Project Checkpoint: Sudoku Solver

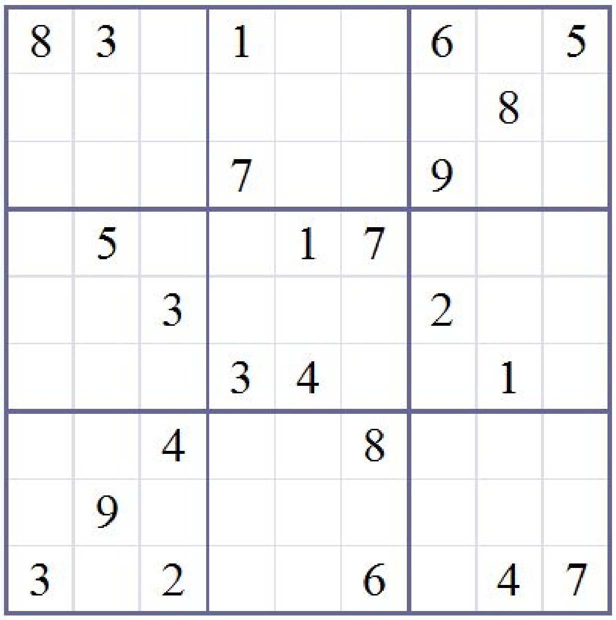
Rhett Crismon and Dylan Ahearn
CSCI4830/ECEN4313
March 25, 2017

Introduction

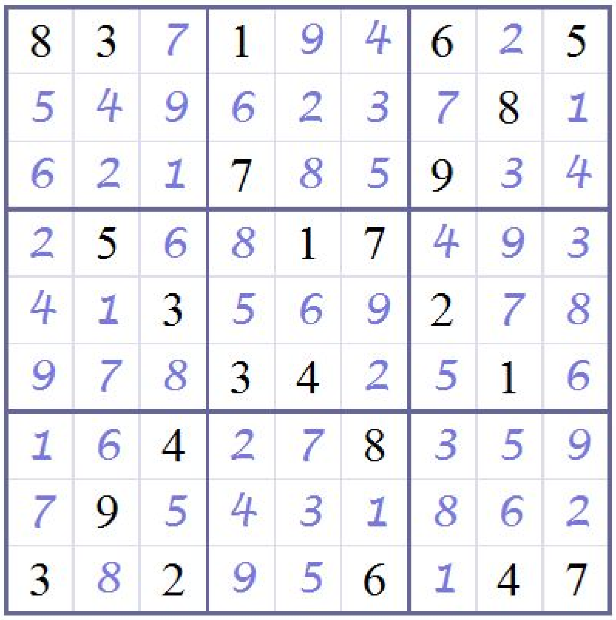
The sudoku puzzle, a partially completed constraint problem, can be solved algorithmically using several well known solutions. As a final project in Concurrent Programming, we will approach these algorithms with the main objective of adding parallelization and concurrency. The secondary objective is to analyze the performance of the modified algorithms with different computational constraints and puzzle difficulties. As a final objective, we will look for a hybrid solution that incorporates features and methods of the other concurrent algorithms as a higher performance solution.

Example Problem

The Sudoku puzzle is a mental game that consists of a nine by nine square grid (81 unique spaces) divided into nine three by three quadrants. Some spaces are filled in at the beginning of the puzzle, building the constraint (figure left) and the unique solution (figure right). The goal of solving the puzzle is to fill in the entire grid with the numbers one through nine, such that within each column, each row, and each three by three quadrant there is exactly one of each number 1-9.



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



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

Proposed Solution

The two algorithms we will parallelize and analyze are the backtracking algorithm, and the simulated annealing algorithm.

- Simulated annealing is an algorithm that starts with a randomly generated solution, in our case a randomly filled in Sudoku board. The algorithm makes a small change to this solution and based on how good this new solution is decides to either accept or reject the new solution. Then it repeats the process. A way of parallelizing this could be to have multiple threads generate and measure the fitness of a new solution. Then a lock would be placed on the current favored solution while it is compared and switched with a new solution.
- Backtracking is an algorithm that selects a cell and places an uninformed guess in it. If there is a conflict in the puzzle then a different number is selected. Then the next cell is filled in and the same check is done. If all possible values cause a conflict then the algorithm goes to the previous cell and trys another value. This process is continued until all the cells are filled in and no conflicts are found. This process could then be parallelized to have each thread run the same algorithm and maintain a master list of solutions that do not work. There would only need to be locks on the object that keeps track of failed solutions.
- We think a hybrid solution does not make much sense as applied to the combination of these two algorithms, where backtracking requires empty spaces and guessing while simulated annealing requires swapping of filled in slots. One algorithm will undo and restrict the other if applied to the same puzzle.

Proposed Quantitative Evaluation

The quantitative methods and tools we will use to evaluate the properties of our project includes timing the processing time and number of operations of the algorithm with varying difficulties of puzzles and varying degrees of concurrent constraints. We will compare the timing of sequential iterations of the algorithm with the concurrent iteration to determine a speedup factor and correlation between varying degrees of concurrent constraints and hybridization. It will be useful to run these tests both on a laptop and on a server to see if there is a difference in speedup depending on the hardware's ability to parallelize. The average value and variance of the results would be graphed on the same plot to show the results in a clear, concrete way.

Implementations So Far

Sudoku Library

- The generator package of the project contains the main sudoku class, with methods and attributes:
    - import a puzzle and its solution
    - Find what the puzzle needs to be completed and fill it in regardless of errors
    - Data getting
        - Number of correct indices, errors, set elements, empty elements
        - If the puzzle is valid, if it is the solution
    - Print the puzzle or solution
    - Swap two elements in the puzzle, randomly or specifically
    - Get parts of the puzzle: column, row, 3x3 box
    - Cell Subclass
        - Locks, values, flags for every element in the sudoku puzzle
- Using a command line program QQWing, we were able to generate 80 test puzzles and solutions
    - 20 at each of the four difficulties: simple, easy, intermediate, expert
    - Saved to text files, parsed, and organized by difficulty and number.
    - Imported by java and stored in the cell class array
    - Website: https://qqwing.com/download.html
- The simulated annealing library and backtracking library both extend the main sudoku class, so that they can use these methods and attributes to implement their solve and helper methods.
- The main Sudoku library was tested with n random swaps by n threads, to ensure that we could concurrently swap on the same puzzle and to get data on how this helped solve the puzzle or not.
    - Using the same puzzle, as the number of random swaps increased, the minimum number of errors would approach a value far above zero.
    - This shows that completely random swapping is not an effective way to solve a sudoku puzzle.

Simulated Annealing

- The non-concurrent version of simulated annealing is slow.
    - The algorithm itself cannot guarantee that it will find a valid solution because it will only do a set number of loops based on the variables. Even when the algorithm is set to run until it finds a correct solution, it takes a unreasonable amount of time. The logical reason behind this outcome is that the simulated

    - annealing settles into a local min (where min is a measurement of errors), but not the absolute min.
    - To account for this issue we picked variables that would cause one attempt at annealing to take a few seconds at most. Then we ran a loop to make anneal attempts until a solution was found. At this point a correct solution was eventually found, usually in under 5 minute.
- We then set out to improve the time of the solution using concurrency.
    - We took the solution we had that ran a loop of annealing attempts. What we did differently was to spawn ten threads that each try annealing.
    - There is a shared variable called flag that marks if a thread has found to solution. When a thread finds a solution it sets the flag to be true and sets the solution to what it found. The other threads check at each loop if the flag is set to true and quit their loops if it is true.
    - The flag variable does not need to be locked because if there is a race condition while writing the flag, the worst that can happen is a read that returns false instead of true. This results in an additional loop of annealing before the program finishes.
    - This resulted in an observed speedup of about 4 times faster, where the program finds a solution, usually in under 2 minutes.
- While developing this solution, we tried a loaded simulated annealing strategy, where instead of annealing on the number of errors in the puzzle we annealed on the number of correct indices in the puzzle in relation to the known solution.
    - This produced a correct solution to the puzzle very quickly, which makes sense, since there are significantly fewer local maximums in annealing over the number of correct indices, only the absolute maximum of 81. In other words, when the algorithm accepts a swap that makes the number of correct indices go up, it is surely moving forward towards the correct solution and not an incorrect local maximum.

Backtracking

- The non-concurrent implementation of the backtracking algorithm works well.
    - The basic idea is to find the next open spot and try all 9 numbers, keeping the first of the 9 that does not create an error.
    - Then recursively move on to try the next open spot, and if eventually there is an error you backtrack to a previous cell to try another value and move forward again.
    - Eventually, there will be no more open spots and no more numbers to try. At this point, you will have the correct solution.

Results

Simulated Annealing

- Sequential Avg Completion time: 2.5 min
- Concurrent Avg Completion time: 1 min
- Best parameters: a=.99, t=1,k=1,min_t=.0001, loops per temp=100
- No theoretical guaranteed solution
- Current number of threads= 10

Backtracking

- Sequential Avg Completion: 1.2 seconds
- Guaranteed to find a solution if one exists
- Concurrent results to come

Plan Going Forward

Data Collection

- With a working concurrent implementation of simulated annealing, it is time to start taking data, comparing the sequential solution's performance with the concurrent solution's performance.
- The variables to consider:
    - Difficulty of puzzle
    - Processor speed
    - Number of processor cores
    - Annealing parameters
    - Number of threads
- Data to be collected:
    - Average and Variance of number of annealing attempts to get solution
    - Average and Variance of and Variance of time taken to get solution
    - In failed attempts
        - Average and Variance of number of correct vertices at exit
        - Average and Variance of number of errors at exit
- Data for each of these items will be associated with the chosen variables and compared directly against the sequential performance.

Improvements

- In simulated annealing, we want to try to doing the random swaps in a different way. As of right now, the algorithm take two random cells to swap. We believe that changing the swap so that a cell in the puzzle must swap with a neighbor instead of at random could result in the simulated annealing converging to a solution better.
- We would like to take the backtracking algorithm and make it concurrent to see if we can speed up the back tracking.
    - Spawning 9 threads at each open place to branch off and try to solve the puzzle. Eight will fail at each, and there will be one path to the solution made by threads that never actually backtracked.
    - The problem here is that you will spawn 9^x threads, which quickly becomes unreasonable for the finite computer to handle.
    - If you always limit the total number of working threads to some value, the branching threads should be able to still divide and conquer to an extent, but eventually wait for other threads to finish before being able to move on.