Sudoku solver challenge

Christian Rossi (10736464) - Kirolos Shroubim (10719510) - Antonio Sulfaro (10742266)

November 21, 2024

1 EXPERIMENTAL SETUP

We conducted our experiments using the provided materials, which included: a set of C++ files where to write the algorithm, a bash script designed to execute the application with OpenMP that changes the number of threads, and Sudoku templates of different sizes. Since we worked on a Windows system, we employed the `mingw` toolchain for compilation and used the bash CLI to manage the build and execution of the application.

2 DESIGN CHOICES

The Sudoku solver was implemented using a brute-force approach based on Depth-First Search (DFS). Each empty cell in the Sudoku grid is filled by recursively attempting all valid numbers, backtracking when a conflict is encountered. This was achieved by creating OpenMP tasks at each recursive level. However, this approach faced a major challenge: the number of tasks could grow exponentially, leading to excessive system resources usages and memory consumption.
To address these challenges, we introduced these strategies:

- *Depth-based cutoff:* tasks are generated until a specific depth level is reached.
- *Serial algorithm:* the last levels of recursion are solved using a serial implementation. This approach significantly reduces memory usage as the serial algorithm modifies the Sudoku board in place, eliminating the need to copy the board at every step.

A combination of the depth-based and serial cutoffs was also implemented to test their combined effectiveness, in two variations: strict (the cutoffs occurs on an deeper level of depth), and very strict (even deeper level). These design choices allowed us to systematically explore the trade-offs between parallelism, memory usage, and computational overhead.

3 Performance measurment

The performance of our Sudoku solver was evaluated on inputs of varying sizes: 9×9, 16×16, and 25×25 grids. All measurements were performed on a machine with the following specifications:

OS	Microsoft Windows 11 Home
CPU	Intel Core i5-1035G1 1.00GHz, 4 Cores, 8 Logical processors
RAM	DDR4 8.0 GB

The cases taken into account are:

1. 9×9 *Sudoku:* this size was trivial for all configurations, with execution times consistently close to zero. Due to its simplicity, it did not allow us to meaningfully observe the effects of parallelization or cutoff strategies.

2. 16×16 *Sudoku:* this input size provided a better benchmark for evaluating performance. We observe a significant difference in the running times of algorithms with and without a serial approach. The configuration without a cutoff shows no improvement when an additional thread is introduced, highlighting the overhead caused by excessive task creation. In contrast, adopting a serial approach at lower levels of recursion drastically reduces running time. Notably, the strict versions outperform the non-strict ones.

3. 25×25 *Sudoku:* this input proved to be highly resource-intensive, especially with the configuration using only depth-limited cutoff. This implementation caused the system to crash, so we were unable to test those scenarios. However, the configurations with a cutoff demonstrated significant improvement as the number of threads increased. Interestingly, the best performance was observed with the non-strict cutoff configuration.

The performance measure of the algorithm in the given machine are as follows:

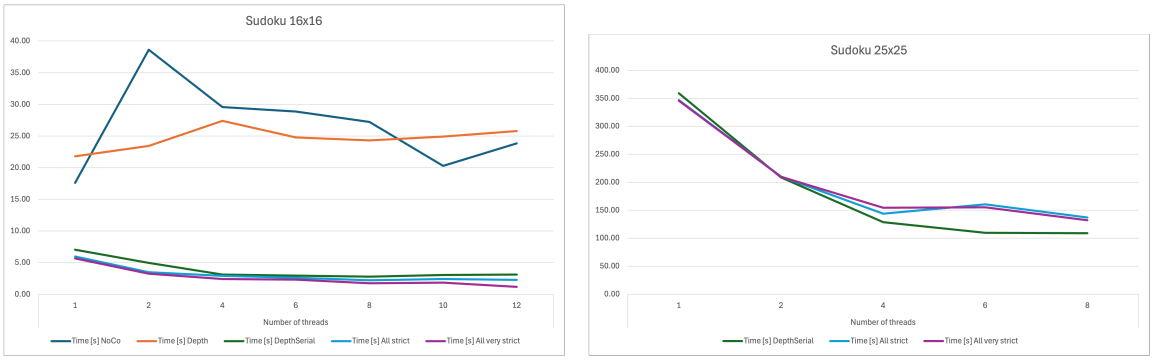

Figure 3.1: Performance on Intel Core i5

The performance of the algorithm varied significantly across different hardware setups. This variability demonstrates that parallel algorithms must be fine-tuned for the specific machine they are run on.