# AAPP Challenge Report

Giacomo Orsenigo - Francesco Spangaro - Tosetti Luca

21 November 2024

Academic Year 2024 - 2025

## Contents

# 1 Introduction

The aim of this project is to implement a sudoku solver by using a bruteforce approach and compute it in parallel using OpenMP directives.
The problem consists in finding and printing all possible solutions to a given sudoku board.

# 2 Design choices

The design choices made were:

- Adopt a "smart" bruteforce approach that allows us not to try all numbers in each cell, but only the ones that are not already present in the same row, column, or block. We implemented this with a bitmask that allows to check in an efficient and easy way whether a specific value can be inserted or not in a cell. In brief, we created a linearized 3-dimensional array of booleans, so that each cell is associated with a number of booleans equal to the field size of the board. Each boolean array associated to a certain cell is used to check whether the value corresponding to the boolean value's position in the array can be inserted in such cell or not (***True*** if it can be inserted, ***False*** otherwise).

- Adopt a solution in which we compute some initial partial solutions of the sudoku board. These partial solutions are all the different sudoku boards obtainable by filling the first ***CELLS_TO_PERMUTATE*** cells (we decided to set this variable to 7, since with this value we obtained the best results in terms of performance after many configuration attempts) with the permutation of all the possible values that can be inserted in those specific cells. The partial solutions are computed sequentially and then the algorithm can iterate over them and compute in parallel all the different final solutions for such partial solutions.
  After a number of tries, we opted to use a ***taskloop*** directive and set the number of tasks spawnable by OpenMP for such loop to the number of partial solutions divided by 10.

- Solve each of the partially solved solutions by using a recursive approach accessing each cell and checking:

  1. If the cell is already set the algorithm ignores it and continues to the next one;
  2. If the cell is **not** already set, the algorithm iterates over all the values that can be inserted in the sudoku board, and checks if the number of the current iteration is insertable in the cell according to the bitmask. If that's the case then the value is inserted and the algorithm continues to the next cell, otherwise it tries the next value. If none of the values can be inserted in the cell, it means that one or more values inserted in the previously considered cells are wrong and therefore the algorithm returns to the upper recursive calls and tries other values for the previous cells.

  From the parallelization point of view, we opted to spawn a task for each recursive call, but in order to prevent the number of tasks from growing uncontrollably we used a cutoff mechanism: through the ***final*** clause associated to the OpenMP ***task*** directive we made sure that OpenMP wouldn't spawn new tasks for all the columns other than the first one. These tasks will be in fact executed on the same threads previously created.

- When a solution is found (the board is filled), the variable containing the number of solution is atomically incremented. In addition, if the ***PRINT_SOLUTIONS*** flag is true, the algorithm prints the current solution using a OpenMP critical section, so that only one thread at time can print.

# 3 Experimental setup

We built and tested the program in a WSL environment on two different machines:

- A laptop with a 12th Gen Intel(R) 8-core, 12 logical threads, i5-12450H @2.00 GHz CPU
- A laptop with a 10th Gen Intel(R) 4-core, 8 logical threads, i7-1065G7 @1.50 GHz CPU

We tested the program with different configurations w.r.t. OpenMP in order to tune the number of tasks and the cut-off strategy. We have used the ***OMP_NUM_THREADS*** environment variable to test the algorithm using different number of threads (ranging from 1 to 48 at most).
To simplify the tuning process, we created a python script to automatically run the program with the different configurations, parse the running times and plot the results.
All this test were done with the ***PRINT_SOLUTIONS*** flag disabled, since the prints require additional synchronization, thus reducing parallelism.

## 4 Performance measurements

In order to properly measure the performance of the sudoku solver we choose to run the program with different number of threads and different sudoku boards to solve. To compare the different results obtained we decided to plot such measures in some graphs that can be seen below.

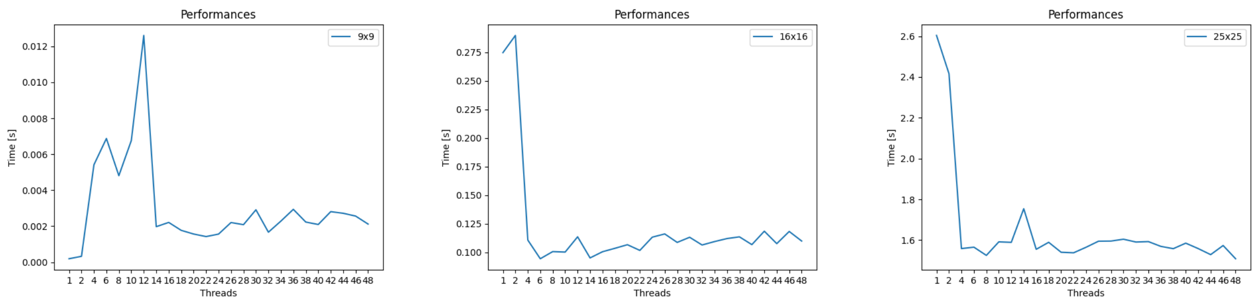

Figure 1: Computation time measured on 12th Gen Intel(R) 8-core, 12 logical threads i5-12450H @2.00 GHz

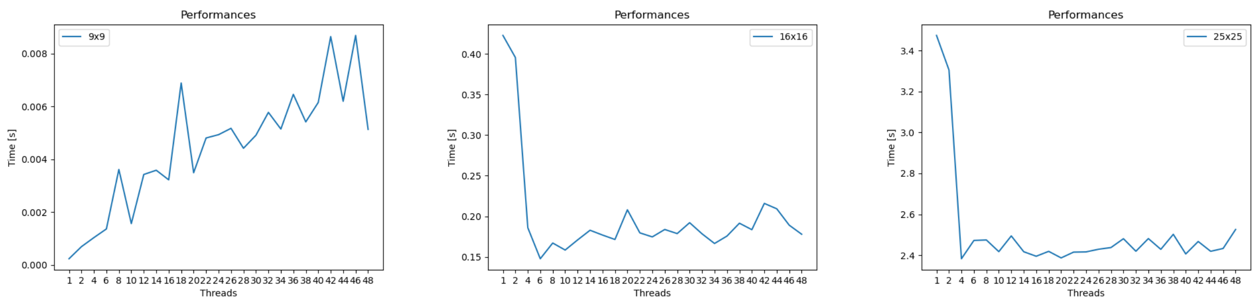

Figure 2: Computation time measured on 10th Gen Intel(R) 4-core, 8 logical threads, i7-1065G7 @1.50 GHz

As can be seen from the graphs, we can conclude that:

- With a 9x9 sudoku board, the performances partially decrease as we increase the number of threads used. This is mainly due to the fact that the test case is rather small w.r.t. the computational effort needed to solve it. For this reason the algorithm creates a lot of tasks that perform very few/fast operations, thus having more threads just cause troubles to OpenMP since it needs to schedule the different threads and tasks, thus causing a decrease in performance.
- With a 16x16 sudoku board, we can really start seeing the parallelization effect since, as we can see from the graph, the computation time is heavily reduced if we are using 4 or more threads. However in this case more threads don't necessarily translate in better performance, since the tasks' computational effort may be split by OpenMP amongst the different threads in a suboptimal way, resulting in performance degradation.
  In particular, even if it's quite difficult to draw exact conclusions, it seems that we have the best performances by using either 14 threads in the first CPU or 6 threads in both CPUs.
- With a 25x25 sudoku board, the differences between processing the solution in a sequential way (a.k.a. with only 1 thread) and in a parallel way is even clearer. In fact, with the latter, we decreased the running time by roughly 1 second in both CPUs. In this case it's quite difficult to understand precisely which is the best number of threads to use, since the noise in the data is quite high, but we can conclude from the graphs above that using 4, 20 or 44 threads are the best choices.

## 5 Conclusions

The experimental results confirm what we expected from theory. With a small problem the overhead added by OpenMP threads is too high to benefit from parallelization. With bigger problems, we can exploit the benefits of parallelization until we fill the cores of our machines. Using more threads than cores has no discernible advantage and the running time remains almost constant.