CS 442 Project Final Report

Group 11
Alp Ege Baştürk 21501267
Aldo Tali 21500097
Oğuz Kaan Ağaç 21503895

Introduction

In this project, we have implemented Google's Pagerank algorithm using MPI in C++. Pagerank is an algorithm used to rank node importances based on links of the graph. We have implemented a power iteration based Pagerank using sparse matrix-vector multiplication in which dead ends and spider traps are considered using teleportation.

Pagerank operation depends on $r = \beta M \cdot r + \left[\frac{1-\beta}{N}\right]_N$ calculation for rank vector, where β is the transition probability. This matrix operation is iterated until convergence or maximum iteration number is reached. Matrix-vector multiplication is a well known problem and it can be distributed to different nodes for calculation. We implemented the matrix as a sparse representation using adjacency lists to decrease memory usage as much as possible. In addition, the proposed implementation of the distributed Pagerank takes into consideration the leakage of ranking probabilities from the dead ends and spider traps that are commonly present in real world graphs [1]. In order to re-insert leaked pagerank values the formula was modified to $r = \beta M \cdot r + \left[\frac{1-S}{N}\right]_N$ where S is $\sum_i r_i$ [1].

Algorithm:

$r^{old} = [1/N]_N$

$Do:$

$r^{new} = \beta M r^{old} + [\frac{1-S}{N}]_{NxN}$

$r^{old} = r^{new}$

$Until\ Convergence$

1.0 Requirements

The implementation of this proposal is fully based on the MPI abstraction of the Message Passing Paradigm. In order for the algorithm to be working properly a MPI environment is required to be available on all the computation nodes of the distributed network (any number of distributed CPU's). The current implementation is based on the stable MPI version 4.0.1, meaning that any version starting at this point or higher is advised.

In order to compile the code in all the devices a valid g++, mpic++ compiler is required and at the time of the implementation C++11 was used to write the algorithm as a whole. Consequently to achieve the results described in this report the same compiler is advised to be used. As any other regular MPI implementation of parallel workload computation, the number of computation nodes/processes is required to be specified on runtime using the -np flag. Having the number of processes supplied by the users allows for a better distribution of the work division and the mapping of loads to physical nodes.

The last requirement for this project is the structure of the webgraph that is to be computed. The graph is to be received as a text file that may include comments as long as these comments start with the symbol "#". The graph itself is encoded as a tab separated nodeID to outgoing nodeID pairs. Each line contains one such pair. A representation for these can be found on the datasets used.

2.0 Internal design & Implementation Details

Initially we have implemented a parser which reads graphs in edge list format, and creates a custom graph data structure.

Graph Structure

Our graph structure has a map to map given IDs to continuous list of ids, a vector of vertex ids and adjacency list to store the graph in a sparse matrix representation. Adjacency list is implemented as a dynamic array of dynamic arrays (i.e standard library vectors of vectors). Given a node 'V' the graph will hold V's neighbours and its outdegree in order to facilitate

Pagerank computation. This map and vector of vertex ids were necessary so that we can parse any GML file that we want.

PageRank Implementation

Main function starts with creating the MPI environment. Master node reads the graph by calling the parser. Parser creates and returns one of our custom graph structure for the given data file. Master broadcasts number of nodes, then it writes out degrees of nodes into a vector and broadcasts it too. Out degrees are required to divide matrix values during calculation. These are required for all nodes so we have broadcasted them to prevent message passing during calculations. In addition master generates a vector of individual sizes of lists in the adjacency list representation, given that only master knows them at this stage and this information will be useful later. This vector is broadcasted to other nodes since it is a small 1D vector which will help reducing number of messages later.

Later, master reads and distributes common information, all nodes calculate partition sizes and their displacements in the adjacency list structure. This calculation is redundant and master could do that however these are simple calculations and we concluded that it is more efficient to do them rather than passing messages. Also MPI_Gatherv() function can use this information directly to make implementation more robust.

After setting up common information, master initializes pagerank by broadcasting a flag to other nodes. This flag is broadcasted at each iteration. Non-master nodes wait for this flag before continuing with the pagerank iteration and stop if it is the sentinel.

performIteration(...) method is called for each iteration, which implements the aforementioned algorithm. Then each node calculates sum of its part. These values are reduced to all nodes which adds $\left[\frac{1-S}{N}\right]$ to their rank values in order to compensate for the leaks. Later partial rank vectors are gathered on the r_old of master using GatherV. This completes an iteration.

3.0 Experiments & Results

Experiments were done on Ubuntu 18.04 with mpich library. Test machine had Intel i7-4720HQ CPU with 4 cores at 2.6 GHz. 2.6 GHz was fixed by manually setting the frequency using cpu_freq. Python 3.7 was used for comparison. Memory usage was measured by subtracting use before execution and during execution, no new programs were executed during the execution to stabilize the results, however analysis on the memory performance was limited since precision could be a problem. We have compared results of our MPI implementation with our serial implementation in order to validate correctness.

Data Sets

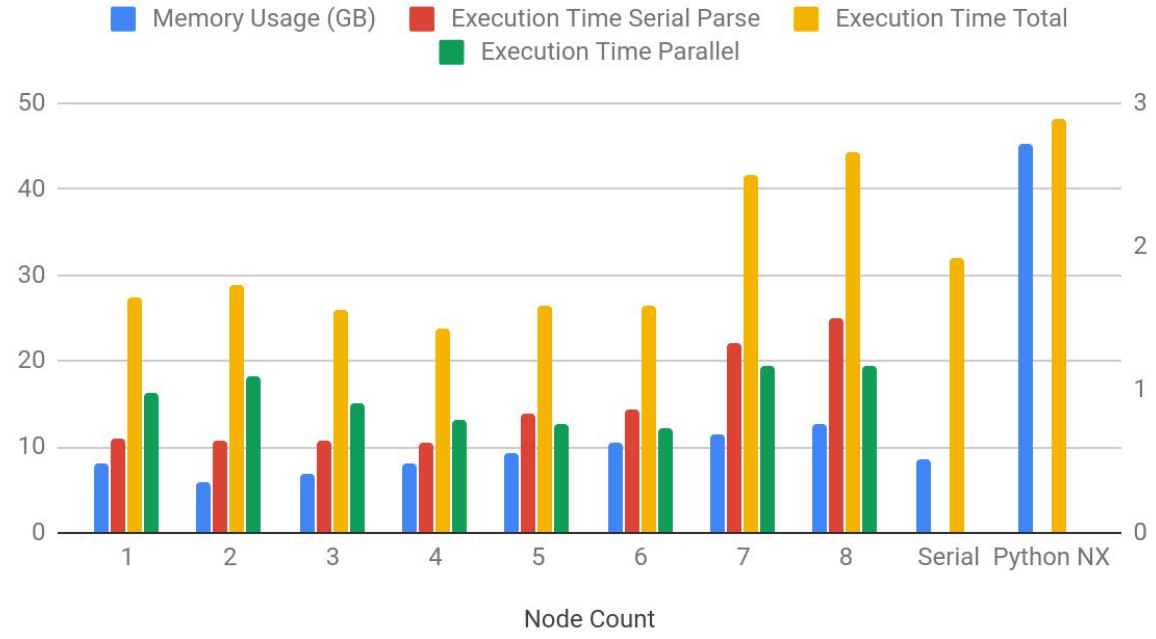
Following datasets from Stanford SNAP [2] were used for testing with real graphs; web-Google [3], [4] and Soc-LiveJournal1 [3], [5]. web-Google dataset has 875,713 nodes, 5,105,039 edges and soc-LiveJournal1 has 4,847,571 nodes, 68,993,773 edges.

Execution Time Analysis

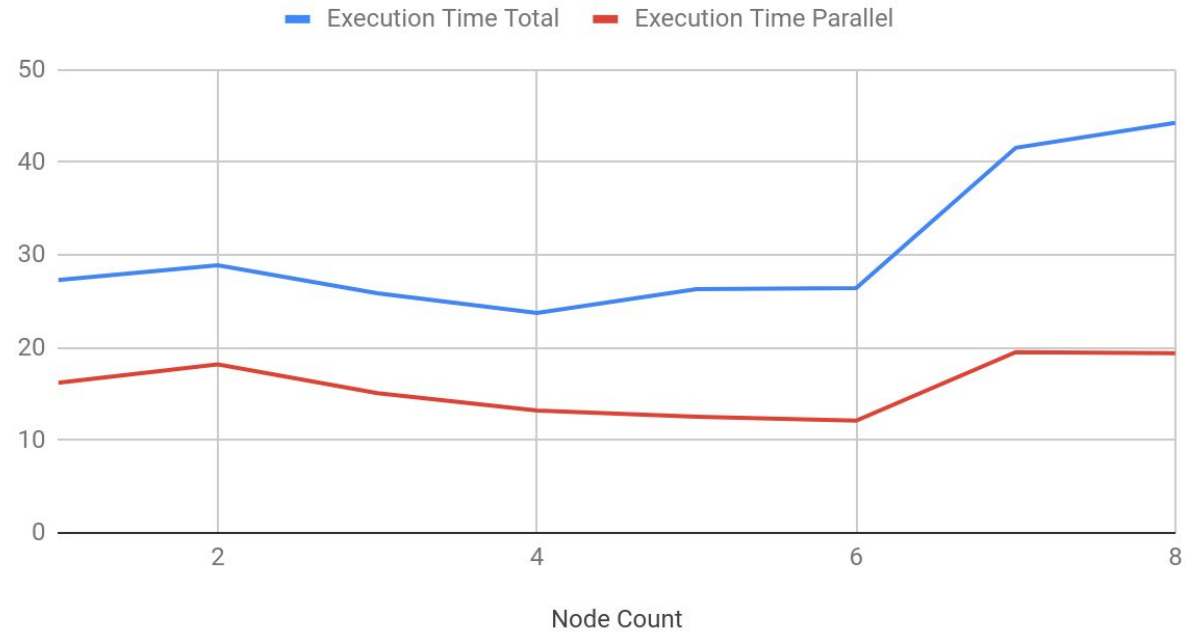
Following table shows execution data of our implementation. Also we have data for Python Networkx library test for comparison.

web-Google.txt				
Node Count	Memory Usage (GB)	Execution Time Serial Parse	Execution Time Total	Execution Time Parallel
1	0.48	11.081595	27.297761	16.216166
2	0.35	10.712839	28.897591	18.184752
3	0.41	10.794633	25.87791	15.083277
4	0.48	10.527559	23.738345	13.210786
5	0.56	13.762895	26.308949	12.546054
6	0.63	14.309938	26.418219	12.108281
7	0.69	22.063021	41.570934	19.507913
8	0.76	24.872617	44.278853	19.406236
Serial	0.51		32	
Python NX	2.71		48.17290401	

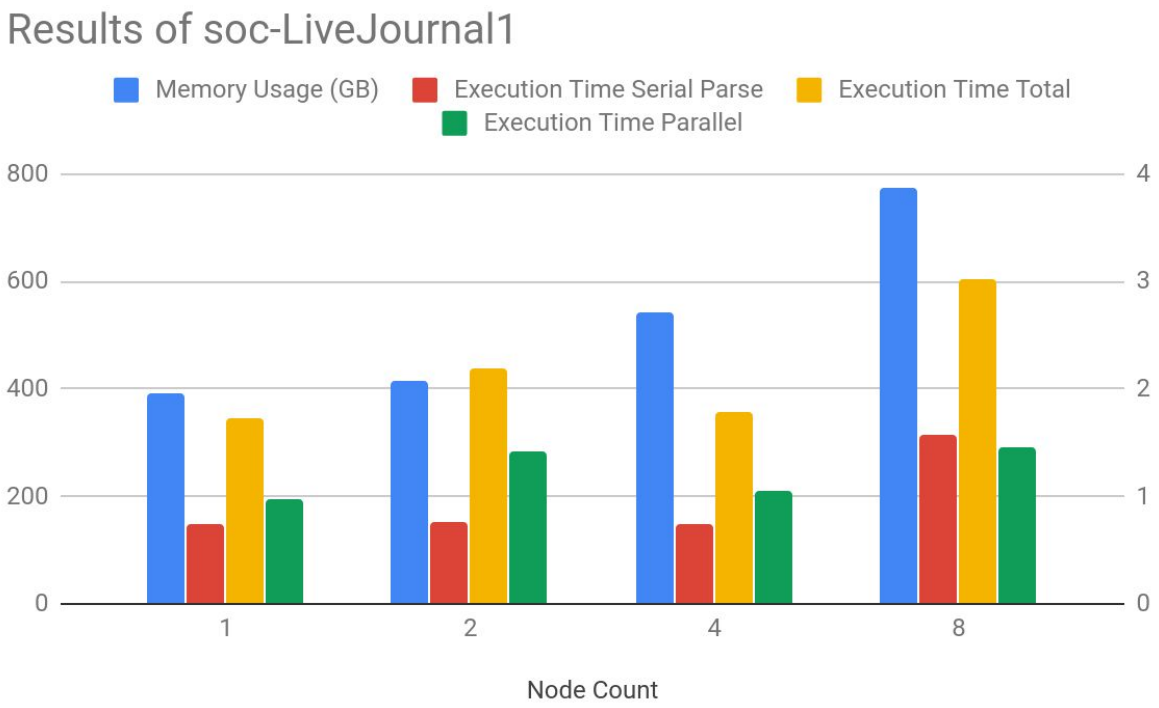
Results of web-Google Graph

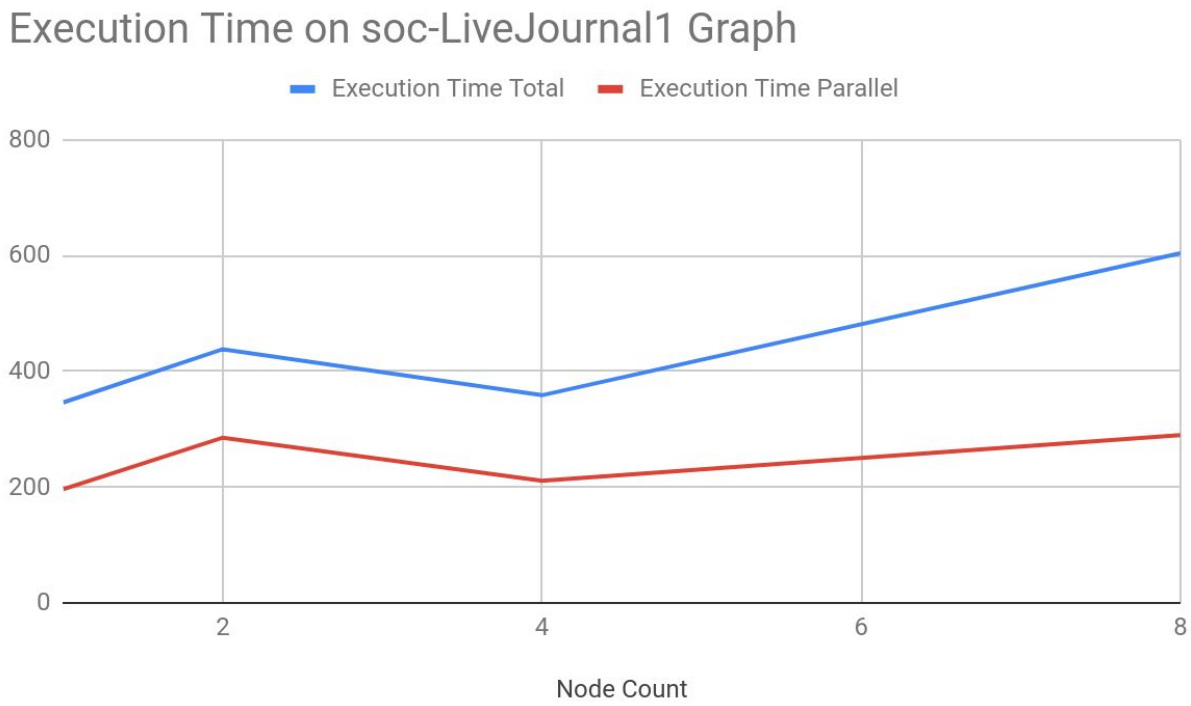


Execution Time on web-Google Graph



soc-LiveJournal1.txt				
Node Count	Memory Usage (GB)	Execution Time Serial Parse	Execution Time Total	Execution Time Parallel
1	1.95	149.815532	346.101324	196.285792
2	2.07	152.755086	438.113445	285.358359
4	2.72	147.79381	358.597089	210.803279
8	3.87	314.224893	603.957577	289.732684





Discussion

As it can be seen from the graphs, performance increases up to 4 nodes in our test machine. In the google graph, performance increase continues up to 6 nodes. This is close to our expectations since the machine has 8 cores with Hyper Threading enabled and a degradation is expected when physical limits are reached since virtual threads will be executed concurrently but not in parallel. Jump in performance for 2 nodes might be due to message passing costs. It can be deduced from the graphs that 2 nodes do not provide enough benefits to overcome the extra costs of message passing introduced in the MPI implementation.

Our serial implementation and Python networkx implementations performed worse then MPI implementation up to 4 nodes in terms of execution time, which can be seen from the results of the google graph.

Memory Usage Analysis

Using Python Networkx Package

Python Networkx library was used to cross check the performance of our implementation.

web-Google

Memory Usage:
Before Execution - during execution (peak)
2.8 GB - 5.51 GB = 2.71 GB use

Execution Time:
48.1729040145874 seconds

Soc-LiveJournal1

2.8 GB -7.70 GB (Run out of Physical Memory and aborted)

Discussion

As it can be seen from the results of Google and Live Journal Graphs, memory usage increases with the number of nodes used. Most notable comparison is with the Networkx implementation. Both serial and MPI implementations use much less memory compared to Networkx library. This might be because the matrix representation. Our adjacency list representation reduces the unnecessary information. soc-LiveJournal1 did not fit into the physical memory of the test machine thus it was aborted however our implementation fits into the memory even when running 8 nodes. Note that these nodes should be running on different machines with different memories, thus use is less on nodes.

Scalability

As explained in the discussions about memory and execution time performance our implementation can use multiple nodes in parallel which can improve execution performance with more physical machines. Also memory performance is manageable on the nodes since sent adjacency lists are partitions of the original data. Though, additional optimizations might be required on the master node and parser to improve this part.

References

[1] M. Özdal. “Lecture1: PageRank Formulation”, Bilkent University, 2018.

[2] Jure Leskovec and Andrej Krevl. “SNAP Datasets: Stanford Large Network Dataset Collection”. June 2014. http://snap.stanford.edu/data

[3] J. Leskovec, K. Lang, A. Dasgupta, M. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics 6(1) 29--123, 2009.

[4] Google programming contest, 2002

[5] L. Backstrom, D. Huttenlocher, J. Kleinberg, X. Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. KDD, 2006.