CS 5300 Project 2

Jiabin Dong, Eric Tan, Ray Weng

Jb836, eyt4, rw422

README

To run our code locally, we have included make files that both builds/compiles our java source code, and then runs the resulting JAR file in HADOOP. Our makefile assumes that there exists a source directory, an input directory, and an output directory for the files. The makefile also assumes that the HADOOP version is 2.6.0 and it is installed in the /opt/ directory. The same JAR file built here can also be deployed to Amazon's EMR framework, given that there is also an S3 bucket set up for src, input, and output directories.

Our block page rank implementation defaults to using a Jacobi reducer; one can slightly alter the source code in BlockedPageRank.java to use the GaussReducer class instead of the BlockedReducer class by changing the reducer class. As a result, you will need to build GaussReducer in the makefile.

Our output of residual error at each iteration as well as the final selected nodes' pagerank is printed to terminal. One can simply add a pipe to log.txt in the makefile to save the iterations' residual errors and pageranks to file.

Simple Page Rank

The input value is formatted as "nodeID PageRank {outgoing0, outgoing1, ...}". The output of the mapper is a NodeOrDouble, which can store nodes or pagerank values (double). The mapper reads in a node, and emits that node, and the destination node of all outgoing pagerank. The reducer (which is node by node in simple page rank), receives a node and all incoming page rank, so it sums that all up as the new page rank value of this node and just emits a single node in the above text format. We use HADOOP counters to track the residual error from each reducer node (after scaling).

Block Page Rank

The input value is, for all nodes u with edges u-v, we have a text file of nodes: "<nodeID(u), pr(u), {nodeID(v)}>". The output is the same as simple page rank, using a placeholder class NodeOrEdge so that the mapper can emit both the nodes as well as the edges that carry page rank. Our mapper reads in that text format, and emits a node for every node, with the block id of that node as the key. For each node *u it* reads, it also observes all the edges attached to it. The mapper has this information because it is part of the input format. Then it also emits an edge with the page rank of the original node *u* divided by the outdegree of *u*. The key for these emits are the block ids of the destination node *v* of that edge *u-v*.

In both versions of the reducer, a reducer is equivalent to one block. It receives all the nodes in a block, and all the relevant edges, iterating through all of them. As it iterates, it builds a hashmap nodeTable of all the nodes in the block. For all the received edges, it builds a hashmap srcNodeTable that collects all the pagerank incoming for a specific node (based on the edges generated in the mapper), or a hashmap boundaryConditions if the edge source is not within this block. Once it does this, then it runs the function iterateBlockOnce().

This is where the implementation differs because we implemented both the Jacobi version and the Gauss-Seidel version for the reducer. The Jacobi version is basically what is given in the original project write up. First, we create a new (temporary) hashmap newPageRank . For all the nodes in our block, (equivalent to NPR in the writeup) we just add up all the page ranks incident to a given node from the hashmap srcNodeTable (or the boundaryConditions hashmap if the node has an incoming edge), ultimately adding the random surfer probability (and applying damping factors) and calculating the residual error for this pass of iterateBlockOnce().

For the Gaussian implementation, we still construct the temporary hashmap newPageRank. We still iterate over all nodes *v* in the block. But for all the edges *u*-*v*, if we already calculated the pagerank of *u* during the iteration, we use the new pagerank of *u* instead of the old one from srcNodeTable. This gives us a more updated value to determine pagerank with, allowing us to converge faster.

Finally (both Jacobi and Gaussian implementations are the same here on after), we emit all the nodes in the text format described earlier so that the mapper can read it in the next superstep. Additionally, we also calculate the block residual error and add it to the HADOOP counter to determine the termination condition for this mapreduce. After it converges, we run a final mapreduce job that reads the output of the pagerank mapreduce jobs to extract the final page rank values for the selected nodes.

Deployment on EMR

To deploy on EMR, we have to modify our input arguments a little bit, i.e. while running the code, the first args[0] represents the input directory, args[1] stands for the output. We also get rid of all the write to file code and use println directly, because EMR always gives us “cannot find file” error while writing to file. We guess EMR asks us to create a blank output.txt first in S3 if we would like to write file, but this seems to be unrelated to the expectation of our implementation. So we skip this step.

While running the code on EMR, we need to upload our input file to S3 and create a blank output folder first. Create a cluster under EMR use a valid key-pair. Then, upload our “jar” file to the cluster:

```
scp - i key.pem BlockedPageRank.jar hadoop@XXXXX.compute.amazonaws.com:BlockedPageRank.jar
```

Login the cluster:

```
ssh hadoop@XXXXX.compute.amazonaws.com - i key.pem
```

Run the code:

```
bin/hadoop jar BlockedPageRank.jar BlockedPageRank s3://<bucket>/input/SimpleInput.txt
s3://<bucket>/output
```

NOTE

1) Parameters for filtering edges:

Netid: jd836

rejectMin: 0.5742

rejectLimit: 0.5842

Number of edges selected: 7524817

2) Our results are under output folder

3) As shown in blocked_residual.txt and gauss_blocked_residual.txt, compared to Jacobi reducer, Gauss reducer takes less average inner iterations to converge, especially in the first two outer passes, though they take same number of outer passes to converge overall. Thus, Gauss reducer achieves higher performance than Jacobi reducer.

CHANGES

After the first submission and project presentation with Professor Demers, we made fundamental changes to our architecture. The simple page rank was working when we presented, so no changes were to it.

As for the blocked page rage, we decided to alter the entire architecture. Rather than only using edges as data structures to carry information between mapreduce phases, we changed to a NodeOrEdge Writable object, to allow transmitting information via both nodes and edges. This made the implementation more intuitive, as it followed a variant architecture of the simple page rank. It also allowed us to more clearly implement the page rank pseudocode given in the write up. Besides this major architecture revamp, the professor also gave us guidance on the distinction between the block residual for determining termination at the mapreduce super step level versus the local residual for terminating the loop over iterateBlockOnce(). For the most part, we made substantial changes after the presentation that enabled us to make a functional blocked page reduce mapreduce implementation.