PPAR Tutorial 1 **Master 1 in CS**

Let's get parallel (or at least try to)

This hands-on session requires a GCC compiler, your favourite code editor to develop with C, and a terminal.

1 Data aggregation

We want to parallelize data aggregation. We will assume that the data to be aggregated is contained in a static array and that the function you will use to aggregate is `SUM`.

The objective of this part is to write code that *could* be easily parallelized. In other words, the goal is to *think* parallel even if you do not have yet means to actually *go* parallel.

The file named `aggregation.c` provides a simple program calculating the sum of the elements contained in an array. The interesting part (the `sumTab()` function) cannot be easily parallelized: it relies on an intrinsically sequential procedure: the iterations of this loop, if done at the same time, could lead (and will most probably) to an incorrect result.

1.1 First parallelizable version: pointer jumping

One way to restructure the code so it is easily parallelizable is to rely on a technique similar to pointer jumping (as presented in the P-RAM lectures): let us assume that we have N processors at our disposal (N being the size of the array). In this case, we can have one processor associated with one element. Then we need $\log N$ steps: at each step $s \in \{0, \log N - 1\}$, every processor i for which $i \mod 2^{s+1} = 0$, in parallel, sums $T[i]$ with $T[i + 2^s]$.

▷ Q1. Make a picture of such an algorithm.

▷ Q2. Write such an algorithm, implement it in the `sumTabPar()` and test it.

▷ Q3. Assuming the loop that can be safely parallelized is actually run in parallel, what is the complexity of your aggregation function?

1.2 Second parallelizable version: partial aggregation

We will now assume, to be a bit more realistic, that we actually have a limited number of processes (or threads), namely n. Recall that N refers to the size of the data array. In this case, another way to parallelize data aggregation is to have n threads each, doing sequential aggregation on a subpart of the array. In other words, in parallel, each thread t_i sums the N/n element starting at $i * N/n$.

Once we are done, in a second step, the partial aggregation results are merged in a sequential manner.

▷ Q4. Make a picture of such an algorithm.

▷ Q5. Write such an algorithm, implement it in the `sumTabPar2()` and test it.

▷ Q6. Assuming the loop that can be safely parallelized is actually run in parallel, what is the complexity of your aggregation function?

1.3 Computing the weight of tree branches

A tree $T = (V, E, w)$ is a graph, with vertex set V and edge set E, such that each of its vertices is connected, through an edge, to one and only one other vertex. Our tree T is weighted, in the sense that weights, i.e. real numbers, are associated to its edges. We are interested in the weight of tree branches, i.e. for each node, the weight of the branch leading to this node from the root. The problem we consider consists in computing the weight of all such branches.

The code for generating a random tree and for printing its topology on the screen is available in the `trees.c` file. In this file, there is a for loop containing no instructions. Fill the for loop with a set of instructions for computing the branch weights. You will again assume that you can actually have access to multiple machines: assume a CREW model, with n processors, here n being the size of the tree.

▷ Q7. Write such an algorithm, implement it in the `sumTabPar2()` and test it.

▷ Q8. Assuming the loop that can be safely parallelized is actually run in parallel, what is the complexity of your function?