcased how the bootstrap method effectively achieves balanced workload distribution, leading to improved computational efficiency in parallel environments.

References

[1] [n. d.]. Hidden Pairs, Hidden Triples, Hidden Quads Strategy. http://www.thonky.com/sudoku/hidden-pairs-triples-quads. Accessed: 2024-10-17.

[2] [n. d.]. Sudoku solving techniques. http://www.su-doku.net/tech.php. Accessed: 2024-10-17.

[3] M. Asif and R. Baig. 2009. Solving NP-complete problem using ACO algorithm. In *Emerging Technologies, 2009. ICET 2009. International Conference on*. IEEE, 13–16.

[4] Tom Davis. 2012. The Mathematics of Sudoku. https://example.com.

[5] Huaming Huang. 2023. *Parallel Sudoku Solver*. Retrieved October 17, 2024 from https://github.com/huaminghuangtw/Parallel-Sudoku-Solver/tree/master

[6] Shawn Lee. 2023. Efficient Parallel Sudoku Solver via Thread Management & Data Sharing Methods. *Department of Computer Science & Engineering, University of California, Riverside* (2023). Email: slee208@ucr.edu.

[7] Halliday Rutter. 2008. Sudoku Generation Using Human Logic Methods.

[8] Wikipedia contributors. 2023. *Backtracking – Wikipedia*. Retrieved October 17, 2024 from https://zh.wikipedia.org/zh-tw/%E5%9B%9E%E6%BA%AF%E6%B3%95 [Online; accessed 17-October-2024].