Assignment 2

COL380: Introduction to Parallel and Distributed Programming

In this assignment, you will work with Twitter's Who to Follow algorithm. Note the following excerpt from [GGL+13]:

The Twitter graph consists of vertices representing users, connected by directed edges representing the "follow" relationship, i.e., followers of a user receive that user's tweets. A key feature of Twitter is the asymmetric nature of the follow relationship — a user can follow another without reciprocation. This is unlike, for example, friendship on Facebook, which can only be formed with the consent of both vertices (users) and is usually understood as an undirected edge.

WTF: The Who to Follow Service at Twitter

Twitter uses a variant of pagerank algorithm, Random Walk with restart(**RWR**). Its personalised page rank described below to compute a 'circle of trust' for recommending whom to follow.

RWR is a variant of pagerank. In this algorithm, you are given a Directed Graph G and List of nodes L, and the goal is to find "high pagerank" nodes of G that are well connected to nodes in L. **RWR** initiates a new random walk **from each node** v **in** L. At each step, the walk considers the following two possibilities:

1. `Choice1`: Randomly select an outgoing edge from the current node w and 'walk to it' (set it as the next node and add it to the circle if not already in it). If the current node has no outgoing edge, then restart from v.
2. `Choice2`: Choose to return to v with a given probability α (and restart).

In any case the step count is increased by 1. In computing the circle of trust for a user u in the graph, **RWR** can also be used to rank influential nodes. Every time a walk reaches a node w, its influence score with respect to node u is increased by one. Nodes with high scores are marked as more influential for u.

If a user u follows v_1 and v_2, the recommended users in the circle of trust of u are computed by the **RWR** algorithm, starting with the hub, $L = \{v_1, v_2\}$. Given that Twitter graphs are massive graphs (over 20 billion edges), one needs parallel algorithms for doing such computations for every user node in the graph.

You can read more about **RWR** algorithms from this medium blog.

Problem Statement

Given a graph, you have to find out top recommendations for whom to follow using the circle of trust, for all the users in the graph. You need to implement a parallel algorithm using the OpenMPI Interface. You can use any method of communication.

Your code will be provided with(via command line arguments, refer to makefile):

- A graph G in form of `edges.dat`. This is a binary file, which is more convenient for reading the file in parallel. The corresponding `txt` file is also provided for you to match if you're reading the input correctly. Each line represents a directed edge. Each line is of the form `a<single-space>b`, meaning a directed edge from `a` to `b`. In this case it means, `a` follows `b`. The corresponding

representation(bigendian-format) in the binary file is:
$\langle 4 - byte - representing - a \rangle \langle 4 - byte - representing - b \rangle$. Note that since byte size is fixed special separators are not needed.

- The number of nodes n and number of edges m in the graph G.
- *alpha*, the restart probability.
- *num_steps*, the number of steps to take in each random walk. Note that a restart, either by `Choice1` or `Choice2`, is also counted as 1 step, and the step count is incremented. Essentially, *num_steps* tells the number of edges to traverse in each walk, where a restart(either by `Choice1` or `Choice2`) counts as a "pseudo" edge. However, when a walk ends because *num_steps* for that walk has exhausted, then a new walk is initiated, in turn acting like a restart, but that restart doesn't increment the step count, instead the step count is initialized to 0.
- *num_walks*, the number of random walks for each node. For a node u, you will initiate *num_walks* traversals **from each node v** that are followed by u. In each of these traversals, *num_steps* are taken.
- The number of recommendations to give for each user, *num_rec*.
- *seed* for the randomizer. You should evaluate your code with multiple seed and observe the change in results.

Your code should output the node ids of top *num_rec* recommendations for each user in decreasing order of their influence score in `output.dat` binary file. Again, the binary file is convenient for parallel I/O. The rows in `output.dat` are ordered by the node id i.e., `row`(definition below) 0 corresponds to output for node 0, and so on. For each user, dump its outdegree, its recommendations and their corresponding influence score in a new line. If for a given user u, at least *num_rec* number of people can't be recommended then pad the output for recommended node id and influence score with "NULL" string(occupying 4 bytes). Note that the final influence scores are not known until all walks for the user u are completed. The format is described below:

- The recommendation id and the corresponding influence score is single space seperated.
- `row`: For each user first output the outdegree, then recommendation id 1, then influence score 1, then recommendation id 2, then influence score 2 and so on. Note that, since byte size is fixed special separators are not needed.
- Wherever recommendation id and influence score is not available, output `NULL` in string format, consuming 4 byte.
- The outdegree is an int taking 4 bytes, recommendation id takes 4 byte, influence score takes 4 byte, and `NULL` takes 4 byte.
- Say a user u has outdegree x, and number of recommendations are 50, then the corresponding representation in binary file(bigendian-format) would be:
$\langle 4 - byte - representing - x \rangle \langle 4 - byte - representing - r1 \rangle \langle 4 - byte - for - i1 \rangle \cdots \langle 4 - byte - representing - r50 \rangle \langle 4 - byte - for - i50 \rangle$, where $r1, r2, \cdots, r50$ are recommendation ids and $i1, i2, \cdots, i50$ are corresponding influence scores.

Your code will be evaluated based on correctness and the time taken to run. Note that it includes the time taken to read the input and to print the output to a file.

Note that a Randomizer class is provided to help the walk. It is important that you use the randomizer object in the way it has been instructed in the comments of `randomizer.hpp` and `main.cpp`. You have to use only one randomizer object per rank, and across rank each randomizer object should have same seed.

Say, you are running **RWR** for user u, if the walk restarts because u has no outgoing child then the randomizer object is not called, i.e. according to `Choice1`.

Otherwise, the randomizer object will be called to give a random number r[Called only once in a step], which will determine whether to restart($r < 0$), or to go to a next node $w(w = r\%(outdegree\_u)$. Note that, you always pass u while calling the random function, no matter if you're at some random node w in the walk.

Deliverables

- A zip archive with the filename `<entry_num>.zip`. On unzipping it should produce a directory with the name as your `<entry_num>`(all in caps).
- The directory should contain `main.cpp` file and all the other `.cpp` as well as header files that are required to run your code. Do not refrain from this format.
- The directory should also contain a `report.txt` file that contains the runtime(for each of the three given graphs) versus different number of MPI process in {1, 2, 4, 8, 16}.
- Do not include the `librandomizer.a`, `randomizer.hpp`, and `makefile` in your submission. Also, do not include the data given to you.

References

[GGL+13] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 505–514, New York, NY, USA, 2013. Association for Computing Machinery.