Distributed PageRank - Final Report

Aaron Myers, Megan Ruthven

December 11, 2014

Contents

1 Introduction

The purpose of this project is to investigate distributed PageRank by applying Pagerank methods currently in use in the multi-thread case and implement it in the distributed setting. It is to determine if these methods can be scaled to multi-machine systems. We also take a method that has already proven to be effective in the distributed setting and apply it to the Pagerank problem (ADMM ¡– what does this stand for [3]). Immediately below is a more detailed description of each approach. The primary metrics for performace will be: speedup, scalability, and ease of implementation, all of which will be explained in the results section. The last metric is included to suggest that ease of coding and understanding of a method has an impact on the adoption rate in industry and therefore should be included in this investigation.

We chose to investigate ADMM [3] because it would have higher performance for ease of coding and understanding. We applied ADMM to the linear form of the Pagerank problem (Ax=b), but it is more difficult to apply a data-driven approach to the separated minimization problems, so to investigate data-driven approaches outlined in [4], we implemented data-driven power iteration with MPI and openMP. Typical power iteration will be refered to as Topology driven. The data-driven methods were Pull, Pull-Push, and Push based. The details of each permutation of the data-driven methods will be discussed in following sections. Because each method is implemented on distributed nodes on a big dataset, each method took advantage of the separate memories, and divided the connection matrices between nodes. There, we implemented a static load balancing regime based on amount of nonzeros in the assigned rows, which proved to be more efficient than load balancing on the number of x elements to calculate. Load balancing will be explained more in the following sections.

The methods' performance will be compared on total time, speed up, and scalability over varying amount of nodes (MPI), and cores per node (openMP). And finally, we will go over the limitations of the methods in MPI and openMP, and future work.

2 Related Work

3 Algorithms and Implementation

This section discusses the implementations of different methods for computing pagerank. This includes power iteration, ADMM, and data-driven approach.

3.1 Topology-driven Power Iteration

This is the classic implmentation for calculating Pagerank, seen in algorithm **??** . It is straight forward, but recalculates all X values every iteration. This could be redundant on a large percentage of the X indices, and therefore, it could waste computational power.

Algorithm 1 Power Iteration with Worklist

Initialize x, δ (threshold)
Compute Px for all nodes
while Worklist is not empty **do**
  if x_i in worklist **then**
    take x_i off the worklist
    $x_i^{new} = (1 - \alpha) * P_i * x + \frac{\alpha}{\#[x]}$
    if $|x_i^{new} - x_i| > \delta$ **then**
      $x_i = x_i^{new}$
      add x_j onto worklist : $\forall x_i \rightarrow x_j$
    end if
  end if
end while

3.2 Data-Driven Pagerank

In addition to the ADMM implementation, we implemented power iteration, as well as three permutations of the data-driven pagerank method (pull, pull-push, and push). The data-driven method (taken from [4]) aims to minimize unnecessary computation by only updating pageranks of elements whose incoming connections were upated to a satisfactory degree.

Algorithm 2 Power Iteration with Worklist

Initialize x, δ (threshold)
Compute Px for all nodes
while Worklist is not empty **do**
  if x_i in worklist **then**
    take x_i off the worklist
    $x_i^{new} = (1 - \alpha) * P_i * x + \frac{\alpha}{\#[x]}$
    if $|x_i^{new} - x_i| > \delta$ **then**
      $x_i = x_i^{new}$
      add x_j onto worklist : $\forall x_i \rightarrow x_j$
    end if
  end if
end while

3.3 Linear System Approach

This approach requires that we form the Pagerank problem into a linear system (Ax=b) in which case solving for x would provide the list of Pagerank values. Below is a simple derivation taken from [2].

$$P' = P + dv^T \tag{1}$$
$$P'' = cP' + (1-c)ev^T \tag{2}$$
$$x^{k+1} = P''^T x^k \tag{3}$$

Where P′ and P″ are the modified PageRank matrices that have the modifications necessary to create a connected graph and add a personalization factor and e is a vector of all 1's, resulting in equation 3, the Power Iteration approach to Pagerank.

Given the additional information below, we can derive the linear system for Pagerank.

$$e^T x = x^T e = \|x\|_1 = \|x\| \tag{4}$$
$$d^T x = \|x\| - \|P^T x\| \tag{5}$$
$$x = [cP^T + c(vd^T) + (1-c)ve^T]x \tag{6}$$

Combining the information above, we arrive at the following equation:

$$(I - cP^T)x = kv \tag{7}$$

We now have Pagerank in a linear form (Ax=b), where A = I-cP^T and kv = b. If in addition, we normalize x, we have the following:

$$k = \|x\| - c\|P^T x\| = (1-c)\|x\| + d^T x \tag{8}$$
$$k = 1 - c \tag{9}$$

3.4 ADMM

Many of the articles we encountered for solving parallel pagerank in the linear form used Jacobi iteration or some Krylov Subspace method (GMRES, BiCGSTAB, etc), but we attempted to implement something we were introduce to in this course, namely ADMM [3]. This is an extremely simple way to parallelize a linear solve. This process attempts to split the linear problem into subsections, solve separately, and combine the information in a very specific way. We will compare these results to the implementation of GMRES and BiCGSTAB for the same problem parallelizing using PETSc [1]. We expect ADMM to have worse performance, measured by speedup, but we would like to quantify the loss in accuracy/time relative to the ease of implementation and scalability.

Below is a brief description of the ADMM idea and algorithm [3].
We take the linear problem and split up the data accordingly:

$$A = [A_1 ... A_n]' \tag{10}$$
$$b = [b_1 ... b_n]' \tag{11}$$

Our orignal minimization of Ax-b with a certain norm and regulariztion on x now becomes:

$$minimize \quad \sum_{i=1}^{N} l_i(A_i x_i - b_i) + r(z) \tag{12}$$
$$subject \;\; to \;\; x_i - z = 0 \;\; \forall i \tag{13}$$

Where x_i are local variables that we force to match the global solution z at each step and N is the number of processes used to solve the problem.
The resulting algorithm, using the augmented lagrangian presented in the ADMM method [3], is as follows:

Algorithm 3 ADMM Iteration

1: $x_i^{k+1} = argmin_x \;\; l_i(A_i x_i - b_i) + \frac{\rho}{2}\|x_i^k - z^k - u_i^k\|_2^2$
2: $z^{k+1} = argmin_z \;\; r(z) + \frac{N\rho}{x}\|z^k - \bar{x}^{k+1} - \bar{u}^k\|_2^2$
3: $u_i^{k+1} = u_i^k + x_i^{k+1} - z^{k+1}$

Where $u_i^k = \frac{1}{\rho} y_i^k$ and for our implementation, we chose the L^1 regularization term (also known as lasso) and used a gradient solver with Eigen to solve the minimization for x. Using the lasso regulariztion the z update becomes soft-thresholding update. Considering the Lasso method, the updated algorithm is below.

Algorithm 4 ADMM Iteration with Lasso

1: $x_i^{k+1} = argmin_x \;\; \|(A_i x_i - b_i)\|_2^2 + \frac{\rho}{2}\|x_i^k - z^k - u_i^k\|_2^2$
2: $z^{k+1} = S_{\lambda/\rho N}(\bar{x}^{k+1} - \bar{u}^k)$
3: $u_i^{k+1} = u_i^k + x_i^{k+1} - z^{k+1}$

Where S is defined componend-wise in the following way:

$$S_{\lambda/\rho N}(x_i) := (x_i - \frac{\lambda}{\rho N})_+ - (-x_i - \frac{\lambda}{\rho N})_+ \tag{14}$$

4 Load Balancing

First we implemented our methods where each node had the same amount of X's to compute. To allow for more equality in the amount of computations each node had to do, our implementations moved to a method where each node were assigned a continuous set of indices where each node had the same amount of nonzero connections between the indices and other nodes.

5 Results

The different methods had varying quantitative and qualitative results. These included the time to complete, speedup, scalability, and ease of coding.

5.1 Power Iteration Results

We implemented the the of the algorithms discussed in [4] in the distributed setting using MPI and MPI with openMP, and compared the time to convergence. On the A dataset from HW4, the pagerank values converged in fewer iterations (25 to 20) and on average, takes less time to run one iteration (0.189 to 0.095 seconds) running on 16 threads. This resulted in a total calculation time of 4.72 seconds for the baseline power iterations and 1.90 seconds for the delta method. A summary table is below.

tablePower Iteration - A

Method Comparison		
Method	Iteration Count	Time(s)
Power Iteration	25	4.72
Delta Update	20	1.90

tablePower Iteration - Friendster

Method Comparison		
Method	Iteration Count	Time(s)
Power Iteration	23	49.6
Delta Update	30	19.5

5.2 ADMM Results Compared to other Linear methods

Below are the inital results for ADMM programmed in Matlab for a simple data set (a disconnected synthetic 11 node graph).

tableLinear Method Table

5 iter results	
Method	$\|\hat{x} - x\|$
GMRES	0.0047
BiCGSTAB	0.0056
ADMM	0.0079

6 Future Work

Now that we have seen the value of both ADMM and data-driven PageRank, our next steps will involve parallelizing using MPI

1. Parallelize the ADMM method using MPI and compare the results to running GMRES and BiCGSTAB with PETSc
2. Parallelize the data-driven PageRank problem using MPI and compare these results to other parallel Power Iteration approaches
3. Collect all results and provide conclusions about all methods and the value that each provides.

7 Conclusion

This bib actually needs to be in a separate bib file

References

[1] David Gleich, et al. *Scalable Computing for Power Law Graphs: Experience with Parallel PageRank*

[2] David Gleich, et al. *Fast Parallel PageRank: A Linear System Approach*

[3] Stephen Boyd, et al. *Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers*

[4] Joyce Whang, et al. *Scalable Data-driven PageRank: Algorithms, System Issues '&' Lessons Learned*

[5] mcs.anl.gov *http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src/jacobicmpl/C/main*

[6] fsu.edu *http://people.sc.fsu.edu/ jburkardt/cppsrc/mpi/mpi.html*

[7] Xiaoyi (Eric) Li *Parallel PageRank Computation using MPI*

[8] David Gleich, Leonid Zhukov, Pavel Berkhin *Fast Parallel PageRank: Methods and Evaluations*

[9] Shubhada Karavinkoppa and Jayesh Kawli *Page Rank Algorithm Using MPI*

[10] Bundit Manaskasemsak, Putchong Uthayopas, Arnon Rungsawang *A Mixed MPI-Thread Approach for Parallel Page Ranking Computation*