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

Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S. Dhillon, and Keshav Pingali

University of Texas at Austin, Austin TX 78712, USA,
{joyce,lenharth,inderjit,pingali}@{cs,ices,cs,cs}.utexas.edu

Abstract. Large-scale network and graph analysis has received considerable attention recently. Graph mining techniques often involve an iterative algorithm, which can be implemented in a variety of ways. Using PageRank as a model problem, we look at three algorithm design axes: work activation, data access pattern, and scheduling. We investigate the impact of different algorithm design choices. Using these design axes, we design and test a variety of PageRank implementations finding that data-driven, push-based algorithms are able to achieve more than 28x the performance of standard PageRank implementations (e.g., those in GraphLab). The design choices affect both single-threaded performance as well as parallel scalability. The implementation lessons not only guide efficient implementations of many graph mining algorithms, but also provide a framework for designing new scalable algorithms.

Keywords: scalable computing, graph analytics, PageRank, multi-threaded programming, data-driven algorithm

1 Introduction

Large-scale graph analysis has received considerable attention in both the machine learning and parallel programming communities. In machine learning, many different types of task-specific algorithms have been developed to deal with massive networks. In parallel computing, many different parallel programming models and systems have been proposed for both shared memory and distributed memory settings to ease implementation and manage parallel programs.

Recent research has observed that distributed graph analytics can have a significant slowdown over shared-memory implementations, that is, the increase in communication costs are not easily made up for by increase in aggregate processing power or memory bandwidth. Furthermore, a remarkable number of “large” graphs fit in the main memory of a shared memory machine; it is easy to fit graphs with tens of billions of edges on a large workstation-class machine. 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 help distributed implementations, as they tend to use shared-memory abstractions within a host.

Many graph mining techniques usually involve iterative algorithms where local computations are repeatedly done at a set of nodes until a convergence criterion is satisfied. Let us define *active nodes* to be a set of nodes where computations should be performed. Based on how the active nodes are processed, we can broadly classify these iterative graph algorithms from three different points of view: work activation, data access pattern, and scheduling. In this paper, we present general approaches for designing scalable data-driven graph algorithms using a case study of the PageRank algorithm. In particular, using the three different algorithm design axes (i.e., work activation, data access pattern, and scheduling), we present eight different formulations and in-memory parallel implementations of PageRank algorithm. We show that by considering data-driven formulations, we can have more 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.

2 Work Activation

We first classify algorithms into two groups based on work activation: topology-driven and data-driven algorithms. In a topology-driven algorithm, active nodes are defined solely by the structure of a graph. For example, an algorithm which requires processing all the nodes at each iteration is referred to as a topology-driven algorithm. On the other hand, in a data-driven algorithm, the nodes are dynamically activated by their neighbors, i.e., the 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 allow us to concentrate more on "hot spots" in a graph where more frequent updates are needed.

2.1 Topology-driven PageRank

To explain the concepts in more detail, we now focus our discussion on PageRank which is a key technique in web mining [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 the set of incoming neighbors of node v, and $\mathcal{T}_v$ to be the 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 iterate, and α is a teleportation parameter $(0 < \alpha < 1)$. Algorithm 1 presents this iteration, which is the traditional power method that can be used to compute PageRank. Given a user defined tolerance ϵ, the PageRank vector $\mathbf{x}$ is initialized to be $\mathbf{x} = (1-\alpha)\mathbf{e}$ where $\mathbf{e}$ denotes the vector of all 1's. 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, it is a topology-driven algorithm.

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: $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$
```

2.2 Basic Data-driven PageRank

Instead of processing all the nodes in rounds, we can think of an algorithm which dynamically maintains a working set. Algorithm 2 shows a basic data-driven PageRank. Initially, the worklist is set to be the entire vertex set. The algorithm proceeds by picking a node from the worklist, computing the node's PageRank, and adding its outgoing neighbors to the worklist. To examine 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. Then, the PageRank problem requires solving the linear system $(\boldsymbol{I} - \alpha\boldsymbol{P}^T)\mathbf{x} = (1-\alpha)\mathbf{e}$, and the residual is defined to be $\mathbf{r} = (1-\alpha)\mathbf{e} - (\boldsymbol{I} - \alpha\boldsymbol{P}^T)\mathbf{x}$. In this setting, it has been shown in [9] that each local computation in Algorithm 2 decreases the residual. Indeed, when a node v's PageRank is updated, its residual r_v becomes zero, and it can be shown that $\alpha r_v/|\mathcal{T}_v|$ is added to each of its outgoing neighbors' residuals. Thus, we can show that Algorithm 2 converges, and on termination, it is guaranteed that the residual $\|\mathbf{r}\|_\infty < \epsilon$. From the next section, we will focus on the data-driven formulation of PageRank, and build up various variations.

3 Data Access Pattern

Data access pattern (or memory access pattern) is an important factor in designing a scalable graph algorithm. 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, we can classify algorithms into three categories: pull-based, pull-push-based, and push-based algorithms.

Algorithm 3. Pull-Push-based 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:     $x_v = \alpha \sum_{w \in \mathcal{S}_v} \frac{x_w}{|\mathcal{T}_w|} + (1-\alpha)$
15:     for $w \in \mathcal{T}_v$ do
16:         $r_w^{old} = r_w$
17:         $r_w = r_w + \frac{r_v \alpha}{|\mathcal{T}_v|}$
18:         if $r_w \geq \epsilon$ and $r_w^{old} < \epsilon$ then
19:             worklist.push($w$)
20:         end if
21:     end for
22:     $r_v = 0$
23: end while
24: $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$
```

Algorithm 4. Push-based 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:     $x_v^{new} = x_v + r_v$
15:     for $w \in \mathcal{T}_v$ do
16:         $r_w^{old} = r_w$
17:         $r_w = r_w + \frac{r_v \alpha}{|\mathcal{T}_v|}$
18:         if $r_w \geq \epsilon$ and $r_w^{old} < \epsilon$ then
19:             worklist.push($w$)
20:         end if
21:     end for
22:     $r_v = 0$
23: end while
24: $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$
```

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 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.

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 as they require both *read* and *write* operations on neighbors. However, in terms of information propagation, pull-push-based algorithms can have advantages 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 transform the basic data-driven PageRank into a pull-push-based algorithm. In Algorithm 2, whenever a node's PageRank is updated, the residuals of its outgoing neighbors are 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. However, if we explicitly compute and maintain the residuals, we do not need to add all the outgoing neighbors of an active node, instead, we only need to add the outgoing neighbors whose residuals become greater than or equal to ϵ. In this way, we can filter out some work in the worklist. In Algorithm 3, the initial residual $\mathbf{r}^{(0)}$ is computed by $\mathbf{r}^{(0)} = (1-\alpha)\alpha \boldsymbol{P}^T \mathbf{e}$ (lines 3–8). Each active node pulls its incoming neighbors' PageRank values (line 14), and pushes residuals to its outgoing neighbors (line 17). Then, an outgoing neighbor w of the active node v is added to the worklist only if the updated residual r_w is greater than or equal to ϵ and its old residual is less than ϵ. The second condition allows us to avoid having duplicates in the worklist (i.e., we add a node to the worklist only when its residual crosses ϵ). In this algorithm, there is a trade-off between overhead for residual computations and filtering out work in the worklist. We empirically observe that in many cases, the benefit of filtering overcomes the overhead for residual computations.

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 they require more *write* operations. However, push-based algorithms invoke more frequent updates, which might be helpful to achieve a faster information propagation over the network. Compared to pull-push-based algorithms, push-based algorithms can be more efficient because they only require *write* operations instead of *read* & *write* operations. To design a push-based PageRank, we need to notice that the $(k+1)$-st PageRank update of node v is equivalent to the sum of the k-th PageRank of v and its k-th residual. This can be derived from the linear system formulation discussed in Section 2.2. Thus, we can formulate a push-based PageRank as follows: for each active node v, its PageRank is updated by $x_v^{(k+1)} = x_v^{(k)} + r_v^{(k)}$. Algorithm 4 shows the full procedure. Note that the only difference between Algorithm 3 and Algorithm 4 is line 14. In Algorithm 4, an active node updates its own PageRank and the residuals of its outgoing neighbors.

4 Scheduling

Task scheduling, the order in which tasks are executed, can be very important to graph algorithms [11]. For example, in data-driven PageRank, we see that whenever a node v has residual r_v, and its PageRank is then updated, the total residual is decreased "at least" or "exactly" by $r_v(1-\alpha)$. This suggests that if we process "large residual" nodes first, the algorithm might converge faster. Thus, we can define a node v's priority p_v to be the residual per unit work, i.e., $p_v = r_v/d_v$ where $d_v = |\mathcal{S}_v| + |\mathcal{T}_v|$ for the pull-push-based PageRank, and $d_v = |\mathcal{T}_v|$ in the push-based algorithm. Realizing the potential benefits in convergence 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 (here, the "set-semantics" means that there are no duplicate work items in the worklist). In general, both priority fidelity and set-semantics require significant global knowledge and communication, thus are not scalable. To investigate the sensitivity of PageRank to different design choices in a priority scheduler, we use two different designs: one which favors priority fidelity but gives up set-semantics and one which preserves set-semantics at the expense of priority fidelity. We compare these with scalable non-priority schedulers to see if the improved convergence outweighs the increased cost of priority scheduling.

The first scheduler we use is the scalable, NUMA-aware OBIM (ordered-by-integer-metric) priority scheduler [7]. This scheduler uses an approximate consensus protocol to inform a per-thread choice to search for stealable high-priority work or to operate on local near-high-priority work. Various underlying data structures and stealing patterns are aware of the machine's memory topology and optimized to maximize information propagation while minimizing cache 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 every time 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 consumes 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 threshold to be chosen for each round based on the distribution of priorities observed for that round. This organization makes priority updates simple; priorities are recomputed every round. Further, set-semantics may be trivially maintained. However, to minimize 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.

We also consider FIFO- and LIFO-like schedules (parallel schedulers cannot both scale and preserve exact FIFO and LIFO order). It is obvious that a LIFO scheduler is generally bad for PageRank. Processing nodes after a single neighbor is visited will process the node once for each in-neighbor. FIFO schedulers provide time for a node to accumulate pending changes from many neighbors before being processed. We use a NUMA-aware scheduler, similar to that from Galois and QThreads, to do scalable, fast FIFO-like scheduling.

5 Related Work

Our approaches of considering three different algorithm design axes are mainly motivated by the Tao analysis [12] where the concepts of topology-driven and data-driven algorithms have been studied in the context of amorphous data-parallelism. While Tao analysis has been proposed for a general parallel programming framework, our analysis is geared more towards designing new scalable data mining algorithms.

For scalable parallel computing, many different types of parallel programming models have been proposed, e.g., Galois [10], Ligra [13], GraphLab [8], Priter [15], and Maiter [16]. Since PageRank is a popular benchmark for parallel programming models, various versions of PageRank have been implemented in different parallel platforms in a rather ad hoc manner. Also, in data mining communities, PageRank has been extensively studied, and many different approximate algorithms (e.g., [1], [6]) have been developed over the years [3]. The Gauss–Seidel style update of PageRank is studied in [9], and parallel distributed PageRank also has been developed [5]. Our PageRank formulations can be considered as variations of these previous studies. Our contribution in this paper is to systematically analyze and discuss various PageRank implementations with the perspective of designing scalable graph mining methodologies.

Even though we have focused our discussion on PageRank in this manuscript, our approaches can be easily extended to other data mining algorithms. For example, in semi-supervised learning, label propagation is a well-known method [2] which involves fairly similar computations as PageRank. We expect that our data-driven formulations can be applied to the label propagation method. 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 be another interesting application of our analysis.

6 Experimental Results

Experimental Setup. To see the performance and scaling sensitivity of PageRank to the design considerations in this paper, we implement a variety of PageRank algorithms, trying different scheduling and data access patterns. All implementations are written using the Galois System [10]. Table 1 summarizes the design choices for each implementation. Pseudo-code and more detailed discussions of each appear in previous sections. We also compare our results to a third-party baseline, namely GraphLab, varying such parameters as are avail-

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	Pull-Push	FIFOs w/ Stealing
dd-pp-prs	Data-driven	Pull-Push	Bulk-sync Priority
dd-pp-prt	Data-driven	Pull-Push	Async Priority
dd-basic	Data-driven	Pull	FIFOs w/ Stealing
power-iter	Topology	Pull	Load Balancer

Table 1. Summary of algorithm design choices

	# nodes	# edges	CSR size	source
pld	39M	623M	2.7G	webdatacommons.org/hyperlinkgraph/
sd1	83M	1,937M	7.9G	webdatacommons.org/hyperlinkgraph/
Twitter	51M	3,228M	13G	twitter.mpi-sws.org/
Friendster	67M	3,623M	14G	archive.org/details/friendster-dataset-201107

Table 2. Input Graphs

	dd-push	dd-push-prs	dd-push-prt	dd-pp-rsd	dd-pp-prs	dd-pp-prt	dd-basic	power-iter
sd1	20.9	21.8	13.7	10.9	9.1	7.0	6.5	1.4
frd	18.5	17.1	9.0	14.7	11.5	6.2	9.2	6.1

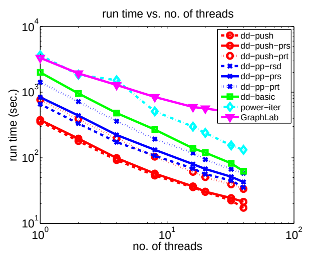
Table 3. Speedup on 40 threads relative to best serial on sd1 and friendster (frd)

able in that implementation. For all experiments, we use $\alpha = 0.85$, $\epsilon = 0.01$. We use a 4 socket Xeon E7-4860 running at 2.27GHz with 10 cores per socket and 128GB RAM. GraphLab was run in multi-threaded mode.

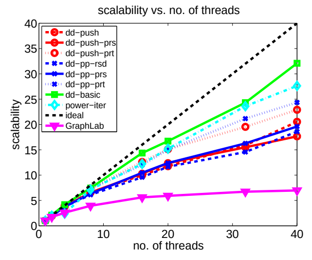
Datasets. We use four real-world networks, given in Table 2. Twitter and Friendster are social networks, and pld and sd1 are hyperlink graphs. These graphs range from about 600 million edges to 3.6 billion edges. These range in size for in-memory compressed sparse row representations from 2.7GB to 14GB for the directed graph. Most of the algorithms require tracking both in-edges and out-edges, making the effective in-memory size approximately twice as large.

Results. Figure 1 shows runtime, self-relative scalability, and speedup against the best single-threaded algorithm for the pld and twitter graphs. In Table 3, the final speedups are shown on the other inputs. We note that GraphLab ran out of memory for all but the smallest (pld) input. On pld, the serial GraphLab performance was approximately the same as the closest Galois implementation, power-iter, but GraphLab scaled significantly worse. Several broad patterns can be seen in the results. First, all data-driven implementations outperform topology implementation. The best data-driven PageRank implementation is 28x faster than GraphLab, and 10-20x faster than Galois power-iter, depending on the thread count. Second, push-only implementations outperform pull-push implementations which outperform a pure pull-based version. Finally, priority-scheduled versions scale better but perform worse than a fast, non-priority scheduler.

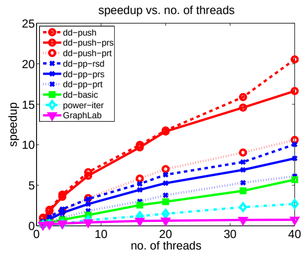
One surprising result is that pulling to compute PageRank and pushing residuals outperforms a pure pull-based version (dd-pp-* vs. dd-basic). The read-mostly nature of pull-based algorithms are generally more cache friendly. Push-based algorithms have a much larger write-set per iteration, and writes to common locations fundamentally do not scale. The extra cost of the pushes, however, is made up by a reduction in the number of tasks. Table 4 shows the number of completed tasks for each algorithm, and we see that pull-push methods (dd-pp-rsd) lead to 70-80% reduction in the number of tasks executed (compared to dd-basic). The pushing of residual allows a node to selectively activate a neighbor, and thus greatly reduces the total work performed (effectively, PageRanks are only computed when they are needed). On the other hand, the basic pull algorithm must unconditionally generate tasks for each of a node's neighbors



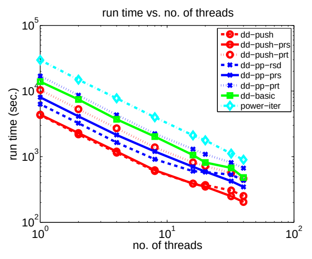
(a) pld run time



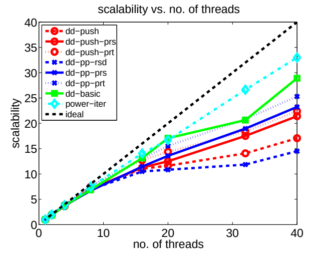
(b) pld scalability



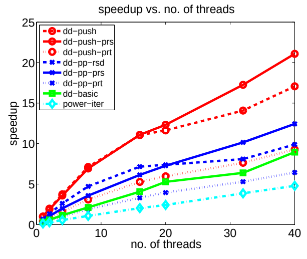
(c) pld speedup



(d) twitter run time



(e) twitter scalability



(f) twitter speedup

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

	pld		sd1		twitter		friendster	
threads	1	40	1	40	1	40	1	40
dd-push	134	133	282	273	393	417	476	581
dd-push-prs	330	319	758	740	888	850	1076	1069
dd-push-prt	246	244	538	535	395	418	504	619
dd-pp-rsd	131	130	279	271	386	410	473	540
dd-pp-prs	311	303	712	716	963	835	1239	1212
dd-pp-prt	138	136	289	286	394	419	489	611
dd-basic	655	536	1029	896	1629	1526	1482	1356
power-iter	2606	2606	6716	6716	4297	4297	3104	3104

Table 4. The number of completed tasks (unit: 10^6)

when the node is updated. 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 (instructions). Serially, a deterministic scheduler processes the same nodes, thus it does not save on total number of tasks, as can be seen in Table 4 rows for dd-push and dd-pp-rsd. The variation in those rows is due to the variation in scheduling order, especially at higher thread counts, though the variation is relatively minor.

In Table 4, all reported numbers include all tasks (nodes) considered to make scheduling decisions. For *-prt methods, this includes the nodes which are dupli-

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

Table 5. Runtime of different PageRank implementations on pld dataset

cates in the worklist. For *-prs methods, this includes each round's examination of all the nodes in the worklist to pick the priority threshold. Priority scheduling favoring priority order, *-prt, shows the high cost of duplicate items in the worklist. This priority scheduler must insert duplicate tasks every time a node moves to a new priority bin. This means that many tasks are useless, they discover as their first action that there is nothing to do and complete. Figure 1 shows that this has a distinct time cost. Although filtering out duplicates is not expensive, the total work doing so is significant. Priority scheduling favoring set semantics, *-prs, also must examine a significant number of nodes to determine which tasks to pick at each scheduling round. We observe that the total number of nodes in the worklist decreases rapidly, making the working set after several rounds significantly smaller than the entire graph. This boost in locality helps offset the extra data accesses.

It is interesting to see 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-*) is entirely read-write access, while the pull-only version (dd-basic) does one write per node processed. In general, read-mostly access patterns are significantly more cache and coherence friendly. From this perspective, the pull-push 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 weigh nodes, allowing nodes to only be processed when profitable. This improved scheduling makes up for the increased write load. Given the scheduling benefits of the residual push, it is easy to see that the push-only version is superior to the pull-push version as it reduces the memory load and work per iteration. We do note that when looking at the self-relative scalability of the implementations, the read-mostly algorithms, while slower, have better scalability than the push and pull-push variants.

Third party comparison. Table 5 shows a comparison between our data-driven PageRank algorithms (implemented using Galois) and GraphLab's PageRank implementations when varying the scheduling on pld dataset. GraphLab supports different schedulers, though we find the simple synchronous one the best. We note that the GraphLab's asynchronous method refers to a Gauss–Seidel style solver, which still is a bulk-synchronous, topology-driven approach. The power-iter version (in Galois) is actually a classic synchronous implementation in this sense, but still notably faster. While GraphLab's topology-driven synchronous implementation has similar single threaded performance to the Galois topology-driven synchronous implementation, power-iter scales much better than GraphLab. Also, all the data-driven implementations (dd-*) are much faster than GraphLab's PageRank implementations. We see that, using 40 threads, the fastest GraphLab's method takes 478 seconds whereas our push-based PageRank takes 17 seconds.

7 Discussion

Priority scheduling needs some algorithmic margin to be competitive as it is more costly. While it is not 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 hard to support efficiently and scalably. Even many high-performance, serial priority queues do not 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 its 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.

8 Conclusions

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. 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 28x over standard power iterations, these considerations can apply to many other convergence-based graph analytic algorithms.

Acknowledgments. This research was supported by NSF grants CCF-1117055 and CCF-1320746 to ID, and by NSF grants CNS-1111766 and XPS-1337281 to KP.

References

1. Andersen, R., Chung, F., Lang, K.: Local graph partitioning using PageRank vectors. FOCS pp. 475–486 (2006)
2. Bengio, Y., Delalleau, O., Le Roux, N.: Label Propagation and Quadratic Criterion. MIT Press (2006)
3. Berkhin, P.: A survey on PageRank computing. Internet Mathematics 2, 73–120 (2005)
4. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Computer Networks and ISDN Systems 30(1-7), 107–117 (1998)
5. Gleich, D.F., Zhukov, L., Berkhin, P.: Fast parallel PageRank: A linear system approach. Tech. Rep. YRL-2004-038, Yahoo! Research Labs (2004)
6. Jeh, G., Widom, J.: Scaling personalized web search. WWW pp. 271–279 (2003)
7. Lenharth, A., Nguyen, D., Pingali, K.: Concurrent priority queues are not good priority schedulers. In: Euro-Par 2015 Parallel Processing (2015)
8. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning and data mining in the cloud. VLDB Endowment pp. 716–727 (2012)
9. McSherry, F.: A uniform approach to accelerated PageRank computation. WWW pp. 575–582 (2005)
10. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. SOSP pp. 456–471 (2013)
11. Nguyen, D., Pingali, K.: Synthesizing concurrent schedulers for irregular algorithms. ASPLOS pp. 333–344 (2011)
12. Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.H., Lenharth, A., Manevich, R., Mndez-Lojo, M., Prountzos, D., Sui, X.: The Tao of parallelism in algorithms. PLDI pp. 12–25 (2011)
13. Shun, J., Blelloch, G.E.: Ligra: A lightweight graph processing framework for shared memory. PPoPP pp. 135–146 (2013)
14. Whang, J.J., Gleich, D., Dhillon, I.S.: Overlapping community detection using seed set expansion. CIKM pp. 2099–2108 (2013)
15. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Priter: A distributed framework for prioritizing iterative computations. IEEE Transactions on Parallel and Distributed Systems 24(9), 1884–1893 (2013)
16. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. IEEE Transactions on Parallel and Distributed Systems 25(8), 2091–2100 (2014)