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 Alternating Direction Method of Multipliers (ADMM) [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

1. [2] - This paper focuses on 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. 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 nor 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 [2].

2. [7] - The method descibed in this paper, used a weighted graph so we would expect the implementation to be slightly different. However, the primary take=away from this paper is that we should expect the compute time to decrease as the MPI processes increases, up 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.

3. [4] - 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. Aside of our ADMM solver, our paper focuses on implementing her ideas in a distributed setting (as her implementation focuses only on the multi-thread case) either to confirm or deny the idea that her algorithms could also be effective in a distributed setting.

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 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.

Algorithm 1 Topology-driven Pagerank

Input: graph $P_r = (V_r, E_r)$, α, ϵ
Output: Pagerank $\mathbf{x}$
Initialize $\mathbf{x} = \alpha\mathbf{e}$
$\exists\, \mathbf{x_r}$ in $\mathbf{x}$ as P_r is to P
while true **do**
  for $v \in V_r$ **do**
    $x_v^{(k+1)} = \alpha + (1-\alpha)\sum_{w \in S_v} \frac{x_w^{(k)}}{|T_w|}$
    $\delta_v = |x_v^{(k+1)} - x_v^{(k)}|$
  end for
  sync all $\mathbf{x_r}$ between nodes
  if $\|\delta\|_\infty < \epsilon$ **then**
    break;
  end if
end while
$\mathbf{x} = \frac{\mathbf{x}}{\|x\|_1}$

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.

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 Pull Data-driven Pagerank

Input: graph $P_c = (V_c, E_c)$, α, ϵ
Output: Pagerank $\mathbf{x}$
Initialize $\mathbf{x} = \alpha\mathbf{e}$ and $\mathbf{t_c} = true \times \mathbf{e_c}$
$\exists\, \mathbf{x_c}$ in $\mathbf{x}$ as P_c is to P
while $\exists\, v\ s.t.\ t_v = true$ **do**
  $t^{new} = false \times \mathbf{e}$
  for $v \in V_c$ **do**
    if $t_v = true$ **then**
      $x_v^{new} = \alpha + (1-\alpha)\sum_{w \in S_v} \frac{x_w}{|T_w|}$
      if $|x_v^{new} - x_v| \geq \epsilon$ **then**
        $x_v = x_v^{new}$
        for $w \in T_v$ **do**
          $t_w^{new} = true$
        end for
      end if
    end if
  end for
  sync all $\mathbf{x_c}$ between nodes
  logical or all $t = t^{new}$ between nodes
end while
$\mathbf{x} = \frac{\mathbf{x}}{\|x\|_1}$

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}.

Algorithm 3 Pull-Push Data-driven Pagerank

Input: graph $P_c = (V_c, E_c)$, α, ϵ
Output: Pagerank $\mathbf{x}$
Initialize $\mathbf{x} = \alpha\mathbf{e}$
$\exists\, \mathbf{x_c}$ in $\mathbf{x}$ as P_c is to P
Initialize $\mathbf{r} = \mathbf{0}$
for $v \in V_c$ **do**
  for $w \in S_v$ **do**
    $r_v = r_v + \frac{1}{|T_w|}$
  end for
  $r_v = (1-\alpha)\alpha r_v$
end for
while $\exists\, v\ s.t.\ r_v \geq \epsilon$ **do**
  $r^{new} = \mathbf{0}$
  for $v \in V_c$ **do**
    if $r_v \geq \epsilon$ **then**
      $x_v = \alpha + (1-\alpha)\sum_{w \in S_v} \frac{x_w}{|T_w|}$
      for $w \in T_v$ **do**
        $r_w^{new} = r_w^{new} + \frac{r_v \alpha}{|T_v|}$
      end for
    else
      $r_v^{new} = r_v^{new} + r_v$
    end if
  end for
  sync all $\mathbf{x_c}$ between nodes
  add all and scatter $r_c = r^{new}$ between nodes
end while
$\mathbf{x} = \frac{\mathbf{x}}{\|x\|_1}$

Algorithm 4 Push Data-driven Pagerank

Input: graph $P_c = (V_c, E_c)$, α, ϵ
Output: Pagerank $\mathbf{x}$
Initialize $\mathbf{x} = \alpha\mathbf{e}$
$\exists\, \mathbf{x_c}$ in $\mathbf{x}$ as P_c is to P
Initialize $\mathbf{r} = \mathbf{0}$
for $v \in V_c$ **do**
  for $w \in S_v$ **do**
    $r_v = r_v + \frac{1}{|T_w|}$
  end for
  $r_v = (1-\alpha)\alpha r_v$
end for
while $\exists\, v\ s.t.\ r_v \geq \epsilon$ **do**
  $r^{new} = \mathbf{0}$
  for $v \in V_c$ **do**
    if $r_v \geq \epsilon$ **then**
      $x_v = x_v + r_v$
      for $w \in T_v$ **do**
        $r_w^{new} = r_w^{new} + \frac{r_v \alpha}{|T_v|}$
      end for
    else
      $r_v^{new} = r_v^{new} + r_v$
    end if
  end for
  add all and scatter $r_c = r^{new}$ between nodes
end while
sync all $\mathbf{x_c}$ between nodes
$\mathbf{x} = \frac{\mathbf{x}}{\|x\|_1}$

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 **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 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 want. This means that all of the independent nodes have to update the values in batch. This is was not ideal because it meant that partial residuals were silowed in different nodes. The distributedness of each residual disallows for a priority queue and asynchronis updating.

NOTE: THIS NEEDS MORE INFORMATION

3.3 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 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 (Generalized Minimal RESidual method) and BiCGSTAB (BiConjugate Gradient method with STABilization) for the same problem parallelizing using PETSc (a software with great tools for parallizing linear solvers) [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 origninal 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. This method also includes an 'augmented lagragian' 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 [3]. Below is the augemented lagrangian which is used to derive the resulting algorithm.

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

$$\tag{15}$$

The resulting algorithm, using the augmented lagrangian presented in the ADMM method [3], is as follows:

Algorithm 5 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 6 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:

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

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

This is where graphs would go

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).

Table 1: Linear 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, the next steps would be the following.

1. Optimize our MPI approach with the data-driven methods in an attempt to get the results to match the multi-thread implementation of [4].
2. Take the results of this investiation as an indicator that expanding the galois code from [4] to a distributed setting might be valuable.
3. Apply ADMM to with other minimization solver as an attempt to find the optimal solver.

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*