15418

Final Report

Parallel Sudoku Generator and Solver with OpenMP and MPI

Liyao Fu(liyaof), Zhongyi Cao(zhongyic)

Github: github.com/LiyaoFu/15418-LiyaoZhongyi-Final
Website: zhongyi-liyao-15418-site.webflow.io/

Spring 2023

Table of Contents

1. SUMMARY

In this project, we designed and implemented multiple parallel sudoku generators and solvers with OpenMP and MPI on the CPU with 8 cores. By conducting runtime and workload analysis, we demonstrate that the parallel sudoku solver with work stealing strategy achieves the best performance.

Github: https://github.com/LiyaoFu/15418-LiyaoZhongyi-Final

2. BACKGROUND

Sudoku is a number placement problem that relies on numbers and combinatorics[1]. For a classic sudoku, the objective is to fill a 9 × 9 grid with digits so that each column, each row, and each of the nine 3 × 3 subgrids that compose the grid contain all of the digits from 1 to 9. The goal of a Sudoku solver is to give solutions to an input that has certain predetermined digits so that the answer will fulfill the requirements for all the columns, rows, and subgrids. A 9 by 9 sudoku puzzle and its solution is shown in Figure 1.

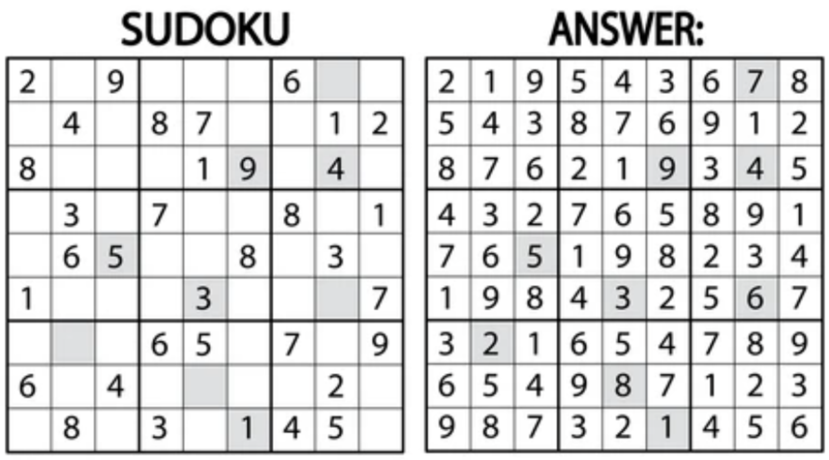


Figure 1: A Classic Sudoku

While the problem is straightforward, the time complexity can be very large. If each Sudoku square has n choices of numbers, and there are m squares to fill in, a naive Sudoku solver will take O(n^m) to finish, which will be terrible even on the latest CPUs. The Intel Core i9, with 18 cores and a price tag of $1,999, can do roughly a trillion operations per second. However, if there are around 30 empty spaces to fill in, the i9 CPU will take around (9^30)/(10^12)=4.24e+16 seconds to compute the result, which is over one billion years. In fact, it was proved in 2001 that the Sudoku problem is NP-hard[2].

While the naive solution is too time consuming, parallelism can be a very useful tool in improving the efficiency to generate sudoku and solve the Sudoku problem. For example, we can parallelize the steps to try different numbers for each grid, and gather the information for whether the assignment is successful from the result of each thread that handles a subgrid, a row, or a column. In our project, we focus on parallelism of both sudoku generator and sudoku solver.

2.1 Sudoku Generator

2.1.1 Generator Constraints

To generate a random Sudoku, we have three key constraints to keep:

a. The sudoku needs to be random. In other words, we will use the thousands of sudokus generated by our generator to be test cases for the solver, and we don't want the generated sudokus to be the same or similar.

b. The sudokus need to satisfy the sudoku constraints. Recall that in an n*n sudoku, every row and column has the numbers 1 to n exactly once, and every square root(n) * square root(n) sub-sudokus has the numbers 1 to n exactly once. The generated sudokus, although many grids are intentionally left blank, should still satisfy the constraints.

c. The sudokus need to be solvable, since we want to use it to test the performance of our sudoku solvers. Notice that not all sudokus that satisfy constraints 1 and 2 are valid sudokus. For example, consider the sudoku in Figure 2. It does not contradict with any of the sudoku rules, with each number appearing at most once in each row, column, and sub-sudoku. However, we cannot insert a 4 into the first row, since we can only insert into column 3 to 8, which will contradict with the existing 4s. We do not want our generator to output a sudoku like this.

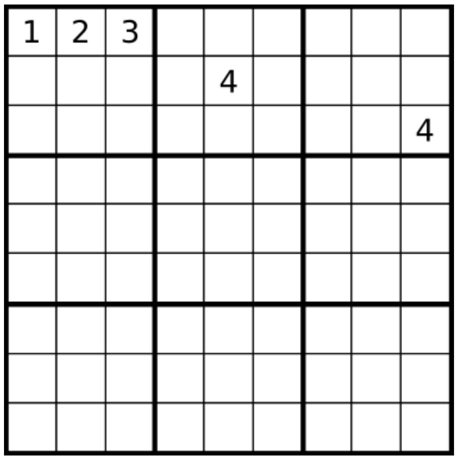


Figure 2. An Unsolvable Sudoku[3]

2.1.2 Sequential Generator Design

With the above three constraints in mind, we can design our sudoku generator. To implement it, we will use matrices and vectors in C++. Basically, we will use a backtracking algorithm to randomly decide what values to fill in for each grid, and build a complete Sudoku first. After that, we randomly remove some of the grids to create a Sudoku puzzle. In the backtracking step, for each of the grids, we will first scan what values are available for the current grid. If there is none, it means our current assignment does not work, and we return to the previous grid. If there are some, we pick a random choice to start, and we keep trying from that random choice until we find a solution for all grid assignments. The detailed design is shown in Figure 3.

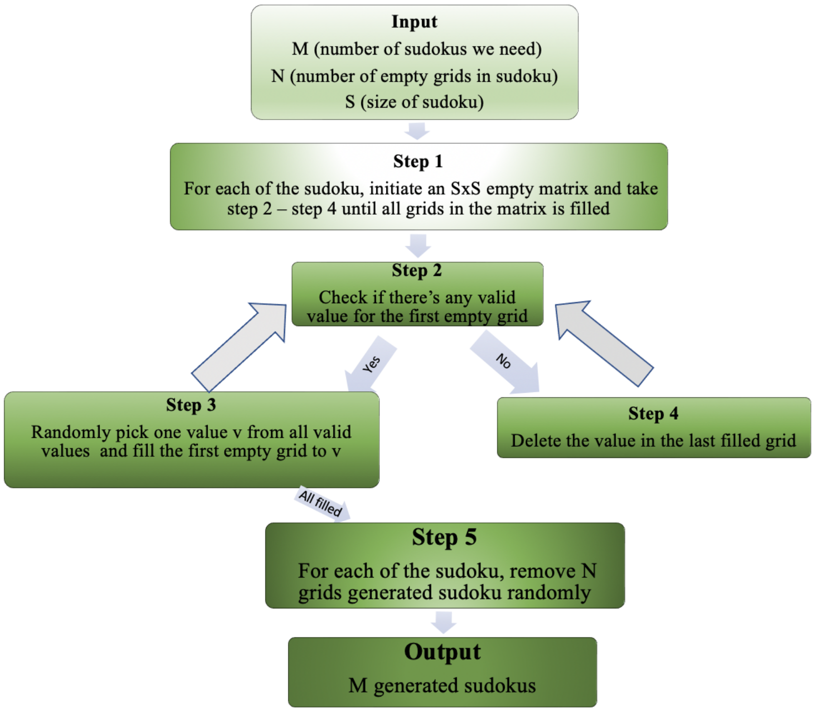

Input
M (number of sudokus we need)
N (number of empty grids in sudoku)
S (size of sudoku)
Step 1
For each of the sudoku, initiate an SxS empty matrix and take step 2 – step 4 until all grids in the matrix is filled
Step 2
Check if there's any valid value for the first empty grid
Yes
No
Step 3
Randomly pick one value v from all valid values and fill the first empty grid to v
Step 4
Delete the value in the last filled grid
All filled
Step 5
For each of the sudoku, remove N grids generated sudoku randomly
Output
M generated sudokus

Figure 3: Sequential Generator Design

Since we choose the numbers randomly in each step and remove grids randomly, the final result will be different every time, and constraint 1 is satisfied. Since when we run the backtracking step, the values are chosen so that the Sudoku constraints are satisfied, constraint 2 is satisfied. Since we construct a full sudoku first then remove some of its grids, the output must have at least one solution, so constraint 3 is satisfied.

Notice that since we will generate thousands of Sudokus, the trials of different values will take the most amount of time. As the backtracking algorithm and the removing algorithm both take in one sudoku to work on at a time, we should parallelize the generation and let each thread work on a part of all the sudokus, since the generation of different sudokus do not depend on each other. Also, the choice of grids can be computed in parallel as well, since not all choices will lead to valid solutions.

The program will benefit significantly from parallelizing the generation of different sudokus and the choice of grids. Since there is no dependency between different sudokus' generations, we can parallelize it easily. For choosing different values of the same grid, we need to be careful about the dependency tree, since all choices of the same grid depend on the same previous choice, and there is high locality between different choices.

2.2 Sudoku Solver

To compute a valid sudoku solution, we need to ensure that every number from 1 to s appears exactly once in every row, every column, and every square root(s) * square root(s) sub-sudoku. In our project, we decided to use a backtracking algorithm similar to the algorithm of the generator.

2.2.1 Solver Constraints

The constraints for the solver is simple, as indicated in the definition of the sudoku:

a. On each row, values 1 to s appear exactly once.
b. On each column, values 1 to s appear exactly once.
c. In each square root(s) * square root(s) sub- sudoku, values 1 to s appear exactly once.

2.2.2 Sequential Solver Design

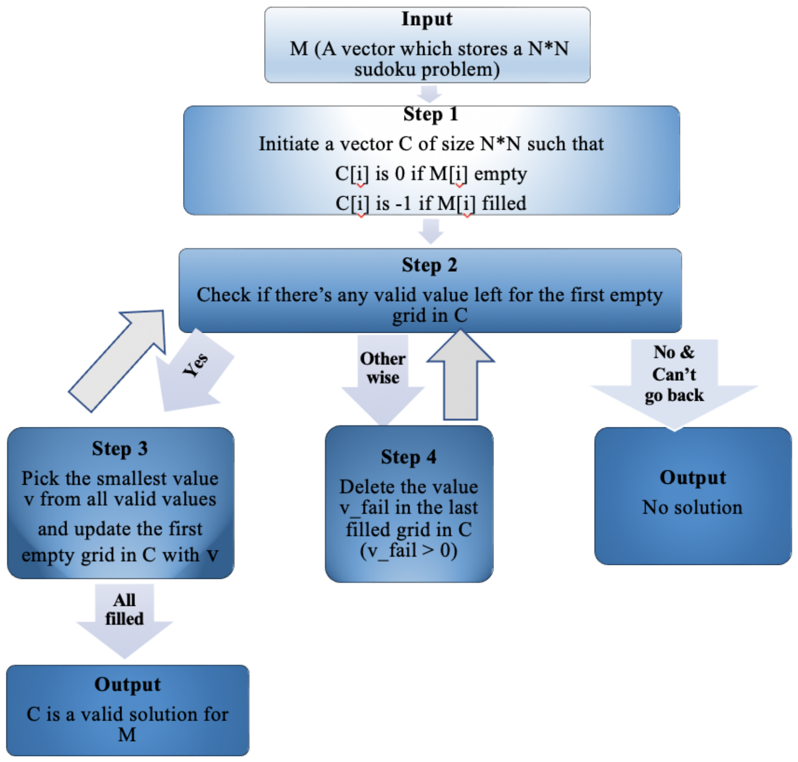

Input
M (A vector which stores a N*N sudoku problem)
Step 1
Initiate a vector C of size N*N such that
C[i] is 0 if M[i] empty
C[i] is -1 if M[i] filled
Step 2
Check if there's any valid value left for the first empty grid in C
Yes
Other wise
No & Can't go back
Step 3
Pick the smallest value v from all valid values and update the first empty grid in C with v
Step 4
Delete the value v_fail in the last filled grid in C (v_fail > 0)
Output
No solution
All filled
Output
C is a valid solution for M

Figure 4: Sequential Solver Design

We use a vector to keep track of the answers, and use backtracking search in the sequential order of the grids.[5] The major difference between solver's algorithm and generator's algorithm is that, instead of starting from a random value, we just start from 1 when performing the trial for each grid. The detailed design is shown in Figure 4 above.

Specifically, in our solve function, we iterate through all the grids from the last solved grid in order to find out which of them does not have a prefilled value. If there is none, we finish the search and have an answer. If there is one, we try values 1 to s on this first grid that requires a value. For each value we try, we first check if it satisfies the three solver constraints. If it does, we recursively solve for the next value. If this assignment has a solution, we return that we found a solution. If the value does not match the constraints or this assignment has no solution, we continue to the next value. If we tried all the values on the current grid but none has a solution, we return to the previous grid with no solution.

Similar to the generator, the key operation is still the trial of values since we will solve for thousands of sudokus. We should parallelize the backtracking solve function as solving each sudoku has no dependency on the others. Also, in the trial of values, we want to try different values in parallel, since unlike the generator, there is no requirement for the order in the solver - we only need one correct solution. Therefore, we can run the trial of sudokus in parallel.

3. APPROACH

3.1 OMP Sudoku Generator

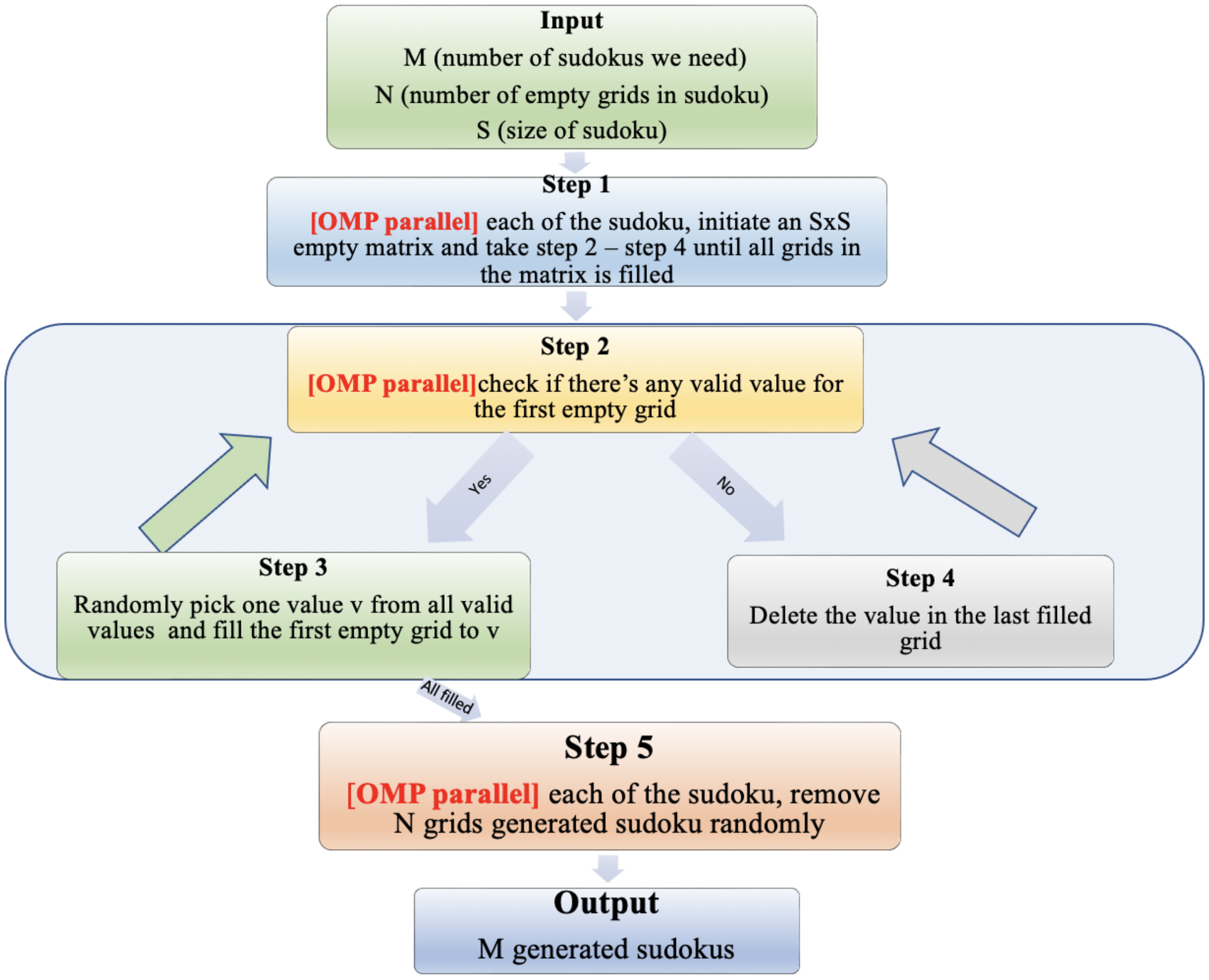

Input
M (number of sudokus we need)
N (number of empty grids in sudoku)
S (size of sudoku)
Step 1
[OMP parallel] each of the sudoku, initiate an SxS empty matrix and take step 2 – step 4 until all grids in the matrix is filled
Step 2
[OMP parallel]check if there's any valid value for the first empty grid
Yes
No
Step 3
Randomly pick one value v from all valid values and fill the first empty grid to v
Step 4
Delete the value in the last filled grid
All filled
Step 5
[OMP parallel] each of the sudoku, remove N grids generated sudoku randomly
Output
M generated sudokus

Figure 5: OMP Sudoku Generator Design

3.1.1 Parallel Design

In our parallel Sudoku generator, we used OpenMP on C++, and we targeted the GHC machines with 8 cores. Our implementation is that the generator runs the solve function and removes the grids for *n* iterations in parallel. Each available thread will work on an evenly distributed proportion of all the sudokus with dynamic scheduling. To store the results, we declare a matrix as a three dimensional global variable with size *n*s*s*, and the threads will store the randomly generated sudokus in the variable. The detailed design is shown in Figure 5 above.

In the solve function, we aim to generate a sudoku for a given index *idx* at matrix[idx], which is passed into the function as a parameter. First, we solve for the possible choices of the current grid in parallel. To distribute the work, we let the available threads examine if each choice of the grid satisfies the sudoku constraints, and store the possible choices in a vector. If the size of the vector is zero, it means we don't have a choice with our current assignment, so we return to the previous step. If not, we randomly select one possible value from the vector, and check if we have a solution. If we are finished, we return. Otherwise, we traverse the possible choices in order from the initially chosen value, and check if each of them leads to a solution in order. If one of them has a solution, we return. If none of them has a solution, it means we don't have a choice with our current assignment, so we return to the previous step.

In the removing step, we randomly remove m grids from the selected matrix, and save the remaining grids that we generated in the solve function.

3.1.2 Change from Serial Version
Compared to the original sequential implementation, we changed a lot of code to enable accelerated parallel computations. Specifically, in the sequential generator, we used only one s*s matrix to store the results for each iteration, and clear it after storing the result. However, since we want to calculate the sudokus on different threads in parallel, we need to make sure that each sudoku does not get overwritten by another thread, so we use a 3-dimensional array instead. Also, since each step of choosing a grid is dependent on the previous step, instead of checking the global matrix variable in the sequential version, we use arrays to represent if each element has been used in each row, column, and sub-sudoku for better parallel backtracking.

3.1.3 Versions and Optimizations
We took six iterations to optimize our code.
[Version 1] First, we started by writing an extremely slow generator. It does not consider the available choices at each grid, and instead keeps choosing a random value for the grid until the value does not violate the rule. This does not work since it cannot satisfy the third generator constraint, as a random assignment does not guarantee it must have a solution. Also, it is extremely slow to run.

[Version 2] We then figured out a way to randomly choose a possible value, but it still does not satisfy the third constraint.

[Version 3] Finally, we start from a random possible value, then iterate through all possible values to ensure the assignment can work.

[Version 4] To optimize the parallelism, we started by applying parallel computation in step 1: solve possible sudokus in parallel. With this, we have a speedup of 1 to 1.5, which is not ideal.

[Version 5] We then applied parallel computation to step 5, and removed the grids in parallel as well. The speedup is higher with this approach, but it's not significant.

[Version 6] Finally, we applied parallel computation to step 2, and we checked the valid values in parallel. This led to a further improvement in the speedup.

3.2 OMP Sudoku Solver

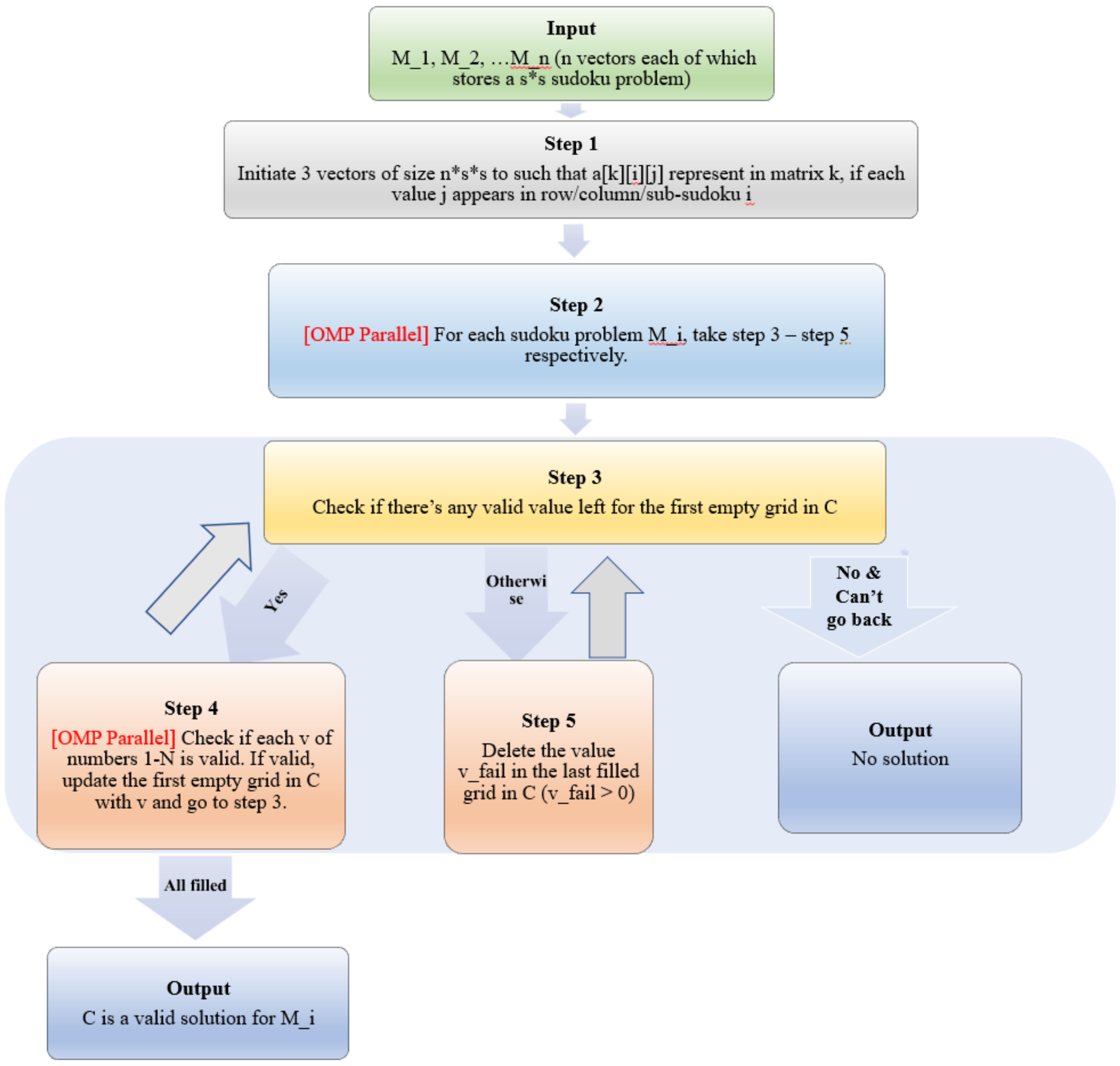

Input
M_1, M_2, …M_n (n vectors each of which stores a s*s sudoku problem)
Step 1
Initiate 3 vectors of size n*s*s to such that a[k][i][j] represent in matrix k, if each value j appears in row/column/sub-sudoku i
Step 2
[OMP Parallel] For each sudoku problem M_i, take step 3 – step 5 respectively.
Step 3
Check if there's any valid value left for the first empty grid in C
Yes
Otherwise
No & Can't go back
Step 4
[OMP Parallel] Check if each v of numbers 1-N is valid. If valid, update the first empty grid in C with v and go to step 3.
Step 5
Delete the value v_fail in the last filled grid in C (v_fail > 0)
Output
No solution
All filled
Output
C is a valid solution for M_i

Figure 6: OMP Solver Design

3.2.1 Parallel Design

In our OpenMP parallel sudoku solver, we used OpenMP on C++, and we targeted the GHC machines with 8 cores. Our implementation is that the solver runs the solve function and determines if each sudoku has an answer for *n* iterations in parallel[6]. Like the parallel generator, each available thread will work on an evenly distributed proportion of all the sudokus with dynamic scheduling. To store the results, we declare a matrix as a three dimensional global vector with size *n*s*s*, and the threads will store the results from backtracking search in the variable. We also have three other vectors to keep track of the backtracking results of the current values in each row, column, and sub-sudoku with size *n*s*s*. The detailed design is shown in Figure 6 above.

In the solve function, we aim to solve a sudoku's first available grid from position [row, col] for a given index *idx* at matrix[idx], while *idx, row, col* are passed in as parameters. We first evaluate the current grid with a while loop: if the grid is empty, we end the while loop and start solving the value for this grid. If we reach the end of the sudoku, the search is complete and we return that we find a solution. Otherwise, we move to the next grid. After this while loop, we are on the first empty grid which needs a solution.

Then, we run a search of the current grid's value in parallel. Each thread will work on one possible value for the current grid, and check if the sudoku constraints are satisfied. If they are, the thread recursively calls the solve function to check whether the current assignment can lead to a correct solution. If it does, we set a return flag to true, since we cannot return that we found a solution in the parallel loop. Otherwise, we just do nothing. After the parallel search, if the return flag is set to true, we return that we found a solution.

3.2.2 Change from Serial Version

To allow fast parallel computation in the OMP solver, we changed a lot of data structures and code. Specifically, in the original sequential version, we used explicit functions to determine the recursive relationship between grid choices and the availability of numbers. However, since we want to enable parallel computation across different sudokus, we change that to a shared memory instead. Also, to allow trials of different values in parallel, we initialized vectors to represent the existence of values in rows, columns, and sub-sudokus and kept them updated in the parallel search.

3.2.3 Versions and Optimizations

We had 3 iterations to optimize the solver.

[Version 1] We started our sequential version by editing an existing sudoku solver at hand [4]. The original code has many bugs and does not support multi-sudoku solving, and we fixed these issues.

[Version 2] Then, when we were trying to parallelize this code, we realized that it's impossible to enable parallelism across different sudokus or within different choices of the same sudoku due to the original data structure's lack of parallelizable work. Therefore, we rewrote the OpenMP code from scratch and fixed all these issues by adding the new data structures to store sudoku status.

[Version 3] When we tried different parallel methods, we found the current two approaches have the largest speedup. Other parallel OpenMP approaches, like in finding the first empty grid or initialization, leads to slower performance due to the large overhead and the small amount of work required to solve those parts sequentially.

3.3 MPI Sudoku Solver

3.3.1 Parallel Design

In our MPI Sudoku solver, we used MPI on C++, and we targeted the GHC machines.To enable better mapping to the message communication in parallelism, we implement the backtracking algorithm with depth-first search instead of recursion in sequential design and maintain the path by a work vector and a status vector[7]. The master process reads in the sudoku problem, assigns work vectors to workers evenly and sends the sudoku problem (Step 1-2). The worker then pops the last node in the work vector, updates the status vector, and adds nodes for the next empty grid to the work vector repeatedly until it finds a solution, or a "stop work" signal is received, or the work vector is empty (Step 3-10). The master then gathers messages from all workers to determine if there's a solution and which solution to print (Step 11). The MPI solver is implemented from scratch and the detailed MPI solver design and mapping is shown in Figure 7 below.

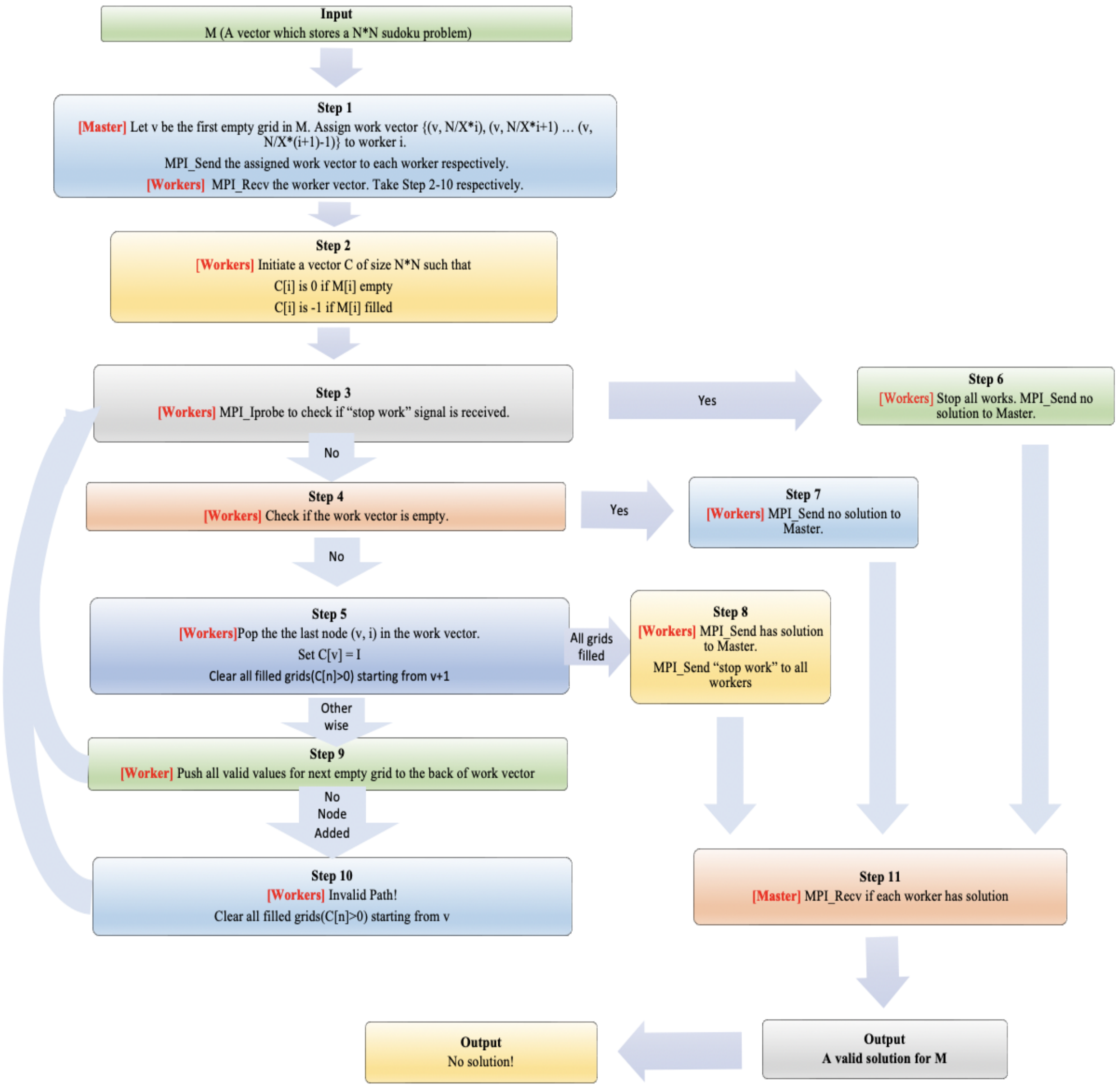

Input
M (A vector which stores a N*N sudoku problem)
Step 1
[Master] Let v be the first empty grid in M. Assign work vector {(v, N/X*i), (v, N/X*i+1) … (v, N/X*(i+1)-1)} to worker i.
MPI_Send the assigned work vector to each worker respectively.
[Workers] MPI_Recv the worker vector. Take Step 2-10 respectively.
Step 2
[Workers] Initiate a vector C of size N*N such that
C[i] is 0 if M[i] empty
C[i] is -1 if M[i] filled
Step 3
[Workers] MPI_Iprobe to check if "stop work" signal is received.
Yes
Step 6
[Workers] Stop all works. MPI_Send no solution to Master.
No
Step 4
[Workers] Check if the work vector is empty.
Yes
Step 7
[Workers] MPI_Send no solution to Master.
No
Step 5
[Workers]Pop the the last node (v, i) in the work vector.
Set C[v] = I
Clear all filled grids(C[n]>0) starting from v+1
All grids filled
Step 8
[Workers] MPI_Send has solution to Master.
MPI_Send "stop work" to all workers
Other wise
Step 9
[Worker] Push all valid values for next empty grid to the back of work vector
No Node Added
Step 10
[Workers] Invalid Path!
Clear all filled grids(C[n]>0) starting from v
Step 11
[Master] MPI_Recv if each worker has solution
Output
A valid solution for M
Output
No solution!

Figure 7: MPI Solver Design

3.3.2 Work Stealing

Workload imbalance is the major issue that limits our speedup in MPI solver. The master distributes the work to workers by dividing the number of starting nodes evenly. However, the length of path for each starting node varies. Thus, some workers might finish early while other workers still have a lot of work to do.

To solve this problem, we design the work stealing strategy to keep all workers busy until all work is done. According to Figure 8, when a worker's work vector is empty, in Step 4 in our MPI solver, it sends "ask for work" status to other workers to check if there's extra work, one at a time. If there's work for it, it receives the new work node and status vector and goes to Step 5. When a worker's work vector has more than one node, in every iteration in the while loop, it does MPI_Iprobe to test if a message is received and if this message is "ask for work". If yes, it sends "work ready" back, and then sends the first node in the work vector and its corresponding status vector.

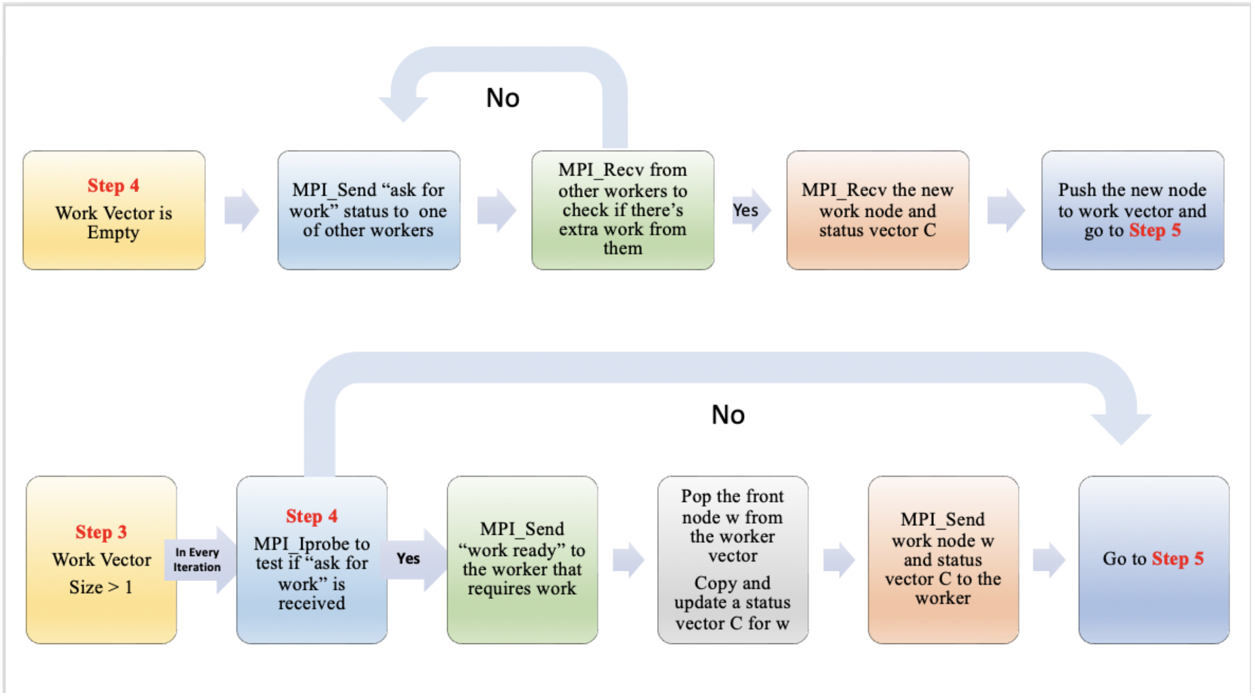

No
Step 4
Work Vector is Empty
MPI_Send "ask for work" status to one of other workers
MPI_Recv from other workers to check if there's extra work from them
Yes
MPI_Recv the new work node and status vector C
Push the new node to work vector and go to Step 5
No
Step 3
Work Vector Size > 1
In Every Iteration
Step 4
MPI_Iprobe to test if "ask for work" is received
Yes
MPI_Send "work ready" to the worker that requires work
Pop the front node w from the worker vector
Copy and update a status vector C for w
MPI_Send work node w and status vector C to the worker
Go to Step 5

Figure 8: Work Stealing Flow

3.3.3 Versions and Optimizations

When we implement the MPI sudoku solver, there are 3 major iterations of optimizations.

[Version 1] In our naive MPI implementation, master performs the major solving step and divides and distributes work evenly by grids to workers in every iteration for checking if the current path is valid. However, we gain even worse performance as the communication cost is more than the parallelism gain in this design.

[Version 2] In our second attempt, the master distributes distinct groups of work to each worker, and the worker communicates with the master and other workers when they find a solution. All workers stop working as long as one of them finds a solution. However, the performance is not ideal as the workload imbalance limits the speedup.

[Version 3] In our final attempt, we add the work stealing strategy to solve the work imbalance issue (see 3.3.2). In short, if a process is available, it asks for work from only one of the other processes. And if a process has extra work, it pops one node from the top and sends its corresponding status vector.

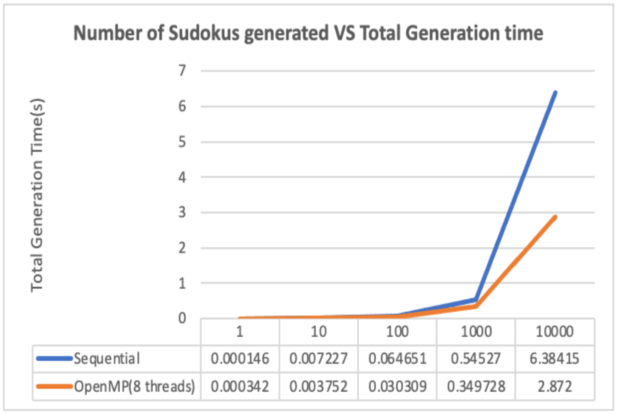
4. RESULTS

4.1 Generator

4.1.1 Metrics

To measure the performance of our generator, we want to compare the runtime of the OpenMP generator against the sequential generator given the same number of sudokus to generate, the same number of empty grids, and the same sudoku sizes. We will use different settings of these parameters to test the speedup on different thread numbers.

4.1.2 Speedup & Analysis



	1	10	100	1000	10000
Sequential	0.000146	0.007227	0.064651	0.54527	6.38415
OpenMP(8 threads)	0.000342	0.003752	0.030309	0.349728	2.872

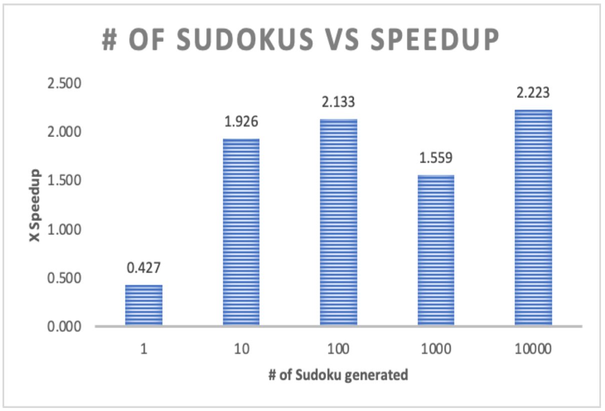
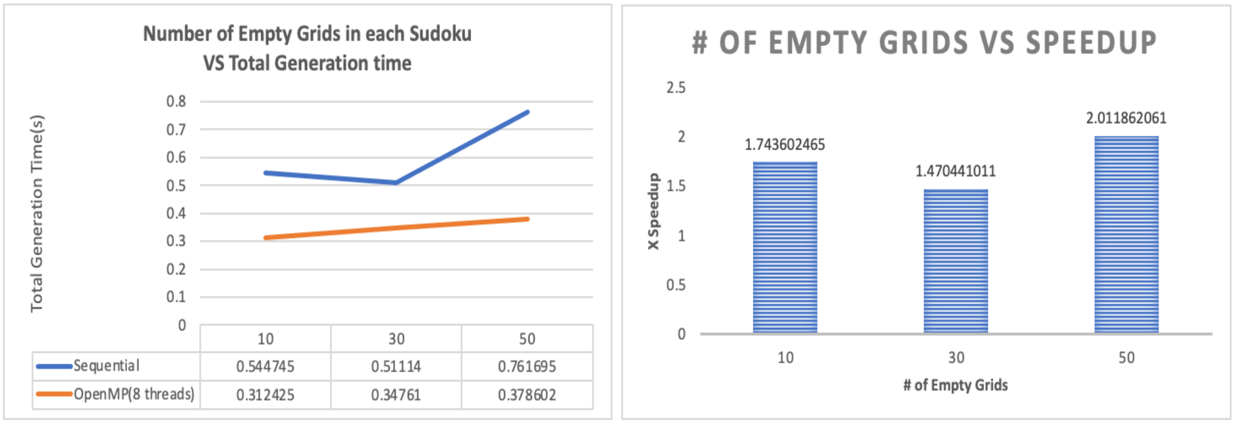


Figure 9: Generator Speedup Graph for different Number of Sudokus

For the sudoku with 30 empty grids, we compared the performance of the sequential generator and the OpenMP generator on 8 threads with different numbers of sudokus to generate. The total generation time for both generators increases approximately linearly with the number of sudokus generated.

According to Figure 9, when only generating 1 sudoku, the OpenMP generator is slower than the sequential generator, due to the workload being smaller than the extra overhead cost by creating the OpenMP threads. When generating 10, 100, 1000, and 10000 sudokus, the generator has a stable speedup of around 2. This is because when the number of sudokus is larger than the number of threads, the reduction in time from speedup is similar.



	10	30	50
Sequential	0.544745	0.51114	0.761695
OpenMP(8 threads)	0.312425	0.34761	0.378602

Figure 10: Generator Speedup Graph for different Number of Empty Grids

For 1000 sudokus, we compare the performance of the sequential server against the OpenMP solver on 8 threads when there are 10, 30, and 50 empty grids respectively. According to Figure 10, the total generation time is approximately the same, so is the speedup.

Consider that we first generate a random sudoku, then remove the grids from it in our algorithm to create empty grids which could satisfy our generator constraints defined in 2.1.1. Since the workload to generate a complete sudoku is much more than the workload to remove random grids, we expect the speedup to be generally similar for the same number of generated sudokus. We measured the time each section takes and the time for reducing the grids takes less than 2% of the total time.

4.1.3 Limitations

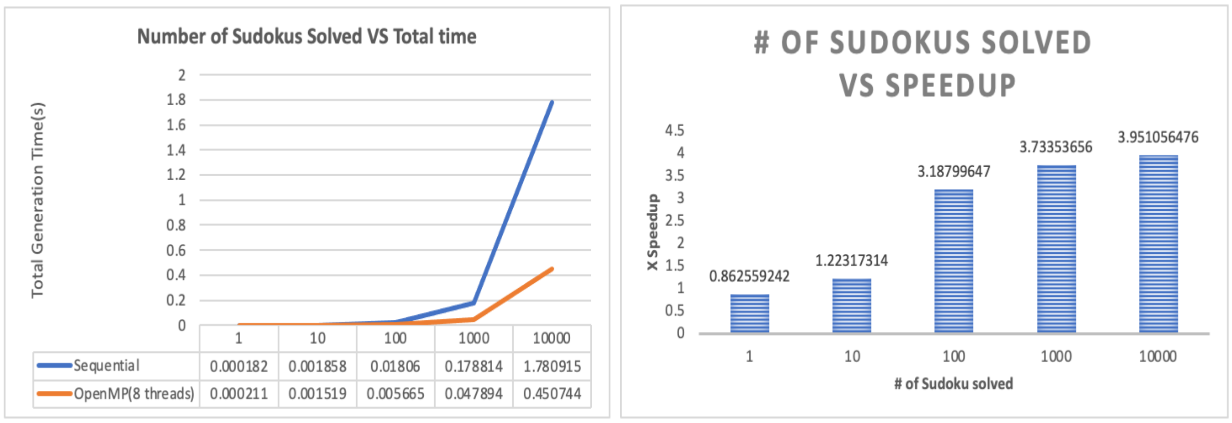
The major limitation of our generator is that we cannot distribute the workload to generate different sudokus fully evenly, especially when the sudoku number is large. Since we apply randomness in our algorithm, we solve a sudoku for an uncertain amount of time. Therefore, the time it takes for each iteration to generate a random sudoku varies a lot. As we measured for each of the 1000 iterations, the lowest runtime and the highest runtime to generate one sudoku differs by 3.723 times. However, those threads which completed first just kept idle, which led to a huge waste of resources.

4.2 OpenMP Solver

4.2.1 Metrics

Similar to the generator, we measure the runtime of the sequential solver and the OpenMP solver with the same number of sudokus to solve or the same number of empty grids. We calculate the speedup by dividing the runtime of the sequential solver with the runtime of the OpenMP solver on 8 cores. Since the complexity of solving a sudoku problem is exponentially related to the number of empty grids, we expect to see change in the runtime and speedup for both different sudoku numbers and different empty grids.

4.2.2 Speedup & Analysis



	1	10	100	1000	10000
Sequential	0.000182	0.001858	0.01806	0.178814	1.780915
OpenMP(8 threads)	0.000211	0.001519	0.005665	0.047894	0.450744

Figure 11: Solver Speedup Graph for different Number of Sudokus

For the sudokus with 30 empty grids, we compared the performance of the sequential solver and the OpenMP solver on 8 threads with different numbers of sudokus to solve first. Similar to the case of the generator, due to the linear increase of problem size, the total generation time for both solvers increases approximately linearly with the number of sudokus generated.

According to Figure 11, when the size is 1, the speedup is smaller than 1 as expected, since the overhead to distribute the jobs is greater than the total amount of work. We do have a certain speedup from 10 sudokus, but it is not sufficient, as we still do not have much work for parallelism. When there are 100, 1000, and 10000 sudokus, the speedup is approximately the same, with a maximum of 4. This is because when we have sufficient sudokus to solve, the overhead of distribution is much smaller than the work to solve the sudokus.

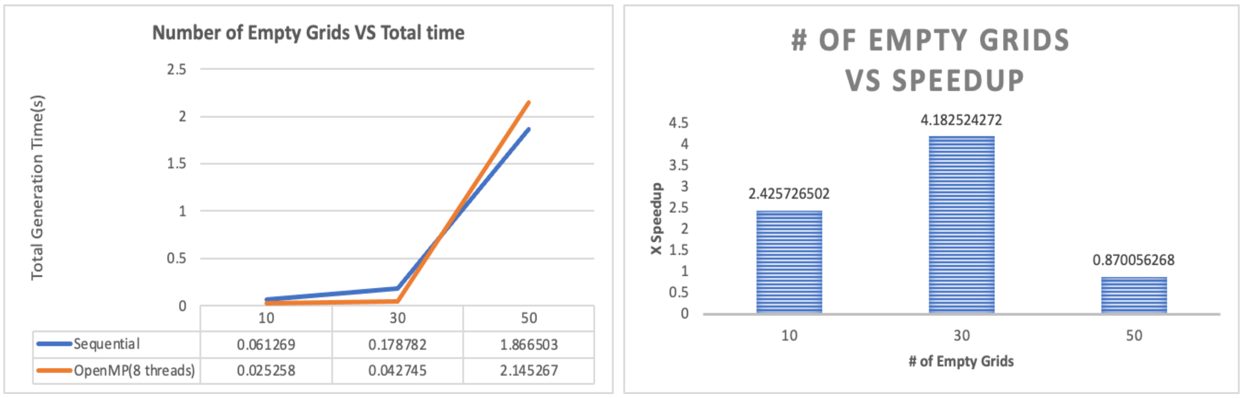


Figure 12: Solver Speedup Graph for different Number of Empty Grids

For the 1000 sudokus with different numbers of empty grids to solve, we compared the performance of the sequential solver and the OpenMP solver on 8 threads. Since the sudoku problem is exponentially harder, the total generation time for both solvers increases approximately exponentially with the number of sudokus generated.

Observing the speedup graph in Figure 12, we see that when there are 10 empty grids, the speedup is around 2.4. When there are 30 empty grids, the speedup is maximized to 4.18. This is because the search tree of the backtracking algorithm is deeper, so it benefited from the parallel computation of different value choices more. However, when there are 50 empty grids, the speedup is reduced to less than 1. This could happen because the width of the search tree for a random grid is highly dependent on the specific sudoku pattern as well, especially when the depth is 50. This will lead to varying time to solve the sudokus in parallel. As we measured for each sudoku, the shortest solving time for 50 empty grids is 0.4432 s, while the longest time is 2.091 s. This 4.7 times difference makes many threads idle when they complete their work, so the speedup is lower than 1 with such waste of resources.

4.2.3 Limitations

While the solver improves performance when solving many sudokus, its performance is not ideal when solving sudokus with many empty grids. The uneven workload for solving random sudokus makes it impossible to distribute the workload evenly with any OpenMP scheduling types, and our parallel OpenMP does not improve the performance at all when it tries to solve a mostly empty Sudoku. The lowest and the highest differ by 5.781 times when there are 50 empty grids. To address this problem, we will switch to MPI solver in the next section, where we deploy work stealing to achieve balanced work distribution.

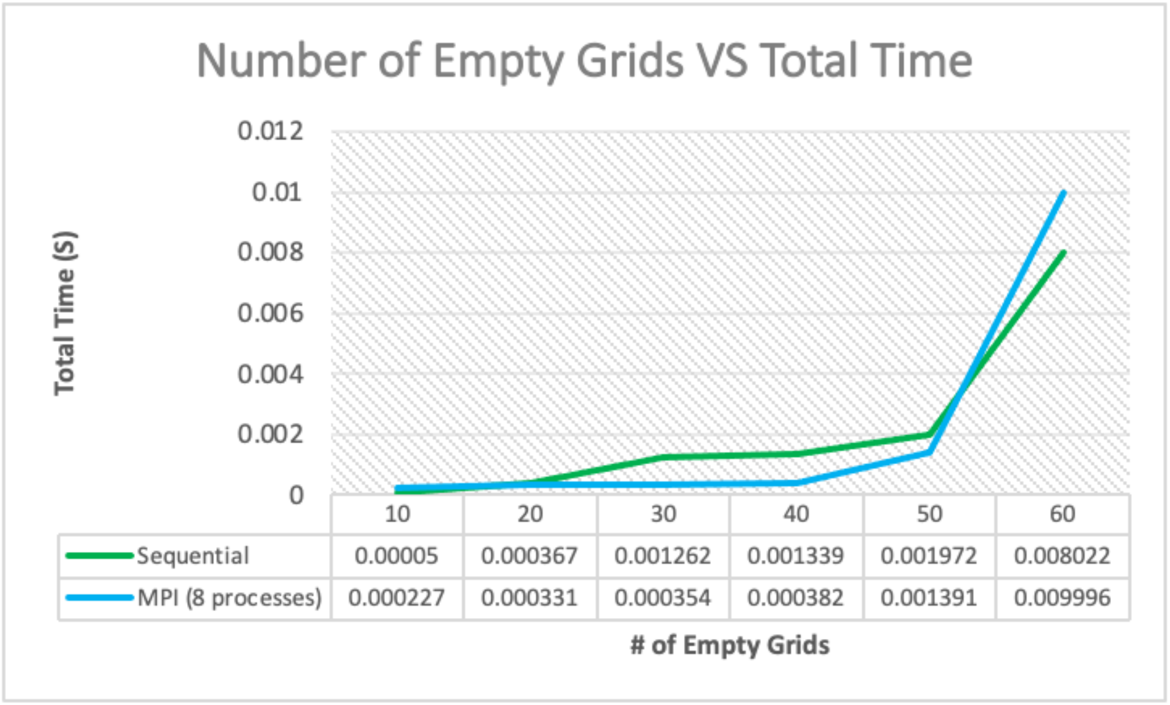
4.3 MPI Solver

4.3.1 Metrics

In the MPI solver, we measure the performance of our solver when solving one sudoku with a different number of empty grids. The speedup will be calculated by the runtime of the sequential solver dividing the MPI solver with 8 workers. Specifically, we will focus on the performance of our MPI solver with and without work stealing when the number of empty grids is large, and evaluate the speedup between the OpenMP version and the MPI version.

4.3.2 Speedup & Analysis

[Version 1] No work stealing



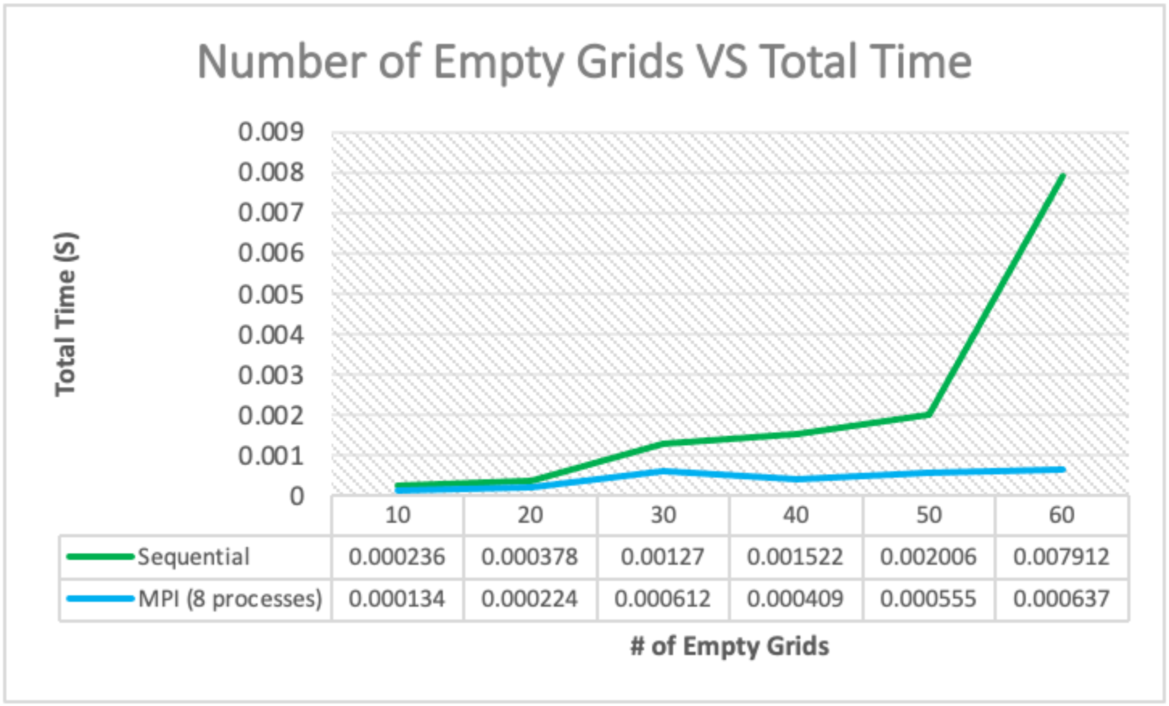
	10	20	30	40	50	60
Sequential	0.00005	0.000367	0.001262	0.001339	0.001972	0.008022
MPI (8 processes)	0.000227	0.000331	0.000354	0.000382	0.001391	0.009996

Figure 13: MPI Solver Runtime for different Number of Empty Grids

According to Figure 13, when there is no working stealing, we see that the MPI solver has similar performance as the OpenMP solver for empty grids 10 to 40. When there are only 10 empty grids, the MPI solver is slower than the sequential version due to the lack of workload compared to the large communication cost. When there are 20 empty grids, the MPI solver is faster by a limited proportion. When there are 30 and 40 empty grids, the MPI solver is much more efficient compared to the sequential solver, but we see that the advantage of parallelism has already declined when there are 40 empty grids. When there are 50 empty grids, the MPI solver is faster by a very limited amount. When there are 60 empty grids, the MPI solver is slower than the sequential version.

Generally, the MPI solver without work stealing has lower performance when the number of empty grids is too small or too large, and performs the best when the workload is in a medium range. This does not improve from the OpenMP solver, so now we implement work stealing.

[Version 2] With work stealing

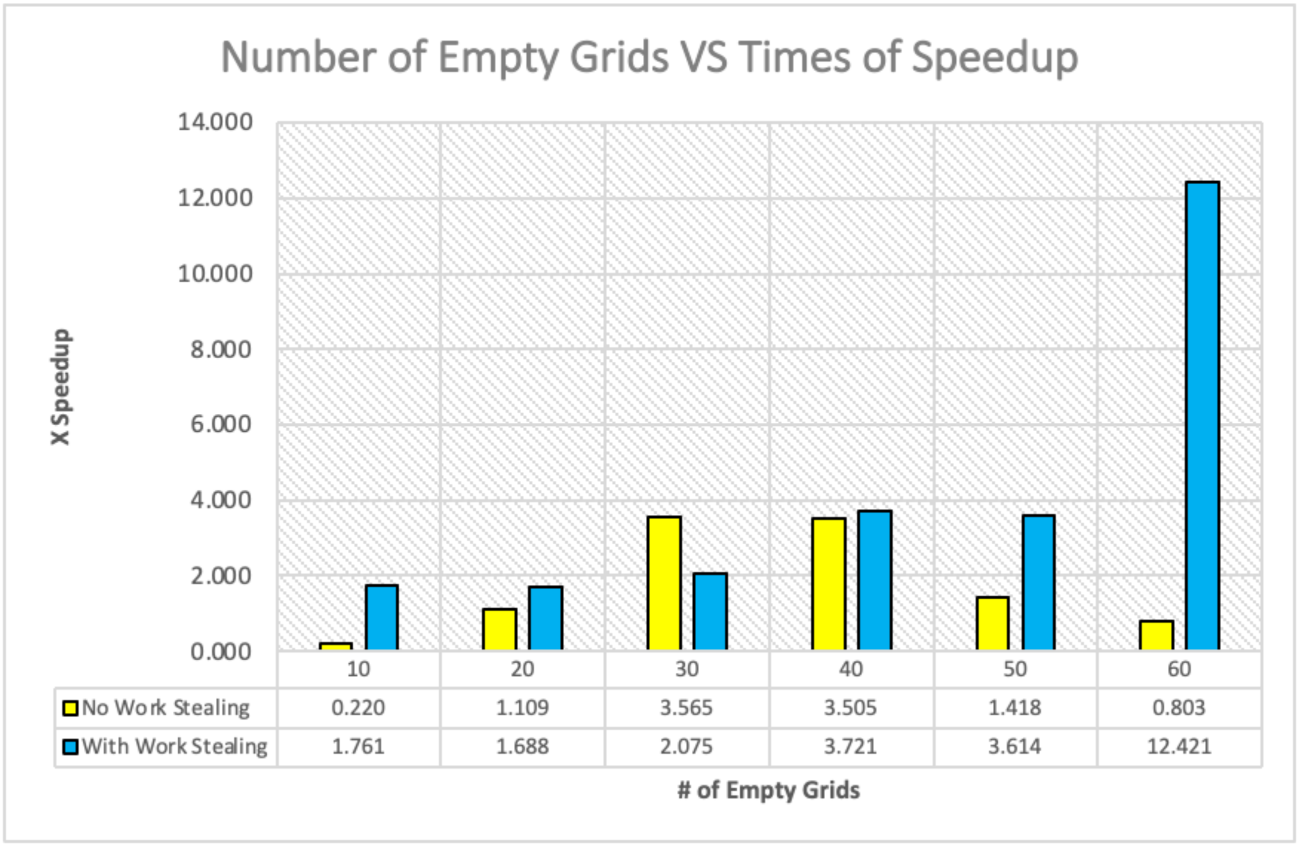


	10	20	30	40	50	60
Sequential	0.000236	0.000378	0.00127	0.001522	0.002006	0.007912
MPI (8 processes)	0.000134	0.000224	0.000612	0.000409	0.000555	0.000637

Figure 14: MPI With Work Stealing Runtime for different Number of Empty Grids

We notice that in Figure 14 with work stealing, the MPI solver performs better than the sequential worker under all circumstances, no matter when the work is too small or too large. While the runtime increased exponentially from 30 empty grids to 60 for the sequential solver, the runtime only increased by a very small amount for the MPI solver with work stealing.

This drastic improvement in performance is due to the way we deal with idle workers. Since most workers will be idle when they finish checking the incorrect values in the original version, we keep sending them other choices to work on so that all workers will approximately have the same workload. Therefore, the balanced workload leads to a significant reduction in the runtime for the MPI solver.



	10	20	30	40	50	60
🟨 No Work Stealing	0.220	1.109	3.565	3.505	1.418	0.803
🟦 With Work Stealing	1.761	1.688	2.075	3.721	3.614	12.421

Figure 15: MPI Without Work Stealing vs MPI With Work Stealing Speedup

From the speedup graph in Figure 15, we find out that the speedup for MPI with work stealing is better than MPI without work stealing for all empty grid numbers, except for 30 empty grids. While the MPI solver without work stealing reaches its peak performance when there are 30 empty grids to solve, the MPI solver with work stealing's speedup improves steadily. When there are 60 empty grids, while the normal OpenMP and MPI solvers perform even worse than the sequential version, the balanced workload helps the work stealing solver reach a speedup of over 12.

It might be surprising that we can achieve a speedup of over 8 since we are running on 8 workers. In fact, this is normal for the large workload cases, because of the way we communicate the workers in the MPI solver. Since idle workers immediately receive hypotheses from other workers to solve, and all workers end whenever one worker finds a solution, they can exploit all the possible choices much earlier than the sequential version without trying all the wrong choices.

4.3.3 Limitations

While our MPI solver with work stealing performs extremely well on large test cases, it still has room for improvement for smaller test cases. Due to the randomness of the sudoku, the runtime of a solver is strongly related to the choices we make as the initial hypothesis. If we stepped into a wrong hypothesis on the top of the tree, all the later choices would be a waste of time. According to our measurement, when the empty grid number is 30, we tried 5 values for the 7th empty grid in the work stealing solver, but only 3 for the original solver. This is why we cannot reach the best performance when the test case is not large enough, since exploiting the hypotheses by redistributing the work could waste extra time when the choices are limited on small test cases.

5. REFERENCES

[1] Wikipedia of Sudoku,
https://en.wikipedia.org/wiki/Sudoku

[2] Sudoku NP problem,
https://www.sciencedirect.com/science/article/pii/S097286001630038X#:~:text=In%202003%2C%20the%20generalised%20Sudoku,be%20certified%20in%20polynomial%20time

[3] Unsolvable sudoku puzzle figure,
https://math.stackexchange.com/questions/4345629/minimum-number-of-clues-for-unsolvable-sudoku#:~:text=A%20Sudoku%20is%20unsolvable%20if,invalid%20Sudoku%20is%20obviously%202.

[4] Sequential solver starter code,
https://stackoverflow.com/questions/68320530/openmp-sudoku-solver-parallelize-algorithm-with-openmp-and-or-mpi

[5] Backtracking sudoku solver design,
https://www.geeksforgeeks.org/sudoku-backtracking-7/

[6] Parallel backtracking,
https://github.com/huaminghuangtw/Parallel-Sudoku-Solver

[7] MPI communication for sudoku,
https://studylib.net/doc/10453280/parallel-sudoku-solver-using-mpi-and-c

6. WORK DISTRIBUTION

Zhongyi and Liyao work together throughout the project. Work was distributed evenly. The total credit should be distributed as **50%-50%**.