Scalable Data-driven PageRank: Algorithms, System Issues & Lessons Learned

Joyce Jiyoung Whang
University of Texas at Austin
Texas, USA
joyce@cs.utexas.edu

Andrew Lenharth
University of Texas at Austin
Texas, USA
lenharth@ices.utexas.edu

Inderjit S. Dhillon
University of Texas at Austin
Texas, USA
inderjit@cs.utexas.edu

ABSTRACT

Large-scale network and graph analysis has received considerable attention in both data mining and parallel programming communities. In data mining, many different types of task-specific approximate algorithms have been developed to deal with massive networks. In parallel computing, many different parallel programming models and systems have been developed to ease implementation and manage parallel concerns. Graph mining techniques often involve an iterative, convergence-based algorithm. Such algorithms can be implemented in a variety of ways. Using PageRank as a model problem, we look at three algorithm design axis: work activation, data access pattern, and scheduling. We investigate the impact of different algorithm design choices. Using these design axis, we test a variety of PageRank implementations finding data-driven, push-based algorithms able to achieve more than 28x the performance of standard PageRank implementations (e.g. that in GraphLab). The design choices affect both single-threaded performance as well as parallel scalability. The implementation lessons not only guide good implementations of many graph mining algorithms, but also provide a framework for designing new, more scalable algorithms.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.2 [**Database Management**]: Database Applications—*Data mining*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Scalable Computing, Graph Analytics, PageRank, Multi-threaded Programming, Data-driven Algorithm

1. INTRODUCTION

Massive networks have emerged in recent years, and mining these large-scale networks is a challenging task. Large-scale network analysis has received considerable attention in both data mining and parallel computing communities. In data mining, many different types of task-specific approximate algorithms have been developed to deal with massive networks. For example, community detection is an important task in network analysis, and various large-scale graph partitioning or community detection algorithms have been developed over the recent years, e.g., [15], [16]. In parallel computing, many different parallel programming models and systems have been proposed for both shared memory and distributed memory settings. Due to the sparsity of real-world networks, developing efficient parallel framework for graph analytics especially has been recognized as a tough problem.

A remarkable number of "large" graphs fit in memory. It is easy to fit graphs with tens of billions of edge on a large workstation-class machine. Widely benchmarked graphs, such as the Twitter graphs, even fit in memory on laptops. Machines able to compute graph analytics on graphs of this size are both readily available, even as cheap cloud instances at dollars an hour. Various researches have observed (e.g. in [9]) that distributed graph analytics can have a significant slowdown over shared-memory implementation, that is, the increase in communication costs are not easily made up for by increase in aggregate processing power or memory bandwidth. Given these factors, it is worth understanding how to efficiently parallelize graph analytics on shared-memory machines. A better understanding of how to implement fast shared-memory analytics both greatly reduces the costs and enables richer applications on commodity systems. Better implementation strategies also helps distributed implementations, as they tend to use shared-memory abstractions within a host.

Graph mining techniques usually involve iterative computations and convergence-based algorithms where local computations are repeatedly done at a set of nodes until a convergence criterion is satisfied. At a high level algorithm design perspective, we can broadly classify these iterative graph algorithms based on three different points of view: work activation, data access pattern, and scheduling.

Let us define *active nodes* to be a set of nodes where computations should be performed. When we classify algorithms based on work activation, the focus is how the active nodes are decided, and we can divide algorithms into two groups: topology-driven and data-driven algorithms. In a topology-driven algorithm, active nodes are defined solely by the structure of a graph whereas in a data-driven algorithm, the nodes are dynamically activated by their neighbors. The concepts of topology-driven and data-driven have been introduced in parallel programming models, and a new parallel programming framework called Galois has been recently developed, which is intended to support data-driven parallelism [9]. From the perspective of developing a graph mining algorithm, how to design scalable data-driven algorithms is still an open problem.

Another important view point is data access pattern. When an active node is processed, there can be a particular data access pattern. For example, some algorithms require reading a value of an active node and updating its outgoing neighbors, whereas some algorithms require reading values from incoming neighbors of an active node and updating the active node's value. Based on these data access patterns for each local computation, we can classify algorithms into three categories: pull-based, push-based, and pull-push-based algorithms.

Finally, the order of processing active nodes can be also important especially for convergence-based algorithms because a particular ordering might lead to a faster convergence. Thus, scheduling is another important issue when we design an algorithm.

In this paper, we present a general approach for designing scalable data-driven graph algorithms with the case study of PageRank. In particular, using three different algorithm design axis (i.e., work activation, data access pattern, and scheduling), we present eight different formulations of PageRank algorithm. We show that by considering data-driven formulations, we can have much flexibility in processing the active nodes, which enables us to develop work-efficient algorithms. We focus our analysis on PageRank in this manuscript, but our approaches and formulations can be easily extended to other graph mining algorithms.

Since data-driven algorithms tend to generate fine-grained tasks, paralleizing them might be challenging. But, we show that using an appropriate programming model, we can achieve a reasonably good speedup. Experimental results show that our parallel data-driven PageRank algorithm achieves the speedup of 24 using 32 cores in a graph with 82,924,685 nodes and 1,937,489,264 edges. Also, we observe that our parallel data-driven PageRank algorithm is much faster than the state-of-the-art parallel PageRank implementation.

2. WORK ACTIVATION

We begin our discussion by topology-driven and data-driven algorithms. In a topology-driven algorithm, active nodes are defined solely by graph structure. For example, an algorithm which requires processing all the nodes at each iteration is referred as a topology-driven algorithm. On the other hand, in a data-driven algorithm, active nodes are dynamically changed. Starting from an initial set of active nodes, nodes become active or inactive in an unpredictable way. In many applications, data-driven algorithms can be more work-efficient than topology-driven algorithms because the former allows us to concentrate more on "hot spots" in a graph where more frequent updates are needed. While topology-driven algorithms tend to generate coarse-grained tasks, data-driven algorithms are likely to generate fine-grained tasks. This indicates that parallelizing data-driven algorithms can be challenging mainly because of load balancing problem. Thus, it is critical to choose an appropriate programming model and data structure to get a good speedup.

Algorithm 1 Topology-driven PageRank

```
Input: graph $G = (\mathcal{V}, \mathcal{E})$, $\alpha$, $\epsilon$
Output: PageRank $\mathbf{x}$
 1: Initialize $\mathbf{x} = (1-\alpha)\mathbf{e}$
 2: while true do
 3:   for $v \in \mathcal{V}$ do
 4:     $x_v^{(k+1)} = \alpha \sum_{w \in \mathcal{S}_v} \frac{x_w^{(k)}}{|\mathcal{T}_w|} + (1-\alpha)$
 5:     $\delta_v = |x_v^{(k+1)} - x_v^{(k)}|$
 6:   end for
 7:   if $\|\boldsymbol{\delta}\|_\infty < \epsilon$ then
 8:     break;
 9:   end if
10: end while
11: $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$
```

Algorithm 2 Data-driven PageRank

```
Input: graph $G = (\mathcal{V}, \mathcal{E})$, $\alpha$, $\epsilon$
Output: PageRank $\mathbf{x}$
 1: Initialize $\mathbf{x} = (1-\alpha)\mathbf{e}$
 2: for $v \in \mathcal{V}$ do
 3:   worklist.push($v$)
 4: end for
 5: while !worklist.isEmpty do
 6:   $v$ = worklist.pop()
 7:   $x_v^{new} = \alpha \sum_{w \in \mathcal{S}_v} \frac{x_w}{|\mathcal{T}_w|} + (1-\alpha)$
 8:   if $|x_v^{new} - x_v| \geq \epsilon$ then
 9:     $x_v = x_v^{new}$
10:     for $w \in \mathcal{T}_v$ do
11:       if $w$ is not in worklist then
12:         worklist.push($w$)
13:       end if
14:     end for
15:   end if
16: end while
17: $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$
```

2.1 Topology-driven PageRank

To explain the concepts of topology-driven and data-driven in more detail, we now focus our discussion on PageRank which is a key technique in Web mining. We select PageRank as our model problem because of its popularity and simplicity. PageRank is one of the most widely used centrality measures in network analysis, which was originally designed for measuring importance of web pages in the Web [4]. Given a graph $G = (\mathcal{V}, \mathcal{E})$ with a vertex set $\mathcal{V}$ and an edge set $\mathcal{E}$, let $\mathbf{x}$ denote a PageRank vector of size $|\mathcal{V}|$. Also, let us define $\mathcal{S}_v$ to be a set of incoming neighbors of node v, and $\mathcal{T}_v$ to be a set of outgoing neighbors of node v. Then, node v's PageRank, denoted by x_v, is iteratively computed by

$$x_v^{(k+1)} = \alpha \sum_{w \in \mathcal{S}_v} \frac{x_w^{(k)}}{|\mathcal{T}_w|} + (1-\alpha),$$

where $x_v^{(k)}$ denotes the k-th update of x_v, and α is a teleportation parameter ($0 < \alpha < 1$). Using a random surfer model, the PageRank can be interpreted as the stationary distribution of a random walk that with probability α, follows a step of a random walk to one of its outgoing nodes, and with probability $(1-\alpha)$, jumps to a random node.

Algorithm 1 shows the Power method which is a traditional way to compute PageRank. Given a user defined tolerance ϵ, the PageRank vector $\mathbf{x}$ is firstly initialized to be $\mathbf{x} = (1-\alpha)\mathbf{e}$ where $\mathbf{e}$ denotes a vector having all the elements equal to one. The PageRank values are repeatedly computed until the difference between $x_v^{(k)}$ and $x_v^{(k+1)}$ is smaller than ϵ for all the nodes. Since the Power method requires processing all the nodes at each round (i.e., all the nodes are active at each round), it is a topology-driven algorithm. In other words, when a graph is given, active nodes are fixed and not changed while the algorithm proceeds.

2.2 Basic Data-driven PageRank

Instead of processing all the nodes in rounds, we can think of an algorithm which dynamically maintains a working set such that if a node's PageRank is changed, its outgoing neighbors are inserted into the working set. Maintaining a dynamic working set is a key concept for a data-driven algorithm. Algorithm 2 shows a basic data-driven PageRank. Initially, the worklist is set to be the entire vertex set. The algorithm proceeds by piking a node from the worklist, computing the node's PageRank, and adding its outgoing neighbors to the worklist. Notice that the worklist keeps changing until it becomes empty. Even though it seems a reasonable algorithm, it is not straightforward to see if Algorithm 2 will converge. To see the convergence of the data-driven PageRank, let us rewrite the problem in the form of a linear system. We define a row-stochastic matrix $\boldsymbol{P}$ to be $\boldsymbol{P} \equiv \boldsymbol{D}^{-1}\boldsymbol{A}$ where $\boldsymbol{A}$ is an adjacency matrix and $\boldsymbol{D}$ is the degree diagonal matrix. We assume that there is no self-loop in the graph, i.e., the diagonal elements of $\boldsymbol{A}$ are all zeros. Then, the PageRank computation can be written as follows:

$$\mathbf{x} = \alpha \boldsymbol{P}^T \mathbf{x} + (1-\alpha)\mathbf{e}.$$

This is the linear system of

$$(\boldsymbol{I} - \alpha \boldsymbol{P}^T)\mathbf{x} = (1-\alpha)\mathbf{e}.$$

and the residual is defined to be

$$\mathbf{r} = (1-\alpha)\mathbf{e} - (\boldsymbol{I} - \alpha\boldsymbol{P}^T)\mathbf{x} = \alpha\boldsymbol{P}^T\mathbf{x} + (1-\alpha)\mathbf{e} - \mathbf{x}.$$

In this setting, if we show that each local computation in Algorithm 2 decreases the residual (and the residual is always positive), we can say that Algorithm 2 will converge. As also shown in [8], we can show that whenever x_v is updated, the total residual is decreased at least by $r_v(1-\alpha)$ where r_v is the residual of node v (r_v is defined to be the v-th element of $\mathbf{r}$). We formally dictate it in Theorem 1. Please see the Appendix for the full proof.

Theorem 1. *In Algorithm 2, when $x_v^{(k)}$ is updated to $x_v^{(k+1)}$, the total residual is decreased at least by $r_v(1-\alpha)$. More specifically,*

$$\mathbf{e}^T\mathbf{r}^{(k+1)} = \begin{cases} \mathbf{e}^T\mathbf{r}^{(k)} - r_v^{(k)}(1-\alpha) & : \mathcal{T}_v \neq \emptyset \\ \mathbf{e}^T\mathbf{r}^{(k)} - r_v^{(k)} & : \mathcal{T}_v = \emptyset \end{cases}$$

Indeed, when a node v's PageRank is updated, its residual r_v becomes zero, and $\alpha r_v/|\mathcal{T}_v|$ is added to each of its outgoing neighbors (details are presented in the proof of Theorem 1). Followed from this, we can also show Theorem 2.

Algorithm 3 Push-Pull-based Data-driven PageRank

```
Input: graph $G = (\mathcal{V}, \mathcal{E})$, $\alpha$, $\epsilon$
Output: PageRank $\mathbf{x}$
 1: Initialize $\mathbf{x} = (1-\alpha)\mathbf{e}$
 2: Initialize $\mathbf{r} = \mathbf{0}$
 3: for $v \in \mathcal{V}$ do
 4:     for $w \in \mathcal{S}_v$ do
 5:         $r_v = r_v + \frac{1}{|\mathcal{T}_w|}$
 6:     end for
 7:     $r_v = (1-\alpha)\alpha r_v$
 8: end for
 9: for $v \in \mathcal{V}$ do
10:     if $r_v \geq \epsilon$ then
11:         worklist.push($v$)
12:     end if
13: end for
14: while !worklist.isEmpty do
15:     $v$ = worklist.pop()
16:     $x_v = \alpha \sum_{w \in \mathcal{S}_v} \frac{x_w}{|\mathcal{T}_w|} + (1-\alpha)$
17:     for $w \in \mathcal{T}_v$ do
18:         $r_w = r_w + \frac{r_v \alpha}{|\mathcal{T}_v|}$
19:         if $w$ is not in worklist & $r_w \geq \epsilon$ then
20:             worklist.push($w$)
21:         end if
22:     end for
23:     $r_v = 0$
24: end while
25: $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$
```

Theorem 2. *Algorithm 2 guarantees $\|\mathbf{r}\|_\infty < \epsilon$ when it is converged.*

Proof. Whenever a node's PageRank is updated, the residual of each of its outgoing neighbors is increased. Thus, if we ever change a node's PageRank, we need to add its outgoing neighbors to the worklist to verify that their residual is sufficiently small. This is what Algorithm 2 does. □

Putting all these together, we can see that Algorithm 2 will converge, and it guarantees $\|\mathbf{r}\|_\infty < \epsilon$ when it is converged. While the topology-driven PageRank processes all the nodes at each iteration, in the data-driven PageRank, computations are only performed on a set of nodes where the computations are needed. Thus, it is likely that the data-driven PageRank is more work-efficient than the topology-driven PageRank. From the next section, we will focus on the data-driven formulation of PageRank, and build up various variations of the basic data-driven PageRank.

3. DATA ACCESS PATTERN

Data access pattern (or memory access pattern) is an important factor one should consider for designing a scalable graph algorithm. Based on the types of operations an active node performs, there can be three different data access patterns: pull-based, pull-push-based, and push-based algorithms.

3.1 Pull-based PageRank

In *pull-based* algorithms, an active node *pulls (reads)* its neighbors' values and updates its own value. Note that pull-based algorithms require more *read* operations than *write* operations in general because the *write* operation is only

Algorithm 4 Push-based Data-driven PageRank

```
Input: graph $G = (\mathcal{V}, \mathcal{E})$, $\alpha$, $\epsilon$
Output: PageRank $\mathbf{x}$
 1: Initialize $\mathbf{x} = (1-\alpha)\mathbf{e}$
 2: Initialize $\mathbf{r} = \mathbf{0}$
 3: for $v \in \mathcal{V}$ do
 4:     for $w \in \mathcal{S}_v$ do
 5:         $r_v = r_v + \frac{1}{|\mathcal{T}_w|}$
 6:     end for
 7:     $r_v = (1-\alpha)\alpha r_v$
 8: end for
 9: for $v \in \mathcal{V}$ do
10:     worklist.push($v$)
11: end for
12: while !worklist.isEmpty do
13:     $v$ = worklist.pop()
14:     $x_v^{new} = x_v^{old} + r_v$
15:     for $w \in \mathcal{T}_v$ do
16:         $r_w = r_w + \frac{r_v \alpha}{|\mathcal{T}_v|}$
17:         if $r_w \geq \epsilon$ then
18:             worklist.push($w$)
19:         end if
20:     end for
21:     $r_v$=0
22: end while
23: $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$
```

performed on the active node. In the PageRank example, Algorithms 1 and 2 are both pull-based algorithms because an active node pulls (reads) its incoming neighbors' PageRank values and updates its own PageRank value. By the definition of PageRank, one might think that the pull-based implementation is the most natural design choice for PageRank computation. However, from the next subsections, we show that the basic data-driven PageRank can be improved by considering other design choices.

3.2 Pull-Push-based PageRank

In *pull-push-based* algorithms, an active node *pulls (reads)* its neighbors' values and also *pushes (updates)* its neighbors' values. When we consider the cost for processing an active node, pull-push-based algorithms might be more expensive than pull-based algorithms because it requires both *read* and *write* operations for the active node's neighbors. However, in terms of information propagation, the pull-push-based algorithms can have advantages over pull-based algorithms because in pull-push-based algorithms, an active node can propagate information to its neighbors whereas in pull-based algorithms, an active node passively receives information from its neighbors.

Now, we reformulate the basic data-driven PageRank as a pull-push-based algorithm. Recall that in Algorithm 2, whenever a node's PageRank is updated, the residual of each of its outgoing neighbors is increased. Thus, to guarantee that the maximum residual is smaller than ϵ, all the outgoing neighbors of an active node should be added to the worklist (see Theorem 2). However, an important observation is that if we explicitly compute the residual of each node, the worklist can be maintained more efficiently. That is, if each node explicitly maintains its residual, we do not need to add all the outgoing neighbors of an active node, instead, we add the outgoing neighbors only if their residuals are greater than or equal to ϵ. In this way, we can filter out some work in the worklist. See Algorithm 3 for details. For each active node, it pulls its incoming neighbors' PageRank values, and pushes residuals to its outgoing neighbors. Thus, Algoritm 3 can be referred as a pull-push-based algorithm. In this algorithm, there is a trade-off between overhead for residual computations and filtering out work in the worklist. We empirically observe that the benefit of filtering is greater than the overhead for computing residuals in many cases. As a result, the pull-push-based PageRank is more efficient than the basic data-driven PageRank.

3.3 Push-based PageRank

In *push-based* algorithms, an active node updates its own value, and only *pushes (updates)* its neighbors' values. Compared to pull-based algorithms, push-based algorithms can be more costly in the sense that it requires more *write* operations. However, push-based algorithms invoke more frequent updates on nodes' values, which might be helpful to achieve a faster information propagation over the network. In other words, in push-based algorithms, an active node aggressively propagates information to its neighbors.

It is not straightforward to formulate the PageRank computation as a push-based algorithm. To design a push-based PageRank, we need to closely look at the proof of Theorem 1. In particular, in the proof, (2) implies that the $(k+1)$-th PageRank update of node v is equivalent to the sum of the k-th PageRank of v and its k-th residual, i.e., $x_v^{(k+1)} = x_v^{(k)} + r_v^{(k)}$. This indicates that we can update PageRank of v by just adding its current PageRank value and the residual. For example, $x_v^{(1)} = x_v^{(0)} + r_v^{(0)}$. Notice that the initial residual $\mathbf{r}^{(0)}$ is defined to be $\mathbf{r}^{(0)} = (1-\alpha)\alpha \boldsymbol{P}^T\mathbf{e}$ as shown in (1). Thus, we can reformulate the basic data-driven PageRank as follows: we initialize $\mathbf{x}^{(0)}$ and $\mathbf{r}^{(0)}$, and for each active node v, we update its PageRank by $x_v^{(k+1)} = x_v^{(k)} + r_v^{(k)}$. And then, for each outgoing neighbor w of v, we update its residual to be $r_w = r_w + r_v\alpha/|\mathcal{T}_v|$. If r_w is greater than or equal to ϵ, we add w to the worklist. This procedure is repeated until the worklist becomes empty. Algorithm 4 shows the full procedure. In this algorithm, an active node updates its own PageRank and updates the residual of its outgoing neighbors, so it is a push-based algorithm.

The push-based PageRank only requires accessing outgoing neighbors of an active node. Thus, it can be more efficient than pull-push-based PageRank which requires accessing both incoming and outgoing neighbors. By rewriting the PageRank computation using residuals, the push-based PageRank also allows us to efficiently maintain the worklist because we only process the nodes whose residuals are greater than or equal to ϵ instead of adding all the outgoing neighbors (of an active node) to the worklist. As a result, compared to the pull-based PageRank, the push-based PageRank is also much more efficient.

4. SCHEDULING

Task scheduling, the order in which tasks are executed, can be very important to graph algorithms [10]. For some problems, such as single-source shortest-path, particular orders change the asymptotic behavior of the algorithm. For other problems, certain orders may enhance temporal locality. Implementing a good task scheduler involves a balance between the ordering requested by the user and the cost of achieving that order. Fewer ordering constraints allows cheaper and more scalable implementations. For example, per-thread FIFO or LIFO scheduling with workstealing is a commonly used task scheduling strategy due to its efficient implementation and good scalability for workloads which are relatively insensitive to task order.

4.1 Scheduling Motivation for PageRank

To see how the scheduling affects the convergence of PageRank, we look at Theorem 1 again. We see that in a data-driven PageRank, whenever a node v's PageRank is updated, the total residual is decreased at least by $r_v(1-\alpha)$. This might suggest that one should choose the node that has the largest residual to update next to achieve the maximum reduction in residual leading to a fast convergence. But, if we always select the maximum residual node to process, it might increase the total number of arithmetic operations because a large residual node is likely to have a large number of neighbors. For example, in the pull-push-based PageRank (shown in Algorithm 3), processing an active node requires accessing both incoming and outgoing neighbors, which implies that the number of arithmetic operations is proportional to the sum of the node's in-degree and out-degree. Thus, we incorporate a node's degree into our priority scheduling function. That is, we define a node v's priority p_v to be $p_v = r_v/d_v$ where $d_v = |\mathcal{S}_v| + |\mathcal{T}_v|$, i.e., we consider the residual per unit work. On the other hand, in the push-based PageRank (shown in Algorithm 4), processing an active node only requires accessing its outgoing neighbors. So, in this case, d_v is defined to be $d_v = |\mathcal{T}_v|$.

4.2 Priority Scheduling

Realizing the potential benefits in convergence noticed for PageRank requires priority scheduling. In priority scheduling, each task is assigned a value, the priority, and scheduled in increasing (or decreasing) order. More sophisticated schedulers allow modifying the priority of existing tasks, but this is an expensive operation not commonly supported in parallel systems.

Practical priority schedulers have to trade off several factors: efficiency, communication (and thus scaling), priority fidelity, and set-semantics. In general both priority fidelity and set-semantics require significant global knowledge and communication, thus are not scalable on non-trivial machines.

To investigate the sensitivity of PageRank to different design choices in a priority scheduler, we used two different designs: one which favored priority fidelity but gave up set-semantics, and one which preserved set-semantics at the expense of priority fidelity. These we compared with scalable non-priority schedulers to see if the improved convergence outweighted the increased cost of priority scheduling.

The first scheduler we used is the scalable, numa-aware OBIM priority scheduler from the Galois system ("numa" stands for non-uniform memory access) [9], [6]. This scheduler uses an approximate consensus protocol to inform a per-thread choice to search for stealable high-priroity work or to operate on local near-high-priority work. Various underlying data-structures and communion and stealing patterns are aware of the machine's memory topology and optimized to maximize information propogation while minimizing coherence cost. OBIM favors keeping all threads operating on high priority work and does not support either set-semantics or updating the priority of existing tasks. To handle this, tasks are created for PageRank everytime a node's priority changes, potentially generating duplicate tasks in the scheduler. Tasks with outdated priorities are quickly filtered out at execution time (a process which takes only a few instructions).

The second scheduler we use is a bulk-synchronous priority scheduler. This scheduler operates in rounds. Each round, all items with priority above a threshold are executed. Generated tasks and unexecuted items are placed in the next round. The range and mean are computed for the tasks, allowing the theshold to be chosen for each round based on the distribution of priorities observed for that round. This organization makes allowing priority updates simple, priorities are recomputed every round. Further, set semantics may be trivially maintained. However, to minimized the overhead of bulk-synchronous execution, each round must have sufficient work to amortize the barrier synchronization. This produces a schedule of tasks which may deviate noticeably from the user requested order.

5. RELATED WORK

In parallel computing community, the concepts of topology-driven and data-driven algorithms have been studied in the context of amorphous data-parallelism [11], where a new abstraction of parallel algorithms, called TAO analysis, has been introduced. The key concept of TAO analysis is to express a parallel algorithm as its action on data structures, which enables extracting important algorithmic properties for parallelism. Our approaches of dividing graph mining algorithms into three different algorithm choices (i.e., work activation, data access pattern, and scheduling) are mainly motivated by the TAO analysis. While TAO analysis has been proposed for a general parallel programming framework, our analysis and approaches are more geared towards developing new scalable data mining algorithms.

For scalable parallel computing, many different types of parallel programming models have been proposed, e.g., Galois [9], Ligra [13], GraphLab [7], Priter [17], and Maiter [18]. Actually, PageRank is a popular benchmark for parallel programming models, and almost every different parallel programming framework provides its own built-in implementation of PageRank. To show the new features a newly proposed parallel framework might support, various versions of PageRank have been implemented in different parallel platforms in a rather ad-hoc manner.

GraphLab is one of the most well-known parallel frameworks for scalable graph mining [7]. GraphLab abstraction is based on a vertex program which can be interpreted as a little program which is executed on a vertex. GraphLab supports both synchronous and asynchronous engines. For asynchronous execution, it provides several different types of queues with different scheduling policies. We compare GraphLab's PageRank implementations with our data-driven PageRank formulations and implementations on Galois [9]. We decide to use Galois to implement our PageRank algorithms because Galois supports more flexible data structures and more efficient data-driven parallelism.

The scheduling issue, which is presented as one of our three different algorithm design axis, has been recognized as a critical factor for designing a scalable graph mining algorithm. For example, the importance of prioritized execution in iterative algorithms has been studied in [17] where a distributed computing framework, called PrIter, has been developed. Also, in [18], it has been discussed that delta-based accumulative iterative computations can improve the scalability of graph-based algorithms in a distributed memory setting. This suggests that our push-based formulations have benefits over pull-based or pull-push-based formulations.

In data mining communities, PageRank has been extensively studied, and many different approximate algorithms have been developed over the years [3]. The Gauss-Seidel style update of PageRank is studied in [8]. Also, [1] can be seen as one possible instance of our pull-push-based formulations. From linear system perspective, parallel distributed PageRank also has been developed [5]. Since PageRank has been used in many different context and many variations have been formulated and proposed, we might not be able to exhaustively enumerate all of them. However, our main contribution of this paper is not only proposing a new PageRank formulation, but also presenting a more general approach for formulating a scalable data-driven algorithm (PageRank is just used for a case study).

6. EXPERIMENTAL RESULTS

To see the performance sensitivity of PageRank to the design considerations in this paper, we implement and evaluate a variety of data-driven and topology-driven PageRank implementations, trying different schedulings and data-access patterns. All implementations are written using the Galois System [9]. We compare our results to a third-party base-line, namely GraphLab, varying such parameters as are available in that implementation. We use for reference the implementations of PageRank shipped with GraphLab and Galois.

6.1 Datasets

We test on six real-world networks, given in Table 1. pld and sd1 are hyperlink graphs and Twitter and Friendster graphs are social networks. These graphs range in size from about 600M edges to 3.6B edges. These range in size for in-memory compressed sparse row representations from 2.7GB to 14GB for the directed graph. Most of the algorithms require both in-edges and out-edges tracked, making the effective in-memory size approximately twice as large. The sources of these data sets are given in the table.

	No. of nodes	No. of non-zeros	CSR size
pld [1]	39,497,204	623,056,312	2.7G
sd1 [1]	82,924,685	1,937,489,264	7.9G
twitter-rv [2]	41,652,230	1,468,364,884	5.8G
twitter-40m [2]	41,652,230	2,405,026,092	9.3G
twitter-50m [3]	51,161,011	3,227,785,184	13G
friendster-67m [4]	67,492,106	3,622,565,232	14G

Table 1: Input Graphs

6.2 Experimental Setup

For all experiments, we use $\alpha = 0.85$, $\epsilon = 0.01$. That is, we run each algorithm until the maximum update to PageRank is less than 0.01. We use a 4 socket Xeon E7-4860 running at 2.27GHz with 10 core per socket, 24MB L3 cache, and 128GB ram. GraphLab was run in multi-threaded mode on this machine, rather than using distributed memory. All code was compiled with gcc 4.9. Experiments were run under Linux with hugepages enabled.

For each input we present:

- Time: Runtime of each method as a function of threads
- Scalability: $\frac{\text{run-time of method } m \text{ with a single thread}}{\text{run-time of method } m \text{ with } t \text{ threads}}$
- Speedup: $\frac{\text{run-time of the fastest single-thread method}}{\text{run-time of method } m \text{ with } t \text{ threads}}$

Algorithm	Activation	Access	Schedule
dd-push	Data-driven	Push	FIFOs w/ Stealing
dd-push-prs	Data-driven	Push	Bulk-sync Priority
dd-push-prt	Data-driven	Push	Async Priority
dd-pp-rsd	Data-driven	Push-Pull	FIFOs w/ Stealing
dd-pp-prs	Data-driven	Push-Pull	Bulk-sync Priority
dd-pp-prt	Data-driven	Push-Pull	Async Priority
dd-basic	Data-driven	Pull	FIFOs w/ Stealing
tp	Topology	Pull	Load Balancer

Table 2: Summary of algorithm design choices

6.3 Results

Table 2 summarizes the design choices for each tested implementation. Pseudo-code and more detailed discussions of each appear in previous sections.

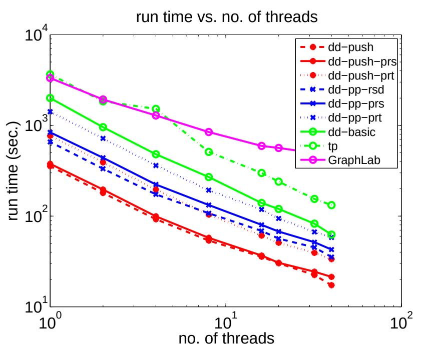
Figures 1 and Figure 2 show the time, scalability, and speedup of the various algorithms. We note that GraphLab ran out of memory for all but the smallest (pld) input. While it was approximately the same performance as the closest Galois implementation, tp, it scaled significantly worse. The best data-driven implementation was 10-20x faster, depending on the thread count.

Several broad patterns can be seen in the results. First, all the data-driven implementations outperform the topology implementation. Second, push-only implementations outperform push-pull implementations which outperform a pure pull-based version. Finally, priority-scheduled versions scale better, though perform worse than using a fast, non-priority scheduler.

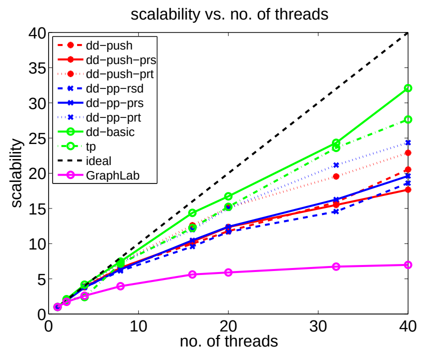
One surprising result is that pulling to compute PageRank and pushing residual outperforms a pure pull-based version. The read-mostly nature of pull-based algorithms are generally more cache friendly. Push-based algorithms have a much larger write-set per iteration, and writes to common locations fundamentally don't scale. The extra cost of the pushs, however, is made up for in a reduction in the number of iterations. The pushing of residual allows a node to selectively activate a neighbor only if enough of it's in-edges have changed that it would compute a PageRank update greater than the threshold. This filtering greatly reduces the total work performed, as effectively PageRanks are only computed when they are needed. The pure pull version must compute them every time they might be significant.

It is more understandable, though, that the push-only version outperforms all others. The pushing of residual is equivalent to the computation of PageRank deltas, thus, the pull can be eliminated, with no extra cost. This both reduces the number of edges inspected for every node, from in and out to just out, and reduces the total computation.

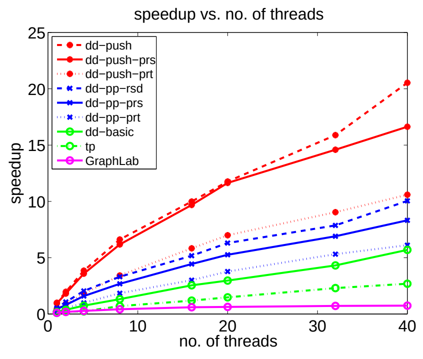
[1] `http://webdatacommons.org/hyperlinkgraph/`
[2] `http://an.kaist.ac.kr/traces/WWW2010.html`
[3] `http://twitter.mpi-sws.org/`
[4] `https://archive.org/details/friendster-dataset-201107`



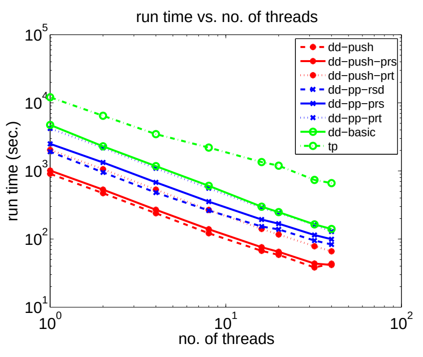
(a) pld run time



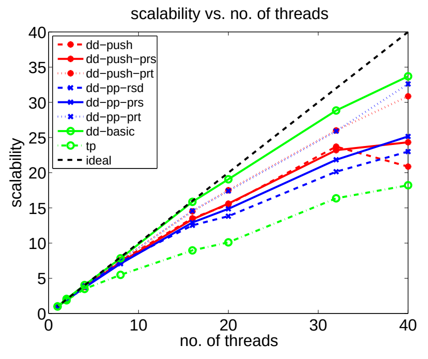
(b) pld scalability



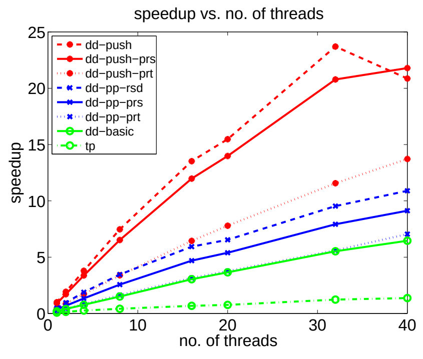
(c) pld speedup



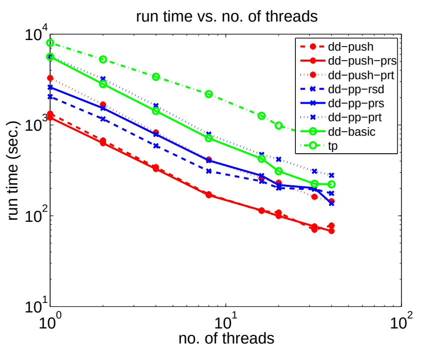
(d) sd1 run time



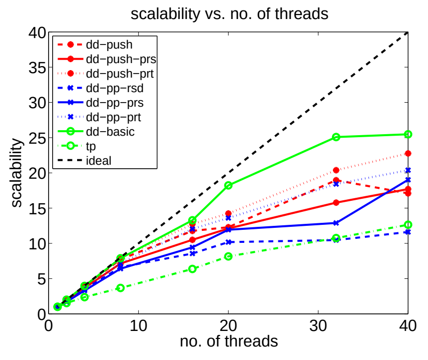
(e) sd1 scalability



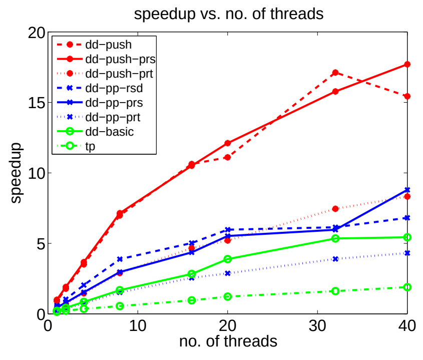
(f) sd1 speedup



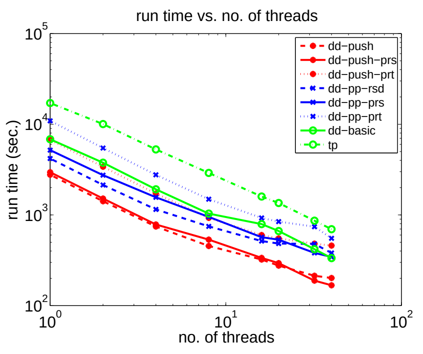
(g) twitter-rv run time



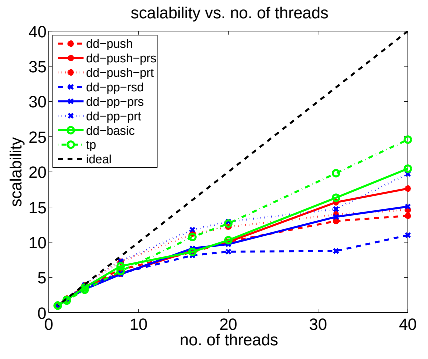
(h) twitter-rv scalability



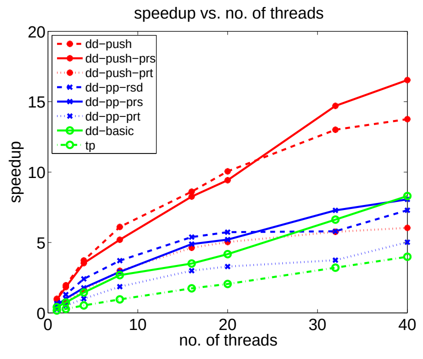
(i) twitter-rv speedup



(j) twitter-40m run time

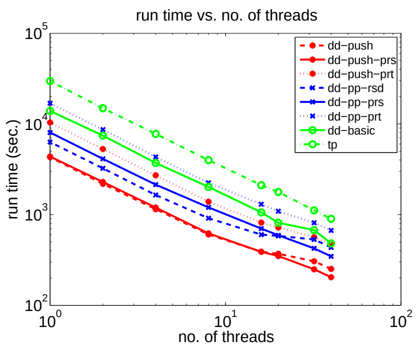


(k) twitter-40m scalability

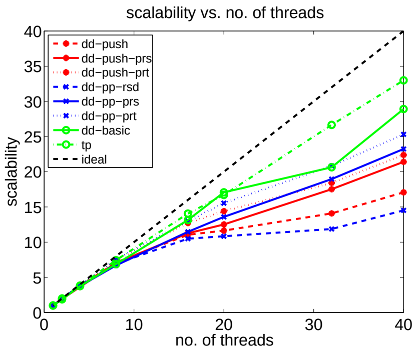


(l) twitter-40m speedup

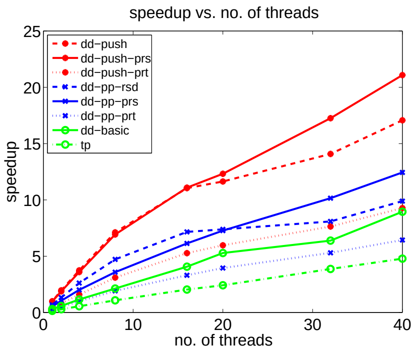
Figure 1: Runtime, scalability and speedup on pld, sd1, twitter-rv, and twitter-40m graphs. Our data-driven, push-based PageRank achieves the best speedup.



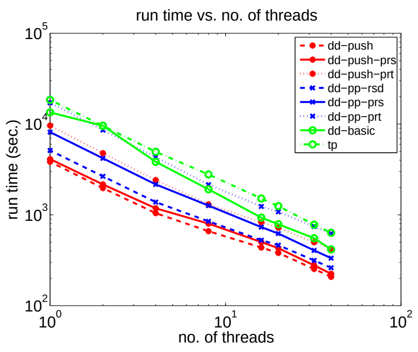
(a) twitter-50m run time



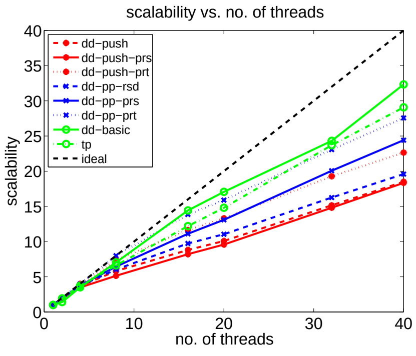
(b) twitter-50m scalability



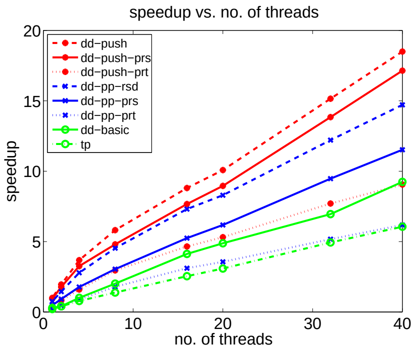
(c) twitter-50m speedup



(d) friendster-67m run time



(e) friendster-67m scalability



(f) friendster-67m speedup

Figure 2: Runtime, scalability and speedup on twitter-50m and friendster-67m graphs.

When looking at scheduling, we see that the costs of priority scheduling are not made up for by the improvement in convergence. For most inputs, the general scheduler outperforms priority scheduling. We see, however, a marked difference between the two priority scheduling strategies. The "-prt" scheduler favors following priority at the cost of set-semantics. We observe that priorities on nodes increase until the node is processed, causing items to be duplicated at multiple priority levels. Although filtering out duplicates is not expensive, the total work doing so is significant. The "-prs" scheduler favors set semantics and allows priority updates. This proves to be more important for the pattern of priorities in PageRank. This scheduler is often near the best performing implementation.

Table 3 shows a direct comparison of several variants to GraphLab's implementation when varying the scheduling. GraphLab supports different schedulers, though we find the simple synchronous one the best. It should be noted that unlike our implementation (tp), GraphLab only runs in each synchronous round nodes which may update their value, effectively filtering out some portion of the graph. While we do not consider this to be a pure data-driven asynchronous implementation, it is a notable improvement over the simple "do everything" approach of the tp implementation.

We should also note that the GraphLab asynchronous refers to a Gauss-Seidel style solver, which still is a bulk-synchronous, topology-driven approach. The tp version is actually a classic synchronous implementation in this sense, but still notably faster. The GraphLab version within each bulk-synchronous round filters out update computations which would result in the same PageRank value as already exist. The tp version doesn't do this, rather it performs a more classic update to all nodes each round.

7. DISCUSSION

Priority scheduling needs some algorithmic margin be competitive as it is slower. While it isn't surprising that priority scheduling is slower than simple scalable scheduling, this has some important consequences. First, the benefit is dependent on both algorithmic factors and input characteristics. When scheduling changes the asymptotic complexity of an algorithm, there can be huge margins available. In PageRank, there is a theoretical margin available, but it is relatively small. This limits the extra computation that can be spent on scheduling overhead without hurting performance. Secondly, the margin available depends on input characteristics. For many analytic algorithms, scheduling increases in importance as the diameter of the graph increases. Since PageRank is often run on power-law style graphs with low diameter, we expect a small margin available from priority scheduling.

Good priority schedulers can scale competitively with general purpose schedulers. We observe that multiple priority scheduler implementations scale well. We implement two very different styles of priority schedulers which pick different points in the design and feature space. This is encouraging as it leads us to believe that such richer semantic building blocks can be used by algorithm designers.

PageRank updates priorities often, a use case which is

		GraphLab					Galois			
		sync	async-fifo	async-qfifo	async-sweep	async-prt	tp	dd-basic	dd-pp-prt	dd-push
Time	40 threads	478 sec	500 sec	788 sec	4,186 sec	> 4 hrs	132 sec	62 sec	58 sec	17 sec
	32 threads	496 sec	580 sec	804 sec	5,162 sec	> 4 hrs	155 sec	82 sec	67 sec	22 sec
	16 threads	594 sec	618 sec	970 sec	9,156 sec	> 4 hrs	299 sec	140 sec	118 sec	36 sec
	8 threads	845 sec	898 sec	1,292 sec	> 4 hrs	> 4 hrs	510 sec	269 sec	193 sec	53 sec
	1 thread	3,332 sec	5,194 sec	5,098 sec	> 4 hrs	> 4 hrs	3,650 sec	2,004 sec	1,415 sec	355 sec

Table 3: Different PageRank implementations in GraphLab and Galois

hard to support efficiently and scalably. Even many high-performance, serial priority queues don't support this operation. Constructing a concurrent, scalable priority scheduler which maintains set semantics by adjusting priorities for existing items in the scheduler is an open question. The reason is simply one of global knowledge. Knowing whether to insert an item or whether it is already scheduled and thus only needs it's priority adjusted requires global knowledge of the system. Maintaining and updating global knowledge concurrently in a NUMA system is rarely scalable. For scalability, practical implementations will contain multiple queues, meaning that not only does one need to track whether a task is scheduled, but on which queue the task is scheduled. The scheduler we produced for *-prs stores set semantics information by marking nodes in the graph and periodically rechecks priority. This essentially introduces latency between updating a priority and having the scheduler see the new priority. The amount of latency depends on how many iterations proceed before rechecking. This number determines the overhead of the scheduler.

One surprising result is that optimizing for cache behavior (pull-based) may not always be as effective as optimizing for pushing maximum information quickly (push-based). The push-only pagerank, dd-push-*, touches, on average, the same amount of data as the pull-only, dd-basic, but is entirely read-write access, while the pull-only version does one write per node processed. In general, read-mostly access patters are significantly more cache and coherence friendly. From this perspective, the push-pull versions, dd-pp-*, should be worst as they have the read set of the pull versions and the write set of the push versions. The extra writes are not just an alternate implementation of the PageRank update, but rather influence the scheduling of tasks. The "extra" writes weight nodes, allowing nodes to only be processed when profitable. This improved scheduling makes up for the increased write load by each iteration. Given the scheduling benefits of the residual push, it is easy to see that the push-only version is superior to the push-pull version as it has the same scheduling approach, but reduces the memory load and work per iteration.

Even for similar algorithm implementations, performance of graph frameworks varies wildly. This has been widely observed, see, for example, [12], [9]. However, from a practical standpoint, this makes understanding algorithmic changes hard to understand. The performance results of algorithm changes becomes very dependent on the implementation of the underlying runtime. We see in the results different performance and scalability between topology-driven Galois implementations and GraphLab's more optimized topology-driven implementation. We also observe that a global-barrier free data-driven implementation, although significantly faster, is not even possible in many graph analytic frameworks.

Even though we have focused on our discussion on PageRank in this manuscript, our approach can easily be extended to other data mining algorithms. For example, in a graph-based semi-supervised learning, label propagation is a well-known method [2]. Given a set of data points where only a small subset of the data points is labelled, i.e., only limited number of data points have class labels, the goal of the semi-supervised learning is to classify the unlabelled data points to the correct classes. Label propagation is a popular method to achieve this goal. In label propagation, a similarity graph is constructed based on similarity values between data points. In the graph, nodes represent data points and edges represent similarities between the data points. Then, known labels are propagated through the network until all the node labels are stabilized. Many different types of label propagation have been proposed, but the basic mechanism of the label propagation methods is very similar to PagaRank computation in the sense that for each active node, label propagation algorithms also involve reading its neighbors' labels and propagating (updating) its neighbors' labels until the convergence. Thus, we can think of developing a scalable data-driven label propagation algorithm using the approaches we propose in this paper.

Also, it has been shown that there is a close relationship between personalized PageRank and community detection [14], [1]. So, parallel data-driven community detection can also be another interesting application of our analysis about data-driven PageRank.

8. CONCLUSION

Although PageRank is a simple graph analytic algorithm, there are many interesting implementation details one needs to consider to achieve a high-performance implementation. We show that data-driven implementations are significantly faster than traditional power iteration methods. While not surprising given we know many convergence algorithms can be implemented this way, one has to ensure that convergence still holds when making this kind of transform. PageRank has a simple vertex update equation. However, this update can be mapped to the graph in several ways, changing how and when information flows through the graph, which vary significantly in performance. Within this space, one can also profitably consider the order in which updates occur to maximize convergence speed. While we investigate these implementation variants for PageRank, seeing performance improvements of 10x over standard power iterations, these considerations apply to any convergence-based graph analytic algorithm.

Acknowledgments

This research was supported by NSF grants CCF-1117055 and CCF-1320746.

APPENDIX

We show the convergence proof of the basic data-driven PageRank which is shown in Algorithm 2.

PROOF OF THEOREM 1. We define a row-stochastic matrix $\boldsymbol{P}$ to be $\boldsymbol{P} \equiv \boldsymbol{D}^{-1}\boldsymbol{A}$ where $\boldsymbol{A}$ is an adjacency matrix and $\boldsymbol{D}$ is the degree diagonal matrix. We assume that there is no self-loop in the graph, i.e., the diagonal elements of $\boldsymbol{A}$ are all zeros. Let $\mathbf{x}$ denote a PageRank vector (x_v denotes PageRank of node v), and $\mathbf{e}$ denote a vector having all the elements equal to one. Then, the PageRank computation can be written as follows:

$$\mathbf{x} = \alpha \boldsymbol{P}^T \mathbf{x} + (1-\alpha)\mathbf{e}.$$

This is the linear system of

$$(\boldsymbol{I} - \alpha \boldsymbol{P}^T)\mathbf{x} = (1-\alpha)\mathbf{e}.$$

and the residual is defined to be

$$\mathbf{r} = (1-\alpha)\mathbf{e} - (\boldsymbol{I} - \alpha\boldsymbol{P}^T)\mathbf{x} = \alpha\boldsymbol{P}^T\mathbf{x} + (1-\alpha)\mathbf{e} - \mathbf{x}.$$

Let $x_v^{(k)}$ denote the k-th update of x_v. In Algorithm 2, we initialize $\mathbf{x}$ to be $\mathbf{x}^{(0)} = (1-\alpha)\mathbf{e}$. Thus, the initial residual $\mathbf{r}^{(0)}$ can be written as follows:

$$\mathbf{r}^{(0)} = (1-\alpha)\mathbf{e} - (\boldsymbol{I} - \alpha\boldsymbol{P}^T)(1-\alpha)\mathbf{e} = (1-\alpha)\alpha\boldsymbol{P}^T\mathbf{e} \geq 0. \quad (1)$$

We can see that the initial residual is positive. Let us define $\mathcal{S}_v$ to be a set of incoming neighbors of node v, and $\mathcal{T}_v$ to be a set of outgoing neighbors of node v. In Algorithm 2, for each active node v, we update its PageRank as follows:

$$x_v^{(k+1)} = (1-\alpha) + \alpha \sum_{w \in \mathcal{S}_v} \frac{x_w^{(k)}}{|\mathcal{T}_w|}$$

Indeed, this is equivalent to:

$$x_v^{(k+1)} = x_v^{(k)} + r_v^{(k)}. \quad (2)$$

because:

$$x_v^{(k+1)} = x_v^{(k)} + r_v^{(k)} = x_v^{(k)} + \underbrace{(1-\alpha) - x_v^{(k)} + \alpha[\boldsymbol{P}^T\mathbf{x}^{(k)}]_v}_{r_v^{(k)}}.$$

Also, after such an update, we can show that $\mathbf{r}^{(k+1)} \geq 0$. Let $\gamma = \mathbf{r}_v^{(k)}$. Then,

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \gamma\mathbf{e}_v$$

$$\mathbf{r}^{(k+1)} = (1-\alpha)\mathbf{e} - (\boldsymbol{I} - \alpha\boldsymbol{P}^T)\mathbf{x}^{(k+1)}$$

$$\mathbf{r}^{(k+1)} = (1-\alpha)\mathbf{e} - (\boldsymbol{I} - \alpha\boldsymbol{P}^T)(\mathbf{x}^{(k)} + \gamma\mathbf{e}_v)$$

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \gamma(\boldsymbol{I} - \alpha\boldsymbol{P}^T)\mathbf{e}_v \quad (3)$$

Note that the vth component of $\mathbf{r}^{(k+1)}$ goes to zero, and we only add positive values to the other components. Recall that the initial residual is positive shown in (1). Thus, by induction, we can see that $\mathbf{r}^{(k+1)} \geq 0$.

Now, by multiplying $\mathbf{e}^T$ in (3), we get:

$$\mathbf{e}^T\mathbf{r}^{(k+1)} = \begin{cases} \mathbf{e}^T\mathbf{r}^{(k)} - r_v^{(k)}(1-\alpha) & : \mathcal{T}_v \neq \emptyset \\ \mathbf{e}^T\mathbf{r}^{(k)} - r_v^{(k)} & : \mathcal{T}_v = \emptyset \end{cases}$$

Thus, any step decreases the residual by at least $\gamma(1-\alpha)$, and moves $\mathbf{x}$ closer to a solution. □

A. REFERENCES

[1] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using PageRank vectors. In *FOCS*, 2006.

[2] Y. Bengio, O. Delalleau, and N. Le Roux. Label propagation and quadratic criterion. In *Semi-Supervised Learning*, pages 193–216. MIT Press, 2006.

[3] P. Berkhin. A survey on pagerank computing. *Internet Mathematics*, 2:73–120, 2005.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[5] D. F. Gleich, L. Zhukov, and P. Berkhin. Fast parallel PageRank: A linear system approach. Technical Report YRL-2004-038, Yahoo! Research Labs, 2004.

[6] A. Lenharth, D. Nguyen, and K. Pingali. Priority queues are not good concurrent priority schedulers. In *UTCS Technical Report, TR-11-39*, 2011.

[7] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *VLDB Endowment*, 2012.

[8] F. McSherry. A uniform approach to accelerated pagerank computation. In *WWW*, 2005.

[9] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[10] D. Nguyen and K. Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *ASPLOS*, 2011.

[11] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. MÃl'ndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI*, pages 12–25, 2011.

[12] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *ACM SIGMOD*, pages 979–990, 2014.

[13] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[14] J. J. Whang, D. Gleich, and I. S. Dhillon. Overlapping community detection using seed set expansion. In *CIKM*, pages 2099–2108, 2013.

[15] J. J. Whang, X. Sui, and I. S. Dhillon. Scalable and memory-efficient clustering of large-scale social networks. In *ICDM*, pages 705–714, 2012.

[16] J. Yang and J. Leskovec. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *WSDM*, pages 587–596, 2013.

[17] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritizing iterative computations. *IEEE Transactions on Parallel and Distributed Systems*, 24(9):1884–1893, 2013.

[18] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Transactions on Parallel and Distributed Systems*, 25(8):2091–2100, 2014.