Sudoku Solver

High Performance Programming – Individual Project

Ting-Hsuan Lien
dept. Information Technology
Uppsala Universitet
Uppsala, Sweden
sophielien09@gmail.com

Abstract

This report presents the implementation and performance evaluation of a parallel Sudoku solver using OpenMP. The core algorithm is based on recursive backtracking, enhanced with validation steps and parallel task distribution. The project explores the efficiency and limitations of parallel recursion in solving large-scale Sudoku puzzles.

I. Introduction

Sudoku [1] is a logic-based combinatorial puzzle that has become a widely recognized benchmark for constraint satisfaction problems. Solving large Sudoku puzzles, such as 36x36 boards, can take several minutes using a standard serial implementation due to the exponential nature of the search space. As the board size increases, the computational demand grows rapidly, making it impractical to rely solely on sequential algorithms.

To address this challenge, efficient techniques are required to accelerate the solving process. This project leverages two key strategies to improve performance: bitmasking [2] and parallelization using OpenMP. Bitmasking allows for compact and efficient representation of possible values, significantly reducing validation overhead. Meanwhile, OpenMP enables parallel exploration of the solution space, particularly effective in the early stages of the recursive backtracking algorithm where branching is abundant.

II. Problem description

The goal of this project is to create a fast Sudoku solver that can handle large puzzle boards, such as 25x25, 36x36 and 64x64 grids. The input files are in binary format. The first number tells us the base size b (for example, $b = 5$ means the board size is 25×25). The second number is the side length $n = b^2$, followed by $n \times n$ numbers that represent the board values, given in row order.

The task is to fill in all the empty cells so that each number from 1 to n appears exactly once in each row, each column, and each $b \times b$ box. This is a hard problem when the board size is large, because the number of possibilities grows very fast.

To make the solver fast and efficient, the project uses:

- Backtracking to try different number combinations.
- A validation function to quickly check if a guess is allowed.
- Bitmasking to represent the possible values in a fast and memory-efficient way.
- OpenMP to add parallelism and allow the program to use multiple CPU cores.
- A shared flag to stop other tasks when a correct solution is found.

The program needs to manage memory and tasks carefully. Tasks should not be too small, or there will be too much overhead. Also, different tasks should not work on the same board at the same time to avoid conflicts.

III. Solution Method

The experiments in this project are divided into four versions of the solver, each adding new optimizations or improvements. These versions are:

1. Naive Version

The first version is a basic backtracking algorithm. It works by trying every possible value for each empty cell, and uses a validation function to check whether the current value is allowed in its row, column, and box.

The core idea is shown in the pseudocode below:

```
function ValidateBoard(board, x, y)
    if DuplicateInRow(board, x) return false
    if DuplicateInCol(board, y) return false
    if DuplicateInBox(board, x, y) return false
    return true

function Solve(board, unAssignInd, N_unAssign)
    if N_unAssign == 0
        return true
    index = unAssignInd[N_unAssign]
    for val in [1, BoardSize]
        board[index.x][index.y] = val
        if ValidateBoard(board, index.x, index.y)
            if Solve(board, unAssignInd, N_unAssign - 1)
                return true
    board[index.x][index.y] = 0
    return false
```

2. Parallel Version

This version adds parallelism using OpenMP. Since each guess in the first level of recursion is independent, we can create parallel tasks for each value. However, care must be taken to avoid multiple tasks working on the same board at the same time. To solve this, each task uses a separate copy of the board. Once a valid solution is found, all other tasks are stopped using a shared flag.

3. Bitmask Version

In this version, the board is optimized using bitmasks to track the numbers already used in each row, column, and box. For a board of size $n \times n$, we use three arrays of 64-bit integers: `board_row`, `board_col`, and `board_box`, each representing the state of used values.

To insert a value into the board and update the bitmasks, we use bitwise OR operations.

```
// Mark value in bitmasks
uint64_t mask = 1ULL << (val - 1);
board_row[x] |= mask;
board_col[y] |= mask;
board_box[(x / base) * base + y / base] |= mask;
```

To remove a value (i.e., during backtracking), we use bitwise AND with the complement of the mask.

```
// Remove value during backtracking
board_row[x] &= ~mask;
board_col[y] &= ~mask;
board_box[(x / base) * base + y / base] &= ~mask;
```

Validation is done by checking if the bit is already set in any of the corresponding bitmasks.

```
// Validation function using bitmasks
bool validate_board_bitmap(uint8_t* board, uint64_t* board_row,
                           uint64_t* board_col, uint64_t* board_box,
                           uint8_t x, uint8_t y, uint8_t val) {
    uint64_t mask = 1ULL << (val - 1);
    if (board_row[x] & mask) return false;
    if (board_col[y] & mask) return false;
    if (board_box[(x / base) * base + y / base] & mask) return false;
    return true;
}
```

Using bit operations greatly reduces the time required for value checking and updates, especially in large boards. Since each check is just a few CPU instructions, this approach significantly speeds up the recursive solving process.

4. Bitmask + Parallel Version

This version combines the two previous improvements: the board uses bitmasks for fast validation, and OpenMP is used to parallelize the recursive backtracking.

IV. Experiments

A. Experimental Setup

All experiments were run on a Windows 11 machine using WSL (Windows Subsystem for Linux). The CPU used was an AMD Ryzen 5 7535HS with Radeon Graphics. I tested four different solver versions:

- **Naive:** Basic recursive backtracking.
- **Parallel:** Naive version with OpenMP parallelization.
- **Bitmask:** Bitmask optimization without parallelism.
- **Bitmask + Parallel:** Combines bitmask and OpenMP.

Each version was tested on 25×25, 36×36, and 64×64 boards. The naive version was excluded from the larger boards, as it could not complete within 10 minutes. Both parallel versions were executed using 12 threads.

B. Performance Results

Version	**25x25 Time (s)**	**36x36 Time (s)**	**64x64 Time (s)**
Naive	0m0.082s	-	-
Parallel	0m0.010s	0m2.248s	4m25.088s
Bitmask	0m0.008s	0m0.175s	0m10.949s
Bitmask + Parallel	0m0.016s	0m2.405s	4m55.577s

TABLE I
EXECUTION TIME FOR EACH SOLVER VERSION (REAL TIME).

C. Observations

The simplest version, naive backtracking, was not able to solve larger boards within a reasonable time. However, after adding parallelism, the performance improved significantly on larger inputs.

The most efficient version overall was the bitmask implementation. It was able to solve the 64×64 board in just about 10 seconds.

Interestingly, combining bitmask with parallelism resulted in slower performance compared to using bitmask alone. This was unexpected; in fact, the bitmask + parallel version was even slightly slower than the parallel-only version.

One likely reason is that the overhead from creating and managing tasks in OpenMP offsets the benefits of parallel execution, especially since bitmasking already reduces the search space significantly. Additionally, the bitmask version requires extra memory to store three separate masks (for rows, columns, and boxes). When parallelized, this leads to frequent copying and allocation of large arrays, which may introduce significant overhead and memory pressure, further impacting performance.

V. Conclusion

The experimental results were somewhat different from what was originally expected. The bitmask-only version achieved excellent performance, while the addition of parallelism actually caused a noticeable slowdown.

I believe this is primarily due to two factors: the overhead of copying large bitmask-related data structures in memory, and the cost of creating and managing tasks using OpenMP. While the parallel version significantly improved upon the naive implementation, the bitmask + parallel version, which followed the same logic, did not provide further benefit. In fact, its runtime was slightly worse than the parallel-only version.

This observation can be justified by looking at the computational load. In the naive version, checking whether a value already exists in the same row, column, or box requires $O(n)$ operations per check. In contrast, the bitmask version reduces each check to a single bitwise operation, effectively lowering the validation cost to $O(1)$. As a result, most of the computational work is already eliminated, leaving little room for further improvement through parallelization.

The slightly worse performance of the bitmask + parallel version compared to the parallel-only version may be explained by the overhead of allocating and copying the additional bitmask data structures, which are needed for each parallel task.

Potential Improvements: One possible optimization could be to apply a pruning strategy based on the bitmask structure. Instead of filling in empty cells in order, the solver could prioritize the cell with the fewest remaining possible values. This heuristic may significantly reduce the number of recursive calls required. However, such an approach could also result in unpredictable performance depending on the initial state of the puzzle, making it harder to evaluate its general effectiveness.

Source Code: The complete implementation and test data are available on GitHub: https://github.com/sophie8909/High-performance-programming

References

[1] W. Contributors, "Sudoku," Mar. 2025, page Version ID: 1278717121. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Sudokuoldid=1278717121

[2] GeeksforGeeks, "Solving sudoku using bitwise algorithm," 2024. [Online]. Available: https://www.geeksforgeeks.org/solving-sudoku-using-bitwise-algorithm/

Appendix

During the development of the project, the following AI tools were used:

- **GitHub Copilot (VSCode extension):** Used for code autocompletion. This was helpful for speeding up the implementation process, especially for repetitive structures.
- **ChatGPT (OpenAI):** Used for debugging support, code explanation, and discussing ideas for performance improvement. Additionally, ChatGPT was used to help formulate and polish the content of this report.

All content generated with the help of AI tools was reviewed, tested, and edited to ensure correctness and relevance. No sensitive data was uploaded to any AI system.