Page Rank Algorithm Using MPI

Indiana University, Bloomington

Fall-2012

A project report submitted to Indiana University

By

Shubhada Karavinkoppa and Jayesh Kawli

Under supervision of Prof. Judy Qiu

Table of Contents

1. Abstract

PageRank algorithm is used by Google search engine to measure the relative importance of the web pages on the web. It assigns a numerical weighting to each of the pages in the set. The PageRank of a page depends on the number pages that link to it i.e. every inbound link to a page increases its PageRank value.

This is the second part of page rank sequential algorithm. This is implemented as parallel version of previous part using MPJ library which is also called as open source java message passing library. If the application code is written in suitable way, this library can be used to execute pagerank algorithm in parallel manner. Although not so visible for small input file, parallel implementation makes pagerank calculation efficient in terms of both memory and execution time.

2. Introduction

In this project we have implemented MPI PageRank algorithm. The algorithm calculates the PageRank value of each page depending on the inbound links to that page. Each inbound link is like a vote. These votes are used to determine which pages are more important.

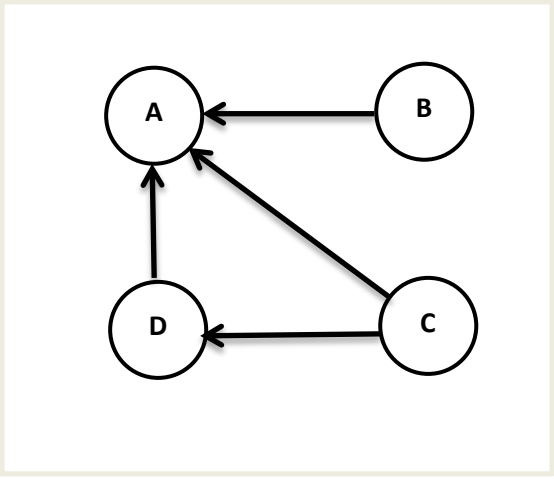
We implemented this algorithm using MPJ open source library. This algorithm is revision of the first part in which page rank was implemented in a sequential manner. We used java for parallel page rank implementation because it is considered as an excellent language for running high performance parallel applications on grid and clusters.

3. Problem Statement

We are given with an input file that contains 1000 pages along with the list of pages to which they are linked. This is similar to adjacency matrix. The aim of this project is calculate PageRank value of each page by taking into account dangling nodes and determine the ten most popular pages using java MPJ libraries for parallel processing.

4. Theory

PageRank calculation is based upon the graph of web where each webpage is like a node and each hyperlink is like an edge. Consider a web graph of 4 nodes A, B, C and D as shown in below figure. Initial PageRank values are assigned using probability distribution between 0 and 1 i.e. 1 divided by total number of web pages. In the following example, the initial PageRank for each page is 0.25.



Suppose the nodes B and D have links to A and C has links to A and D as shown in figure 1.1, then the PageRank of A is calculated as,

PR(A) = PR(B)/1 + PR(C)/2 + PR(D)/1

In general, the PageRank of any page u is given as,

$$PR(u) = \sum_{v \in Set} \frac{PR(v)}{L(v)}$$

The PageRank of u is dependent on the PageRank value of each page v in the Set (Set containing all nodes that have outbound link to u) divided by the number outbound links from page v.

The Damping Factor is taken into account when we consider that an imaginary surfer who is randomly clicking on the links will eventually stop. The probability that the person will continue is a damping factor and it is generally set to 0.85.

The PageRank formula considering damping factor is given as,

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

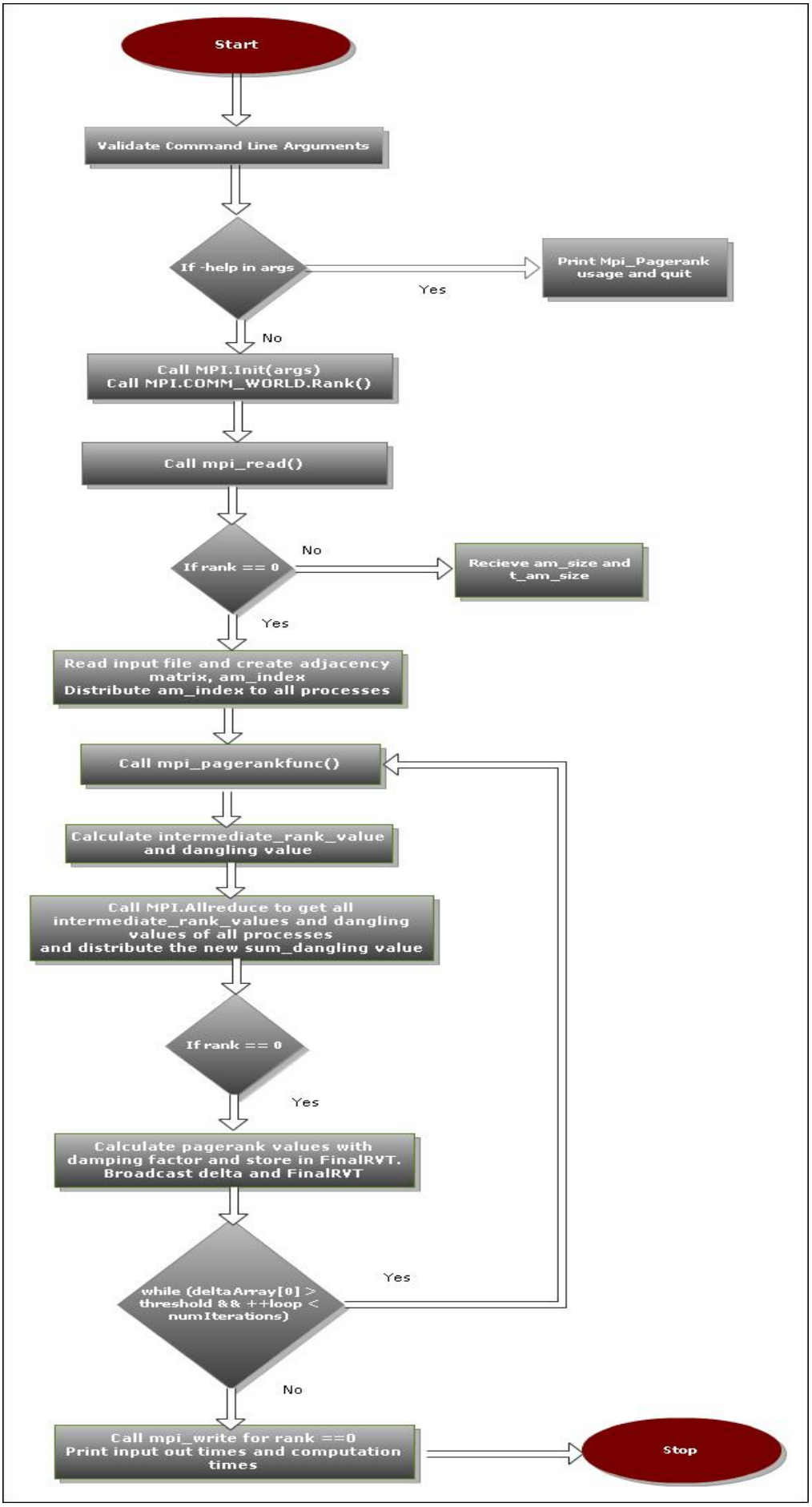
Dangling Links: Dangling link is simply a link to a page that does not have any outgoing links i.e. it does not link to any other page. These dangling links do not affect the PageRank value of any other page directly. Such links are kept aside from the system until PageRank values of all other pages are calculated. After all the PageRank values are calculated, they can be added back in without affecting significantly. The PageRank of all such links are added and then divided by the total number of webpages and this result is equally distributed among all the webpages to minimize the effect of dangling links.

In our project, we have used the MPI interface. Following are some of the MPI communicator objects and functions that are being used in our project.

- MPI.*Init(args):*Passes program inputs to MPI.
- MPI.*COMM_WORLD.Rank():* Obtains the rank (id) of the current process.
- MPI.*COMM_WORLD.Size():* Gets the total number of processes
- MPI.*Finalize():*Terminate and finalize the MPJ program
- *MPI.COMM_WORLD*.Send(): Sends user-specified data which has been split by the master process from master to destination rank processes
- *MPI.COMM_WORLD*.Recv(); Receives data from other ranks processes.
- MPI_Allreduce(): This function combines values from all processes and distributes the result back to all processes.
- MPI_Bcast(): Broadcasts a message from the root process to all other processes.

5. Architecture

The diagram below shows the flowchart of program.


Start
Validate Command Line Arguments
If -help in args
Yes
Print Mpi_Pagerank usage and quit
No
Call MPI.Init(args)
Call MPI.COMM_WORLD.Rank()
Call mpi_read()
If rank == 0
No
Recieve am_size and t_am_size
Yes
Read input file and create adjacency matrix, am_index
Distribute am_index to all processes
Call mpi_pagerankfunc()
Calculate intermediate_rank_value and dangling value
Call MPI.Allreduce to get all intermediate_rank_values and dangling values of all processes
and distribute the new sum_dangling value
If rank == 0
Yes
Calculate pagerank values with damping factor and store in FinalRVT.
Broadcast delta and FinalRVT
while (deltaArray[0] > threshold && ++loop < numIterations)
Yes
No
Call mpi_write for rank ==0
Print input out times and computation times
Stop

6. Implementation

6.1 Reading command line input

We run page rank algorithm by giving string of input parameters though command line interface. These parameters are briefly described below.

As per the MPJ standard, first two commands thus given which specify how many processes pagerank algorithm should be implemented with.
This is specified as
-np [Number of processes to create]

args[3] = Name of the input file containing list of URLs and its associated nodes
args[4] = Name of the output text file which stores Top 10 URLs along with their page rank value and sum of all N page rank values.
args[5] = Specifies total number of iterations to be done. However in implementation, this is not a strict requirement. Algorithm will automatically stop when convergence occurs.
args[6] = Specifies threshold value which determines convergence condition. In each iteration we take sum of absolute difference between current and previous page rank values for each page. When this difference becomes less than or equal to this threshold value, convergence is said to have occurred and algorithm stops iterations.

Additional parameters

-o =Used to output timing results. This includes time taken by reading URL file into adjacency matrix, time taken to process all the URLs to give respective page rank value and finally time taken to write list of top 10 URLs in output file. By default this option is set true.

-d=Enables debug mode. Used for debugging purpose. When debug mode is set, program will write details of each and every activity on a console. If there is an error in the program logic, lot of time can be saved by enabling debug mode and going through the program flow data thus produced.

-help=Used to print help information. Explains how to input command line arguments. Also gives details of additional parameters to analyse program behaviour.

Note: Threshold functionality of an algorithm is to improve its time complexity. When user gives number of iteration as any nonzero positive value, it will not necessarily be calculating page ranks for those numbers of iterations. When there is no longer much difference in current and previous page rank values, calculations will stop. At this point convergence is said to have occurred.

6.2 Read the input file of given URLs

A name of the input file is given from command line input. This is the input file containing N URLs. For each node it is provided with the list of nodes to which is connected through outbound links. A node may or may not have an outbound link. In case node doesn't have an outbound value, we treat that node as dangling node and give special treatment in page rank calculation.

We use java functions FileInputStream, DataInputStream to create object of input file. Once object is created, file is read until null pointer is encountered. Since file might contain very large number of URLs, we use MPJ library functions to perform input file operations. MPJ function 'send' is used to send split data to various processes and 'receive' to accept data from all the other running processes. Input URLs are stored in adjacency matrix format along with the nodes to which is connected through outbound links.

In order to take advantage of parallel capabilities provided by MPJ, we divide given URLs among n processes, process on them and then combine the result.

6.3 Storing Graph of web pages in a Linked Hash Map

Once file is successfully read, we store all the nodes along with its neighbours in a LinkedHashMap data structure. Each node is stores as key and its associated successors are stored in a Arraylist data structure which acts as a LinkedHashMap value. One advantage of using LinkeHashMap is that, it stores the value in an order in which it has read from input file.

6.4 Number of URLs in input file

We count the number of URLs in the given input file by querying on size of HashMap which stores all the input URLs as a key. This value allows us to deal with dangling node issue and also results in close value of page rank for each web page.

6.5 Initializing Page rank values table

At the very first iteration of loop, page rank values of web pages are not known. As a part of initialization we set page initial page rank value as 1/N for all the input URLs. Where N is a total number of URLs.

6.6 Calculating actual page rank value

We will call the function

mpi_pagerankfunc(HashMap<Integer,ArrayList<Integer>>adjacencyMatrix,ArrayList<Integer>amIndex, int numUrls, int totalNumUrls, int numIterations, double threshold, double[] FinalRVT, Intracomm communicator, int debug)

Where,

FinalRVT – Array which stores final value of page rank for all the URLs in sequential manner

adjacencyMatrix –HashMap to store URLs along with their successors in the ArrayList data read from input file using MPJ libraries

amIndex – Has total N entries. Each entry is composed of pair of parameters stored consecutively in an arraylist data structure. First parameter is URL and second parameter specifies how many outbound node current URL has.

totalNumUrls –Total number of input URLs read from input file

numIterations – Number of page rank iterations to perform. Each iteration successively inrements page rank accuracy. However, this is not a strict requirement. We set convergence condition based on a input threshold. Algorithm stops when this condition is reached.

threshold – Used to specify stopping criteria for convergence. Small threshold value results in more number of iteration, but it increases accuracy of final result.

communicator – Specifies communicator in which processes are running.

debug – Optional command line parameter. When set prints detailed information of each activity on console.

This function will calculate page rank value for each node in successive iteration using following formula

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

We first calculate intermediate page rank value for each URL and distribute average value of dangling URLs to each web page. We multiply this summation by damping factor d to reduce the pagerank of predecessor. In short term, when speaking of overall effect, a page loses its page rank by some extend when it has an outbound link to another page.

Once this is done, we add an additional term which corresponds to the probability that random user will click a page only given the total number of input URLs.

6.7 Number of Iterations

We give the input of how many iterations we require to do to get final and most clean value of page ranks for each page. However, code does not necessarily iterate for given number of iterations. Code is designed to be more efficient in sense that it doesn't have to do every iteration and sometimes convergence occurs beforehand if numbers of input iterations are large.

Small value of iteration may give wrong results while large value might give more correct results but with more computation time.

6.8 Writing results to output file

After convergence occurred (Or loop has executed specific number of times) we write the results of computation in output file which includes,

A. List of Top 10 URLs along with their page rank values
B. Sum of all page rank values (Ideally equal to one)

We do not use MPJ libraries for parallel writing in output file. Since number of URLs to be written are only 10. Using MPJ libraries will bring unnecessary program overhead.

6.9 Optimizations and Efficiency boosting

Break the loop if computation converges to final solution.
Instead of going over for loop for number of iterations given from command line, we try to limit the iterations until convergence happens.

Each time page rank values are calculated, we compare them against earlier page rank computations. If this difference goes below specific threshold (Set in the program) we break the loop and return results of iteration in output file.

This algorithm is highly optimized compared to first sequential implementation. We divide input file data into number of subdivisions which is equal to number of processes to run in parallel. We then assign fixed number of input URLs to each subdivision. If the total number of URLs is not a multiple of number of input processes, we adjust remaining processes into last subdivision. So last subdivision might contain more number of URLs than rest ones. All the processes run concurrently passing message between them using 'Send' and 'Recv' MPJ APIs.

We use MPJ library function 'Bcast' to broadcast messages from root process to all the other n processes created which are responsible for individual page rank evaluation. 'Allreduce' is used to combine values from all processes and sending final result back.

6.10 Timing details

To get more insight regarding runtime behaviour of program, we introduced frequent checkpoints to get estimate of time required to execute that function. We use this information to make our program more efficient in terms of both time and space.

Timing details were added at three checkpoints

1. Reading list of URLs from input file and storing them in adjacency matrix
2. Calculating page rank value using MPJ libraries for n simultaneous processes
3. Writing top 10 URLs in output file along with their respective page rank values

7. Discussion:

7.1 Result

List of Top 10 URLs was accumulated in output file along with their page rank value As follows

Top 10 URLs with Highest Page Rank values

URL	Page Rank
4	0.13809182637339654
34	0.12293720129122925
0	0.11259168048995331
20	0.07746633682627928
146	0.05723789230252064
2	0.04795732052423655
12	0.02010149462956356
14	0.01791527501372222
16	0.01303327335882033
6	0.01294396513791297

Cumulative Sum of Page Rank values

0.9999999999999813

7.2 Performance Analysis

1. We also calculated time taken to perform input output operations. Read input file with list of 1000 URLs. After page rank calculation, top 10 URLs were written in output file in descending order of page rank values. Sum of page rank value of all the pages was also written as a part of output results.
2. Total Time taken for input/output operations was evaluated to be 616 milliseconds.
3. Computation time was also calculated when page rank data was divided in n processes. Used MPJ libraries to divide input URL data among n processes and collect and distribute result back. Total computation time was 88 milliseconds.

7.3 Findings

Thus we successfully implemented page rank algorithm which gives the top 10 URLs in terms of the page rank from given list of N input URLs. Following put forward our findings as follows

A. Program considers both regular nodes and dangling nodes with no outbound link
B. When page rank values of dangling nodes are ignored, we were still able to get expected list of top 10 URLs. However sum of observed page rank values thus obtained was much below 1 (0.75)
C. When we considered dangling nodes into our calculation, sum of 0.9999999999999813 was obtained which is very close to ideal value (1).
D. It was observed that, it is not at all necessary to run page rank algorithm for any number of times because in some cases if node configuration is ideal, convergence occurs even before specified number of iterations.
E. For fixed values of threshold and iteration count of 50, convergence occurred only at 14^{th} iteration.
F. We get good approximation of page rank algorithm after algorithm is ran for fixed number of times when convergence happens
G. Minimum value of page rank could be (d-1)/N according to given formula when it has no inbound links
H. Maximum value of page rank can be ((d-1)/N)+(d*(N-1)) when all remaining (N-1) nodes point only to current node
I. If the page has more number of inbound links, then it results in increasing page rank of that page which is evident from given page rank calculation formula

We calculated cumulative page rank value going through each iteration. It was observed that, this value always remains close to one

8. Assumptions:

To maintain fairness in page rank calculation, we introduce following assumptions in algorithm implementation

A. For the given pair of web pages, one page cannot have more than one outbound links on the same page
B. Likewise, one page cannot have more than one inbound links from same page
C. Any web page cannot have outbound link pointing to itself

Assumptions mentioned above are necessary, as without this constraint it will be possible for any website administrator to exploit page links to elevate overall page rank in unfair manner.

9. Conclusion and Future Work

A. These algorithms when tested on file containing N=1000 URLs, gave excellent results in terms of final list of top 10 webpages. However, behaviour of this project is still unknown when relatively large input file (N>100000) is given. We will try to study its behaviour on large inputs and check its performance.
B. Next part of this project is to run parallel page rank version on Multi core Supercomputer Future grid provided by Indiana University and analyse its performance where it actually utilizes multiprocessor power for which it was designed.

10. Acknowledgements

We would like to thank Prof. Judy Qiu, Assistant Professor of Computer Science and Informatics at School of Informatics and Computing at Indiana University for helping us to develop an idea of this wonderful algorithm. Presentation slides of lecture notes and lab material provided on course web page helped us to get quick start on this project as we got deep insight about how MPJ library could be utilised to boost page rank algorithm performance.

We also appreciate the help from Ila Jogaikar, for her patience and co-operation throughout this project.

We would also like to thank all those anonymous people all around the internet, who shared their knowledge and experience on various blogs and webpages. This helped us to know background and current implementation of the page rank algorithm. Finally appreciation to all those developers who helped creating MPJ library.

11. References

http://en.wikipedia.org/wiki/PageRank
http://www.webworkshop.net/pagerank.html
http://www.sirgroane.net/google-page-rank/
http://pr.efactory.de/e-pagerank-algorithm.shtml
http://infolab.stanford.edu/~backrub/google.html
http://mpj-express.org/