Sudoku Algorithm using Parallel Processing

Hemchand Kalugotla
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
E-Mail: kalugotla@oakland.edu

Abstract— Sudoku is a logic-based, combinatorial number-placement puzzle that has been popular since the late 20th century. Solving a *nxn* blocks tends to become increasingly difficult due to combinatorial explosion. As a result, a sequential implementation of a Sudoku puzzle solver can be time consuming. In this project, parallel implementation of the Sudoku puzzle-solving algorithm using back tracking is implemented with an aim to reduce the time to solve puzzle. Concepts of pthreads like condition variables and mutexes used in this project.

I. INTRODUCTION

A Sudoku puzzle is a nxn grid that contains numbers from 1 to n, with box size of √n x √n. A standard Sudoku contains 81 cells, in a 9x9 grid and has 9 boxes (3x3 grid). Example of this is shown in figure 1. The goal of puzzle game is to fill in the empty cells on board such that each column, row and box contains every number in the set (1 to 9). Sudoku puzzle usually comes partially filled (clues) and the difficulty varies on how many clues are provided along with the location.

Rules of Sudoku are:

- Each number must appear exactly once in each row.
- Each number must appear exactly once in each column.
- Each number must appear exactly once in each box.

There are many algorithms available to solve puzzle [1]. However, as the size of puzzle gets larger, the combinational explosion occurs and thus leads to exponential growth of overall solving time. This combinational explosion also creates limits to the properties of Sudokus that can be constructed, analyzed, and solved. So, this project aims at taking an algorithm and implementing the Parallelization technique to solve the puzzle more efficiently. A comparison of time consumption between sequential and Parallel execution will be done.

II. METHODOLOGY

A. Algorithms to solve sudoku

There are different algorithms available to solve Sudoku. Three different algorithms are chosen and one of them is selected for implementation in this project. Three different algorithms chosen are: backtrack, rule-based and Boltzmann machine. A short description is given below with further in-depth details on algorithm selected for implementation in the following subsections.

- Rule-based:

This method uses several rules that logically proves that a square either must have a certain number or roles out numbers that are impossible (which for instance could lead to a square with only one possible number). This method is very similar to how humans solve Sudoku and the rules used are in fact derived from human solving methods. The rule-based approach is a heuristic, meaning that all puzzles cannot be solved by it.

- Boltzmann machine:

The Boltzmann machine algorithm models Sudoku by using a constraint solving artificial neural network. Puzzles are seen as constraints describing which nodes that cannot be connected to each other. These constraints are encoded into weights of an artificial neural network and then solved until a valid solution appears, with active nodes indicating chosen digits. This algorithm is a stochastic algorithm in contrast to the other two algorithms.

- Backtracking:

Backtrack is probably the most basic Sudoku solving strategy for computer algorithms. This algorithm is a brute-force method which tries different numbers, and if it fails it backtracks and tries a different number. Backtracking uses depth first search because it will completely explore one branch to a possible solution before moving to another branch.

B. Algorithm uesd for Project

Among the three algorithms described above, backtracking algorithm is chosen for project because:

- It guarantees a solution if input puzzle is valid.
- It has exhaustive search involving lot of iterations. Parallel processing can be used to reduce number of iterations/times to solve the puzzle.

Detailed explanation of the algorithm along with the flowchart and implementation has been shown in the following sections.

C. *Backtracking Algorithm*

A common algorithm to solve Sudoku boards is called backtracking, which is a type of brute-force search. This is essentially a depth-first search (DFS) by completely exploring one branch to possible solution before moving to another branch. The algorithm uses recursive calls on a test-and-backtrack approach. All the cells of the board will be visited.

- Upon arriving, if the cell is not empty, it is skipped. Otherwise, temporary value is assigned from 1 to 9. In below figure, we can observe that the cells (0,0) and (0,1) are skipped and that the value of '1' has been assigned to the next cell (0,3)

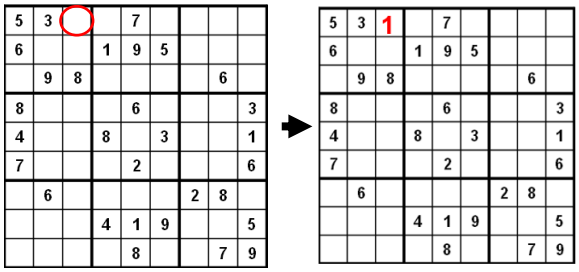

Fig 1: Assign temporary value in first non-empty cell.

- After this, a test is performed to see if the attempted value is allowed there. This is done by scanning the 3 dimensions of neighbors, the new value is compared to all the elements contained in its row, its column, and its sub square.
- If the test fails, the next value is tested, and the algorithm repeats the test. If all the values have been exhausted, the algorithm backtracks to the previous cell.
- This process is repeated until program finds element that could potentially exist at the location.
- Once this condition is satisfied, the function is recursively called to the next location.

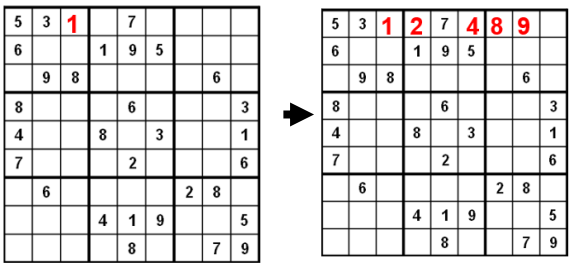

Fig 2: Puzzle after few iterations

- Once there are no unsolved cells, we know that all the cells have been visited. If no solution was found, we backtrack one cell and keep trying all the possibilities on that one by recalling the recursion.
- This keeps happening until all the possibilities have been tried out.

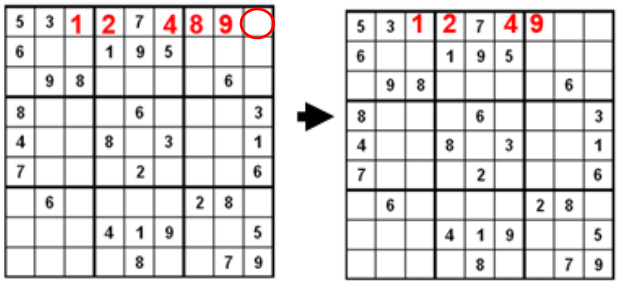

Fig 3: Deadlock and backtracking to previous cell

This algorithm is better drawn in the form of a flowchart and shown the figure 4 below. Details of Sequential and parallel implementation of this algorithm are mentioned in follow subsections.

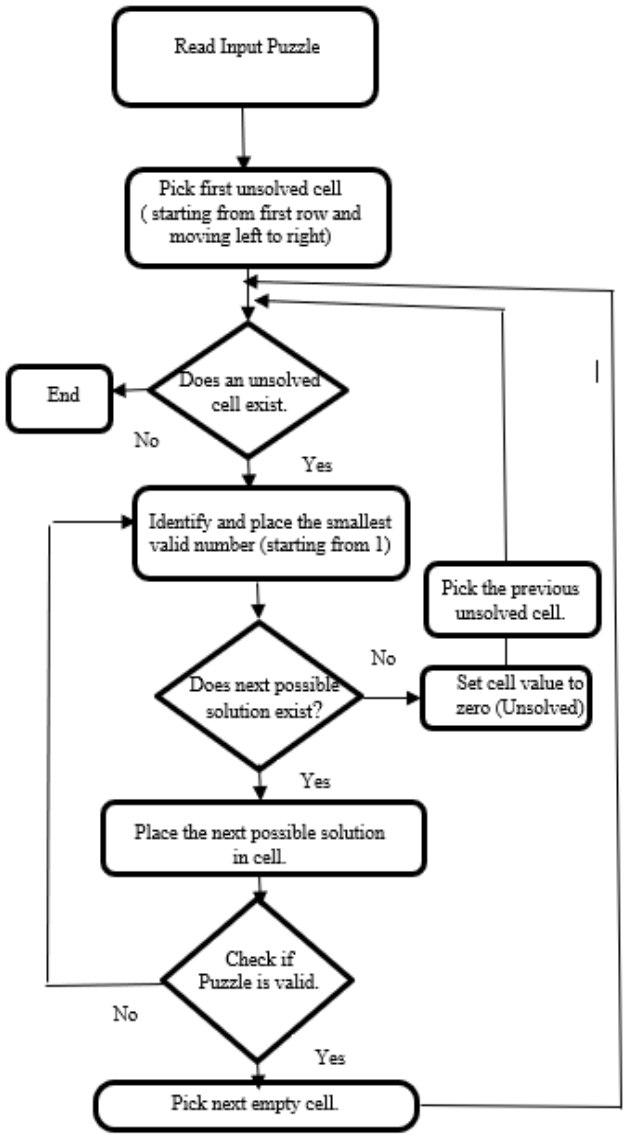

Fig 4: Flowchart of the algorithm.

D. Sequential approach of Bracktracking algorithm

Algorithm can be easily understood looking at the animation available in Wikipedia [1]. Sequential implementation is very simple and can be implemented using a recursive function. The sequential algorithm is implemented iteratively, and it simply explores the tree of all the legal candidates' assignment of each empty cells.

This serial approach has been implemented first and verified on laptop and on Terasic board. Definition of the function is shown below:

```
int solveSudoku ()
{
   int row, col,num;
   if (!findEmptyPosition(&row, &col))
  {
      return 1;
  }
  for (num = 1; num <= SIZE; num++)
 {
      if (isSafe(row, col, num))
     {
         sudokuGrid[row][col] = num
         if (solveSudoku())
        {
            return 1;
         }
         sudokuGrid[row][col] = 0;
     }
  }
  return 0.
}
```

Input puzzle is hardcoded in the software and needs to be re-compiled if the input needs to be changed. All the empty cells needs to be shown as '0' for the algorithm. As shown above, sequential implementation is very straight forward and makes use of recursive functions and if the function returns '0', it fills the current position with '0' and back tracks to the previous cell.

E. Parallel approach of Bracktracking algorithm

Parallel approach of this algorithm is little tricky as working on each cell/row/column/block in parallel is difficult. This is because of following reasons:

- Every cell need to know its peers to make a decision. So, work done in a thread is dependent on other threads or must wait till other threads finish the work.
- It is hard to determine what was the previous step and backtrack to it if multiple threads are doing same task.
- Thread may have to create another thread dynamically making the synchronization more complex.

Because of these reasons, below shown approach has been chosen for parallel implementation:

- Create a board for each thread with all permutations of valid numbers possible.
- Use the boards created as an input for each thread.

This approach chosen is better explained along with comparison with sequential implementation is shown below:

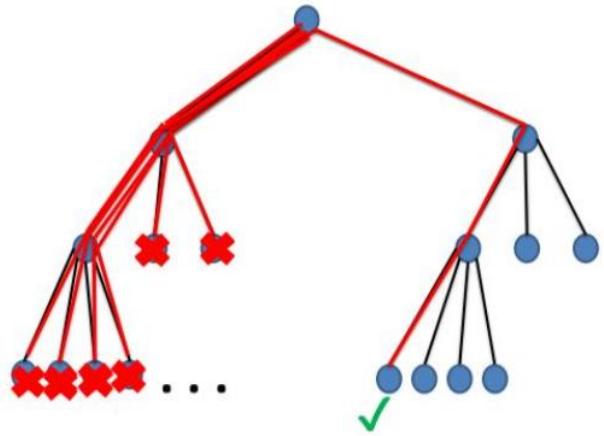

Fig 5: Iterations required for Sequential processing.

The puzzle shown in this example has solution in first child in level 4 on right hand side. Figure 5 shows the details on how this puzzle is solved using sequential approach. Algorithm checks all the child nodes in level 4 on left hand side and then back tracks to level3. As the solution is not found, it searches for the node in level 2 on the right-hand side to find the solution.

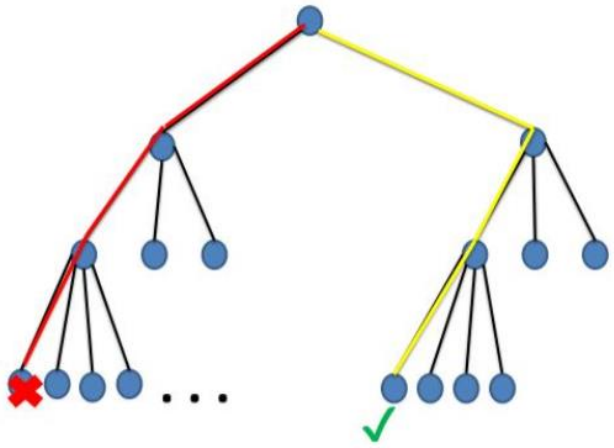

Fig 6: Iterations required for Parallel processing.

With parallel processing, two threads can be created with second thread trying to search for the solution on the right-hand side directly. With this number of iterations required to solve the puzzle will be less as the second thread solves puzzle in few steps. So, the time required to find solution will be less compared to sequential approach.

F. Implementation using pthreads

Since the challenge is to test a very large possibility space, the use of parallel computing can greatly aid. For this problem, it has been tested with many threads to solve sudoku puzzles of different complexity levels. **pthreads** has been used as a platform because of its ease of implementation using the concepts learnt in this course[5].

The main takes care of creating threads along with the input puzzle for each thread with all the valid permutations possible for each empty cell. Every thread takes its own input puzzle and tries to search for solution. As soon as one of them solves the problem, it must signal main along with updating solution. Then main takes care of printing out the solution and the time it took to solve it along with canceling other threads that are in process. To handle synchronization, we used a global variable called 'puzzle_solved'. When the main function starts, if sets 'puzzle_solved' to false. As soon as a thread solves the puzzle, it sets the variable to true. Meanwhile, in the main function, after all the threads have been initialized, the program falls into a while loop. As soon as the barrier is crossed, we know that a thread has solved the puzzle, so the solution is print along with the timings and all the threads are destroyed.

To accomplish this, concepts of **mutex** and **condition variables** learnt in this course has been used. The below paragraph in this subsection shows some screenshot of source code used in parallel implementation Below is screenshot from code showing the declaration and initialization of global variables, mutex and condition variables.

```
int puzzle_solved = 0;
int (*solution)[SIZE];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

If solution is found, threads will:

- Lock Mutex
- Update global variables indicating puzzle solved along with solution.
- Issue Signal condition that puzzle is solved.

Below is the screenshot of source code used in the project. Global variable 'sol' stores index of thread that solved puzzle, 'solution' has puzzle solution.

```
pthread_mutex_lock(&mutex);
puzzle_solved = 1;
sol = (*args_ptr).index;
solution = (*args_ptr).tempGrid;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

Main will:

- Lock Mutex
- Wait until the puzzle is solved by any thread.

```
// waiting for solution
pthread_mutex_lock(&mutex);
while(puzzle_solved == 0)
{
    pthread_cond_wait(&cond,&mutex);
}
pthread_mutex_unlock(&mutex);
```

All the threads will still be running and trying to find the solution. As the solution is already available, all these threads can be canceled, one approach is to use pthead_cancel in main once solution is obtained. This approach is being implemented in the project and uses pthread_cancel. Another approach that is not implemented in the project is to have all threads exit if puzzle is solved and main can wait till all threads complete. This can be done by using signal broadcast and pthread_join in main.

III. Experimental Setup

Sequential implementation of algorithm is implemented in C language first on laptop running windows and verified with multiple puzzles as input. Once the sequential implementation is working, it has been verified on Terasic DE2i-150-FPGA development kit. Parallel implementation is implemented and verified directly on the Terasic board. As mentioned in above sections, puzzle is hardcoded inside the source file and need to be compiled if a different puzzle needs to be solved.

IV. Results

Project has been verified with different input parameters for a stand sudoku puzzle of 9x9. Input puzzle has been categorized as Easy, medium, and hard based on the level of backtracking required to solve. Puzzles have been taken from the dadsworksheets.com [2]. Results for each of the input types along with the observations is shown below:

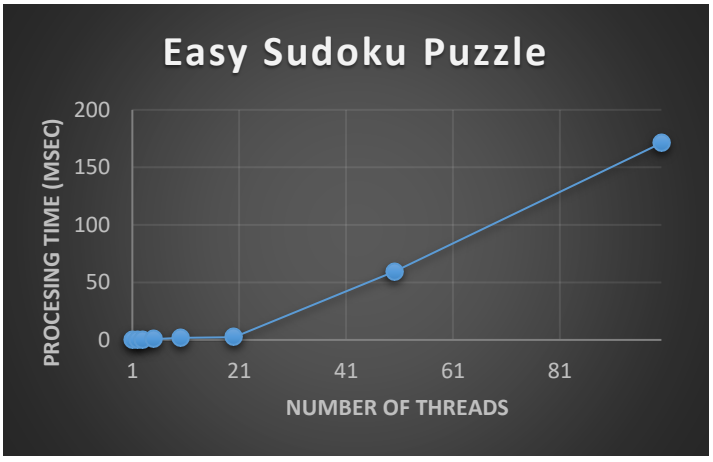

Fig 7: Processing Time comparison for Easy Sudoku

Observations for Easy sudoku puzzle execution:

- No Backtracking required.
- Processing time for Sequential execution is 98.6usec and 443usec for Parallel processing with 2 threads.
- Search Path for solution is very less, so sequential approach is better in this case and adding number of threads increases processing time.

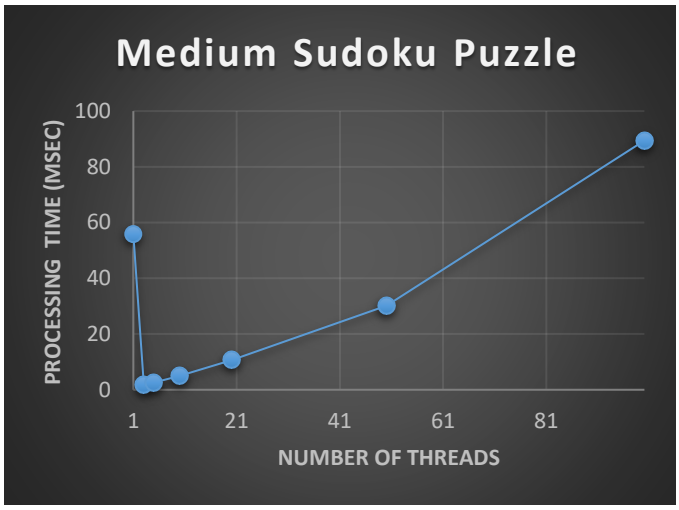

Fig 8: Processing Time comparison for Medium Sudoku

Observations for Medium Sudoku puzzle:

- Minimal Backtracking required.
- Processing time for Sequential execution is 55.7msec and 1.7msec for Parallel processing with 3 threads.
- Even after the solution is found, time to find the solution increases:
    - Amount of time each thread gets is less if there are more threads created.
    - Overhead in creating threads.

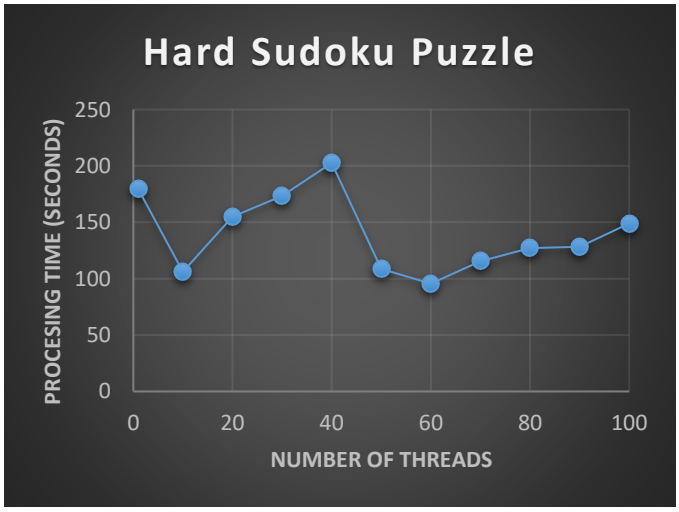

Fig 9: Processing Time comparison for Hard Sudoku

Observations for Hard Sudoku puzzle:

- Input puzzle has lot of empty grids in the beginning and multiple backtrackings required to solve the puzzle.
- Sequential approach took around 3 minutes to solve. While parallel approach (best) took ~1.5 minutes
- Execution time reduces if the number of input grids crated is close to the solution.

CONCLUSIONS

Sudoku puzzle game with backtracking algorithm has been implemented using both sequential and parallel approaches. Adding number of threads to solve the sudoku puzzle will reduce the time to solve only if there is a solution found in search path. If the solution is not found, execution time increased because of the overhead to create threads. This can be seen clearly in all the complexity level testing done. For further research, more efficient thread mechanism can be created covering the complex cases such as, when a thread cannot solve the puzzle in its search path, new work can be assigned to that thread.

REFERENCES

[1] https://en.wikipedia.org/wiki/Sudoku_solving_algorithms#Brute-force_algorithm

[2] https://www.dadsworksheets.com/puzzles/sudoku.html

[3] https://alitarhini.wordpress.com/2012/02/27/parallel-sudoku-solver-algorithm/

[4] https://hpc-tutorials.llnl.gov/posix/

[5] Tutorials from course.

[6] (PDF) Recursive Backtracking for Solving 9*9 Sudoku Puzzle (researchgate.net)

[7] Depth-first search - Wikipedia