N×N Sudoku Solver

Sequential and Parallel Computing

Abdulaziz Aljohani
Computer Science. Rochester Institute of Technology, RIT
Rochester, United States
aaa4020@rit.edu

William Smith
Computer Science. Rochester Institute of Technology, RIT
Rochester, United States
wjs3641@rit.edu

***Abstract*— 'Sudoku' is a logic-based number-placement puzzle. It is a popular Japanese game and a type of Latin square puzzle with additional constraints. Many algorithms have been published for solving Sudoku. Most of the existing solutions use guess-based heuristic methodology. Therefore, we have chosen larger Sudoku puzzles as a good candidate for parallelism and measuring backtracking algorithm efficiency.**

Keywords: Sudoku puzzle; Backtracking; Parallelism; Algorithm

I. COMPUTATIONAL PROBLEM

Solving Sudoku problem is an NP-Complete problem. As any problem from the same class, there is no way to find an optimal solution in polynomial running time. However, if we have a given a solution, we can check it for correctness in polynomial time. In the case of a 9*9 Sudoku, The time complexity would be O (N^M) where N is the number of possibilities for each cell and M is the number blank cells. By scaling the size of the board, we will get much larger problem since the problem is exponential in complexity. In addition, the difficulty of Sudoku board can also be ranked based on the number of empty cells it contains[1]. Table 1 shows the difficulty level for standard Sudoku[1].

TABLE I. SUDOKU DIFFICULTY

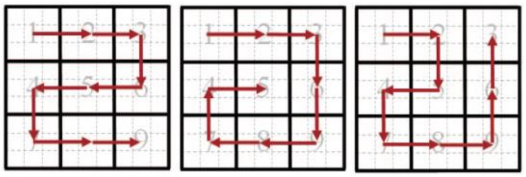
Difficulty	Number of Clues (Empty Cells)
Extremely Easy	More than 46
Easy	36-46
Medium	32-35
Difficult	28-31
Evil	17-27

II. RELATED WORK

A. Research Paper 1

Maji and Pal [1] introduce an approach that aims to solve Sudoku puzzles by using a minigrid-based novel technique that is guess-free. The proposed algorithm divides the 9*9 Sudoku into 9 minigrids, and each may identify as 1 through 9. Each minigrid may also have some prefilled numbers as clues. This approach starts with the first minigrid and then finds out the right permutations based on the number of clues that present in the same row and column within that minigrid. Then, it moves to the next minigrid. The movement from one minigrid to another can be performed in different patterns, including the Zigzag, Spiral, and Semi-Spiral, as shown in Figure 1.

FIGURE 1 Moving Patterns

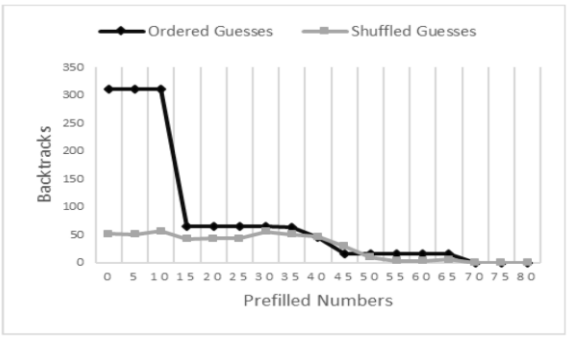


We considered using a similar approach to solving the problem by dividing the N*N Sudoku into N mini grids. We may consider using the same transition from one minigrid to another by using the approach that has been proposed above.

B. Research Paper 2

In paper [2], Schottlender discusses that the backtracking algorithm uses a list of legal numbers in the cell to attempt a solution before moving to the next cell. If this attempt does not lead to a correct solution, it backtracks and tries with another attempt with different numbers. As the errors increase, the efficiency of the algorithm decreases dramatically. Therefore, the author tests 195 base solutions with a variety of board configurations, comparing the number of recursive attempts when the guessed numbers have been sequentially selected or randomly distributed. After analyzing both approaches, he found that the shuffled guesses approach gives less errors than the ordered guesses, as shown in Figure 2.

FIGURE 2 Backtracks vs Number of Prefilled Cells



Our research project creates a smaller array of legal values by applying constraints to each square at the start. However, our implementation did not use the approach of shuffling the legal value array in order to further reduce the number of times that the project needs to backtrack due to the method of creating the list.

C. Research Paper 3

Most Sudoku solvers take a backtracking approach across all of the cells in the board, resulting in N^2 positions to backtrack on a given board. According to cited paper [3], a 9*9 board requires 17 clues at minimum to guarantee there is only a single solution. The authors proposed a method of backtracking column-wise instead of using the individual cells. This results in N objects to backtrack on, decreasing the time spent on the calculations. The authors' algorithm uses a tree to generate all of the possible permutations of each column, then tries each.

At the start of the research paper, the authors addressed multiple algorithms for generating valid backtracking solutions, which will be applicable to our research project. The paper also addressed a method of backtracking with less computations for each solution, but we decided against using that direct implementation.

III. IMPLEMENTATION

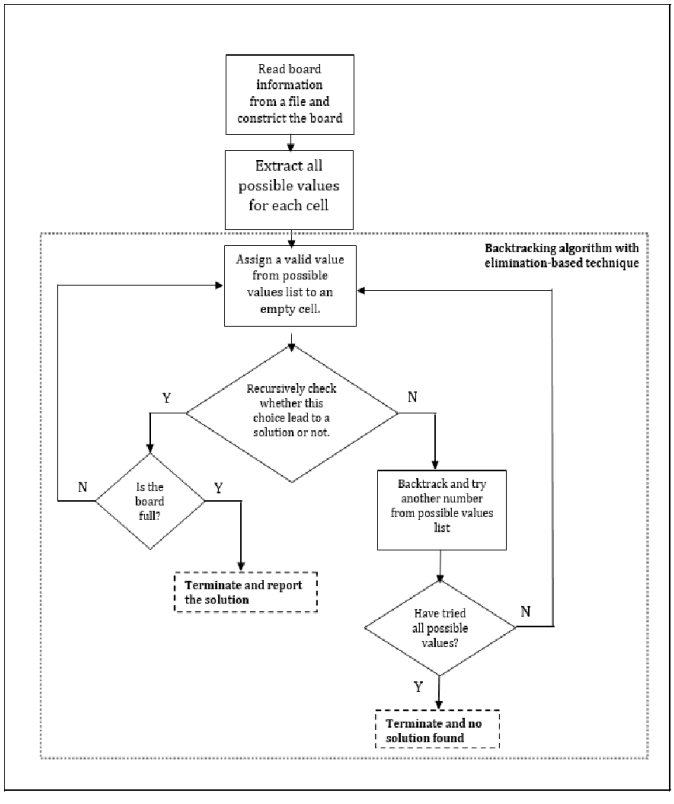
This project uses the Parallel Java 2 library developed by Professor Alan Kaminsky [4].

D. Sequential Design

Our design for sequential program uses a backtracking algorithm with an elimination-based technique (Figure 3).

Firstly, the program will receive a file that contains all of the needed information to construct a Sudoku board. This file must follow a specific pattern. In Listing 1, there are two examples of different board sizes: 9*9 and 16*16. In the first line, we need to identify the size of the board; then, from the second line, we set a series of numbers that represent the board. The value 0 in the given board means that the cell is empty; otherwise, it is a clue that was given with the board. The clues must range from 1 to the board size. Furthermore, each clue must not present twice in the same row, column, and 3*3 box when the board size is 9*9 and 4*4 box for 16*16 boards.

Figure 3 Sequential Program Flow



Next, we iterate through all columns and rows and visit each cell to generate a list with all possible values. Each cell will have its own list, which will be used while we make the recursive call attempting to find the solution. This process of elimination will let the program avoid unnecessary computation, since we exclude any redundant numbers within the rows and columns.

Finally, we find the row and column of each unassigned cell and assign a number from a possible values list that belongs to the same unassigned cell. This assignment must be valid by avoiding any conflict for the digit at the row, the column, and the 3*3 square. Then, we recursively check whether this choice leads to a solution or not. If the current choice does not lead to a solution, we backtrack and try with another digit from the cell's possible values list. If we have tried all the digits in the list, the program will terminate and report that no solution is found; if there are still digits that have not been used, though, we assign another digit and repeat the process until the program terminates. However, sometimes we might make a promising choice, then do a further check of whether the board is full or not. If it is full, we have reached the solution; otherwise, we move to the next unassigned cell and perform the whole process again until the program terminates.

LISTING 1. BOARD'S FILE FORMAT

9*9 Board

```
9
0 0 0 0 0 0 1 4 0
0 0 0 2 0 0 0 7 0
0 0 0 8 5 4 3 0 0
8 0 0 0 3 0 4 0 0
0 9 0 0 7 0 0 6 0
0 0 1 0 2 0 0 0 5
0 0 6 9 8 2 0 0 0
0 2 0 0 0 7 0 0 0
0 7 5 0 0 0 0 0 0
```

16*16 Board

```
16
12 10 8 9 2 6 15 5 4 3 7 11 14 16 13 1
1 0 0 2 4 0 14 0 0 13 0 16 0 0 0 0
11 0 0 7 13 3 0 0 1 0 2 0 0 0 10 15
4 13 0 0 0 0 0 0 16 14 8 15 12 0 0 0 0
13 0 0 11 0 9 3 2 6 15 1 8 0 0 16 14
9 3 15 0 0 0 0 0 7 10 16 14 0 7 2 5
10 0 1 0 5 0 0 0 13 4 0 0 3 11 9 7
6 2 12 0 0 16 4 0 0 0 11 9 0 0 1 8
15 9 10 16 0 0 5 0 0 0 0 4 7 1 14 6
5 0 0 0 6 4 7 0 2 9 0 0 16 10 15 11
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
7 6 2 13 9 0 0 0 0 16 14 15 8 3 5 4
8 0 0 0 14 0 0 0 0 0 6 13 5 7 4 16
14 12 0 0 16 5 0 0 0 0 4 0 1 9 3 10
2 1 3 0 15 7 0 0 0 14 9 0 6 13 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

This approach uses a special type of backtracking algorithm. We do use DFS (Depth-First Search) to take advantage of tree structures. We select one cell as a root, then explore as far along each branch as possible before backtracking to each parent cell and traversing to its children.

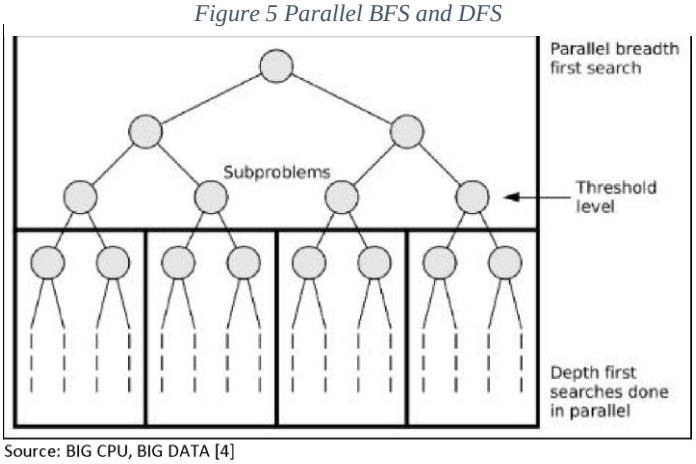
E. Parallel Design

Our parallel program uses the same algorithm that we used with sequential program with one modification. The sequential program takes the advantage of DFS (Depth-First Search) to get everything structured which is very efficient in terms of storage that it needs to visit all possible path or found the

solution for the problem. However, DFS is not easy to parallelize due to its nature. We needed to come up with another technique that help use to tackle this issue. Prof. Kaminsky proposed an idea in his book using a parallel work queue, which takes advantage of the BFS (Breadth-First Search) algorithm and the ease of parallelizing it[4]. Even though BFS is easy to parallelize, it still required large amount of space to store each traversed state in the graph. We can't use it with a problem form NP complete class such as our problem "Sudoku Solver," so we use a combination of the two techniques, similar to the approach that Prof. Kaminsky uses in chapter 13 of his book [4].

Firstly, we will start the parallel program with BFS to examine all branches of a tree until it hits a certain level which we called threshold, given with a command line. When the program reaches the threshold, it switches the strategy from BFS to DFS as shown in (Figure 4).

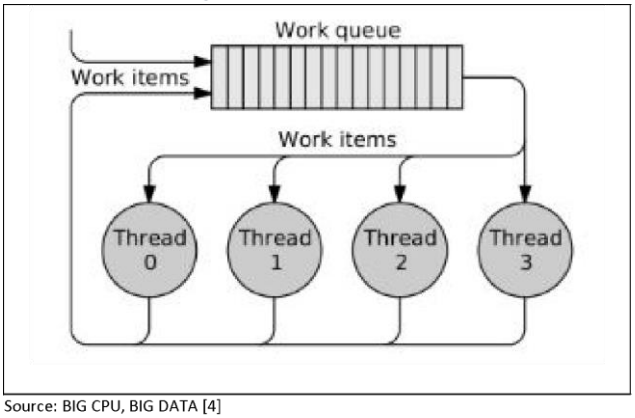
Figure 5 Parallel BFS and DFS



Source: BIG CPU, BIG DATA [4]

By using this approach, we get the benefits of both strategies and overcome the drawbacks. We were able to do the DFS in parallel and each DFS being done sequentially in a separate thread. In addition, we did not need to use large amount of storage by BFS since we are progressing to a threshold, then switching algorithms. Finally, we need to track the current level of execution in the search tree. This is done by increasing the state's level value each time BFS or DFS is run on a state. Until a threshold depth is reached, BFS adds new states to the parallel work queue. Once the threshold is met, we use an object parallel for loop over the work items in the work queue from the Parallel Java 2 Library (PJ2) (Figure 5). Each core will get a work item and perform DFS on the state, attempting to find the solution. If the core finishes it's task and does not find the solution, it will receive another Sudoku State from the work queue and all cores will keep doing this until the work queue becomes empty. Once a solution is found, that core notifies all of the other cores and threads, and informs them execution should be ended.

Figure 4 Parallel Work Queue



Source: BIG CPU, BIG DATA [4]

IV. MANUAL

F. Developer

The sequential and parallel program have been tested on both Nessie and Kraken machines on RIT Computer Science Department. Project's files contain:

- NSudokuSolverSeq.java: A drive class for the sequential version of the program.
- NSudokuSolverSmp.java: A drive class for the parallel version of the program.
- SudokuState.java: Main computation class holding the board state and solver methods. It is used by both the sequential and parallel versions but it needs different arguments to differentiate between them.

Before executing the project, we need to do a little bit of set up. The PJ2 library is installed on each of the RIT Computer Science Department's parallel computers that includes Nessie and Kraken. Before we run our program, we need to set Java classpath to include the PJ2 distribution on the CS Machines. This is done by the following bash shell command:

```
$export CLASSPATH=.:/home/fac/ark/public_html/pj2.jar
```

Then compile all java source codes:

```
$ javac *.java
```

These two commands are necessary to run both programs.

G. User

In order to run the sequential version, use the following bash shell command:

```
$ java pj2 NSudokuSolverSeq <file>
```

Where <file> is a file that contains all information about the board, and it has to be in specific pattern as shown in Listing 1.

To run the parallel version of the program, use the following bash shell command:

$ java pj2 NSudokuSolverSmp <file> <threshold>

Where <file> is a file that contains all information about the board, and it has to be in specific pattern as shown in Listing 1. <threshold> is a limit that BFS will used to determine when it switch the strategy to DFS, it must be an Integer data type.

V. DATA

H. Strong Scaling

To test the scaling performance of our program, we created a set of five test puzzles. The test set contained two 16x16 puzzles (one easy and one hard), two 25x25 puzzles (one easy and one hard), and one 36x36 puzzle. The resulting strong scaling performance is documented in Table 2, with each test being run on one, two, four, and eight cores. Figure 6 contains a graph of the running time of each test case versus the number of cores the test was running on. Figure 7 contains a graph of the efficiency versus the number of cores for that test.

TABLE 2 STRONG SCALING PERFORMANCE

TEST	CORES	RUNNING TIME	SPEED UP	EFFICIENCY
16X16 EASY	1	14684.66667	1	1
	2	7317.333333	2.00683309	1.003416545
	4	4074.666667	3.603893979	0.9009734948
	8	2441.666667	6.014197952	0.751774744
16X16 HARD	1	57239.33333	1	1
	2	22127.33333	2.586815703	1.293407852
	4	12726.66667	4.497590361	1.12439759
	8	32956	1.736841041	0.2171051301
25X25 EASY	1	48325	1	1
	2	40597.66667	1.190339346	0.5951696731
	4	26055.66667	1.854682922	0.4636707306
	8	11257.33333	4.292757314	0.5365946642
25X25 HARD	1	79829	1	1
	2	42055.33333	1.898189687	0.9490948433
	4	24094.33333	3.313185673	0.8282964182
	8	13783	5.791845026	0.7239806283
36X36	1	100234	1	1
	2	39693.66667	2.525188737	1.262594369
	4	39232	2.55490416	0.63872604
	8	39584	2.532184721	0.3165230901

As the data shows, the strong scaling is not ideal for most of the test cases. For the 25x25 easy test case, the 36x36 test case, and the eight core test on the 16x16 hard puzzle, the efficiency of our parallel algorithm was very low compared to the expected value. One reason for this is that the backtracking Sudoku algorithm we implemented relies on the Parallel Work Queue programming pattern. As work items are placed in the queue, they are guaranteed to be pulled out of the queue in first-in first-out order, but the execution of the threads that operate on these work items is non-deterministic, meaning that at lower state levels, more incorrect work items could be added to the queue in front of the correct work item state. This problem is more likely to happen as you increase the number of cores, as there is a higher chance the correct work item's thread blocks. Another case that would affect the efficiency is the case where one of the last work items in the queue is used in the derivation of the solution. In this case, the amount of time to solve the puzzle gets closer to the amount of time for the sequential solver.

Figure 7 Running Time vs Cores (Strong Scaling)

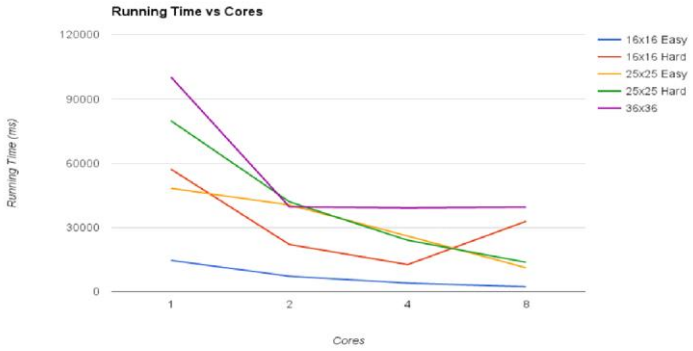


Figure 6 Efficiency vs Cores (Strong Scaling)

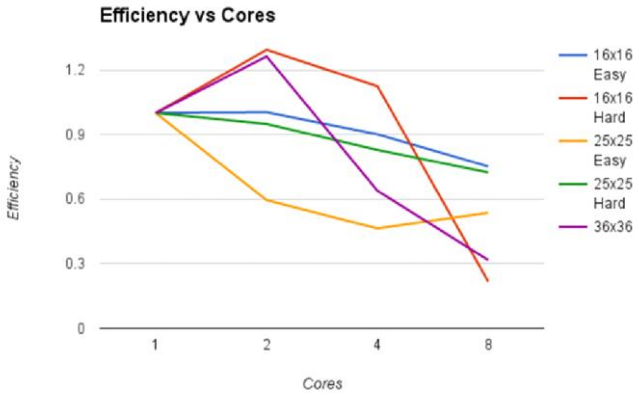


TABLE 3 16X16 ANOMALY

Test	Cores	Running Time	Speed Up	Efficiency
16x16 Hard	2	10790.33333	5.304686293	2.652343147
	4	3063.666667	18.68327712	4.67081928
	8	932	61.41559371	7.676949213

One interesting anomaly we found while testing the proper threshold for each test case can be found in Table 3. The test results from running the 16x16 hard test with a threshold value of 14 instead of 12 resulted in a high speed up and efficiency on all core tests. This is most likely due to the fact at least 14 positions were filled at the top of the board, and most of the running time in this problem space is caused by backtracking on the earlier positions.

I. Weak Scaling

For the weak scaling performance test, we used the same set of test puzzles from before, but modified each puzzle to fill in additional spaces on the board. By doing this, we attempted to create test cases where each puzzle was quantifiably more difficult that the test before in order to calculate the weak scaling The results from the weak scaling tests are documented in Table 4, showing the test cases, the number of cores used, the running time, size up and finally the efficiency received. Figure 8 contains a graph of the running time of each test case versus the number of cores the test was running on. Figure 9 contains a graph of the efficiency versus the number of cores for that test, and Figure 10 contains a graph of the size up versus the number of cores for each test.

From our data, we found that our weak scaling for our project was non-ideal for several cases, and overly ideal in other cases. The most likely cause of this is the exponential increase in difficulty of puzzles based on how many open spaces there are. For our weak scaling test cases, we tried to use the worst case scenario of one additional empty space equating to 16 additional possible states, but this is not always the case. In addition, the added values for the smaller number of cores in some cases, such as in the 25x25 Hard test case, creates a trivially easy board, resulting in drastic slowdowns as the puzzle get more difficult. Due to this reason, the 8 core test cases show low efficiency for several tests, but the data gives good or better than possible efficiency for other tests. As such, it is better to look at the strong scaling performance results for our algorithm as they give a better sense of the overall parallel efficiency.

TABLE 4 WEAK SCALING DATA

TEST	CORES	RUNNING TIME	SPEED UP	EFFICIENCY
16X16 EASY	1	1077.333333	1	1
	2	782.6666667	2.752981261	1.37649063
	4	361.3333333	8.944649446	2.236162362
	8	2615	1.647928617	0.2059910771
16X16 HARD	1	24604	1	1
	2	10786.66667	4.561928307	2.280964153
	4	8570.666667	11.48288737	2.870721842
	8	25733.66667	7.648812839	0.9561016049
25X25 EASY	1	21319	1	1
	2	10385.33333	3.079198228	1.539599114
	4	4139.333333	15.45103881	3.862759704
	8	25154.33333	5.085167566	0.6356459457
25X25 HARD	1	113	1	1
	2	372	0.6075268817	0.3037634409
	4	29258.66667	0.01544841415	0.0038621035
	8	15883.33333	0.05691500525	0.0071143757
36X36	1	40735	1	1
	2	43031	1.893286236	0.9466431178
	4	35684	4.566192131	1.141548033
	8	39584	8.23261924	1.029077405

Figure 8 Running Time vs Cores (Weak Scaling)

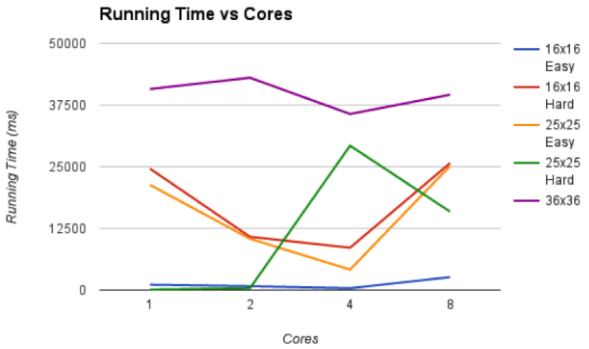


Figure 9 Efficiency vs Cores (Weak Scaling)

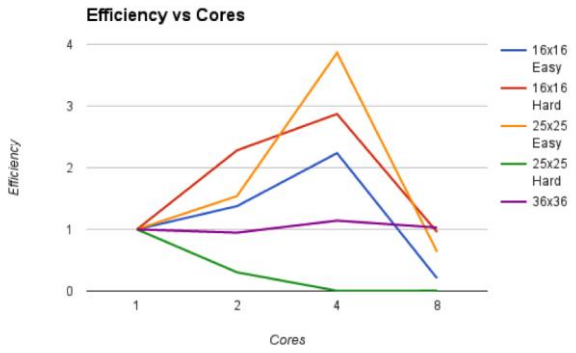
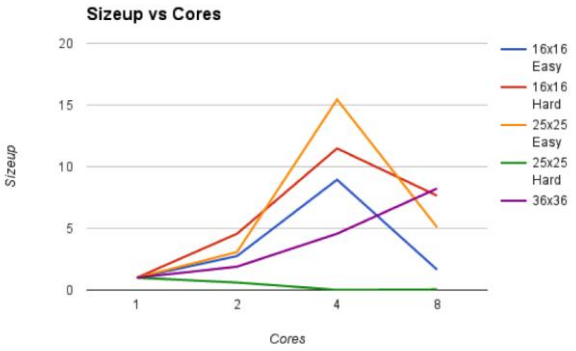


Figure 10 Size-Up vs Cores (Weak Scaling)



VI. Discussion

J. Future Work

Our solution uses a backtracking algorithm with an elimination-based technique and a parallel work queue from the PJ2 library. If we are still not satisfied with the performance after we complete the project, we are planning to redesign our program entirely. This time, we will implement a parallelized CUDA program that can efficiently solve a Sudoku puzzle using a backtracking algorithm. We will compare this solution with the previous one and see if there are any improvements in the performance.

K. What we learned?

Overall, we learned a lot while working on this project. We learned how to focus on one single problem and narrow the search based on that problem by reading and analyzing research papers specific to our topic. We also learned different approaches to solve Sudoku puzzles and picked the backtracking algorithm as our candidate. Most importantly, we dug deep inside the PJ2 library and explored many different features of it.

L. Team Contribution

Overall, this was a collective effort by both team members. We worked on all tasks together, such as preparing the report and making the design decisions for both the sequential and the parallel programs. For the coding, we set a weekly schedule to work on the sequential and parallel programs. Abdulaziz was responsible for the initial draft of the final write-up and William was responsible for the initial design of the code. However, once the initial design was written, Abdulaziz was integral in implementing the code, especially the parallel portion of the project. In addition, the initial draft of the paper was added to and revised multiple times by William. Overall, we both feel that the project was very evenly split between both members and the work was completed equally each member.

VII. References

[1] Maji A. and Pal R. (2014). Sudoku Solver Using Minigrid based Backtracking. IEEE International Advance Computing Conference (IACC)

[2] Schottlender M. (2014). The Effect of Guess Choices on the Efficiency of a Backtracking Algorithm in a Sudoku Solver. Systems, Applications and Technology Conference (LISAT).

[3] Jana, Sunanda, Maji, Arnab Kumar, and Pal, Rajat Kumar. 2015. A Novel Sudoku Solving Technique using Column based Permutation. International Symposium on Advanced Computing and Communication (ISAAC).

[4] A. Kaminsky. (2016) Big CPU, Big Data: Solving the World's Toughest Computational Problems with Parallel Computing, RIT.