

Longest branch of a tree

1 Longest tree branch

This exercise is about parallelizing a Depth First Search (DFS) traversal of a random tree. We assume that each node has a weight which corresponds to the time it takes to process it. Our DFS traversal finds the longest branch in the tree, i.e., the branch such that the sum of the weights of its nodes is maximum.

The tree is nodes of type `node_t` which contain the following members

- `weight`: the weight of the node;
- `branch_weight`: the weight of the branch that connects the node to the root of the tree; this is set to zero at the beginning and is updated during the DFS traversal;
- `id`: the node number;
- `nc`: the number of children of the node.
- `*children`: an array of pointers to the children of the node.

The traversal is done recursively using the following code

```
void longest_branch_seq_rec(node_t *root,
                           unsigned int *longest_branch_weight,
                           unsigned int *longest_branch_leaf){
    int i;
    process(root);
    root->branch_weight += root->weight;

    if(root->nc>0) {
        for(i=0; i<root->nc; i++){
            root->children[i].branch_weight = root->branch_weight;
            longest_branch_seq_rec(root->children+i,
                                   longest_branch_weight,
                                   longest_branch_leaf);
        }
    } else {
        if(root->branch_weight > *longest_branch_weight){
```

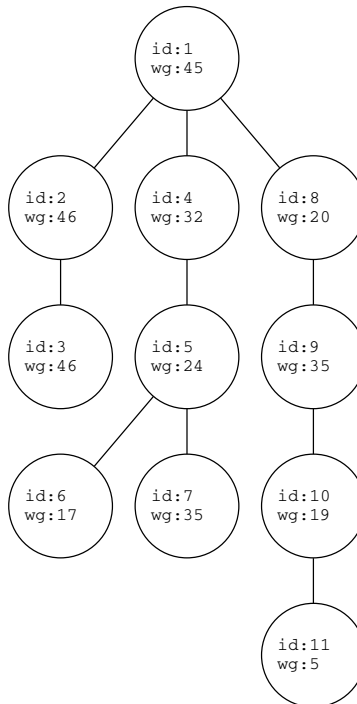
```

    *longest_branch_weight = root->branch_weight;
    *longest_branch_leaf   = root->id;
  }
}
}

```

The `longest_branch_weight` and `longest_branch_leaf` arguments of this function are meant to return the weight of the longest branch and the corresponding leaf. When we visit a node, first we process it using the `process` routine and we update the weight of the branch that connects it to the root (i.e., we add its weight to the branch weight of its father). Then, if it has children, we recursively call this code on each one of them, if not then it means that we have reached a leaf of the tree; in this case if the weight of the current branch is greater than the current maximum contained in the `longest_branch_weight` variable, we update `longest_branch_weight` and `longest_branch_leaf`.

For example, on the tree below, this method would return the leaf number 3 and the associated weight of 137.



2 Package content

In the `tree_branch` directory you will find the following files:



- `main.c`: this file contains the main program which first initializes the tree for a provided number of maximum levels. The main program then calls a sequential routine `longest_branch_seq` containing the above code, then calls the `longest_branch_par` routine which is supposed to contain a parallel version of the traversal code.
- `longest_branch_seq.c`: contains a routine implementing a sequential traversal with the code presented above.
- `longest_branch_par.c` contains a routine implementing a parallel tree traversal. **Only this file has to be modified for this exercise.**
- `aux.c`, `aux.h`: these two files contain auxiliary routines and **must not be modified.**

The code can be compiled with the `make` command: just type `make` inside the `tree_branch` directory; this will generate a `main` program that can be run like this:

```
$ ./main 1 s
```

where `1` is the number of levels in the tree. The argument `s` is the seed for the random number generation (which is used to build the tree), and can be used to create trees of different shapes for a fixed number of levels.

3 Assignment

-  At the beginning, the `longest_branch_par` routine contains an exact copy of the `longest_branch_seq` one. Modify these routine in order to parallelize it. Make sure that the result computed by the three routines (sequential and parallel ones) is consistently (that is, at every execution of the parallel code) the same; a message printed at the end of the execution will tell you whether this is the case. Note that there may be multiple branches of the same length; in this case any of them will be considered a correct result. Also, modify the code in order to count the number of nodes updated by each of the working threads.
-  Report your answers to the two questions below in the `responses.txt` file.
 - In the TP on tree traversal we had to use the OpenMP `taskwait` construct to enforce some precedence constraint. Is this needed in this exercise? Explain your answer.
 - Report the execution times for the implemented parallel version with 1, 2 and 4 threads and for different tree sizes. Analyze and comment on your results: is the achieved speedup reasonable or not?

Advice

- As usual, when developing and debugging choose trees of small size. When evaluating performance it's better to choose a larger tree size.