Sudoku Solver Final Report

Team Members: Madeleine Hagar and Jennifer Chou

Project Webpage: https://github.com/jcchou12/Sudoku-Solver

SUMMARY:

For our project, we implemented sudoku solvers both sequentially and in parallel using several algorithms and parallel programming frameworks, including OpenMP and MPI. We evaluated the performance of each solver on the GHC machines, compared their respective performances, and analyzed the limitations of our implementations. Additionally, we researched ways to further enhance the performance of our parallel sudoku solvers using CUDA without actually implementing these enhancements.

BACKGROUND:

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

Example of a 9x9 Unsolved Sudoku Puzzle

We parallelized the problem of solving a sudoku puzzle. A sudoku puzzle is an NxN grid of cells with N sqrt(N)xsqrt(N) sub-grids, which must be solved by filling each cell with a number between 1 and N such that every row, column, and sub-grid contains every number without repetitions. This is a problem with a high number of dependencies between cells (every cell is directly dependent on 3N-3 other cells), making it a difficult task to parallelize.

Our key data structure was an NxN integer array representing the sudoku board. However, each algorithm we implemented had at least one other essential data structure, including a work queue and an NxNxN candidate array. The input to our algorithms is a partially filled NxN sudoku board, and the output is the completely solved board.

The part of the problem that is computationally expensive is attempting to find the correct value for each cell in the board. This could benefit from parallelism in many ways including: using multiple threads or processes to solve the board assuming certain values in unfilled cells, eliminating candidates for cells in parallel, or using multiple workers to propagate constraints to neighboring cells when a new cell is filled.

APPROACH:
Our algorithms, as described below, were all implemented in C++ and run on the GHC machines. We chose these machines due to their high core count, which allows us to achieve high performance when utilizing multithreading. All three algorithms were parallelized in a shared address space using OpenMP and the last one was additionally implemented using message passing via MPI.

Serial Backtracking
We first start with our initial board as an input.
Recursively:

- If there are no unassigned spaces in the board, we return true and print out the board.
- If there is an unassigned space, we try to assign a number from 1-N and recursively assign the next number from 1-N until the new board generated is safe and the function returns true

Serial Backtracking with Stacks
We start with our initial board as an input and push it onto a stack. Then, we pop the first board off the stack and use that to generate some number of possible boards to safely fill in a certain number of unassigned spaces and push these new boards onto the stack. We continue to pop boards off of the stack and proceed in this manner until we have no more unassigned spaces.

Parallel Backtracking with Stacks
We used OpenMP for our parallelization of the stack of boards algorithm. The following is the parallelization of the serial algorithm:

- We repeat a constant number of times to pop a board off of the stack and if there are any unassigned spaces, generate new boards (ensuring that there are no conflicts from the assigned number) from the popped board to push back onto the stack. If there were not any unassigned spaces, we could return that the board was valid and a solution had been found. This loop was parallelized using dynamic scheduling since the number of boards that were generated from each unassigned board would affect the workload of that particular thread in the future.
- For generating boards, we loop through all possible numbers that could be assigned, so we had one thread check/generate each possible board.

While we could have parallelized the code for checking the rows, columns, and boxes were safe since these were in a loop, we felt that these operations were too simple and would not be necessary to make our algorithm more efficient. We decided it would be wasting resources since threads would be used to do one operation before having to change tasks right away.

Pattern-Based Elimination
We also implemented a rule based elimination algorithm. This algorithm was first implemented sequentially and had two key data structures: an NxN grid representing the sudoku board and a "candidate" board, which was an NxN board of arrays that held all possible candidates for its corresponding cell in the sudoku board. The sequential algorithm worked as follows.

First using the input board, we build the candidate board by looping through all cells, and for each one checking if each number between one and nine could possibly be placed in that cell. This is done by ensuring that no cells in the input board that share a row, column, or box with the cell we are considering

are filled with that number. If a number i is a candidate, we set the i^{th} entry in its array in the candidate board to be 1. Then we enter a phase searching for patterns in the board.

The first pattern is called a “naked single”. This occurs when a cell only has one candidate and therefore that candidate must be in that cell. The second pattern is called a “hidden single” which occurs when some number i is a candidate for only one cell in a given row, column, or box, and therefore must go in that cell. The third pattern is called a “naked pair”, which occurs when a pair of neighboring cells (i.e. cells that share a row, column, or box) both can only contain the same pair of candidates. This means that no other cells in their row, column, or box can contain either of those candidates. The fourth pattern is called a “hidden pair”, which occurs when two numbers both only occur twice in a row, column, or box and both occur in the same pair of cells. This means those two cells cannot have any other candidates besides those two candidates.

We also implemented functions to detect several other patterns such as naked and hidden triples (which extends the idea of the naked and hidden pairs to three cells/candidates) , box reduction (i.e. a candidate only occurs in a single row/column of a box so its neighboring boxes cannot have that candidate in that row/column), and X-Y wings (which involves a pair of cells who share a neighbor whose value can be cased on to determine certainties for their neighbors values). However, these patterns ended up being more computationally expensive to detect than they were helpful, so we did not end up using them in our implementation.

The patterns we did use were able to solve easy and medium boards very efficiently, but stopped making progress in harder boards. To address this, we implemented a method called “forcing chains”, which cased on the value of cells with very few (two or three) candidates and tried to solve the board recursively for each of them.

We successfully parallelized this algorithm using OpenMP. The main sources of parallelism came from being able to search for a single pattern across multiple rows/columns/boxes simultaneously as well as being able to search for all patterns in parallel. Because patterns are pretty rare to find, and usually two different patterns cannot occur in the same cell or pair cells at once, this led to a read-heavy workload, which minimized the amount of synchronization that was necessary. Another source of parallelism that was very significant occurred when using the forcing chains method. Here, we split the board into two possibilities and attempt to solve them recursively. Since we could solve them in parallel, this allowed us to obtain a good amount of speedup compared to the sequential version. We also implemented this algorithm in MPI. In this approach we assigned each node a portion of the board in which to search for patterns. Each node would search for each pattern, update their portion as necessary, and then report back their changes to the root node. The root node would then modify its own board and notify each node of the updates from other relevant nodes. This was not as successful as a parallelization method, most likely due to the communication cost and high number of dependencies. If we had more time, we really would have liked to take a different approach to using message passing to solve sudoku to achieve better speedup.

A limitation of this algorithm that should be noted is that it only guarantees a solution for boards with a unique solution. If a board has multiple solutions, it may not (and likely will not) find any of them. This is something that we would have liked to explore if we had more time.

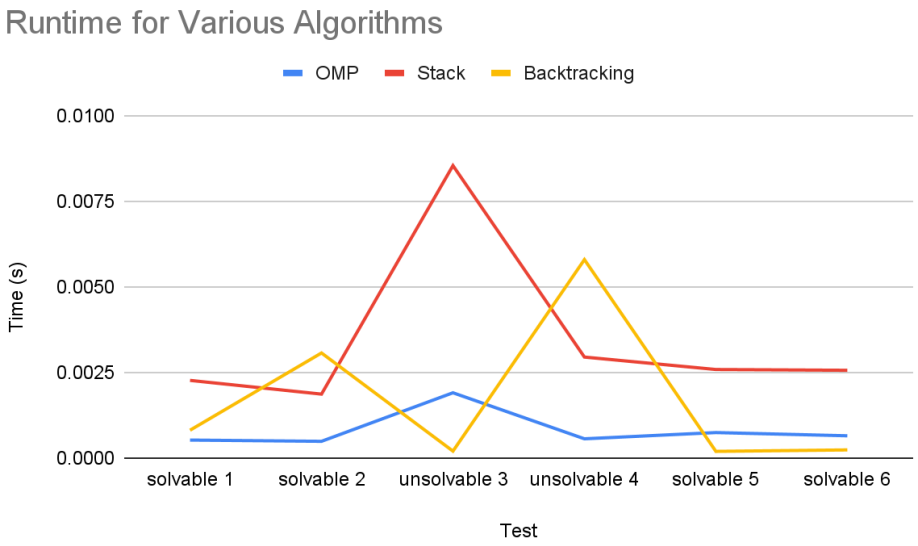
RESULTS:

We measured the performance by using the built-in timer to time how long it took to run our implementation and calculating the speedup compared to the sequential implementation. The inputs to our algorithms were set unsolved 9x9 sudoku boards of varying difficulty. Difficulty was measured by the number and placement of unfilled cells.

Our baseline code for our algorithms using OpenMP were just single-threaded versions of the programs. For the MPI implementation, the baseline was just the code running on a single node/process (i.e. the entire board assigned to that node as its portion).

For smaller workloads, which we defined as “easy” boards or boards with many filled cells upon input, we experienced much less speedup due to parallelism. This is because an “easy” sudoku board can usually be solved in a single pass by a sequential algorithm. Thus in the case of our stacks algorithm, since we push x new boards to the stack if the cell we are looking at has x candidates, if the board is easy many cells will only have one candidate, so we continually push and pull the same board from the stack which is the same as solving it sequentially. In the case of our pattern-based algorithm, easy boards are usually solved in only one pass of looking for simple patterns like naked and hidden singles, so there is not a lot of benefit from parallelism.

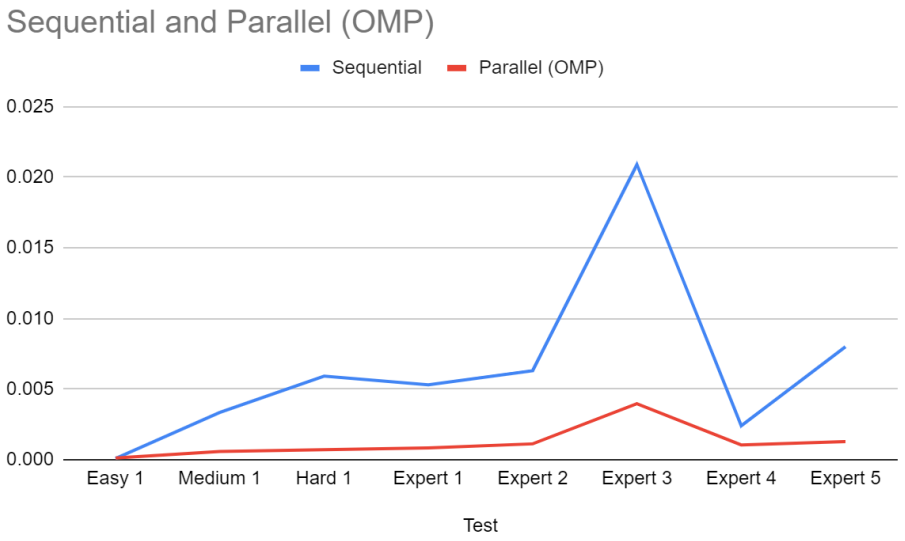
However, for harder boards, each cell has many more candidates, leading us to consider many more boards in our stacks algorithm and many more pattern applications in our pattern-based algorithm. This is where we benefit a lot from parallelism because that work can be done simultaneously by many threads or processes respectively. In other words, with smaller workloads, the synchronization and communication overheads of parallelization outweighs any potential speedup it may yield, but with larger workloads processing all possibilities and searching for all patterns sequentially is exceedingly slower than splitting up the work and synchronizing threads/communicating between processes.



Graph of runtimes for serial and parallel stack algorithms as well as serial backtracking.

	Average Time for Solvable Boards (s)	Speedup	Average Time for Unsolvable Boards (s)	Speedup
Serial	0.001555	1	0.00491	1
OMP Stack	0.000360	4.3205	0.00162	3.0360

For the figures above, our baseline was our serial algorithm implementation on the CPU.



Graphs of runtimes for sequential and parallel pattern-based algorithms.

	Average Time for All Boards	Speedup
Sequential Pattern-Based	0.00653025	1
OMP Parallel Pattern-Based	0.001206875	5.42

A large factor for the limited speedup was the amount of dependencies in the algorithms. A change in one section of the board would affect a majority of the board. Additionally, there is a lot more overhead because of an increase in tasks depending on the board. We can see this from the speedup differences between the solvable and unsolvable boards in the first table above. For unsolvable boards, we have a lot more board possibilities that need to be considered which increases the threads spawned in the recursive portion of the algorithm.

We feel that the CPU was a good choice, since our problem relied a lot on communication between threads, fast memory access, and often had different threads performing very different types of operations. It did not have a lot of data parallelism or opportunity for SIMD execution, which is where a GPU would have come in handy. However, we do think that with more time and experimentation, we could have used a GPU to speed up both our parallel stacks and pattern-based algorithms. This is actually something we really wanted to get to, but just did not have time especially since we were unsure of how much it would have benefited our performance.

REFERENCES:

- For a list and description of various serial algorithms for sudoku solvers: https://www.geeksforgeeks.org/sudoku-backtracking-7/
- For more information on rule based sudoku solving: https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group6Alexander/final/Patrik_Berggren_David_Nilsson.report.pdf
- For more information on Crook's algorithm: http://pi.math.cornell.edu/~mec/Summer2009/meerkamp/Site/Solving_any_Sudoku_II.html#:~:text=Crook's%20method%20of%20preemptive%20sets,important%20tool%20in%20our%20algorithm.
- For more information on parallelization of recursive algorithms and its challenges: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-28.pdf

LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT:

Maddie: 50%

- Generating different level difficulty tests with timing
- Backtracking algorithm and serial stacks algorithm
- OpenMP Crook's algorithm
- Analysis and evaluation of Crook's algorithm compared to other algorithms
- Research and experimentation with MPI implementation
- Reports and poster presentation

Jennifer: 50%

- Generating different level difficulty tests with timing
- Backtracking algorithm and serial stacks algorithm
- Parallel OpenMP stacks algorithm
- Analysis and evaluation of stacks OpenMP algorithm compared to other algorithms
- Research and experimentation of Cuda implementation
- Reports and poster presentation