Individual Project:
Large Sudoku Solver

Linjing Shen

2024/3/5

Contents

1 Introduction

Sudoku, originating from Switzerland in the 18th century[2], is a numerical logic game that most commonly takes the form of a 9×9 grid, known as the board, subdivided into nine 3×3 squares called boxes. Some of the cells in the grid are already filled with numbers from 1 to 9, and players must deduce the numbers for the remaining empty cells based on the given clues. The rules dictate that each column, row, and box must contain unique numbers from 1 to 9. The difficulty of Sudoku puzzles varies based on the amount of given information provided. Strictly speaking, only puzzles with a unique solution are considered Sudoku.

Large Sudoku puzzles are a derivative of standard Sudoku, featuring larger grids and more constraints. They typically have an $n^2 \times n^2$ board with $n \times n$ boxes. Common examples for large Sudoku include 16×16 and 25×25 boards. Solving these high-order puzzles using computers can be time-consuming, especially as n increases[1]. Therefore, the objective of this project is to utilize parallelization techniques to implement a parallel Sudoku solver algorithm, reducing the time required to solve large Sudoku puzzles.

2 Problem description

The Sudoku solver adopts the backtracking algorithm, which is a deep search algorithm, following the trial-and-error approach to attempt step-by-step problem-solving. When it discovers that the current step's answer cannot lead to an effective and correct solution, it cancels the calculations of the previous step or even several steps, and then tries to find the solution through other possible stepwise approaches again. The backtracking method is typically implemented using the simplest recursive approach, and in the worst-case scenario, it can lead to exponential time complexity.

When aiming for parallelism, we can view each guess as an independent parallel task, but two tasks cannot work on the same Sudoku simultaneously. When a solution is found, no more tasks should be issued, and previous tasks should be canceled. Tasks can be created internally within tasks to generate enough workload for threads in the task queue, but creating overly small tasks should be avoided to reduce memory and task overhead.

3 Solution method

3.1 Serial method

When optimizing the serial Sudoku solver code for the first time, I mainly considered the following aspects. On the one hand, I utilized more efficient data structures to represent the Sudoku grid, using a one-dimensional array instead of a two-dimensional one. This improves memory access efficiency and reduces code complexity. Each cell's index in the one-dimensional array is calculated using the formula ($row \times BoardSize + column$), enabling faster cell access. On the other hand, while reading the Sudoku problem, I simultaneously saved the positions and quantities of empty cells. This reduces the need for subsequent searches and validations of the entire Sudoku grid, thus enhancing solving efficiency.

When I attempted to parallelize the algorithm using a sequence of sub-Sudoku tasks, I found that splitting the large Sudoku board significantly improves the efficiency of the execution even without performing any parallel operations. By breaking large Sudoku puzzles into smaller sub-puzzles, the algorithm can focus on solving the smaller parts independently. This division of labour allows for a better use of resources, which leads to faster solving. In addition, the continuity of the algorithm benefits from being able to tackle smaller parts of the puzzle more efficiently, as this reduces the complexity of the problem at hand.

3.2 Parallel method

The parallel optimization of the Sudoku solver is implemented using OpenMP. In the initialization stage, the code creates a BoardQueue structure for each Sudoku problem, containing an array to store pending Sudoku problems, their sizes, and solving statuses. During the solving stage, the program uses the OpenMP directive `#pragma omp parallel for` to parallelize solving sub-tasks of Sudoku problems. Each thread in the parallel region attempts to solve one sub-task of a Sudoku board. If successful, the thread saves the solution in the solutions array and marks the Sudoku grid as solved. If unsuccessful, the code calls `PartionSolve` to further partition the sub-task and rearrange the problem queue for subsequent solving attempts. Moreover, to ensure balanced workloads among threads, the program uses the `schedule(dynamic)` directive for dynamic scheduling. This dynamically allocates tasks based on the actual workload of threads during runtime, thereby improving overall parallel execution efficiency. Through these parallelization strategies, the Sudoku solver effectively utilizes multi-threaded parallel computing capabilities, accelerating the Sudoku problem-solving process, and ensuring balanced workloads among threads, thus optimizing solving performance.

The specific roles of some of the important functions in the parallel code are as follows:

- **ValidateBoard**: Validates whether a given number can be placed at a certain position on the Sudoku board, ensuring that the number does not repeat in the same row, column, or 3x3 subgrid.
- **PushBack**: Adds a Sudoku board to the end of the queue.
- **PopFront**: Removes a Sudoku board from the front of the queue.
- **PartionSolve**: Generates a series of subtasks for a Sudoku puzzle to be solved; it is the core of the entire parallel code.
- **Shuffle**: Randomly shuffles the queue of Sudoku boards to reorder the solving sequence, preventing repeated solving attempts of incorrect subtasks.
- **SolveSudoku**: Recursively solves the Sudoku puzzle by attempting to fill in numbers through trial and error until a valid solution is found.

3.3 Accessorys

In order to implement and verify the functionality of the Sudoku solver, a corresponding Sudoku generator is needed to generate legal Sudoku for solving, and a Sudoku detector also be needed to verify that the Sudoku results are correct. So I developed a Sudoku generator and a Sudoku detector to verify that the Sudoku results are correct.

There are two ways to generate Sudoku puzzles: one is to dig holes in a complete Sudoku board, and the other is to generate an empty board and fill in the numbers step by step. The method I selected is the first one. First, I obtained a complete and legal Sudoku board, then randomly shuffled the rows and columns and removed a certain number of numbers to generate the Sudoku puzzle. Which used a backtracking algorithm to verify that the generated Sudoku puzzle had a unique solution. If more than one solution exists, more candidate numbers need to be added to the board to ensure the uniqueness of the solution.

The test code implements a simple Sudoku solution detector. It checks each row, column, and sub-block of $n \times n$ of the Sudoku matrix to make sure that there are no duplicate numbers, thus verifying the correctness of the given Sudoku puzzle.

4 Experiments

4.1 CPU model

For subsequent performance evaluations, I used arrhenius.it.uu.se. Here is the system hardware configuration and processor information:

- **Architecture:** x86_64
- **CPU op-mode(s):** 32-bit, 64-bit
- **Address sizes:** 40 bits physical, 48 bits virtual
- **Byte Order:** Little Endian
- **CPU(s):** 16
- **On-line CPU(s) list:** 0-15
- **Vendor ID:** GenuineIntel
- **Model name:** Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
    - **CPU family:** 6
    - **Model:** 26
    - **Thread(s) per core:** 2

  - **Core(s) per socket:** 4

- **Socket(s):** 2

4.2 Correctness testing

The Sudoku detector provides the functionality to validate the accuracy of Sudoku solutions. It not only checks the correctness of each Sudoku result but also verifies if the generated number of Sudoku solutions matches the expected count. All Sudoku solutions provided by the Sudoku solver are stored in a file named ”sudoku_solutions.txt”. The detector reads this file and tests each Sudoku solution for legality. If a solution is valid, the detector proceeds to check the next solution; if invalid, it marks the detection of at least one invalid solution. Finally, based on the detection outcome, the detector outputs the corresponding message: if all Sudoku solutions are valid, it prints ”All [number] Sudoku solutions are valid.”; otherwise, it outputs ”One or more Sudoku solutions are invalid.”. Through this process, the program is able to verify the accuracy of Sudoku solutions and provide appropriate feedback.

When facing a 16x16 Sudoku board, with 10,000 Sudoku puzzles as an example for testing, the serial code takes 0.479717 seconds to execute, while the 4 threads parallel code takes 0.121107 seconds. The correctness test shows ”All 10,000 Sudoku solutions are valid.” Similarly, when facing a 25x25 Sudoku board, with 10,00 Sudoku puzzles as an example for testing, the serial code takes 0.145688 seconds to execute, while the 4 threads parallel code takes 0.037497 seconds. The correctness test shows ”All 10,00 Sudoku solutions are valid.”

4.3 Performance

Table 1: Efficiency Comparison of Serial Codes

Board size	Board number	Serial (s)	Serial with task sequencing (s)	Speed up
9x9	1000	0.040439	0.017895	2.26
	10000	0.404966	0.149847	2.70
16x16	1000	0.134239	0.045087	2.98
	10000	1.154057	0.454617	2.54
25x25	1000	0.311416	0.132246	2.35
	10000	3.164957	1.312254	2.41

Compared to the original serial code, adopting a one-dimensional array to accommodate Sudoku data and record the number and position of empty cells only offer marginal improvements. However, experiments revealed that setting the thread number as 1 to conduct serial testing on the parallel code resulted in approximately a 2.5-fold enhancement. Table.1 presents the efficiency of the serial code after initial optimization and the serial code using task sequencing. It compares the runtime, in seconds, and the calculated speedup for Sudoku boards of sizes 9x9, 16x16, and 25x25, as well as puzzle sets of 1000 and 10000.

Similarly, Table.2 presents the execution times and speedups for three different board sizes: 9x9, 16x16, and 25x25, along with two different problem counts: 1000 and 10000, when processed serially and in parallel using 4 threads.

Table 2: Efficiency Comparison of Serial and Parallel Code

Board size	Board number	Serial (s)	Parallel with 4 threads (s)	Speed up
9x9	1000	0.011288	0.005548	2.03
	10000	0.171493	0.043544	3.94
16x16	1000	0.042557	0.012523	3.40
	10000	0.479717	0.121107	3.96
25x25	1000	0.145688	0.037497	3.89
	10000	1.289777	0.323374	3.99

We can observe that when dealing with large-scale Sudoku problems, parallel processing with four threads typically achieves nearly 4-fold speedup. However, for smaller-scale 9x9 Sudoku problems with a sample size of 1000, only a twofold improvement is achieved. Due to the uniform utilization of the same maximum number of calls to the subtask-splitting function for different data sizes in the code, there may be a performance loss when handling smaller datasets. This situation could be attributed to the substantial management and scheduling overhead caused by excessively fine-grained task subdivision, which is mismatched with the actual problem scale, thereby diminishing the efficiency of parallel processing. Attempts to reduce the maximum number of calls to the splitting function revealed that parallelization could achieve nearly a 4-fold improvement.

The acceleration of parallel code often depends on the number of threads and the proportion of the parts of the program that can be parallelized. Its calculation often involves Amdahl's Law, $S = \frac{1}{(1-P)+\frac{P}{N}}$, where P represents the proportion of the program that can be parallelized, and N represents the number of threads. When P equals 1, the formula calculates the expected speedup. Figure.1 and Table.3 compare the ideal speedup and the actual speedup with different number of threads when processing 1000 problems with 16x16 boards and 25x25 boards, where for each board specification, the top row of the table is the time in 10^{-1} seconds and the bottom row is the acceleration ratio.

Table 3: Parallelism speeds up with thread growth

Ideal Speedup	1	2	3	4	5	6	7	8	9	10
16x16	0.426	0.266	0.170	0.118	0.112	0.090	0.082	0.074	0.062	0.068
	-	1.605	2.514	3.620	3.803	4.716	5.188	5.756	6.934	6.313
25x25	1.257	0.750	0.470	0.379	0.262	0.259	0.217	0.198	0.192	0.185
	-	1.676	2.677	3.314	4.801	4.854	5.793	6.334	6.549	6.785

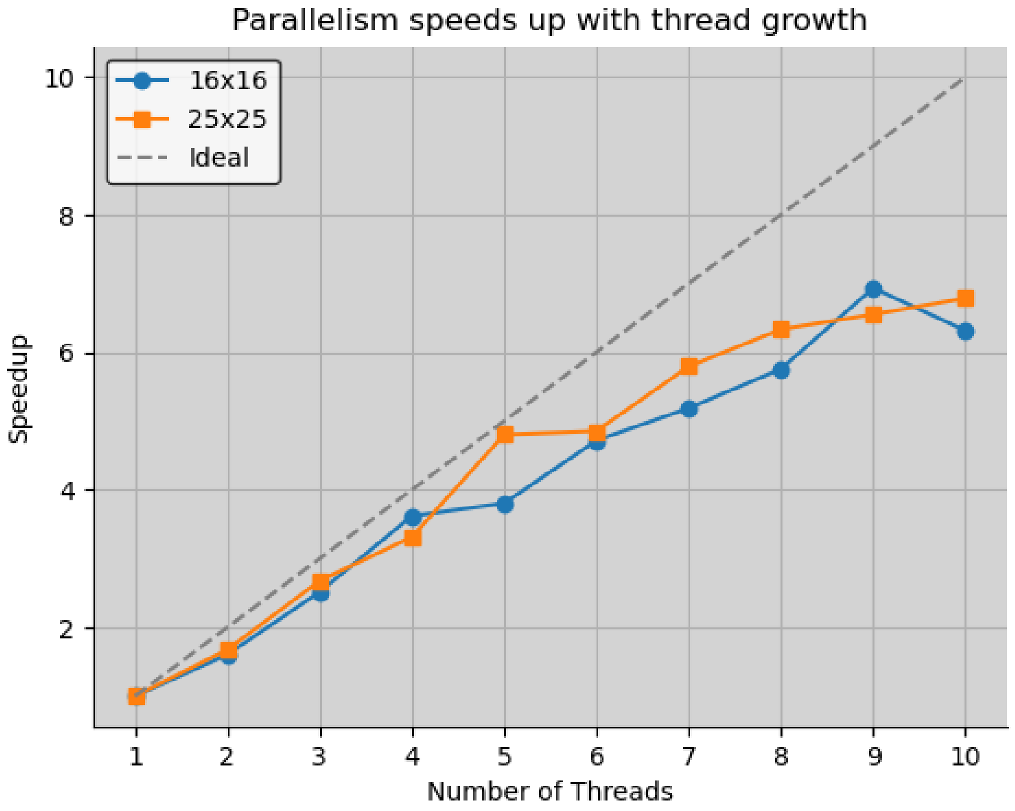

Figure 1: Parallelism speeds up with thread growth

It is evident that when the number of threads increases from 1 to 4, parallel processing achieves nearly ideal speedup. However, as the number of threads increases to 6, the rate of performance improvement slows down or no longer matches the expected ideal speedup ratio. This may be due to factors such as resource contention, management overhead, and Cache efficiency. When the number of threads exceeds 8, the processor running the program has only 8 physical cores and supports dual-threading. This means that when the number of simultaneous threads exceeds 8, two threads start running on the same core, resulting in inefficient utilization of resources and potential performance degradation.

Table 4: Compiler Optimisation for Parallel Code

	-	-O1	-O2	-O3	-O4	-Ofast
Time (s)	0.333980	0.145276	0.127269	0.111731	0.119273	0.115270

Table.4 illustrates the effect of different optimization levels on parallel algorithms during compilation. As the optimization level increases, there is a gradual decrease in the algorithm's execution time. Starting from the baseline without any optimization, significant reduction in execution time is observed after applying basic '-O1' optimization. Subsequently, higher optimization levels including '-O2' and '-O3' further decrease the execution time. However, optimization beyond '-O3' level, such as '-O4', appears to introduce some performance loss. Ultimately, under the performance-oriented '-Ofast' level, the execution time remains at a relatively low level.

5 Conclusions

The implementation of a parallel Sudoku solver algorithm has demonstrated significant improvements in solving large Sudoku puzzles. By leveraging parallelization techniques, the solver efficiently distributes the workload across multiple threads, reducing the time required to find solutions for high-order Sudoku boards such as 16×16 and 25×25.

The backtracking algorithm employed in the solver effectively explores possible solutions through a trial-and-error approach, canceling ineffective paths and backtracking when necessary. However, the serial implementation of this algorithm may become time-consuming as the puzzle size increases, especially for large $n^2 \times n^2$ boards. Parallelization addresses this challenge by dividing the Sudoku puzzle into smaller sub-puzzles, allowing independent processing of each segment concurrently.

However, this parallel code also has certain problems in that the program is prone to timeouts when the data set becomes large, and it is more prone to running timeouts than the serial code. The main reasons could stem from the different algorithms and data structures utilized in serial versus parallel programs, concurrency control overhead, uneven workload distribution, and issues related to resource contention. Parallel computing involves complex synchronization mechanisms that increase system performance overhead, potentially leading to performance degradation and timeouts.

Alternatively, the primary concern within the serial code may stem from the operations related to the ‘Board‘ structure, such as the copying operations and queue manipulation involving the movement of array elements, both of which substantially amplify time and space overhead. To enhance the code, one may explore utilizing pointer passing to mitigate the overhead associated with copying structures and adopting more efficient data structures in lieu of arrays. But after further testing, it has been found that using pointers is comparable in terms of efficiency and performance to using arrays. To further enhance the performance of the code, it may be necessary to adopt more powerful programming languages and frameworks, such as C++ and MPI.

Therefore, although parallelization techniques have brought significant performance improvements in solving large Sudoku puzzles, it is still necessary to consider algorithm complexity, data structure selection, and the implementation details of parallelization techniques in practical applications to ensure the stability and efficiency of the program on large-scale datasets.

References

[1] Shiyun Qiu, Yiqi Xie, Xiangru Shu, and Yuyue Wang. Parallelized giant sudoku solver, 2018.

[2] SwayDy. Sudoku in c, 7 2021.