Universitat Pompeu Fabra Barcelona

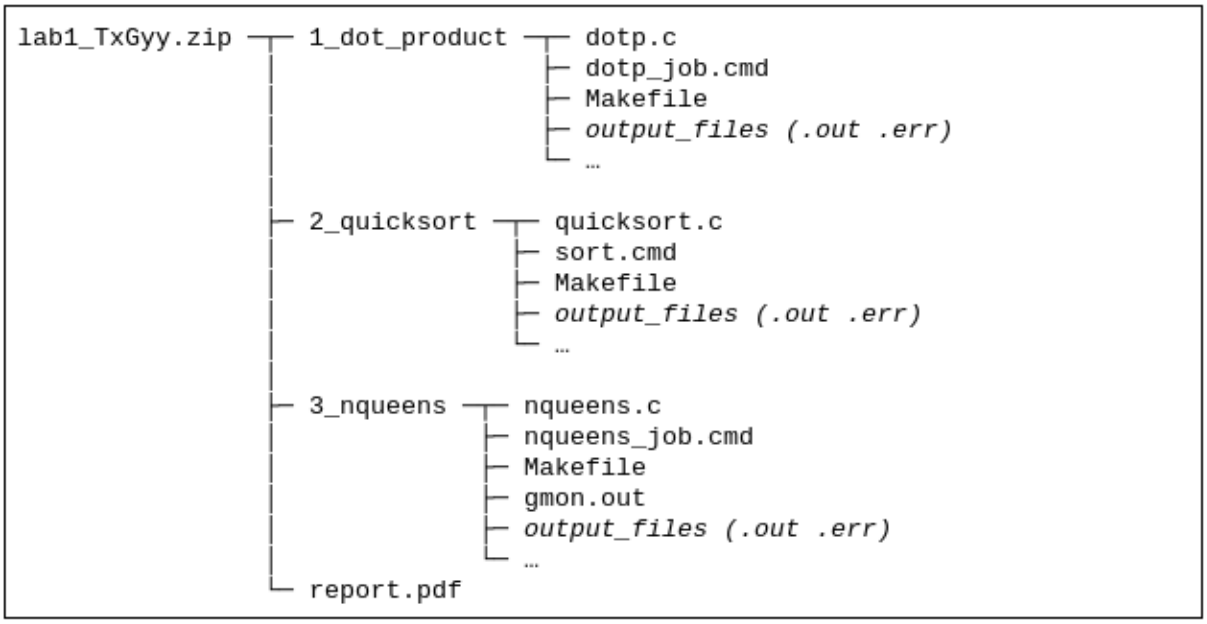
Lab 1

Ricard Borrell Pol[†], **Sergi Laut Turón**[‡]

Instructions

The first Lab session consists of 3 problems with several exercises each. You have to compile your answers to the exercises in a report. This lab assignment has a weight of 15% of the total grade of the subject and the deadline for it is May 3rd at 23:59.

Each group will have to submit a compressed file named lab1_TxGyy.zip, where TxGyy is your group identifier. A .tar or a .tgz file is also accepted (e.g. lab1_T1G1.zip or lab1_T2G21.zip). The compressed file has to contain three folders (1_dot_product, 2_quicksort and 3_nqueens) and all the requested files with the following structure.



A sample file named lab1_TxGyy.zip containing the reference codes has been published in the Aula Global. To ease the process we provide sample job files and Makefiles. You may need to modify the execution lines in the job scripts to perform your tests and the batch lines to request more cores. The Makefiles should not be modified unless the exercise says so. Focus on the code and the work requested for each exercise. In the case the compilation fails, the job will finish and it will not try to run the binary.

For any arising question, please, post it in the Lab class forum in the Aula Global. However, do not post your

[†] ricard.borrell@upf.edu.
[‡] sergi.laut@upf.edu.

code in the Forum. For other questions regarding the assignment that you consider that cannot be posted in the Forum (e.g. personal matters or code), please, contact the responsible of the lab sergi.laut@upf.edu.

Criteria

The codes will be tested and evaluated on the same cluster where you will be working. The maximum grade on each part will only be given to these exercises that solve in the most specific way and that tackle all the functionalities and work requested. All the following criteria will be applied while reviewing your labs in the cluster.

Exercises that will not be evaluated:

- A code that does not compile.
- Code giving wrong results.

Exercises with penalty:

- A code with warnings in the compilation.
- lab1_TxGyy.zip delivered files not structured as described previously.

1. Dot product (50%)

In this exercise we will implement a sequential and two parallel versions of the dot product vector operation. The dot product is an algebraic operation that performs a vector-vector operation as described in the following formula:

$$y = \vec{p} \cdot \vec{q} \tag{1}$$

Where:

- $\vec{p}\,\vec{q}$ are vectors.
- y is a scalar.

Exercise 1 — Code. 20%

1 Write a code in C that implements a dot product with integer vectors. Name it *dotp.c*. The code must accept three integer arguments. The first argument is to decide the execution mode:

- 1: sequential execution (Exercise 1)
- 2: parallel execution (Exercise 2)
- 3: parallel execution with vectorization (Exercise 3)

The second argument must be the number of threads of the execution. For the sequential case this value is useless. The third argument defines the size (SIZE) of the vectors.

Initialize the random seed with the instruction *srand(1);*.

Initialize the vectors with random values between -SIZE/2 and SIZE/2. E. g.:

```
for (int i = 0; i < SIZE; i++){
    a[i] = (int) (rand() % SIZE - SIZE/2);
    b[i] = (int) (rand() % SIZE - SIZE/2);
}
```

The output of the code must be the time spent performing the dot product and the result of the dot product. Print it as follows using your variable names:

```
printf("%.4e\t%i\n", runtime, sum);
```

Note: you may verify the result with small vectors printing them and the solution. Do not include your verification tests in the final code.

Exercise 2 — OpenMP Parallelization. 20%

1 Parallelize the code using OpenMP. The second argument of the code must be the number of threads of the parallel region.

2 Submit a job executing the code with a vector size of 100M integers and obtain results for 1, 2, 4 and 8 threads (you can use the sample slurm script dotp_job.cmd as a base for your job files). How does it scale? Plot the strong speedup.

3 Using the time spent in the dot product function compute throughput in *GBytes/s* and *GFLOP/s* for each configuration (present them in a table).

Exercise 3 — Vectorization. 10%

1 Use vectorization to further parallelize. If the loop is properly vectorized the compilation will output the message: *dotp.c:xx:yy: optimized: loop vectorized using 32 byte vectors*

2 Submit a job executing the code with a vector size of 100M integers and obtain results for 1, 2, 4 and 8 cores with the option of vectorization inside the parallel region. How does it scale? Add a table to your report comparing the values with the simple parallelization.

3 Change the flag *-O2* in the Makefile for *-O3*. Submit a job executing the code with a vector size of 100M integers and obtain results for 1, 2, 4 and 8 cores with the options of Exercise 2 and Exercise 3. Compare the results. What does *-O3* do?

Include your code, job files, outputs and answers to the questions to your report.

2. Sorting with *Quicksort* (40%)

In this exercise we will use tasks to parallelize the *Quicksort* algorithm.

There exist many algorithms to sort values. *Quicksort* is a Divide and Conquer type of algorithm. Essentially, given an array of values with a known size, it picks an element to pivot and divides the array around this element according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Mathematical analysis of *Quicksort* shows that, on average, the algorithm takes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons.

In the Lab 1 files you have a sequential implementation of the *Quicksort* algorithm. The key subroutine in *Quicksort* is *partition()*. The target of *partition()* is to, given an array and an element x of the array as pivot, put x at its correct position in the sorted array, put all elements smaller than x before x, and put all elements greater than x after x. All this should be done in linear time.

Observe the implementation is recursive, meaning that a function calls itself. In every call *partition()* is executed, however, this function is not parallelizable. We need tasks to parallelize the recursive calls to *Quicksort*.

Exercise 4 — Parallelization with tasks. 30%

1 Use tasks to parallelize the recursive calls to *Quicksort*. A parallel region has to be created and a single thread must create tasks.

2 Submit a job executing the code with a vector size of 1M doubles and obtain results for 1, 2, 4, 8 and 16 cores (you can use the sample job file dotp_sort.cmd as a base for your job files). How does it scale? Plot the speedup.

3 Submit a job executing the code with a vector size of 100M doubles and obtain results for 1, 2, 4, 8 and 16 cores (you can use the sample job file dotp_sort.cmd as a base for your job files). Does it scale better or worse than before? Why?

Note: verify the result with small vectors printing them and the solution. Do not include your verification tests in the final code.

Exercise 5 — Questions. 10%

1 Add the clause $if(hi - lo \geq (X))$ to the pragmas of the recursive calls to *Quicksort* and repeat the tests in Exercises 4.2 and 4.3. Try different values for (X) (5, 10 and 1000) and different sizes for the problem. What do you observe? What does it do? Add a table to your report with the different values you have obtained.

Include your code, job files and answers to the questions to your report.

3. N-queens problem with Genetic Algorithm (10%)

In this exercise we will work with a code that solves a generalization of the 8-queens problem using a genetic algorithm.

The 8-queens problem is a puzzle consisting of putting 8 queens in a chessboard in such a way they do not kill each other according to chess rules. The problem can be generalized to the N-queens problem, where we have a chessboard of NxN places and we have to put N queens.

In computer science, a genetic algorithm is a method to find the solutions to optimization and search problems. It is inspired on biological operators such as mutation, crossover and selection. In a genetic algorithm, a population of possible solutions to a problem is randomly built.

The provided code requires three arguments: the size of the problem, the population of possible solutions and the maximum amount of iterations. We will set all values to 100, so we will solve the 100-queens problem.

Exercise 6 — Profiling. 10%

1 Use *gprof* to get a profile of the code. What is the function that takes more time? You may need to modify the Makefile and the job script.

2 Parallelize the most time consuming function and run the code requesting 4 cores. Is the new code faster?

Note: the result should always be found with 73 iterations.

Include your code, job files and answers to the questions to your report.