Bilkent University

CS 425

Algorithms for Web-Scale Data

Playing with PageRank

Comparing Different Implementations

Authors:
Naisila Puka
Endi Merkuri
Fatbardh Feta

Student Numbers:
21600336
21600337
21600334

December 28, 2018

Contents

1 Introduction

In this project we have implemented PageRank in three different paradigms: serial, parallel shared-memory and distributed Map-Reduce. We used OpenMP application programming interface that supports multi-platform shared memory multiprocessing programming in C++, and Apache Spark unified analytics engine for the distributed implementation. After implementing the algorithms, we tested them with three different datasets taken from Stanford Large Network Dataset collection[1], namely the NotreDame web graph with 1.5 Million egdes, the Google web graph with 5.1 Million edges, and the Patent Citation Network with 16.5 Million edges. Then we compared the amount of time taken to complete the execution in each of the implementations as well as the PageRank results.

2 PageRank Review

PageRank algorithm is the reason why Google Search became famous: it is an efficient and reliable algorithm used to rank web pages. The main idea behind ranking the web pages is: "Believe what other pages say about you instead of what you say about yourself", as also illustrated in Figure 1.

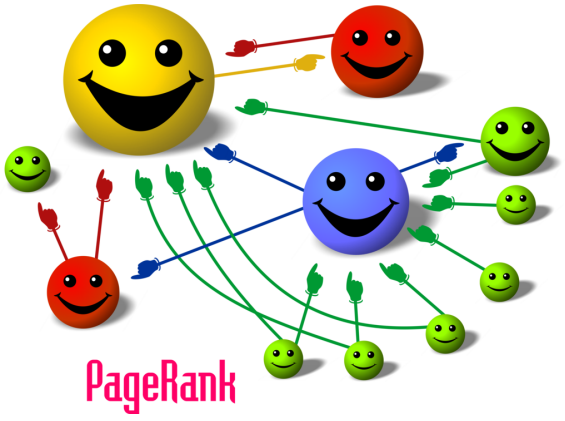

Figure 1: Cartoon illustrating the basic principle of PageRank [2]

It is a link-analysis algorithm which assigns numerical weighting to each element of the web graph. This weight measures the "relative" importance of the web page within the graph. One of the most famous interpretations of it is the Random Walk one: PageRank outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page after a

sufficiently long amount of time. The basic formula for the PageRank of one page would be as in Formula 1.

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)} \tag{1}$$

In Formula 1, $PR(u)$ means the PageRank of u (i.e. the probability that the random walker will be at page u). B_u means the list of in-links of page u, and $L(v)$ means the out-degree of page v. Since this value of the PageRank needs to be computed for each of the pages, we use a vector of webpages, r, where webpage at entry i is denoted by r_i. Starting with initial pagerank values as $1/|r|$ and repeating Formula 1 a sufficient amount of times, convergence within a certain value between r_{old} and r_{new} is guaranteed to be reached under certain conditions [3]. For convergence, different distances between two vectors can be used, such as $L1 - norm$. Taking care of spider traps and pages without out-links (deadends), the random walker follows probability distributions with a probability of β, where $\beta < 1$, and teleports to any webpage with teleportation probability of $1 - \beta$. Considering Formula 2, sum of all page ranks is always 1 and convergence is achieved.

$$\forall j \quad PR_{new}(r_j) = \sum_{v \in B_{r_j}} \frac{PR_{old}(v)}{L(v)} \tag{2}$$

$$PR_{new}(r_j) = 0 \; if \; r_j \; has \; no \; inlinks$$

$$\forall j \quad PR_{new}(r_j) = PR_{new}(r_j) + \frac{1 - S}{|r|}$$

$$where \; S^* = \sum_j PR_{new}(r_j)$$

$$r_{old} = r_{new}$$

$$repeat \; until \; \sum_j |PR_{new}(r_j) - PR_{old}(r_j)| < \epsilon$$

* S is exactly β if there are no deadends!

We will be using all the steps in Formula 2 throughout the implementation explanations.

3 Datasets

We have implemented our algorithms in a way that they receive an input file of data of the form shown in Figure 2. Each line of the file represents an edge in the form $OUTLINK- > INLINK$. The Notre-Dame dataset consists of a total of 325729 nodes and 1497134 edges, the Google dataset consists of a total of 875713 nodes and 5105039 edges, and the Patent Citation Network consists of a total of 3774768 nodes and 16518948 edges.

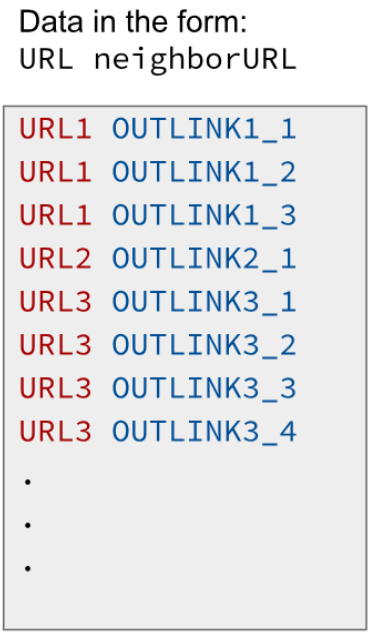

Figure 2: Dataset File Depiction

4 Serial Implementation

Our first implementation of PageRank was the serial one using C++. We decided to firstly implement the algorithm in this way so that we could create a base for later implementing in parallel using OpenMP and comparing these two. Each line of our dataset comprised of two integers, the first one being one of the nodes of the graph and the second one being one of the outgoing links of this node. To implement PageRank we used a Node C++ "*struct*" which contains an array of the outgoing links of the current node and their number. This "*struct*" also keeps track of the

current and the previous PageRank value of the current node. The nodes are then stored in an array.

4.1 PseudoCode

Following is the pseudocode of serial PageRank algorithm, completely based on Formula 2 (**in all the implementations we used β value of 0.85 and a convergence threshold of 0.0001 using $L1 - norm$**). We observed that it is easier for us to implement the algorithm in Formula 2 by doing a transition from pull-based rank calculation to push-based, as shown in Algorithm 1.

Algorithm 1 Serial PageRank

Data Preprocessing Procedure *ignored in the explanation*

Serial PageRank Procedure
While $L1 - norm$ is greater than the threshold
    For each node
        Update previous rank and set current rank to 0
    For each node
        Add its pagerank contribution to its outgoing links
    For each node
        Multiply its current rank with β
        Calculate the sum of all the ranks
        Find the leakage due to dead-ends and calculate teleportation probability
    For each node
        Add the teleportation probability to the nodes current rank
        Calculate the error using L1-norm

4.2 Computer Configuration

We ran our serial implementation in a machine which had a dual-core Intel Core i5 CPU with four hardware threads and 4GB of RAM.

5 Parallel Implementation

After implementing the serial version of PageRank, we used it as a basis for the parallel implementation using OpenMP. Since each iteration in the outer loop of the PageRank algorithm is dependent on the previous iterations it could not be parallelized, so we used OpenMP to run the rank update part of the algorithm. By doing this, the nodes of the graph are distributed to the threads that are created. In this part of the implementation, we faced the problem of the data race conditions, because the array of nodes was shared between the different threads. In order to deal with these, we used the OpenMP "*critical*" and "*atomic*" constructs to allow only one thread to be able to write to the array of nodes at a time. This can also cause a small decline in the improvement from our serial to parallel implementation of PageRank.

5.1 OpenMP

Before we continue with the pseudocode, we give some basic information on how OpenMP works. OpenMP is an API that supports multi-platform shared memory multiprocessing programming using languages as C, C++ and Fortran[4]. This API uses a method called multithreading in which a master thread distributes the work to different "*slavethreats*" which carry their job in parallel, as shown in Figure 3. The Pragma Parallel provided by OpenMP allowed us to fork additional threads to carry out the work enclosed in the construct in parallel. The construct makes sense for us since PageRank's construct is based on looping since rank calculations have to continue until convergence is reached. In this project in order to further analyze and understand how a parallel shared-memory implementation of the PageRank algorithm performs, we used OpenMP API with C++.

5.2 PseudoCode

We have made use of OpenMP parallelizations in three consecutive loops necessary to be computed for PageRank, as shown in Figure 4. For each stage, four hardware threads are running.

The pseudocode is very similar to the serial implementation pseudocode, except that now we execute the loops in parallel. It goes like in Algorithm 2. As we can see, the first stage of PageRank shown in Figure 4 is completed in two consecutive loops both executed in parallel by OpenMP.

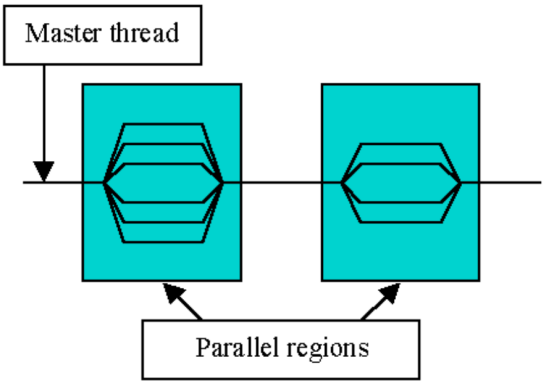

Figure 3: Basic Illustration of Multi-Threading[5]

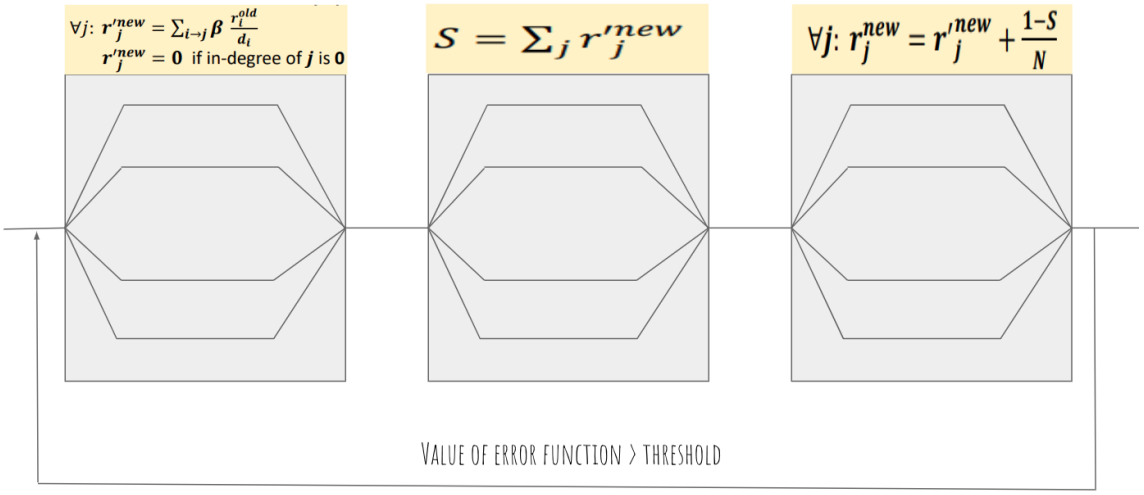

Figure 4: Three stages of PageRank done in parallel

5.3 Computer Configuration

We ran our parallel implementations of the Pagerank algorithms in the same machine as the serial implementation. It had a dual-core Intel Core i5 CPU with four hardware threads and 4GB of RAM.

6 Distributed Implementation with Map-Reduce

As we learned from CS425 Course, Map-Reduce is a programming model and an associated implementation for processing and generating big data sets with a parallel,

Algorithm 2 Parallel PageRank

Data Preprocessing Procedure *ignored in the explanation*

Parallel PageRank Procedure
While $L1 - norm$ is greater than the threshold
PARALLEL For each node
Update previous rank and set current rank to 0
PARALLEL For each node
Add its pagerank contribution to its outgoing links
PARALLEL For each node
Multiply its current rank with β
Calculate the sum of all the ranks
Find the leakage due to dead-ends and calculate teleportation probability
PARALLEL For each node
Add the teleportation probability to the nodes current rank
Calculate the error using L1-norm

distributed algorithm on a cluster.[6] PageRank can be computed using Map-Reduce approach to distribution and parallelization. Spark framework is a really efficient framework for Map-Reduce, and we used it to test our implementation. Some information about Spark is given in the following section.

6.1 Spark

Spark is a general-purpose distributed data processing engine, perfectly compatible with Map-Reduce, and the way it works is briefly illustrated in Figure 5. Spark main structure are RDD's (resilient distributed dataset) which are a fault-tolerant collection of elements that can be operated on in a distributed manner. The RDD was the main structure used in the Map-Reduce implementation, because it enabled file-chunking. More specifically, we have used RDD's in PySpark for Map-Reduce, as explained further in the following subsection.

6.1.1 PySpark

PySpark is the branch of Spark that helps Python users have access to Spark's API and work with the convenient RRD-s to chunk the files properly in Map-Reduce implemented algorithms. Below are some of the most important PySpark functions[8] that we used in this Map-Reduce implementation of PageRank (including RDD's):

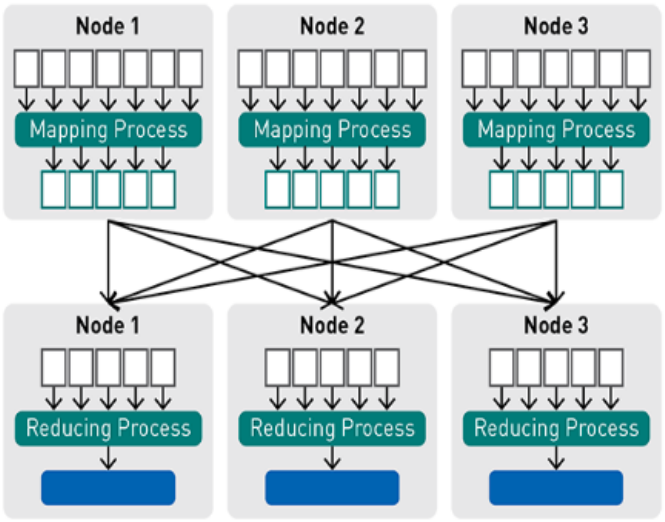

Figure 5: High-Level Map-Reduce in Spark setup[7]

- **rdd**: creates a resilient distributed dataset(RDD) from a given Data Structure.
- **flatMap()**: is a transformation operation that is applied to all the elements of the RDD and returns a new RDD which can have a different size from its ancestor (**map()** is an other similar function to flatMap).
- **reduceByKey()**: is an RDD function that merges the values of each key using an associative reduce function.
- **join()**: joins two given data frames on a column based on the function that is passed.

6.2 Pseudocode

To implement the algorithm, we went through two consecutive Map-Reduce steps, although in theory we had planned only the second Map-Reduce step, which is the PageRank step. We realized during the project making that we could preprocess data and bring it in the way we wanted by performing an initial Map-Reduce on them. So basically we have:

- **Data Preprocessing**: Initial Map-Reduce to prepare the data
- **PageRank**: Main Map-Reduce of the algorithm.

In more detail, the way it goes is like this:

With $\beta = 0.85$

- **1)** Map Reduce Algorithm for Data Processing
- **2)** Another Map Reduce Algorithm for PageRank iterations
- **3)** Reinsert PageRank Leakage
- **4)** Check for Convergence, if not met go back to **1)**

We will explain the pseudocode of each Map-Reduce in detail in the following subsections.

6.2.1 First Map-Reduce: Preparing the Dataset

What we aimed to do in this Map-Reduce step was to control the data in the way that is more feasible for us to use in the main Map-Reduce. Figure 6 depicts the aim of this Map-Reduce.

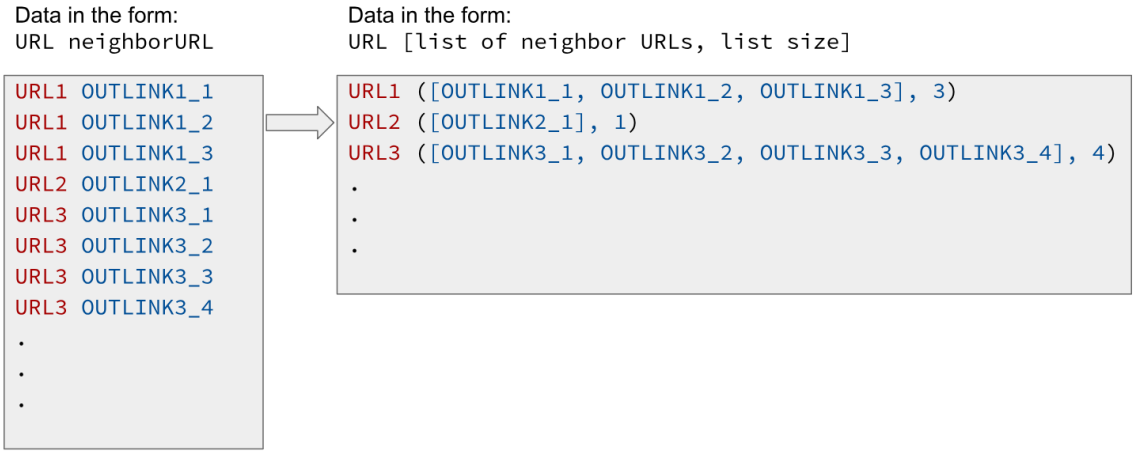

Figure 6: Preparing the dataset

The *Map* function is going to be a simple Identity function, meaning that it will just generate the same key and value that came as an input from the file chunk. The *Reduce* function is going to compute the out-degree of each node, and provide it as output together with the list of out-links. The pseudocode of the algorithm is explained in Algorithm 3.

Algorithm 3 Data Preprocessing Map-Reduce

Map Procedure
for each $URL\ OUTLINK$ line of the document
    generate (key = URL, value = $OUTLINK$)

Reduce Function
for each (URL, $OUTLINK\_LIST$)
    $OUT\_DEGREE \leftarrow length$(OUTLINK_LIST)
    generate (key = URL, value = ($OUTLINK\_LIST$, OUT_DEGREE))

6.2.1.1 Complexity Evaluation

We have based our complexity evaluation of Map-Reduce algorithms on these 4 factors:

- **Replication rate**: 1
- **Average Reducer Size**: $\bar{d}$
- **Communication Cost**: $2E$
- **Number of Reducers**: N

where $\bar{d}$ stands for the average number of out-links of a node, E stands for the number of edges in the graph, and N stands for the number of nodes in the graph.

6.2.2 Second Map-Reduce: PageRank

After getting the data in the first Map-Reduce in the way that interests us, we used the output of that Map-Reduce as an input to the main one. With this Map-Reduce we aim to do this part of the Formula 2

$$\forall j \quad PR_{new}(r_j) = \sum_{v \in B_{r_j}} \frac{PR_{old}(v)}{L(v)}$$

$$PR_{new}(r_j) = 0 \ if \ r_j \ has \ no \ inlinks$$

As in the previous implementations, we will use the push-based approach instead of the pull-based one because it fits perfectly with the concept of Map-Reduce. The *Map* function will output all the outlinks of the current node key along with the rank contribution coming from the node key as the value. It will also output the node key with a value of 0 to take care of the case with no in-links. The *Reduce*

function will sum up all the values coming as contributions from inlinks of a specific node key. For a better understanding, the pseudocode of the algorithm is explained in Algorithm 4.

Algorithm 4 PageRank Map-Reduce

Map Procedure
for each (URL, ($OUTLINK\_LIST$, OUT_DEGREE, $CURRENT\_RANK$))
    generate (key = URL, value = 0)
    for $i = 1$ to OUT_DEGREE
        generate (key = $OUTLINK\_LIST[i]$, value = $\frac{1}{CURRENT\_RANK}$)

Reduce Function
for each (URL, $CONTRIBUTIONS\_LIST$)
    $NEW\_RANK \leftarrow sum$(CONTRIBUTIONS_LIST)
    generate (key = URL, value = NEW_RANK)

6.2.2.1 Complexity Evaluation

Similarly to previous Map-Reduce, we have based our complexity evaluation of Map-Reduce algorithms on these 4 factors:

- **Replication rate**: $\bar{d}$
- **Average Reducer Size**: $\bar{d}$
- **Communication Cost**: $2E$
- **Number of Reducers**: N

where $\bar{d}$ stands for the average number of out-links of a node, E stands for the number of edges in the graph, and N stands for the number of nodes in the graph.

6.2.3 Finishing the iteration

After doing the two consecutive Map-Reduce steps, we do the routine finishing steps for that iteration, as shown in Algorithm 5.

Algorithm 5 Finishing the Iteration

Leaked PageRank Calculation Procedure
$S \leftarrow sum$(NEW_RANK)

Leaked PageRank Reinsertion Procedure
for each NEW_RANK in NEW_RANK[]
    NEW_RANK ←NEW_RANK+(1−S)/N

Convergence Procedure
Calculate L1 Norm of Rank Vectors and Check for convergence

6.3 Computer Configuration

We created a cluster in Google Cloud to test our algorithm. Each of us had a free trial of 300$, which allowed us to experiment and come up with meaningful conclusions. Google Cloud Dataproc is a cloud-based managed Spark and Hadoop service offered on Google Cloud Platform.[9] The machines of the cluster setup we have used are found in East Europe and the configuration is as follows:

- **Master Node**: 2 cores
- **Worker 1**: 2 cores
- **Worker 2**: 2 cores
- **Worker 3**: 2 cores

7 Comparisons

In this section we will show comparisons between the different implementations. A clarification is important at this point: When we implemented the algorithm in Serial and in Parallel, we didn't observe any problem in calculating the amount of leaked PageRank ($1 - S$ as in Formula 2). After all, it is just a similar loop to the others loops executed before it. However, in the Map-Reduce implementation, adding a Map-Reduce step or a loop for calculating the amount of leaked PageRank took a significant amount of time because summing all the elements of an RDD data structure is not a feasible computation. For this reason, and since our datasets were used just for testing, we have changed our datasets to eliminate the dead-ends by adding just one outlink to every dead-end. We chose this out-link among the dead-end's in-links to make more sense. This resulted in comparing the implementations

in two different ways: datasets with and without deadends. Following sections show data for each way of comparison.

7.1 Without Deadends

As expected, in the case of no dead-ends, the distributed PageRank algorithm is much faster, especially when the dataset gets larger. However, it has a drawback in the Spark setup time. On the other hand, the data preprocessing time gets faster in distributed implementation as the dataset gets larger. So we see performance gain from shared memory to distributed. Also, as expected the parallel implementation is faster than the serial one. Figure 7 shows the results for the Notre-Dame dataset, Figure 8 shows the results for the Google dataset, and Figure 9 shows the results for the large Patent Citation Network. **We have obtained a really good result in the PageRank computation of the large Patent Citation Network, which finished in 0.79 seconds, which is extremely fast.**

	Setup	Data Processing	PageRank	Total
Serial	-	0.83	4.64	5.47
Shared-Memory	-	0.91	4.39	5.3
Distributed	11.8	1.78	3.14	16.72

Figure 7: Results for Notre-Dame dataset without deadends, with a total of 33 iterations, all times in seconds

	Setup	Data Processing	PageRank	Total
Serial	-	3.19	19.4	22.6
Shared-Memory	-	3.06	14	17.1
Distributed	12.1	2.16	3.39	17.65

Figure 8: Results for Google dataset without deadends, with a total of 35 iterations, all times in seconds

	Setup	Data Processing	PageRank	Total
Serial	-	9.77	29.3	39.1
Shared-Memory	-	10.3	22.7	33
Distributed	11.9	1.78	0.79	14.47

Figure 9: Results for Patent Citation Network dataset without deadends, with a total of 10 iterations, all times in seconds

7.2 With Deadends

In the case of deadends, as explained, we see a major drawback in time when implementing PageRank with Map-Reduce. This happened due to **RDD**'s and **map()** functions incompatibility with a full sum function on the whole data. Parallel and Serial implementations have almost the same configuration, but because the dataset is small to see the difference, we got almost same times for Serial and Parallel implementation on Notre-Dame dataset. Figure 10 shows the results for the Notre-Dame dataset, whereas Figure 11 shows the results for the Google dataset. We didn't run the computations on the large Patent Citation Network since the distributed one would take a huge amount of time to complete.

	Setup	Data Processing	PageRank	Total
Serial	-	0.69	3.41	4.1
Shared-Memory	-	0.74	4.04	4.78
Distributed	12	1.78	655	669

Figure 10: Results for Notre-Dame dataset with deadends, with a total of 33 iterations, all times in seconds

8 Conclusions

As expected, we observed that the parallel implementation takes shorter amount of time than the serial one. Also, we saw that the distributed implementation using Map-Reduce is actually really efficient compared to Parallel implementation. Also, data preprecessing is done faster in Spark. However, there is a trade-off of this result:

	Setup	Data Processing	PageRank	Total
Serial	-	2.76	17.6	20.4
Shared-Memory	-	2.98	14.2	17.2
Distributed	18 s	2.16	2700	2720

Figure 11: Results for Google dataset with deadends, with a total of 35 iterations, all times in seconds

Spark setup requires its own time. Also, the distributed implementation is not feasible with the calculation of PageRank leakage in case the graph has dead-ends. However, considering much larger graphs than the datasets we used, the Spark setup time will eventually become unsignificant, so it is better to use that implementation and not parallel one. Also, with more sophisticated methods of calculating the PageRank leakage due to dead ends, this problem can also be overcome in the distributed implementation.

9 Contributions

Naisila Puka: Distributed Implementation

Endi Merkuri: Serial and Parallelized Implementation

Fatbardh Feta: PySpark functions and OpenMP API research

10 What We Learned

To make this project we have learned Spark (in particular PySpark) framework and OpenMP API, as well as how to deploy a Google Cloud Dataproc cluster. Also, we have learned to appreciate Map-Reduce more since there are very powerful algorithms which can be implemented using Map-Reduce and be very efficient.

References

[1] Stanford Large Network Dataset Collection
https://snap.stanford.edu/data/, Accessed 24 December 2018

[2] PageRank Image
https://en.wikipedia.org/wiki/PageRank#/media/File:PageRank-hi-res.png,
Accessed 24 December 2018

[3] PageRank Lecture http://www.cs.cmu.edu/~elaw/pagerank.pdf, Pg. 25, Accessed 24 December 2018

[4] Gagne, Abraham Silberschatz, Peter Baer Galvin, Greg. Operating system concepts (9th ed.). Hoboken, N.J.: Wiley. pp. 181–182.

[5] What is OpenMP
https://www.dartmouth.edu/~rc/classes/intro_openmp/,
Accessed 24 December 2018

[6] Map-Reduce https://en.wikipedia.org/wiki/MapReduce,
Accessed 24 December 2018

[7] What is Spark
https://mapr.com/blog/spark-101-what-it-what-it-does-and-why-it-matters/,
Accessed 24 December 2018

[8] PySpark Package
http://spark.apache.org/docs/2.1.0/api/python/pyspark.html,
Accessed 24 December 2018

[9] Cloud Dataproc https://cloud.google.com/dataproc/, Accessed 24 December 2018