

Breadth-first traversal of a graph

1 Breadth-first traversal

This exercise is about parallelizing the traversal of a random graph in a breadth-first order. This graph is supposed to have a single origin node (i.e., only one node is the successor of no other node) but the total number of nodes is not known. The objective of this traversal is to visit each of its nodes **only once**. Upon visiting a node, we must update the node and compute its level, i.e., its distance from the origin node.

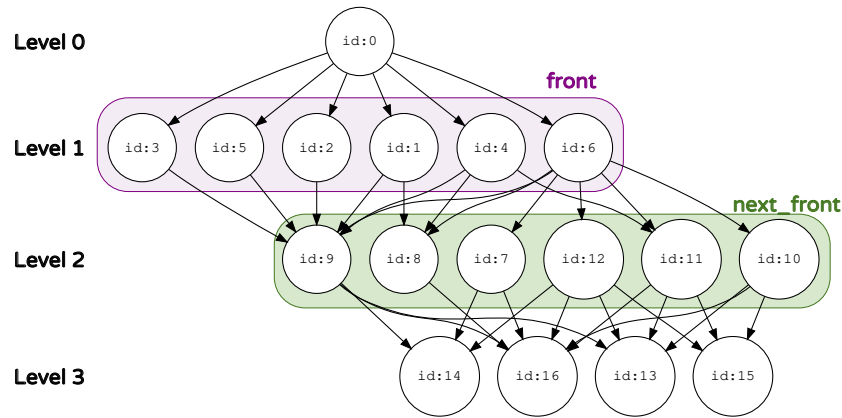
The graph has nodes of type `node_t` which contain the following members

- `unsigned int id`: the node number;
- `unsigned int ns`: the number of successors of the node.
- `node_t **successors`: an array of pointers to the successors of the node.
- `int value`: a value associated to each node. It is modified when the node is updated.
- `int l`: the level of the node, i.e., the distance from the origin

The traversal can be done using the idea of a front which is a set that contains all the nodes of a certain level. Based on this idea, the breadth-first traversal can be implemented with this algorithm:

```
while front is not empty do
  for all nodes in front do
    for all successors of node do
      if successor not already added then
        add successor to next front
        update successor and set its level
      end if
    end for
  end for
  Next front becomes current front
end while
```

This is illustrated in the figure below.



Note that in the provided code we use two arrays

1. **front**: this contains the nodes of the current front. The number of these nodes is **n_front**.
2. **next_front**: this contains the next front and it is filled-up with the successors of all the nodes in **front**. The number of these nodes is **n_next_front**.

At the end of one iteration of the **while** loop, the two arrays are swapped. i.e., **next_front** becomes **front**; instead **front** becomes **next_front** and it is set to be empty, i.e., **n_next_front**=0. In this way, with only two arrays we can traverse the whole graph.

Also note that the **update** function used to update nodes is relatively expensive and, therefore, the objective is to update multiple nodes simultaneously in order to reduce the total execution time.

2 Package content

In the **bfs_traversal** directory you will find the following files:

- **main.c**: this file contains the main program which first initializes the graph for a provided number of nodes. The main program then calls a sequential routine **bfs_seq** containing the above algorithm, then calls the **bfs_par** routine which is supposed to contain a parallel version of the traversal code. **Only this routine must 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 **bfs_traversal** directory; this will generate a **main** program that can be run like this:

```
$ ./main n s
```

where `n` is the maximum number of nodes in the graph. The argument `s` is the seed for the random number generation (which is used to build the graph), and can be used to create graphs of different shapes for a fixed number of nodes. Upon execution, this program will also generate a graph file called `graph.dot`; this graph can be visualized using the `xdot` command:

```
xdot graph.dot
```

3 Assignment

-  At the beginning, the `bfs_par` routine contains an exact copy of the `bfs_seq` one. Modify this routine in order to parallelize it. **The parallelization must be done WITHOUT the OpenMP task feature.** Make sure that the result computed by the two routines (sequential and parallel) 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.

4 Hints

The object of this exercise is to reduce the execution time of the traversal by updating multiple nodes simultaneously. Nevertheless, care must be taken to avoid updating the same node multiple times. You can think of some little modifications of the algorithm above to make this easier to achieve.