# HPC: Parallel Sudoku

Letao Chen, Weijian Feng
{lc5187,wf2099}@nyu.edu
New York University
NY, USA

## ABSTRACT

Final Project of High-Performance Computing, Spring 2023. Solving Sudoku using various parallel methods and comparing their performance.

Letao Chen is responsible for OpenMP section and Weijian Feng is responsible for sequential and Cuda sections.

Code is available at https://github.com/SFARL/HPC_project.git

## KEYWORDS

Sudoku, OpenMP, Cuda

## 1 INTRODUCTION

In Classic Sudoku, the objective is to fill a $9 \times 9$ grid with digits so that each column, each row and each of the nine $3 \times 3$ subgrids that compose the grid contain all the digits from 1 to 9 without duplicates.[4] While the traditional brute force algorithm checks all possibility for each empty cell, we perform parallelization to this algorithm in the checking step in order to accelerate the solving speed for even larger size, i.e., $16 \times 16$. We use two parallelization methods, OpenMP and CUDA.

## 2 ALGORITHM

### 2.1 Sequential Solver

We use the backtracking algorithm as a sequential solver. Backtracking is a general-purpose technique used to systematically explore all possible configurations of a problem space, searching for a valid solution. In the context of Sudoku, backtracking generates solutions by iteratively placing numbers and undoing incorrect choices until a valid solution is found or all possibilities have been exhausted.

**Input** : Sudoku board
**Output**: Solved Sudoku board or Not

**Function** `Backtrack(`*board*`)`**:**
    **if** *board is filled* **then**
        **return** true;
    **end**
    Select next empty cell in the board;
    **for** *digit* ← 1 **to** *order* **do**
        **if** *digit is valid in the selected cell* **then**
            Place the digit in the selected cell;
            **if** `Backtrack(`*board*`)` **then**
                **return** true;
            **end**
            Remove the digit from the selected cell;
        **end**
    **end**
    **return** false;
**end**

**Algorithm 1:** Sequential Solver

### 2.2 Build Sudoku

To accurately assess the average performance of various algorithms, a sufficient number of Sudoku puzzles is required. We used an approach to construct these Sudoku puzzles. The construction process consists of three essential steps. Firstly, a Sudoku board is generated, with numbers specifically placed along the diagonal [1]. Then, a solver is utilized to find the solution for the generated board. Finally, random masking a certain number of cells of the Sudoku puzzle.

**Input** : Order of Sudoku $n$
**Input** : Mask rate of Sudoku $m$
**Output**: A $n$ Sudoku with $m$ empty

**Function** *SudokuBuilder(n)*:
    *board* ← `GenerateSudokuBoardWithdiagonal(`*n*`)`;
    *solution* ← `SolveSudoku(`*board*`)`;
    *puzzle* ← `RandomlyMaskCells(`*solution, m*`)`;
    **return** *puzzle*;
**end**

**Algorithm 2:** Random Build

### 2.3 OpenMP Parallel

*2.3.1* ***Elimination:*** Instead of checking all possible values for an empty cell,we can first elimination the possible values using three strategies explained in [2].

- Elimination: This occurs when there is only one valid value for a cell. In the first example in Fig.1 we can see that te only possible value of the yellow grid is 6.
- Lone Ranger: The number is valid for only one cell in a row, column or a box and it doesn't have anywhere else to go. In the second example in Fig.1 we can see that the only possible value of the yellow grid is 7.
- Twins: A pair of numbers that appears together twice in the same cells and only in those two cells. In the third example we can see that 5,7 is the pair of twins we find in this row.

*2.3.2* ***Stack:*** After we eliminate the possible values for each cell, we implement a stack algorithm for parallelization.

We first create number of threads Sudoku grids with different possible values filled in of the first couple empty cells, then push them to the stack. Each thread pops one grid from the stack, finds the first empty cell, and checks the validation of its possible values, i.e., whether the value filled in will duplicate row/column/box or not. If it is valid to fill in, we create a new grid with this possible value


**Figure 1: Eliminations**

fill in, and check if it solves the Sudoku. If it does, then the solved flag is true and we can store this solved Sudoku and terminate. Else, we update the possible values of other empty cells since they can't be filled in this same value, and push the new grid to the stack. After all children grids are pushed, we delete the parent grid. This whole process can be done using different threads parallelly since they are all individual process, and the algorithm terminates when 1) stack is empty, i.e., we can't find a solution, 2) solved flag is true, which means we find the solution.

```
Input  : Sudoku Stack
Function OpenMPSudoku(Stack):
    while Stack not empty and Not solved do
        Pop the top grid from the stack;
        Find the first empty cell;
        for each possible value of the empty cell do
            Check validation;
            if valid then
                Create a new grid;
                Fill in;
                if grid is the valid solution then
                    solved = true;
                    solvedgrid = grid;
                end
                Update the possible values of other empty
                  cells;
                Push to the stack;
            end
        end
        Delete the parent grid;
    end
end
```

**Algorithm 3:** OpenMP Solver

## 2.4 Cuda Parallel

The parallel backtracking Sudoku algorithm used here is inspired by this GitHub repository[3]. However, this repository can be inefficient in terms of memory usage and may result in missing certain solutions. We have made improvements to address these issues.

The parallel backtracking algorithm for solving Sudoku can be divided into two main steps. First, a breadth-first search is performed to find some possible boards by filling the first X empty spaces, resulting in Y potential starting boards. Then, utilizing parallel processing on the GPU, each of these Y boards is independently attempted to be solved using a sequential backtracking Sudoku solver. If a solution is found for any board, the program terminates and returns. This parallel approach enables simultaneous exploration of the solution space.

Our improvement involves a dynamic approach where we no longer fix the size of the array. Instead, we calculate the number of possible boards on the CPU and then perform parallel processing on the GPU. This allows us to utilize the maximum memory and threads available without the need for recursion in each thread, resulting in reduced processing time. Additionally, following our presentation in class, we discovered that we can leverage more threads on the GPU (up to 1024) and conducted further experiments to explore various thread combinations.

```
Input  : Sudoku s
Input  : BFS Level l
Output : Does solve Sudoku s with BFS level l
Function Cuda Solver(s, l):
    // Calculate the number of possible start boards on CPU
    Nums ← CalBFS(s, l);
    // Generate all the possible boards with numbers Nums
    AllBoards ← GenerateBoard(s, Nums);
    // Solve all boards in parallel on GPU
    CudaSequentialSolver(AllBoards);
end
```

**Algorithm 4:** Cuda Solver

# 3 PERFORMANCE

## 3.1 Sequential Solver Performance

We run our sequential Sudoku Solver in CIMS Machine, with 4 AMD EPYC Processor (with IBPB) CPU and 8GB Memory. Each CPU runs at a frequency of 2894 MHz

In order to assess the average performance of our sequential solver on Sudokus with varying orders and mask rates, we executed the solver on 10 random seed mask Sudokus.

Based on the data presented in Table 1, the average time required to solve Sudokus as the mask rate and order of the Sudokus increase. Particularly, when dealing with an Order 16 Sudoku and a mask rate of 0.6, our sequential solver took more than 10 minutes to find the solution on our machine.

**Input** : Sudoku $s$
**Input** : BFS Level $l$
**Output**: Possible number $Nums$ of Sudoku $s$ with BFS level $l$
**Function** *CalBFS*($s$, $l$):
    $Nums \leftarrow 0$
    **for** $iter \leftarrow 1$ **to** $l$ **do**
        Select next empty cell in the board; **for** $digit \leftarrow 1$ **to** $order$ **do**
            **if** *digit is valid in the selected cell* **then**
                $Nums+ = 1$
            **end**
        **end**
    **end**
**end**
**return** $Nums$;

**Algorithm 5:** BFS find

| Order | Mask Rate | Avg Time |
|---|---|---|
| 9 | 0.2 | 0.000005 |
| 9 | 0.3 | 0.000011 |
| 9 | 0.4 | 0.000020 |
| 9 | 0.5 | 0.000061 |
| 9 | 0.6 | 0.000151 |
| 9 | 0.7 | 0.017999 |
| 16 | 0.2 | 0.000022 |
| 16 | 0.3 | 0.000056 |
| 16 | 0.4 | 0.000315 |
| 16 | 0.5 | 0.138878 |

**Table 1: Sequential Sovler Performance**

## 3.2 OpenMP Performance

We run our sequential Sudoku Solver in CIMS Machine, with 4 AMD EPYC Processor (with IBPB) CPU and 8GB Memory. Each CPU runs at a frequency of 2894 MHz

*3.2.1* ***Increase thread:*** Theoretically, increasing the number of threads will accelerate the speed of computation. However since the architecture we use has 4 processor, the decrease of spent time is significant when number of threads increase from 1 to 4, and no much improvement afterwards.

*3.2.2* ***Increase mask rate:*** There are two problem sizes we use: 9×9 and 16×16, and we want to compare the execution time needed for two problem sizes with different mask rate. Theoretically time needed for both sequential and OpenMP method increases as mask rate increases, and OpenMP should spend less time than sequential method. However, sequential solver beats OpenMP solver when the order is 9 for every mask rate as shown in Fig 3. This is possibly because the problem size of $9 \times 9$ is so small that the time needs for brute force is less than our elimination checking, pushing and popping from the stack. We can also tell from the plot that time needed for OpenMP solver is stable compared with sequential solver,

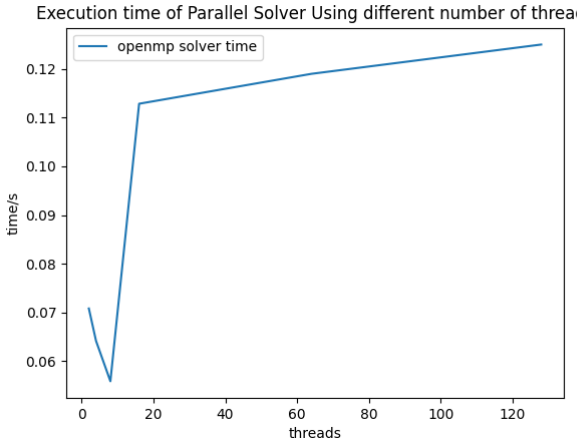


**Figure 2: Order 16, mask rate =0.6**

which is also a consequence of the simplicity of the Sudoku when the order is 9.

When the size is 16×16, we can observe a significant acceleration especially when mask rate = 0.6 as shown in Fig 4. It takes the sequential solver more than 20 seconds to solve such a Sudoku while OpenMP solver can usually finish within 2 seconds.

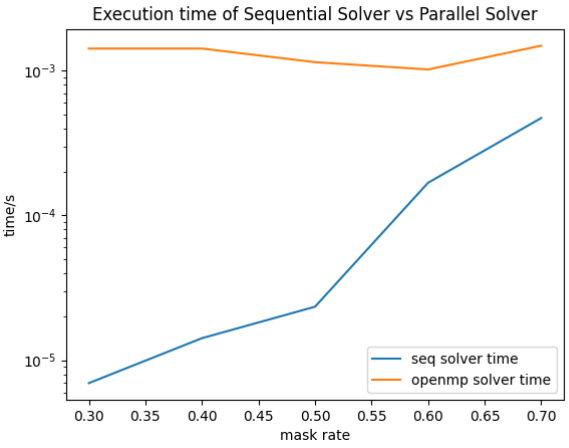


**Figure 3: Order 9: Seq v.s. OpenMP**

## 3.3 Cuda Solver Performance

We deployed our CUDA Sudoku Solver on the Greene Cluster, employing an NVidia V100 GPU with 10GB of memory. By harnessing the GPU's parallel processing capabilities, we aimed to enhance the solver's performance.

To assess the solver's average performance across Sudokus of varying orders and mask rates, we carried out a series of 10 experiments. These experiments involved the utilization of seed mask Sudokus generated randomly. We ensured that the random seed

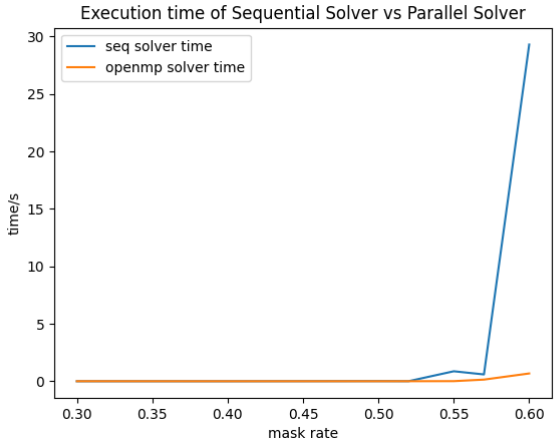


**Figure 4: Order 16: Seq v.s. OpenMP**

settings remained consistent with those employed in the sequential solver, guaranteeing a fair and unbiased comparison between the two approaches.

*3.3.1* ***Increase thread:*** Theoretically, increasing the number of threads in parallel processing should lead to a decrease in search time. This is primarily because a higher number of threads allows for a greater portion of the search tree in the backtracking process to be explored simultaneously. Consequently, the height of the search tree, which corresponds to the number of backtracking steps required, decreases as well. In terms of the relationship between thread numbers and search time, it is expected to follow a logarithmic curve. Initially, as the number of threads increases, the reduction in search time is less significant.

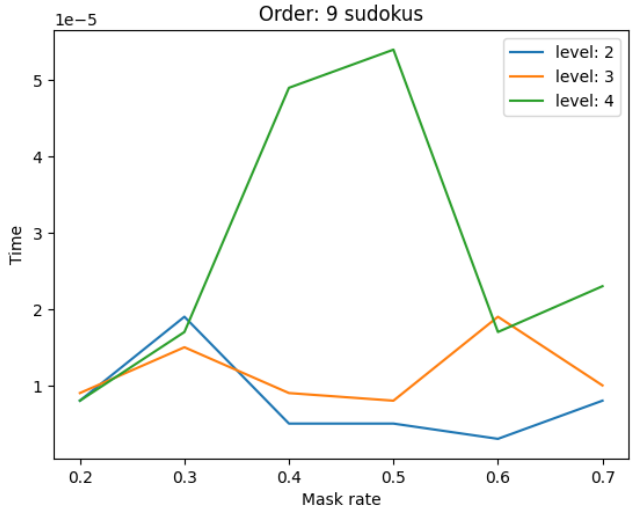


**Figure 5: Order 9: Level V.S. Time**

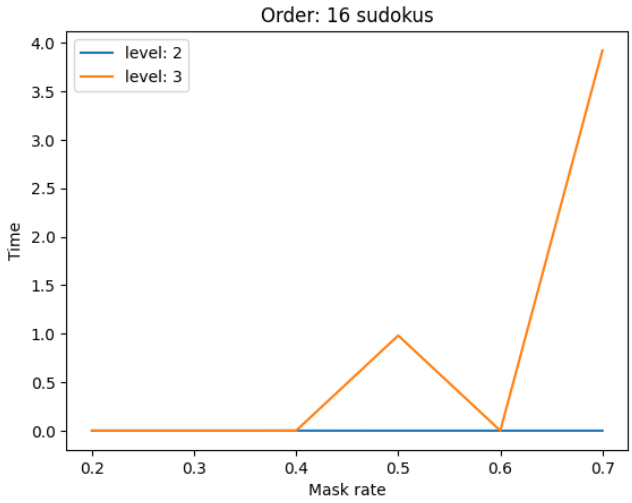


**Figure 6: Order 16: Level V.S. Time**

Based on the observation from Figure 5 and 6, where the time did not decrease as the BFS search level increased (indicating an increase in the number of start boards and threads).

*3.3.2* ***Increase problem size:*** Theoretically, as the problem size increases, the search time in solving Sudoku is expected to increase. This is because the larger problem size corresponds to a greater number of possible configurations and increases the search space that needs to be explored. The relationship between problem size and search time often follows a pattern similar to an exponential curve. Initially, as the problem size grows, the search time tends to increase at a relatively faster rate. This is due to the exponential growth in the number of possible configurations to be evaluated.

The behavior is seen in Figure 5, where the solving time does not consistently increase as the mask rate increases. As for Figure 6, the increase in solving time with higher mask rates can be partially attributed to the splitting of grids and the increased time required for accessing global memory. Splitting grids can introduce additional overhead, especially when multiple threads access global memory simultaneously. This overhead can impact performance and result in longer solving times.

*3.3.3* ***Using Block or Grim:*** When all threads are confined within a single block, the best performance is observed. This behavior is intuitive since solving Sudoku puzzles involves frequent memory access. Utilizing shared memory within a block proves advantageous, as it provides faster memory access compared to accessing data from global memory.

Regarding the impact of a larger grid dimension (grim dim), it is noteworthy that solving times increase significantly. Specifically, for order 9 Sudokus, it takes at least 1 second, while for order 16 Sudokus, the solving time extends to approximately 2 seconds.

## 4 CONCLUSION

The sequential algorithm for a 9x9 Sudoku puzzle often exhibits faster solving times due to its simplicity and reduced overhead associated with memory access and communication between parallel algorithms. Since the puzzle size is smaller, the sequential approach can efficiently explore the search space and find a solution without the need for parallelization.

There are many ways to design a suitable parallelization algorithm using OpenMP or MPI, our project shows one that involves simple elimination strategies and a stack algorithm. The stack might overflow and crushes the program when the number of threads is large, and further attempt can be done by combining parallelization and the brute force to accelerate the speed for both easy and difficult Sudokus.

Utilizing a GPU for parallel Sudoku solving may not be the most effective approach due to the inherent characteristics of the problem. Sudoku solving is not well-suited for extensive parallelization since it involves frequent memory access and requires relatively less computational work.

However, as the puzzle size and difficulty increase, parallel algorithms can offer advantages in solving more complex Sudokus (Order 16, Mask Rate > 0.6) and lead to performance improvements. The parallel approach allows for distributing the workload among multiple threads or processors, enabling simultaneous exploration of different branches of the search tree.

## REFERENCES

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Unsupervised Multitask Learners. *OpenAI Blog* 1, 8 (2020). https://chat.openai.com/

[2] Sruthi Sankar. 2014. Parallelized Sudoku Solving Algorithm Using OpenMP. (2014). chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Sankar-Spring-2014-CSE633.pdf

[3] vduan. 2015. Parallelized Sudoku Solver on the GPU. (2015). https://github.com/vduan/parallel-sudoku-solver/tree/master

[4] Wikipedia. 2013. Sudoku. (2013). https://en.wikipedia.org/wiki/Sudoku