Parallelization in Sudoku Solving

Bella Liu, Derek Duenas

```
Input board:
0 0 0 | 0 0 0 | 0 1 0
4 0 0 | 0 0 0 | 0 0 0
0 2 0 | 0 0 0 | 0 0 0
------+-------+------
0 0 0 | 0 5 0 | 4 0 7
0 0 8 | 0 0 0 | 3 0 0
0 0 1 | 0 9 0 | 0 0 0
------+-------+------
3 0 0 | 4 0 0 | 2 0 0
0 5 0 | 1 0 0 | 0 0 0
0 0 0 | 8 0 6 | 0 0 0
```

```
Solution board:
6 9 3 | 7 8 4 | 5 1 2
4 8 7 | 5 1 2 | 9 3 6
1 2 5 | 9 6 3 | 8 7 4
------+-------+------
9 3 2 | 6 5 1 | 4 8 7
5 6 8 | 2 4 7 | 3 9 1
7 4 1 | 3 9 8 | 6 2 5
------+-------+------
3 1 9 | 4 7 5 | 2 6 8
8 5 6 | 1 2 9 | 7 4 3
2 7 4 | 8 3 6 | 1 5 9
```

Summary

The project implements a Sudoku solver using a backtracking algorithm, and optimizes it with parallelization using OpenMP and MPI.

- Link to GitHub repository: https://github.com/bella713/15418-final-project
- Link to project website: https://bella713.github.io/15418-final-project/

Introduction

Sudoku is a classical logic-based combinatorial puzzle game originated from Japan. In classic Sudoku, the player is given a partially filled 9 x 9 grid. The objective is to fill the grid with digits such that each row, column, and 3 x 3 sub-grid contain all the digits from 1-9. In this project, different N * N of Sudoku boards will be tested, such as 16 x 16 and 25 x 25, which will further assess the performance of the project.

Sudoku-solving algorithms include testing for different possible answers for each of the empty cells. The inherent nature of this problem allows for parallelization to optimize it.

Background

The project implementes Sudoku solver in three ways: serial backtracking, openMP parallelized backtracking, and MPI backtracking.

The naive solution to solving sudoku is to use brute-force algorithm, which generates all possible configurations by filling all the empty cells, and then tries every

configuration one by one until the correct configuration is found. The brute-force algorithm is slow and can take a significant amount of time for puzzles with many empty cells. The time complexity is $O(k^{(N*N)})$, where k is the size of board (9, 16, 25, etc)

The project uses backtracking algorithm for the sequential algorithm, which improves on the brute-force algorithm. The algorithm is as follows:

- Before assigning a number, check whether it is safe to assign. Check that the same number is not present in the current row, current column and current 3X3 subgrid.
- After checking for safety, assign the number, and recursively check whether this assignment leads to a solution or not.
- If the assignment doesn't lead to a solution, then try the next number for the current empty cell.
- And if none of the number (1 to 9) leads to a solution, return false and print no solution exists.

The sequential algorithm serves as a benchmark for measuring the performance of parallel algorithms.

Approach

The project is programmed in C++ and uses OpenMP and MPI library for parallelization.

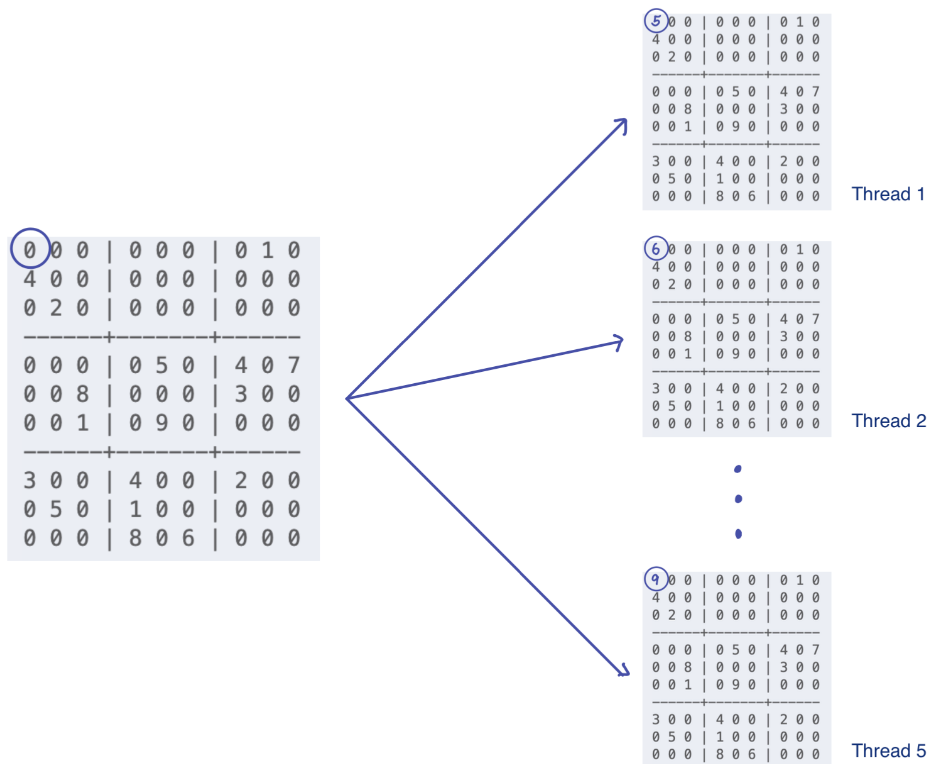
1) OpenMP

For OpenMP implementation, parallelization was built upon the sequential backtracking algorithm.

To adapt the code to OpenMP programming, the recursion section of the code was modified. In the sequential code, for each empty cell in the grid, we set the cell in the original grid to a valid number between 1-N, and then recurse on this new original grid. If a solution cannot be found, the cell is set back to 0, and then another number will be tried. However, in the OpenMP implementation, when there is am empty cell in the grid, the grid is duplicated, and the copy grid will have the empty cell updated with a number in 1-N and be recursed on. If no solution is found, we delete and free the copy grid to free up memory. This prevents different threads trying to modify the same board at the same time, which would cause racing and memory inconsistency.

The program initially ran into problems with exceeding memory due to the large number of duplicates of boards. The problem was solved by deleting the duplicated board immediately after using it.

The program parallel the code by invoking *#pragma omp task firstprivate(grid, row, col, num, level)*, which parallelizes the tests for all valid numbers in 1-N for an empty cell. To avoid racing and memory inconsistencies, *#pragma omp taskwait* is placed before the function returns that no solution can be found.



For instance, for the empty cell circled above, all possible valid inputs are 5, 6, 7, 8, 9. Each of these possibilities is sent to a separate thread, where a duplicate board is created and then recursed upon.

In the main function, #pragma omp parallel shared(grid) num_threads(#thread) is placed before invoking the solveSoduku function, which indicates the number of threads and that the input sudoku board is shared by all threads.

2) MPI

For the MPI implementation, we considered two different implementations. A work queue implementation and a constraint satisfaction implementation. We did not use a work queue implementation because in MPI it is very expensive to send tasks and

their results back and forth. Also, it is very hard to determine when a process needs to create new tasks from a sudoku puzzle or solve the puzzle completely.

We decided to add constraint satisfaction to the sequential backtracking algorithm. Before we do a step of brute force search we check the board to see if any squares can only be a single number. If a square has only 1 option then we fill it in and repeat until no progress is made. If no progress is made then we do one step of the brute force search and repeat. This constraint checking however is very inefficient when a board is not very filled so we added a threshold on the number of squares in the board that must be filled before we do constraint checking. This limits the amount of speedup we can get since the parallelism only comes into effect in later parts of the problem.

Having too low of a threshold value will cause the constraint checking to start when the board is too empty and slow down the program. Having too high of a threshold value will cause the constraint checking to start too late and not give the maximum speedup from parallelism.

One major change that we did for MPI was to change the grid from a 2d N by N array to a 1d array of length N*N. This made sending and receiving parts of the grid easier.

We divided the grid into equal sequential sections based on the process id for splitting up the work when doing constraint checking. For example, if there are two processes for a 9x9 board then process 0 will get the indices 0 to 40, and process 1 gets indices 1 to 80.

Results

To thoroughly test the results of both OpenMP and MPI implementations, we used sudoku puzzles of different sizes and different difficulty levels. The tests files are in forms of .txt files. The compiled code takes in a .txt file, and parse the input puzzle into a 2D array of integers for further processing.

For the sizes, we used board sizes of 9x9, 16x16, and 25x25. The complexity increases exponentially with the increase of board size.

For boards with the same size, the difficulty level can vary drastically depending on the number of empty cells. If there are more empty cells, the difficulty increases greatly and can increase the computation time significantly. We labeled the test cases of increasing difficulty with increasing number, where the difficulty is determined by the

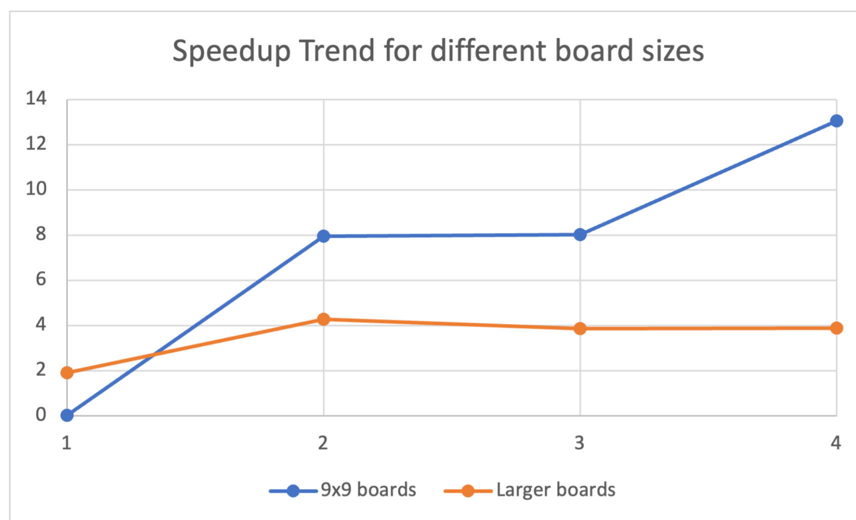
serial solution computation time. For instance, 9x9 board 1 is the easiest in 9x9 test boards, while 9x9 board 4 is the hardest in 9x9 test boards.

All tests were run on GHC machines.

1) OpenMP

Table 1 | Computation time & Speedup for different board sizes/difficulty levels

	Serial solution	OpenMP optimal solution	Speedup
9x9 board 1	0.000252532	0.0149266	0.01691
9x9 board 2	0.989897	0.124703	7.9380
9x9 board 3	10.3193	1.286	8.0243
9x9 board 4	27.1888	2.08194	13.0593
16x16 board 1	0.927975	0.490307	1.8926
16x16 board 2	13.173	3.0864	4.2680
25x25 board 1	66.2631	17.2111	3.8500
25x25 board 2	74.2406	19.1571	3.8753



From Table 1 and the figure we can see that OpenMP implementation achieves better speedup for a board with higher difficulty level. If we examine the result for all 9x9

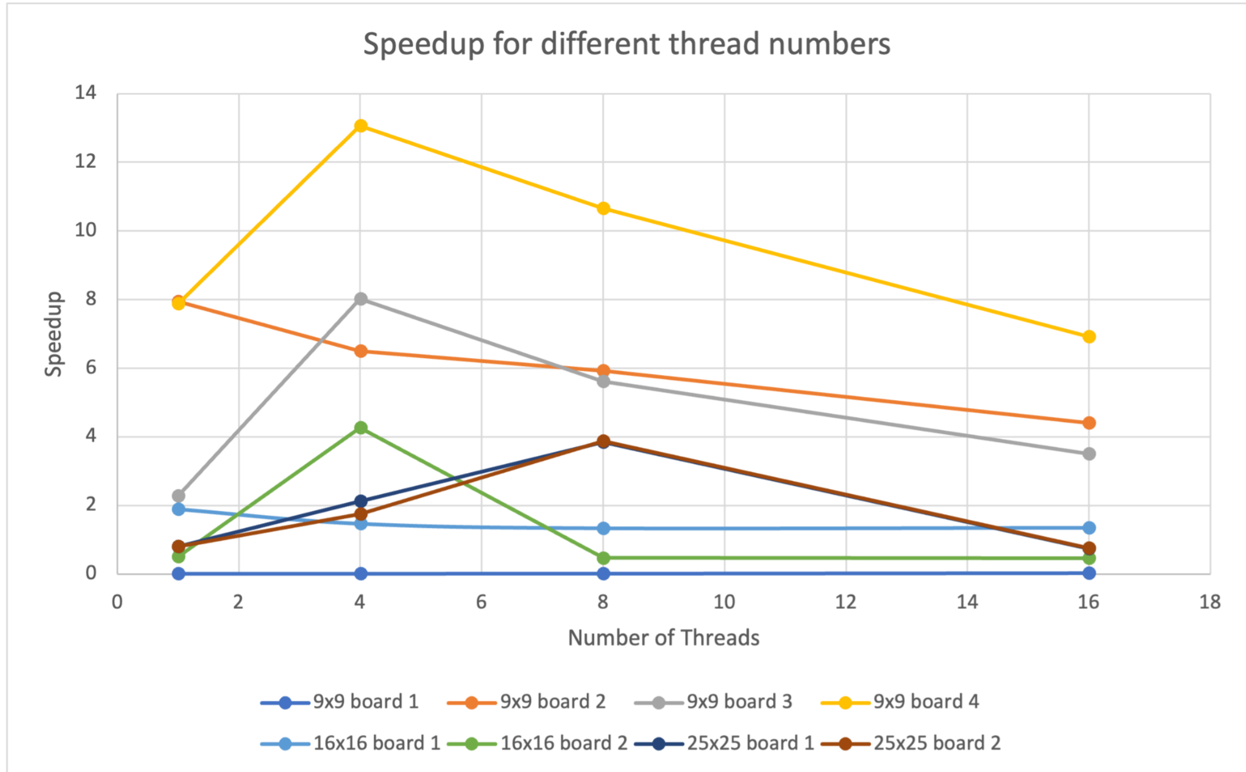
test cases, the algorithm achieves ~13x speedup for board #4, but has worse performance than the serial solution for board #1. This is possibly due to the fact that for easy boards, the overhead cost for distributing parallel work and copying the board for each thread overshadows the parallelization, which leads to better performance for the sequential algorithm.

We can also see that the OpenMP implementation has a overall better speedup for board of size 9x9 compared to boards of larger sizes. This is also a result of overhead cost of the need for duplicating the board for parallel node. When the board is of a larger size, it takes longer for duplicating and deleting boards.

The limitation of the speedup mostly comes from the significant computation time needed for duplicating and deleting the board for each possibility, which increases exponentially with the increase of board size. This contributes to the result that larger boards have lower speedups compared to 9x9 boards.

Table 2 | Computation time for different number of threads

	1	4	8	16
9x9 board 1	0.021069	0.04391	0.0467064	0.0149266
9x9 board 2	0.124703	0.152443	0.166985	0.225213
9x9 board 3	4.5511	1.286	1.83806	2.94243
9x9 board 4	3.45426	2.08194	2.55148	3.93144
16x16 board 1	0.490307	0.635637	0.699994	0.692075
16x16 board 2	26.3241	3.0864	28.3096	28.5189
25x25 board 1	82.4493	31.3824	17.2111	89.9501
25x25 board 2	92.4533	42.4904	19.1571	97.79782



From Table 2 and the figure we can see that, for easier boards, such as 9x9 board 1, 9x9 board 2, 16x16 board 1, the speedup decreases with the increase of thread numbers, largely because larger threads create communication overhead that worsens performance.

For the rest of the boards with higher difficulty, the speedup increases with the increase of thread count, optimizing at either 4 or 8 threads, and then decreases with the continue increase of thread count. This is also a result of the significant communication overhead created by large thread counts, which would decrease performance of the parallelization.

2) MPI

Table 3 | Computation time & Speedup for different board sizes/difficulty levels

	Serial solution	MPI optimal solution	Speedup
9x9 board 1	0.000252532	0.0004293	0.588
9x9 board 2	0.989897	0.727328	1.36

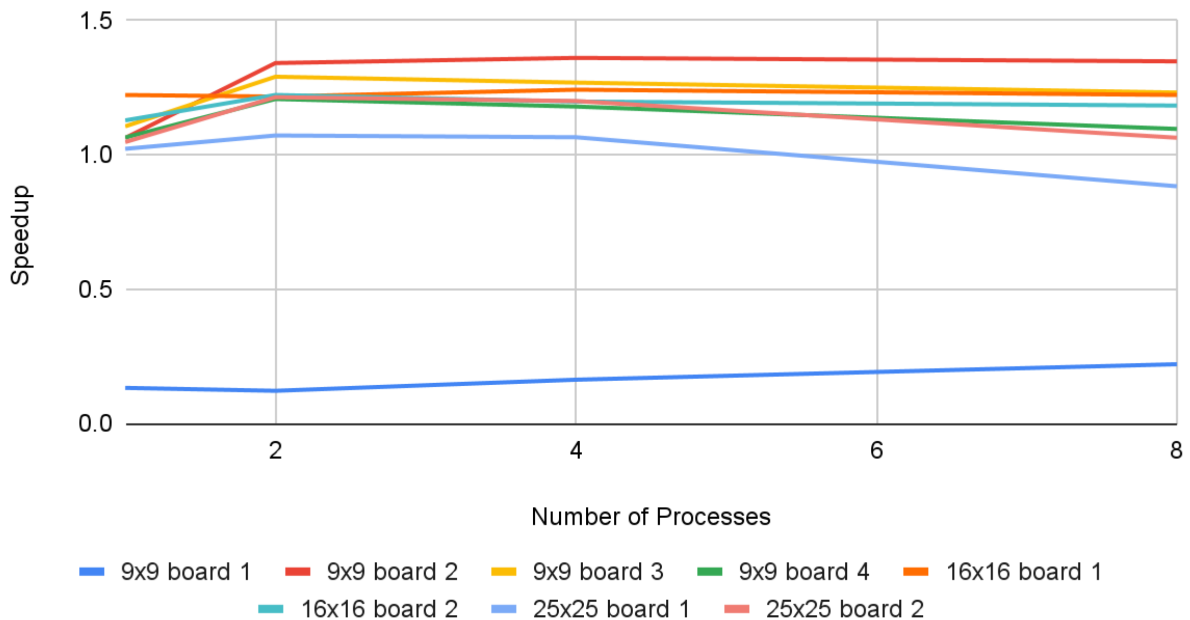
9x9 board 3	10.3193	7.98002	1.29
9x9 board 4	27.1888	22.5078	1.21
16x16 board 1	0.927975	0.746758	1.24
16x16 board 2	13.173	10.737	1.23
25x25 board 1	66.2631	61.0266	1.09
25x25 board 2	74.2406	61.1183	1.22

From Table 3 we that the maximum speed up we get is around 1.2-1.3x. It is fairly constant for different board difficulties and sizes. This is likely due to the threshold value limiting the maximum speedup possible.

Table 4 | Computation time for different number of processes

	1	2	4	8
9x9 board 1	0.00190458	0.0020746	0.00154951	0.00114444
9x9 board 2	0.93137	0.737699	0.727328	0.734227
9x9 board 3	9.32821	7.99397	8.1321	8.37526
9x9 board 4	25.5392	22.5078	23.0442	24.7944
16x16 board 1	0.758806	0.762445	0.746758	0.75876
16x16 board 2	11.674	10.7705	10.9907	11.1303
25x25 board 1	64.8023	61.804	62.1865	75.0331
25x25 board 2	70.861	61.1183	61.844	69.793

*(threshold is at 0.9)

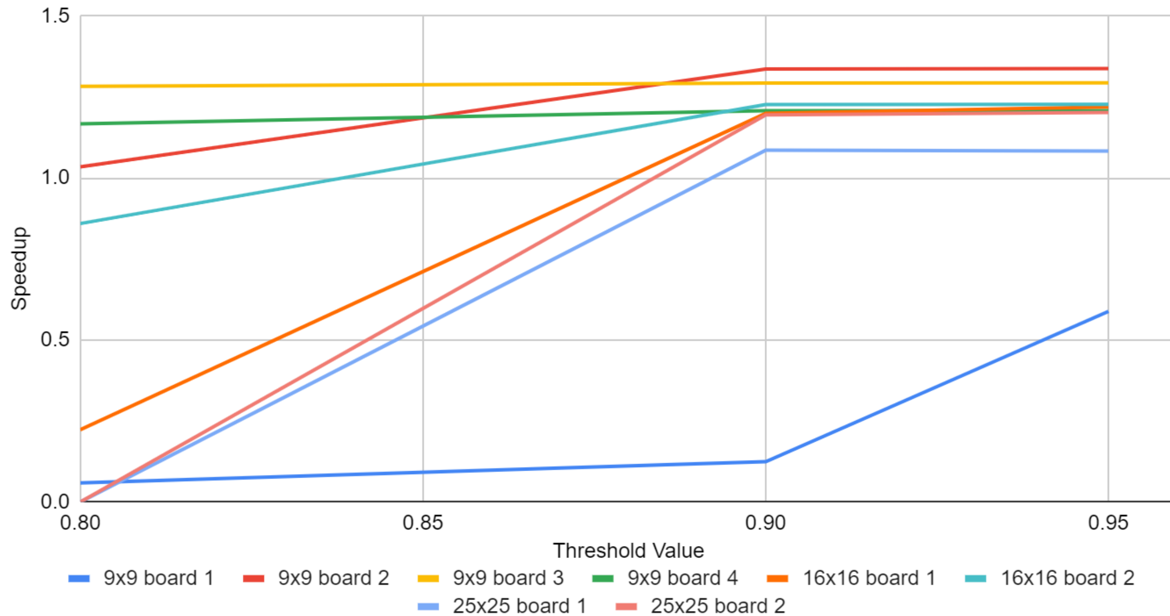


From Table 3 and the figure we can see that speedup increases from 1 to 2 processes, but then the speedup decreases for more processes due to the larger overhead. One thing to note is that the 9x9 board 1 had a very poor speedup. This indicates that for very easy boards it is better to do backtracking the checking constraints and sending messages.

Table 4 | Computation time for different threshold values

	0.8	0.9	0.95
9x9 board 1	0.00428873	0.00203122	0.0004293
9x9 board 2	0.957156	0.741066	0.740079
9x9 board 3	8.04665	7.98351	7.98002
9x9 board 4	23.2974	22.5146	22.5166
16x16 board 1	4.16391	0.77392	0.761235
16x16 board 2	15.3309	10.7411	10.737
25x25 board 1	>10mins	61.0266	61.1863
25x25 board 2	>10 mins	62.1447	61.7627

*(num threads = 2)



From Table 4 and the figure, we can see that 0.9 is a good threshold value for most boards, and too low of a value will result in a speedup of less than 1. It was not shown but a high threshold value will cause the program to behave like the sequential one and have a speedup slightly less than 1 due to overhead from MPI.

Overall the MPI parallelization was not too successful. The constraint satisfaction we used limited the amount of parallelism that could be achieved and we found it difficult to parallelize sudoku using the MPI interface.

References:

1) Backtracking algorithm, https://www.geeksforgeeks.org/sudoku-backtracking-7/#
2) Sudoku solving algorithms, https://en.wikipedia.org/wiki/Sudoku_solving_algorithms
3) A study of Sudoku solving algorithms, https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group6Alexander/final/Patrik_Berggren_David_Nilsson.report.pdf

Work distribution

We distributed the work by 50/50.
Work done by each team member:
Bella:

- Programming the base code and sequential algorithm
- Programming OpenMP algorithm
- Maintaining the project website
- Writing 50% of the report

Derek:

- Programming work queue MPI implementation
- Programming constraint satisfaction MPI implementation
- Writing 50% of the report