PPAR Tutorial 2 **Master 1 in CS**

Open MP

This hands-on session requires a GCC compiler, your favourite code editor to develop with C, and a terminal, or alternatively, your favourite IDE. Your GCC compiler must support OpenMP. To check if this is the case, the `-fopenmp` must be enabled. For instance, in the terminal, invoke:

```
$ gcc -o file file.c -fopenmp
```

If your GCC compiler does not seem to support OpenMP (getting it working on MacOS can be difficult), one alternative is to use the online GCC compiler here:

```
https://www.onlinegdb.com/online_c_compiler
```

Enter `-fopenmp` as an extra compiler flag through the setting menu (Orange gear top right of the menu bar).

1 Data aggregation (revisited)

With the knowledge acquired during the previous hands-on session about what can be done and what can not be done to parallelize a data aggregation procedure, we will now be able to actually parallelize it. Retrieve the solutions written during the last hands-on (you can download the file `aggregation-openmp.c` from the `TP2` folder).

▷ Q1. Function `sumTabSeq()` implements a basic sequential aggregation of the value of an array through addition. Inject a naive parallelization of the loop with `#pragma omp for`. Ensure the number of threads is greater than 1. Observe the result. Does several executions return always the same result? Why ?

▷ Q2. Parallelize functions `SumTabPar()` and `SumTabPar2()` which implements two parallelizable solutions for aggregation. Parallelize them by injecting OpenMP pragmas and OpenMp primitives into the code.

▷ Q3. OpenMP includes a natively parallel construct for aggregation, available using the `reduction` keyword. Write the code of an aggregation using it and implement it in the `SumTabParOpenMp()` function.

2 OpenMP MapReduce

A parallel programming pattern that can be used for a lot of problems is *MapReduce*. When a problem is solved using this model, the data are processed using a composition of two possible processing function. A *Map*, which applies a given function on all the data. Data are typically cut into chunks, each chunk is processed in parallel. A *Reduce*, which merges the results of the Map, typically using an aggregation function. An example of a problem that can be easily solved by the MapReduce model is the *Char Count* problem.

The goal of this exercise is to write a parallel code for counting the total number of characters of an array of words. Your code should work in three phases:

1. Map: for each word, compute the number of occurrences of each letter composing it;
2. Reduce1: Based on the output of the Map phase, compute the total number of occurences of each letter
3. Reduce 2: Based on the output of the Reduce1 phase, compute the total number of characters in the text.

▷ Q4. The file `mapreduce-openmp.c` gives you a partial, sequential version of such a program. Produce a parallel, OpenMP-based program to solve the *Char Count* problem, by extending `mapreduce-openmp.c`.

3 OpenMP traffic monitoring

We now assume we want to process data from a traffic monitoring application. Data are assumed to be an array of `records`. A record is a C structure composed of three fields: `id`, `speed`, `place` (there are 4 possible neighbourhoods, identified with a unique ID between 0 and 3). We will assume that each car appears a single time in the dataset. We first want to compute two things in parallel:

- the array of stopped cars (those with a speed of 0)
- the array of cars in Place 2

Then, in a second step, we want to merge both results to compute the number of stopped cars in Place 2.

▷ Q5. Write the code implementing such a processing. Get inspiration from the `traffic-openmp.c` file.

4 OpenMP philosophers

The dining philosophers problem is a classical example of synchronization needed in concurrent settings. N philosophers sit at a round table with a delicious dish. Forks are placed between each pair of adjacent philosophers. Each philosopher should alternately eat and think, both activities taking a finite (yet possibly random) amount of time.

However, to be able to eat, every philosopher needs to acquire both his/her left and right forks. Moreover, a philosopher can not release a fork as long as he/she has not acquired both forks and eaten. If they are not careful, the philosophers can fall into a deadlock: they may end up each having one fork and waiting the other.

We will consider the solution to this problem proposed by Dijkstra in 1965. An unique ID will be assigned to the forks. Every philosopher will be allowed to take firstly the lower-numbered fork, and then the higher-numbered one, exception made for one philosopher only, which will adopt the inverse behaviour. This general behaviour avoids the situation where all philosophers wait forever for the others to release the forks.

▷ Q6. Write a code simulating Dijkstra's solution to this problem. Each philosopher should be assigned to a different thread, and a fork can be represented by an OpenMP lock.