PAR Laboratory Assignment
Lab 4: Recursive exploration with OpenMP: N-Sudoku puzzle

E. Ayguadé, J. R. Herrero, D. Jiménez, N. Navarro, J. Tubella and G. Utrera

Fall 2013-14

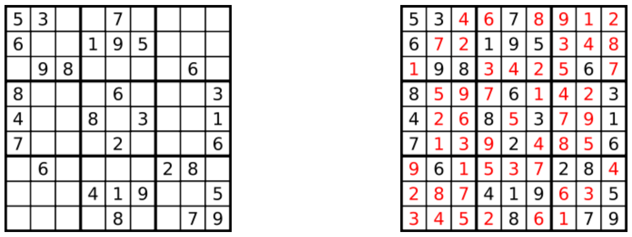
UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Index

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.



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

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

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 two initial puzzles (`puzzle.in` and `puzzle-small.in`).

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. For this we suggest that you compile the sequential code with *Extrae* instrumentation (already included in the source code with conditional compilation, `make sudoku-i`), and execute submitting `submit-i.sh` (which uses the small puzzle in order to easily follow the recursive exploration, `puzzle-small.in`). The user events in the trace will help you to visualize how this recursive traversal is done (we provide a *Paraver* configuration file named `APP_userevents_stacked.cfg` to visualize the events). 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` already used in

[1] In `/scratch/nas/1/par0/sessions/lab4.tar.gz`

previous assignments. 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`?

2

Parallelization of 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` and analyze the performance that is obtained by using *Paraver*. For this section you will use `puzzle.in`.

1. Copy the original sequential code into `sudoku-omp.c` and do all necessary modifications in the sequential code in order to make it amenable for parallelization (understanding the dependences reported by *Tareador*).

2. Parallelize the code with `OpenMP` making sure that the parallelization overheads are not "killing" your potential parallelization gains. We suggest that you use recursion level in order to control the generation of tasks. Compile and and submit `submit-omp.sh` for execution, checking that the reported number of solutions is the same.

3. Make sure that the instrumentation we provided in the sequential version is still placed at the appropriate places in the source code. Compile with *Extrae* enabled, submit `submit-omp-i.sh` and use *Paraver* to analyze the parallel execution and extract the appropriate conclusions about your parallelization (use the same configuration file `APP_userevents_stacked.cfg` that you used before).

4. Modify the `submit-omp.sh` in order to execute with 1, 2, 4, 6, 8, 10 and 12 processors. Plot the execution time and the speed–up that is obtained, with respect to the original sequential code.

5. Analyze the impact of the maximum recursion level used to generate parallel tasks for a given number of processors (e.g. 8). Plot the variation of the speed–up with respect to the maximum recursion level. Which overheads are causing the variation observed?

3

Deliverable

You should deliver a compressed tar file (`GZ` or `ZIP`) with the `PDF` that contains a report for the work done in this session and the parallel `OpenMP` code, including *Extrae* instrumentation (which should be activated when compiling with `-D_EXTRAE_` in the `Makefile`). The report should have an appropriate structure, using sections to cover the different aspects that you want to describe, and make adequate use of tables and figures in order to present the information in the most effective way. In this report you should address, at least, the following issues:

1. Explain how the program implements the recursive exploration and the main data structures that are used.
2. From the analysis with *Tareador*, name the variables and accesses to them that cause the serialization of the tasks in the initial task decomposition that we provide. Include the final task graph that is generated by *Tareador* before and after filtering these variables.
3. Explain any changes done in the original source code in order to make it amenable for parallelization.
4. In order to reduce the overheads introduced by the parallelization, did you needed to control the recursivity level for which tasks are generated? How?
5. The basic architecture of the system you have used for the experimental evaluation and the input files used. Comment about the size of the puzzle, number of solutions found, same solution or not reported by the serial and parallel code, ...
6. Comment about the execution time and speed–up, with respect to the original sequential code, for the parallel execution with up to 12 processors. Reason here about the numbers reported and how did you decide about the most appropriate maximum recursion depth for which tasks are generated.

Generic competence "Tercera llengua". As you know, this course contributes to the generic competence "Tercera llengua", in particular G3.2 "To study using resources written in English. To write a report or a technical document in English. To participate in a technical meeting in English." If you want this competence to be evaluated, write the whole document in english. We suggest that you take a look at the "rubrics" published at FIB:Racó in order to have an idea of what we expect in this report.