# EGRE 531 Multicore and Multithread Programming

# Laboratory Number 4

# Date: 03/30/18

**Luis Barquero**

**PLEDGE: _____Luis Barquero_____**

**“On my honor, I have neither given nor received unauthorized aid on this assignment”**

VCU | Virginia Commonwealth University

## Introduction:

Part 1 of the lab involved implementing a Sudoku solver either through backtracking recursion or through stack backtracking. Part 2 involved the process of parallelizing the Sudoku solver.

## Lab Content:

**Part 1:** Using the source files provided, the Sudoku solver was implemented using backtracking recursion. Backtracking recursion requires the solver to locate an empty slot and populate it with a number, check for conflicts in the rows, columns and blocks, and if there are no conflicts, the solver proceeds to the next empty slot. If there are conflicts, the solver will go back a previous step and change the number, checking for conflicts once again. This process is repeated until all empty slots are filled.

The solver was implemented so that once an empty spot is found, the solver populates the spot with a number, checks for conflicts, and if checks pass, it recursively calls on the function again so that the solver moves on to the next slot and populates it with another number. Once again, it checks for conflicts, and once it is found that the whole grid is filled a number and there are no conflicts in rows, columns, and slots, the grid is set to be solved, at which point the solved grid is printed.

Figure 1 shows the function used to implement the solver.

```
void RecursiveSolver::solve(int count) {
    // Write your own recursive solver here!
    //int tn = 2; //the number of threads
    //omp_set_num_threads(tn); //you can also set the nu
    int depth;
    if(count == grid.get_slot_number())
    {
        solved = true;
        grid.print_grid();
    }

    else
    {
        Slot s;

        for(int i = 1; i <= 9; i++)
        {
            s = grid.get_empty_slot(count);
                {
                    if(!solved)
                    {
                        grid.set_digit(s.row, s.column,
                        if(check(s))
                        {
                            solve(count + 1);
                        }
                    }
                }
        }
        if(!solved)
            grid.set_digit(s.row, s.column, 0);
    }
}
```

***Figure 1 – Figure 1 shows the Sudoku Solver function.***

**Part 2:** To properly parallelize it, the solver class was cloned and a depth of less than 10 was implemented to prevent a spawn of infinite amount of threads. The reason for this is that the solver function is a recursive solver, so every time it gets called, it will spawn a new thread, and since the function is populating and checking the number for conflicts, if the depth is not implemented, there will be hundreds of threads spawned, which produces overhead.

## Test Results:

**Part 1:** For part 1, the solver takes in a pre-defined sudoku problem, and the program recursively finds the solution, at the end of which it prints out the solved Sudoku grid. Figure 2 shows the solved Sudoku grid, alongside the time elapsed.

```
C:\Users\ASUS\Documents\VCU_Courses\EGRE 531\Sudoku\Sudoku>make
g++ -c -o Sudoku.o Sudoku.cpp -fopenmp
g++ -o Sudoku Sudoku.o RecursiveSolver.o StackSolver.o SudokuGrid.o -fo

C:\Users\ASUS\Documents\VCU_Courses\EGRE 531\Sudoku\Sudoku>sudoku
Please type in the problem file name: test.txt

Please select a solver:
       1. Recursive Solver
       2. Stack Solver
What is your choice? (1/2):
1
The problem to be solved is:

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


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

0Elapsed time: 0.127



C:\Users\ASUS\Documents\VCU_Courses\EGRE 531\Sudoku\Sudoku>
```

***Figure 2 – Figure 2 shows the output from Part 1, which solves the Sudoku problem, in addition to the time elapsed.***

**Part 2:** To properly parallelize the loop, a child-inheritance is implemented to check for the depth of the program in order to prevent an infinite amount of threads being spawned. In this case, the highest depth will be assigned to 9, since a full sudoku grid is 9x9, with 9 3x3 smaller grids containing numbers between 1 and 9. From there, a #pragma omp parallel default(private) to allow all variables from the outer scope to be shared in a parallel region.

From there, the number of threads were modified, and each thread gets 5 cases to fully test the parallelization. The following is a table for all tests and their execution time, alongside Figures 3,4,5 and 6 which shows a graph of the parallelization results with 2, 3, 4 and 5 threads respectively.

| | | | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 |
|---|---|---|---|---|---|---|---|
| Threads | 2 | | | | | | |
| | | Thread 1 | 14.247 | 7.269 | 10.595 | 7.131 | 11.263 |
| | | Thread 2 | 4.164 | 4.001 | 3.43 | 3.589 | 4.589 |
| | 3 | | | | | | |
| | | Thread 1 | 12.619 | 13.848 | 12.148 | 14.444 | 11.611 |
| | | Thread 2 | 5.292 | 8.851 | 8.401 | 8.162 | 8.543 |
| | | Thread 3 | 3.854 | 4.549 | 3.842 | 4.407 | 5.572 |
| | 4 | | | | | | |
| | | Thread 1 | 7.569 | 9.59 | 8.484 | 9.972 | 8.059 |
| | | Thread 2 | 3.614 | 2.351 | 2.978 | 2.3 | 2.774 |
| | | Thread 3 | 1.661 | 1.827 | 1.842 | 1.464 | 1.865 |
| | | Thread 4 | 1.374 | 1.284 | 1.627 | 1.172 | 1.238 |
| | 5 | | | | | | |
| | | Thread 1 | 2.932 | 3.151 | 3.168 | 4.023 | 3.766 |
| | | Thread 2 | 2.788 | 2.803 | 3.128 | 3.018 | 2.55 |
| | | Thread 3 | 2.382 | 2.417 | 2.714 | 2.302 | 2.42 |
| | | Thread 4 | 1.833 | 1.394 | 2.277 | 2.292 | 2.25 |
| | | Thread 5 | 1.701 | 1.17 | 1.855 | 2.01 | 1.808 |

***Table 1 – Table 1 shows the data acquired for the parallelization of the Sudoku Solver.***

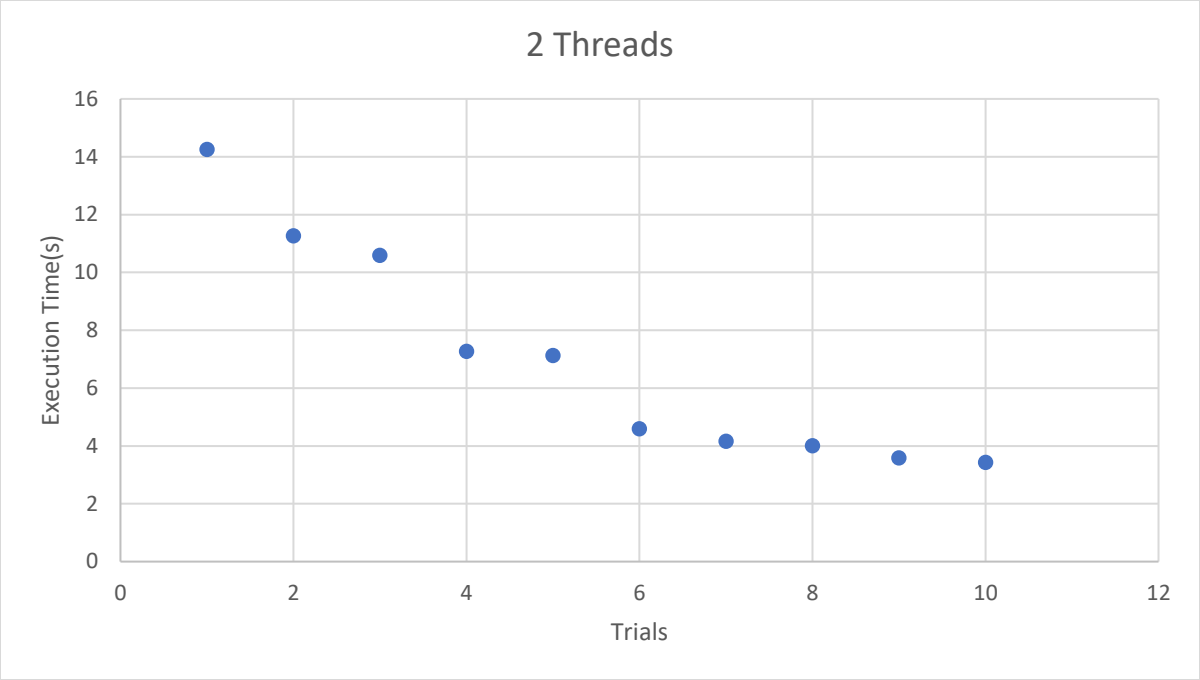


***Figure 3 – Figure 3 shows the parallelization results of the Sudoku solver using 2 threads.***

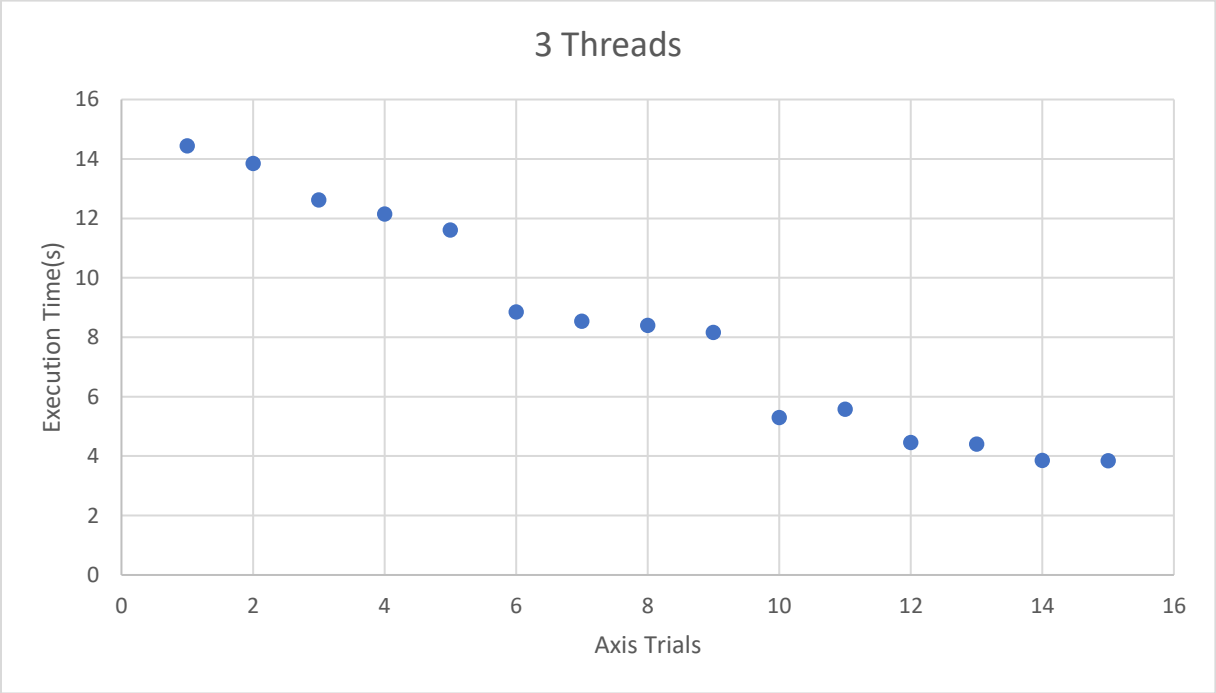


***Figure 4 – Figure 4 shows the parallelization results of the Sudoku solver using 3 threads.***

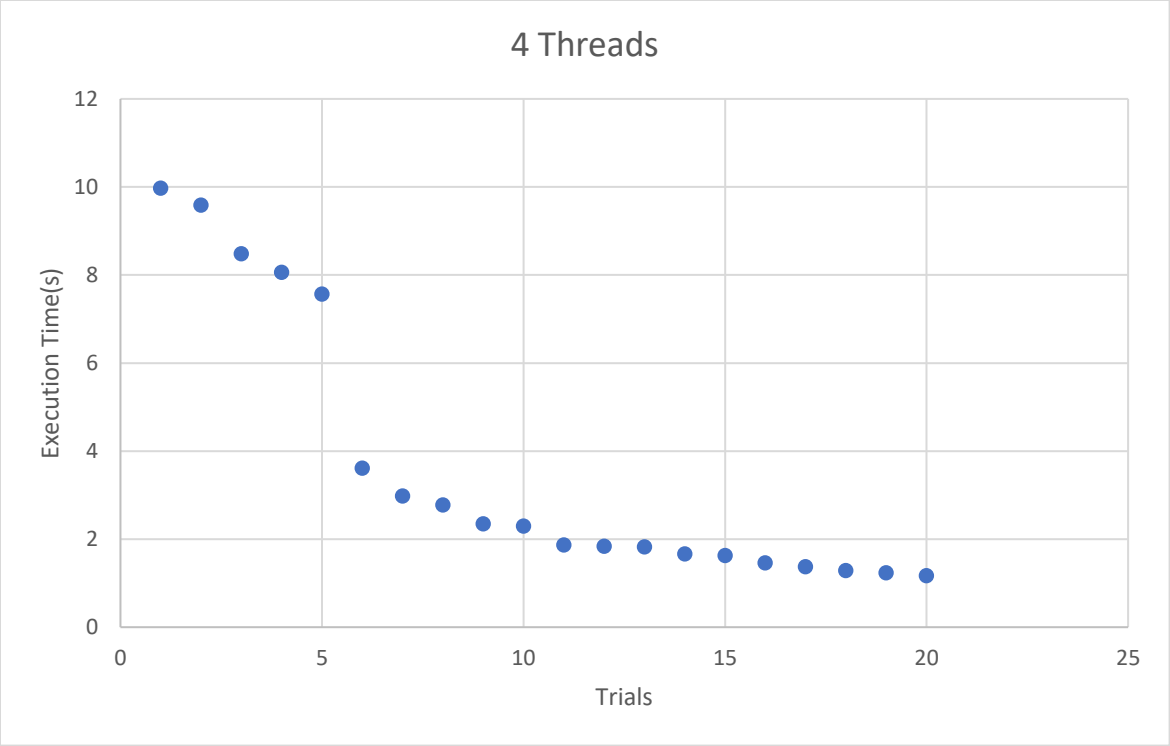


***Figure 5 – Figure 5 shows the parallelization results of the Sudoku solver using 4 threads.***

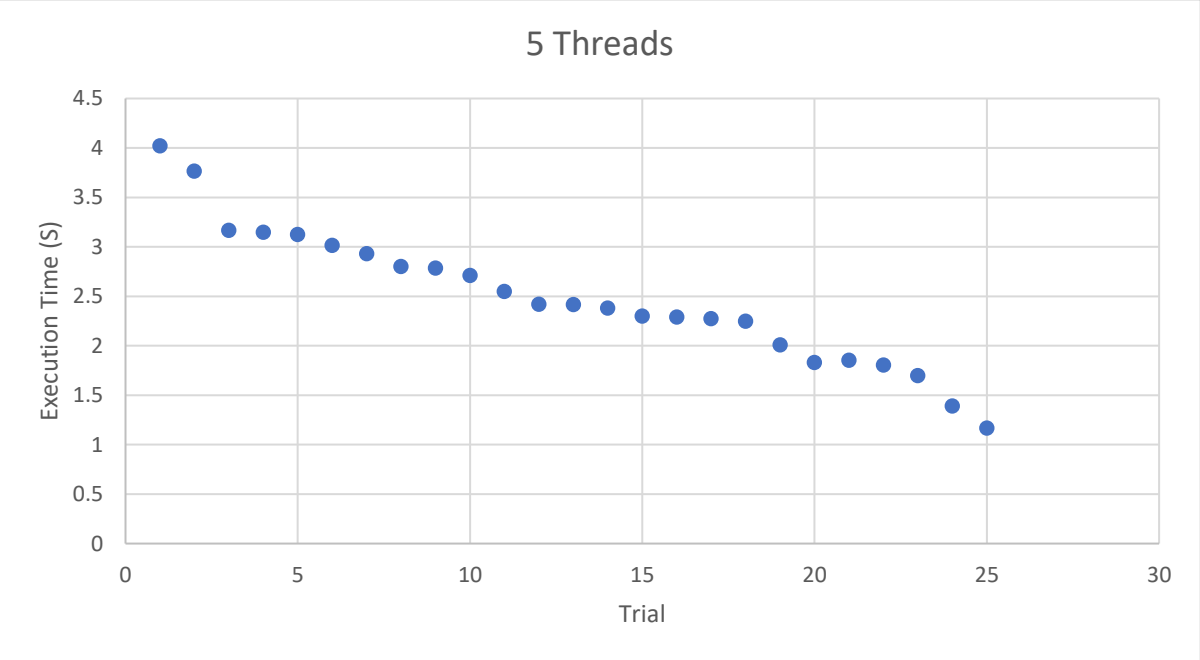


***Figure 6 – Figure 6 shows the parallelization results of the Sudoku solver using 5 threads.***

Overall, using the table and graphs, the execution time didn't drastically, since recursion programs are usually a lot harder to parallelize. However, after every run, there is evidence of a speedup between each run, so parallelization is evident.

**Lab Questions:**

1) **What decomposition method is used?**

   **Answer:** For this type of algorithm, the type/method of decomposition used is task decomposition because the solver receives the task of finding an empty slot, populating with numbers, checking conflicts, and finally, if the there are no conflicts, it moves on to the next square to repeat the process; otherwise, if conflicts are found, it moves back to the previous slot and starts the process anew.

2) **Is there any synchronization concern? Why?**
   **Answer:** There are synchronization concerns regarding the Sudoku solver, to be exact, data synchronization. The program has to ensure that the data is shared appropriately (the empty slots) and it has to make sure the all threads are acting accordingly, for example they are each placing one number at a time and checking for conflicts.

3) **Is the workload balanced? How do you achieve that?**
   **Answer:** The workload is balanced, since all threads are working together to achieve the overall goal of solving the sudoku problem. Some threads will locate an empty space, others will populate it, and the rest will check for conflicts. There will be threads that have an easier task, but in the end, they are all working together to achieve the goal.

4) **What are the performance results of both the serial and parallel implementation? Are they what you expected?**
   **Answer:** The serial implementation is the faster one of the two. This is due because parallelizing a recursive program is not the easiest, and it produces more overhead than if it was just implemented serially. As a result, just because a program is parallelizable, that doesn't mean it has to be.

   The parallel implementation starts out at various times, but generally, there is speedup between the times. The times are also inconsistent, since they never start within the same range, nor do the execution times decreases according to the number of threads, since more threads equals more overhead.

   The results were as expected, since the Midterm Quicksort implementation involved a recursive function, and the comparison between the serial and parallel implementation varied by miniscule milliseconds. Similar to the sudoku solver, sometimes the times were inconsistent.

**Problems Encountered:**

The main problem encountered was the parallelization of the program. The parallelization performed is not as best as it should. The parent-child inheritance was not properly implemented, and as a result, the depth implementation was also not implemented fully. Therefore, the parallelization results are weak at their best.