Efficient Parallel Sudoku Solver via Thread Management & Data Sharing Methods

Shawn Lee
Department of Computer Science & Engineering, University of California, Riverside
slee208@ucr.edu

ABSTRACT

Sudoku is a logic-based, combinatorial number-placement puzzle that has been popular since the late 20th century. Solving n^2xn^2 grids of *nxn* blocks tends to become increasingly difficult due to combinatorial explosion. As a result, a sequential implementation of a Sudoku puzzle solver can become both data- and compute-intensive. In this work, we construct parallel implementations of the Sudoku puzzle-solving algorithm using constraint propagation, storing already-explored grids, and storing to-be-explored grids. Savings in execution time result from different thread management methods and data sharing methods. Threads attempt to solve the puzzle using depth-first search. Also, the implemented Sudoku puzzle-solving algorithm specifically solves *16x16* grids of *4x4* blocks.

In the first phase of reducing execution time via multi-threading, we propose to implement a global data structure to store already-explored grids, a thread-local data structure to store to-be-explored grids, and have threads that complete its branch to begin work on a new branch until a solution is found. In the second phase of reducing execution time, we propose to implement a global data structure to store to-be-explored grids along with already-explored grids, and have threads work in the same branch until it is exhausted, then begin work in a new branch until a solution is found. Our experiments show that our different thread management and data sharing methods enable significant reductions in execution time (2.42-45.04x) while correctly solving the Sudoku puzzle.

CCS CONCEPTS

• **Computer methodologies → Parallel computing methodologies**; Parallel programming languages;

KEYWORDS

Sudoku Solving; Constraint Propagation; Multi-threaded

1 INTRODUCTION

Sudoku was designed in 1979 as a numerical combinatorial puzzle by Howard Garns, an American architect. In late 2004, Sudoku grew in popularity after it was published in newspapers as a regular feature. The 9-by-9 grid has been the most common grid size, with 6,670,903,752,021,072,936,960 possible combinations. As the grid size grows larger, the puzzle rapidly becomes more complex, with the 16-by-16 grid having approximately $5.96x10^{98}$ possible combinations.

Regardless of the size of the grid, Sudoku has two rules to follow at all times. Each of n^2 blocks must contain all the numbers from 1 to n^2 and can only appear once in a row, column, or *nxn* block. When all the cells in the grid contain a number and satisfy the rules, the puzzle is solved.

Table 1: Example 9x9 Solved Sudoku Square

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

Table 2: Example 9x9 Sudoku Problem

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

There are various Sudoku solving algorithms that aim to solve the puzzle quickly with a computer, through the use of rapid guessing, stochastic search, backtracking, and/or constraint programming.

This work aims to reduce the execution time of solving 16x16 Sudoku puzzles through the use of constraint satisfaction, backtracking, and parallel programming. Section 2 gives a brief explanation of the basic rules of solving a Sudoku puzzle and explains the sequential implementation of the algorithm stated above. Section 3 explains the first attempt at reducing execution time by parallelizing the algorithm. Section 4 explains the execution time improvements made upon Section 3 by using a different thread management and data sharing implementation.

The experiments were conducted on a desktop with the following configuration:

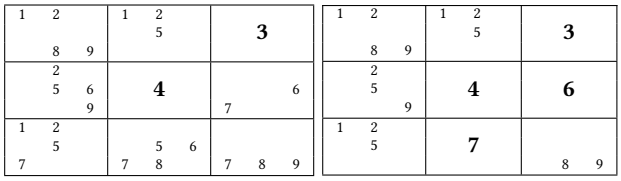
- Intel Core i7-6800K, 6 cores with hyper-threading (resulting in 12 logical threads) @ 3.91 GHz
- 16 GB DDR4 RAM
- Bash on Ubuntu on Windows 10

2 SEQUENTIAL IMPLEMENTATION

This section provides an overview of the sequential implementation of the Sudoku algorithm. While many methods to solve Sudoku puzzles exist, such as stochastic search, we focus on constraint satisfaction and backtracking algorithms as it would be able to solve all sudokus and have a fast solving time.

June 2017, University of California - Riverside, Riverside, California USA

Figure 1: Constraint satisfaction example.



(a) Before inserting value 7 to (2, 3) **(b) After inserting value 7 to (2,3)**

Algorithm 1 presents our implementation of the constraint satisfaction algorithm. The function `REDUCE()` removes `value` from the set of possible values in each cell in the respective row, column, and the *nxn* block of the `index`. Each of the cells in a n^2xn^2 grid stores a set of possible values from 1 to n^2. As cells are reduced to only one possible value, the cardinality of the set of possible values in neighboring cells become smaller. This increases the probability that one of the values in the set is valid to reach the solution.

2.1 Constraint Satisfaction Algorithm

Algorithm 1 Constraint Satisfaction

```
1:  procedure REDUCE(index, value, grid)
2:      row ← get_row(index)
3:      col ← get_col(index)
4:      for each cell ∈ row do
5:          cell.remove(value)
6:      end for
7:      for each cell ∈ col do
8:          cell.remove(value)
9:      end for
10:     for each cell ∈ get_subgrid(index) do
11:         cell.remove(value)
12:     end for
13: end procedure
```

Figure 1 illustrates constraint satisfaction by setting the value 7 to cell $(2, 3)$, and removing the value 7 from all cells in the same row, column, and 3-by-3 block.

2.2 Determining Next Derived Grid

Algorithm 2 Determining Next Derived Grid

```
1:  procedure MIN_POSSIBLE_VALUES(grid.unsolved)
2:      ▷ unsolved contains a set of index values of cells with more
    than one possible value
3:      for each index ∈ unsolved do
4:          if index.num_possible_values() < MIN then
5:              MIN ← index
6:          end if
7:      end for
8:      return MIN, index.value
9:  end procedure
```

Algorithm 2 presents our implementation to determine the next grid to explore. Since there is a higher probability that a value in a smaller set is correctly placed in the grid, we use `unsolved` to determine which cell has the minimum set of possible values due to constraint satisfaction updates. Each grid has an `unsolved` set, which contains the index value of cells that have more than one possible value.

2.3 Solving Algorithm

Algorithm 3 Grid Validity

```
1:  procedure VALID_GRID(grid)
2:      ▷ checks uniqueness of values in row
3:      for each cell ∈ row do
4:          if cell.num_possible_values() == 1 then
5:              row_index_set.push_back(cell.value)
6:          end if
7:      end for
8:      if row_index_set.is_not_unique() then
9:          return false
10:     end if
11:     ▷ checks uniqueness of values in column
12:     for each cell ∈ col do
13:         if cell.num_possible_values() == 1 then
14:             col_index_set.push_back(cell.value)
15:         end if
16:     end for
17:     if col_index_set.is_not_unique() then
18:         return false
19:     end if
20:     ▷ checks uniqueness of values in subgrid
21:     for each subgrid ∈ grid do
22:         for each cell ∈ subgrid do
23:             if cell.num_possible_values() == 1 then
24:                 subgrid_index_set.push_back(cell.value)
25:             end if
26:         end for
27:     end for
28:     if subgrid_index_set.is_not_unique() then
29:         return false
30:     end if
31:     return true
32: end procedure
```

Sudoku has two simple rules to follow, which is presented in the implementation of Algorithm 3. First, each of the n^2 blocks must contain all the numbers from 1 to n^2. Second, each number can only appear once in a row, column, or *nxn* block. Algorithm 3 presents our implementation of determining the validity of the grid after completing `REDUCE()`. The algorithm can also be viewed as an exploration of a game tree (see Figure 2) such that each node of the tree represents a game state (i.e., a grid). The `REDUCE()` function performs constraint satisfaction over a game state $\mathcal{G}$ to generate $\mathcal{G}'$; this $\mathcal{G}'$ becomes a child of $\mathcal{G}$ in the game tree.

`VALID_GRID()` is called after each call to `REDUCE()` in order to guarantee a grid $\mathcal{G}$ satisfies the rules of Sudoku. In some occasions,

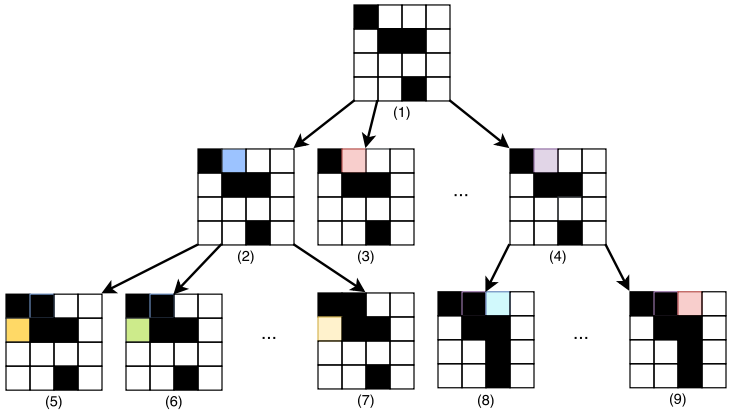


Figure 2: A representation of the game tree. **Black cells** are cells that have one value. **Colored cells (red cells represent values that causes an invalid grid)** are values being tested for that game state. **White cells** are cells with more than one possible value. Note that the children of grid 4 have an addition black cell, suggesting that **constraint satisfaction** caused that cell to have only one possible value.

after constraint satisfaction was completed on a grid $\mathcal{G}$ to generate $\mathcal{G}'$, cells in the same row, column, or *nxn* block may result in having only one possible value. Although this is not an issue, these cells may also have the same possible value, and thus making the $\mathcal{G}'$ invalid.

If $\mathcal{G}'$ is valid, $\mathcal{G}'$ is compared to a set of already expanded grids (described as the *explored* set), which is stored as a red-black tree. To order grids in the tree, we compare the bitstrings of $\mathcal{G}'$ to the bitstring of the grids in the tree. This bitstring, which represents a grid, contains all sets of possible values of that grid. If the bitstrings are equal while traversing the tree, $\mathcal{G}'$ is ignored and another grid $\mathcal{G}$ is popped off the fringe.

This ensures that the solving algorithm, presented in Algorithm 4, does not traverse an invalid or already expanded subtree in the game tree. If $\mathcal{G}'$ is not found in this explored set, it is then stored into a queue (described as the *fringe*) for further expansion as well as the aforementioned explored set. These algorithms are repeated until a solution is found, wherein all cells in the n^2xn^2 grid have only one possible value and is still a valid grid. However, if the reduced grid is invalid, it will not be stored in the fringe.

Algorithm 4 Grid Solving Algorithm

```
1:  procedure SOLVE(fringe, explored, grid)
2:      while grid.unsolved ≠ ∅ do
3:          temp ← fringe.pop()
4:          index, value ← MIN_POSSIBLE_VALUES(grid.unsolved)
5:          REDUCE(index, value, grid)
6:          if VALID_GRID(grid') then ▷ grid' is the derived grid
7:              fringe.push(grid')
8:              explored.push(grid)
9:          else
10:             explored.push(grid)
11:         end if
12:     end while
13:     return grid'
14: end procedure
```

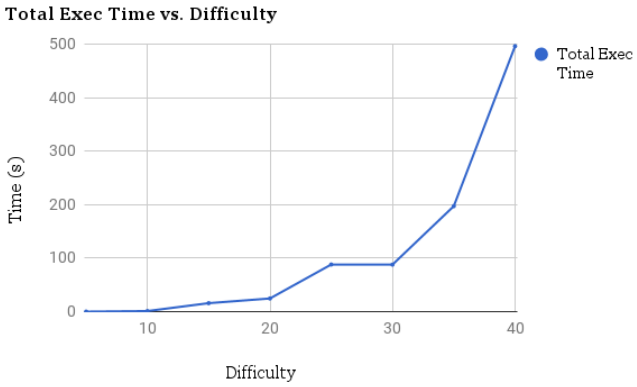


Figure 3: Graph of Sequential Time vs. Difficulty

2.4 Sequential Performance

We study the correlation of different puzzle difficulties to the amount of time spent to solve the it. Figure 3 shows the total time to solve 8 different puzzles of increasing difficulty. The difficulty begins at 5 and increases in increments of 5, where the difficulty value correlates to different number of unsolved cells and location of each value during puzzle setting. In sparser puzzles, the location of each value may reduce the possibilities of another cell, thus making the puzzle easier to solve. As we can see in the graph, as the difficulty of the puzzle increases, the time required to solve the puzzle also increases. From these baseline results, we can improve the time by parallelizing the algorithm and having threads work towards the solution in different subtrees of the game tree.

3 PARALLEL PHASE 1: IMPLEMENTATION

Our first attempt to improve the solving algorithm is by working towards the solution in parallel. In order to parallelize the algorithm, we start by creating threads that work on a different subtree based

Algorithm 5 Grid Solving Algorithm - Parallel BFS

```
1:  procedure SOLVE(explored, grid)
2:      fringe ← ∅
3:      while grid.unsolved ≠ ∅ do
4:          temp ← fringe.pop()
5:          index, value ← MIN_POSSIBLE_VALUES(grid.unsolved)
6:          REDUCE(index, value, grid)
7:          if VALID_GRID(grid') then ▷ grid' is the derived grid
8:              fringe.push(grid')
9:              explored.push(grid')
10:         else
11:             explored.push(grid)
12:         end if
13:     end while
14:     return grid'
15: end procedure
16:
17: procedure MAX_POSSIBLE_VALUES(grid.unsolved)
18:     ▷ unsolved contains a set of index values of cells with more
    than one possible value
19:     for each index ∈ unsolved do
20:         if index.num_possible_values() > MAX then
21:             MAX ← index
22:         end if
23:     end for
24:     return MAX, index.value
25: end procedure
26:
27: procedure THREAD_SPAWN(num_threads, grid)
28:     for num_threads do
29:         index, value ← MAX_POSSIBLE_VALUES(grid.unsolved)
30:         REDUCE(index, value, grid)
31:         explored.push(grid)
32:         spawn_thread(SOLVE(explored, grid))
33:     end for
34: end procedure
```

off the initial game state. In other words, threads are pre-assigned reduced grids based off the cell with the highest branching factor, which is found after calling `MAX_POSSIBLE_VALUES()`.

By giving each thread a different subtree, we can utilize more compute power to find the solution. We can not only increase the number of grids to expanded, but also effectively search many different paths towards the solution.

The number of active threads are limited to *num_threads*, where *num_threads* is a user defined argument to the algorithm. By increasing the number of threads, we can observe the speedup of solving the Sudoku puzzle.

However, by introducing parallelization to the algorithm, we need to introduce data sharing methods for both the explored set and the fringe. As presented in algorithm 5, the explored set is global to all spawned threads, and, similar to the sequential version, reduced grids are placed into this set. In contrast, the fringe is local to each thread, and is created after the thread has spawned. By doing so, threads would independently have its own set of grids to reduce and work towards the solution.

In order to solve the race conditions while inserting and searching for already explored grids, we introduce a lock. By doing so, only one thread can access the explored set at any one time.

When the solution is found, by meeting the same conditions as those in the sequential version, all threads will join and the solver will end.

3.1 Parallel Phase 1: Performance

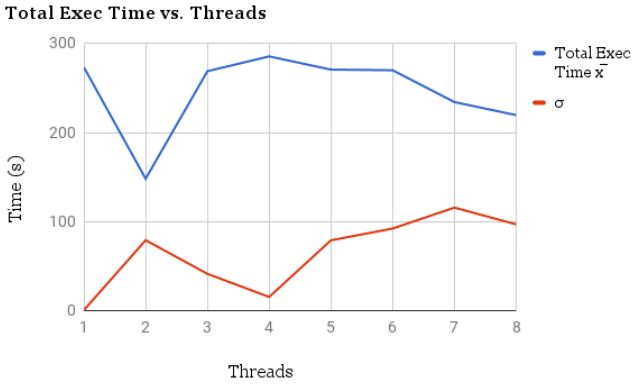


Figure 4: Graph of Sequential Time vs. Difficulty

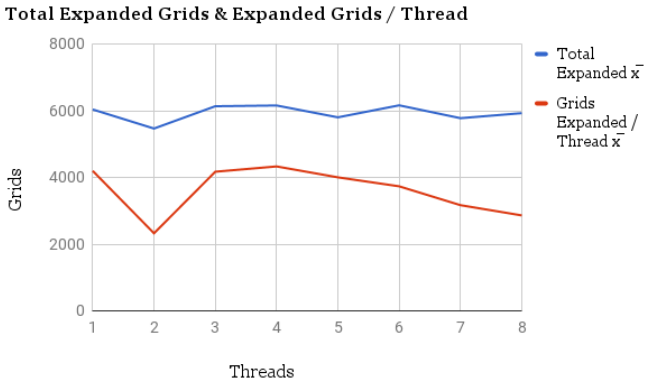


Figure 5: Graph of Sequential Time vs. Difficulty

In contrast to the sequential version, we only use the most difficult puzzle from the sequential version to test the speedup of the algorithm. By doing so, we can observe the difference in time between fewer spawned threads to more spawned threads.

3.1.1 Results. In Figure 4, we illustrate the average solve time of 15 runs from 1 to 8 spawned threads as well as its standard deviation. From the graph, it is apparent that the time it takes to solve the Sudoku puzzle remains the same, with the exception of

two spawned threads. Since our hypothesis where speedup would occur as more threads are spawned is proven false, we needed another set of statistics to understand why.

Figure 5 shows the average number of grids stored in the explored set as well the average number of grids explored by each thread. From these results, we can predict that the solution is found after, on average, around 6000 grids are explored in the set. With this set of data, we can assume that the solution has a minimum depth the tree. With the current algorithm, threads do not explore deeper in the tree faster, but rather explore widely into shallow branches in the tree. From this observation, we can also conclude that threads that are working on separate subtrees in the game tree are unnecessarily exploring grids, for the reason that their subtrees do not contain the solution. Furthermore, we can also conclude that this version of parallelizing the algorithm does not result in any significant speedup.

4 PARALLEL PHASE 2: IMPLEMENTATION

From the analysis in phase 1 of our parallel implementation, we needed another method to achieve speedup in solving Sudoku puzzles. As observed in the results illustrated in Figure 5, we concluded that a solution is found deeper in the game tree (typically at the leaves), rather than at a higher level in the tree. Therefore, it would be more relevant to use depth-first search to expand these grids. This entails that threads work on the same subtree to quickly traverse and explore deeper in that subtree. To do so, we returned back to normal spawning methods, where threads are spawned at the same time and are assigned work when available. As presented in Algorithm 6, reduced grids are no longer pre-assigned to each spawned thread. Rather, threads will obtain a new grid from the fringe before deriving new grids.

Due to the changes from having threads work on completely separate subtrees to having threads work on the same subtree of the game tree, we have made the set to be global, similar to that of the explored set.

4.1 Synchronization of the Global Fringe

One problem that arose was work starvation in spawned threads. Two scenarios generally result in this issue: threads are able to complete work quicker than they are able to explore new grids, or threads have already explored all possible grids and no solution is found. To resolve this issue, a condition variable and a mutex is introduced to the algorithm.

For the first scenario, the condition variable will check if there is work available in the fringe, such that the `fringe` $\neq \emptyset$. If `fringe` $= \emptyset$, then the thread will wait until another thread has inserted another grid to be explored. For the second scenario, a counter will increment when a thread is waiting, and decrement when the same thread is no longer waiting. If the value of the counter is equal to *num_threads* (user-defined), it means that there is no work available for any thread, and therefore suggests that a solution cannot be found.

4.2 Parallel Phase 2: Performance

4.2.1 Statistical Disturbance. As presented in Figure 6, the increase in performance is evident in relation to the number of

Algorithm 6 Grid Solving Algorithm - Parallel DFS

```
1:  procedure SOLVE(fringe, explored, grid)
2:      while threads_waiting < num_threads do
3:          while fringe = ∅ do
4:              ▷ Increments by 1 to track waiting thread
5:              threads_waiting + +
6:              ▷ Condition variable that tells thread to wait
7:              cv.wait()
8:              ▷ Decrements by 1 after thread is no longer waiting
9:              threads_waiting − −
10:         end while
11:         temp ← fringe.pop()
12:         index, value ← MIN_POSSIBLE_VALUES(grid.unsolved)
13:         REDUCE(index, value, grid)
14:         if VALID_GRID(grid') then ▷ grid' is the derived grid
15:             fringe.push(grid')
16:             explored.push(grid')
17:             ▷ Condition variable to tell waiting threads that
    work is available
18:             cv.notify()
19:         else
20:             explored.push(grid)
21:         end if
22:     end while
23:     return grid'
24: end procedure
25:
26: procedure THREAD_SPAWN(num_threads, grid)
27:     for num_threads do
28:         spawn_thread(SOLVE(fringe, explored, grid))
29:     end for
30: end procedure
```

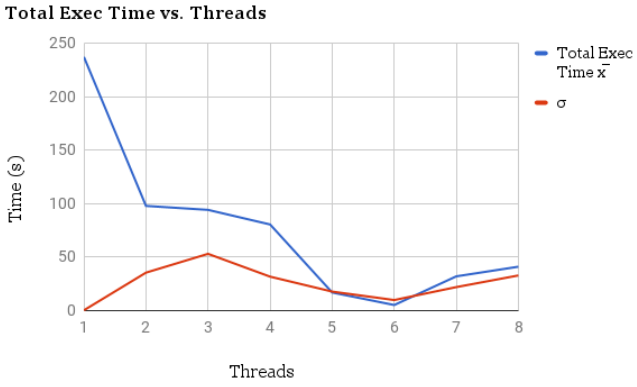


Figure 6: Graph of Sequential Time vs. Difficulty

spawned. However, two observations can also be made in Figure 6 in regards to the increased execution time after 6 threads as well as the large standard deviation.

One hypothesis about the increased execution time after 6 threads is due to an increased rate at which new grids are stored on the

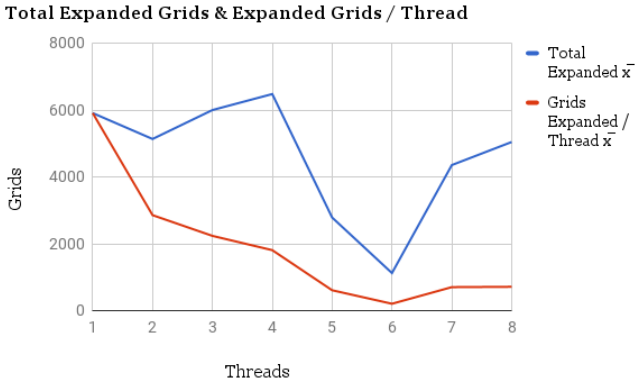


Figure 7: Graph of Sequential Time vs. Difficulty

Table 3: Speedup from 1 Thread to t Threads

Threads	Total Execution Time	Speedup
1	237.6147333	-
2	98.07906667	2.42x
3	94.37231333	2.52x
4	80.69686	2.94x
5	16.99702618	13.98x
6	5.275369348	45.04x
7	32.13147333	7.40x
8	41.14463667	5.78x

fringe, but do not lead to the puzzle's solution. This is because an increase in threads cause them to explore wider in the game subtree, rather than following a particular path in the subtree. From this, the exploration becomes more similar to breadth-first search rather than depth-first search, and thereby causing performance issues like that in Parallel Phase 1. In contrast, we found that 5 and 6 threads are optimal to solve the puzzle we tested, as their speedups (Table 3) compared to 1 thread are $13.98x$ and $45.04x$, respectively.

For the other observation, the large standard deviation is likely due to the order in which grids are inserted and removed from the fringe. Since threads that are waiting on a lock may acquire it in a random order, and also because certain grids are closer to the solution than others, the algorithm may find the solution earlier. This can be supported by the single-threaded case, where it has a near-zero standard deviation, but significantly higher execution time.

4.2.2 Results. In Figure 7, we can see that threads are expanding grids evenly to find the solution. For example, in thread 2, there was an average of 5144 grids expanded, with an average of 2861 grids expanded per thread. Furthermore, an increase in performance is evident by the graph in Figure 6. Table 3 presents the speedup of the execution time from t threads to execution time of the serial.

5 CONCLUSION

We proposed an implementation of a Sudoku puzzle solver and a reduction in execution time through parallelization and different data sharing methods. We started with a sequential version, which contained the necessary algorithms for constraint satisfaction and the validity of the grid.

From the sequential version, we derived our first parallel version, which allowed threads to work separately from one another and maintained a global explored set that was thread-safe. This first version explored widely into shallow branches in the tree, which did not result in any noticeable speedups in the algorithm. We also observed that the solutions to the puzzle have a minimum depth in the tree.

From this observation, we created the second parallel version, which allowed threads to work together on the same branch in the tree to quickly explore its depth. As a result, there was a significant speedup in the algorithm, giving us an increase between 2.42x and 45.04x, depending on the number of spawned threads.

6 THINGS LEARNED

Many programming and analysis techniques were learned throughout this project. In programming, we learned a few new algorithms and parallel programming techniques to increase the performance of the program, such as condition variables and constraint satisfaction. We also learned about the impact on performance from different thread-spawning and data sharing methods. In analysis, we learned the importance and necessity to frequently check the correctness and performance of each part of implementation process. By doing so, we can reduce the amount of time debugging and focus on improvements to the program.

As a whole, we were able to observe a significant difference in performance between different thread management and data sharing methods. Particularly, it is optimal to quickly search the depth of the tree to find the solution to the Sudoku puzzle, rather than widely searching shallow branches. Thus, changes to the algorithm may be a necessary part to ensuring speedups to the project at hand.

ACKNOWLEDGMENTS

The author thanks Keval Vora and Zachary Benavides for the guidance and knowledge in assisting with the algorithm and final version of this paper. This project would not as educational without their support and assistance.