

Linked list traversal

1 Binary tree traversal

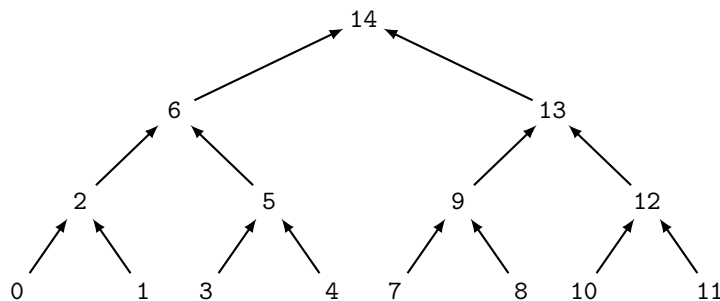
This exercise is about parallelizing the traversal of a postordered binary tree.

The tree is described by two integers and three arrays:

- **l**: the number of levels in the tree.
- **n**: the number of nodes in the tree. Note that $n = 2^l - 1$.
- ***data**: `data[i]` contains an integer data associated with node `i`.
- ***left**: `left[i]` is the index of the left child of node `i`.
- ***right**: `right[i]` is the index of the right child of node `i`.

Nodes are numbered from 0 to $n - 1$ in post-order: this means that all the nodes in a subtree are numbered consecutively. This implies that a node of the tree has a index greater than all its descendants; as a results, visiting the nodes in the natural order 0, 1, 2, ... implies that a node is always visited after its children. Here is an example of binary tree with the corresponding data structure:

```
l      = 4;  
n      = 15;  
left   = [-1, -1, 0, -1, -1, 3, 2, -1, -1, 7, -1, -1, 10, 9, 6];  
right  = [-1, -1, 1, -1, -1, 4, 5, -1, -1, 8, -1, -1, 11, 12, 13];
```



Note that both `left[i]` and `right[i]` are equal to -1 for a leaf node.

The following code is used to traverse the tree and compute, for each node a data, `data[i]` by combining the index of the node `i`, and the data computed on its left and right children:

```
for(i=0; i<n; i++){
    l = left[i];
    r = right[i];
    data[i] = process(data[l], data[r], i);
}
```

2 Package content

In the `tree_traversal` 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 levels. The main program then calls a sequential routine `treetrav_seq` containing the code above, then reinitializes the tree and call the `treetrav_par` routine which is supposed to contain a parallel version of the traversal code.
- `treetrav_seq.c`: contains a routine implementing a sequential traversal with the code presented above.
- `treetrav_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_traversal` directory; this will generate a `main` program that can be run like this:

```
$ ./main 1
```

where `1` is the number of levels in the tree.

3 Assignment

-  At the beginning, the `treetrav_par` routine contains an exact copy of the `treetrav_seq` one. Modify these routine in order to parallelize it using the OpenMP `task` construct. 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.

-  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? Report your answer in the `responses.txt` file.

Advice

- When using the OpenMP `task` construct, always think about data scoping to make sure input data has the correct value upon execution of the task and returned results do not go out of scope when the task is finished.
- Note that, unlike the TP, it is not possible to use the `taskwait` construct to make sure dependencies are respected unless large portions of the code are re-written (which is, obviously, not the objective of the exercise).