

PRISM – PageRank Implementations and Simulation Models

Author:

*Rushabh Kela**

Author addresses:

** Vellore Institute of Technology, Vellore*

Emails: 1 rushabhbhagwandas.kela2019@vitstudent.ac.in

Abstract: *PageRank (PR) is an algorithm developed by Google Search to rank web pages in their search engine results. It is named after both the term "web page" and co-founder Larry Page. 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. PageRank works by counting the number of links to a page to determine a rough estimate of how important the website is. It was first used to rank web pages in the Google search engine. Nowadays, it is more and more used in many different fields, for example in social network graphs of users in social media, ranking authors based on research paper citations etc. The assumption is that more important websites are likely to receive more links from other websites.*

Keywords: *PageRank, Algorithms, Distributed environments, Parallel processing, Serial, PySpark, OpenMP, Personalization, Weighted Graphs.*

1. Introduction

In this project, I will be implementing and improvising the PageRank algorithm. The PageRank algorithm is not only used in Google searches, but it is also used in various social media applications, author rankings from research papers and articles and other domains. In most scenarios, the dataset is huge and computing the PageRank scores using the traditional implementation of PageRank can be quite inefficient and computationally ineffective.

Serial implementations of algorithms will take a lot of running time, especially when the algorithms are computationally complex and time – consuming, or the input value is quite large. A parallel processing approach can reduce the running time of the algorithm, by dividing the computations among several processors. This leads to faster execution time and throughput. To improvise better, a distributed processing approach can link together multiple computer servers over the network, to share data and coordinate processing power. Apart from reducing the execution time, it also offers advantages in scalability, performance, resilience and cost – effectiveness.

In this project, I first analysed various literature articles in the PageRank domain and their various approaches. The traditional serial implementation of the PageRank algorithm is modified using OpenMP parallel processing methods to run on a parallel environment and improved on the shortcomings of previous approaches related to

dead – ends and spider – traps. It was further improvised using Apache PySpark distributed processing approach deployed on Google Cloud. Finally, I also present a personalised PageRank algorithm, taking into considerations the various factors the affect the results of a query put by a user. At the end, a comparative analysis of all the algorithms is done. The results were as desired and the running time of the algorithm was reduced substantially, with accurate results provided as the output.

2. Dataset description

2.1. Basic dataset for validation of results

Table 1. Basic dataset

Source	Destination
1	2
1	3
1	4
2	3
3	1
3	2
4	3

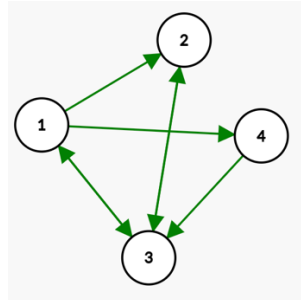


Fig. 1. Graphical view of the basic dataset

2.2. Actual dataset

The actual dataset is taken from the Stanford Large Network Dataset Collection repository which contains many datasets that can be used for PageRank Algorithm Analysis and Implementations. Few of the network datasets include:

- Social networks
- Citation networks
- Amazon networks
- Internet networks and many others

I have taken the citations networks dataset where the nodes represent papers and edges represent citations. A PageRank implementation on this dataset will give the output as the PageRank value associated with each node. This can be used to rank the

papers, suppose on Google Scholar to give appropriate and best possible results to users.

Dataset Statistics :

- Nodes : 3774768
- Edges : 16518948
- Size of dataset : 307.3 MB

3. Literature review

One of the main disadvantages of conventional page rank algorithm is that it takes huge computing time which is solved in [1] using sparse matrix and parallel processing but it takes lot of memory and it will be mostly filled with zeros which is inefficient here. In [3] computational time of the algorithm has been improved by using parallel processing but it did not consider spider trap drawback of PageRank algorithm. Same drawback can be seen in [2] where a method for measuring the importance of scientific papers based on the Google's PageRank is developed.

Generalization of page rank is done in [7] in which usage of page rank for systems analysis of road networks, as well as biology, chemistry has been explained. To introduce the web surfer behaviour in the page rank algorithm a model was created in [4]. In both the models described above spider traps and dead-ends were not considered In [20] also twitter a model which can identify the issuer of a topic is designed on the basis of page rank algorithm but it has many inaccurate cases as it did not consider the dead-ends. Same problems were observed in citation analysis application of page rank as observed in [9]. In [10] though dead-end problems were considered and taken care of it was became possible only with computational time as trade off.

Page rank can be used not only for ranking but other purposes as well like in [12] where page rank is used in local graph partitioning using page Rank vectors and as in [3] it can be used for social network graph analytics which is again time consuming. Scope of personalized page rank algorithm is discussed in [17]. An influence model of paper citation is created by taking attributes of PageRank and HITS algorithm is proposed in [15] has problem of starvation of new members, spider traps and dead-ends event though it improved computation time.

As explained before computation time is always a problem in page-rank so running it in parallel paradigm is a great breakthrough but link analysis algorithm cannot be fully parallelized because most of the time their result of current iteration dependent on the result of previous iterations so on parallelizing it may lead to race conditions as described in [6] where link analysis algorithms such as shortest path algorithms specifically Dijkstra's, Floyd's algorithm were parallelized. Another attempt of

parallelization of link analysis algorithm can be seen in [8] where Geometric graph problems such as minimum spanning tree are implemented parallelly.

A theoretical models of page rank in distributed environment was described in [16] where the computational time of the algorithm is expected to be improved substantially with trade off as high computation power requirement. In [13] asynchronous and distributed implementation of PageRank using stochastic approach is presented which has great improvement in execution time but spider traps and dead ends problem is not taken care off. In [14] a model is proposed with several distributed randomized schemes for the computation of the PageRank, where the pages can locally update their values by communicating to those connected by links which required use of mean square metric to calculate convergence which is computationally more complex. In [19] an overview of recent advances of distributed Randomized Algorithms for PageRank Computation is given which Introduced the problem of PageRank computation from the perspective of systems and control and provided a short overview on the recent developments on distributed algorithms.

Google cloud can be used for testing and implementation of distributed algorithms a glimpse of it is described in [5]. Performance evaluation of same is described in [18] where performance of distributed computing environments on the basis of Hadoop and Spark frameworks is estimated for real and virtual versions of clusters. Efficiency and accuracy of using search frameworks is shown.

4. Proposed Models

4.1. Reducing the running time substantially

I will be modifying and developing the serial implementation of the PageRank algorithm for it to be run in a parallel environment, which shall further be improved to a distributed implementation.

- *Serial Implementation: C++*
A normal implementation of the PageRank Algorithm in C++. This shall form a base for the various improvements that can be brought in the PageRank algorithm.
- *Parallel Implementation: C++ (using OpenMP API, a portable, scalable model that gives shared-memory parallel programming interface)*
A parallel implementation of the PageRank algorithm will reduce the running duration. The independent computations shall be modified to be able to run in parallel with other processing. I will use OpenMP application programming interface that supports multi-platform shared memory multiprocessing programming in C++.

- *Distributed Implementation : Python (using the MapReduce algorithm in PySpark)*

Spark is a general purpose distributed data processing engine. MapReduce implements various mathematical algorithms to divide a task into small parts and assign them to multiple systems.

Since the distributed implementation will require a number of remote machines running the code at once and linked through a network, I will be using Google Cloud DataProc Clusters, to create a distributed system of a master machine and three worker nodes.

4.2. Improvement Parameters:

Table 2. Scope of improvements and solutions

Drawback	Explanation	Solution Implemented
Time consuming	Page rank algorithm involves repeated execution of same steps until the result get converged which is time consuming as each iteration involves multiple operations.	To improve the execution time of the algorithm parts of the code which can be parallelized were identified and parallelized. To further improve the execution time Distributed implementation is also done.
Dead ends	Pages without out-links are called as dead ends. Effectively any page that can lead to dead end means it will lose all its page rank eventually because once a surfer reaches a page that is dead end no other page has a probability of being reached.	To solve this teleportation probability was implemented and because of this a surfer can teleport to any other website randomly.
Spider traps	These are set of pages whose out - links reach pages only from that set. When a surfer lands on this set of pages and starts clicking on links in this page he will keep on travelling in that set alone and could not get out of it because of which that particular set unusually gets more page rank.	To reduce this increasing page rank value these values were multiplied with β so even if the surfer got into the spider trap he can come out of it with probability $1 - \beta$ also known as teleportation probability

5. Math

The mathematical equations and concepts used in the algorithm formation and analysis are mentioned below along with the areas, where they are applied :

5.1. Purpose 1: *Convergence Calculation*

L1 – norm is used for this purpose. L1 – norm is the sum of the magnitudes of vectors in space.

- (1) If v_i indicates i^{th} element in vector V and u_i indicates i^{th} element in vector U then L1-Norm between V and U is $\sum_{i=1 \text{ to } n} |v_i - u_i|$

5.2. Purpose 2: *PageRank Calculation*

It gives 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.

- (2) If $PR(u) = \sum_{v \in B_u} PR(v)/L(v)$. $PR(u)$ 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 .

5.3. Purpose 3: *Taking care of spider traps and pages without out-links (dead-ends)*

The random walker follows probability distributions with a probability of β , where $\beta < 1$, and teleports to any webpage with teleportation probability of $1 - \beta$.

- (3) Calculate leakage in ranks: $\forall j \ PR(u_j) = \beta * PR(u_j)$
- (4) Calculating teleportation leakage: $x = \frac{1 - \sum_{j=1 \text{ to } N} PR(u_j)}{N}$
- (5) Adding teleportation leakage to all nodes: $\forall j \ PR(u_j) = PR(u_j) + x$

6. Implementations

6.1. Serial Implementation

The first implementation of PageRank is a serial one using C++ Programming. This algorithm and code would serve as a base for later implementing it in a parallel environment.

Each line of the 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.

NODE_LINK \longrightarrow OUT_LINK

In order to implement PageRank, I 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.

PSEUDOCODE:

Data Pre – processing procedure (reading the dataset, populating the struct variables, initialization of ranks for each node)

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  $\beta$ 
    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
```

The serial implementation was run in a machine which had a Quad-core Intel core i7 CPU with eight hardware threads and 16 GB RAM.

6.2. Parallel Implementation

After implementing the serial version of PageRank, I 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 I 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, I 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, I 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.

OpenMP is an API that supports multi-platform shared memory multiprocessing programming using languages as C, C++ and Fortran. This API uses a method called

multithreading in which a master thread distributes the work to different “slave threads” which carry their job in parallel. 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 analyse and understand how a parallel shared-memory implementation of the PageRank algorithm performs, I used OpenMP API with C++.

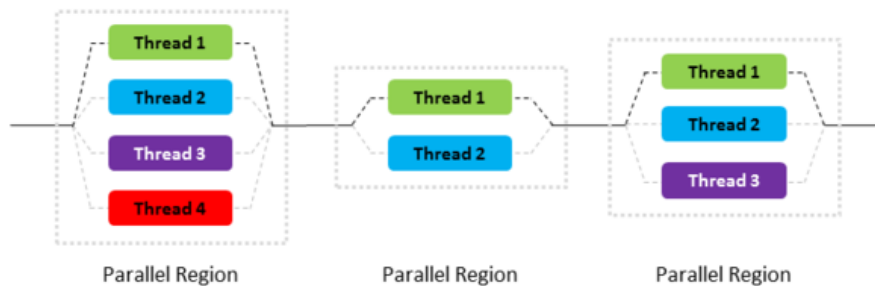


Fig 2. OpenMP : Shows how the computation is divided into different threads parallelly

PSEUDOCODE:

Data Pre – processing procedure (reading the dataset, populating the struct variables, initialization of ranks for each node)

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

The parallel implementation was run in a machine which had a Quad-core Intel core i7 CPU with eight hardware threads and 16 GB RAM.

6.3. Distributed Implementation

Map-reduce Algorithm is used in this implementation. It is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. It assigns fragments of data across the nodes in a cluster. Spark framework is a really efficient framework for Map-Reduce, and I used it to test our implementation.

Spark Framework:

Spark is a general-purpose distributed data processing engine, perfectly compatible with Map-Reduce. 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, I have used RDD's in PySpark for Map-Reduce.

PySpark is the branch of Spark that helps Python users have access to Spark's API and work with the convenient RDD-s to chunk the files properly in Map-Reduce implemented algorithms. Below are some of the most important PySpark functions that I have used

- 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.

PSEUDOCODE:

The code implementation includes 4 steps:

1. *Initial Map Reduce to prepare the data (Data Pre – processing)*
2. *Main Map Reduce Algorithm for PageRank iterations*
3. *Reinsert PageRank Leakage*
4. *Check for convergence, if not met go back to step 1*

A step – by – step explanation is shown below. The algorithm was first validated using the basic dataset as mentioned in section 2. Once the results were as desired, it was deployed to Google Cloud Dataproc Cluster to run it in a distributed environment.

Step 1: Initial Map – Reduce to Pre – Process the data

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 below transformation depicts the output of this Map – Reduce step.

DATA		PRE – PROCESSED DATA
URL neighbourURL		URL [list of neighbour URLs, size]
1 2		+---+-----+
1 3		_1 _2
1 4		+---+-----+
2 3	➔	1 {[2, 3, 4], 0, 3}
3 1		2 {[3], 0, 1}
3 2		3 {[1, 2], 0, 2}
4 3		4 {[3], 0, 1}
		+---+-----+

Fig 3. Output of Map – Reduce 1 : Pre – processing of dataset in distributed implementation

PSEUDOCODE:

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))

Step 2: PageRank Computation

The output of the above Map-Reduce function is passed as an input into this main one. The PageRanks are initialized and the following equation is applied to the dataset:

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

$$= 0, \text{ if } r_j \text{ has no inlinks}$$

The Map function will output all the out – links 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 in – links of a specific node key.

PSEUDOCODE :

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, CONTRIBUTION_LIST)
 NEW_RANK \leftarrow sum (CONTRIBUTION_LIST)
 generate (key = URL, value = NEW_RANK)

Step 3 and 4: Ending the iterations

In this step, the leakage of every node is calculated and then teleportation probability is added to every node. Finally, the convergence is calculated using following equations:

Calculate leakage in ranks: $\forall j \ PR(u_j) = \beta * PR(u_j)$

Calculating teleportation leakage: $x = \frac{1 - \sum_{j=1 to N} PR(u_j)}{N}$

Adding teleportation leakage to all nodes: $\forall j \ PR(u_j) = PR(u_j) + x$

Convergence: $\sum_{i=1 to n} |vi - ui|$

PSEUDOCODE:

Leaked PageRank Calculation Procedure

S \leftarrow sum(NEW_RANKS)

Leaked PageRank Reinsertion Procedure

for each NEW_RANK in NEW_RANKS[]
 NEW_RANK \leftarrow NEW_RANK + $\frac{1-S}{N}$

Convergence Procedure

Calculate L1-norm of Rank Vectors and check for convergence.

The above code was validated on a small dataset using Google Colab. Once ready to implement, it was deployed to run on a Google Cloud Cluster. Google Cloud Dataproc is a cloud-based managed Spark and Hadoop service offered on Google Cloud Platform. The machines of the cluster setup I 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. Outputs and comparison

Table 3 : Execution Time of the three PageRank Implementations

	Setup	Data processing	PageRank	Total
Serial	-	26.768	40.871	67.639
Parallel	-	35.293	19.592	55.885
Distributed	11.498	4.409	1.118	17.025

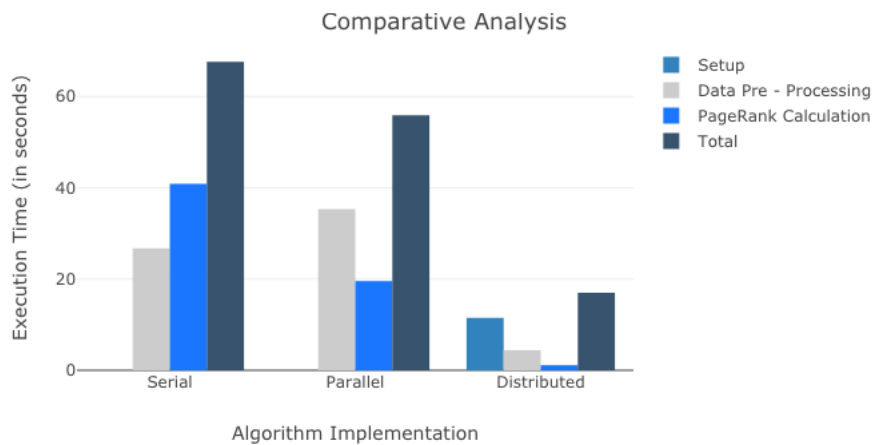


Fig 4. : A comparative analysis on the results and execution time of the above mentioned PageRank implementations

8. Adding Personalization

Above approach of PageRank generalize the rank but in order to make it more relevant to the user, personalization feature is added. In the model that I previously presented a web-surfer will be randomly clicking on the links that are available to him and when he ends up at a dead end with a certain probability known as teleportation probability he will move to any other website in world wide web, there this teleportation probability is calculated by leaking the page rank of accumulated nodes and distributing to all other nodes equally so he can end up at any page which may provide a generalized view for the browser to rank the webpages but in the perspective of a single user it may not be feasible because in the previous implementation can go from a dead-end to a completely random page which may not be even remotely related to what he is browsing. In this algorithm I have incorporated the user's perspective and the common behaviour of the user to give more personalized experience for the user. User's behaviour taken to consideration mainly in 2 ways as follows:

- When a user reaches a dead-end instead of teleporting him to a random completely unrelated webpage he will be teleported to source of this webpage. This is similar to clicking the back button in a browser when a web-surfer stuck at a point.
- Getting redirected to the home page after a session ended due to unresponsiveness or time limit reached or the user just stopped surfing.

This model runs at the user's end instead of the browser's server as it is personalized to the user and different from one user to the other, moreover it should run in the user end only as the dataset that will be used here will be built according to user's preference and needs user's browsing data which a browser server should not have access.

I have made some changes in the convergence calculation as it is going to run at the users end. If I follow the convergence method which has been done in prior models it will eat away users' computational resources because here the computational time of this convergence calculation dominates other operations so the algorithm spends more time on convergence calculation rather than producing PageRank whereas in the previous model there is teleportation probability distribution involved so it won't be affected by convergence calculation. In order to counter this problem, I have clustered several iterations into a step and at the end of each step I will be calculating convergence instead of doing it for every iteration which will not eat away much of the user's computational resources.

As different users will have different kind of behaviours and preferences and our algorithm should work for every kind of preference, I generate a random web graph taking into consideration different aspects of the user and test our algorithm on it.

Implementation of the algorithm is as follows:

Inputs: a directed graph of N nodes, source node index s , teleport probability α , maximum iteration time, maxIter , tolerance ϵ , convergence checking steps step , a function produces random numbers in interval $[0,1)$

Outputs: an array PPR for Personalized PageRank value of each node in the graph.

PSEUDOCODE :

```
if out( $p_s$ )  $\neq \emptyset$  then:
    count[] := (0,0,...)T
    countold[] := (0,0,...)T
    i := s
    for (iter:=0 to (maxIter-1) :
        if random() <  $\alpha$  or out( $p_i$ ) =  $\emptyset$  then:
            j := s
            i := j
        else:
            choose a node  $p_i$  in out( $p_i$ ) randomly
        end if
        count[j] := count[j] + 1
        i := j
        if iter+1  $\equiv$  0(mod step) then:
            if iter+1 > step then:
                r = sampleCorrelationCoefficient(countold,count)
                if r  $\geq$  (1- $\epsilon$ ) then:
                    iter:=iter+1
                    break
                end if
            end if
            countold[] = copy(count[])
        end if
    end for
    PPR[] := ( $\frac{1}{\text{iter}}$ ) * count[]
else:
    PPR[] := (0,0,...)T

return PPR[]
```

9. Future Research Directions and Conclusion

The PageRank algorithm was implemented and improvised to overcome the shortcomings related to the dead – ends, spider – traps and time of execution when large datasets are involved. As expected, I observed that the parallel implementation takes shorter amount of time than the serial one. Also, I saw that the distributed implementation using Map-Reduce is actually really efficient compared to Parallel implementation. Also, data pre-processing is done faster in Spark. There is a trade-off for this result, as Spark requires its own setup time. However, in terms of overall running time, the Distributed implementation performs the best. The output of each implementation is shown in Appendix A.

Finally, a personalised variant of the PageRank algorithm was also implemented, and I found it to be an apt solution. The personalised PageRank is biased towards a set of nodes, and can be used in user's browsers, recommender systems to provide customized and personalised results to users based on his profile from past experiences. Future work on this algorithm can be done, by embedding the code with the user profile, collecting user specific data and providing it as an input to the algorithm. Several other data mining, deep learning, machine learning methods may also be applied together for predictive analysis.

References

1. Gleich, D. F. (2015). PageRank beyond the web. *SIAM Review*, 57(3).
2. Luo, S. (2019). Distributed PageRank computation: An improved theoretical study. 33rd AAAI Conference on Artificial Intelligence, AAAI 2019, 31st Innovative Applications of Artificial Intelligence Conference, IAAI 2019 and the 9th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019.
3. Kloumann, I. M., Ugander, J., & Kleinberg, J. (2017). Block models and personalized PageRank. *Proceedings of the National Academy of Sciences of the United States of America*, 114(1).
4. Park, S., Lee, W., Choe, B., & Lee, S. G. (2019). A Survey on Personalized PageRank Computation Algorithms. *IEEE Access*, 7.
5. Page, L., Brin, S., Motwani, R., & Winograd, T. (1998). The PageRank Citation Ranking: Bringing Order to the Web. *World Wide Web Internet And Web Information Systems*, 54(1999–66).
6. Ma, N., Guan, J., & Zhao, Y. (2008). Bringing PageRank to the citation analysis. *Information Processing and Management*, 44(2), 800–810.
7. Eirinaki, M., & Vazirgiannis, M. (n.d.). Web Mining for Web Personalization.

8. Yan, E., & Ding, Y. (2011). Discovering author impact: A PageRank perspective. *Information Processing and Management*, 47(1), 125–134.
9. Taran V. & Rojbi A. (2017). Performance Evaluation of Distributed Computing Environments with Hadoop and Spark Frameworks.
10. Amrani, A. (n.d.). PageRank and Personalized PageRank applied on sparse graphs using parallel computing.
11. Maad, K. M., & Sh, M. M. (n.d.). Mastering Google cloud: building the platform that serves your needs.
12. Lu, Y., Ma, K., & Duan, J. (2021). Influence Model of Paper Citation Networks with Integrated PageRank and HITS. *Proceedings of the 2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design, CSCWD 2021*, 1081–1086.
13. Ishii, H., & Suzuki, A. (2018). Distributed randomized algorithms for pagerank computation: Recent advances. In *Systems and Control: Foundations and Applications* (pp. 419–447). Birkhauser.
14. Ishii, H., & Tempo, R. (2010). Distributed randomized algorithms for the PageRank computation. *IEEE Transactions on Automatic Control*, 55(9), 1987–2002.
15. Huang, B., Liu, Z., & Wu, K. (2020). Accelerating pagerank in shared-memory for efficient social network graph analytics. *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS, 2020-December*, 238–247.
16. He, Y., & Wai, H. T. (2021). Provably fast asynchronous and distributed algorithms for pagerank centrality computation. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings, 2021-June*, 5050–5054.
17. Andoni, A., Nikolov, A., Onak, K., & Yaroslavtsev, G. (2014). Parallel algorithms for geometric graph problems. *Proceedings of the Annual ACM Symposium on Theory of Computing*, 574–583.
18. Awari R. (2017). Parallelization of Shortest Path Algorithm Using OpenMP and MPI
19. Ashyralyyev S., Cambazoglu B., Akyanat C., Incorporating the surfing behaviour of Web Users into PageRank.
20. Andersen, R., Chung, F., & Lang, K. (2006). Local Graph Partitioning using PageRank Vectors.

Appendix A

Output Screenshots of the Implementations :

A.1. Serial Implementation :

```
PS C:\Users\Rushabh Kela\Desktop\DM_PROJECT> g++ -fopenmp Serial_PageRank.cpp
PS C:\Users\Rushabh Kela\Desktop\DM_PROJECT> .\a.exe "PageRank_Dataset.txt" 3774768 0.0001 0.85
Preprocessing time: 26.7680001259 seconds.
Difference in iteration 1 = 0.744733.
Difference in iteration 2 = 0.393814.
Difference in iteration 3 = 0.203311.
Difference in iteration 4 = 0.104611.
Difference in iteration 5 = 0.055330.
Difference in iteration 6 = 0.029810.
Difference in iteration 7 = 0.016209.
Difference in iteration 8 = 0.008835.
Difference in iteration 9 = 0.004803.
Difference in iteration 10 = 0.002598.
Difference in iteration 11 = 0.001399.
Difference in iteration 12 = 0.000752.
Difference in iteration 13 = 0.000403.
Difference in iteration 14 = 0.000216.
Difference in iteration 15 = 0.000116.
Difference in iteration 16 = 0.000062.

Number of iterations: 16.
Elapsed time: 40.8710000515 seconds.
Writing nodes and corresponding pagerank to outputSerial.txt
Output Complete.
```

Fig 5 : Terminal Output of Serial Implementation

```
≡ outputSerial.txt X
≡ outputSerial.txt
1 Node 3858241 has pagerank value : 1.38212e-07
2 Node 956203 has pagerank value : 6.3242e-08
3 Node 1324234 has pagerank value : 7.68802e-08
4 Node 3398406 has pagerank value : 6.94457e-07
5 Node 3557384 has pagerank value : 4.96332e-07
6 Node 3634889 has pagerank value : 4.25723e-07
7 Node 3858242 has pagerank value : 4.05683e-07
8 Node 1515701 has pagerank value : 1.25944e-07
9 Node 3319261 has pagerank value : 2.83399e-07
10 Node 3668705 has pagerank value : 4.90197e-07
11 Node 3707004 has pagerank value : 8.02569e-07
12 Node 3858243 has pagerank value : 2.03687e-07
13 Node 2949611 has pagerank value : 9.56577e-08
```

Fig 6 : All Nodes and corresponding PageRanks written into a file

A.2. Parallel Implementation :

```
PS C:\Users\Rushabh Kela\Desktop\DM_PROJECT> g++ -fopenmp Parallel_PageRank.cpp
PS C:\Users\Rushabh Kela\Desktop\DM_PROJECT> .\a.exe "PageRank_Dataset.txt" 3774768 0.0001 0.85
Preprocessing time: 35.2929999828 seconds.
Difference in iteration 1 = 0.744733.
Difference in iteration 2 = 0.393814.
Difference in iteration 3 = 0.203311.
Difference in iteration 4 = 0.104611.
Difference in iteration 5 = 0.055330.
Difference in iteration 6 = 0.029810.
Difference in iteration 7 = 0.016209.
Difference in iteration 8 = 0.008835.
Difference in iteration 9 = 0.004803.
Difference in iteration 10 = 0.002598.
Difference in iteration 11 = 0.001399.
Difference in iteration 12 = 0.000752.
Difference in iteration 13 = 0.000403.
Difference in iteration 14 = 0.000216.
Difference in iteration 15 = 0.000116.
Difference in iteration 16 = 0.000062.
Difference in iteration 15 = 0.000116.
Difference in iteration 16 = 0.000062.

Number of iterations: 16
Elapsed time: 19.5929999352 seconds.
Writing nodes and corresponding pagerank to outputParallel.txt
Output Complete.
```

Fig 7 : Terminal Output of Parallel Implementation

```
≡ outputParallel.txt ×
≡ outputParallel.txt
1 Node 3858241 has pagerank value : 1.38212e-07
2 Node 956203 has pagerank value : 6.3242e-08
3 Node 1324234 has pagerank value : 7.68802e-08
4 Node 3398406 has pagerank value : 6.94457e-07
5 Node 3557384 has pagerank value : 4.96332e-07
6 Node 3634889 has pagerank value : 4.25723e-07
7 Node 3858242 has pagerank value : 4.05683e-07
8 Node 1515701 has pagerank value : 1.25944e-07
9 Node 3319261 has pagerank value : 2.83399e-07
10 Node 3668705 has pagerank value : 4.90197e-07
11 Node 3707004 has pagerank value : 8.02569e-07
12 Node 3858243 has pagerank value : 2.03687e-07
13 Node 2949611 has pagerank value : 9.56577e-08
14 Node 3146465 has pagerank value : 2.39468e-07
```

Fig 8 : All Nodes and corresponding PageRanks written into a file

A.3. Distributed Implementation :

Spark setup time is: 11.498010635375977 s.

Lines preprocessing time is: 4.40936017036438 s.

End of iteration #1

End of iteration #2

End of iteration #16

PageRank Loop time is: 1.1180577278137207 s.

4283599 has rank: 3.9188848094973355e-06.
3677886 has rank: 1.2841189722203754e-06.
4219693 has rank: 8.353010969348107e-07.
5660048 has rank: 1.8710290120907917e-06.
4719761 has rank: 3.4636241633048204e-07.
5238052 has rank: 2.6259305597977385e-07.
4590124 has rank: 1.267630385172966e-06.
4679895 has rank: 1.4232718422053252e-06.
3764030 has rank: 3.328641768241835e-07.

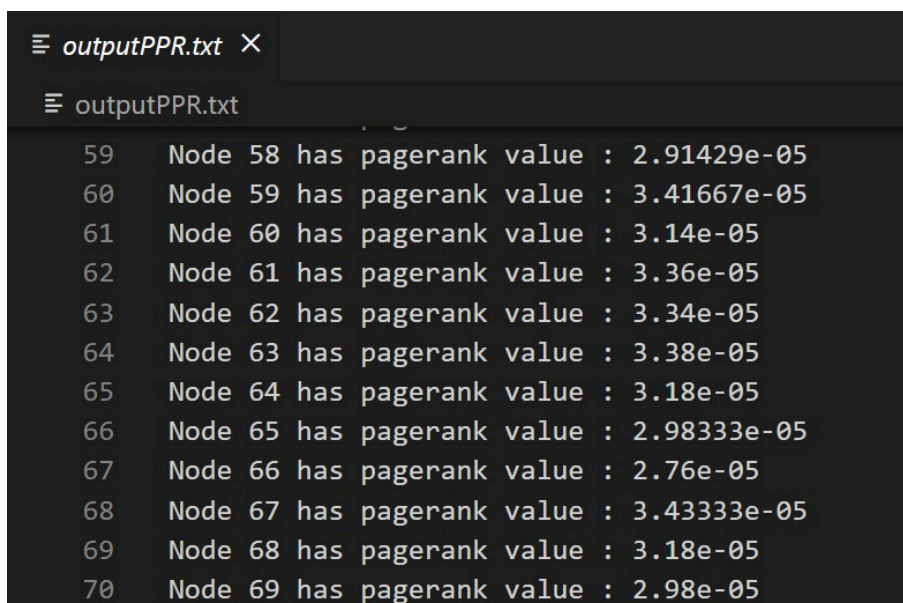
Fig 9 : Google Cloud Console Output of Distributed Implementation

A.4. Personalized PageRank Implementation :

```
PS C:\Users\Rushabh Kela\Desktop\DM_PROJECT> g++ PPR.cpp
PS C:\Users\Rushabh Kela\Desktop\DM_PROJECT> .\a.exe
Graph Initialized
Number of nodes: 10000
Number of edges: 94680
Average number of edges: 9.47
Total iteration time: 5164200
Average iteration time: 516.42
Total execution time: 7864.00 ms

Writing nodes and corresponding pagerank to outputPPR.txt
Output Complete.
```

Fig 10 : Terminal Output of the Personalized PageRank Algorithm



```
59 Node 58 has pagerank value : 2.91429e-05
60 Node 59 has pagerank value : 3.41667e-05
61 Node 60 has pagerank value : 3.14e-05
62 Node 61 has pagerank value : 3.36e-05
63 Node 62 has pagerank value : 3.34e-05
64 Node 63 has pagerank value : 3.38e-05
65 Node 64 has pagerank value : 3.18e-05
66 Node 65 has pagerank value : 2.98333e-05
67 Node 66 has pagerank value : 2.76e-05
68 Node 67 has pagerank value : 3.43333e-05
69 Node 68 has pagerank value : 3.18e-05
70 Node 69 has pagerank value : 2.98e-05
```

Fig 11 : All Nodes and corresponding PageRanks written into a file

Fig 10 : Google Cloud Console Output of Distributed Implementation