Sudoku solver challenge

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

November 19, 2024

1 SUDOKU PROBLEM

We were required to solve a $n \times n$-dimensional Sudoku using parallel programming on OpenMP.

2 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 and different number of threads.
- Sudoku examples of different size.

Since we worked on a Windows system, we employed the `mingw32` toolchain for compilation and used the bash CLI to manage the build and execution of the application.

3 PERFORMANCE MEASURMENT

The performance of our Sudoku solver was evaluated on inputs of varying sizes: 9×9, 16×16, and 25×25 grids. These inputs allowed us to assess how the algorithm scales with increasing complexity. All measurements were performed on a machine with the following specifications: The cases taken into account are:

OS	Microsoft Windows 11 Pro
CPU	Intel Core i9-10850K 3.60GHz, 10 Cores, 20 Logical processors
RAM	DDR4 16.0 GB

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 can observe:
   - The initial increase in the number of threads resulted in a noticeable speedup, as the workload was effectively split among the threads, however, beyond a certain point, increasing the thread count led to diminishing returns in efficiency. This behaviour highlights the overhead of task creation.
   - Configurations without cutoff strategies or with partial cutoff are slower and consume more memory.
   - The best-performing configuration is the depth with strict serial cutoff. This configuration balances parallel task creation at higher levels with memory-efficient serial computation at lower levels.
3. 25 × 25 *Sudoku:* this input size was extremely resource-intensive, causing our machine to crash across all configurations, including the strict ones. The combination of high memory usage and CPU demand exceeded the capabilities of the system, making this input impractical for evaluation on our hardware.

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

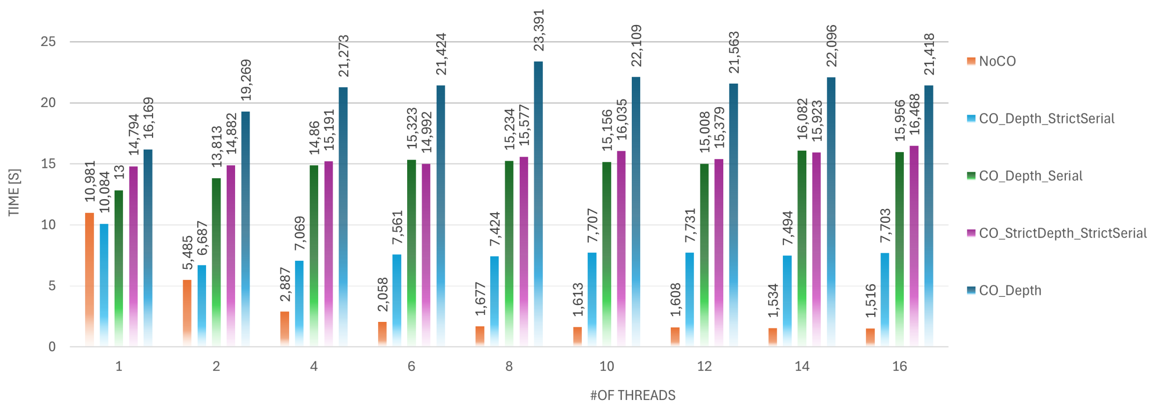

Figure 3.1: Performance on a Intel Core i9-10850K

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.

4 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. To optimize performance and manage the computational demands of the problem, we incorporated OpenMP to parallelize the recursive calls. This was achieved by creating OpenMP tasks at each recursive level. However, this approach faced two major challenges:

1. *Excessive number of tasks:* for large Sudoku puzzles, the number of parallel tasks generated could grow exponentially, overwhelming system resources.

2. *High memory usage:* at each node of the recursion tree, a new Sudoku board had to be created and stored, leading to significant memory consumption.

To address these challenges, we introduced cutoff strategies:

- *Depth-based cutoff:* tasks are generated only at specific levels of recursion, i.e. n levels. This reduces the number of parallel tasks while maintaining the benefits of parallelism, to reduce overheads and memory.

- *Serial algorithm cutoff:* tlast 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.

- *Combination of cutoffs:* a combination of the depth-based and serial cutoffs was also implemented to test their combined effectiveness, in two configurations:
  - *Strict configuration:* the depth-based cutoff occurs more frequently, so the tasks are generated every larger n level, and the serial cutoff is applied at a lower level. This reduces the number of tasks and memory usage more aggressively.
  - *Non-strict configuration.*

These design choices allowed us to systematically explore the trade-offs between parallelism, memory usage, and computational overhead. The different cutoff configurations were evaluated to determine their impact on performance for various Sudoku inputs.