

Distributed PageRank - Final Report

Aaron Myers, Megan Ruthven

December 12, 2014

Contents

1	Introduction	2
2	Related Work	2
3	Load Balancing	3
4	Algorithms and Implementation	3
4.1	Topology-driven	3
4.2	Data-Driven Pagerank	4
4.3	Linear System Approach	8
4.4	ADMM	9
5	Results	10
5.1	Power Iteration Results	10
5.2	ADMM Results Compared to other Linear methods	11
6	Future Work	11
7	Conclusion	11

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 Alternating Direction Method of Multipliers (ADMM) [1]). Immediately below is a more detailed description of each approach. The primary metrics for performance 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 [1] 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 [2], we implemented data-driven power iteration with MPI and openMP. Typical power iteration will be referred 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

Pagerank is the algorithm that Larry Page and Sergey Brin researched, and subsequently used their research to form the company Google. Since then, many researchers have focused on improving different aspects of Pagerank. Our project focused on different implementations of Pagerank to compare and contrast implementations' performance in a distributed setting.

Iterative solvers for the formulation $Ax=b$ using well known methods including but not limited to: GMRES, BiCGSTAB, CGS, Chebychev iterations, along with Jacobi iteration methods [3]. Their results indicated that normal Power Iteration and Jacobi methods almost always performed better than the other linear solvers and although we do not have access to their data sets or machines, we will attempt to generally compare our speedup curves to theirs and expect some correlation. We will also expect that our power iteration and data-driven approaches will perform better than our ADMM solver for $Ax=b$ as they did for [3].

Previous implementations of Pagerank with MPI [4] showed that it is expected for com-

pute time to decrease as the number of MPI processes increase to a certain point. As seen in the figure presented in the paper, with a 1M node graph, after about 30 cores, the solve time starting to trend slightly upward. We assume this increase in solve time is primarily due to the increase in message passing overhead. This would suggest that we should see similar results and also that there may be an optimal core number depending on the size of the data.

We have worked extremely closely with Joyce Whang as many of our algorithms were taken directly from her research paper and other members of her research group [2]. In her submitted paper, the algorithms proposed are implemented on a Galois multithread system. Our data-driven implementations are taken directly from her paper and implemented with the MPI and openMP frameworks. This extraction to the distributed setting is to confirm or deny the idea that her algorithms could also be effective across nodes.

3 Load Balancing

Before this paper details the different algorithms we implemented, it will talk about load balancing. Load balancing on distributed systems could be scheduled in task or job manner, as discussed in [5], where jobs were a collection of tasks. Because MPI is best suited for batch passing of messages, our project used a job based approach. The two approaches were to balance jobs by number of tasks (indices to compute Pagerank), or balance by total average work for each node (summation of nonzeros of each incoming node for the assigned indices). First we implemented our methods where each node had the same amount of tasks to compute. Then, we implemented balancing by total work, and compared the differences. The topological and data-driven Pagerank algorithms consistently sped up where the total work balancing was about two times faster than the balanced tasked method. For these reasons, the results reflect algorithms which used the work balancing method.

4 Algorithms and Implementation

This section discusses the implementations of different methods for computing pagerank. This includes topological, data-driven, and ADMM Pagerank.

4.1 Topology-driven

This is the classic implementation for calculating Pagerank, seen in algorithm 1 . 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.

In the topological algorithm, every x_v is updated. Each node computes a subsection of \mathbf{x} , named \mathbf{x}_r , then all of the nodes combine their new result in batch to create a new \mathbf{x} in order to continue to compute the newer values of \mathbf{x}_r . All of the recomputation within each node only updates the current x_v , this creates for a thread safe algorithm.

Algorithm 1 Topology-driven Pagerank

```
1: Input: graph  $P_r = (V_r, E_r)$ ,  $\alpha$ ,  $\epsilon$ 
2: Output: Pagerank  $\mathbf{x}$ 
3: Initialize  $\mathbf{x} = \alpha \mathbf{e}$ 
4:  $\exists \mathbf{x}_r$  in  $\mathbf{x}$  as  $P_r$  is to  $P$ 
5: while true do
6:   for  $v \in V_r$  do
7:      $x_v^{(k+1)} = \alpha + (1 - \alpha) \sum_{w \in S_v} \frac{x_w^{(k)}}{|T_w|}$ 
8:      $\delta_v = |x_v^{(k+1)} - x_v^{(k)}|$ 
9:   end for
10:  sync all  $\mathbf{x}_r$  between nodes
11:  if  $\|\delta\|_\infty < \epsilon$  then
12:    break;
13:  end if
14: end while
15:  $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$ 
```

4.2 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 [2]) aims to minimize unnecessary computation by only updating pageranks of elements whose incoming connections were updated to a satisfactory degree.

The Pull data-driven method in algorithm 2 selectively updates the values of x_v based on if an incoming connected node updated itself above the threshold, ϵ . This is determined by the array t_c , which is a set of c indices for that node. In the computation of x_v , if the difference between the new and old value is above ϵ , set all outgoing nodes' t_w to true. Notice that each element, v , updates its value, x_v , and indices of t both in and out of set c . This requires the syncing of all copies of t^{new} between nodes with a logical or. Additionally, this algorithm accesses more memory by accessing incoming nodes' pagerank and outgoing nodes' t^{new} .

The Pull-Push data-driven method in algorithm 3 selectively updates the values of x_v based on if its residual is above a threshold, ϵ . The residuals are accounted for in the \mathbf{r} vector and batch updated to transfer residual values between nodes. Notice that each element, v , updates its value, x_v , and indices of r both in and out of set c . This requires the syncing of all copies of r^{new} between nodes by summing. Additionally, this algorithm accesses more memory by accessing incoming nodes' pagerank and outgoing nodes' r^{new} .

The Push data-driven method in algorithm 4 selectively updates the values of x_v based on if its residual is above a threshold, ϵ . It updates x_v by only using r_v . This accesses less memory by bypassing fetching all of the incoming nodes current values x_w . The residuals are accounted for in the \mathbf{r} vector and batch updated to transfer residual values between nodes. Notice that each element, v , updates its value, x_v , and indices of r both in and out of set c . This requires the syncing of all copies of r^{new} between nodes by summing.

Although MPI allowed for a distributed Pagerank algorithm, there were a couple of

Algorithm 2 Pull Data-driven Pagerank

```
1: Input: graph  $P_c = (V_c, E_c)$ ,  $\alpha$ ,  $\epsilon$ 
2: Output: Pagerank  $\mathbf{x}$ 
3: Initialize  $\mathbf{x} = \alpha \mathbf{e}$  and  $\mathbf{t}_c = true \times \mathbf{e}_c$ 
4:  $\exists \mathbf{x}_c$  in  $\mathbf{x}$  as  $P_c$  is to  $P$ 
5: while  $\exists v$  s.t.  $t_v = true$  do
6:    $t^{new} = false \times \mathbf{e}$ 
7:   for  $v \in V_c$  do
8:     if  $t_v = true$  then
9:        $x_v^{new} = \alpha + (1 - \alpha) \sum_{w \in S_v} \frac{x_w}{|T_w|}$ 
10:      if  $|x_v^{new} - x_v| \geq \epsilon$  then
11:         $x_v = x_v^{new}$ 
12:        for  $w \in T_v$  do
13:           $t_w^{new} = true$ 
14:        end for
15:      end if
16:    end if
17:  end for
18:  sync all  $\mathbf{x}_c$  between nodes
19:  logical or all  $t = t^{new}$  between nodes
20: end while
21:  $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$ 
```

Algorithm 3 Pull-Push Data-driven Pagerank

```
1: Input: graph  $P_c = (V_c, E_c)$ ,  $\alpha$ ,  $\epsilon$ 
2: Output: Pagerank  $\mathbf{x}$ 
3: Initialize  $\mathbf{x} = \alpha \mathbf{e}$ 
4:  $\exists \mathbf{x}_c$  in  $\mathbf{x}$  as  $P_c$  is to  $P$ 
5: Initialize  $\mathbf{r} = \mathbf{0}$ 
6: for  $v \in V_c$  do
7:   for  $w \in S_v$  do
8:      $r_v = r_v + \frac{1}{|T_w|}$ 
9:   end for
10:   $r_v = (1 - \alpha)\alpha r_v$ 
11: end for
12: while  $\exists v$  s.t.  $r_v \geq \epsilon$  do
13:   $\mathbf{r}^{new} = \mathbf{0}$ 
14:  for  $v \in V_c$  do
15:    if  $r_v \geq \epsilon$  then
16:       $x_v = \alpha + (1 - \alpha) \sum_{w \in S_v} \frac{x_w}{|T_w|}$ 
17:      for  $w \in T_v$  do
18:         $r_w^{new} = r_w^{new} + \frac{r_v \alpha}{|T_v|}$ 
19:      end for
20:    else
21:       $r_v^{new} = r_v^{new} + r_v$ 
22:    end if
23:  end for
24:  sync all  $\mathbf{x}_c$  between nodes
25:  add all and scatter  $r_c = r^{new}$  between nodes
26: end while
27:  $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$ 
```

Algorithm 4 Push Data-driven Pagerank

```
1: Input: graph  $P_c = (V_c, E_c)$ ,  $\alpha$ ,  $\epsilon$ 
2: Output: Pagerank  $\mathbf{x}$ 
3: Initialize  $\mathbf{x} = \alpha \mathbf{e}$ 
4:  $\exists \mathbf{x}_c$  in  $\mathbf{x}$  as  $P_c$  is to  $P$ 
5: Initialize  $\mathbf{r} = \mathbf{0}$ 
6: for  $v \in V_c$  do
7:   for  $w \in S_v$  do
8:      $r_v = r_v + \frac{1}{|T_w|}$ 
9:   end for
10:   $r_v = (1 - \alpha)\alpha r_v$ 
11: end for
12: while  $\exists v$  s.t.  $r_v \geq \epsilon$  do
13:   $\mathbf{r}^{new} = \mathbf{0}$ 
14:  for  $v \in V_c$  do
15:    if  $r_v \geq \epsilon$  then
16:       $x_v = x_v + r_v$ 
17:      for  $w \in T_v$  do
18:         $r_w^{new} = r_w^{new} + \frac{r_v \alpha}{|T_v|}$ 
19:      end for
20:    else
21:       $r_v^{new} = r_v^{new} + r_v$ 
22:    end if
23:  end for
24:  add all and scatter  $r_c = r^{new}$  between nodes
25: end while
26: sync all  $\mathbf{x}_c$  between nodes
27:  $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$ 
```

restrictions on the implementation because of the MPI and openMP framework we chose to use. When syncing values between nodes, MPI requires all nodes to simultaneously request and send the information they require. This means that all of the independent nodes have to update the values in batch. The original implementation in Galois ?? utilized the asynchronicity of value updates unique to Galois to be able to have threadsafe updates, and shared residual values between threads. This is not a feature in openMP or MPI. In openMP, each thread kept track of its own residuals for the whole set of indices, and reduced after all were calculated within each node, then shared its values to sum up all of the nodes' residual values. These 2 steps of synchronization were not necessary in the Galois implementation. Additionally, the batch update of residuals and the boolean update vector meant that a meaningful worklist was impossible to keep without the vectors from the other nodes. This meant, to selectively compute a new value the Pagerank, each node had to go through each value of the residual or update vectors to check if it met the standards of update.

The new vectors of information about each index needed to be passed between nodes. More information to send corresponds with a longer amount of time spend on sending data. In the case of algorithms 2 and 3, they both send \mathbf{x}_c and an update vector. This does not scale well for big

NOTE: THIS NEEDS MORE INFORMATION

4.3 Linear System Approach

This approach requires that we form the Pagerank problem into a different linear system ($Ax=b$) where we fundamentally look at methods of iterating or directly solving for an inverse to solve for x which would provide the list of Pagerank values. Below is a simple derivation taken from [3].

$$P' = P + dv^T \quad (1)$$

$$P'' = cP' + (1 - c)ev^T \quad (2)$$

$$x^{k+1} = P''^T x^k \quad (3)$$

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.

$$e^T x = x^T e = \|x\|_1 = \|x\| \quad (4)$$

$$d^T x = \|x\| - \|P^T x\| \quad (5)$$

$$x = [cP^T + c(vd^T) + (1 - c)ve^T]x \quad (6)$$

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

$$(I - cP^T)x = kv \quad (7)$$

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:

$$k = \|x\| - c\|P^T x\| = (1 - c)\|x\| + d^T x \quad (8)$$

$$k = 1 - c \quad (9)$$

4.4 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 introduced to in this course, namely ADMM [1]. 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 (Generalized Minimal RESidual method) and BiCGSTAB (BiConjugate Gradient method with STABILization) for the same problem parallelizing using PETSc (a software with great tools for parallelizing linear solvers) [?]. 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 [1].

We take the linear problem and split up the data accordingly:

$$A = [A_1 \dots A_n]' \quad (10)$$

$$b = [b_1 \dots b_n]' \quad (11)$$

Our original minimization of $Ax=b$ with a certain norm and regularization on x now becomes:

$$\text{minimize} \quad \sum_{i=1}^N l_i(A_i x_i - b_i) + r(z) \quad (12)$$

$$\text{subject to} \quad x_i - z = 0 \quad \forall i \quad (13)$$

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. This method also includes an 'augmented lagrangian' term. This term is used to bring robustness to the dual ascent problem and result in convergence without criteria like strict convexity or finiteness of the function. This is discussed in much greater detail in [1]. Below is the augmented lagrangian which is used to derive the resulting algorithm.

$$L_\rho(x, y) = f(x) + y^T(Ax - b) + \frac{\rho}{2}\|Ax - b\|_2^2 \quad (14)$$

$$(15)$$

The resulting algorithm, using the augmented lagrangian presented in the ADMM method [1], is as follows:

Algorithm 5 ADMM Iteration

- 1: $x_i^{k+1} = \operatorname{argmin}_x l_i(A_i x_i - b_i) + \frac{\rho}{2}\|x_i^k - z^k - u_i^k\|_2^2$
 - 2: $z^{k+1} = \operatorname{argmin}_z r(z) + \frac{N\rho}{x}\|z^k - \bar{x}^{k+1} - \bar{u}^k\|_2^2$
 - 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 regularization the z update becomes soft-thresholding update. Considering the Lasso method, the updated algorithm is below.

Algorithm 6 ADMM Iteration with Lasso

- 1: $x_i^{k+1} = \operatorname{argmin}_x \|(A_i x_i - b_i)\|_2^2 + \frac{\rho}{2}\|x_i^k - z^k - u_i^k\|_2^2$
 - 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:

$$S_{\lambda/\rho N}(x_i) := (x_i - \frac{\lambda}{\rho N})_+ - (-x_i - \frac{\lambda}{\rho N})_+ \quad (16)$$

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

This is where graphs would go

5.2 ADMM Results Compared to other Linear methods

6 Future Work

7 Conclusion

This bib actually needs to be in a separate bib file

References

- [1] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–122, Jan. 2011.
- [2] Andrew Lenharth Joyce Jiyoung Whang and Inderjit S. Dhillon, “Scalable data-driven pagerank: Algorithms, system issues ’&’ lessons learned,” Submitted.
- [3] David Gleich, “Fast parallel pagerank: A linear system approach,” Tech. Rep., 2004.
- [4] Xiaoyi (Eric) Li, “Parallel pagerank computation using mpi,” <http://www.cse.buffalo.edu/faculty/miller/Courses/CSE633/Li-Fall-2012-CSE633.pdf>.
- [5] Asser N. Tantawi and Don Towsley, “Optimal static load balancing in distributed computer systems,” *J. ACM*, vol. 32, no. 2, pp. 445–465, Apr. 1985.