Parallel Sudoku Solver

Implementation based on OpenMP

Cheng-Che Lu†
Department of Computer Science
National Yang Ming Chiao Tung University
Taiwan
cclu.c@nycu.edu.tw

Hong-Hua Liu
Department of Cyber Security
National Yang Ming Chiao Tung University
Taiwan
hhliu.cs11@nycu.edu.tw

Heng-Yi Chiu
Department of Computer Science
National Yang Ming Chiao Tung University
Taiwan
5542045aa@gmail.com

ABSTRACT

Sudoku is a logical-based, number-placement puzzle. To solve the Sudoku puzzle, most programmers apply backtracking-based methods with optimizations such as *Crook's pruning* or cross-hatching.

This report proposes a parallelized Sudoku algorithm based on the optimized serial solver. Our method best utilizes the parallel computation resources by efficiently reducing the potential communication overhead between threads, achieving a speedup of at most 5.67x using 16 threads. The report also analyzes the relationship between the difficulty of Sudoku and performance improvement. We find out that the harder the puzzle is, the more speedup we could get from parallelization.

KEYWORDS

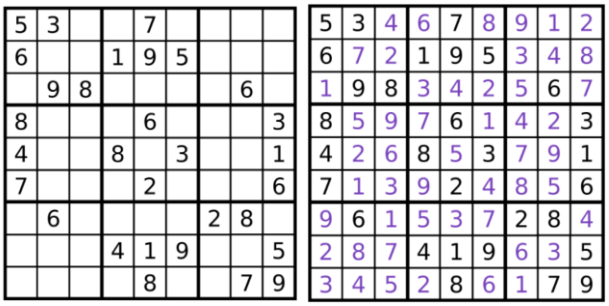
OpenMP; *Crook's Algorithm*; Data-level Parallelism

1. INTRODUCTION

The objective of classic Sudoku is to fill a 9 × 9 grid with digits so that each column, each row, and each of the nine 3 × 3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contain all of the digits from 1 to 9. For a correct solver Sudoku, no number is repeated within a single row, column, or subgrids. The player should use logical deduction to fill the valid values into the empty cells.

Typically, the difficulty of a Sudoku puzzle depends on the total count of filled digits initially and its distribution. A standard Sudoku puzzle and one of its solutions are given in figure 1 as follow.

From the viewpoint of graph theory, the Sudoku puzzle could be viewed as a special case of a graph coloring problem. As a special case, there are few real-life applications related to the Sudoku problem. It usually serves as a game to train our logical thinking ability. While on the other hand, since the graph coloring problem has been proven to be NP-complete, there is no guarantee of any algorithm whose time complexity is within polynomial. In the worst case, the solved time could be exponential to the puzzle size. Such property gives the parallel optimization some chance to play a role when the puzzle is hard and the solving time is critical for the user. In the following part, we introduce the commonly-used serial Sudoku solving algorithm first, then propose parallel-based optimization in the next chapter.



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

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

Figure 1. An example of classic Sudoku

A naive solution to the Sudoku puzzle is to apply the backtracking algorithm. The backtracking algorithm visits all the unfilled cells in some order, fills the digits incrementally, and then backtracks when a filling attempt is invalid. It could be viewed as a brute-force method with early stop checking. This method guarantees that a solution will be found eventually but does not guarantee the program's running time.

Based on the backtracking algorithm, two commonly-used, serial-based optimizations are proposed to improve the performance. *Crook's algorithm*, proposed by *J.F. Crook* [2], performs the pruning operations, such as elimination and lone-ranger, to reduce the search space every time after a filling operation. cross-hatching selection [5] prioritizes the cells to be filled according to the count of the fillable digit to reduce the search space more efficiently with fewer guess (uncertain filling) attempts.

Other more advanced techniques, such as the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm [7] or utilizing the strongly connected components property of the Sudoku puzzle, could further reduce its solving time. Due to the limited scope and time, however, they are not included in this report. Instead, the report focuses on utilizing parallel-based optimization to improve the program performance.

2. PROPOSED SOLUTION

The parallelized algorithm consists of two steps, splitting and solving. First, the program splits the Sudoku puzzle into several independent sub-problems. This could be achieved by assigning

different digits to the same cell without overlapping the search space. The splitting operation is performed recursively in a BFS manner until enough branches are divided for parallel computation. Secondly, the program simultaneously launches all available threads to solve the sub-problems.

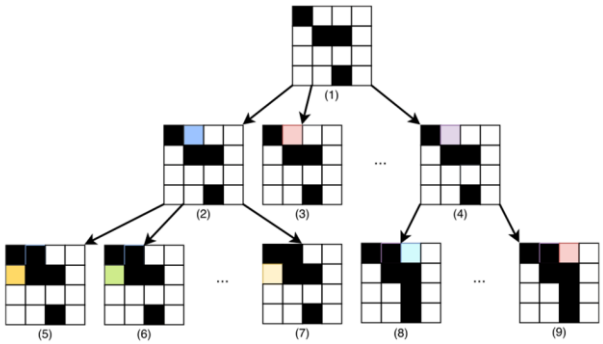


Figure 2. Illustration of splitting search space

Take Figure 2. as an example. The program divides the search space into three non-overlapping ones for the first split according to one unfilled cell. Following the same mechanism, each subspace is further split into smaller pieces based on their next unfilled cell. Such splitting operation continues until the number of branches meets the minimum requirement (the number of branches should be at least equal to the number of available threads).

There are two highlights for our methodology. The first is that the splitting operation is performed statically, i.e., the parallel computation starts until all the splitting operation is done. This way, the threads are mutually independent when solving the sub-problem, reducing the communication overhead. Secondly, the *Crook's pruning* and the cross-hatching selection are adopted during the splitting step. Such optimizations ensure the size of search space (all the possible digits combinations of the Sudoku puzzle) is best reduced and best divided.

3. METHODOLOGY

We implement two serial Sudoku solvers with different degrees of optimizations (backtracking + *Crook's pruning* / cross-hatching selection) and compare them with the parallelized ones. Datasets *puzzles2_17_clue* and *puzzles5_forum_hardest_1905_11+* are used to evaluate the program performance. Table 1. and Table 2. are the basic description of the dataset and evaluation platform.

4. EXPERIMENTAL RESULTS

4.1 Dataset and Environment

Based on the methodology mentioned above, this report evaluates the program performance on two hard datasets, *puzzles2_17_clue* and *puzzles5_forum_hardest_1905_11+*, with different numbers of threads.

Table1. Details of evaluation dataset

Dataset	Size	Difficulty
Puzzles_2_17_clue	49,158	Hard
Puzzles5_forum_hardes_1905_11+	48,877	Even Harder

Table 2. Details of evaluation platform

Component	Specification
CPU	Intel® Xeon® Silver 4214 with 24 Cores
RAM	128 GB
OS	Ubuntu server 20.04.4LTS
Compiler	g++ 9.4.0
OpenMP	4.5

4.2 Serial Solver Performance

Table 3. shows the performance of a wholly serialized program as the baseline. For both datasets, the simple backtracking solver runs slower than the one with *Crook's pruning* or cross-Hatching selection by at least an order. With both optimizations, the solvers complete a puzzle in an average of 0.44 ms for *puzzles2_17_clue* and 1.93 ms for *puzzles5_forum_hardest_1905_11+*. We observe such a huge improvement mainly because of the reduced puzzle search space. When the search space, i.e., the total number of guessing choices, is large, it requires more attempts to correctly choose the correct answer, leading to a longer running time — vice versa.

4.3 Parallelized Solver Performance

Considering the poor performance of simple backtracking serialized solver, we only parallelize the optimized solver for better results. Table 4. summarizes the parallelized program performance.

It could be observed that for dataset *puzzles2_17_clue*, the average solved time is reduced from 0.92 ms / 0.44 ms (1 thread used) to 0.19 ms / 0.09 ms (16 threads used), achieving a 4.72x / 4.57x speedup. Parallel computation indeed improves the solved time by efficiently utilizing more computation resources.

From another aspect, consistent improvement exists if comparing the running time of different solvers with the same number of threads used. That is to say, the serialized-based optimizations (*Crook's algorithm* & cross-hatching selection) and parallel-based optimizations are almost orthogonal. Both methods could be applied at the same time to achieve the best performance.

Table 3. Serialized program results

Dataset	Serial Solver	Average Solved Time	Search Space Size
Puzzles2_17_clue	Backtracking Solver	> 1000 ms	> 5M
	+ *Crook's Pruning*	0.92 ms	121
	+ *Crook's* & cross-hatching	0.44 ms	76
Puzzles5_forum_hardest_1905_11+	Backtracking Solver	> 50 ms	> 300K
	+ *Crook's Pruning*	2.69 ms	346
	+ *Crook's* & cross-hatching	1.93 ms	278

Table 4. Parallelized program results of *puzzels2_17_clue* dataset

Serial Solver	No. of thread	Average Solved Time	Search Space Size	Relative Performance	Parallel Efficiency
Crook's Pruning	1	0.92 ms	121	1x	1
	2	0.72 ms	82	1.27x	0.63
	4	0.37 ms	38	2.46x	0.61
	8	0.24 ms	21	3.76x	0.47
	16	0.19 ms	13	4.72x	0.30
Crook's Pruning + cross-hatching	1	0.44 ms	76	1x	1
	2	0.27 ms	31	1.63x	0.82
	4	0.15 ms	14	2.96x	0.74
	8	0.10 ms	6	4.36x	0.55
	16	0.09 ms	4	4.57x	0.29

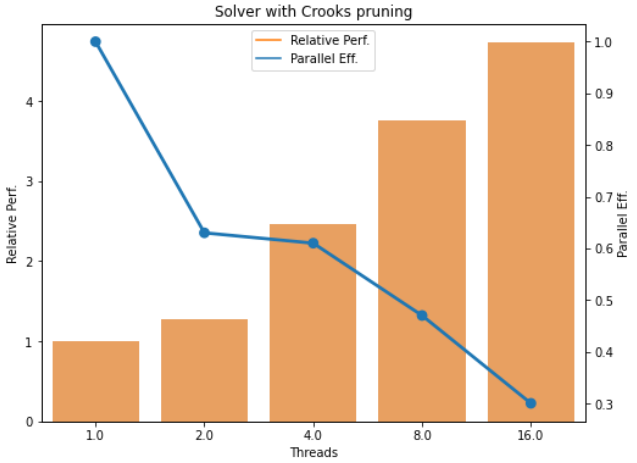


Figure 4. Relative performance and parallel efficiency of parallelized solver with *Crook's pruning*

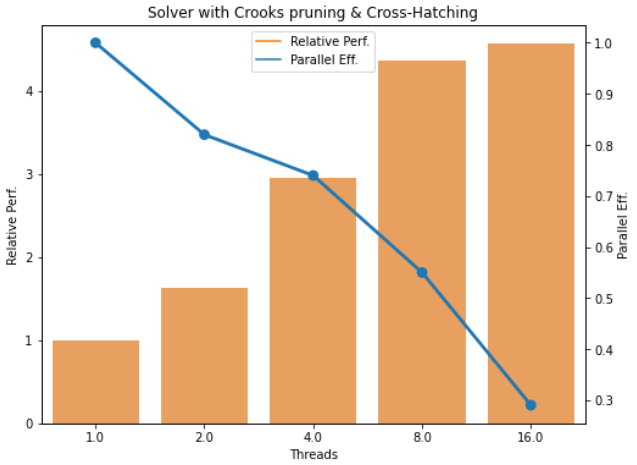


Figure 5. Relative performance and parallel efficiency of parallelized solver with *Crook's pruning* and cross-hatching selection

The parallel program could achieve similar improvement for the more difficult dataset *puzzles5_forum_hardest_1905_11+*, with a speedup of up to 5.67x. The detailed results are shown in Table 5.

4.4 Breakdown Analysis

Despite significant improvement of parallelized program, a major concern is that the parallel efficiency, i.e., speedup divided by number of threads used, is quite low – it could be as low as 0.29 when 16 threads are used. A reasonable assumption of such issue would be that the parallelizable computation does not dominate whole solving program. Concretely, since the time used for initialization and splitting Sudoku puzzles must be serialized, the overall improvement would be limited by its execution overhead ratio.

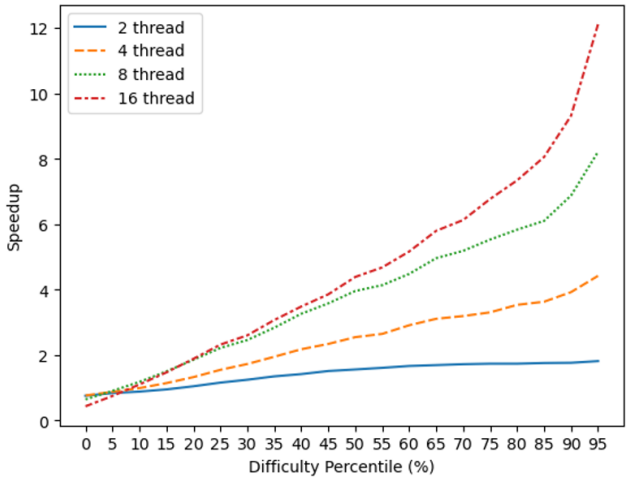


Figure 6. Breakdown analysis of dataset puzzles2_17_clue speedup

To verify it, we split the Sudoku puzzles into twenty groups according to their serialized solved time and evaluates each group's corresponding speedup performance, as shown in Figure 6. Obviously, the harder the puzzles are, the better the parallel program performs. For programs with 4 and 8 threads, there exists almost a linear improvement for the top ten percent of most difficult puzzles.

4.5 Ablation study

According to our methodology, each Sudoku puzzle is divided into several independent sub-problems for parallel computation. There exists a trade-off when setting the number of branches. On one hand, the number of branches directly affects the performance of program's first half part. Increasing branches requires more splitting operations and may degrade the program runtime. On the other hand, such granularity also affects the balance of workload, which determines the overhead of each thread during the second half of program. To select the best hyperparameter, the experiment below examines the relationship between the number of branches and the program performance.

As the Table 6. have shown, where the ratio refers to the number of branches divided by the number of threads, the program achieves best performance when number of branches equals to two to three times num. of threads. Similar trends apply to experiments using other dataset or solvers mentioned in previous chapter.

By the same token, the scheduling strategy: *static, dynamic, auto* or *guided*, specified in OpenMP would affect the performance too. Experiment shows that the default auto strategy performs better than all other methods.

Table 5. Parallelized program results of *puzzele5_forum_hardest_1905_11+* dataset

Serial Solver	No. of thread	Average Solved Time	Search Space Size	Relative Performance	Parallel Efficiency
Crook's Pruning	1	2.69 ms	346	1x	1
	2	1.97 ms	233	1.36x	0.68
	4	0.96 ms	112	2.80x	0.70
	8	0.67 ms	70	3.4x	0.50
	16	0.47 ms	41	5.67x	0.35
Crook's Pruning + cross-hatching	1	1.93 ms	278	1x	1
	2	1.20 ms	155	1.61x	0.80
	4	0.66 ms	84	2.91x	0.73
	8	0.43 ms	47	4.55x	0.57
	16	0.37 ms	28	5.21x	0.33

Table 6. Relationship between number of branches and performance

No. of thread	Ratio = 1	Ratio = 2	Ratio = 3	Ratio = 4
2	0.73 ms	0.54 ms	0.54 ms	0.54 ms
4	0.38 ms	0.34 ms	0.38 ms	0.34 ms
8	0.25 ms	0.25 ms	0.23 ms	0.22 ms
16	0.20 ms	0.17 ms	0.18 ms	0.18 ms

5. Related work

Shivanshu Gupta [6] splits the Sudoku problem in a DFS manner and solves them parallelly. The author applies the recursive method to validate all possible digit combinations of the unknown cells. Although the algorithm is compact and intuitive, the performance is poor due to underutilization.

Shawn Lee [1] similarly splits the search space by creating threads and filling possible values to unknown cells. He adopts a master thread to combine all results returning from workers after each iteration. However, the messages between threads are costly since it requires coordination between masters and workers.

The most similar work to ours is *Sruthi Sankar* [3]'s study. The author also splits the puzzle into multiple independent sub-problems and adopts *Crook's algorithm* [2] to improve the performance. He maintains a shared data structure to aggregate the results of sub-problems from all threads.

Such communication and synchronization overhead between different threads significantly reduce parallelization gains. Correspondingly, our implementation requires communication once only when initiating a thread. In addition, we adopt an efficient serial *Crook's solver* and a cross-hatching selection to solve each subproblem independently.

Conclusions

In this project, we parallelize the optimized serial Sudoku solver and achieves at most 5.67x speedup with 16 threads. We verify that serial-based optimization is necessary to solve Sudoku efficiently, while parallel optimization could push the performance further by utilizing more computation resource. Also, compared to the easy Sudoku puzzles, hard Sudoku puzzles gains more significant improvement from the parallel program.

References

[1] S. Lee, "Efficient Parallel Sudoku Solver via Thread Management & Data Sharing Methods," https://shawnjzlee.me/dl/efficient-parallel-sudoku.pdf
[2] Crook, J. F. "A pencil-and-paper algorithm for solving Sudoku puzzles." Notices of the *AMS*, pp. 460-468, 2009.
[3] A. Chiu at el. "Parallelization of Sudoku," http://individual.utoronto.ca/rafatra-shid/Projects/2012/SudokuReport.pdf
[4] S. Sankar, "Parallelized sudoku solving algorithm using OpenMP," https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Sankar-Spring-2014-CSE633.pdf
[5] GeeksforGeeks, "Sudoku using cross-hatching with backtracking", https://www.geeksforgeeks.org/sudoku-backtracking-7/
[6] Shivanshu-Gupta, "Parallel-Sudoku-Solver," https://github.com/Shivanshu-Gupta/Parallel-Sudoku-Solver
[7] R. Nieuwenhuis at el. "Solving SAT and SAT modulo theories: from an abstract Davis—Putnam—Logemann—Loveland procedure to DPLL(T)," in *Proc.* of *ACM*, pp. 937-977, Nov. 2006