Parallelized Sudoku Solvers

Abdullah Zahran Abdullah Al Hinaey
Undergrad, Dept. Computer Science
University of Central Florida
Orlando, FL, USA
ab618186@ucf.edu
JohnMichael Kane
Undergrad, Dept. Computer Science
University of Central Florida
Orlando, FL, USA
jo502519@ucf.edu
Marlon Masia
Undergrad, Dept. Computer Science
University of Central Florida
Orlando, FL, USA
ma904984@ucf.edu
Soleil Cordray
Undergrad, Dept. Computer Science
University of Central Florida
Orlando, FL, USA
so649100@ucf.edu

***Abstract*—In computing, most problems that can be solved sequentially also have a parallel solution. Not all the time but in some cases parallelization can make a program more efficient than its sequential counterpart, especially when it comes to large data. When it comes to algorithms that can solve a Sudoku board, some of these include brute force, dancing links, and propagation and search. These algorithms can be implemented using sequential techniques or parallelization techniques. Our goal with this paper is to prove that parallelization techniques are more efficient at finding solutions than sequential techniques.**

I. Introduction

Modern day Sudoku is a popular combinatorial puzzle game that gained its prominence in Japan, 1986. The game came to the United States a few decades later in 2004, and has since become one of the staple puzzle pastimes. Sudoku boards, depending on their difficulty and how familiar the player is with the game, can take anywhere from 5 to 25 minutes to solve. Computers can obviously solve these kinds of puzzles a lot faster, but how do different algorithms compare to one another? and what is the effectiveness of parallelization techniques to help improve those times?

Essentially the puzzle takes place in a n-by-n grid, where each grid has n boxes that are $\sqrt{n}$-by-$\sqrt{n}$ in size and have n cells within them. In a standard sudoku board it would be a 9x9 board with 9 different regions of 3x3. Each solvable Sudoku board has a certain number of empty cells, the more empty cells there are in a solvable grid, the higher the difficulty of the Sudoku board. For this project we will be using 3 different board sizes: 9-by-9, 16-by-16 and 25-by-25.

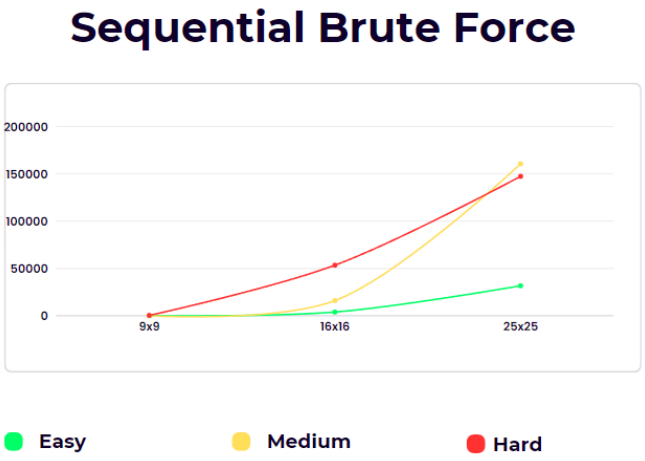
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Our project aims to explore various Sudoku board solving techniques, including Bruteforce, Propagation and Search, and Dancing Links, with a focus on comparing their runtime efficiency. In addition to analyzing the runtime of these techniques, we will also investigate how their performance varies across

different board difficulties. To get a good pool of data when it comes to runtimes and to ensure that we are getting accurate results, we will be testing all these algorithms with the board sizes stated above. To make sure that there are not small inaccuracies in the data the same boards will be used in testing all the algorithms. Board difficulty in Sudoku is represented by the amount of numbers given in a solvable board. The fewer numbers given at the start of a solvable board the harder the difficulty of the board. By conducting thorough comparisons, we are seeking insights into the strengths and weaknesses of each approach, ultimately contributing to a deeper understanding of Sudoku solving algorithms.

II. BRUTE FORCE TECHNIQUE

The First technique we will take a look at is the least complex of algorithms used for solving any kind of problem, Brute Force. Brute force algorithms tend to go through all possible choices of a problem until the solution is found. This means that the runtime of most brute force algorithms scale with the input size given. In the case of solving Sudoku boards the initial reaction is that any kind of changes to the board size or any change to the amount of empty cells the algorithm has to figure out will lead to an exponential change in time. When we actually implement the algorithm to solve boards in different sizes what we this graph below



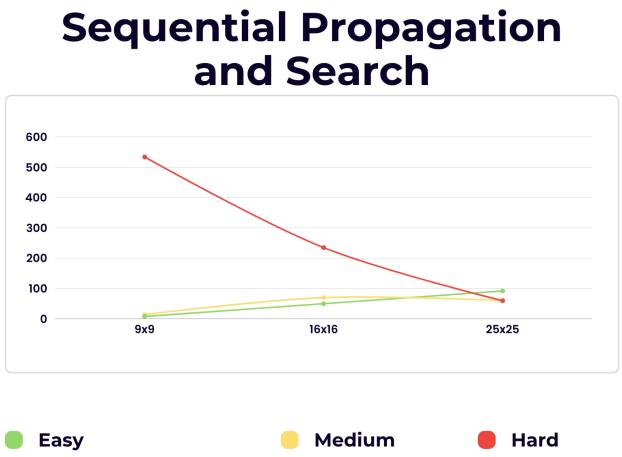
As we can see in the image, the exponential growth of the graph is present. The bigger the size of the board the more time it takes for the algorithm to solve the board. However what is interesting and what is important to note is the actual amount of empty cells the algorithm is solving for. For example, when looking at the easy boards, the 9x9 has 45 empty cells, 16x16 has 140 empty cells, and 25x25 has 191 empty cells. The question one may have when seeing this is why the exponential growth in time if there is not exponential change in empty cells to solve? The reason for this is the fact that for every empty cell the algorithm in the brute force technique – which is using the backtracking algorithm – needs to iterate through that cell n times, n being the size of the board. This means that if you are dealing with a 25x25 board, the algorithm needs to go through an empty cell at least 25 times. All of this exponential growth of runtime is just looking at the easy level boards. When looking at higher difficulty boards the time growth is even more exponential. The one exception to this rule for some reason is the 25x25 board which in the hardest difficulty is faster than the medium version. An early theory as to why this might be the case has to do with luck of the hard difficulty board just having better starting positions for their numbers compared to the medium one. In this case it seems that the difference in these starting positions is saving the program around a 20 second time difference to solve the hard version as opposed to the medium one.

III. PROPAGATION AND SEARCH TECHNIQUE

This second method is the propagation and search technique which has a complex combination of many algorithms. Unlike the broad tactic of the brute force technique, this method combines grid analysis, constraint propagation, and depth-first search (DFS) into an efficient strategy that slowly makes itself easier. At the core, the method begins with a meticulous examination of the puzzle grid, leveraging the PossibleGrid class to map out all potential numerical candidates for each unsolved cell. This step is not just the start but it's done many times throughout the process in order to make it quicker and more efficient as more information is gained each run. By discerning the landscape of possibilities, the method significantly narrows down the search space, pinpointing areas that can be solved first.

The cornerstone of this approach is its constraint propagation mechanism. Rather than a single elimination run, this phase actively applies Sudoku's rules to iteratively refine the pool of possibilities. Each cell, row, column, and $\sqrt{n}$-by-$\sqrt{n}$ block is checked not only for immediate solutions but also for patterns that might indicate a singular path forward. This process reduces puzzle complexity by a considerable amount, allowing many cells to be solved without the need for guessing. However, some cells need to be guessed for sudoku, which is where the transition to depth-first

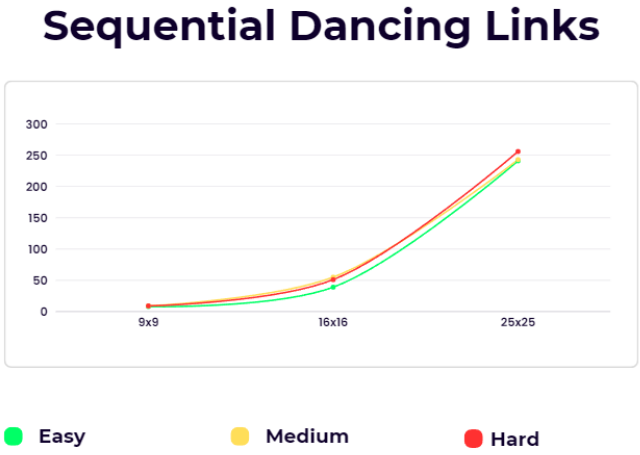
search (DFS) exemplifies the method's versatility. Recognizing that not all puzzles yield to logic alone, the solver embarks on a calculated exploration of the remaining possibilities. Each choice is a hypothesis tested against Sudoku logic, with backtracking serving as a corrective measure for missteps along the way. When implementing the technique to solve Sudoku boards, we get the graph below.



Unlike the steep exponential inclines characteristic of pure Brute Force approaches, this method's graph would manifest a more gradual ascent. Initial stages of constraint propagation effectively flatten the curve for simpler puzzles, with the DFS component ensuring that even as puzzle complexity increases, solve times remain within a reasonable bound. This implementation is by far the quickest. The reason being is the hybrid implementation of the search approach using logic and the depth-first search allows every cell to be accurately and quickly solved and accessed. This algorithm adapts for more complex issues and allows the algorithm to get simpler every single cycle.

IV. DANCING LINKS TECHNIQUE

The next technique is the dancing links technique which was invented by Donald Knuth, and is mostly used to solve combinatorial problems like sudoku. The algorithm operates on a data structure known as DLX, doubly linked toroidal list. This data structure consists of nodes arranged in a matrix-like format with columns and rows. Each column represents a constraint, and each row represents a possible solution. Each cell in the matrix represents a potential choice or decision. The goal is to find a combination of choices that satisfies all constraints. At each step the algorithm selects a column(constraint) to satisfy and it recursively does this until it finds either a solution or a dead end. If the algorithm finds a dead end then it backtracks until a new solution is found. The reason for the name Dancing Links is because the algorithm traverses the DLX matrix and selectively removes and restores the links between nodes in the matrix to efficiently explore the search space. This process allows the algorithm to backtrack efficiently without needing to reconstruct the entire search tree. When implementing this technique to solve Sudoku boards we get the following graph.



Since Dancing Links technique is technically an improved backtracking algorithm the same exponential line graph can be seen as in the brute force technique however the times are greatly improved upon. *See comparisons chapter for further analysis on the differences in time*. The belief up to this point was that the exponential growth seen in these algorithms which use backtracking as its base was that their improvements would be seen once we implement their. Looking at the runtime across different difficulties shows how consistent of an approach the dancing links technique is and that is just in its sequential version.

V. PARALLELIZATION

Parallelization in theory can help reduce runtime of problem-solving processes compared to their non-parallel counterparts. This is done by leveraging the computation power of multiple processing units – that would otherwise be done by one centralized processing unit in sequential processing. However, parallelization might not always lower runtime and can also unnecessarily complicated a simpler more efficient program. To maximize the benefits of parallelization on must carefully address complex ideas such as load balancing, synchronization, and communication overhead. For this

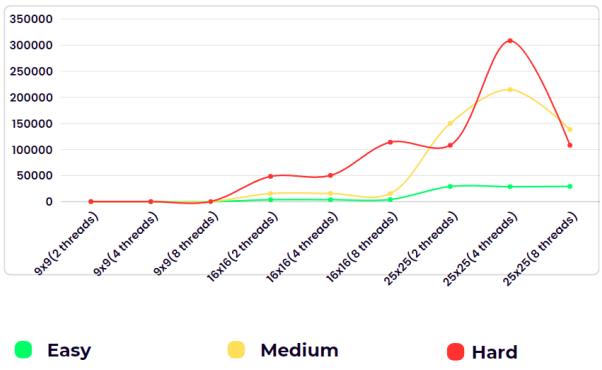
project introducing parallelization to the three Sudoku solving techniques used may alleviate some of the computational stress of running the techniques sequentially, causing it to run faster. To start off with this theory let us analyse how it was possible to transform these techniques to their parallelized versions as well as how they performed based on the number of threads used (2,4,8).

The first technique we will look at is the brute force technique. In this implementation all threads share the same Sudoku board, which requires effective management to ensure they all cease operations once a solution is found. We employed 'ExecutorService' to manage the lifecycle of these threads, including their inititation,execution, and termination. Additionally, we utilized an 'AtomicBoolean' flag to guarantee that, upon finding a solution, all threads are immediately informed to halt further processing.

```
ExecutorService executor = Executors.newFixedThreadPool(numThreads);
startTime = System.currentTimeMillis();

for (int i = 0; i < numThreads; i++) {
    executor.execute(new SudokuSolver(board));
}

executor.shutdown();

while (!executor.isTerminated()) {
    // wait for all tasks to finish
}
```

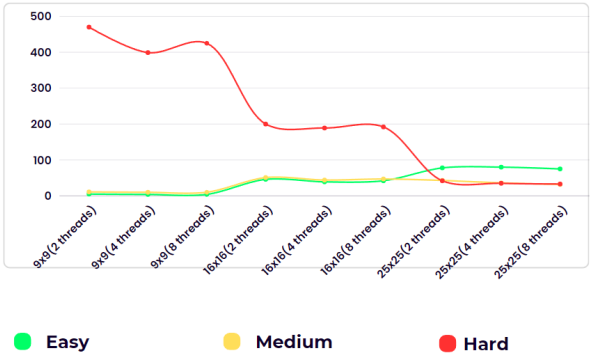
When we start to look at the runtime of solving these sudoku boards, especially with 9x9 and 16x16 grid sizes, compared to a sequential implementation it ran slower than we had imagined. The primary reason for these slower speeds might be because of the additional steps required to manage and handle the threads.

Parallelized Brute Force



Both propagation and search can benefit from parallelization because they often involve independent computations or evaluations that can be carried out concurrently. By distributing these computations across multiple processing units, the overall time required for finding solutions can be reduced. In parallelized propagation and search, communication between processing units is crucial. While each unit may work independently on a subset of the problem, they may need to share information periodically to ensure coherence and prevent redundant work. Efficient parallelization requires load balancing to ensure that all processing units are utilized optimally. This involves distributing the workload evenly among the available computing units to avoid bottlenecks and idle resources. In some cases, synchronization may be necessary to coordinate the activities of parallel processes. This ensures that they proceed in a coherent manner and that the final solution is consistent.

Parallelized Propogation and search

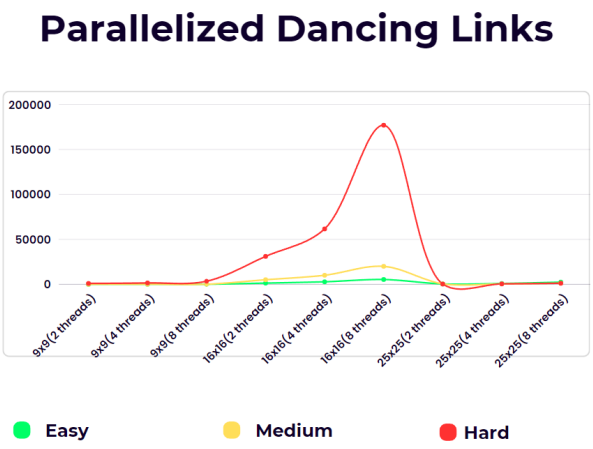


Finally let us examine the implementation and results for the parallelized Dancing Links. Essentially we start off with a board, and we do a basic BFS search to place in a few cells and create a Board for every single thread. Each of these threads will using a dancing-link approach on that board and whichever one comes up with the solution first will end the process.

basic idea: Finally, we can modify the depth first search approach so that threads can function independently without all reading from the same stack. Breadth first search is easy to parallelize, so we can begin by finding all possible valid boards that fill the first, say, 20 empty spaces of the given puzzle. This may give us something like thousands of possible boards. These boards may then be passed to another kernel, where each thread then applies the backtracking algorithm to it's own board. If any thread finds a solution, all threads stop, and that solution is passed back to the cpu.

We can see that both of these steps are much easier to parallelize than the original depth first search solution. In the end, this is the primary approach we chose to focus

on.



When implementing the Parallelization for Dancing Links there were issues with the 25x25 board implementation. The numbers for the 25x25 boards don't accurately represent the runtimes one might get when using parallelized dancing links in that board size. Either way the other two boards sizes show that threads are negatively impacting the runtime of the program. This problem is the most glaring when tested on the hard difficulty boards sizes, for other sizes it does not seem to make much of an impact on run times.

VI. COMPARISONS

Now that all the parallelization techniques have thoroughly explained, it is time to compare the techniques to one another. Techniques will be compared to one another based on which sequential implementations were the most efficient and how those efficiencies changed once converted to parallelization.

When comparing sequential brute force versus its parallelized version, it can be noted that there is not much of a difference. When using two threads compared the sequential version the there actually is an improvement in run time across most difficulties and board sizes, however this improvement is so marginal – sometimes improving runtime by a measly 20ms – that it can be looked as not worth the effort of implementing parallelization. This feeling is further supported by the fact that as more threads get added the run time actually starts to get worse. The only exception to this rule can be seen with the difference between four and eight threads in the 25x25 size, medium and hard boards, where there was an improvement of time when it came to solving those. However, although eight threads shows to be an improvement over four, the eight thread implementation runs just as fast as the two thread implantation, showing once again that it is not worth implementing parallelization.

Sequential propagate versus parallelized propagate show very little difference between one another. Threads do tend to run faster than not using anything, however, more threads do not improve this time any further.

The sequential implementation of dancing-links copies the given board and then sends it over to DancingLinkSolver to solve that board using dancing-links. In the parallel implementation of dancing-links, the program goes through a BFS (Breadth-First-Search), and creates n boards for each thread n. Then each thread will then be sent to DancingLinkSolver, and will attempt to solve their board until a thread finds a solution, in which case the process ends similar to the sequential.

VII. CONCLUSION

In the lengthy pursuit of optimizing code and diminishing runtime, parallelization – the potential to harness collective power of multiple processing – emerged as a promising contender in slashing runtime inefficiencies. However after putting all the parallelization to the test, trying to optimize Sudoku solving techniques, it seems like it may not be all that beneficial, and in some cases less effective. For example, in cases like the brute force technique it seems like there are diminishing returns when excessively threading possibly due to increased overhead. Other papers that have looked into this matter also seem to come to a similar conclusion of the use of threads in the brute force technique [1]. When looking at the test cases for the Parallelized dancing links we can see that parallelization negatively impacted the run times of the program the more threads were added.

Ultimately, while we are sure there are cases were parallelization may be resourceful to optimize certain programs, parallelized Sudoku solving programs – at least to the extent of this paper's research – do not benefit. The results of this paper, while slightly disappointing to those that might have seen parallelization as a programming cheat code to optimize programs, has been a very insightful learning process. Not only have we learned a lot about parallelization but we feel like we have gained a better grasp of programming as a whole.

REFERENCES

[1] S. Foucher, *Multi Thread Performance analysis of a Brute-Force Sudoku Solver*. Montreal, Canada: McGill publications, 2009.