Parallel Sudoku Solver

High-Performance-Computing (CS-301) Report

Luv Patel (201501459)
Harshal Khodifad (201501461)

• Introduction

Sudoku is a logic-based number-placement puzzle game where the player's goal is to complete a NxN table such that each row, column and box contains every number in the set {1, . . . , n} exactly once. In this project, we attempted to generate a highly parallelized Sudoku solver using the OPENMP. The standard Sudoku grid is 9x9, however without changing the nature of the rules, we can extend the definition of the Sudoku game to any n^2x n^2grid, n=3 being the standard game.

INPUT

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

OUTPUT (UNIQUE ONLY)

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

• Motivation:

Sudoku is today a popular game throughout the world and it appears in multiple media, including websites, newspapers and books. As a result, it is of interest to find effective Sudoku solving and generating algorithms. Numerous serial Sudoku solver implementations exist and are readily available. However, due to its recency, there is relatively little work on parallel implementations. Sudoku is also, a NP-Complete problem which means that it is one of a set of computational difficult problems. Parallelizing a serial Sudoku solver can improve its speed and increase the viability of solving larger size Sudoku. The Sudoku solver algorithm also conveys some of the basic trade-offs of parallel software development such as dependencies and work-sharing which are interesting to inspect and study.

ALGORITHM:

1. *Brute Force Algorithm*

Solving sudoku is proven to be an NP - Complete problem.There are no serial algorithms that can solve sudoku in polynomial time.The number of valid Sudoku solution grids for the standard 9×9 grid was calculated by Bertram Felgenhauer and Frazer Jarvis in 2005 to be **6,670,903,752,021,072,936,960.** Trying to populate all these grids is itself a difficult problem because of the huge number, Assuming each solution takes 1 micro second to be found, then with a simple calculation we can determine that it takes **211,532,970,320.3 years** to find all possible solutions. If that number was small, say 1000, it would be very easy to write a Sudoku solver application that can solve a given puzzle in short amount of time. The program would simply enumerate all the 1000 puzzles and compares the given puzzle with the enumerated ones. Unfortunately, this is not the case since the actual number of possible valid grids is extremely large so it's not possible to enumerate all the possible solutions. This large number also directly eliminates the possibility of solving the puzzle with brute force technique in a reasonable amount of time. Therefore, a method for solving the puzzle quickly will be derived that takes advantage of some "logical" properties of Sudoku to reduce the search space and optimize the running time of the algorithm.

2. *DFS Bactracking Algorithm*

The serial algorithm we implemented is the **DFS BACKTRACKING ALGORITHM.** The basic algorithm is to ***search*** for a solution: to systematically try all possibilities until we hit one that works.
This is a ***recursive*** search, and we call it a ***depth-first*** search because at first we assign a value to an empty cell then we (recursively) consider all possibilities when value is assigned to remaining empty cells before we consider a different value for the current empty cell. And finally we undo the change we did to current empty cell when we hit a dead end. This is known as ***backtracking search***.(i.e we go back to the recently changed value and increase the value.)

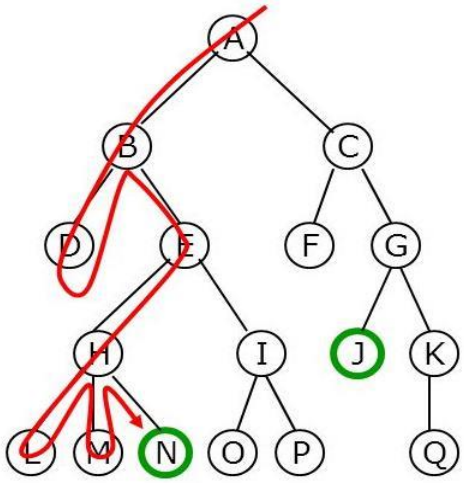
In this algorithm, we keep on assigning numbers one by one to empty cells. Before assigning a number, we check whether it is safe to assign. We basically check that the same number is not present in current row, current column and current 3X3 subgrid. After checking for safety, we assign the number, and recursively check whether this assignment leads to a solution or not (i.e. we assign a number to the next empty cell and continue forward till no number is permissible at that empty cell). If the assignment doesn't lead to a solution we backtrack to the recently filled empty cell , then we try next number for the empty cell. And if none of number (1 to 9) lead to solution, we would say that the solution does not exist.

Pseudocode:

1. Find an empty cell.
2. If there is none, return true.
3. For digits from 1 to 9
   a) If there is no conflict for digit at row,col
      assign digit to row,col and try filling in the rest of grid.
   b) If all empty cells are filled, return true.
   c) Else, backtrack and fill the recently filled empty cell with next number possible.

- If all digits have been tried and nothing worked, return false

- **Example :**



Now we will try to solve the above given input Sudoku using our algorithm. We start at the first free square on the board, row 0 (the top row) column 1 (the second column from the left). This is the first empty square on the board. There are a number of symbols we can use for this square, let's assume: 1, 7 and 9; all other symbols would break the rules of Sudoku. Therefore there are three branches to investigate – see figure 1. The depth first algorithm attempts to solve the problem by traversing down the left hand side of the decision tree, so we investigate the first branch by placing a 1 in square 0, 1.

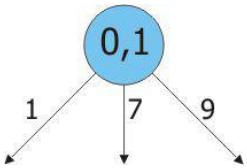


Figure 1: First branch of the decision tree.

Once we've guessed at a symbol for the first square (0,1) we must attempt to find a symbol to go in square 0,2 (the second free square as we traverse the board in a raster-scan style). At this point we have two options, placing either a 7 or a 9 in the square. Therefore the tree branches into two here; we take the left hand branch by placing a 7 in square 0,2 – see figure 2.

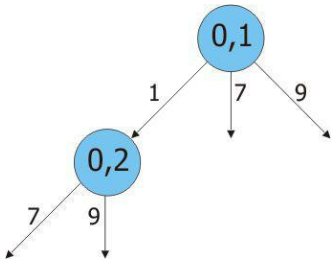


Figure 2: Second branch of the decision tree

Figure 3 shows the next step through the tree. We have two choices for square 0,3: 5 and 9. We select 5 and continue.

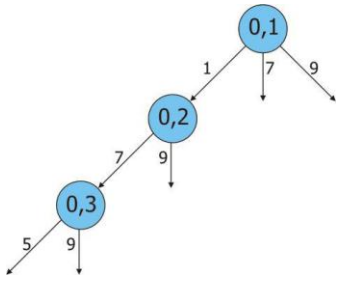


Figure 3: Third branch of the decision tree

At some point one of two things will happen, either…

- We have filled every square on the board and therefore solved the puzzle

- We reach a square where we can't place any symbol. In this case we backtrack by moving back to the previous square and trying the next possible symbol.

Figure 4 shows us backtracking. The algorithm has investigated every possible branch with 5 in the 3rd square and none lead to a solution. Therefore we try again with a 9 in the 3rd square.

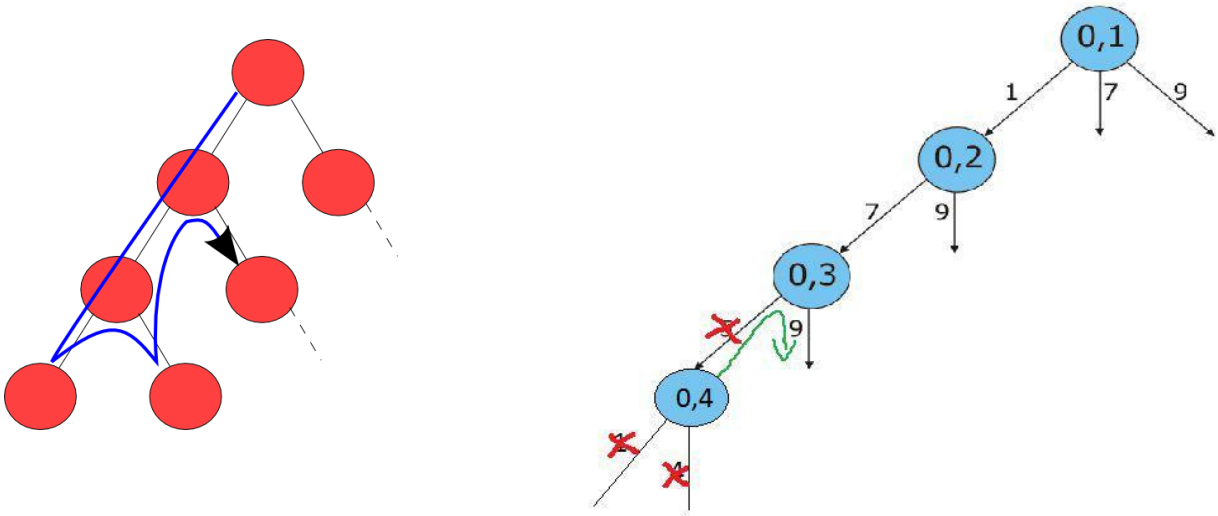


Figure 4: Backtracking through the decision tree

Similarly, we will continue to solve the Sudoku according to our algorithm.

Computational Complexity of DFS Backtracking Algorithm

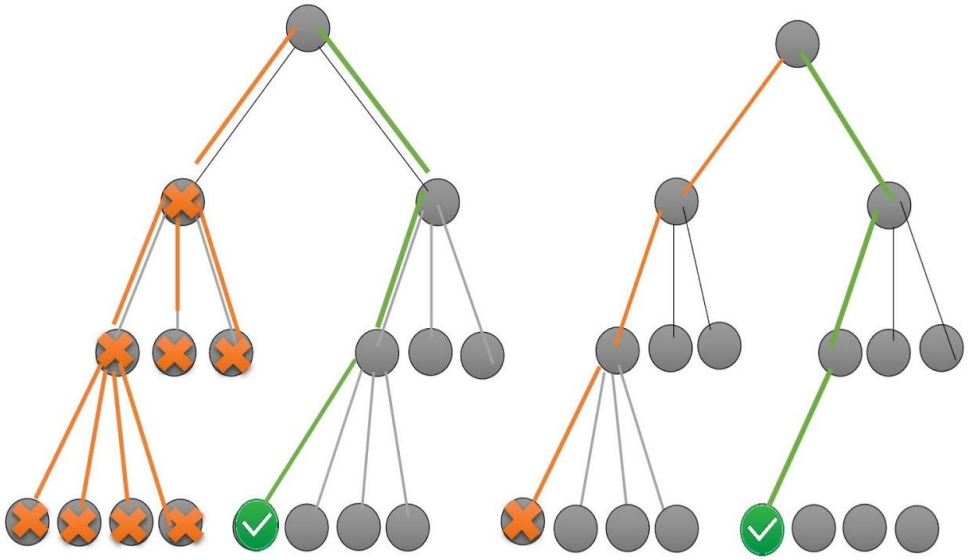
The complexity of DFS Backtracking algorithm is O(n ^ m) where n is the number of possibilities for each square (i.e., 9 in classic Sudoku) and m is the number of spaces that are blank.

This can be seen by working backwards from only a single blank. If there is only one blank, then you have n possibilities that you must work through in the worst case. If there are two blanks, then you must work through n possibilities for the first blank and n possibilities for the second blank for each of the possibilities for the first blank. If there are three blanks, then you must work through n possibilities for the first blank. Each of those possibilities will yield a puzzle with two blanks that has n^2 possibilities.This algorithm performs a depth-first search through the possible solutions. Each level of the graph represents the choices for a single square. The depth of the graph is the number of squares that need to be filled. With a branching factor of n and a depth of m, finding a solution in the graph has a worst-case performance of O(n ^ m).

SCOPE OF PARALLELIZATION:

By parallelizing this code we can reduce the computational complexity greatly as explained below:

SERIAL IMPLEMENTATION PARALLEL IMPLEMENTATION



Here in the Left diagram which executes serially ,the solution is checked for values of first branch, if there is a contradiction ,we check for the second branch and so finally the solution is found in the first part. Whereas in the Right Diagram which executes parallely both branches are executed in parallel which in turn reduces the computation complexity as the solution is found in the second branch at the same instant when we get a contradiction in first branch.Thus when one of the threads finds the solution, it can tell the other threads to stop.

However, the flipside may also be possible. **If the solution is located in the first branch or the problem size is small**,the parallel algorithm may be slower due to the thread management overhead and executing multiple branches in parallel. However, the runtimes of these cases are very low in the first place,so the overhead becomes a small penalty when amortized .This is what we have observed in the serial vs parallel graph when the input size of problem is small.

- ***Please note that problem size is dependent on the depth of the recursion while solving and thus is not dependent on input and thus we cannot directly say that 16x16 input matrix problem size is larger than 9x9 input matrix hard problem size.***

Parallelization Strategy of DFS backtracking algorithm:

The proposed algorithm marks the cells which are fixed(cells whose value is already in input and thus cannot be changed) and the cells in which we have to enter the proper values as not-fixed (Variable).This is implemented in the **read_sudoku() function**.

Our algorithm follows a depth first search approach with backtracking on non-fixed cells. However, backtracking on many core is not an easy task since each core is processing a path independent of the other cores so it's hard to determine what was the previous step and backtrack to it. To solve that problem we make copies of matrix and assign each copy of the matrix to individual core to work on (This is implemented in **create_list() and pop_node() functions**)and then implement our own explicit functions to keep track of previous state of the matrix.(This is implemented in **go_forward() and go_backward() functions**in our code).

We also need to determine the granularity of a task or a work item, that is, the smallest unit of work done by any core because each core will be handling a work item at any point during the running time of the algorithm. Defining the granularity clearly shows whether each work item is dependent on other work items or that it is totally independent. In this case, the work item is not dependant of previous inputs and does not require any kind of inputs from other cores in order to be processed.

The algorithm starts by splitting the work items equally among available cores. Given N cores and n^2 work items, where N is the number of core and n is the width of the square grid, and to distribute the loads equally across the cores, each core is assigned n^2/N work items. Each core will determine the set of possible values for each of its cells and store these possible value. The process of assigning each copy of the matrix to individual core to work upon will also be a **CRITICAL SECTION** because at a time only one core can be assigned which core will work on finding the solution to a particular initial matrix that has been assigned to it. Once the work items are assigned, the cores are ready to start processing their work items.

When a core tries to set a value for a cell but does not find a value because the cell has no more possible values, meaning that the current values in all the cells in the grid yielded an invalid Sudoku puzzle. The core has to backtrack to a previous state and take another branch. Here, the state of the puzzle is defined as a stable instance of it, that is, a puzzle that could be solved and is not yet determined as unsolvable and is further processed to obtain a solution.

Terminating conditions:

This is implemented by the *while (level > 0 && i < SIZE && found == 0)* condition in the parallel section of the code.

1)If a valid solution is found then all the other cores have to be informed to stop processing and return to the main function to inform the user that a solution is found.

2)If no solution is found till the end of the algorithm exhausts all possible combination the algorithm terminates saying that no solution is found and the problem is unsolvable for the given input and requires more input cells to solve that problem.

Parallel overhead time

OpenMP version on 1 Thread core vs serial without openMP

(Overhead = Parallel Time on 1 thread -Serial Time)

Problem	Serial Time	Parallel Time	Overhead
9 x 9 (Easy 1)	0.000371	0.000334	-0.000037
9 x 9 (Easy 2)	0.000322	0.000275	-0.000047
9 x 9 (Easy 3)	0.06813	0.055042	-0.013088
9 x 9 (Medium)	0.004323	0.004313	-0.00001
9 x 9 (Hard)	21.123579	21.327317	0.203738
16 x 16	1.266641	1.270904	0.004263
25 x 25	4.994336	5.073587	0.079251
64 x 64	42.934602	43.173908	0.239306
Impossible	0.000118	0.000097	-0.000021

Here Impossible Problem implies that the problem is not solvable and our code detected that the Sudoku cannot be solved. The serial and parallel time shown implies that the Sudoku gave a contradiction at the given time.

Analysis:

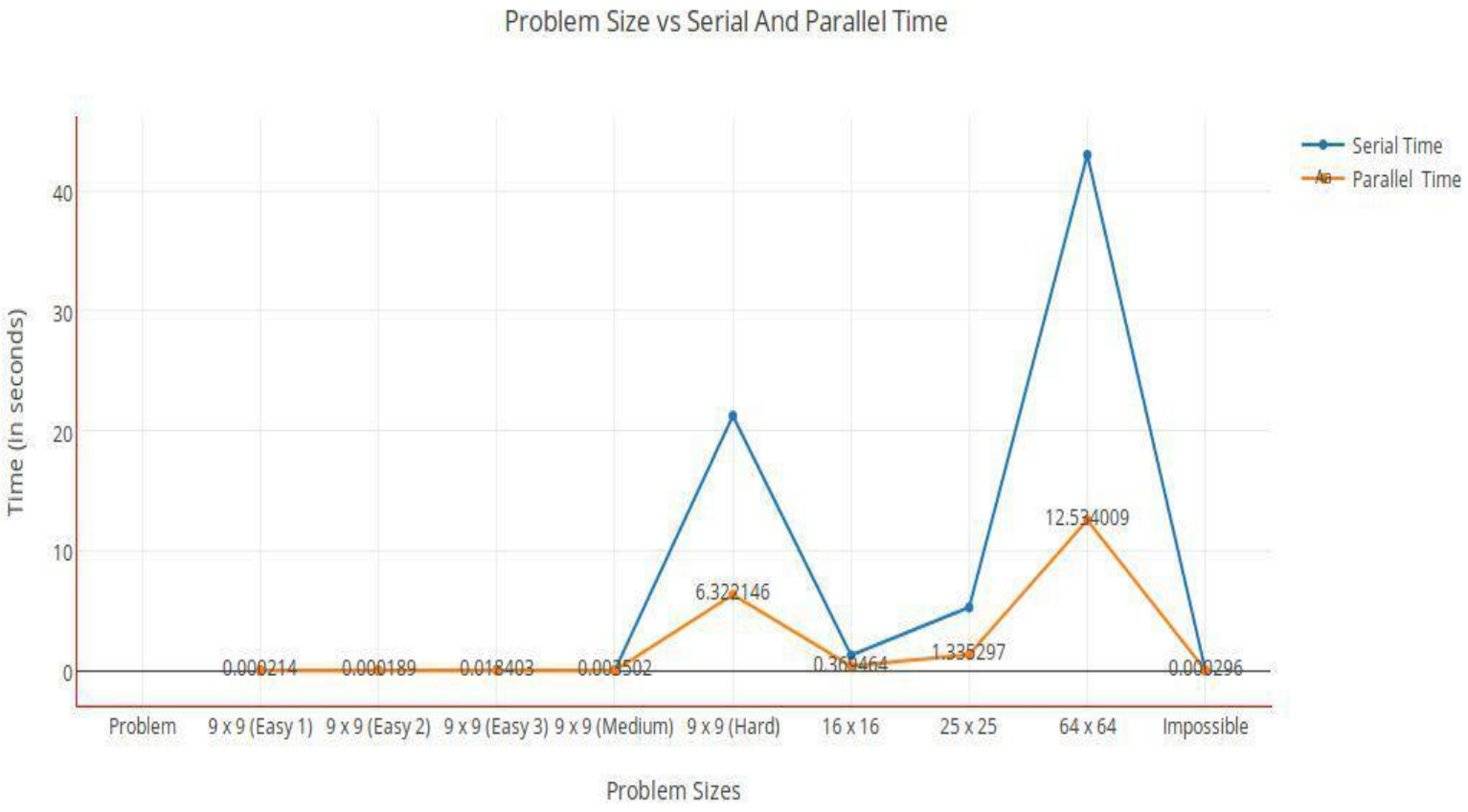
1) Parallel overhead for small cases is ignored because small problems are solved too quickly (serially) which gives inaccurate speedup (Negative overhead).

2) Software overhead is also imposed by parallel languages, libraries, operating system, etc.As problem size increased overhead increases. This is because when 1 thread is spawned for calculating overhead it requires some processing time which would not be required when there will be serial code execution.

Total overhead is total time spend by all processors combined in non-useful work.Total overhead is a function of both problem size Ts and the number of processing elements p.The total overhead function is generally an increasing function of problem size if p=constant. This is because every program must Contain some serial component. If this serial component of the program takes time t , then during this serial time all the other processing elements must be idle. Thus when problem size increases t increases and serial thus the time idle time increases and the overhead imposed to spawn thread increases.

Problem Size vs Serial And Parallel Time (for default 4 cores):

Problem	Serial Time	Parallel Time	Speedup
9 x 9 (Easy 1)	0.000158	0.000214	0.73694
9 x 9 (Easy 2)	0.000158	0.000189	0.83819
9 x 9 (Easy 3)	0.041165	0.018403	2.236808
9 x 9 (Medium)	0.007428	0.003502	2.120929
9 x 9 (Hard)	21.241433	6.322146	3.359846
16 x 16	1.271536	0.369464	3.441569
25 x 25	5.265887	1.335297	3.943607
64 x 64	42.990588	12.534009	3.429915
Impossible	0.000245	0.000296	0.828515

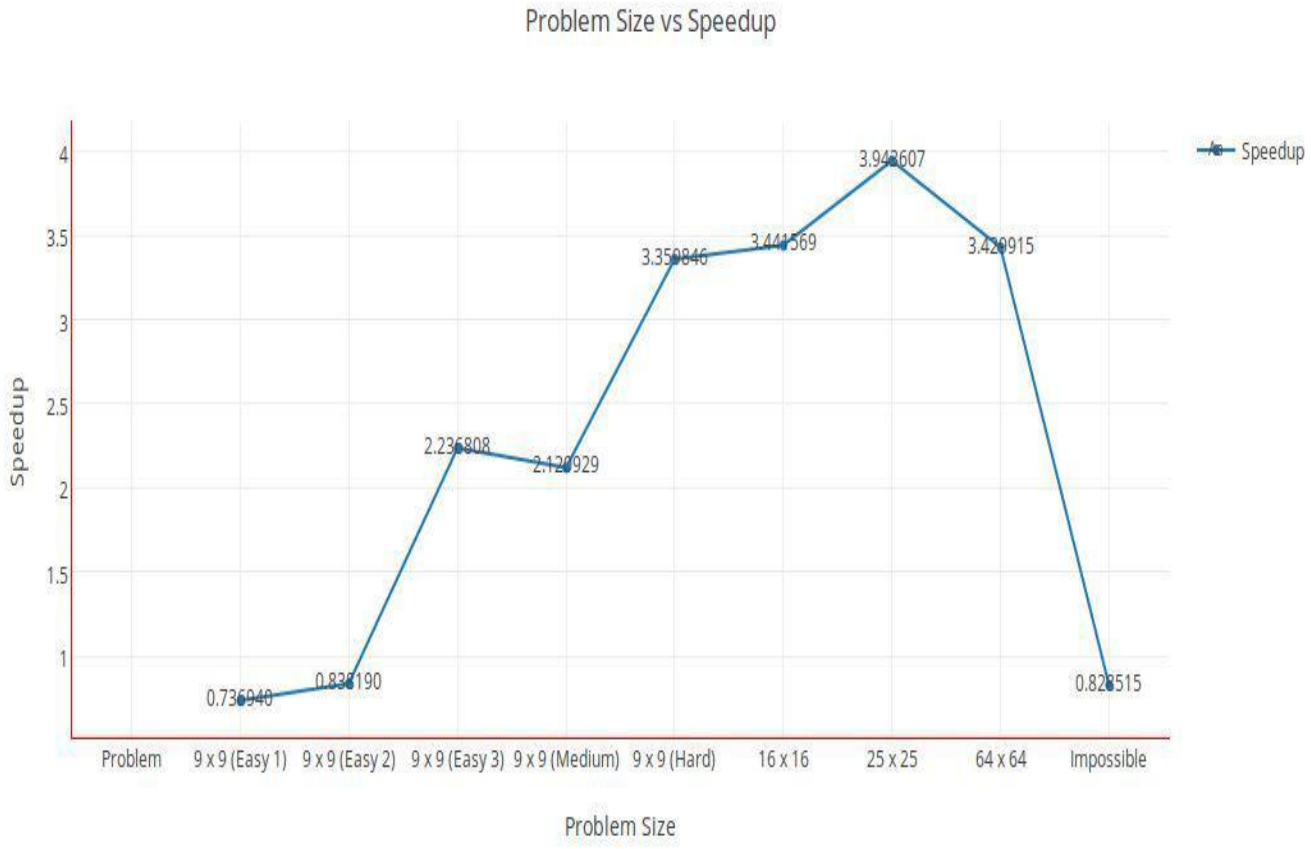


Problem Size vs Serial And Parallel Time

(for default 4 cores) : (Above)

Problem Size vs Speedup

(for default 4 cores) : (below)



➔ *Easy problems are solved too quickly (serially) which gives inaccurate speedup.*

Explanation for the above 2 graphs Analysis 1:

1)As we can see for smaller inputs we have serial and parallel code executing and giving result in same time , infact serial code is better when size is very small.Such behaviour is expected because the relative overhead is high as compared to problem size.

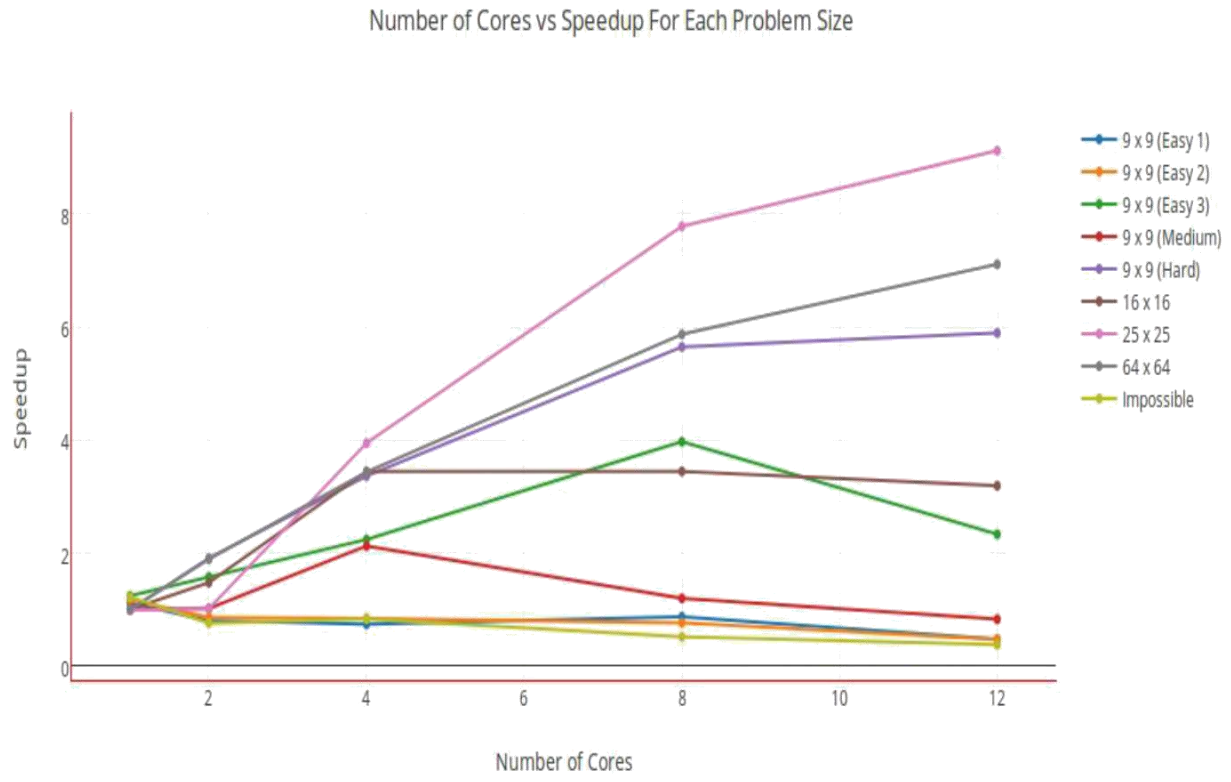
2)When problem size becomes larger and larger that's when speedup increases in a multithreaded code as the work gets shared among the threads and computation time decreases.

3)Thus we say that parallel computing is very useful in cases where the size of problem is very large.As serial code would take a large time to solve that problem as evident to comparision between ts and tp.

4) The parallel code speedup depends on the number of cores which we have in the hardware machine .In our case it is approaching 3.94 (25x25 case) as we have set the default number of threads to 4.This shows that we are able to utilize the cores properly and able to get a respectable speedup form our code.

No. of cores Vs Speed-Up curve for a couple of Problem Sizes:

Number of cores	9 x 9 (Easy 1)	9 x 9 (Easy 2)	9 x 9 (Easy 3)	9 x 9 (Medium)	9 x 9 (Hard)	16 x 16	25 x 25	64 x 64	Impossible
1	1.110378	1.169444	1.237778	1.002158	0.990447	0.996645	0.98438	0.994457	1.217213
2	0.803519	0.85497	1.573381	1.01373	1.904711	1.472901	1.008544	1.888478	0.756374
4	0.73694	0.83819	2.236808	2.120929	3.359846	3.441569	3.943607	3.429915	0.828515
8	0.86847	0.761295	3.968814	1.193693	5.645896	3.442577	7.781546	5.869634	0.513624
12	0.469471	0.467897	2.329027	0.825685	5.893337	3.188149	9.121506	7.110071	0.375324



Explanation for the above 2 graphs Analysis 2:

1) As problem size increases the speedup generally increases for a fixed number of core.This is where the domain of HPC lies as discussed in Analysis 1 .As discussed earlier that problem size is not the size of matrix but it is dependent on the depth of the recursion tree while solving thus we can see that 25x25 input matrix requires more computation than 64x64 input matrix.

2) For a particular problem size the speedup increases when there is increase in the number of cores.But the Efficiency (Speedup / Number of Cores)Decreases this behaviour is expected as Linear speedup is not achievable, in general, because of contention for shared resources, the time required to communicate between processors and between processes, and the inability to structure the software so that an arbitrary number of processors can be kept usefully busy.

3) Figures above Illustrate that the speedup tends to saturate and efficiency drops as a consequence of Amdahl's law as if linear speedup had occurred we would have got the speedup of 12 but we in the best possible case get a speedup of 9 in 12 cores. Furthermore, a larger instance of the same problem yields

Higher speedup and efficiency for the same number of processing elements, although both speedup and efficiency continue to drop with increasing p.

Performance Metrics 1:

Efficiency: Efficiency is a measure of the fraction of time for which a processing element is usefully employed. Mathematically, it is defined as Efficiency = Speedup/Processors.

- **Problem vs Efficiency for default 4 cores:**

Problem	Efficiency
9 x 9 (Easy 1)	0.184235
9 x 9 (Easy 2)	0.2095475
9 x 9 (Easy 3)	0.559202
9 x 9 (Medium)	0.53023225
9 x 9 (Hard)	0.8399615
16 x 16	0.86039225
25 x 25	0.98590175
64 x 64	0.85747875
Impossible	0.20712875

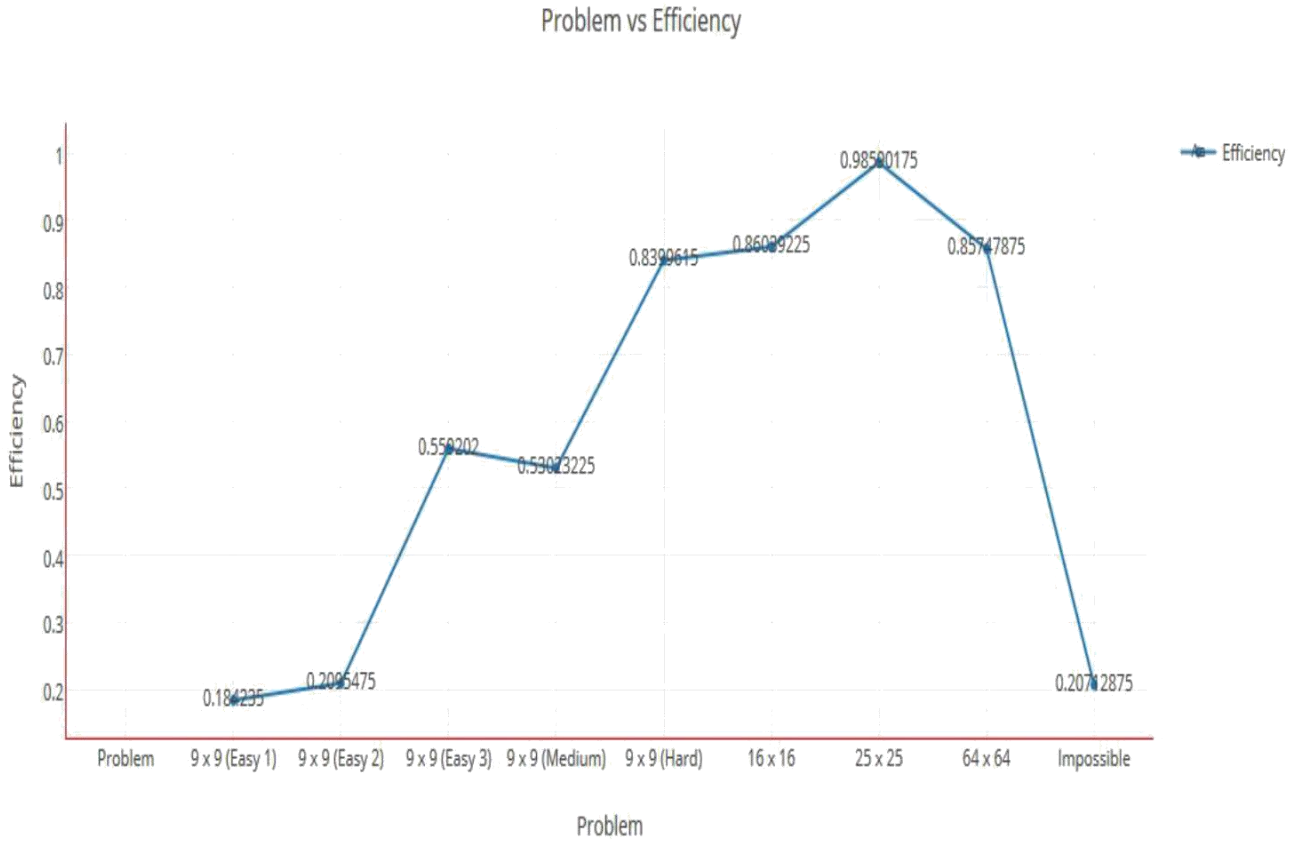


Figure for default 4 cores : The above graph helps us to know if our code is scalable.

Analysis for the above graph:

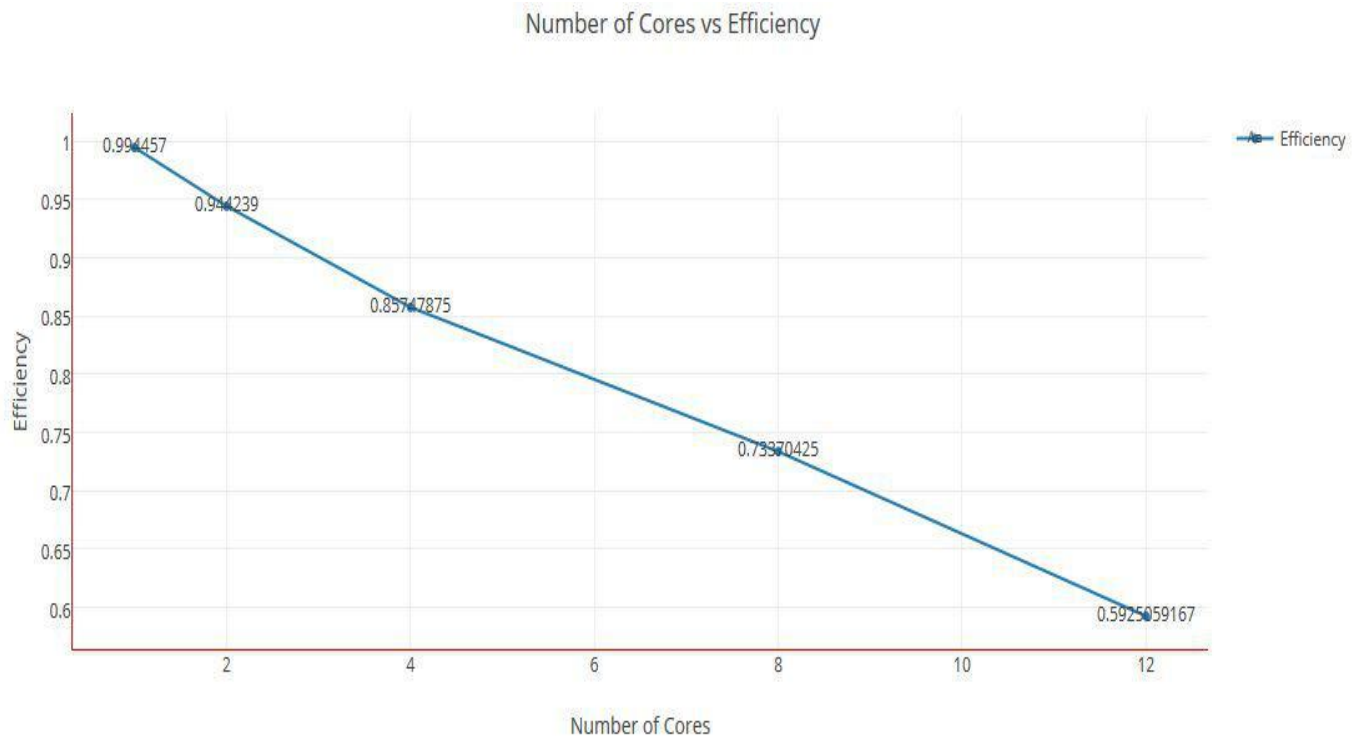
1) The efficiency increases if the problem size is increased keeping the number of processing elements constant.

2) For such systems, we can simultaneously increase the problem size and number of processors to keep efficiency constant. We call such systems scalable parallel systems thus our code is scalable.

Performance Metrics 2 :

Number of cores vs Efficiency for a particular Problem Size(in this case 64 x 64):

Number of cores	Efficiency
1	0.994457
2	0.944239
4	0.85747875
8	0.73370425
12	0.5925059167



Analysis of the above graph:

1) For a given problem size (i.e., Serial Time remains constant), as we increase the number of processing elements, Total Overhead increases as it is an increasing function of p as discussed earlier in overhead analysis.

2) The Overall efficiency of the parallel program goes down. This is the case for all (most) parallel programs.

Future work

We limited our implementation to solve puzzles sequentially, with subsequent puzzles being solved after taking measurements. We can improve the performance by attempting to solve multiple puzzles in parallel at the same time. This would reduce thread downtime, as unused threads could begin attempting to solve subsequent puzzles, amortizing the overall runtime. Beyond our implementations, we did not attempt to use GPUs, OpenCL and FPGAs to accelerate our solvers. We could attempt to try solving many puzzles at the same time with GPUs in the future. We believe this may be very difficult, as the overhead of GPUs is only paid for when we are working on hundreds of threads at the same time.

Conclusion

In this project, we proposed a parallel algorithm for solving an NxN Sudoku grid. The algorithm is designed to maximize the utilization of all cores available on the system. We have shown the efficiency of the algorithm and that it utilized all the available cores on the system. Future work for this algorithm will be testing it for many core architecture and verifying if it will scale on many core (GPU's 100-200 cores) as it did on multicore (1-12 cores).We can conclude that we get speedup when our code runs in parallel on different cores i.e. We can parallely solve a Sudoku with a considerable amount of speedup. Also by calculating the efficiency for different number of processors and different problem sizes, we can say that our code is scalable. The justification of speedup is already given in the scope of parallelization described above.

References

For Concept: https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

Research Papers:
http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand13/Group1Vahid/report/henrik-viksten.viktor-mattsson-kex.pdf