CMPUT 681 – Final Project

Saeed Sarabchi
Dept. of *Computing Science, University of Alberta*
sarabchi@ualberta.ca

1. Introduction

The selected paper's title for this article is "*An Experimental Comparison of Iterative MapReduce Frameworks*"[1], and the scope of this report, as it was emailed earlier, is to experiment on the **Page-Rank algorithm** execution based on "**LiveJournal**" Graph Dataset on two Mapreduce frameworks, namely Apache **Hadoop** and Apache **Spark**, reproduce the related results and compare them with that of the original paper.

The Page-Rank code, both for Hadoop and Spark, was downloaded from **github**[2] and the **input** Graph Dataset was available on **Stanford large network dataset collection**[3].

The structure of this report is similar to that of previous course projects, in this sense that the source codes are explained in implementation section, and the results of the executions are detailed in the experimentation section.

2. Implementation

In this section, the implementation of Page-Rank algorithm for both Hadoop and Spark frameworks are investigated. Both codes are written in **Java** and compiled on **Ubuntu 16.04**.

The goal of the Page-Rank algorithm is to calculate a specific rank for each of the URLs (in this case, the nodes from the graph file) based on the number of incoming links and their ranks. In the paper, the authors used LiveJournal data set as their input for the algorithm. LiveJournal is a 1Gig text social network dataset, containing the graph data for 4,847,571 vertices and 68,993,773 edges. The file is in a text format, each line containing the space-delimited of one pair of connected vertices (one edge) and its size is 1 Giga byte.

The major building blocks of this algorithm are listed as the following:

1. **Initial Rank Assignment Job,** which sets the initial ranks for each URL(vertice, or simply node) to 1.
2. **Iterative Page-Rank Computation,** which computes the ranks based on the input iterations that the user provides. This step is divided into two jobs:

   2.1 Compute Page-Rank Join Job, which computes the Markov Chain matrix after one step transition.

   2.2 Rank Aggregate Job, which aggregates the column scores of the computed matrix related to each node and assign each node the computed rank.

In the following, it is explained that how each code implements the aforementioned building blocks.

HADOOP

The Initial Rank Assignment job is implemented by InitialRankAssignmentMapper and InitialRankAssignmentReducer:

```
conf = new JobConf(HadoopPR.class);
conf.setJobName("PageRank-Initialize");
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
conf.setMapperClass(InitialRankAssignmentMapper.class);
conf.setReducerClass(InitialRankAssignmentReducer.class);
conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);
FileInputFormat.setInputPaths(conf, new Path(inputPath));
FileOutputFormat.setOutputPath(conf, new Path(outputPath + "/i"
                        + iteration));
conf.setNumReduceTasks(numReducers);
```

```
                JobClient.runJob(conf);
```

The mapper and the reducer are the following:

```
public static class InitialRankAssignmentMapper extends MapReduceBase
                        implements Mapper<LongWritable, Text, Text, Text> {

                private Text startValue = new Text("1");
                private Text outputKey = new Text();
                private List<String> tokenList = new ArrayList<String>();

                public void map(LongWritable key, Text value,
                                OutputCollector<Text, Text> output, Reporter reporter)
                                throws IOException {
                        tokenList.clear();

                        String line = value.toString();
                        StringTokenizer tokenizer = new StringTokenizer(line);

                        while (tokenizer.hasMoreTokens()) {
                                tokenList.add(tokenizer.nextToken());
                        }

                        if (tokenList.size() >= 2) {        // save source node to destination node
                                                                                        // for initial ranking
                                outputKey.set(tokenList.get(0).getBytes());
                                output.collect(outputKey, startValue);
                                outputKey.set(tokenList.get(1).getBytes());
                                output.collect(outputKey, startValue);
                        }
                }
        }

        public static class InitialRankAssignmentReducer extends MapReduceBase
                        implements Reducer<Text, Text, Text, Text> {
```

```
            Text initValue = new Text("1.0");

            public void reduce(Text key, Iterator<Text> values,
                        OutputCollector<Text, Text> output, Reporter reporter)
                        throws IOException {
                  output.collect(key, initValue);
            }
}
```

Then, the Iterative Page-rank Computation is executed in a loop:

```
do {
                  startMinor = System.currentTimeMillis();
                  /****************** Page Rank Join Job ********************************/
                  conf = new JobConf(HadoopPR.class);
                  conf.setJobName("PageRank-Join");
                  conf.setInt("iter_count", iteration);

                  conf.setOutputKeyClass(Text.class);
                  conf.setMapperClass(ComputeRankMap.class);
                  conf.setReducerClass(ComputeRankReduce.class);
                  conf.setMapOutputKeyClass(TextPair.class);

                  conf.setInputFormat(TextInputFormat.class);
                  conf.setOutputFormat(TextOutputFormat.class);

                  conf.setPartitionerClass(FirstPartitioner.class);
                  conf.setOutputKeyComparatorClass(KeyComparator.class);
                  conf.setOutputValueGroupingComparator(GroupComparator.class);

                  // relation table
                  FileInputFormat.setInputPaths(conf, new Path(inputPath));
                  // rank table
                  FileInputFormat.addInputPath(conf, new Path(outputPath + "/i"
                              + (iteration - 1)));
```

```
// count table
FileInputFormat.addInputPath(conf, new Path(outputPath + "/count"));
FileOutputFormat.setOutputPath(conf, new Path(outputPath + "/i"
                + iteration));
conf.setNumReduceTasks(numReducers);

JobClient.runJob(conf);
iteration++;

/******************** Rank Aggregate Job ***********************/

conf = new JobConf(HadoopPR.class);
conf.setJobName("PageRank-Aggregate");

conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
conf.setMapOutputKeyClass(Text.class);

conf.setMapperClass(RankAggregateMapper.class);
conf.setCombinerClass(RankAggregateCombiner.class);
conf.setReducerClass(RankAggregateReducer.class);

conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf, outputPath + "/i"
                + (iteration - 1));
FileOutputFormat.setOutputPath(conf, new Path(outputPath + "/i"
                + iteration));
conf.setNumReduceTasks(numReducers);
conf.setInt("haloop.num.nodes", numNodes);

JobClient.runJob(conf);
iteration++;
endMinor = System.currentTimeMillis();
System.out.println(iteration / 2 + "-iteration  "
                                                        + (endMinor - startMinor) / 1000 + "s");
                } while (iteration < 2 * specIteration);
```

The mapper and reducer for Page-rank Join Job are the following:

```
public static class ComputeRankMap extends MapReduceBase implements
                        Mapper<LongWritable, Text, TextPair, Text> {

                /**
                 * tag for rank_value tuple
                 */
                private String tag0 = "0";

                /**
                 * tag for relation tuple
                 */
                private String tag1 = "1";
                private String tag2 = "2";

                private List<String> tokenList = new ArrayList<String>();
                private TextPair outputKey = new TextPair(new Text(), new Text());
                private Text outputValue = new Text();

                public void map(LongWritable key, Text value,
                                OutputCollector<TextPair, Text> output, Reporter reporter)
                                throws IOException {
                        tokenList.clear();

                        String line = value.toString();
                        if (line.startsWith("#"))
                                return;

                        StringTokenizer tokenizer = new StringTokenizer(line);
                        while (tokenizer.hasMoreTokens())
                                tokenList.add(tokenizer.nextToken().trim());
                        if (tokenList.size() >= 2) {
                                String valueToken = tokenList.get(1);
                                if ((valueToken.indexOf(".") >= 0 && valueToken.split("\\.").length <= 2)) { // initial

                                                                                                // rank

                                        outputKey.setSecondText(tag0);
                                        outputValue.set(valueToken.getBytes());

                                } else if (valueToken.endsWith("#*")) { // count

                                        String valueCount = valueToken.substring(0,
                                                        valueToken.length() - 2);
                                        outputKey.setSecondText(tag1);
                                        outputValue.set(valueCount.getBytes());

                                } else { // more than 2 iteration as original data after initial
                                                        // rank

                                        outputKey.setSecondText(tag2);
                                        outputValue.set(valueToken.getBytes());
                                }
                                outputKey.setFirstText(tokenList.get(0));
                                output.collect(outputKey, outputValue);
                        }
                }
        }

        public static class ComputeRankReduce extends MapReduceBase implements
                        Reducer<TextPair, Text, Text, Text> {

                private float srcRank = 0;

                private Text value = new Text();

                public void configure(JobConf conf) {
                    value.clear();
                    byte buffer[] = new byte[100];
                    value.append(buffer, 0, buffer.length);
            }
```

The mapper, combiner and reducer for Aggregate Ranks are the following:

```
public static class RankAggregateMapper extends MapReduceBase implements
                    Mapper<LongWritable, Text, Text, Text> {

            private Text outputKey = new Text();
            private Text outputValue = new Text();
            private List<String> tokenList = new ArrayList<String>();

            public void map(LongWritable key, Text value,
                            OutputCollector<Text, Text> output, Reporter reporter)
                            throws IOException {
                    tokenList.clear();
                    String line = value.toString();

                    StringTokenizer tokenizer = new StringTokenizer(line);
                    while (tokenizer.hasMoreTokens()) {
                            tokenList.add(tokenizer.nextToken());
                    }

                    if (tokenList.size() >= 2) {
                            outputKey.set(tokenList.get(0).getBytes());
                            outputValue.set(tokenList.get(1).getBytes());
                            output.collect(outputKey, outputValue);
                    }
            }
    }

    public static class RankAggregateCombiner extends MapReduceBase implements
                    Reducer<Text, Text, Text, Text> {
```

```
        private Text outputValue = new Text();


        public void reduce(Text key, Iterator<Text> values,
                        OutputCollector<Text, Text> output, Reporter reporter)
                        throws IOException {


                float totalRank = 0;
                while (values.hasNext()) {
                        totalRank += Float.valueOf(values.next().toString());
                }
                outputValue.set(Float.toString(totalRank).getBytes());
                output.collect(key, outputValue);
        }
}


public static class RankAggregateReducer extends MapReduceBase implements
                Reducer<Text, Text, Text, Text> {


        private Text outputValue = new Text();
        private static final float dampingFactor = 0.15f;
        private int numNodes = 0;
        private float prefix = 0f;


        public void configure(JobConf conf) {
                // default number from the number of nodes in soc-LiveJournal1 data
                this.numNodes = conf.getInt("haloop.num.nodes", 4847571);
                this.prefix = (1.0f - dampingFactor) / numNodes;
        }


        public void reduce(Text key, Iterator<Text> values,
                        OutputCollector<Text, Text> output, Reporter reporter)
                        throws IOException {
                float totalRank = 0;
                while (values.hasNext()) {
                        totalRank += Float.valueOf(values.next().toString());
                }
```

```
                                totalRank = prefix + dampingFactor * totalRank;


                                // output total rank
                                outputValue.set(Float.toString(totalRank).getBytes());
                                output.collect(key, outputValue);
                        }
            }
```

SPARK

The Initial Rank Assignment job for spark is the following:

```
            JavaPairRDD<String, Iterable<String>> links = lines.mapToPair(new PairFunction<String, String, String>() {
          @Override
          public Tuple2<String, String> call(String s) {
            String[] parts = SPACES.split(s);
parts[0]=
 parts[0].toString().replaceAll("AAAAAAAAAZAAAAAAAAAZAAAAAAAAAZAAAAAAAAAZAAAAAAAAAZAAAAAAAAAZAAAAAAAAAZ", "");
parts[1]=
parts[1].toString().replaceAll("AAAAAAAAAZAAAAAAAAAZAAAAAAAAAZAAAAAAAAAZAAAAAAAAAZAAAAAAAAAZAAAAAAAAAZ", "");
            return new Tuple2<String, String>(parts[0], parts[1]);
          }
        }).groupByKey(partition).persist(StorageLevel.MEMORY_AND_DISK());

        // Loads all URLs with other URL(s) link to from input file and initialize ranks of them to one.
        JavaPairRDD<String, Double> ranks = links.mapValues(new Function<Iterable<String>, Double>() {
          @Override
          public Double call(Iterable<String> rs) {
            return 1.0;
          }
      });
```

As it is shown in the code, an RDD variable is used to store the initial rank of the URLs, as well as the iterative ranks. That's why the code uses "**.persist**" action to insist on saving the rank values in memory for fast access in the iteration. In fact, by using the RDD and persist actions, we expect that the following iteration part execute much faster than the Hadoop implementation:

```
  // Compute Page-Rank Join Job
   for (int current = 0; current < Integer.parseInt(args[1]); current++)
   {
        // Calculates URL contributions to the rank of other URLs.
        JavaPairRDD<String, Double> contribs = links.join(ranks).values()
          .flatMapToPair(new PairFlatMapFunction<Tuple2<Iterable<String>, Double>, String, Double>() {
            @Override
            public Iterable<Tuple2<String, Double>> call(Tuple2<Iterable<String>, Double> s) {
              int urlCount = Iterables.size(s._1);
              List<Tuple2<String, Double>> results = new ArrayList<Tuple2<String, Double>>();
              for (String n : s._1) {
                results.add(new Tuple2<String, Double>(n, s._2() / urlCount));
              }
              return results;
            }
        });

        // Rank-Aggregation Job
        ranks = contribs.reduceByKey(new Sum()).mapValues(new Function<Double, Double>() {
          @Override
          public Double call(Double sum) {
            return 0.15 + sum * 0.85;
          }
        });
    }
```

As we compare the two codes, we can see that the code for Spark framework is more descriptive and concise than that of Hadoop framework. As for the final results, the two codes differ slightly with each other sine the Spark implementation calculates the Markov Chain transitions with 15% of teleportation probability.

The original codes and the compiled classes are included In the “Code Files” folder of the report attachment.

3. Experimentation

In this section, the results of the experimentations on executing the two codes are included.

For the configuration of **Hadoop** framework, we used 6 instances of **m1.small** size of **Cybera-Rac** cluster, using **1 master node**(name-node), **1 secondary name-node** and **4 data-nodes**. The operating system for each of the instances is **Ubuntu 16.04** and the configuration for each of the instances is as follows:

CPU(s): 2

CPU MHz: 2294.686

Memory: 2000 mb

HDD: 20G

Architecture: x86_64

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 25600K

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

On-line CPU(s) list: 0,1

Thread(s) per core: 1

Core(s) per socket: 1

Socket(s): 2

NUMA node(s): 1

Vendor ID: GenuineIntel

CPU family: 6

Model: 63

```
Model name:  Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
Stepping:  2
BogoMIPS: 4589.37
Virtualization:  VT-x
Hypervisor vendor:  KVM
Virtualization type:   full
NUMA node0 CPU(s):     0,1
```

The Cluster configuration for Hadoop framework is set up based on the tutorial published on e-class. The hadoop version used for the cluster is **Hadoop-2.7.3** and the jdk version used JAVA_HOME environment variable is **jdk-1.8.0** The hdfs-site and mapred-site configuration files are set as the following:

```
vim $HADOOP_CONF_DIR/core-site.xml

<property>
<name>fs.default.name</name>
<value>hdfs://master:9000</value>
</property>
<property>
<name>hadoop.tmp.dir</name>
<value>/home/ubuntu/hadoop-2.7.3/tmp</value>
</property>


vim $HADOOP_CONF_DIR/hdfs-site.xml

<property>
<name>dfs.replication</name>
<value>2</value>
</property>
<property>
<name>dfs.permissions</name>
<value>false</value>
```

```
</property>
<property>
<name>dfs.secondary.http.address</name>
<value>master2:50090</value>
<description>Enter your Secondary NameNode hostname</description>
</property>
```

For the configuration of **Spark**, we used the **same Cybera-Rac** cluster nodes, meaning all of the instances are of m1.small with the same hardware specifications and operating system. However, the only difference of Spark configuration with hadoop is that our Spark cluster uses **1 master node** and **4 worker nodes,** without any secondary master node. We used **Spark-2.0.2** version for **hadoop 2.7 and later** with the **same jdk version** for hadoop.

The cluster and system setup for the original paper was the following:

- **cluster nodes**: 50 m1.xlarge Amazon EC2 spot instances

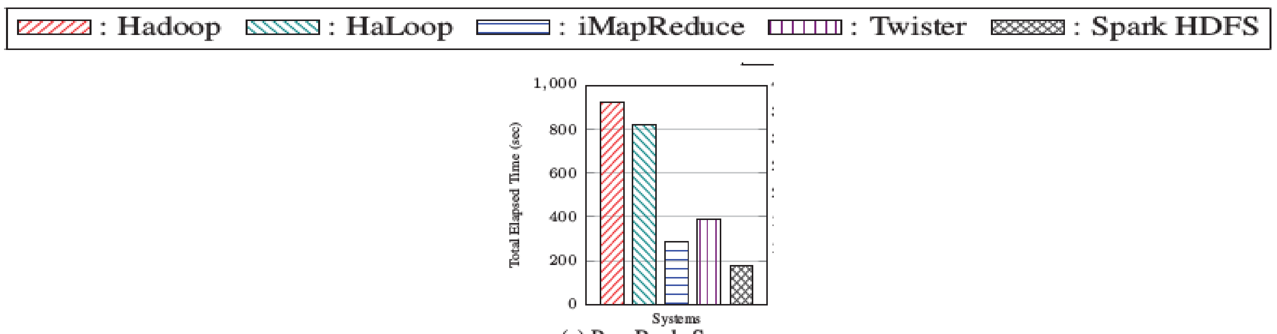
- **Per instance Configuration**:
    - **Number of Virtual CPUs**: 4
    - **Local storage**: 4*420G
    - **Main memory**: 15G
    - **Operating System**: Ubuntu 14.04

- **Hadoop version:** 1.2.1
- **Spark version:** 1.2.0
- **JDK version:** 1.8.0

For the remainder of this section, we are going to compare the experimentation results between the original paper and our execution of the same algorithms. The evaluation metrics that we are investigating on are: overal comparision of total elapsed time, Total HDFS read, Total Reduce Shuffle I/O, the effect of data size and the effect of iteration number. Furthermore, for our experiments, each data-point is computed by the **average of its 2 executions** (due to time limitations).

Total Elapsed Time

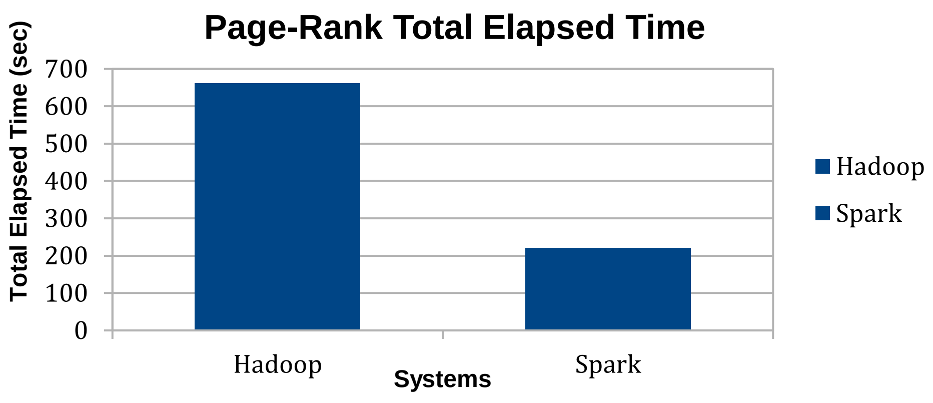
As the paper states, "*the total elapsed time is determined to be the interval between the job finish time and the job start time included in system log files*"[1] and the result including the execution of Page-Rank algorithm on hadoop and spark is the following.



(a) PageRank-Soc.

They substituted all vertex identifiers in LiveJournal input dataset with longer strings to increase its size without changing its network structure. As a result, the data set was extended from 1 GB to 10 GB and used that data as a base-input data set for the total HDFS read and Total Reduce Shuffle I/O evaluations as well. However, The iteration number and the size of the data is not stated.

In our experiment, we divided the LiveJournal text file by 10 (using split linux command) to reduce the input file and execute both codes for Hadoop and Spark with one of the 100M chunks for this experiment as well as the total HDFS read and the total reduce shuffle I/O evaluations. Furthermore we set the iterations to 5 for the mentioned evaluations and the result is the following:

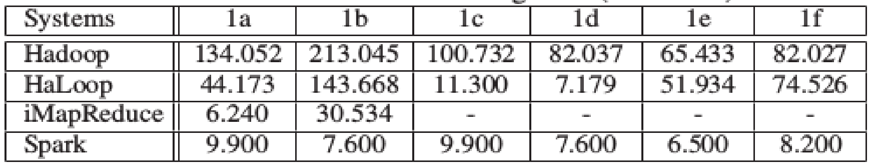


As we expected, Spark outperformed Hadoop by approximately a factor of 3, since it is using RDDs and persists the ranks RDD in memory, which enables spark to save all the needed data in memory(as far as it can) instead of reading from disk before and after each map and reduce. However, in the original paper, This metric is around the factor of 4. One speculation could be that in the paper's experiment, the number of iterations was high, since by raising the iteration number, the coefficient of Spark outperforming Hadoop will rise. The reason could be that by each iteration, Hadoop should read the data from file, but in spark, it is saved in memory. Moreover, by looking at the effect of iteration evaluation in the paper, we can confirm this theory, since as the iteration number goes up, the slope of the normalized elapsed time for Spark goes down in comparison with that of Hadoop.

Total HDFS read

As the paper states, "total HDFS read is the total amount of the data read from the HDFS—not from the local file system—during the entire execution of an algorithm. This metric assesses how effectively each system reduces remote disk I/O. It is the sum of the outputs of the counter "HDFS_BYTES_READ" for Hadoop and it is the first output of the counter "Input" for Spark since the subsequent outputs of that counter indicate the accesses to the RDD which is typically stored in main memory"[1], and the result was the following (1a is the Page-rank algorithm based on LiveJournal input):

Table 3: Total HDFS read in Figure 1 (Unit: **GB**).



Systems	1a	1b	1c	1d	1e	1f
Hadoop	134.052	213.045	100.732	82.037	65.433	82.027
HaLoop	44.173	143.668	11.300	7.179	51.934	74.526
iMapReduce	6.240	30.534	-	-	-	-
Spark	9.900	7.600	9.900	7.600	6.500	8.200

With the assumptions of the Total Elapsed time evaluation (meaning the input file divided to one tenth and the iteration set to 5), the log for hadoop was self descriptive and as the paper referred, the counter "*HDFS_BYTES_READ*" was explicitly included in its execution log, and the summation of that counter was 4497 MB. Here we cannot make a direct comparison with the result for the original paper, since the iteration number is not specified for the original paper but roughly speaking, since for each iteration we have 4 maps and 4 reduces.

For Spark, it was expected that the first Counter "Input" in the execution log be a number around the input data size (100MB). However, there was no "Input" counter explicitly indicated in the execution log. One speculation could be that the version of the spark for the authors is different than our version. Although it could be possible to retrieve this information by adding some code to the program but due to the time limitations we couldn't manage to do that. However the Spark execution log for this experiment is included in the "Code Files>Spark_PageRank" path in the project's attachment named "100M_Spark_5iter.log". I discussed this issue with Hamidreza but unfortunately we weren't able to solve it.

Total Reduce Shuffle I/O

As the paper states, "Reduce shuffle I/O is the total amount of the data shuffled from mappers to reducers during the entire execution of an algorithm. This metric assesses how effectively each system reduces networkcommunication for the shuffle. It is the sum of the outputs of the counter "Reduce Shuffle Bytes" for Hadoop" and the counter "Shuffle Read" for Spark"[1], and the result was the following (1a is the Page-rank algorithm based on LiveJournal input):

Table 4: Reduce shuffle I/O in Figure 1 (Unit: **MB**).

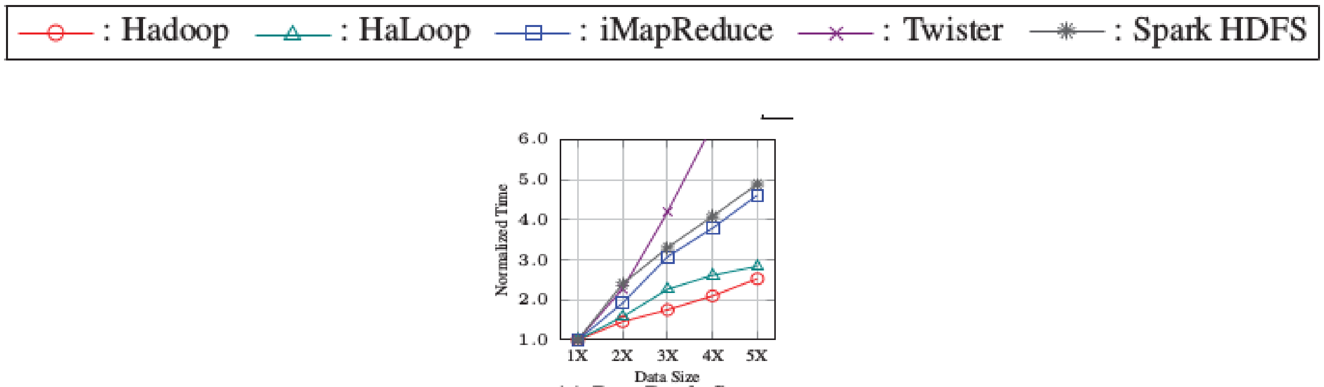
Systems	1a	1b	1c	1d	1e	1f
Hadoop	30,593	241,205	14,943	98,227	0.847	0.241
HaLoop	16,912	141,951	2,979	9,642	0.818	0.739
iMapReduce	970	16,842	-	-	-	-
Spark	8,500	34,600	2,883	2.601	1.988	0

Our experimentation conditions for this part was the same as the “Total HDFS READ” part. The total reduce shuffle I/O size for hadoop is calculated based on what the paper states, so the summation of *Reduce Shuffle Bytes* in the execution log was 2309 MB and as for the Total HDFS READ experiment, we couldn't make a direct comparison with the original paper's result since the iteration is not specified.

For the spark part, just like the previous experiment, there was no “*Shuffle Read”* counter specified in the log.

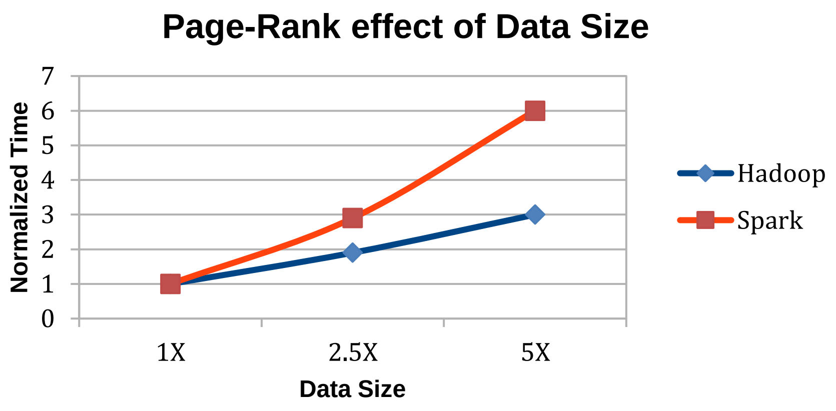
The effect of data size

In the paper, “*he authors increased the size of their base-input data set by replicating the graph file from 1X to 5X and investigated the Normalized time w.r.t. the input size. As the paper states, “normalized time is the ratio of the total elapsed time of a certain configuration to that of the base configuration. For example, in scalability tests, normalized time is the ratio of the total elapsed time for a certain data set to that for the smallest data set. Consequently, normalized time always starts from 1.*” [1] The result was the following:



(a) PageRank-Soc.

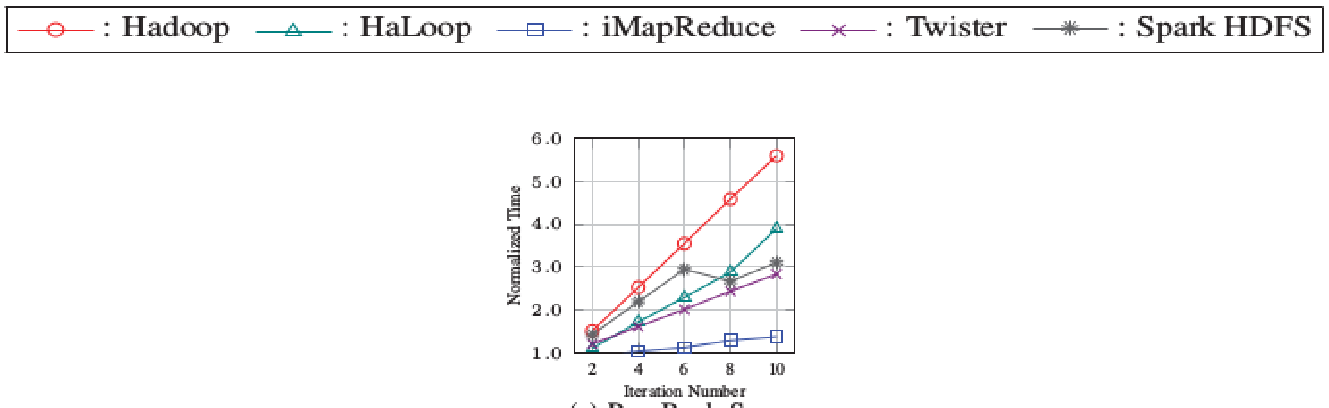
For this task, we chunked the data into 100MB, 250MB and 500MB pieces and experimented based on those data sizes and the normalized time was based on the execution with 5 iterations:



If we compare our results with the original paper's result, we can say that they are approximately on the same track. However, there are some slight differences between them. For instance, the normalized time for spark in 5X is slightly higher than that of the original paper's results. One speculation could be that for the 5X size, we may have some memory spills and this could be a cause for the execution time to go higher than normal.

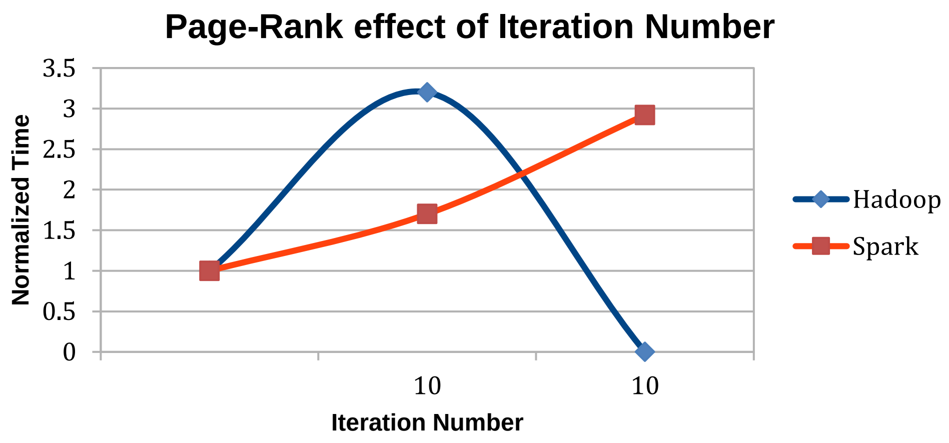
The effect of iteration number

In the paper, the authors executed the Page-Rank algorithm by varying number of the **Iterative Page-Rank Computation** and inversitgated the Normalized time w.r.t the input size:



(a) PageRank-Soc.

For this experiment, we executed the algorithm with the iteration numbers of 1, 5 and 10 and the normalized time was based on the 100M input data type:



In this experimentation, we couldn't execute the program for Hadoop on 10 number of iterations, since during the multiple times of execution, some nodes went stuck each time and the master issued the KILL signal for them with the error log of “"ERROR org.apache.hadoop.hdfs.server.” I consulted with Hamidreza for this issue but unfortunately the issue wasn't solved and that's why the experiment result doesn't have any data point for hadoop with 10 iterations.

But as we can see in the results, it is approximately on the same track with the original results, However the normalized time for spark in this experiment is a little bit lower than the original paper's results, and one probable reason could be because of the fact that the input size of the program is relatively small (100MB) and there is minimum or no spillage.

1. References

[1] H. Lee et al. , *An Experimental Comparison of Iterative MapReduce Frameworks*, CIKM 2016, pp. 2089-2094.

[2] https://github.com/IterativeExperimentsMapReduce/Iterative_Experiments_MR

[3] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, November 2016.