UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Parallel Computing (CPA)

Laboratory 2: Comparison of Genome Fragments

Year 2018/19

Contents

1 Description of the Problem

In this laboratory exercise, we will work on the parallelisation of a program that determines the best candidates for the donation of organs to a list of patients. More precisely, on the one hand, there is a file (`pacientes.txt`) with fragments of the genome of a group of people who are waiting for organ transplantation. On the other hand, there is another file (`donantes.txt`) with fragments of the genome of a group of people who have declared their willingness to become a donor of an organ. Each line in both files is a relevant genome fragment, represented by means of a text string formed by the characters `A`, `G`, `C`, `T`.

We provide a program that processes two files indicated in the command line (for example the file with the information from the donors and the file with the information of the patients), identifying for each patient the closest donor according to their genome fragment. Therefore, the program generates a list of potential donations. Then, the clinical staff will carefully check the compatibility and feasibility of each candidate pair proposed by the program.

To run the program, the user should provide the names of the input files as arguments, as well as the name of the file where the result of the pairing will be stored, as it can be seen in the example:

```
$ ./simil donantes.txt pacientes.txt salida.txt
```

The program includes a function named `distancia` which computes the similarity of two genome fragments, by means of the *Levenshtein distance*[1] between two strings. The function has a quadratic cost with respect to the size of the strings compared. It is important to state that the length of the genome fragments can be different for different patients and donors.

Exercise 1:

Change the program to compute and show the execution time of the call to the function `busca_mas_parecidos`. Run the program and read the source code to understand how it works, paying special attention to this function.

[1] `https://en.wikipedia.org/wiki/Levenshtein_distance`

2 Parallelisation

For the programs requested in the next exercises, you should add the code needed to show on the screen the number of threads used in the execution of the program and the time spent by the function `busca_mas_parecidos`.

The parallel version should provide the same output as the sequential program for the same input files.

Pay particular attention to the behaviour of the sequential program when several donors have the same similarity for a given patient. The program must return the donor with the lowest index. Take into account that despite this behaviour is obtained in the sequential program without including specific code, the parallel program may require changes in the code.

All the execution times in this laboratory exercise must be obtained in the `kahan` cluster.

Exercise 2:

Implement a parallel version of the code that reads both files, using task level (coarse-grain) parallelism. That is, change the code in a way that, if several threads are used, one thread will load one file at the same time that other thread loads the other file.

Exercise 3: Starting from the code in the previous exercise, parallelise the outer loop in the function that searches for the most similar fragments. Check that it works correctly by comparing the output of the sequential and the parallel versions using the command `cmp` or `diff`, like:

```
$ cmp fich1 fich2
```

If both files are the same, the command will not show any output.

Exercise 4: Starting from the code of exercise 2, parallelise the inner loop of the function that searches for the most similar fragments. Check that it works correctly.

Exercise 5: Obtain the execution time, speed-up and efficiency of both previous parallel versions, running the program with 6 threads. Repeat the execution with different scheduling mechanisms, considering, at least, the following modes:

- Static scheduling without specific *chunk* size.
- Static scheduling with a *chunk* size of 1.
- Dynamic scheduling with a *chunk* size of 1.

Fill in tables and draw graphs with the results, and discuss them. In particular, analyse the influence of the scheduling in the performance, describing which scheduling policies get the best results and which could be the reasons for that. Compare the performance of both parallel versions and discuss if there is a difference on the performance of both approaches, and at which extent this is the expected behaviour, according to the theoretical lessons of the course.

Exercise 6: Produce tables and graphs with the execution time, speed-up and efficiency showing up how the performance varies with respect to the number of threads used. Use the best parallelisation approach and scheduling model according to the results of the previous exercise. In case you have not found a significant difference among the versions or scheduling, choose any of them. To reduce the number of runs, we adivse using powers of 2 for the number of threads (2, 4, 8...). Use the number of threads you consider suitable, according to the hardware architecture of the cluster.

Discuss the results and obtain your conclusions.

Exercise 7: Modify the parallel version of exercise 3 to make the program compute and show on the screen the minimum, average and maximum length of the genomic fragments of the patients (`tabla2`) processed by each thread, as well as the total values.

Bear in mind that in C language you can compute the length of a string `s` using the function `strlen(s)`.

Execution example for 3 threads.

```
Thread 0 of 3: Min/Average/Maximum length: 621/880.7/1106
Thread 2 of 3: Min/Average/Maximum length: 589/865.3/1090
Thread 1 of 3: Min/Average/Maximum length: 615/880.6/1135
TOTAL: Min/Average/Maximum length: 589/875.5/1135
```

3 Delivery

There are **two tasks** in PoliformaT for the delivery of this laboratory exercise:

- In one of the tasks you should upload a file **in PDF format** with the report of the exercise. No other format will be accepted.
- In the other task, you should upload a single compressed file with all the source code of the programs you have developed. **Only the source code, do not pack the executable files** (they are large and unnecessary as they can be obtained from the source code). The file must be in `.tgz` or `.zip` compressed format.

  More precisely, you should include the next source files in the compressed file. Please use the names showed in the following table:

Fichero	**Contenido**
`simil1.c`	Implementation of exercise 1
`simil3.c`	Implementation of exercise 3
`simil4.c`	Implementation of exercise 4
`simil7.c`	Implementation of exercise 7

  If you include any other additional source code file, please indicate clearly in the report the purpose of such a file.

  Double check that all the source code compiles correctly.

Take into account the following recommendations:

- You should produce a descriptive report of the source code used and the results obtained.
- The report should have a reasonable size (neither a couple of pages nor several tens).
- Do not include the whole source code of the programs in the report. You can include the parts of the code you have modified.
- Pay special attention when preparing the report. We expect that the report is well structured, with a description of the implementations. Do not merely include a listing of the results.