Supplementary File of the TPDS manuscript

by Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang
yanfengzhang@ecs.umass.edu

Abstract—This supplementary file contains the supporting materials of the TPDS manuscript – "PrIter: A Distributed Framework for Prioritizing Iterative Computations." It improves the completeness of the TPDS manuscript.

1 Additional Motivating Examples

In this section, we describe a number of iterative algorithms that can benefit from the prioritized execution.

1.1 Adsorption

Adsorption [1] is a graph-based label propagation algorithm, which provides personalized recommendation for contents (e.g., video, music, document, product). The concept of *label* indicates a common feature of the entities. Adsorption's label propagation proceeds to label all of the nodes based on the graph structure, ultimately producing a probability distribution over labels for each node.

Given a weighted graph $G = (V, E, W)$, each node v carries a probability distribution L_v on label L, and it is initially assigned with an *initial distribution* I_v. The algorithm proceeds as follows. For each node v, it iteratively computes the weighted average of the label distributions from its neighboring nodes, and then a new label distribution is assigned to v as follows:

$$L_v^{(k)} = p_v^{cont} \cdot \frac{\sum_u \{W(u,v) \cdot L_u^{(k-1)}\}}{\sum_w W(w,v)} + p_v^{inj} \cdot I_v, \quad (1)$$

where p_v^{cont} and p_v^{inj} are constants associated with each node v, and $p_v^{cont} + p_v^{inj} = 1$.

Similar to PageRank, we can derive the incremental update function for the Adsorption algorithm as follows.

$$\begin{aligned} \Delta L_v^{(k)} &= p_v^{cont} \cdot \frac{\sum_u \{W(u,v) \cdot \Delta L_u^{(k-1)}\}}{\sum_w W(w,v)}, \\ L_v^{(k)} &= L_v^{(k-1)} + \Delta L_v^{(k)}, \end{aligned} \quad (2)$$

where $\Delta L_v^{(0)} = p_v^{inj} \cdot I_v$.

In addition, we can perform the selective updates for the Adsorption algorithm, and the selective process converges to the same result as the above incremental updates do. (Due to space limitation, we will not present the proof of it. It is sufficient to state that a proof similar to that of PageRank can be shown.) In order to derive the priority for Adsorption, we note that there is a key difference between Adsorption and PageRank. Adsorption normalizes receiver node's incoming link weights, while PageRank normalizes sender node's outgoing link weights. Rather than simply selecting bigger ΔL_v, we should take the incoming link weight $W(u, v)$ into consideration. That is, the priority is given to the node u with a larger value of $\frac{\sum_v W(u,v) \cdot \Delta L_u}{\sum_w W(w,v)}$. Formally, we can describe the prioritized Adsorption algorithm using the MapReduce programming model as follows.

Map: Compute $p_v^{cont} \cdot W(u, v) \cdot \Delta L_u$ for node u, send the result to its neighboring node v, and reset ΔL_u to 0.

Reduce: Compute ΔL_v by summing node v's current ΔL_v and all the normalized results (normalized by $\sum_w W(w, v)$) received by v, and update $L_v = L_v + \Delta L_v$.

Priority: Node u is eligible for the next map operation only if $\Delta L_u > 0$. Priority is given to the node with a larger value of $\frac{\sum_v W(u,v) \cdot \Delta L_u}{\sum_w W(w,v)}$.

1.2 Connected Components

Connected Components [2] is an algorithm for finding the connected components in large graphs. The main idea is as follows. For each node v in an indirected graph, it is associated with a component id c_v, which is initially set to be its own node id, $c_v^{(0)} = v$. In each iteration, each node propagates its current c_v to its neighbors. Then c_v, the component id of v, is set to be the maximum value among its current component id and the received component ids. Finally, no node in the graph updates its component id where the algorithm converges. The nodes belonging to the same connected component have the same component id.

In the prioritized example of Connected Components, we let the nodes with larger component ids propagate their component ids rather than letting all the nodes do the propagation together. In this way, the unnecessary propagation of the small component ids is avoided since those small component ids will probably be updated with larger ones in the future. The prioritized Connected Components algorithm can be described using the MapReduce programming model as follows.

Map: For node v, send its component id c_v to its neighboring node w.

Reduce: Select the maximum value among node w's current c_w and all the received results by w, and update c_w with the maximum value.

Priority: Node w is eligible for the next map operation only if c_w has changed since last map operation on w. Priority is given to the node w with larger value of c_w.

1.3 Other Algorithms

Prioritized iteration can be applied to many iterative algorithms in the fields of machine learning [3], recommendation systems [1] and link prediction [4]. The link prediction problem aims to discover the hidden links or predict the future links in complex networks such as online social networks or computer networks. The key challenge in link prediction is to estimate the proximity measures between node pairs. These proximity measures include (1) *Katz* metric [5], which exploits the intuition that the more paths between two nodes and shorter these paths are, the closer the two nodes are; (2) *rooted PageRank* [6], which captures the probability for two nodes to run into each other by performing a random walk; (3) *Expected Hitting Time* [4], which returns how long a source node takes (how many hops) to reach a target on average. Similar to PageRank and Adsorption, there is a common subproblem to compute $\sum_{k=1}^{\infty} d^k W^k$, where W is a sparse nonnegative matrix. A broad class of algorithms [4], [6] that have this closed form can be converted to a selective incremental version, where the prioritized execution will accelerate the iterative computation.

2 Proofs

In this section, we prove the correctness of Selective Incremental PageRank and the correctness of Prioritized Incremental PageRank.

2.1 The Correctness of Selective Incremental PageRank

In this subsection, we prove the main theorem that Selective Incremental PageRank converges to the same vector as normal Incremental PageRank. First, Incremental PageRank converges to a vector

$$\sum_{l=0}^{\infty}(1-d)d^l W^l E = (1-d)E/(I-dW), \qquad (3)$$

since W is a column normalized vector. Next, we show one lemma that Selective Incremental PageRank vector at subpass k should be smaller than Incremental PageRank vector at iteration k. Correspondingly, we show another lemma that there is always an subpass at which Selective Incremental PageRank vector is larger than the vector derived from Incremental PageRank at iteration k. Once we prove the two lemmas, it is sufficient to establish the main theorem that Selective Incremental PageRank converges to the same vector as Incremental PageRank does.

Before we formally state and prove the main theorem, we first establish the intuition behind the theorem. Imagine that the PageRank score for a node represents the energy of the node, we can liken the convergence process of the ranking vector to energy propagation. Take Incremental PageRank for example. Initially, each node is assigned with the initial energy $\frac{1-d}{|V|}$. At each node, the energy spreads to outgoing neighboring nodes equally with a damping factor d and retains its energy $\frac{1-d}{|V|}$. In the next iteration, each node retains its energy with the received energy from the previous iteration, and spreads the same energy to the outgoing neighboring nodes equally with a damping factor d. In other words, the energy originated from each node is damped by d^k after k iterations, in the meantime the spread energy is retained by its k hops away neighbors (The total energy retained by the touched neighbors of a node within k hops is $\sum_{l=0}^{k}\frac{d^l(1-d)}{|V|}$). This process goes on till there is no or little energy to spread at each node, and the total retained energy originated from each node is $\frac{1}{|V|}$ (total energy in graph is 1). At this point, the energy retained at a node is the ranking score of that node.

In the case of Selective Incremental PageRank, not all nodes participate in the energy spread in each subpass. For the node that does not participate in the energy spread, the node accumulates its received energy till next time the node is activated to spread its energy, and at that time the accumulated energy is also retained by the node. If any node with the temporarily accumulated energy is eventually activated, the spread energy is never lost, and the energy originated from each node will be eventually spread along any path. Therefore, Selective Incremental PageRank will eventually get the same ranking score as Incremental PageRank does. Now, we proceed to formally establish the theorem.

In order to formally describe Selective Incremental PageRank, we use activation sequence $\{S^1, S^2, \ldots, S^k, \ldots\}$ to represent the series of the node sets that Selective Incremental PageRank activates. That is, S^k is a subset of nodes to be activated in the kth subpass. Clearly, Incremental PageRank is a special Selective Incremental PageRank, in which the activation sequence is $\{V, V, \ldots\}$. Note that since we are interested in the convergence property of Selective Incremental PageRank, we will mainly focus on the activation sequences that activate each node an infinite number of times. That is, for any node j, there are an infinite number of k such that $j \in S^k$.

Lemma 1: In Incremental PageRank, the ranking score of any node j after k iterations is:

$$R_{inc}^{(k)}(j) = \Delta R_{inc}^{(0)}(j) + \sum_{l=1}^{k} \left\{ d^l \sum_{\{i_0,\ldots,i_{l-1},j\} \in P(j,l)} \frac{\Delta R_{inc}^{(0)}(i_0)}{\prod_{h=0}^{l-1} deg(i_h)} \right\}, \quad (4)$$

where $P(j,k)$ is a set of k-hop paths to reach node j.

Proof: In Incremental PageRank, each node is assigned with an initial value $\Delta R_{inc}^{(0)} = \frac{1-d}{|V|}$. From Equation (3) in Section 2.2 of the TPDS manuscript, we have

$$R_{inc}^{(k)} = \Delta R_{inc}^{(0)} + dW\Delta R_{inc}^{(0)} + \ldots + d^k W^k \Delta R_{inc}^{(0)}. \quad (5)$$

Therefore, for each node j, we have the claimed equation. □

Lemma 2: In Selective Incremental PageRank, following an activation sequence $\{S^1, S^2, \ldots, S^k\}$, the ranking score of any node j after k subpasses is:

$$R_{sel}^{(k)}(j) = \Delta R_{sel}^{(0)}(j) + \sum_{l=1}^{k} \left\{ d^l \sum_{\{i_0,\ldots,i_{l-1},j\} \in P'(j,l)} \frac{\Delta R_{sel}^{(0)}(i_0)}{\prod_{h=0}^{l-1} deg(i_h)} \right\}, \quad (6)$$

where $P'(j,l)$ is a set of l-hop paths that satisfy the following conditions. First, $i_0 \in S^l$. Second, if $l > 0$, $i_1, \ldots, i_{l-1}$ respectively belongs to the sequence of the activation sets. That is, there is $0 < m_1 < m_2 < \ldots < m_{l-1} < k$ such that $i_h \in S^{m_{l-h}}$.

Proof: We can derive $R_{sel}^{(k)}(j)$ from Equation (3) in Section 2.2 of the TPDS manuscript. □

Lemma 3: For any activation sequence, $R_{sel}^{(k)}(j) \leq R_{inc}^{(k)}(j)$ for any node j at any subpass/iteration k.

Proof: Based on Lemma 1, we can see that, after k iterations, each node receives the scores from its direct/indirect neighbors as far as k hops away, and it receives the scores originated from each direct/indirect neighbor once for each path. In other words, each node propagates its own initial value $\Delta R_{inc}^{(0)}$ (first to itself) and receives the scores from its direct/indirect neighbors through a path once.

Based on Lemma 2, we can see that, after k subpasses, each node receives scores from its direct/indirect neighbors as far as k hops away, and it receives scores originated from each direct/indirect neighbor through a path at most once. At each subpass, a score is received from a neighbor only if the neighbor is activated. If the neighbor is not activated, its score is stored at the neighbor, and the node will not receive the score until the neighbor is activated.

As a result, $R_{sel}^{(k)}(j)$ receives scores through a subset of the paths from j's direct/indirect incoming neighbors within k hops. In contrast, $R_{inc}^{(k)}(j)$ receives scores through all paths from j's direct/indirect incoming neighbors within k hops. Therefore, $R_{sel}^{(k)}(j) \leq R_{inc}^{(k)}(j)$. □

Lemma 4: For any activation sequence that activates each node an infinite number of times, given any iteration number k, we can find a subpass number k' such that $R_{sel}^{(k')}(j) \geq R_{inc}^{(k)}(j)$ for any node j.

Proof: From the proof of Lemma 3, we know that $R_{inc}^{(k)}(j)$ receives scores from all paths from direct/indirect neighbors of j within k hops away to j. In order to let $R_{sel}^{(k')}(j)$ receives all those scores, we have to make sure that all paths from direct/indirect neighbors of j within k hops away to j are activated by the activation sequence. Since the activation sequence activates each node an infinite number of times, we can always find k' such that $\{S^1, S^2, \ldots, S^{k'}\}$ contains all paths from direct and indirect neighbors of j within k hops away to j. Further, k' satisfies that $R_{sel}^{(k')}(j) \geq R_{inc}^{(k)}(j)$. □

Based on Lemma 4 and Lemma 3, we have the following theorem.

Theorem 1: As long as each node is activated an infinite number of times in the activation sequence, $R_{sel}^{(\infty)} = R_{inc}^{(\infty)}$.

2.2 The Correctness of Prioritized Incremental PageRank

In this subsection, we prove that Prioritized Incremental PageRank will activate each node an infinite number of times. This in turns shows that it will converge to the same ranking score as PageRank does as mentioned in Section 2.2 of the TPDS manuscript.

Lemma 5: Prioritized Incremental PageRank activates each node an infinite number of times.

Proof: We prove the lemma by contradiction. Assume there is a set of nodes, S, that is activated only before subpass k. Then $||\Delta R_{sel}(S)||_1$ will not decrease after k subpasses, while $||\Delta R_{sel}(V-S)||_1$ decreases. Furthermore, $||\Delta R_{sel}(V-S)||_1$ should decrease "steadily". After each of nodes in $V-S$ is activated once, $||\Delta R_{sel}(V-S)||_1$ should be decreased by a factor of at least d. Therefore, eventually at some point,

$$\frac{||\Delta R_{sel}(S)||_1}{|S|} > ||\Delta R_{sel}(V-S)||_1. \quad (7)$$

That is,

$$\max_{j \in S}(\Delta R_{sel}(j)) > \max_{j \in V-S}(\Delta R_{sel}(j)). \quad (8)$$

Since the node that has the largest ΔR_{sel} should be activated in a prioritized activation sequence, a node in S should be activated at this point, which contradicts with the assumption that any node in S is not activated after iteration k. □

Based on Lemma 5 and Theorem 1, we have the following theorem.

Theorem 2: Prioritized Incremental PageRank converges to $R_{inc}^{(\infty)}$.

3 Additional Design of PrIter

In this section, we describe the mechanisms of termination check, load balancing and fault tolerance in PrIter.

3.1 Termination Check

Iterative algorithms typically stop when a termination condition is met. To stop an iterative computation, PrIter provides three alternate methods to do termination check. 1) *Distance-based* termination check. After each iteration, each worker sends the sum of the cState values to the master (the sum operation is performed accompanied with the priority queue extraction). The master accumulates these values from different workers, and it will stop the iteration if the difference between the summed values of two consecutive iterations is less than a threshold. 2) *Subpass-based* termination check. Users set a maximum subpass number. The master keeps tracking the number of subpasses in the workers, and terminates the iterative computation after it has performed a fixed number of subpasses. 3) *Time-based* termination check. Users can also set a reasonable time limit. The master records the time passed, and terminate the iterative computation while timing out.

3.2 Load Balancing and Fault Tolerance

PrIter MRPairs process different graph partitions separately. The workload dispatched to each MRPair depends on the assigned graph partition. Even though the graph is evenly partitioned, the skewed degree distribution may result in the skewed workload distribution. Further, even though the workload is evenly distributed, we still need load balancing since a large cluster might consist of heterogeneous servers [7].

PrIter supports load balancing by *MRPair migration*. The MRPairs are configured to dump their StateTables to DFS every few subpasses, which are considered as the checkpoints for task recovery. The MRPair sends a subpass completion report to the master after completing a subpass. The subpass completion report contains the MRPair id, the subpass number, and the processing time for that subpass. Upon receipt of all the MRPairs' reports, the master calculates the average processing time for that iteration excluding longest and shortest, based on which the master identifies the slower and the faster workers. If the time difference to the average is larger than a threshold, a MRPair in the slowest worker is migrated to the fastest worker in the following three steps. The master 1) kills a MRPair in the slow worker, 2) launches a new MRPair in the fast worker, 3) and sends a rollback command to the other MRPairs. The MRPairs that receive rollback command reload their most recent dumped StateTables from DFS and proceed to extract the priority queue to recover the iterative computation. The new launched MRPair needs to load not only the StateTable but also the corresponding graph partition from DFS.

However, when the data partitions are skewed and every worker in the cluster is exactly the same, the large partition will keep moving around inside the cluster, which may degrade performance a lot and does not help load balancing. A confined load balancing mechanism can automatically identify the large partition and breaks it into multiple small subpartitions assigned to multiple idle workers.

PrIter is also resilient to task failures and worker failures. The master notices some MRPair failures or worker failures when it has not received any response from probing for a certain period of time. The failed MRPair(s) in the failed workers are re-launched in other healthy workers from the most recent checkpoint. Meanwhile, all the other MRPairs roll back to the same checkpoint to redo the failed iterative computation.

4 Optimal Queue Size

The iterative algorithms with prioritized execution converge faster than that without priority. The size of the priority queue is critical in determining how many computations are needed for algorithm convergence. Intuitively, the shorter the queue is, the less computations are needed to be performed. However, the shorter priority queue results more overhead due to more frequent subpasses and as result more frequent queue extractions. In this section, we show how to derive the optimal queue size.

The iterative computation's running time is composed of two parts: the processing time corresponding to the number of computations and the overhead time. We derive these two parts as follows. First, let q be the queue size and $f(q)$ be the total workload needed for convergence when the queue size is set to be q. The workload is reflected by the total number of nodes activations (*i.e.,* map operations) during the iterative process. Let T_{proc} be the average processing time of each node activation, including the time for computation and the time for communication. Thus, the total time spent on processing nodes for all subpasses is $f(q) \cdot T_{proc}$. Second, the overhead occurs in each subpass, and the overhead time is dominated by the time incurred for extracting nodes from the StateTable to the priority queue, which is linear in the StateTable size, N (see Section 3.3 of the TPDS manuscript. Let T_{ovhd} be the average time for scanning a record in the StateTable. Thus, the total overhead time for all subpasses is $\frac{f(q)}{q} \cdot N \cdot T_{ovhd}$, where $\frac{f(q)}{q}$ is the number of subpasses.

Therefore, we have the total running time shown as follows:

$$\min_{q} \left\{ f(q) \cdot T_{proc} + \frac{f(q)}{q} \cdot N \cdot T_{ovhd} \right\}, \tag{9}$$

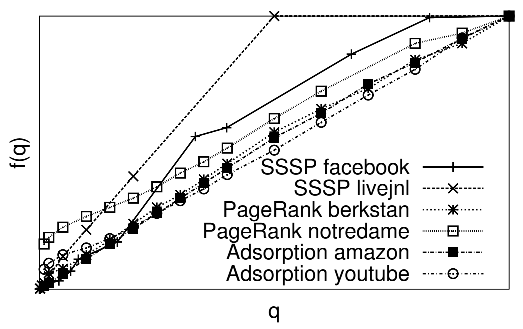


Fig. 1. Function f for real graphs.

where T_{proc} and T_{ovhd} are related to the cluster environments. We want to minimize it by choosing the optimal q, and function $f(q)$ is the key for finding the optimal q.

We estimate $f(q)$ for different algorithms with a series of real data sets. The real data sets are described in Section 6.1. Figure 1 shows $f(q)$, where q and $f(q)$ are normalized for comparison purpose. We can see that $f(q)$ can be estimated with a linear function of q. That is, $f(q) = \alpha \cdot q + \beta$.

Given the linear function of q, we consider the optimization problem shown in Equation (9). Since only q is a variable, the problem becomes the following minimization problem:

$$\min_q \left\{ \alpha \cdot T_{proc} \cdot q + \frac{\beta \cdot N \cdot T_{ovhd}}{q} \right\}. \tag{10}$$

Then we have the optimal q^*:

$$q^* = \sqrt{\frac{\beta \cdot N \cdot T_{ovhd}}{\alpha \cdot T_{proc}}}. \tag{11}$$

To sum up, the optimal queue size depends on a series of factors, *i.e.*, α, β, N, T_{proc} and T_{ovhd}. We only know the number of local nodes, N. In PrIter, we set $\frac{q}{N}$ to be 0.2 by default. As will be shown in Section 6.4, the default setting gives competitive performance. Additionally, we will compare the different settings of $\frac{q}{N}$ to see the performance difference in Section 6.4.

Alternatively, we can decide the optimal queue size dynamically by estimating all the factors. T_{proc} and T_{ovhd} can be estimated by some online measurement method. At the same time, α and β can be estimated by online analysis. For PageRank example, we can use the similar sampling method in Section 3.3 of the TPDS manuscript to approximate the current ΔR's distribution, which reflects $f(q)$. That means, α and β can be approximated by restoring $f(q)$. Nevertheless, realizing dynamic queue size is challenging. This is because estimating these factors accurately in real time is intractable. Moreover, the online measurement will bring implementation complexities and will impact system performance.

5 API

PrIter has a few application programming interfaces exposed to users for implementing an iterative algorithm in PrIter. We first introduce these API and then show how to implement PageRank algorithm in PrIter.

5.1 API Description

According to the design and implementation of the memory-based PrIter described in Section 3 of the TPDS manuscript, we design the API of memory-based PrIter as follows:

- `initStateTable`: specify nodes' initial state;
- `map`: process the iState of a node and map the results to its neighboring nodes;
- `updateState`: update a node's iState and cState;
- `decidePriority`: decide the priority value for a node based on its iState, for prioritized iteration;
- `decideTopK`: decide the priority value for a node based on its cState, for top-k results extraction;
- `resetiState`: reset a node's iState after it has been put into the priority queue;
- `partitionGraph`: (optional) specify the assignment of a node to a worker in order to partition an input graph;
- `readGraph`: (optional) load a certain graph partition to a MRPair.

The implementations of `partitionGraph` and `readGraph` are optional. PrIter supports automatically graph partitioning and graph loading for a few particular formatted graphs (including weighted and unweighted graphs). Users can first format their graphs in our supported formats to avoid implementing these two interfaces. On the other hand, users also have the flexibility to decide their own smart partitioning schemes by implementing `partitionGraph` themselves, such that the workload can be distributed more evenly. The graph partition is loaded to the local memory by default. In-memory StateTable has a field to store the linkage information, which is separated from the main state information when checkpointing is performed. Users can also customize `readGraph` implementation to load the graph partition in other abstract objects. For example, *RDD* [8], [9] that supports failure recovery can be utilized. In addition to these API, there are a series of parameters users should specify, such as the number of graph partitions, the snapshot generation interval, and the termination condition.

The file-based PrIter has minor changes on the API described above. As described in Section 4 of the TPDS manuscript, in the file-based PrIter, iState is updated in the first phase, and cState is updated in the second phase. (Note that the second phase only resets iState). Hence, instead of implementing

```
initStateTable
Input: node subset V, damping factor d
 1: foreach node n in V do
 2:     StateTable(n).iState = (1-d) / |V|;
 3:     StateTable(n).cState = (1-d) / |V|;
 4:     StateTable(n).priority = (1-d) / |V|;
 5: end for

map
Input: node n, ΔR
 6: <links> = look up n's outlinks;
 7: foreach link in <links> do
 8:     output(link.endnode, (d × ΔR) / |<links>| );
 9: end for

updateState
Input: node n, ΔR
10: StateTable(n).iState = StateTable(n).iState + ΔR;
11: StateTable(n).cState = StateTable(n).cState + ΔR;

resetiState
Input: node n
12: StateTable(n).iState = 0;

decidePriority
Input: node n, iState
13: return iState;

decideTopK
Input: node n, cState
14: return cState;

main
15:  Job job = new Job();
16:  job.set("priter.input.path", /user/yzhang/googlegraph);
17:  job.setInt("priter.graph.partitions", 100);
18:  job.setLong("priter.snapshot.interval", 20000);
19:  job.setFloat("priter.stop.dis.threshold", 0.1);
20:  job.submit();
```

Fig. 2. PageRank implementation pseudo-code in memory-based PrIter.

the `updateState` interface, users are required to implement the following two interfaces:

- `updateiState`: update a node's iState in the first phase;
- `updatecState`: update a node's cState in the second phase.

In addition, instead of using the `initStateTable` interface to initialize the StateTable, users are required to initialize the iState file, the cState file, the static data file, and the initial priority queue file through the following interface:

- `initFiles`: initialize various sorted files.

5.2 PageRank Implementation Example

For better understanding, we walk through how prioritized PageRank is implemented in memory-based PrIter. Figure 2 shows the pseudo-code of this implementation. Basically, prioritized PageRank implementation follows the algorithm logic that we illustrated in Section 2.2 of the TPDS manuscript. The node entries in the StateTable are initialized with identical state values $\frac{1-d}{|V|}$ and priority values $\frac{1-d}{|V|}$ (line 1-5). In the map function, each node distributes its equally divided iState values to its neighbors (line 6-9). The *output* in line 8 abides by Hadoop programming style for outputting the intermediate key-value pair. In the updateState function, each node accumulates the partial results from its predecessor nodes and updates the StateTable (line 10-11). We should also set the default iState for reset (line 12). The priority determination rules for prioritized iteration (line 13) and for top-k results extraction (line 14) should be specified respectively. Since the input graph is formatted, we only need to specify the path of the input data set on DFS (line 15). We split the input graph into 100 partitions (line 16), and we let the system to generate a result snapshot every 20 seconds (line 17). Using the distance-based termination check method, the iterative computation will be terminated when the L1-Norm distance between two consecutive iteration results is less than 0.1 (line 18).

6 Data Sets and Additional Experiment Results

In this section, we first describe the data sets used in the experiment in Section 5 of the TPDS manuscript, and then show more results in complementary of the TPDS manuscript.

6.1 Data Sets

Real Data Sets: Four iterative algorithms described in Section 2 of the TPDS manuscript and Section 1 in this supplementary file are implemented: SSSP, PageRank, Adsorption, and Connected Components (ConnComp). The data sets used for these algorithms are described in Table 1. Most of them are downloaded from [10]. The graphs for the SSSP problem are directed and weighted. Since the LiveJournal graph and the roadCA graph are not originally weighted, we synthetically assign a weight value to each edge, where the weight is generated based on a log-normal distribution. The log-normal distribution parameters ($\mu = 0.4$, $\sigma = 1.2$) are extracted from the Facebook user interaction graph [11], where the weight reflects user interaction frequency. The web graphs for the PageRank computation are directed and unweighted. For the three graphs used in the Adsorption algorithm, the weight of a node's inbound links are normalized. The graphs for Connected Components are made undirected simply through adding an inverse direction for each directed link.

Synthetic Data Sets: We prefer real graphs for performance evaluation since they are better for illustrating the effect of prioritized iteration in real life applications, even though they are relatively small. In addition, in order to perform experiments with different

TABLE 1
Data Sets Summary

Algorithm	Graph	Nodes	Edges
SSSP	Facebook	1,204,004	20,492,148
	LiveJournal	4,847,571	68,993,773
	roadCA	1,965,206	5,533,214
PageRank	Berk-Stan	685,231	7,600,595
	Google	875,713	5,105,039
	Notredame	325,729	1,497,134
Adsorption	Facebook	1,204,004	20,492,148
	Youtube	311,805	1,761,812
	Amazon	403,394	3,387,388
ConnComp	Amazon	403,394	3,387,388
	Wiki-talk	2,394,385	5,021,410
	LiveJournal	4,847,571	68,993,773

input sizes. In the generated web graph, the in-degree of each page follows a log-normal distribution, where the log-normal parameters ($\mu = -0.5$, $\sigma = 2.3$) are extracted from the three real web graphs.

6.2 Top Record Emergence Time

As discussed in Section 3.4 of the TPDS manuscript, peoples are always interested in a small collection of more attractive records. For example, users always care about the closest nodes in the shortest path problem, and they are only interested in the first few pages of Google search results. The time it takes to derive the top records is critical for these applications. The speedup of the convergence correspondingly reduces the top record emergence time, so that users are able to obtain the top records online even before the algorithm completely converges.

To illustrate this benefit experimentally, we analyze the timely-generated snapshot results to record the time when the top records are emerged. The top records are the webpages with the highest ranking scores or the nodes with the shortest distance values, except that in the case of Connected Components the top records are postulated as the first nodes that have finalized their component ids. In the case of Hadoop MapReduce, a series of MapReduce jobs are used to perform the iterations, and the intermediate result is accessible only after an iteration job is completed. Thus, the emerging time of top-k records is recorded as the time elapsed when the pre-computed correct top-k records emerge after a certain iteration. In the case of PrIter, the top-k result snapshot is generated periodically. We compare these top-k result snapshots with the pre-computed correct top-k result to determine how many tops are emerged.

We perform the experiment on our local cluster. Figure 3 shows the top record emergence time of different algorithms. PrIter with prioritized execution speeds up the top record emergence time by a factor of 2 to 8 comparing with the priority-off PrIter. Moreover, PrIter achieves up to 50x speedup comparing with Hadoop.

6.3 Effectiveness of Sampling Queue Extraction

As presented in Section 3.3 of the TPDS manuscript, PrIter exploits a sampling-based priority queue extraction technique to approximately extract the top priority nodes. The sampling-based approach is significantly important when the data set is large, which sacrifices a little accuracy for saving computation time. Intuitively, the more samples we picked the more accuracy of this approximation is obtained but the more time is consumed. In this experiment, we evaluate the effectiveness of the sampling approach.

To measure the accuracy, we first define the error rate as $error = \frac{|q'-q|}{q}$, where q is the required queue size (*i.e.* the number of top priority nodes required to be extracted) and q' is the approximated number of nodes to be extracted by applying Algorithm 1 in the TPDS manuscript. We evaluate the sampling extraction technique in the context of PageRank computation with four data sets, including real data sets and synthetic data set on a single machine. We measure the error rate and extraction time after the first PageRank iteration (when the nodes are with different priority values). Fig. 4 shows the (mean/min/max) results by varying the sampling size. We can see that the error rates are all under 10% and the extraction time is increasing quickly when enlarging the sampling size. The error rate can be controlled low (under 5%) within 100 milliseconds by setting sampling size between 1K and 2K. PrIter sets the sampling size as 1K by default.

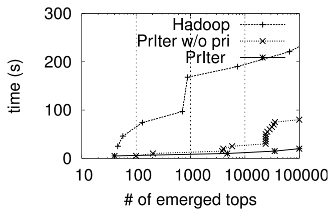
6.4 The Impact of Priority Queue Size

The iterative algorithms with prioritized execution converge faster than that without priority. The size of the priority queue is critical in determining how many computations are needed for algorithm convergence. Intuitively, the shorter the queue is, the less computations are needed to be performed. However, the shorter priority queue results more overhead due to more frequent subpasses and as result more frequent queue extractions. Therefore, there is an optimal priority queue size that results in the best performance.

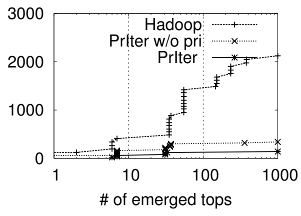
PrIter sets the queue size q in proportion to the total number of local nodes N. To see the effects of different settings of $\frac{q}{N}$, we perform a series of experiments with PrIter on our local cluster, running each algorithm on three different real graphs. Figure 5 shows the convergence speedup results varying algorithms and inputs. We can see that most of them reach their optimal points when $\frac{q}{N}$ is set to be around 0.2, which means that each subpass will process the top-priority 20% of local nodes. This provides positive support to our default selection ($\frac{q}{N} = 0.2$).

7 Related Work

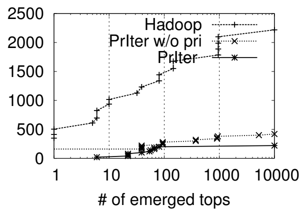
In this section, we review the related work mentioned in Section 6 of the TPDS manuscript in detail.



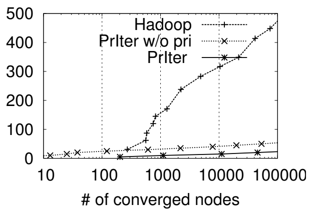
(a) SSSP (Facebook)



(b) PageRank (BerkStan)

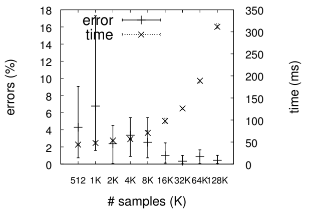


(c) Adsorption (Facebook)

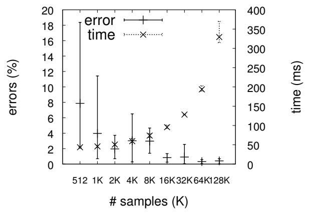


(d) ConnComp (Amazon)

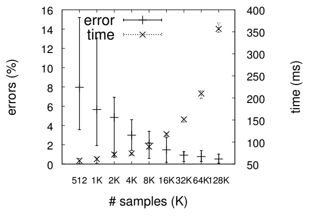
Fig. 3. Top record emergence time.



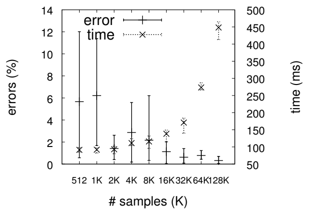
(a) BerkStan



(b) Notredame

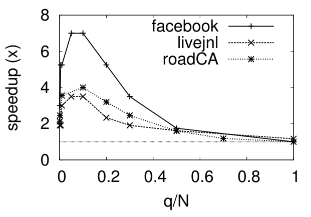


(c) Google

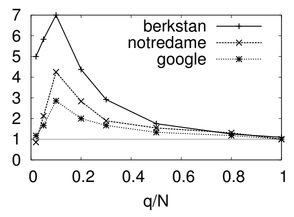


(d) Synthetic

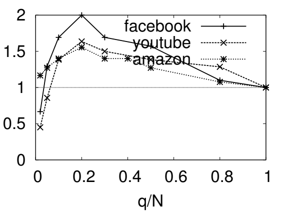
Fig. 4. Error rate and extraction time with various sampling sizes for PageRank computation.



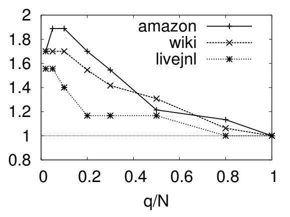
(a) SSSP



(b) PageRank



(c) Adsorption



(d) ConnComp

Fig. 5. The effect of priority queue size.

For the Hadoop implementations of iterative applications, shuffling static data during the iterative process incurs significant communication overhead. HaLoop [12] takes advantage of the task scheduler to guarantee local access to the static data. The task scheduler is designed to assign tasks to the workers where the needed data is located. iMapReduce [13] is a framework that supports iterative processing and avoids the re-shuffling of static data without modifying the task scheduler. With iMapReduce, static data are partitioned and distributed to workers in the initialization phase. By logically connecting reduce to map and the support of persistent tasks, the iterated data are always hashed to the workers where their corresponding static data are located. As described in Section 3 of the TPDS manuscript, PrIter integrates iMapReduce to avoid the unnecessary static data shuffling.

In the meanwhile, a series of frameworks that maintain the iteration state in memory have been proposed for iterative computations. Piccolo [14] allows computation running on different machines to share distributed, mutable state via a key-value table interface. This enables one to implement iterative algorithms that access in-memory distributed tables without worrying about the consistency of the data. Spark [8], [9] uses a caching technique to improve the performance for repeated operations. The main idea in Spark is the construction of *resilient distributed dataset* (RDD), which is a read-only collection of objects maintained in memory across iterations and supports fault recovery. [15] presents a generalized architecture for continuous bulk processing (CBP), which performs iterative computations in an incremental fashion by unifying stateful programming with a data-parallel operator. CIEL [16] supports data-dependent iterative or recursive algorithms by building an abstract dynamic task graph. Twister [17] employs a lightweight MapReduce runtime system with all operations performed in memory cache and uses publish/subscribe messaging system instead of a distributed file system for data communication. These in-memory systems enhance the performance of data access, but do not allow prioritized iterations.

References

[1] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly, "Video suggestion and discovery for youtube: taking random walks through the view graph," in *Proc. Int'l Conf. World Wide Web (WWW '08)*, 2008, pp. 895–904.

[2] U. Kang, C. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proc. IEEE Int'l Conf. Data Mining (ICDM '09)*, 2009, pp. 229 –238.

[3] Chu, Cheng T., Kim, Sang K., Lin, Yi A., Yu, Yuanyuan, Bradski, Gary R., Ng, Andrew Y., and Olukotun, Kunle, "Map-Reduce for Machine Learning on Multicore," in *Proc. Int'l Conf. Neural Information Processing Systems (NIPS '06)*, 2006, pp. 281–288.

[4] D. Liben-Nowell and J. Kleinberg, "The link prediction problem for social networks," in *Proc. ACM Conf. Information and Knowledge Management (CIKM '03)*, 2003, pp. 556–559.

[5] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, 1953.

[6] H. H. Song, T. W. Cho, V. Dave, Y. Zhang, and L. Qiu, "Scalable proximity estimation and link prediction in online social networks," in *Proc. Int'l Conf. Internet Measurement (IMC '09)*, 2009, pp. 322–335.

[7] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. S-toica, "Improving mapreduce performance in heterogeneous environments." in *Proc. USEINX Symp. Opearting Systems Design and Implementation (OSDI '08)*, 2008, pp. 29–42.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proc. USENIX Workshop Hot Topics in Cloud Computing (HotCloud '10)*, 2010.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for. in-memory cluster computing," in *Proc. USEINX Symp. Networked Systems Design and Implementation (NSDI'12)*, 2012.

[10] Stanford dataset. http://snap.stanford.edu/data/.

[11] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *Proc. EuroSys '09*, 2009, pp. 205–218.

[12] Y. Bu, B. Howe, M. Balazinska, and D. M. Ernst, "Haloop: Efficient iterative data processing on large clusters," in *Proc. Int'l Conf. Very Large Database (VLDB '10)*, 2010.

[13] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," in *Proc. IEEE Int'l Workshop Data Intensive Cloud Computing (DataCloud '11)*, 2011.

[14] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. USENIX Symp. Opearting Systems Design and Implementation (OSDI '10)*, 2010.

[15] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," in *Proc. ACM Symp. Cloud Computing (SOCC '10)*, 2010, pp. 51–62.

[16] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: A universal execution engine for distributed data-flow computing," in *Proc. USEINX Symp. Networked Systems Design and Implementation (NSDI '11)*, 2011.

[17] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proc. IEEE Int'l Workshop MapReduce (MapReduce '10)*, 2010, pp. 810–818.