Term Project 1

*This description is assignd on April 15, 2019, and it is **due on May 1st**. You need to work on this project individually. I strongly recommend that you do not leave it for the last minute.*

This project should be completed in two steps. In the first step, you should produce a code for calculating PageRank that works correctly and efficiently on a single node/process which is rather simple and straight forward. In the second and main step, you extend your code to a distributed version that runs across multiple processes. You only need to turn in the results for the second step.

Step 1: Calculation of PageRank on a Large Graph

The goal of this part is to develop a code that estimates the value of PageRank for each node in a large graph.
Background: A graph G has a collection of V nodes and E edges. One way to specify a graph is by specifying its edges which is called "edge view". In this representation, the graph is presented as a collection of individual edges where each edge is specified by two connected nodes (x,y). We focus on undirected graphs, this means that there is no notion of direction associated with individual edges (i.e. order of two nodes that represent an edge is not important). In an undirected graph, the degree of node x (or Degree(x)) indicates the number of other nodes that are connected to node x (or the number of x's neighbors). Given an edge representation of graph G, we can determine the degree of node x simply by counting the number of unique edges that have node x on one end.
You can obtain an edge view snapshot of Flickr (the popular photo sharing social network) graph at the following location on the CS department server (ix-dev): "*/cs/classes/www/16S/cis630/fl_undir.tab.txt*". Each line presents (tab separated) IDs of two connected nodes that indicate an edge. You should verify that this graph has 1,624,992 nodes, and 15,476,836 edges. Note that Node IDs may not be consecutive numbers.

Input: The input of your program is an edge representation of a graph (similar to the Flickr example). Keep in mind that the graph could be very large and have hundreds of millions of nodes and possibly billions of edges.

PageRank Algorithm: To calculate the PageRank of individual nodes, you need to write a code that operates in rounds. The state of the code in each round *t* is captured by the "credit" value associated to that node at the end of round t. At the initial state (t=0), each node i has credit of 1, c(0,i) = 1 for any i. In each round t, the current credit of node i is evenly divided among all its neighbors and this value is sent to individual neighbors. This means that in each round, each node receives a some amount of credit from all of its neighbors. The following equation precisely defines the value of credit for node i at the end of round (t+1) based on the credit of i's neighbors in round t where Neighbor(i) denotes the set of all neighbors for node i.

$$C(t+1, i) = \sum_{j \in Neighbor(i)} \frac{c(t, j)}{Degree(j)}$$

For example, consider node i that has four neighbors with the following credit values at the end of round t: 0.1, 0.03, 0.15, 0.14 and node degree of 5, 27, 19, 8, respectively. The credit for node i at the end of round t+1 would be c(t+1,i)=(0.1/5)+(0.03/27)+(0.15/19)+(0.14/8). As the number of rounds (t) increases, the credit of node i converges to its PageRank value. Note that the total credit across all nodes in each step should be equal to the initial aggregate credit which is V, or the total number of nodes in the graph. Be careful to not lose or add credit as a result of your computation (i.e. rounding credit values).

Issues to consider as you prepare and debug your code:

- You should not use a database for storing and managing the edges. Instead you should properly maintain per node information in the memory to efficiently access/update the credit for each node in each round.
- The provided input file may not have an ordered list of node IDs.
- We test your code against larger graph such as a snapshot of Orkut, one the first online social networks. An edge view snapshot of Orkut and the output of the correct PageRank algorithm for each node are provided in separate file on the CIS server (ix-dev)
    - An undirected snapshot of Orkut graph is available at */cs/classes/www/16S/cis630/PROJ/or_undir.tab.txt*
    - An output file for the PageRank algorithm on Orkut graph is available at */cs/classes/www/16S/cis630/PROJ/or_out.txt*
      Each line of the file shows the PageRank values for each node across different rounds.

Your program should accepts the edge view of a graph and the number of rounds as input and produce an output file. You can assume that the input file has a format exactly similar to the provided sample input files. Your program should write the credit values for all nodes in each round in the out file (using the above format), and then gracefully terminate after completing the specified number of rounds. Furthermore, your program should print the following information on the screen:

- The time to read the input file, "time to read input file = 2.3sec".
- The time for completing each round, "time for round 1 = 1.1sec".
- The time to write the output file, "time to write the output file = 4.4sec".

Make sure that your code compiles and runs on ix-dev, produces correct output, and exhibits reasonable time for reading input, writing output and each round of the algorithm. The correct credit values for individual nodes of Flickr graph in round K can be found in file fl_pageRank2_wlK.txt on ix-dev at */cs/classes/www/16S/cis630/PROJ/*.

Step II: Distributed Calculation of PageRank on a Large Graph

In step I, you developed the centralized version of the code to calculate node PageRank that runs as a single process. In this step, we expand the code to run in a distributed manner, i.e. across multiple processes that are running on a single or multiple computers. The distributed version of the code would be able to handle larger graph and/or achieve better performance.

In the distributed version of the code, we need to partition the graph into P subgraphs (or partitions) and each process will be running the code (i.e. calculating the credit for the corresponding nodes in each round) for its assigned partition. Each partition consists of a collection of connected nodes. The edges that are between two nodes of a partition, are called *internal edges*, and the edges that are between nodes from different partitions are called *external edges* of each partition. Similarly, for each node, all neighbors that are in the same (or different) partition are called *internal (or external) neighbors* of the node.

In this distributed setting, the calculation of the credit in each round is exactly the same as step 1. Given node n in process (or partition) P, the value of credit for n's internal neighbors are easily available in process P. However, the credit for n's external neighbors should be obtained from other processes that manage the corresponding partitions. For example, consider node n1 in partition P1 and has three neighbors of x, y and z that are part of partition (process) P1, P2 and P3, respectively. To calculate the credit for node n1 in round r+1, process P1 can obtain the credit of node x (in round r) from local information but should obtain the credit for node y (in round r) from process P2 and credit for node z (in round r) from process P3.

Therefore, there are two related types of interactions between participating processes as follows:

- **Obtaining Credit of an External Neighbor:** To calculate the credit for node n that is part of the partition/process P in round r+1, for each external neighbor m, (i) process P identifies the process Q that manages the partition where m is located, (ii) P sends a request to Q to obtain credit of node m in round r, (iii) Q responds to that request by providing the credit.
- **Coordinating Rounds:** Given the dependencies due to external edges, all processes should start a new round of algorithm simultaneously to ensure that we have correct information for each node, i.e. start of a new round on different processes should be synchronized. Once a process completes round r, it sends a separate RoundComplete(r) message to all other processes. When a process receives the RoundComplete message from all other processes, it checks whether the specified number of rounds is completed. If more rounds need to be executed, the process starts a new round. In this synchronization approach, all processes are similar (they are peers).
  Note that it is important that all processes get synchronized even for the very first round of calculation.

Input: The input of each process is a single partition which includes a connected and mutually exclusive subset of all nodes in the graph as well as their associated edges. The internal edges are between two nodes in the same partition. Each external edge is represented with an internal node in the partition and its external neighbor along with its corresponding partition. Each process also needs to know the number of partitions, their assignment to different processes, and the address of individual processes.
You can find the following input files on ix-dev at */cs/classes/www/16S/cis630/PROJ/*:

- fl_compact.tab provides the edge representation of the entire Flickr graph as you have seen in part I (i.e. the same graph as the one in fl_undir.tab.txt).
- fl_compact_part.2 and fl_compact_part.4 present the partitioning of all nodes in the Flickr graph into 2 and 4 partitions, respectively. Each one of these files has a separate line for each node that provides the following information : *1) Node ID, 2) Node Degree, 3) Node Partition ID* . Note that partition ID starts from 0.

To run your code with n processes, you should provide two files, namely fl_compact.tab and fl_compact_part.n, to each process. Each process should use this information to label individual nodes by their corresponding partitions. This in turn shows whether an edge is internal or external which is essential for managing the update of node credits correctly.

Using MPI for inter-process Communications: You should use MPI in order to exchange the required messages between participating processes. Two good tutorials for MPI programming are available at tutorial 1, and tutorial 2. A common version of MPI (called mpich) that you should use for your project, is already installed on ix-dev. You can see the documentation and information at http://www.mpich.org. Let me know if you have any problem with this MPI package. To facilitate the process of incorporating mpich in your code, a sample c++ code that uses mpich to send and receive messages is posted here In order to compile the code on ix-dev you have to enter the make command. To run the code, you have to issue the "make run" command. Pls send any question to the canvas email list for this class so other students can share their thoughts.

Output: You program should create the output files on the local directory (with the specified name), print a set of information on the screen (see below), write the credit values for all (local) nodes in each round in a separate file (using the above format), and then gracefully terminate after completing the specified number of rounds. All instances of your program should also print the following set of information on the screen using the provided format:

- The time for each instance of your program to read the input files, "time to read input files, partition y = 35sec".
- The time for each instance of your program to complete a round (including updating the output file), " --- time for round x, partition y = 23sec".
- Total time for completion of a round across all partitions, "total time for round x: 56sec"

Note 1: the total time for each round is the time between start of the round by all instances of your program and the time when the last instance of your program completes that round.
Note 2: the exec time for each round on each partition is printed by the corresponding instance of your program. The lines for the exec time of each round of different partitions may appear on the screen in any order (e.g. partition 3 may print its time before partition 1)."

Error Checking: It is essential to implement proper error checking in order to debug your program. For example, a request for obtaining credit of a node from earlier round

is a clear indication of error. Each process dumps the calculated credit for all nodes in its partition in a separate file, named PartitionId.out, with the following format:
NodeID NodeDegree NodeCreditRound0 NodeCreditRound1 NodeCreditRound2 ...

Turning in your Assignment: Before you submit your program, please ensure that we can pass the parameter to your program as follows:
% prog GraphFile Partitionfile rounds Partitions
where "prog" is the name of your executable, "GraphFile" is the edge representation of the entire (original) graph (which is fl_compact.tab in this case), "Partitionfile" is a file that presents the partitioning of graph nodes into n partitions (which is fl_compact_part.n in this case), "rounds" is the number of rounds to execute, and "Partitions" specifies the total number of partitions. We will compile and test your programs on the cs department server (ix-dev). You should make sure that your code properly compiles and runs on ix-dev before submitting it.

Make sure that your program follow the exact input format that we specified here in order to avoid any problem during its evaluation.

To submit your project, you should email me at reza@cs.uoregon.edu the following THREE files as attachments:

- the source code of your program,
- completed readme.txt file, and
- a Makefile to compile your code

.

The subject line of your email should be "CIS630: Term Project I".

Grading: The grading of your program is based on the following elements:

- the code compiles on ix: 10%
- the code runs on ix: 10%
  - provides the specified output (on screen and output file) with the proper format
  - does not use excessive amount of memory
  - gracefully terminates
- the code produces the correct results: 40%
  - Multiple processes sync correctly in each round.
  - Credits for all nodes are calculated correctly (up to the first few decimal values) across multiple rounds.
- the code robustly runs over 2 and 4 processes and terminates: 20%
- the code exhibits reasonable execution times: 20%

The assessment of the last 20% of your total grade on executation time, would be based on the Flickr input file, over 4 processes/partitions on ix-dev.cs.uoregon.edu. For this setting, a reasonable total execution time is less than 15 second which includes 1) reading the input files, 2) 5 rounds of calculation, and writing the result of each round to the output file of each process. Longer execution time will receive proportionally less points as follows:

Exec time - points
< 15s => 20
15-20s => 15
20-25s => 10
25-30s => 5
30-35s => 2
40-45s => 1