Distributed System
CSCI B534, Fall 2016

Project 2

Name: Mridul Birla (mridul), Kai Zhen (zhenk)

1 Variable partition

Since we adopt outbound approach, for a line of input like "1 4 5", we calculate the contribution from 1 to node 4 and 5 respectively and then aggregate the results.

Concretely, we use localPageRankValues to record the increment of pagerank value within each chunk and if the chunk has dangling node we use danglingContrib to store the dangling node contribution. Then, we aggregate the value from localPageRankValues to globalPageRankValue and dangling node contribution to tempRecv. For example, given the input as follows, we have two partitions. For the first partition, we use outbound algorithm to calculate the incremental value for 4 and 5, recorded in localPageRankValues of this partition; for the second partition, we store the incremental value separately in localPageRankValues of partition2. Let's say we only have two partitions, then we can aggregate two localPageRankValue data structures and update the globalPageRankValue in process with rank 0.

```
 Nodes ID     | Ajacent nodes
------------------------------
      1       |  4 5
      2       |  4           <partition1>
      3       |  4
------------------------------
      4       |  5
      5       |  4           <partition2>
```

Once we've updated the globalPageRankValue, we can use Eq. 1. And then we brodcast it to all the processes.

$$PR(u) = \frac{1-d}{N} + d * globalValue \tag{1}$$

To sum up the difference between sequential version and parallel version in terms of data flow, we list two versions as follows.

```
 Sequential PageRank                              | Parallel Pagerank
-------------------------------------------------------------------------------------------------
Read the file and initialize adjMatrix            |Read the file and initialize adjMatrix
Initialize the PageRankValue Hashmap              |Initialize PageRankValue; partition variables
For each iteration:                               |For each iteration:
  Initialize temprankValues with PageRankValue|    MPI: Initialize chunkPageRankValueIncrem
  For each line of input:                         |    MPI:For each line within that partition/chunk:
    Calculate the incremental value               |      Calculate the incremental value
  Updade the PageRankValue with the equation      |    MPI: Aggregate results into globalPageRankValue
Write the output file.                            |Calculate PageRankValue with the equation
-------------------------------------------------------------------------------------------------
```

2 How to Run and Results.

To execute the code, type following commands.

```
mpjrun.sh -np 4 MPIPageRank input.txt output.txt 10 .85
```

The result which looks like following is saved in a file with the name of arg2.

```
0 :-  0.01634534321875001
1 :-  0.38543613337511734
2 :-  0.3308768754288676
3 :-  0.030798070455108458
4 :-  0.09720673122275134
5 :-  0.030798070455108458
6 :-  0.021707755168859418
7 :-  0.021707755168859418
8 :-  0.021707755168859418
9 :-  0.021707755168859418
10 :-  0.021707755168859418
```

3 Data Structures and Data Flow.

We use two hash maps to store the adjacent nodes, i.e. adjMatrix . The "adjMatrix" hash map is pretty dummy as it basically copies the data from the input file. For example, for the input data,

```
   Nodes ID   | Ajacent nodes
------------------------------
      0       |
      1       |  2
      2       |  1
      3       |  0 1
      4       |  1 3 5
      5       |  1 4
      6       |  1 4
      7       |  1 4
      8       |  1 4
      9       |  4
      10      |  4
```

With these hash maps, interpreting the algorithm is be quite straightforward. For each of the popular nodes, we sum the value over all its contributors, as illustrated by the Eq. 2.

$$PR(u) = \frac{1-d}{N} + d * \sum v \in Set \frac{PR(v)}{L(v)} \tag{2}$$

```
for (int i = 0; i < iterations; i++) {
            for (int j = 0; j < numofLine; j++)
                localPageRankValues[j] = 0.0;
            double danglingContrib = 0.0;
            Iterator it = adjMatrix.entrySet().iterator();
            while (it.hasNext()) {
                Map.Entry<Integer, List> pair = (Map.Entry) it.next();

                //If it is a dangling node,
                if (pair.getValue() == null)
                    danglingContrib += globalPageRankValue[pair.getKey()]/numofLine;
                else{
                    int current_size = pair.getValue().size();
                    Iterator iter = pair.getValue().iterator();
```

```
                // For each outbound link for a node
                while (iter.hasNext()) {
                    int node = Integer.parseInt(iter.next().toString());
                    double temp = localPageRankValues[node];
                    temp += globalPageRankValue[pair.getKey()] / current_size;
                    localPageRankValues[node] = temp;
            }}}
            double tempSend[] = new double[1];
            double tempRecv[] = new double[1];
            tempSend[0] = danglingContrib;
            MPI.COMM_WORLD.Allreduce(tempSend,0,tempRecv,0,1,MPI.DOUBLE,MPI.SUM);
            MPI.COMM_WORLD.Allreduce(localPageRankValues,0,globalPageRankValue,0,numofLine,MPI.DOUBLE
            if(rank==0){
                for(int k=0;k<numofLine;k++) {
                    globalPageRankValue[k] += tempRecv[0];
                    globalPageRankValue[k] = df * globalPageRankValue[k] + (1 - df) * (1.0 / (double)
                }
            }
            MPI.COMM_WORLD.Bcast(globalPageRankValue, 0, numofLine, MPI.DOUBLE, 0);
        }
```

Acknowledgement