Uppsala University

High Peformance Programming

Project

A Parallel Sudoku Solver

Author:
Love Leoson

September 28, 2024

Contents

1 Introduction

Sudoku is a game where one is presented with a board of, in the classical case, size 9x9. In this case there exists 81 cells where some are filled with numbers between one through nine and others are empty. The board is further divided into nine so called minigrids with nine cells in each of them. The goal is to fill in each empty square with a number under a set of rules. The rules are that in each row, column and minigrid each number between one and nine can only appear once. The board sizes can vary. Some other instances are 16x16 with minigrid of size 4x4 and 25x25 with minigrid size of 5x5. Some Sudoku boards might be difficult to solve which could result in the need of a program that solves them.

2 Problem Description

The goal of this project was to write a C script that solves a Sudoku board of varying sizes. The program has its' focus on solving boards of size 9x9, 16x16 and 25x25 but can also be used to solve simple Sudokus of larger sizes. When the program was functioning properly it was to be optimized using serial optimization techniques and subsequently parallelized. The choice of algorithm will greatly impact the performance of the program. There exists several different algorithms to solve a Sudoku board. On such example is the classic depth-first search brute force backtracking where numbers are guessed and presumed to be accurate. The search then continues with recursion and fills in new numbers and if a number assigned produces an invalid board the search backtracks to a valid board and starts over with a new number. The good part of this algorithm is that given enough time a solution is guaranteed to be found. The negative part is the execution time. Since numbers are guessed there are a lot of different combinations which must be tested which can cause a very long execution time. The performance is also severely impacted by the size of the board [1].

Another method is the Stochastic search where numbers are randomly assigned to cells, the number of errors are calculated and then the cells with most errors are shuffled around until no errors exist. This method can cause quick solutions but is very luck-based [1]. A final example of a solution algorithm is the process of logical operations. This method uses the rules of the Sudoku game to fill in as many cells as possible. The positives of this method is that when it works it is very quick but the issues are that the solution is not guaranteed. For instance in difficult Sudokus one must sometimes guess a number to continue, in these boards this method will not find a solution.

3 Solution Method

Although a Stochastic search method to solve a Sudoku board can yield good result and fast execution times this method was deemed to not be suitable for this project. This is due to possible issues with parallelization. For a parallel solution the brute force backtracking is a much better choice, but the problem of the long runtime still persists. To combat the performance issues that come with using backtracking the solution method of using logical operations can be implemented first to fill in as many cells as possible. If then a solution still hasn't been found brute force backtracking is used to fill in the remaining

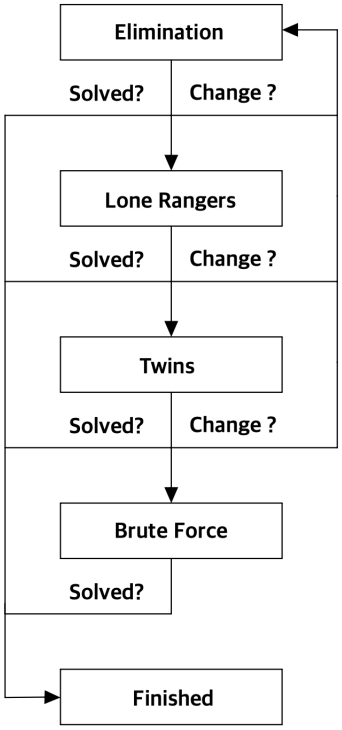

Figure 1: Pipeline of the Algorithm

cells. This way both the cons of backtracking and logical operations are combated while keeping the pros of both methods. The algorithm can therefore be split into two parts: logical operations and brute force backtracking. The algorithm took inspiration from a Sudoku solver by Ali Tarhini [2]. The pipeline can be viewed in figure (1).

3.1 Logical Operations

As before discussed the method of using logical operations has its' origin in using fundamental rules to figure out what value a cell must have and assigning this. The different operations to decide what value a cell must have can be split into three parts, elimination, lone rangers and twins. The pipeline of the method follows a hierarchical structure where the logical operation that produces the most changes is first and whenever a change is made the whole method is started over at the first logical operation. If a logical operation cannot find any changes the method moves on to the next operation.

3.1.1 Elimination

This is the most simple logical operation and is also the one that leads to the most changes. This operation involves checking each cell and locating cases where a cell only has one possible value and assigning this value to the cell.

3.1.2 Lone rangers

In Sudoku a cell has to be the only cell in its' row, column and minigrid with a specific value. Lone rangers refers to cells that have more than one possible value but is the only cell on its' corresponding row, column or minigrid with a specific value. In these cases

since no other cells in the corresponding row, column or minigrid can assume that value the lone ranger cell has to assume that value for the Sudoku to be valid.

3.1.3 Twins

Twins refer to a pair of cells in a row, column or minigrid that have the same exact set of possible values. For instance two cells both have possible values 1 and 2. In these cases both of these cells must have 1 or 2 as their value. This means that no other cells in that row, column or minigrid can attain these values meaning these two numbers can be removed as possible numbers for the rest of the cells.

3.2 Brute Force

The brute force part of the algorithm functions as previously describe under `Problem Description`. A cell is chosen and all of the cells possible values are tested. In each of these tests there is a recursive call that assumes the previous guess is correct. This recursive depth-first search then continues until the Sudoku board is either solved or an invalid board is found. If an invalid board is found the search backtracks and tries new values.

3.3 The Code

The code represents the Sudoku board with a dynamically allocated 2D array where each index represents a cell and its' value the value of the cell (0 for empty cells). The possible values of each cell is also represented by a dynamically allocated 2D array with each index containing a bitmask. A bitmask is an integer of varying size which can contain a lot of information in a very effective manor. The bitmask can represent values from a varying range depending on its size in the space of a single integer. The way this is achieved is that the bitmask contains a boolean one for numbers that exist within the bitmask and zeroes for numbers that do not exist. An example of a bitmask only containing the number zero would be `1000....`. Since a normal integer in C has 32 bits values between 0-31 can be represented. To access these numbers a for loop iterating through values from 0-31 has to be applied where that index of the bitmask is 1 or 0. This is enough to solve a Sudoku board of size 25 but for larger board sizes the integer size hast to be increased to `int long long` to be able to solve that Sudoku board. In this way the possible values of each cell can be kept track of.

The code then reads the Sudoku board to solve as a csv file and assigns the values of the cells to the 2D array of the board. The possible values of each cell is then calculated and assigned as bitmasks to the 2D array of the possible values. These values are calculated by iterating through all cells of the board, studying the row, column and minigirid that cell is part of and assigning the values that do not exist in either of the row, column or minigrid as possible values for that cell. The board is then solved using the above described algorithm. The logical operations and the brute force are implemented as functions.

3.4 Optimization

To ensure the code operates as fast as possible some optimization was done. The use of bitmasks was one of these optimization techniques used. Since the possible values of

the cells are constantly being updated and studied these need to be stored as efficient as possible. Inefficient storage can severely slow the program down. Bitmask offers a very efficient way to store a lot of information while using very little space. All operations that are done one the bitmask are very fast since these are bit operations. Compared to using a standard approach of an array containing the possible values this is a lot more efficient, especially so since using an array also means the 2D array containing possible values must become a 3D array further complicating the issue. The drawback of bitmasks is that they are limited to the number of values they can represent by the bits of an integer. A normal integer has 32 bits which means Sudoku boards of size up to 25x25 can be solved. This integer can be changed to a `int long long` which can be used to solve boards up to 49x49 [3].

The logic of the logical operations were then studied to identify unnecessary work to further improve performance. One of these instances were found in the `lone_rangers` function. Here the way lone rangers are found is that a temporary array called count is created and subsequently each row, column and minigrid are iterated through. The possible values that exist within the row, column or minigrid are then represented by increasing the index corresponding to that number of the count array. This way the number of occurrences of a number could be kept track of. Then another loop was performed over the count array to identify cases when a number only appears once. Then another loop iterates through the row, column or minigrid again to identify which cell should be assigned this value. This last loop is unnecessary, the work of iterating through the row, column or minigrid has already been done. Another temporary array was created called `pos` which when the count array was calculated the pos array was used to keep track of the index corresponding to the number studied. This way when an instance of a lone ranger was found the index where this number should be assigned could easily be acquired through lookup in an array.

The brute force function was then studied to identify where this part could be optimized. A big part of why the brute force is so time consuming is the fact that a lot of unnecessary work has to be done to identify which numbers are inaccurate for each cell. This is due to each number being tested. A way to reduce the execution time is to increase the odds that the first numbers tested are correct. This can be done by first dealing with the cells that have fewer possible values. In this way the number that is first tested has a higher chance to be the correct number and the performance will increase. This was achieved by implementing a function that identifies which index has the least amount of possible values and this cell will be dealt with first. Since brute force uses recursion every cell will still be checked. The negative part of this method is that it does require an iteration over the entire board to identify the cell with least amount of possible values each brute force function call. However, the positives out way the negatives. Another change that can be made is incorporating the elimination logic of the logical operations into the brute force function. In the case of only one possible value we know this value is correct so it can be assigned with certainty. This in combination with the fact that the cells with lowest amount of possible values are dealt with first means that every case of only one possible value will be dealt with immediately offering a lot of potential speedup.

After these changes were done the program was ran with the `operf` command and the `opreport --callgraph` was studied. It was determined that the majority of the time

was spent in the `update_possible` function, the function that calculates the possible values of each cell. The way this function works is that the entire board is iterated through and each row, column and minigrid corresponding to each cell is checked to determine the potential values of that cell. This is a correct approach and a necessary one when no possible values of cells exists. The problem is that the function is called each time a change is made. This will lead to a lot of iterations and a lot of work and more importantly, a lot of unnecessary work. A much better approach is to only update the cells affected by a change on the board. When a cell is changed its only the cells in the row, column and minigrid of the changed cells that are impacted. The affected cells do not need to have their possible values recalculated either. The value assigned to the changed cell can simply be removed from the affected cells. This change is a massive time saving change. In the example of solving a nxn Sudoku board, before each time a cell was changed an iteration through the whole nxn board had to be done, In that iteration for each cell another three iterations of size n had to be performed. This is a complexity of $\mathcal{O}(n^3)$. This is especially costly in the brute force function where many changes are made since each value has to be checked. The updated approach simply has a complexity of $\mathcal{O}(n)$.

3.5 Parallelization

Parallelization of code, if the code is able to run in parallel, can cause significant speedup. As previously discussed the first part of the algorithm, the logical operations, is already very fast when it works. This means that this part of the program is not suitable for parallelization since the work is not big enough to cause a speedup. The brute force function is perfect for parallelization, however it is not a straightforward task to run in parallel since it involves recursive calls. In each brute force function call the cell with the lowest amount of possible values is studied. A naive idea is to parallelize the loop that iterates through the possible values for that cell. This would not cause much of a speedup since the overhead of creating the threads would happen in each recursive call. A much better approach is to define a parallel region outside of the brute force function and create tasks to run the code in parallel instead. The idea is that the different possible values of a cell is distributed across threads using tasks. These task perform the recursive depth first search and create nestled tasks themselves. The idea is that many different possible paths will be explored simultaneously, reducing the size of the search region drastically. One thing to watch out for is not creating too small tasks since the work in these tasks will be too small for any improvement to be seen. This was prevented by keeping track of the number of empty cells left and when this number falls under a certain threshold no more tasks are created. This threshold was chosen to be the size of the board divided by its size of its' minigrid. The reasoning being that the threshold somewhat scales off of the size of the Sudoku board while still keeping an appropriate value.

4 Experiments

To evaluate the correctness of the code two methods were used. The first was simply manually comparing a known solution of a Sudoku to the result of the program. This was done for small Sudokus to ensure the code operates correctly. The second method was an implementation of a function that checks if a produced Sudoku board is a valid solution. This function operates by checking for duplicates in each row, column and minigrid as well as zeroes in the board. If a board has no zeroes and no duplicates it is a

valid solution. These methods were able to verify that the code operates correctly. When a solution is found it is always a valid solution.

The performance of the program differs a lot depending on the nature of the Sudoku. Easy sudokus with many filled in squares have a vastly different execution time compared to that of difficult Sudokus with more empty squares. Since the times are so different they will be discussed seperately.

4.1 Easy Sudokus

The performance of the solution of easy Sudokus is very fast, almost instant. Here the board size does not impact the result much which can be viewed in table (1).

Board Size	Execution Time [s]
9x9	0.003
16x16	0.004
25x25	0.005
36x36	0.007
49x49	0.013

Table 1: Performance on easy Sudoku boards

These tests were ran on a single thread and since the work is almost instant no performance increase will be had when running the program in parallel. The overhead of the parallelization will reduce the performance more than the parallelization can increase.

4.2 More Difficult Sudokus

Although the program operates very well on easy Sudokus things become more difficult when working with computer generated Sudokus with less squares filled in. These computer generated Sudokus do not have unique solutions. Two different tests were done to validate the performance of the Sudoku solver on these types of boards. First the program was run only with the brute force part of the algorithm to get a better view on how the parallelization improves performance. The algorithm was then run on the same Sudoku with the entire algorithm to view the performance of the entire Sudoku solver. It was done this way to get a better understanding of the parallelization. The times were measured by running 10 independent runs on a very difficult computer generated 16x16 Sudoku board and computing the mean and standard deviation for different amount of threads. This was done since the execution time from run to run had some variance. The Sudoku that was solved is the following:

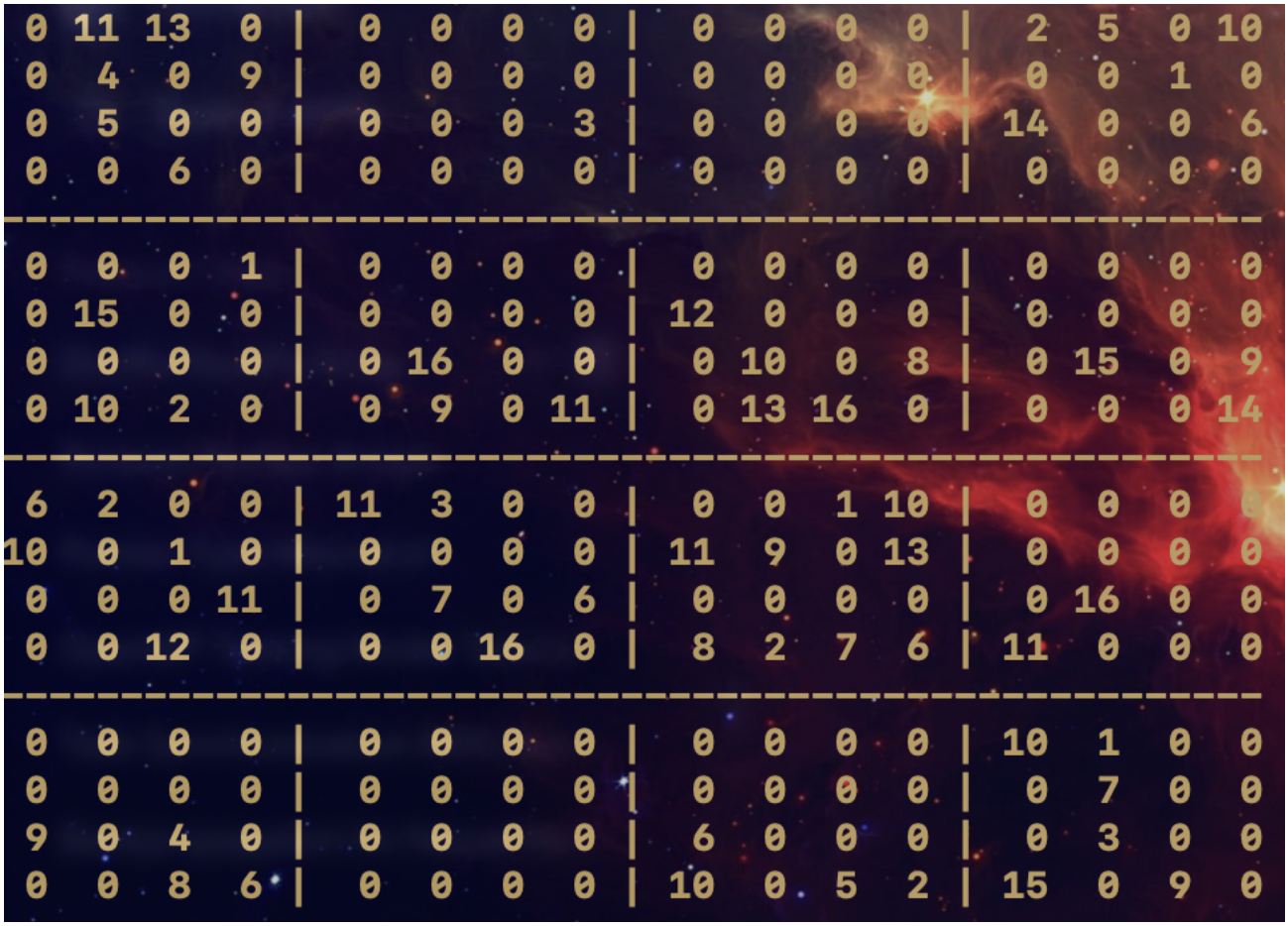

Figure 2: Computer Generated Sudoku

Threads	Mean Execution Time [s]	Standard Deviation [s]
1	>3600	0
2	27.520	27.520
4	0.940	2.800
6	0.326	0.692
8	0.053	0.118
10	0.011	0.000
12	0.014	0.000
14	0.016	0.000
16	0.020	0.000
18	0.024	0.000
20	0.025	0.000
22	0.028	0.000
24	0.036	0.000
26	0.030	0.000
28	0.040	0.000
30	0.047	0.000
32	0.055	0.000

Table 2: Performance on difficult computer generated Sudoku board using brute force

Threads	Mean Execution Time [s]	Standard Deviation [s]
1	>3600	0
2	0.142	0.109
4	0.052	0.089
6	0.017	0.000
8	0.012	0.000
10	0.012	0.000
12	0.015	0.000
14	0.017	0.000
16	0.018	0.000
18	0.022	0.000
20	0.025	0.000
22	0.029	0.000
24	0.035	0.000
26	0.030	0.000
28	0.043	0.000
30	0.043	0.000
32	0.053	0.000

Table 3: Performance on difficult computer generated Sudoku board using entire algorithm

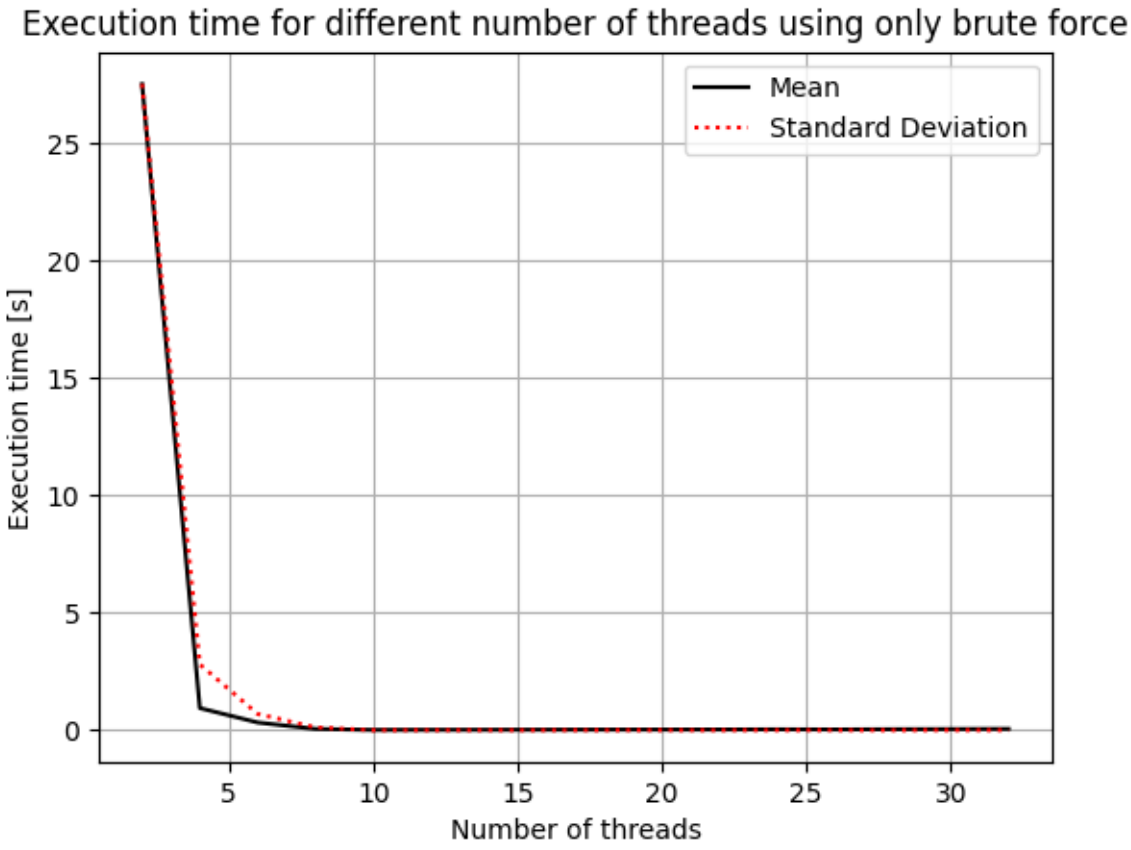

Figure 3: Execution time plotted against number of threads using brute force, board size 16x16

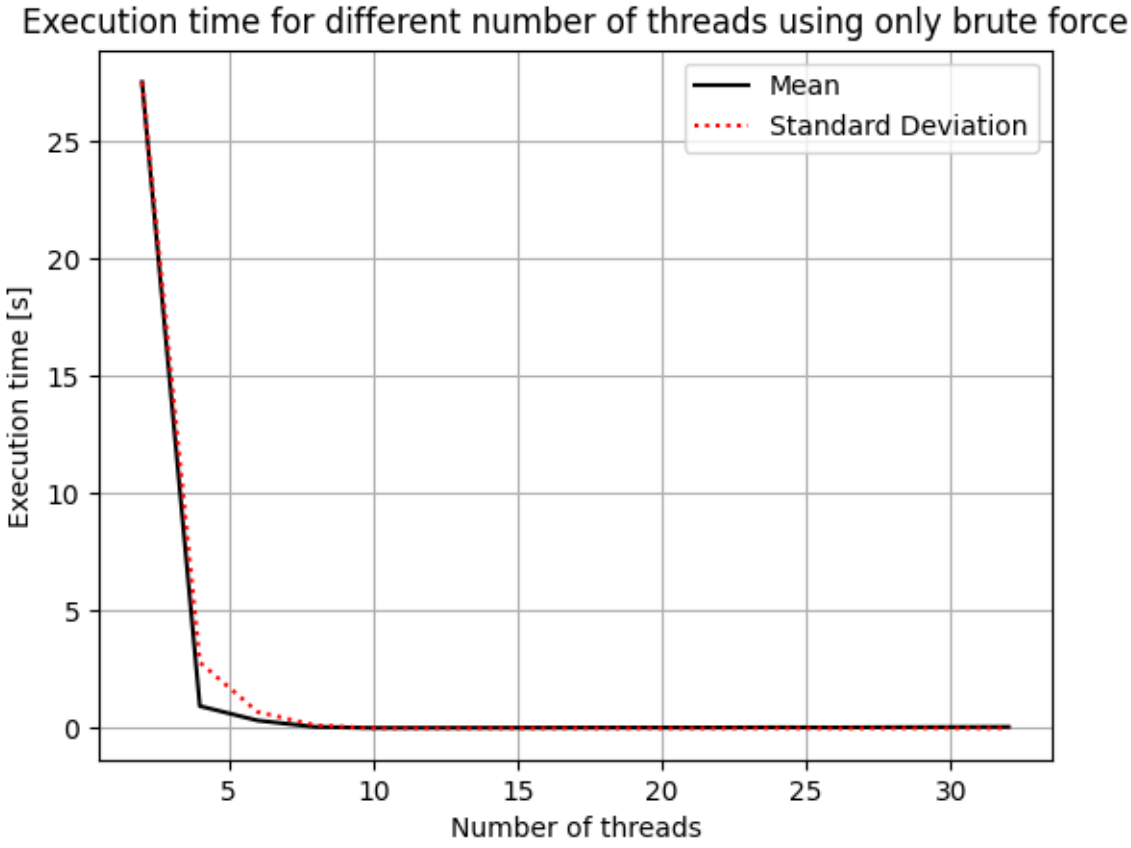

Figure 4: Execution time plotted against number of threads using entire algorithm, board size 16x16

No time was able to be measured for the case of one thread since this took too long, after 1 hour the run was terminated. In the tables the time is at above 1 hour when using both only brute force and using the entire algorithm. The time using only brute force is however very likely to be much longer. Although the fact that no time was able to measure might first appear to be a bad thing, this is a very difficult Sudoku board to solve. The logical operations can only do so much and the brute force method needs to go through a lot of different numbers. It can be shown how the performance drastically increases with the increase of number threads. It is difficult to view the impact of going from one to two threads since there exists no definite time. The impact is at least a 99.2% increase when only using brute force and a 99.996% increase when using the entire algorithm, but likely it is larger than that especially for when only using brute force. Peak performance is achieved at around 10 threads for both brute force and entire algorithm. It can also be viewed how much of an impact adding logical operations has on the performance with the result when using 2 threads going from 27.52 seconds to 0.142 seconds.

The algorithm was then tested on a difficult computer generated Sudoku board size of 25x25. In this case the test was performed only 16 and 32 threads since using less threads would take too much time. It can be viewed in table (4) that the parallelization works good even on larger boards.

Threads	Mean Execution Time [s]	Standard Deviation [s]
16	4.235	7.061
32	0.938	0.875

Table 4: Performance of the entire algorithm on a difficult computer generated 25x25 board

The problem with these times although they showcase a very good performance is that these Sudokus do not have a unique solution. This means that in the depth first search method there exists several branches that result in a solution. The parallelized performance is therefore a bit misleading since there is a high chance an optimal branch is found with a just a few threads. To further measure the performance the method was therefore tried on very difficult 16x16 Sudoku boards that could be found online.

5 Very difficult Sudokus

The Sudoku boards were taken from this website: `https://puzzlemadness.co.uk/16by16giantsudoku/tough/2024/3/31` and the ones selected were the highest difficulty available. Three different Sudoku boards were tried to get an idea how the algorithm functions on different Sudoku boards. An example of these board can be viewed in figure (5).

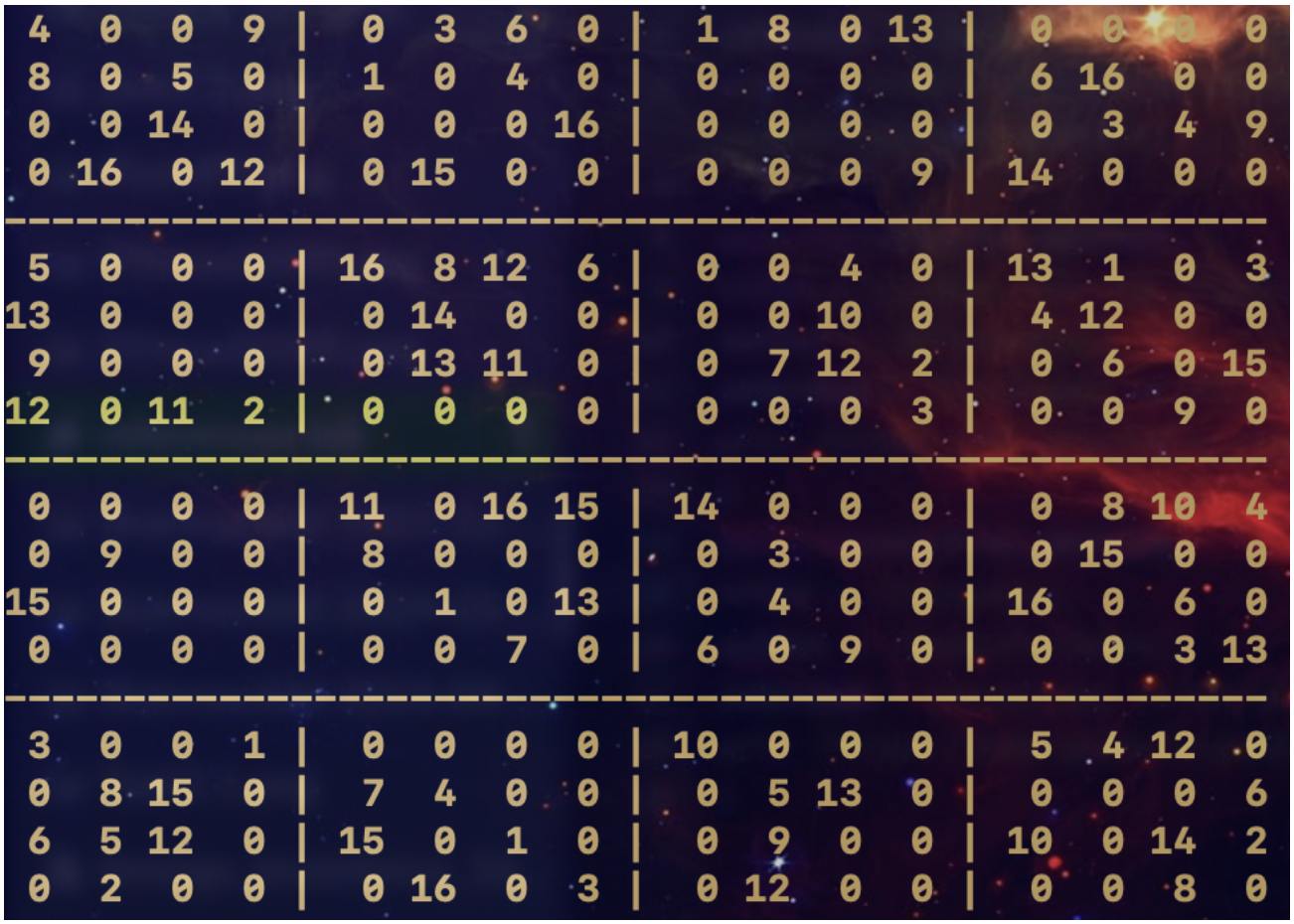

Figure 5: Tough 16x16 board

Threads	Board 1 [s]	Board 2 [s]	Board 3 [s]
1	2.040	0.511	0.252
2	2.351	0.605	0.240
4	0.496	0.342	0.006
6	0.567	0.390	0.006
8	0.621	0.272	0.006
10	0.758	0.440	0.007
16	1.409	0.618	0.010
32	2.121	1.848	0.013

Table 5: Performance of the entire algorithm on a difficult computer generated 25x25 board

Here it can be viewed how the nature of the Sudoku board has a large impact on the performance time.

6 Conclusions

The difference in performance between boards solvable by humans and difficult computer generated boards can be explained by how the algorithm functions, especially the logical operations. The logical operations uses the rules of the Sudoku game to fill in as many cells as possible. When this works the code executes quickly but it has the problem of not always working. These techniques that the logical operations use are the same techniques used by humans when solving a Sudoku board. This means that if a human can solve a Sudoku board this algorithm will very likely be completely able to solve the Sudoku board. These operations are not very computationally expensive which means the performance will be very good. Since the logical operations only use three Sudoku solving techniques there might be cases where some cells are left empty after the logical operations. In these cases the number of empty cells will be few which means the brute force part can quickly finish the rest of the work. The computer generated boards that were used were created by assigning unique numbers to the entire board and then removing random numbers from the board. This results in a board that doesn't have a unique solution and needs some guessing to be solvable. The Sudoku solving techniques will not be useful when solving the Sudoku board in these cases.

The result of the parallelization does not follow the traditional pattern of a parallel in task, that is the time reducing by a factor of $1/N$ where N is the number of threads. The reason behind this is that this is not a traditional parallelization. An example of a traditional parallelization is dividing a loop into N number of parts and splitting this work into N threads. The brute force implementation is a depth-first search algorithm. This means that each possibility is explored until either a solution or an invalid board is found. If the search area is depicted by being a tree that means each branch is explored until the correct one is found. The parallelization of this task is performed by dividing the possible values of a cell between tasks and this task themselves create new tasks through recursive calls. In the tree this can be viewed as several branches being explored simultaneously. The parallelization works so well because it doesn't simply divide the work across a number of threads, it decreases the search region of the algorithm. This is also the reason as to why the performance between runs varies. The point of the search is

to find a path that results in a solved board. Since increasing the number of threads increases the number of branches being explored simultaneously it also increases the chance that a path yielding a solution is found. This can be viewed by how the standard deviation becomes lower for each run. This is a very effective parallelization since it improves the performance greatly while achieving maximum performance using a lower amount of threads than other parallelization, which can be very valuable depending on the hardware.

The reason why the algorithm operates so much faster on the computer generated boards compared to the Sudoku boards found online is that there exists several solutions to the generated boards. The board found online is designed for humans to solve and therefore has one unique solution. The depth-first search method will therefore have a harder time locating the correct branch while when there exists several solutions there are many branches that offer a solution. It will therefore be quite easy to locate a solution, especially when increasing the amount of threads. The varying performance on the Sudoku boards found online can be explained by how this is a depth-first search method. For instance, in the case of Board 1 the correct branch is a branch that takes a while to locate while in the case of Board 3 the correct branch is one of the first ones tried. On these boards the best time was found to be at around 4 threads.

Although the program operates efficiently there is always room for improvement. The logical operations part of the algorithm could be extended to further improve performance. There exists more Sudoku solving techniques than those that were implemented. One such example is the inclusion of triplets, cells that share three possible values. These operate in similar manor to twins but since they have three possible values they become a bit more complicated. The extension of the logical operations would definitely improve the performance of the algorithm.

The program developed succeeds in solving a variety of boards and does so efficiently. The program has limitations in solving difficult computer generated boards over the size of 25x25 or even 25x25 boards when not many cores are available. When operating with Sudoku boards that are easy or designed to be solvable by humans the program operates very efficiently. The limitations here are boards over 49x49 that the program cannot solve due to exceeding the size of the bitmask.

References

[1] Wikipedia. "Sudoku solving algorithms." (), [Online]. Available: https://en.wikipedia.org/wiki/Sudoku_solving_algorithms. Accessed: 17/03/2024.

[2] A. Tarhini. "Parallel depth-first sudoku solver algorithm." (), [Online]. Available: https://alitarhini.wordpress.com/2011/04/06/parallel-depth-first-sudoku-solver-algorithm/. Accessed: 14/03/2024.

[3] GeeksforGeeks. "Bitmasks." (), [Online]. Available: https://www.geeksforgeeks.org/c-bitmasking/. Accessed: 17/03/2024.