PAR laboratory assignments (alternative)
Recursive exploration with OpenMP: N-Sudoku puzzle

Mario Acosta, Lluc Álvarez (Q2), Eduard Ayguadé (Q2), Rosa M. Badia,
Josep Ramon Herrero, Daniel Jiménez-González, Pedro Martínez-Ferrer, Adrian Munera,
Jordi Tubella and Gladys Utrera

Fall 2022-23

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

Warning and Advice

We strongly recomend you to read lab1 document (at least, *Experimental Setup* and *Understanding the execution of `OpenMP` programs* sections) before starting with the Sudoku lab. We have new boada nodes and new tools to perform the execution analysis of the parallel programs. In the lab1 document you will find information about the new nodes and tools. In addition, you can also find in Atenea some slides with an example of an exploratory recursive problem. Parallel exploratory recursive algorithms require a correct memory management to explore different partial solutions in parallel.

1

Recursive exploration in N-Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle, in which the objective is to fill a $N \times N$ grid with digits so that each column, each row, and each of the N $\sqrt[2]{N} \times \sqrt[2]{N}$ sub-grids that compose the grid contains all of the "digits" from 1 to N. Usually $N = 9$. In this final laboratory assignment you have to parallelize with OpenMP the *N-Sudoku* puzzle, which counts all solutions to an initial incomplete puzzle and reports one solution found.

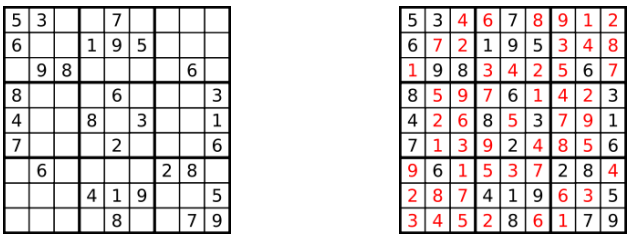

Figure 1.1: Initial incomplete (left) and complete (right) 9x9 Sudoku puzzle

We provide you with the sequential (`sudoku.c` and `sudoku_lib.c` code[1]), the `Makefile` already prepared with all the necessary compilation targets, the scripts to execute and three initial puzzles (`puzzle-small.in`, `puzzle-medium.in` and `puzzle.in`). Puzzle `puzzle-small.in` has to be used to do the analysis with *Tareador*. Puzzle `puzzle-medium.in` has to be used to do the analysis with *Modelfactors*[2]. Puzzle `puzzle.in` can be used to perform the premilinary sequential execution and should be used for the strong scaling analysis.

1. Compile the sequential code and execute it (`./sudoku <puzzle_file>`).
2. Look at the source code and try to understand how the recursive exploration of the possible solutions is done. Which is the main data structure used to store the solution that is being explored? Where is it allocated? How is it initialized? What does function `all_guesses` do?
3. The sequential code is already prepared (using conditional compilation) to analyze with *Tareador* the suggested task decomposition for this problem: each invocation of function `solve` for a cell `loc` for which a value has to be tried. Compile the *Tareador* instrumented version and execute using the `run-tareador.sh` script. For this step, we also use the small puzzle in order to easily visualize the tasks generated. What do you observe in the task graph generated?
4. Which accesses to variables are causing the serialization of all the invocations of the `solve` task? In order to verify your assumptions, you can temporarily filter the analysis for one object (variable) by using the calls to `tareador_disable_object` and `tareador_enable_object`. Filtering the variables you suspect are the causes of serialization, are you increasing the potential parallelism? How will you handle the dependences caused by the accesses to these variables in `OpenMP`?

[1] In `/scratch/nas/1/par0/sessions/sudoku.tar.gz`
[2] *Modelfactors* is an analysis tool that is explained in lab1

2

Implementation of the OpenMP N-Sudoku

Based on the previous analysis of the sequential code, the task graph reported by *Tareador* and your analysis of the data dependences, next you will parallelize the sequential code using `OpenMP`. At this stage we are sure you realised that a recursive task decomposition strategy is the most appropriate for this code. For this section, do not worry about the cost of task creation neither synchronization.

1. We provide you with `sudoku-omp.c` (it is almost the same code than `sudoku.c`). Do all necessary modifications in this code in order to make it amenable for parallelization (understanding the dependences reported by *Tareador*). Insert all necessary `OpenMP` constructs to build your parallel code, making sure that all dependences are honored.

2. In the `Makefile` you have the target to compile your `OpenMP` parallel version. Compile it and generate an executable file. We provide you with scripts to interactively run (`run-omp.sh`) (**Important note:** be careful because the interactive nodes do not have the same architecture than the nodes for execution any more. See laboratory 1), queue (`submit-omp.sh`) and do the strong scalability analysis (`submit-strong-omp.sh`). Each script has a number of arguments that you can see from the script itself. Make sure that your parallel code leads to the correct solution. You can use `puzzle.in` to test the functionality of your `OpenMP` code.

3. Which is the speedup you obtain when doing the strong scalability analysis? Is it the ideal one?

3

Parallelization Analysis of N-Sudoku

You should have observed slowdown on the execution of your parallel program for more than one thread. Analyze the execution of you code with the *Modelfactors* tool. Read (and maybe do) laboratory 1 to know how to run it, have more information and experience with this new tool.

1. We provide you with the script `submit-strong-extrae.sh` that will help you to use *Modelfactors* tool. This script has a number of arguments that you can see from the script itself. You must use puzzle `puzzle-medium.in` for this analysis. This script will generate `modelfactor-tables.pdf` document with three tables summarizing a collection of metrics. This document can be found in the directory that the script creates: `sudoku-omp-puzzle-medium.in-strong-extrae`. This script basically executes several times your code and perform a deep analysis based on the execution traces that are generated in the same directory.

2. Open `modelfactor-tables.pdf` with `xpdf` and look at the *Speedup* row of the first table and the *Overhead per explicit task* rows of the third table. If your implementation has not taken into account the task creation and synchronization cost, the number of tasks and their granularity will force a slowdown from 4 threads. Note that *Overheads* due to synchronization and scheduling may be close to 90% for a certain number of threads.

3. Load trace `sudoku-omp-puzzle-medium.in-strong-extrae/sudoku-omp-12-boada-11-cutter.prv` and analyze it using different user hints and profiles to check what you have seen about the amount of synchronization in the tables of `modelfactor-tables.pdf`.

4. Before continuing, make sure to rename the `sudoku-omp-puzzle-medium.in-strong-extrae` directory since it will may be regenerated with new executions. Do this everytime you work on another version if you need to keep the results for previous ones. However, beware that you can run out of disk quota. Therefore it's advisable to keep the information you regard as useful in another directory and remove all the other files.

5. What is the number of task created? What is the execution time per explicit task for different number of threads threads?

4

Optimization of the OpenMP N-Sudoku

Finally we ask you to improve your OpenMP code taking into account the cost of task creation and synchronization.

1. Modify the parallel code in order to include a *cut–off* mechanism that controls the maximum recursion level for task generation, based on the use of the OpenMP `final` clause and `omp_in_final` intrinsic, to control the number of tasks generated (and their granularity). We recommend that you modify the source code so that a new argument is provided to the main program (`program puzzle -c recursive_level`) to externally provide a value for the recursion level that stops the generation of tasks.

2. Explore different values for the cut–off argument and observe if there is an optimum value for it. We provide you with `submit-cutoff-omp.sh` script to help you with this exploration. The script expects you to have added the new argument (`-c recursive_level`) to the program.

3. Use *Modelfactors* to perform a premilinary analysis of the application with the optimum value of cut–off using puzzles `puzzle-medium.in` and `puzzle.in`.

4. For the optimum value, analyze the scalability.

5. Load the execution trace (`sudoku-omp-12-boada-11-cutter.prv`) generated by *Modelfactors* tool with *Paraver* to make sure the cut–off mechanism works for a particular number of threads.

6. What is the overhead per explicit task for the different number of threads? What is the optimium value of *cut–off*? Compare the execution trace of the new version with *cut–off* and the new *Modelfactors* and scalability results with the first version of your OpenMP code.

5

What to deliver?

Deliver a document that describes the results and conclusions that you have obtained when doing the assignment. Only PDF format will be accepted.

- The document should have an appropriate structure, including, at least, the following sections: Introduction, Parallelisation strategies, Performance evaluation and Conclusions. The document should also include a front cover (assignment title, course, semester, student name, the identifier of the group, date, ...) and, if necessary, include references to other documents and/or sources of information.
- Include in the document and comment, at the appropriate sections, relevant fragments of the C source codes that are necessary to understand the parallelisation strategies and their implementation for *Tareador* instrumentation and for the OpenMP parallelisation strategies.
- You also have to deliver the complete C source codes for *Tareador* instrumentation and all the OpenMP parallelisation strategies that you have done. Include both the PDF and source codes in a single compressed tar file (`GZ` or `ZIP`).

As you know, this course contributes to the **transversal competence "Tercera llengua"**. Deliver your material in English if you want this competence to be evaluated. Please refer to the "Rubrics for the third language competence evaluation" document to know the *R*ubric that will be used.