Parallel Sudoku Solver

Guangyu Chen
(guangyuc@usc.edu)
Xiaowen Tan
(xiaowent@usc.edu)
Siqi Chen
(chen142@usc.edu)
University of Southern California

Introduction

Sudoku is a puzzle in which an N^2x N^2 board must be filled with N^2 numbers each of 1 through N^2 such that each row, column, and N x N box only contains each number once. A traditional Sudoku board in 9 x 9 shape, which means there could be 6×10^{21} valid Sudoku puzzle in total in this shape. Also, solving a Sudoku puzzle is an NP-hard problem, meaning there is no known polynomial-time solution. This makes Sudoku a great problem for parallelization.

In this project, we focus on solving Sudoku by implementing three distinct algorithms backtracking, Crooks Algorithm and SAT (Boolean satisfiability problem) solving in paralleled style. Backtracking is the most straightforward solution for this problem, it simply tries every possible combination until it finds a solution. Crooks algorithm is a sudoku specific implementation, which is utilizing some method to rule out some combination. Sudokus are known to have an equivalent mapping to SAT formula, so we plan to make an SAT solver that can solve sudoku formulas parallel.

Context

Simple Backtracking [6]

According to the rule of Sudoku:a number is not allowed to appear more than once in the same line, same column and the same 3 x 3 box, which means each number we fill in an empty spot is constrained by other numbers that already in the table and is a new constraint to the next empty space we are going to fill in. Moreover, there would be more than one number that can be filled in one space when the existing numbers do not give enough constraints. As different numbers are filled in, different constraints will be added to the rest empty spaces, different results will be achieved. If we list all possible solutions of sudoku, it would appear like a tree as the graph shown below: at each level, one empty spot was filled and each child node represents

the table filled with one possible number form the last table.

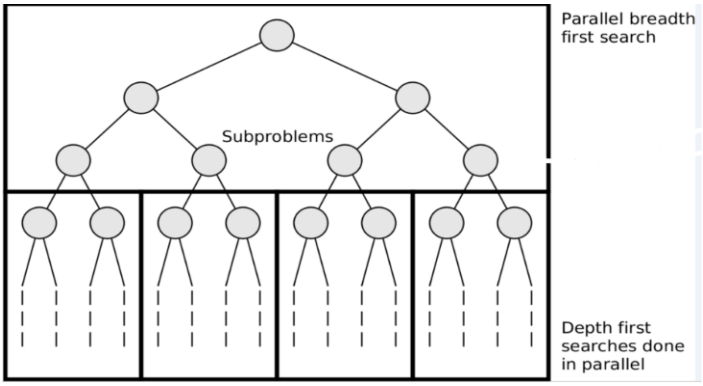

Fig 1: Source: BIG CPU, BIG DATA [1]

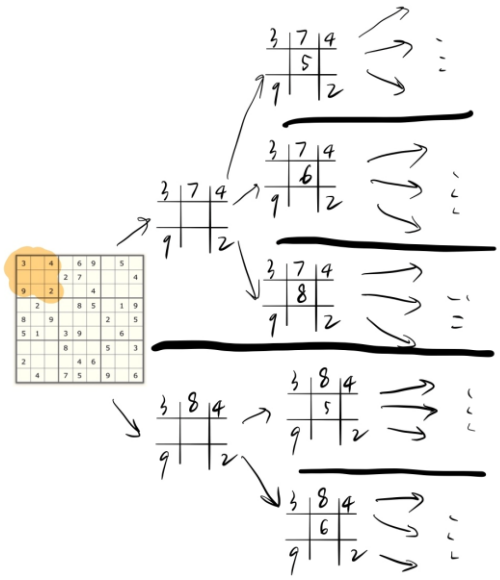

Fig 2: Example of a Sudoku solution tree

The above graph gives a more visual example of why we can analog the Sudoku solving process into a tree.

As long as there is always at least one solution for valid sudoku, the most direct way is to traverse the tree. In considering each node in the tree represents a table that is storage consuming, DFS(Depth-First Search) is used to traverse this tree.
When we get more computing resources(using GPU for computing), we can traverse paths in the tree in a BFS(Breadth-First Search) way. Since paths are distinct from each other: one would not affect one another, each path is assigned to one thread and basically, each thread is doing a DFS operation and all threads are doing this operation at the same time.

Below is the pseudocode of my implementing strategy:

Algorithm 1: Sudoku Solver -- parallel backtracking

```
Initialization:
        SudokuTable <- read from file;
        Init Cuda memory:
                newTable;
                oldTable;
                emptyspaceindex;
                emptyspacecount;
copy SudokuTable to oldTable;
assign one table of oldTable to each thread;
iter <- 0; //can be changed


//generating more tables level by level
for iter=0 to 20:
        Thread(i) do:
                traverse table, find an empty spot index(a,b);
                for j=1 to 9:
                        check whether i is valid at the spot;
                        if yes:
                                copy the table to newTable at index j;
                                assign number i to newTable(j) at index(a,b) ;
                                add index(a,b) to emptyspaceindex(i);
                                emptyspacecount(i)+=1;
                        end if
                end for
        end do
        copy newTable to oldTable;
end for

//assign tables to each thread to do parallel backtracking
Thread(i) do:
        get one table from oldTable(i);
        traverse table, find an empty spot index(a,b);
        if find:
                recursively fill all the empty spot, using empty space index and empty space count
                for help;
        else
                find a valid solution;
                copy the table back to cpu memory;
        end if
end do
```

Crooks

Since there are numerous possible combinations, a naïve backtracking algorithm would cost a lot of resources and time. Thus, this algorithm deletes some combination according to two patterns and then run the backtracking algorithm, which may largely reduce the time for computation.

In order to introduce two patterns, here used a Sudoku puzzle as an example:

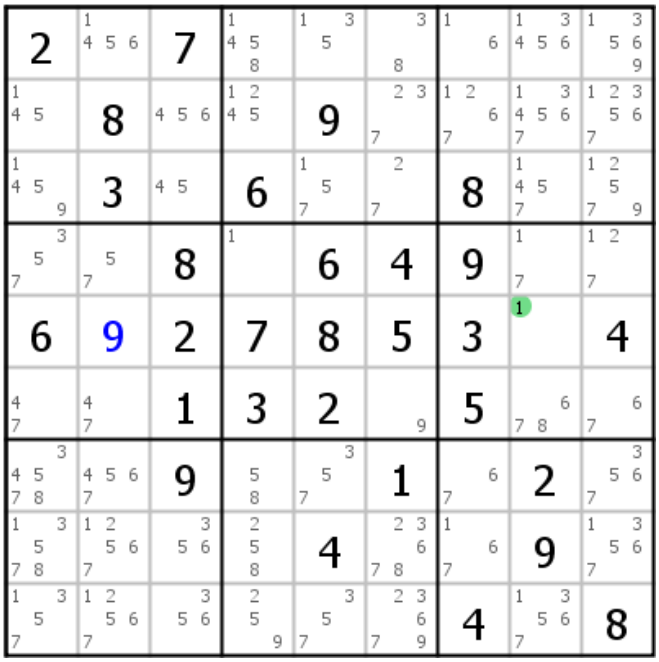

Fig 3 sudoku example[2]

In this example, digits with small font mean candidate numbers for the block and digits with large font mean numbers already given. Also, for convenience, this report assumes that all the indexes start with 0.

The first pattern is that the block only has one candidate number. Thus, that block must be filled with its candidate number. In the sudoku example, block(4,6) in the grid(1,2) only has one candidate number which is 1. Thus, in the solution, the block(4,6) must be filled with 1.

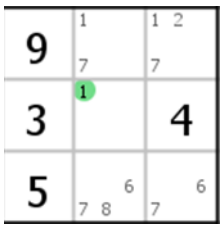

Fig 4 gird(1,2)

The second pattern is a certain kind of preemptive set. This kind of set has two

characteristics. The first one is that the number of blocks in this set must equal to the number of different candidate digits in this set.

The second one is that all the blocks in this set must belong to a grid, a line or a column. Then other blocks in the same grid, line or column with this set will not be filled by candidate digits in this set, which would largely reduce the combination numbers in sudoku solving process. This is because if other blocks can be filled with a candidate digit in the set, the number of candidate digits would be smaller than the number of blocks in the set which means that this sudoku does not have an answer.

For example, in column(6), block(0,6), block(5,6) and block(6,6) can form a preemptive set. This set has 3 candidate digits {1,2,6} and 3 blocks. Thus other blocks, such as block(1,6) cannot have {1,2,6} as candidate digits. Then block(1,6) would have only one candidate digit {7}, which would belong to the first pattern.

Fig 5 column(6)

However, in computation, it has a disadvantage that is if one sudoku does not have these two patterns, it would cost a lot of time to find these two patterns, which would cost more time than naïve backtracking algorithm.

The pseudocode shows as follows:

Algorithm 2: Sudoku Solver-parallel backtracking

The first parallel part begins:

1.1 check every block's situation to see whether it would match two patterns.

1.2 if it matches the first pattern, the block will be filled with its candidate digits.

1.3 if it matches the second pattern, other blocks delete the candidate in the set.

first part ends

The second parallel part begins:

2.1 using backtracking algorithm to go through the left combinations to find the solution

second part ends

SAT [4][5]

Since there are more than 6×10^{21} possible Sudoku grids, a naïve backtracking algorithm would be time-consuming. Sudoku solvers, therefore, combine backtracking with methods for constraint propagation. A Sudoku can be translated into a propositional formula that is satisfiable if and only if the Sudoku has a solution. The propositional formula can be presented to a standard SAT solver, and if the SAT solver finds a satisfying assignment, this assignment can be transformed into a solution for the original Sudoku.

From the paper " Sudoku as a sat problem ", we find a formula to transform a Sudoku to SAT formula. Given a Boolean formula, the SAT problem asks for an assignment of variables so that the entire formula evaluates to true. Let us use the notation s_{xyz} to refer to variables. Variable s_{xyz} is assigned true if and only if the entry in row x and column y is assigned number z. For each rule of Sudoku, we can have a corresponding encoding asserts.

There is at least one number in each entry:
$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigvee_{z=1}^{9} s_{xyz}$
Each number appears at most once in each row:
$\bigwedge_{y=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{x=1}^{8}\bigwedge_{i=x+1}^{9}\left(\neg s_{xyz} \vee \neg s_{iyz}\right)$
Each number appears at most once in each column:
$\bigwedge_{x=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{y=1}^{8}\bigwedge_{i=y+1}^{9}\left(\neg s_{xyz} \vee \neg s_{xiz}\right)$
Each number appears at most once in each 3x3 sub-grid:
$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=y+1}^{3}\left(\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+x)(3j+k)z}\right)$
$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=x+1}^{3}\bigwedge_{l=1}^{3}\left(\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+k)(3j+l)z}\right)$

In this encoding, the resulting CNF formula will have 8829 clauses, apart from the unit clauses representing the pre-assigned entries.

After having the CNF (conjunctive normal form) formula, we can use the algorithm for the SAT problem to solve the Sudoku problem. For this project, we choose a Conflict-driven clause learning (CDCL) algorithm for solving the SAT. This algorithm was inspired by the Davis–Putnam–Logemann–Loveland (DPLL) algorithm.

DPLL algorithm has some similarity with backtracking, it solves an SAT problem as a decision tree. DPLL is based on search, if there is a conflict, it will backtrack to immediate level and assign opposite value to that variable. Continue in this way and finally, it will find the solution to the SAT problem.

The main difference between CDCL and DPLL is that CDCL's backtracking is non-chronological. Conflict-driven clause learning works as follows:

Algorithm 3: Sudoku Solver-parallel backtracking

1. Select a variable and assign True or False. Remember the assignment.
2. apply Boolean constraint propagation.
3. Build the implication graph.
4. If there is any conflict
    1. Find the vertex in the implication graph that led to the conflict
    2. Find a new clause which is the negation of the assignments that led to the conflict
    3. Backtrack to the appropriate decision level, where the first-assigned variable involved in the conflict was assigned
5. Continue from step 1 until all variable values are assigned.

The master-worker model is used for parallel CDCL. In the beginning, all workers are free. The master starts solving by giving an empty partial model to the first free worker. When a worker reaches a branching point, it takes one branch and sends the other branch as a partial model to the master. When the master receives a partial model, it checks whether there exist some free workers. If not, it stores the partial model into its own queue. Otherwise, it sends it to a worker, which finished solving a branch with conflict results and therefore is free and needs further work. This procedure is repeated until some worker finds a result and sends it to the master. The master then outputs the model and stops all workers. If no worker can find a result, the master has an empty queue and all workers are free, then we know that the formula cannot be satisfied. Based on this parallel model, MPI (Message Passing Interface) is a good parallel method for implementing this model.

Objective

Implement both the serial backtracking algorithm and parallel backtracking algorithm using the Cuda framework. Test both code on HPC server with multiple Sudoku problems in different difficulty, compute execution time for further analysis.

For Cuda code, run with different block(1,8,16,32,64,128,256,512,1024) and grid(1, 8, 32, 64, 128) settings.

We plan to implement the crook algorithm with 9 threads utilizing Pthread framework.

We plan to implement a serial CDCL algorithm and measure its execution time. Also, a parallel CDCL implementation which utilizes the Master-Worker model for parallel computing.

For the parallel part, we plan to use 4, 16, 24, 32, 64 cores for our MPI implementation and measure their execution time. Compare with the serial execution time and time for different cores. Examine the performance of serial and parallel code.

USC HPC (Center for High-Performance Computing) server is used for computing all the algorithms. Thus, we can compare our computation results in a more uniform environment compared with our laptop.

Experimental Setup

All the codes including backtracking, Crooks Algorithm and SAT solving are ran in USC HPC with the serial method and different threads or cores. All the methods use the same Sudoku set for computing such that we can compare their execution time.

Results and Analysis

In a regular tree structure, the child level always has more nodes than the parent level. However in the Sudoku tree, as we go deeper in the tree, means we fill more spaces in the tree and more constraints for the rest of the empty space. The number of possible tables will not keep increasing, at some point, the number of the newly generated possible tables will start to decreases.

Therefore, we can expect the result that the execution time decreases as the number of thread uses increases, and reaches a bottleneck at some point.

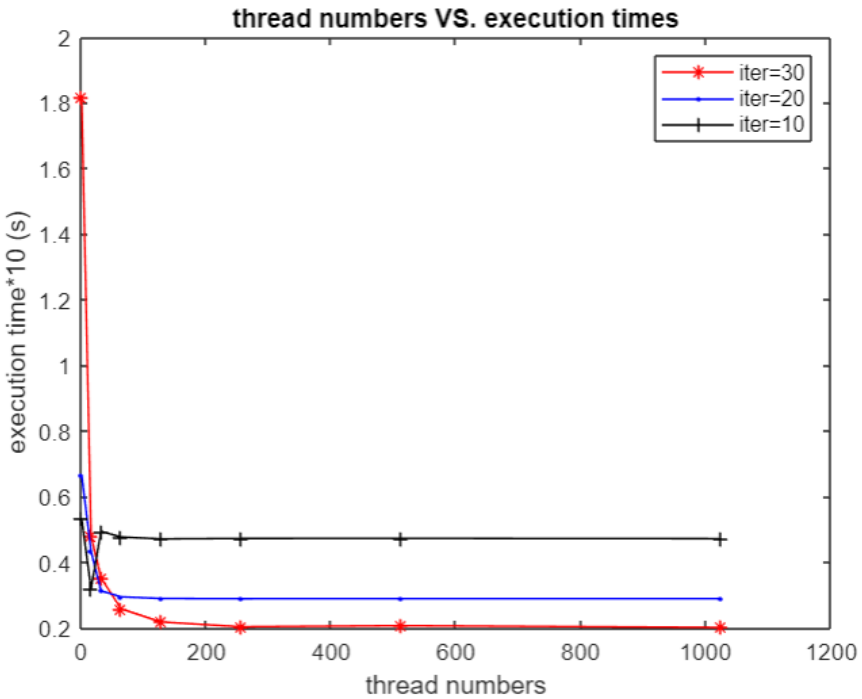

Fig 6: GPU thread numbers VS execution time*10 (s)

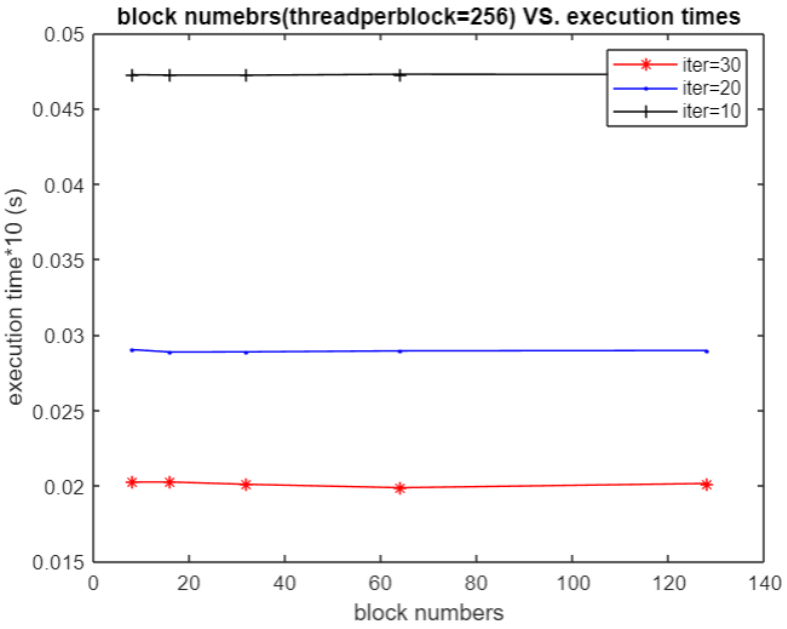

Fig 7: GPU block numbers VS. execution time*10(s)

From the above result, we can see that the relation between the execution time and the thread number is just as what we assumed. Also, the parameter iter represents how deep we go into the tree before we start backtracking, this parameter does affect the execution time, but in a small way.

The execution time of the serial backtracking program on HPC takes about 2.5s, which is 40 times slower than the Cuda program. The speedup is less than expected, this might because all memory access in my program is global memory accessing which is very time consuming and slow down the computation time.

Using a lot of sudokus to compare the Crook algorithm with naive algorithm, it seems that the Crook algorithm sometimes would be faster than naive algorithm. Also, the speedup varies according to different sudoku. And it is because not all the sudokus have these two patterns and when it does not have these two patterns, the Crook algorithm would waste time to search them.

naïve algorithm	1.11	0.23	14.87	0.13	0.09	0.09
pencil and paper algorithm	0.18	0.15	1.35	0.14	0.15	0.15
speedup	6.166667	1.533333	11.01481	0.928571	0.6	0.6

To solve Sudoku in an SAT way, the first step is to convert Sudoku input to a CNF formula. The second step is to use the CDCL algorithm serial or parallel to solve the SAT

problem. The final step is to convert the output CNF formula to Sudoku like formula.

All the execution time only considers the time for solving the SAT problem but not involve the time to convert Sudoku to SAT or SAT to Sudoku. The serial execution time for the CDCL algorithm is 5667 millisecond. Parallel execution time is as follows.

cores	2	4	8	16	24	32	64
Time(ms)	4748	1581	709	335	331	318	298

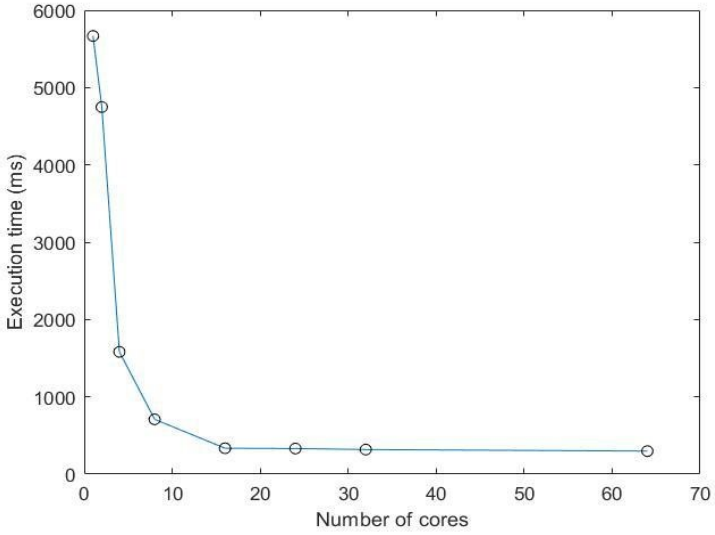

Fig 8 Execution time vs. Number of cores used

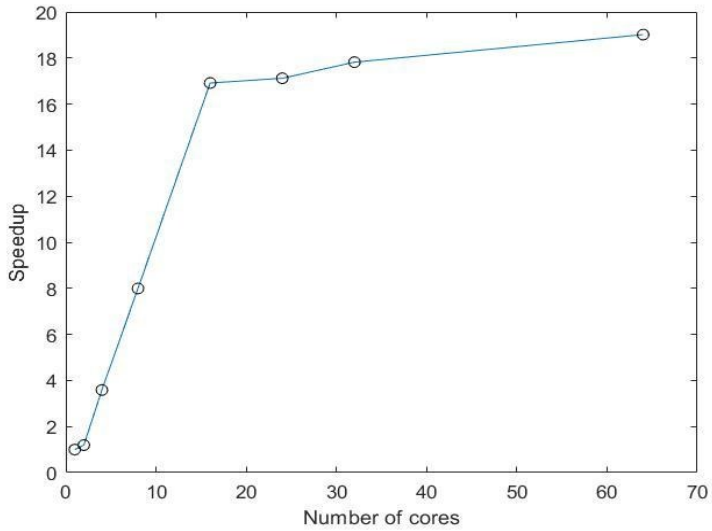

Fig 9 Speedup vs. Number of cores used

Conclusion

From the execution time and speedup data in the Result and Analysis section, we find that the parallelized backtracking algorithm indeed speeds up the Sudoku solving process. However, more improvement still can be made in the implementation. The memory accessing style affects hugely in execution time, so replace some of the global memory access with shared memory access is quite essential.

From the execution time and speedup data in the comparing section, we find that the Crook algorithm may speed up the process of calculation according to the sudoku to be solved.

From the execution time and speedup graph in the SAT section, we can find that with the increment of cores the execution time decreases. However, the speedup portion decreases when the number of cores becomes more than 16. The reason could be there is only one master in this model and too many workers. In that case, the master may not be able to assign jobs to workers as soon as possible which can lead to more idle time for workers and finally lower the speedup.

References

[1] A. Kaminisky, Big CPU, big data: solving the worlds toughest computational problems with parallel computing. North Charleston, SC: CreateSpace Independent Publishing Platform, 2016.

[2] B. Hobiger, “HoDoKu Solving Technique Index: Example for ‘Full House,’” HoDoKu. [Online]. Available: http://hodoku.sourceforge.net/en/show_example.php?file=fh02&tech=Full+House.

[3] J. F. Crook,"A Pencil-and-Paper Algorithm for Solving Sudoku Puzzles", Notices of the AMS, vol. 56, no. 4, Aprial 2009. [Online]. Available: https://www.ams.org/notices/200904/tx090400460p.pdf .

[4] limo1996, “limo1996/SAT-Solver,” GitHub. [Online]. Available: https://github.com/limo1996/SAT-Solver.

[5] Bernarpa, “bernarpa/Parallel-SAT,” GitHub. [Online]. Available: https://github.com/bernarpa/Parallel-SAT.

[6] Vduan, “vduan/parallel-sudoku-solver,” GitHub, 04-Jun-2015. [Online]. Available: https://github.com/vduan/parallel-sudoku-solver.

[7] T. Weber. A SAT-based Sudoku solver. In LPAR-12, Short paper proc., 2005.

[8] I. Lynce and J. Ouaknine. Sudoku as a SAT problem. In 9th International Symposium on Artificial Intelligence and Mathematics, January 2006.