Individual Project:
Large Sudoku Solver

Linjing Shen

2024/4/16

Contents

1 Introduction **1**

2 Problem description **1**

3 Solution method **2**

3.1 Serial method 2

3.2 Parallel method 2

3.3 Accessorys 3

4 Experiments **4**

4.1 CPU model 4

4.2 Performance 4

5 Conclusions **8**

1 Introduction

Sudoku, originating from Switzerland in the 18th century[3], 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 25×25 and 36×36 boards. Solving these high-order puzzles using computers can be time-consuming, especially as n increases[2]. 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. Furthermore, by adopting smaller data types to store small integers representing Sudoku grid cells, memory usage was optimized. Using `char` types effectively saves memory space as they require only one byte to store data, whereas `int` typically requires four bytes. This optimization strategy is particularly suitable for Sudoku problems, where the range of numbers in the grid is usually small. 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.

3.2 Parallel method

The program utilizes the OpenMP library to implement parallelization by adding OpenMP directives within key loops and recursive calls to create tasks and manage threads. In the `main()` function, the program first retrieves and sets the number of threads, `n_threads`, from the command-line arguments. Subsequently, the program establishes a parallel region using `#pragma omp parallel` and ensures that only one thread is responsible for task creation with the `#pragma omp single` clause. These tasks are then placed into a task pool for execution by idle threads within the thread group.

Within the `Solve()` function, task parallelism is implemented using task groups. The pivotal component is `#pragma omp task`, which generates a parallel task to attempt filling the Sudoku grid. Here, `firstprivate` is used to copy specific variables into each task, while `shared` ensures shared access to the global variable `solution`. Upon finding a valid solution, `#pragma omp cancel taskgroup` is used to cancel any pending tasks, thereby terminating unnecessary solving work prematurely. Additionally, `#pragma omp taskwait` is employed to wait for all created tasks to complete, avoiding premature termination due to erroneous solutions returning a value of `0` to the main function.

Furthermore, the parallel code calculates and outputs the program's execution time. It utilizes `omp_get_wtime()` to determine the actual duration of parallel execution and utilizes `clock()` to measure the overall CPU runtime.

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. It can be applied to Sudoku puzzles of various sizes by adjusting the macro definitions of `BoardSize` and `BoxSize`, along with modifying the file path and name for reading. All Sudoku solutions reported here have successfully passed validation tests, ensuring their correctness.

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 Performance

Firstly, for ease of testing, I attempted four different sizes of large-scale Sudoku puzzles and categorized them into three levels of difficulty: easy, medium, and hard, as shown in Table 1. When generating hard level Sudoku puzzles, especially for grid sizes of 49 and 64, it is often encountered that valid Sudoku puzzles cannot be generated. This is because removing more spaces results in fewer known numbers, increasing the potential solutions for the Sudoku puzzle and making it difficult to ensure the puzzle has a unique solution. Consequently, the generation difficulty increases as it becomes challenging for algorithms to find valid solutions.

Table 1: Sudoku Difficulty Setting

Size	Cells Number	Remove Count (r)		
		easy	medium	hard
25x25	625	<=60	60<r<=150	>150
36x36	1296	<=100	100<r<=300	>300
49x49	2401	<=300	300<r<=600	>600
64x64	4096	<=400	400<r<=1000	>1000

Transitioning to the discussion on code optimization, the adaptation from the original serial code to using a one-dimensional array for Sudoku data storage and tracking empty cell positions yielded only marginal improvements in performance. This optimization aimed to enhance efficiency and streamline data handling within the Sudoku solving algorithm, although the impact on solving time was limited.

Figure 1 and Table 2 illustrate the required solving time and speedup ratio when handling Sudoku problems of different sizes and difficulties with varying numbers of threads. First, the table shows the time required to solve the same type of Sudoku problem under different thread configurations. As the number of threads increases, the solving time for `25_hard` decreases gradually, indicating that this more difficult Sudoku problem benefits from parallel processing. On the other hand, `36_easy` and `36_medium`, which are relatively simpler and of medium difficulty, already exhibit fast solving times with a small number of threads, with further increases in thread count providing limited additional benefits. Similarly, `36_hard` shows significant initial improvement in solving time with parallel processing, but further gains diminish as the thread count increases. For `49_easy` and `49_medium`, the solving time decreases slightly with more threads, suggesting that the impact of parallel processing may be influenced by problem size and difficulty. `49_hard`, a challenging Sudoku problem, exhibits high initial solving times with fewer threads, but significant reductions in solving time are achieved by increasing the thread count, highlighting the importance of parallel processing for tackling difficult problems. For `64_easy` and `64_medium`, which are smaller-scale problems, the impact of increasing thread count on solving time is less pronounced.

Next, we can analyze the speedup of each Sudoku problem using a line graph. Speedup refers to the efficiency improvement ratio of parallel processing relative to serial processing, calculated as:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

Here, Serial Time represents the solving time with a single thread, and Parallel Time represents the solving time with multiple threads.

By examining the speedup results, we can observe that `25_hard` shows increasing speedup with thread count, indicating significant performance gains from parallel processing for this difficult Sudoku problem. `36_easy` and `36_medium` already exhibit high speedup with a small number of threads, but further increases in thread count yield diminishing returns. For `36_hard`, substantial speedup is achieved with fewer threads, but the rate of improvement diminishes as the thread count increases. `49_easy` and `49_medium` show modest speedup with increasing thread count, which is relatively small compared to their problem size. `49_hard`, a challenging problem, demonstrates notable speedup with additional threads, underscoring the importance of parallel processing. Larger-scale problems like `64_easy` and `64_medium` show minimal speedup, likely constrained by problem size and difficulty.

In summary, more difficult and larger-scale Sudoku problems benefit from parallel processing, leading to higher speedup. For simpler and medium-difficulty problems, speedup gains are relatively smaller and may already achieve fast solving times with fewer threads. Increasing thread count results in diminishing returns, suggesting that the effectiveness of parallel processing is influenced by problem size and characteristics.

The significant gap between actual speedup and ideal speedup may be attributed to the unique characteristics of parallelizing backtracking algorithms compared to traditional loop or data parallelism. Backtracking involves more complex state management and decision processes, requiring additional communication and synchronization overhead. Task distribution in backtracking algorithms can lead to uneven workload distribution among threads, further impacting the effective parallelization. These factors collectively contribute to the disparity between achieved parallel speedup and the ideal theoretical speedup.

In my algorithm, I used `omp_get_wtime()` and `clock()` for time measurements. `omp_get_wtime()` returns the actual wall-clock time for parallel execution, while `clock()` measures total CPU runtime. When dealing with challenging problems, the results from

Table 2: Parallelism time consumption with thread growth

Puzzles	Threads							
	1	2	3	4	5	6	7	8
25_hard	0.012	0.011	0.009	0.009	0.007	0.009	0.006	0.007
36_easy	0.001	0.002	0.002	0.002	0.002	0.002	0.002	0.002
36_medium	0.008	0.007	0.010	0.008	0.007	0.010	0.010	0.007
36_hard	0.531	0.482	0.273	0.301	0.210	0.177	0.151	0.148
49_easy	0.277	0.170	0.115	0.101	0.091	0.085	0.088	0.097
49_medium	0.010	0.010	0.011	0.008	0.011	0.011	0.011	0.012
49_hard	45.304	23.303	19.463	19.429	19.448	16.931	16.901	15.260
64_easy	0.012	0.012	0.011	0.013	0.012	0.010	0.012	0.013
64_medium	0.941	0.906	0.894	0.532	0.536	0.531	0.382	0.415

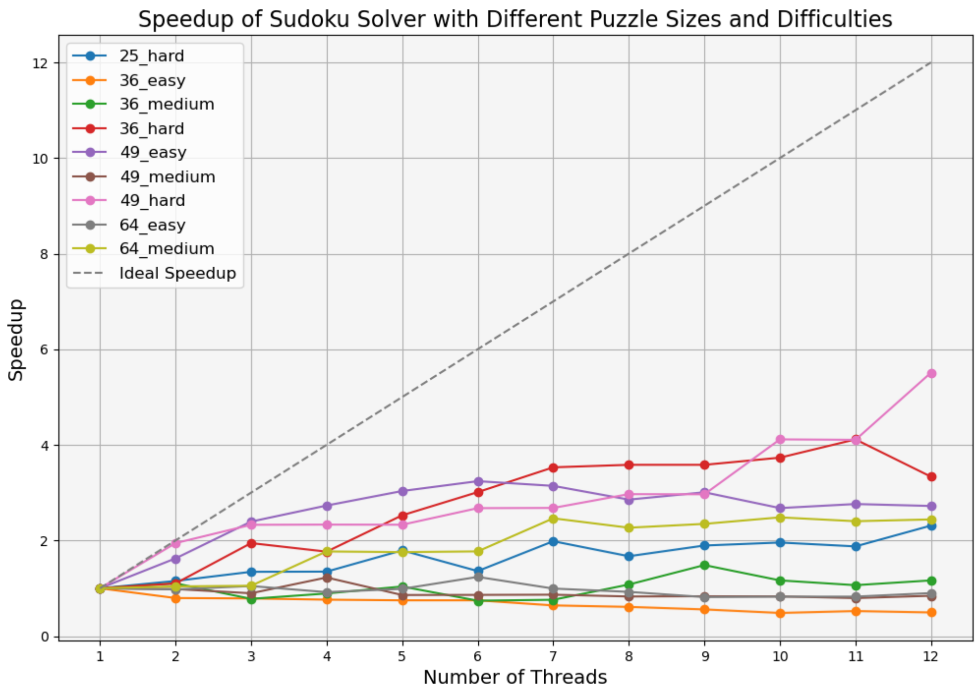

Figure 1: Parallelism speeds up with thread growth

`clock()` were approximately n times greater than those from `omp_get_wtime()`, where n is the number of threads. However, when handling Sudoku puzzles of simple and moderate difficulty, `omp_get_wtime()` and `clock()` returned almost identical times, indicating that parallel computation did not achieve the expected acceleration. Furthermore, by monitoring system resources (e.g., using the `top` command) and inspecting the output of `omp_get_thread_num()`, it was confirmed that parallel processing indeed utilized the specified number of threads.

Table 3: Compiler Optimisation for Parallel Code

	-	-O1	-O2	-O3	-O4	-Ofast	-Os
Time (s)	0.1556	0.1560	0.1108	0.1040	0.1494	0.1128	0.1270

Table.3 illustrates the effect of different optimization levels on parallel algorithms during compilation, which is applied to a 36x36 hard level Sudoku puzzle, where the number of threads is used to be fixed at 8. The execution time of the algorithm decreases progressively as the optimisation level is increased. 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. If focusing on reducing the size of the executable file, specifically optimizing for size with `-Os` can also lead to some performance improvements.

5 Conclusions

In previous considerations of parallelizing Sudoku solvers based on backtracking algorithms, it has often been thought that backtracking algorithms, which are depth-first search algorithms, are challenging to parallelize because they rely on stacks [1]. This dependency on a stack makes parallelization difficult as threads cannot effectively work on the same stack without encountering frequent competition for stack access [4]. However, utilizing the task parallelism features provided by the OpenMP library can effectively achieve parallelization of Sudoku solving based on backtracking algorithms. By employing task parallelization, possible subproblems can be replicated and assigned to multiple threads to handle independently, thus avoiding the scenario where multiple threads share the same stack. Each thread independently attempts to fill in a number, leveraging the computational power of multi-core processors to accelerate the entire solving process. This approach facilitates parallel execution of tasks and ensures reasonable sharing and synchronization of data, enabling multiple threads to work simultaneously and coordinate results accurately.

In testing, we have observed that the parallel performance is inferior to the serial performance, with Wall Time approaching the total CPU running time. This phenomenon may be attributed to the inherent nature of the Sudoku board, where it is challenging during backtracking to encounter error branches of sufficient depth. This can lead to premature termination of certain paths in the problem-solving process, as errors are detected at shallow levels. Even in a parallel setting, the exploration predominantly continues along the correct branch, with deep solutions potentially remaining undiscovered. Consequently, tasks are unable to fully leverage parallel performance, with most of the time spent on single threads or a few threads, rather than utilizing all available parallel resources simultaneously. As a result, the Wall Time almost equals the CPU running time. Additionally, due to parallel overhead, multi-threaded approaches often consume more time than single-threaded approaches. Furthermore, simpler Sudoku problems are more likely to exhibit this behavior, as each cell typically has obvious answers with few error-prone branches and shallower depths.

References

[1] huaminghuangtw. Parallel sudoku solver, 2021.

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

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

[4] vduan. parallel sudoku solver, 2015.