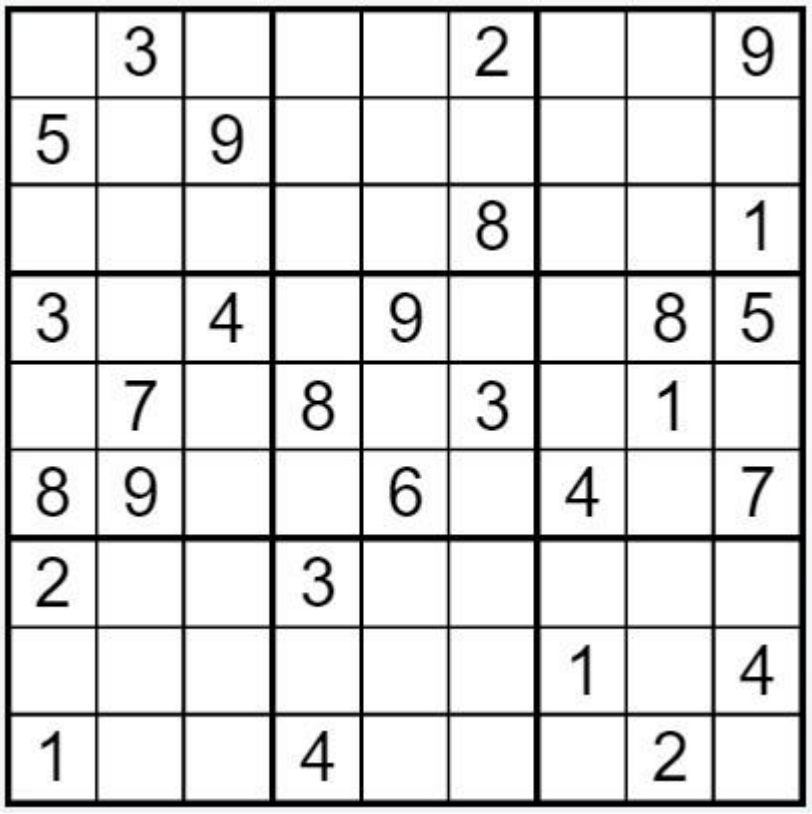
## Introduction

For this project, my teammates and I implemented a **Sudoku solver** both using sequential and parallel programming to evaluate the efficiency of both methods in solving a Sudoku puzzle. The open mpi library was used for the parallel implementation on a High-Performance Computing Center (HPCC).

Sudoku is a challenging logical game played on a board which could be of several grid sizes but typically has a 9 x 9 grid size consisting of 9 cells each consisting of 9 boxes making a total of 81 boxes. The goal of the game is to complete the puzzle filling in numbers from 1 to 9 in all the empty boxes without duplicates in each row, column and cells.



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

*Figure 1: Sudoku Puzzle [1].*

**Implementation**

Sequential Implementation:

The sudoku solver was writing with c++ code. It was written within a single main() function from which a “void solveSudoku()” will be called to implement the sequential sudoku solver. The sudoku board was implemented as a 2D array which will be passed into the “solveSudoku()” function. Deque data structures were used in the code to contain data corresponding to the empty cells in the sudoku and a <vector> was used to contain number choices to be filled into an empty cell. Within the function, (1) two nested for loops where used to navigate through the entire 2D array. From within the for loops, if the iterators(i,j) were on a filled box (represented by a number other than 0 in the 2D array) the iterators will skip but if the iterators where on an empty box (represented by a non-zero number) a for loop will take a number from the vector which will then be checked against three functions (foundinCell,foundinRow and foundinCol) to check that the number was not in a cell, row or column. If the above functions all returned ‘false’ the number could then be used to fill the empty cell. This step continues for as many empty cells can be filled. In the end, all the empty cells might be filled, or some might be empty. If some cells are empty an algorithm like the *Backtracking* algorithm is called. The algorithm consists of two functions “search_empty_cell()” and “sudoBackTrack()”. The search_empty_cell() searches dequeues containing data corresponding to the empty cells for data indicating an empty cell. Once it finds it, it calls sudoBackTrack() to backtrack through the deque until a data corresponding to a filled cell is encountered and then another number is chosen to fill that cell after checking against constraints if it passes the function attempts to solve next box otherwise another backtracking takes place. This process continues until all the cells have been filled.

Parallel Implementation:

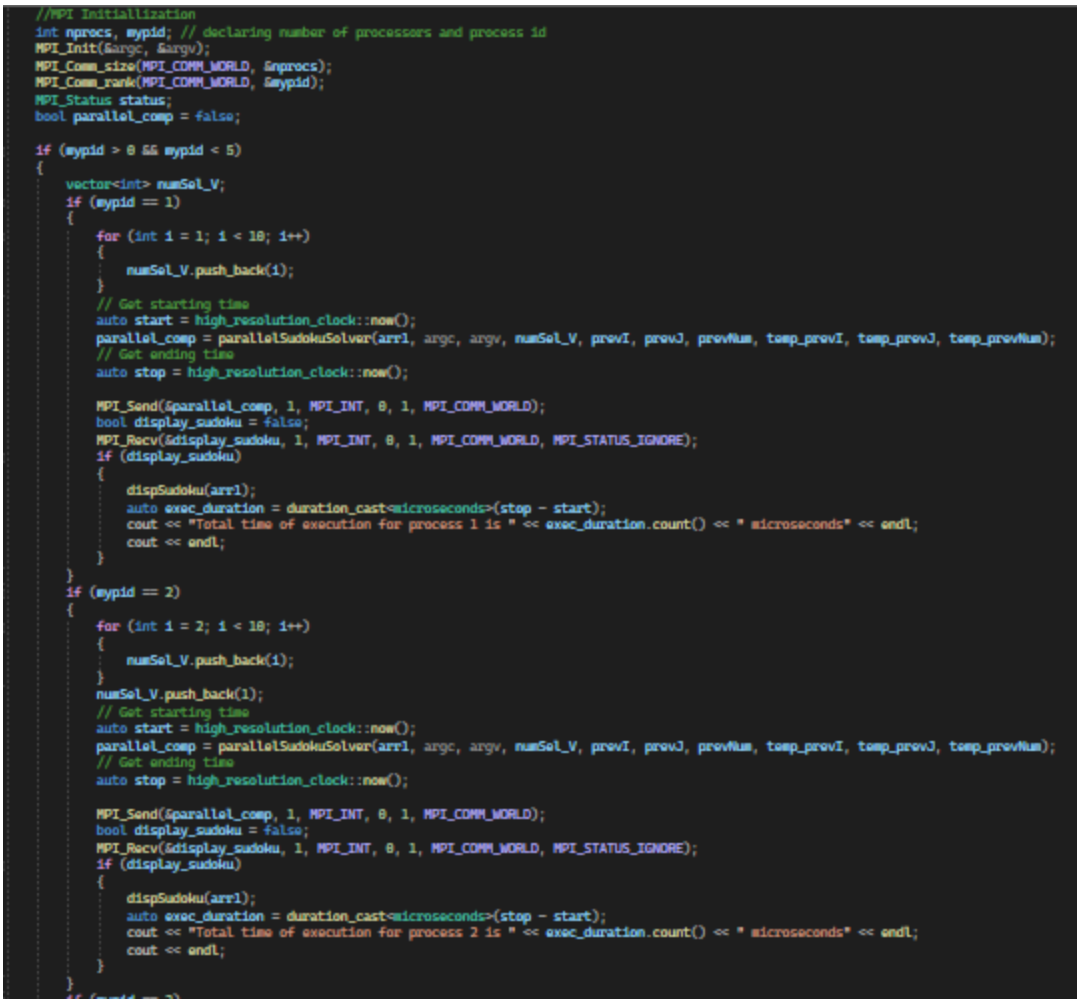
For the parallel implementation, “open-mpi” library is used in the C++ source code. The parallel implementation involves using multiple processes (4 in this case) to each implement different number permutations of 1 – 9 to attempt to fill an empty cell. Whichever process solves the sudoku puzzle first returns its solution and aborts the other process. This will make the sudoku solver more efficient.

## Sample Code

```
void solveSudoku(int arr[9][9], deque<int> provI, deque<int> provJ, deque<int> provNum, deque<int> temp_provI, deque<int> temp_provJ, deque<int> temp_provNum)
{
    int cnt = 0;
    string cell_unsolved_flag = "false";
    string exit_loop = "false";
    for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 9; j++)
        {
            if (arr[i][j] != 0)
            {
                continue;
            }
            else
            {
                for (int num = 1; num < 10; num++)
                {
                    int foundinRo = foundinRow(arr, i,num);
                    int foundinCol = foundinColumn(arr, j,num);
                    int foundinC = foundinCell(i, j, arr, num);
                    if (foundinRo == 0 && foundinCol == 0 && foundinC == 0)
                    {
                        arr[i][j] = num;
                        cnt++;
                        exit_loop = "true";
                        //populate queues with i,j and num values for filled cell
                        provI.push_back(i);
                        provJ.push_back(j);
                        provNum.push_back(num);
                        break;
                    }
                }
                if (exit_loop == "false")
                {
                    cnt++;
                    //populate queues with i,j and 0 value for unfilled cell
                    provI.push_back(i);
                    provJ.push_back(j);
                    provNum.push_back(0);
                    cell_unsolved_flag = "true"; //flag to check if all sudoku empty cells have been solved
                    exit_loop = "false";
                }
                exit_loop = "false";
            }
        }
    }
//  dispSudoku(arr);
    if (cell_unsolved_flag == "true")
    {
        // pass queues for i,j and num values of previous solved empty cells into SudoBackTrack() function to solve remaining unsolved cells
        search_empty_cell(arr, provI, provJ, provNum, temp_provI, temp_provJ, temp_provNum);

    }
}
```

*Figure 2:solve sudoku()*



```
//MPI Initiallization
int nprocs, mypid; // declaring number of processors and process id
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &mypid);
MPI_Status status;
bool parallel_comp = false;

if (mypid > 0 && mypid < 5)
{
    vector<int> numSol_V;
    if (mypid == 1)
    {
        for (int i = 1; i < 10; i++)
        {
            numSol_V.push_back(i);
        }
        // Get starting time
        auto start = high_resolution_clock::now();
        parallel_comp = parallelSudokuSolver(arr1, argc, argv, numSol_V, provI, provJ, provNum, temp_provI, temp_provJ, temp_provNum);
        // Get ending time
        auto stop = high_resolution_clock::now();

        MPI_Send(&parallel_comp, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        bool display_sudoku = false;
        MPI_Recv(&display_sudoku, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if (display_sudoku)
        {
            dispSudoku(arr1);
            auto exec_duration = duration_cast<microseconds>(stop - start);
            cout << "Total time of execution for process 1 is " << exec_duration.count() << " microseconds" << endl;
            cout << endl;
        }
    }
    if (mypid == 2)
    {
        for (int i = 2; i < 10; i++)
        {
            numSol_V.push_back(i);
        }
        numSol_V.push_back(1);
        // Get starting time
        auto start = high_resolution_clock::now();
        parallel_comp = parallelSudokuSolver(arr1, argc, argv, numSol_V, provI, provJ, provNum, temp_provI, temp_provJ, temp_provNum);
        // Get ending time
        auto stop = high_resolution_clock::now();

        MPI_Send(&parallel_comp, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        bool display_sudoku = false;
        MPI_Recv(&display_sudoku, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if (display_sudoku)
        {
            dispSudoku(arr1);
            auto exec_duration = duration_cast<microseconds>(stop - start);
            cout << "Total time of execution for process 2 is " << exec_duration.count() << " microseconds" << endl;
            cout << endl;
        }
    }
```

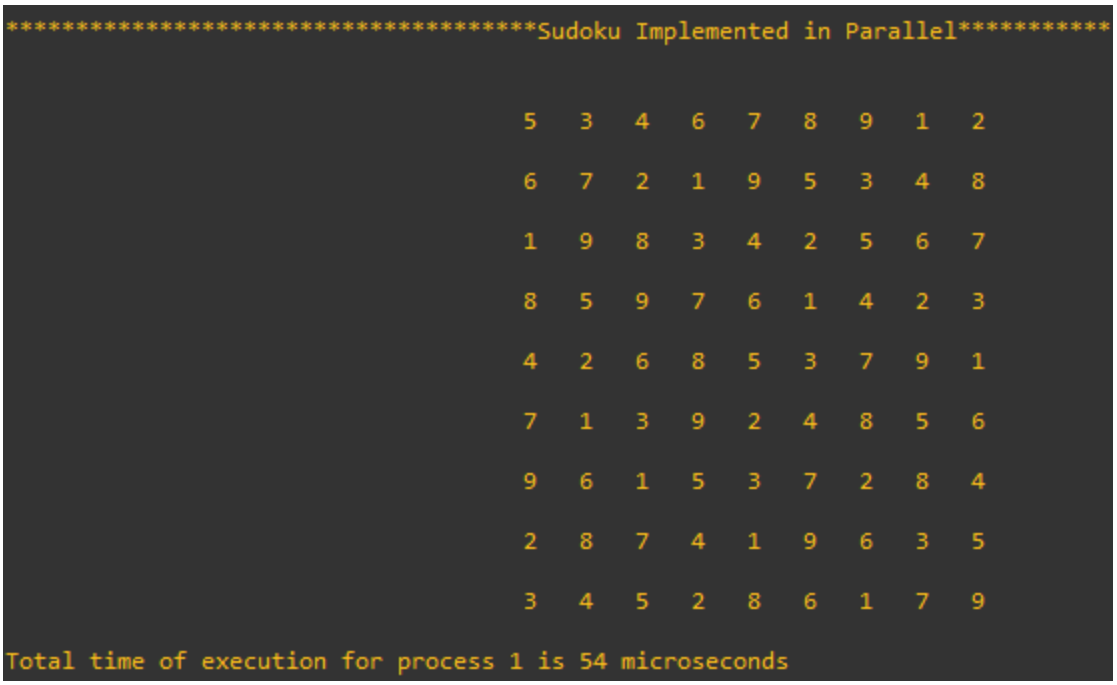
*Figure 3:parallel sudoku implementation using MPI*

## Result

```
**********************************Sudoku Implemented Sequentially**************************

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


                                5  3  4  6  7  8  9  1  2
                                6  7  2  1  9  5  3  4  8
                                1  9  8  3  4  2  5  6  7
                                8  5  9  7  6  1  4  2  3
                                4  2  6  8  5  3  7  9  1
                                7  1  3  9  2  4  8  5  6
                                9  6  1  5  3  7  2  8  4
                                2  8  7  4  1  9  6  3  5
                                3  4  5  2  8  6  1  7  9
Total time of execution is 82 microseconds
```




**Conclusion and Discussion**

By using different number permutations with different processes, the sudoku implementation time is reduced as can be seen in figures above with Sequential implementation taking 82 microseconds to execute while Process 1 (the first process to complete for the parallel process) takes 54 microseconds.

The complexity of the sudoku solver increases depending on the number of empty cells to be filled. With medium to hard puzzles (greater than 45 empty cells), more recursion calls need to be made within the backing algorithm which leads to a Stack Overflow or segmentation fault on C++. The Backtracking algorithm needs to be implemented more efficiently to combat this issue.

## References

**[1]** Sudoku of the Day. https://www.sudokuoftheday.com/free/fiendish.

**[2]** MPI Forumn. https://www.mpi-forum.org/docs/mpi-2.1/mpi21-report-bw/node45.htm

**[3]** Recursive Backtracking for Solving 9*9 Sudoku Puzzle. Bonfring International Journal of Data Mining. Job, Dhanya, Paul, Varghese. https://www.researchgate.net/publication/303553939_Recursive_Backtracking_for_Solving_99_Sudoku_Puzzle