PAR Laboratory Assignment
Lab 4: Branch and bound with OpenMP: N-queens puzzle

Ll. Àlvarez and E. Ayguadé, J. R. Herrero, J. Morillo, J. Tubella, G. Utrera

Fall 2018-19

Index

Note:

- All files necessary to do this assignment are available in `/scratch/nas/1/par0/sessions/lab4.tar.gz`. Copy the compressed file from that location to your home directory in `boada.ac.upc.edu` and uncompress it with this command line: `"tar -zxvf lab4.tar.gz"`.

1

Understanding the potential parallelism in Nqueens

Branch and Bound is a technique that is widely used for speeding up backtracking algorithms. The idea is to have a recursive algorithm that tries to build a solution part by part, and when it gets into a dead end, then it has either built a solution or it needs to go back (backtrack) and try picking different values for some of the parts. Checking whether the solution built is a valid solution is done at the deepest level of recursion (i.e. when all parts have been picked out). However, after building only a partial solution the algorithm can decide that there is no need to go any deeper because it is heading into a dead end.

In this laboratory session you will parallelize with OpenMP the Nqueens branch and bound puzzle. The Nqueens puzzle is the problem of placing N chess queens on an $N \times N$ chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. Solutions to the Nqueens puzzle exist for all natural numbers n with the exception of 2 and 3.

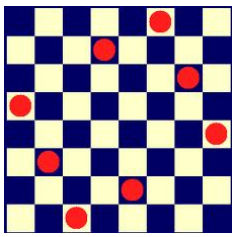

Figure 1.1: Possible solution for the Nqueens puzzle for $N = 8$.

First you should understand how the code `nqueens.c` implements the "branch and bound" strategy, recursively invoking function `nqueens`. You will also analyze, using *T*areador, the potential parallelism that is available and its main characteristics.

1. Compile the sequential version of the program using `"make nqueens-seq"` and execute the binary using `"./nqueens-seq -n12"`. The argument `-n` provides the size of the chessboard. The execution reports the execution time, the total number of solutions found and prints the first solution found.

2. Open the `nqueens.c` source code and try to understand *Tareador* instrumentation that we provide. Observe how tasks are defined (using conditional compilation) and how they are labeled in the task graph for better understanding of how the n-queens algorithm works. Compile the *Tareador* instrumented version of the program using `"make nqueens-tar"` and execute the binary using the `run-tareador.sh` script. This script accepts one argument (the name of the binary to execute) and uses a very small board size (4) to generate a reasonable task graph. Analyze the task graph generated, trying to understand the different tasks that are recursively instantiated (`Trying [...]`) and the tasks that finds a solution (`Solution`). Try to deduce the task ordering and/or data constructs that are needed to enforce the dependences between the tasks that are reported.

2

Shared-memory parallelization

In this second section of the laboratory assignment you will parallelize the original sequential code using OpenMP tasks (either using `task` or `taskloop`) to explore solutions for the possible n positions of a queen in each column. **Before you continue ...** think about the current definition of vector `a` holding the solution; should this vector be shared among the tasks exploring different paths? Just in case you need, you can use functions `alloca(size)` to allocate `size` bytes in memory that is automatically freed and `memcpy(dest, src, size` to copy `size` bytes from from memory area pointed by `src` to memory area pointed by `dest`.

1. Modify the original `nqueens.c` file in order to add all the necessary code (C code and OpenMP constructs) that is necessary to implement the parallel version.

2. Compile your parallel version and execute the binaries generated using either the `run-omp.sh` (interactively) or the `submit-omp.sh` (through execution queues) script in order to check correctness and measure parallel execution time. At this stage we recommend you continue with n=12.

3. In order to avoid excessive task creation overheads, you will need to control the recursion level up to which you create tasks. The `-c<num>` argument in the executable should be used to set this cutoff to the `<num>` value. Modify your parallel code in order to include the cutoff mechanism.

4. Compile again your parallel version and execute the binaries generated using either the `run-omp.sh` (interactively) or the `submit-omp.sh` (through execution queues) script in order to check correctness and measure parallel execution time. At this stage we still recommend you use n=12. Has the cutoff level any visible effect in the execution time?

5. Analyze your parallel code using Extrae/Paraver. Try to obtain a conclusion about the most appropriate value for the cutoff parameter, depending on the number of threads used to execute the program.

6. Once you are sure that the parallelization is correct, as well as you found the appropriate cutoff level, analyze the scalability of your parallelization using the `submit-strong-omp.sh` script, but now with a larger size (n=13) in order to better analyse the scalability.

2.1 Optional

In this optional implementation you should use loop parallelism (`#pragma omp for` to explore solutions for the possible n positions of a queen in each column. Since worksharing constructs cannot be nested, you will have to use of nested parallel regions; as with tasks, you should control the recursion level up to which you spawn new teams of threads in order to avoid excessive overheads.

3

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, students names, 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.
- You also have to deliver the complete C source codes for 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`).

This assignment does not contribute to the evaluation of the **transversal competence "Tercera llengua"**. However if you want to practise your skills then deliver your material in English.