PrIter: A Distributed Framework for Prioritized Iterative Computations

Yanfeng Zhang[1,2], Qixin Gao[1], Lixin Gao[2], Cuirong Wang[1]

[1]Northeastern University, China
[2]University of Massachusetts Amherst, USA
{yanfengzhang, lgao}@ecs.umass.edu; {gaoqx, wangcr}@neuq.edu.cn

ABSTRACT

Iterative computations are pervasive among data analysis applications in the cloud, including Web search, online social network analysis, recommendation systems, and so on. These cloud applications typically involve data sets of massive scale. Fast convergence of the iterative computation on the massive data set is essential for these applications. In this paper, we explore the opportunity for accelerating iterative computations and propose a distributed computing framework, PrIter, which enables fast iterative computation by providing the support of prioritized iteration. Instead of performing computations on all data records without discrimination, PrIter prioritizes the computations that help convergence the most, so that the convergence speed of iterative process is significantly improved. We evaluate PrIter on a local cluster of machines as well as on Amazon EC2 Cloud. The results show that PrIter achieves up to 50x speedup over Hadoop for a series of iterative algorithms.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems

General Terms

Algorithms, Design, Theory, Performance

Keywords

PrIter, prioritized iteration, iterative algorithms, MapReduce, distributed framework

1. INTRODUCTION

Iterative computations are common in myriad of data mining algorithms. PageRank [6], as a well-known iterative algorithm, has been widely used in Web search engines. Other iterative algorithms such as Adsorption [5] and Expected Hitting Time [16] are applied to the problem domains such as link prediction [16] and recommendation systems [33]. In computational biology, iterative algorithms such as K-means clustering algorithm [25] have been adopted in classifying a large collection of data. The massive amount of data involved in these applications exacerbates the need for a computing cloud and a distributed framework that supports fast iterative computation. MapReduce [10], which powered cloud computing, is such a framework that supports data processing of massive scale. Dryad/DryadLINQ [13, 29], Hadoop [2], Pig [21], Hive [27], and Pregel [19] have been proposed as well. In particular, all of the previously proposed frameworks assume that the iterative update is equally important for all data.

However, in reality, selectively processing some portions of the data first has the potential of accelerating the iterative process, rather than simply performing a series of iterations over all the data. Some of the data points play an important decisive role in determining the final converged outcome. By giving an execution priority to some of the data, the iterative process can potentially converge fast. For example, the well-known shortest path algorithm, Dijkstra's algorithm, greedily expands the node with the shortest distance first. This will not only derive the shortest distance for all nodes fast but also be able to quickly return the nearest nodes. Unfortunately, neither MapReduce nor any existing distributed computing framework provides the support of prioritized execution.

In this paper, we demonstrate the potential of prioritized execution for iterative computations with a broad set of algorithms. This motivates the desire of a general priority-based distributed computing framework. We design and implement PrIter, a distributed framework, that supports the prioritized execution of iterative computations. To realize prioritized execution, PrIter allows users to explicitly specify the priority value of each processing data point. In addition, PrIter is designed to support load balancing and fault tolerance so as to accommodate diverse distributed environments.

To evaluate the performance of PrIter, we run a series of well-known algorithms including shortest path and PageRank on a local cluster as well as on Amazon EC2 Cloud [1]. Our experimental results show that PrIter significantly speeds up the convergence of the iterative computations, which achieves up to 50x speedup over the implementations with Hadoop. Furthermore, we show the effectiveness of prioritization by comparing PrIter with prioritized execution and that without prioritized execution. The results show that PrIter with prioritization achieves 2x-8x speedup over that without prioritization.

SOCC'11, October 27–28, 2011, Cascais, Portugal.

The rest of the paper is organized as follows: In Section 2, we illustrate the benefit of prioritized iteration through a series of example algorithms. Section 3 presents the system design and implementation of PrIter, and we have some further discussions about PrIter in Section 4. The experiment results are shown in Section 5, followed by a survey of related work in Section 6. Finally, we conclude the paper in Section 7.

2. MOTIVATING EXAMPLES

In this section, we describe a series of well-known iterative algorithms that benefit from the prioritized execution. We then briefly list many other algorithms that have the similar property.

2.1 Single Source Shortest Path

Single Source Shortest Path (SSSP) is a classical problem that derives the shortest distance from a source node s to all other nodes in a graph. Formally, given a weighted, directed or undirected graph $G = (V, E, W)$, where V is the set of nodes, E is the set of edges, and $W(i, j)$ is a positive weight of the edge from node i to node j. The shortest distance from the source node s to a node j can be computed iteratively as follows:

$$D^{(k)}(j) = \min\Big\{D^{(k-1)}(j),\ \min_i\big\{D^{(k-1)}(i) + W(i,j)\big\}\Big\},$$

where k is the iteration number, and i is an incoming neighbor of node j. Initially, $D^0(s) = 0$, and $D^0(j) = \infty$ for any node j other than s.

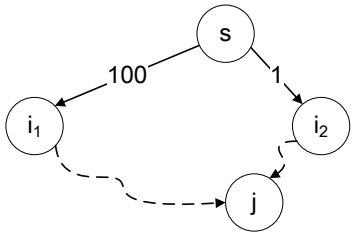


Figure 1: A single source shortest path example.

Although the iterative computation can be easily implemented in a distributed environment, it potentially has the drawback of performing more computations than necessary. For example, in Figure 1, it is more likely that the shortest path from s to j is the path via i_2 than that via i_1. However, in the iterative computation, it explores both paths simultaneously. In contrast, Dijkstra's algorithm selectively picks the node with the shortest distance to expand. In fact, the path via i_1 would not be explored if the path via i_2 is shorter than 100. Therefore, the iterative algorithm performs more computation than necessary if nodes are expanded hop by hop. In order to reduce the unnecessary computations, we propose to perform iterative computation with a priority. That is, the nodes with smaller distance values are given a priority for letting them expand or perform iterative computations first. In a distributed environment, each machine will select the nodes to expand according to the priority values. Formally, the prioritized SSSP can be described by the MapReduce programming model as follows:

Map: Compute $D(i) + W(i, j)$ for node i, send the result to its neighboring node j.

Reduce: Select the minimum value among node j's current $D(j)$ and all the results received by j, and update $D(j)$ with the minimum value.

Priority: Node j is eligible for the next map operation only if $D(j)$ has changed since the last map operation on j. Priority is given to the node j with smaller value of $D(j)$.

2.2 PageRank and Personalized PageRank

PageRank and Personalized PageRank are popular algorithms initially proposed for ranking web pages. Later on, these algorithms have found a wide range of applications, such as link prediction [16, 26]. PageRank ranks web pages by performing a random walk on the web linkage graph. Formally, the web linkage graph is a graph where the node set V is the set of web pages, and there is an edge from node i to node j if there is a hyperlink from page i to page j. Let W be the column-normalized matrix that represents the web linkage graph. That is, $W(j, i) = 1/deg(i)$ (where $deg(i)$ is the outdegree of node i) if there is a link from i to j, otherwise, $W(j, i) = 0$. Thus, the PageRank vector R with each entry indicating a page's ranking score can be computed iteratively as follows:

$$R^{(k)} = dWR^{(k-1)} + (1-d)E, \tag{1}$$

where k is the iteration number, d is the damping factor, and E is a size-$|V|$ vector with each entry being $\frac{1}{|V|}$. The PageRank vector converges to

$$R^{(\infty)} = \sum_{l=0}^{\infty}(1-d)d^lW^lE. \tag{2}$$

Note that $R^{(\infty)}$ does not depend on the initial PageRank vector $R^{(0)}$.

Personalized PageRank differs from PageRank only at vector E. In Personalized PageRank, E indicates the personal preferences, in which only the entries representing the personally preferred pages are non-zero. Below we will focus on discussing prioritized iteration for PageRank. However, similar argument follows for Personalized PageRank.

In order to illustrate the benefit from prioritized iteration, we first present an alternate iterative computation for PageRank, referred to as *Incremental PageRank* that derives the same vector as PageRank:

$$\begin{aligned}\Delta R_{inc}^{(k)} &= dW\Delta R_{inc}^{(k-1)}\\ R_{inc}^{(k)} &= R_{inc}^{(k-1)} + \Delta R_{inc}^{(k)},\end{aligned} \tag{3}$$

where $\Delta R_{inc}^{(0)} = R_{inc}^{(0)} = (1-d)E$. Note that both PageRank and Incremental PageRank converge to the same ranking vector as shown in Equation 2. Therefore, Incremental PageRank can be used for computing PageRank scores.

Furthermore, the Incremental PageRank update can be executed selectively. That is, the update function does not have to be performed by all nodes concurrently. In each iteration, only a selected subset of nodes perform the update function instead. To differentiate the "iteration" used in selective update from the "iteration" used in concurrent update, we refer to an "iteration" used in selective update as a *subpass*.

Let S^k denote the subset of nodes in V that are activated to perform the update function at subpass k. *Selective Incremental PageRank* updates the ranking score as follows.

$$\begin{aligned} \Delta R_{sel}^{(k)} &= \Delta R_{sel}^{(k-1)}(V - S^k) + dW\Delta R_{sel}^{(k-1)}(S^k) \\ R_{sel}^{(k)} &= R_{sel}^{(k-1)} + dW\Delta R_{sel}^{(k-1)}(S^k), \end{aligned} \tag{4}$$

where $\Delta R_{sel}^{(0)} = R_{sel}^{(0)} = (1-d)E$. $\Delta R_{sel}^{(k-1)}(S^k)$ is a vector with only nodes in S^k being accounted for and all the other entries being 0, while $\Delta R_{sel}^{(k-1)}(V - S^k)$ is a vector with only nodes in $V - S^k$ retaining their $\Delta R_{sel}^{(k-1)}$ and all the other entries being 0. That is, once being activated, the nodes in S^k use their ΔR_{sel} to update $\Delta R_{sel}^{(k)}$ and $R_{sel}^{(k)}$, after that they reset their ΔR_{sel} to be $\mathbf{0}$. As long as each node is activated an infinite number of times, the proof in Appendix A shows that Selective Incremental PageRank will converge to the same PageRank vector as Incremental PageRank.

The selective computation of PageRank indicates that the prioritized execution of iterative computations is feasible. Now, we show how to determine the priority and the benefit of the prioritized execution. For the ease of argument, we use L1-Norm distance between the current subpass's PageRank vector $R_{sel}^{(k)}$ and the converged PageRank vector $R_{sel}^{(\infty)}$ to quantify the closeness to convergence. As shown in Equation (4), each entry of $R_{sel}^{(k)}$ is monotonic nondecreasing as k increases. Therefore, the bigger $||R_{sel}^{(k)}||_1$ is, the closer $R_{sel}^{(k)}$ is to the converged PageRank vector. Since W is a column normalized matrix, $||R_{sel}^{(k)}||_1 = ||R_{sel}^{(k-1)}||_1 + d||\Delta R_{sel}^{(k-1)}(S^k)||_1$. Accordingly, node i in S^k with its $\Delta R_{sel}^{(k-1)}(i)$ contributes $d\Delta R_{sel}^{(k-1)}(i)$ for shortening the distance between $R_{sel}^{(k-1)}$ and $R_{sel}^{(\infty)}$. Hence, the larger $\Delta R_{sel}^{(k-1)}(i)$ contributes more for the convergence of $R_{sel}^{(k-1)}$ towards $R_{sel}^{(\infty)}$.

Let S^* denote a subset of nodes that $\min_{i \in S^*} \Delta R_{sel}(i) \geq \max_{i \in V - S^*} \Delta R_{sel}(i)$. *Prioritized Incremental PageRank* performs the iterative computation as shown in Equation (4), which is always selecting nodes in S^* to activate in each subpass but ignoring the nodes in $V - S^*$. That is, in order to accelerate the PageRank computation, the nodes in S^*, a subset of nodes with bigger ΔR_{sel}, are activated in each subpass. Furthermore, Prioritized Incremental PageRank converges to the same PageRank vector as Incremental PageRank (see the proof in Appendix B).

Formally, we describe Prioritized Incremental PageRank using the MapReduce programming model as follows.

Map: Compute $d\Delta R(i)W(i,j)$ for node i, send the result to its neighboring node j, and reset $\Delta R(i)$ to be 0.

Reduce: Compute $\Delta R(j)$ by summing node j's current $\Delta R(j)$ and all the results received by j, and update $R(j) = R(j) + \Delta R(j)$.

Priority: Node j is eligible for the next map operation only if $\Delta R(j) > 0$. Priority is given to the node with a larger value of ΔR.

2.3 Adsorption

Adsorption [5] 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:

$$L_v^{(k)} = p_v^{cont} \cdot \frac{\sum_u \{W(u,v) \cdot L_u^{(k-1)}\}}{\sum_w W(w,v)} + p_v^{inj} \cdot I_v, \tag{5}$$

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.

$$\begin{aligned} \Delta L_v^{(k)} &= p_v^{cont} \cdot \frac{\sum_u \{W(u,v) \cdot \Delta L_u^{(k-1)}\}}{\sum_w W(w,v)}, \\ L_v^{(k)} &= L_v^{(k-1)} + \Delta L_v^{(k)}, \end{aligned} \tag{6}$$

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 $\frac{\sum_v W(u,v) \cdot \Delta L_u}{\sum_w W(w,v)}$.

2.4 Connected Components

Connected Components [14] 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.

2.5 Other Algorithms

Prioritized iteration can be applied to many iterative algorithms in the fields of machine learning [9], recommendation systems [5] and link prediction [16]. 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 [15], 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* [26], which captures the probability for two nodes to run into each other by performing a random walk; (3) *Expected Hitting Time* [16], 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:

$$\sum_{k=1}^{\infty} d^k W^k, \tag{7}$$

where W is a sparse nonnegative matrix. A broad class of algorithms [16, 26] that have the closed form can be converted to a selective incremental version, where the prioritized execution will accelerate the iterative computation.

3. PRITER

In this section, we propose PrIter, a distributed framework for prioritized iterative computations, which is implemented based on Hadoop MapReduce [2]. First, we describe the requirements of a framework that supports prioritized iterative computations.

1. The framework needs to support iterative processing. Iterative algorithms perform the same computation in each iteration, and the state from the previous iteration has to be passed to the next iteration efficiently.

2. The framework needs to support state maintenance across iterations. In MapReduce, only the previous iteration's result is needed for the next iteration's computation, while in PrIter the intermediate iteration state should be maintained across iterations due to the selective update operations.

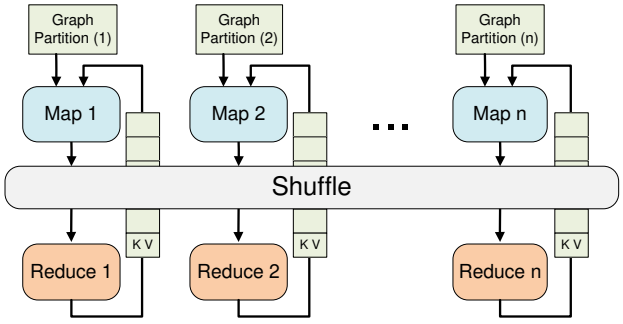


Figure 2: Iterative processing structure.

3. The framework needs to support prioritized execution. That is, an efficient selection of the high priority data should be provided.

PrIter provides the functionalities of iterative processing (Section 3.1), state maintenance (Section 3.2), prioritized execution (Section 3.3), termination check (Section 3.4), and online top-k query (Section 3.5). The framework has been designed for scalable and fault-tolerant implementation on clusters of thousands of commodity computers, so that the load balancing and fault-tolerance mechanisms are provided in PrIter (Section 3.6). Finally, we summarize PrIter's APIs and show a representative PageRank implementation example in PrIter (Section 3.7).

3.1 Iterative Processing

PrIter incorporates the support of *iMapReduce* [32] for iterative processing. Iterative process performs the same operation in each iteration, and the output of the previous iteration is passed to the next iteration as the input. iMapReduce following MapReduce paradigm directly passes the reduce output to the map for the next iteration, rather than writing output to distributed file system (DFS). Figure 2 shows the overall iterative processing structure.

We separate the data flow into two sub data flows according to their variability features, including the static data flow and the state data flow. The static data (e.g., the graph structure) keeps unchanged over iterations, which is used in the map function for exchanging information between neighboring nodes. While the state data (e.g., the iterated shortest distance or the PageRank score) is updated every iteration, which indicates the node state. The static graph data and the initial state data are partitioned and preloaded to workers, and the framework will join the static data with the state data before map operation.

Under the modified MapReduce framework, we can focus on updating the state data through map and reduce functions on the key-value pairs. Each *key* represents a node id, and the associated *value* is the node state that is updated every iteration (e.g., the PageRank score of a webpage). In addition, each node has information that is static across iterations (e.g., the node linkage information), which is also indexed by node ids (*nid*). A hash function F applying on the keys/nodes is used to split the static graph data and the initial node state data evenly into n partitions according to:

$$pid = F(nid, n), \tag{8}$$

where *pid* is a partition id. These partitions are assigned to different workers by the master. Each worker can hold one or more partitions.

A map task with map task id *pid* is assigned for processing partition *pid*, and the output ⟨key, value⟩/⟨node, state⟩ pairs of the map task are shuffled to the reduce tasks according to the same hash function, F. Accordingly, a reduce task with reduce task id *pid* is assigned by the task scheduler to connect to the map task with map task id *pid* in the same worker, by which we establish a local reduce-to-map connection. The reduce merges the results from various maps to update a node's state, and its output ⟨node, state⟩ pairs are directed to the connected map as the map's input. By using the same hash function F for partitioning and shuffling, a node's static data (e.g., neighbors in the web graph) and its dynamic state are always joined in the same map task. Therefore, a paired map and reduce tasks always operate on the same subset of keys/nodes. We refer to the paired map/reduce task as *MRPair*. These tasks are persistent tasks that keep alive during the entire iterative process and maintain the intermediate iteration state. In summary, each MRPair performs the iterative computation on a data partition, and the necessary information exchange between MRPairs occurs during the maps-to-reduces shuffling.

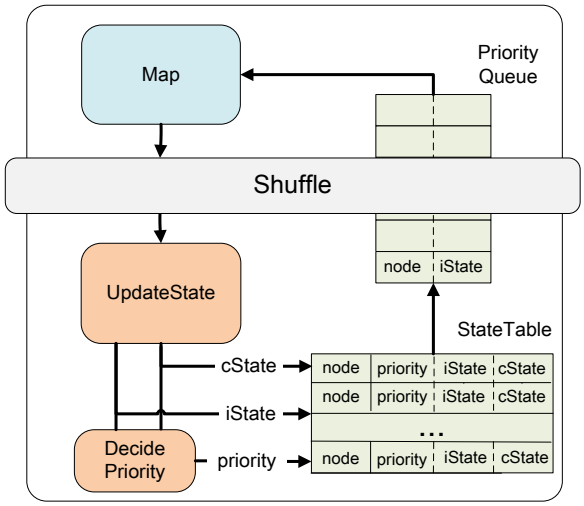


Figure 3: Data flow in a PrIter MRPair.

3.2 State Maintenance

Each MRPair is assigned with a subset of keys/nodes, whose values/states are maintained locally (Note that one or more fine-grained MRPairs could be assigned to a worker for load balancing which will be described in Section 3.6). During the iterative process, a key/node's value/state is updated after an iteration. That is, the value/state for each key/node should be maintained across iterations. To ensure fast access to the value/state, we design a *StateTable* at the reduce side that is implemented with an in-memory hash table.

In the context of incremental update (opposed to concurrent update in normal iteration), two types of state should be maintained. The first is the iterative state or *iState*, which is used for the iterative computation. The second is the cumulative state or *cState* indicating a node's state, which is accumulated from all the previous iterations. For example, in the SSSP algorithm (Section 2.1), the iState for node j is the shortest distance received from j's neighbors that have not been used for updating j's shortest distance, while the cState is the accumulated shortest distance for node j, which will be updated only if its iState is smaller than it. In PageRank (Section 2.2), the iState for page j is $\Delta R(j)$ that is the incremental PageRank score, while the cState is $R(j)$ that is the accumulated PageRank score. The key reason behind the separation of the two types of state is for supporting the incremental update. When performing an incremental update, we not only perform the iterative computation on the records to update their iterative state, but also need to maintain their accumulated state during iterations. Accordingly, two fields of the StateTable are designed to maintain the iState and the cState, which are indexed by node id.

In MapReduce, the output ⟨key, value⟩ pairs of various maps are sorted according to the natural order of keys, then the reduce function is performed on the grouped ⟨key, values list⟩ pair. However, since the StateTable supports random access, it is not necessary to perform sort between the map and reduce in PrIter, so that we eliminate the sort phase, which can significantly improve performance [24]. Moreover, we start the reduce operation immediately upon receiving a map's output. In other words, the "reduce" function is applied on ⟨key, value⟩ rather than ⟨key, values list⟩. It updates the corresponding entry in the StateTable according to a received value, rather than performs a reduce function on all the received values associated with the same key. We replace the reduce function by a *UpdateState* function, which updates the iState and the cState in the StateTable.

In summary, the StateTable stores the state information of each node. The state is updated every iteration by an UpdateState function, which takes map's output ⟨key, value⟩ pairs as input. Users can specify the update rules to achieve their goals.

3.3 Prioritized Execution

In order to perform prioritized execution, PrIter labels each node with a priority value that is specified by users. The priority information of each node is also maintained in the StateTable. During the update of node state, instead of a pass over the entire StateTable as an iteration, a pass through a selected subset as a subpass is performed based on the entries' priority values. A number of nodes with larger priority values are selected for the map operation in the next subpass. Since each MRPair holds only a subset of nodes, the priority value is compared among the nodes residing in the same MRPair instead of a global comparison across workers.

Figure 3 shows the data flow in a MRPair. The StateTable is updated in each subpass based on the output of the UpdateState function. The priority value is determined by another function *DecidePriority*, which is for users to specify each node's execution priority taking account of the state information. For example, in SSSP, the priority value is the negative value of the cState (*i.e.*, the shortest distance), while in PageRank, the priority value is exactly the same value as the iState (*i.e.*, ΔR). Upon the receipt of all maps' output, a priority queue containing the ⟨node, iState⟩ pairs with higher priority values is extracted from the StateTable for feeding the paired map in the next subpass. After a node is decided to be enqueued, its iState and its node id are made a copy in the priority queue, and accordingly its iState in the StateTable is reset.

The size of the priority queue shows the trade-off between the gain from the prioritized execution and the cost from the queue extraction. Setting it too long may degrade the effect of prioritization. In the extreme case that the queue size is the same size as the StateTable, there is no priority in the iterative computation. On the other hand, setting the queue too short may lead to frequent subpasses and as result incurs considerable overhead for the frequent queue extractions. However, the prioritized iteration is shown to improve the performance over a wide range of queue size settings as will be shown in Section 5.5. PrIter also provides a recommended queue size setting. Anyhow, there should be an optimal queue size that results the best performance, which will be discussed in detail in Section 4.1.

Once the queue size q is given, PrIter should extract the top q nodes with the highest priority values in each subpass. Sorting the whole StateTable can be expensive and time consuming. In practice, it is unnecessary to extract the exactly q top priority nodes. PrIter approximates the top records by a sampling method shown in Algorithm 1. The idea of this heuristic is that, the distribution of the priority values in a small (PrIter default 1000) samples set reflects the distribution of priority values in the large StateTable. By sorting the samples in the descending order of the priority values, the lower bound of the priority value of the top q records can be estimated to be the $(\frac{q \cdot s}{N})$th record's priority value in the sorted samples set. Through this approximation, PrIter takes $O(N)$ time on extracting the top priority nodes instead of $O(N \log N)$ time.

Algorithm 1: Priority queue extraction

input : StateTable table, StateTable size N, queue size q, samples set size s
output: priority queue queue

```
1  samples ← randomly select s records from table;
2  sort samples in priority-descending order;
3  cutindex ← q·s / N;
4  thresh ← samples [cutindex].priority;
5  i ← 0;
6  foreach record r in table do
7  |  if r.priority ≥ thresh then
8  |  |  queue [i] ← ⟨r.nodeid, r.iState⟩;
9  |  |  i ← i + 1;
10 |  end
11 end
```

3.4 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.

3.5 Online Top-k Query Support

Since each PrIter MRPair operates on a subset of nodes, after a number of map-reduce subpasses, it only has the knowledge of partial result, *i.e.*, cState values in StateTable. These partial results can be written to DFS for users to access. However, for some applications users might prefer to query the top-k records on-line.

The local top results in each MRPair can be extracted in parallel. A `DecideTopK` function is applied on each node's cState, which indicates a node's final cumulative state, to retrieve its top-k priority value (Note that the priority information based on cState helps the top-k results extraction, while the priority information based on iState helps prioritized iteration). The higher the top-k priority value is, the more likely it is in the top-k list. PrIter adopts the same sampling technique for generating priority queue to derive the local top-k nodes with higher top-k priority. These extracted local top results ($\langle$node, cState$\rangle$ pairs) from the running MRPairs are sent to a merge worker periodically, where they are merged into a global top-k result. Then, the global top-k result is written to DFS by the merge worker, such that users are able to see the top-k result snapshot periodically.

While the mechanism described above is straightforward, it might not scale. Scaling to a large number of MRPairs incurs heavy burden on the merge worker. We have two refinements on the naive mechanism. First, each PrIter MRPair sends less than k tops. Suppose there are m running MRPairs, on average each MRPair contributes only $\frac{k}{m}$ records to the global top-k records. We let each PrIter MRPair sends $\frac{4k}{m}$ top records to approximate the global top-k records. Second, the PrIter worker merges the local MRPairs' top records first before sending them to the merge worker. The pre-merge operation alleviates the merge worker's workload significantly.

3.6 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 [31].

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 sub-partitions 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.

3.7 API

PrIter has a few application programming interfaces exposed to users for implementing an iterative algorithm in PrIter. We summarize the APIs as follows:

- `initStateTable`: specify each node's 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.

Note that, 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* [30] that supports failure recovery can be utilized.

```
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();
```

Figure 4: PageRank example in PrIter.

In addition, there are a series of parameters users should specify, such as the number of graph partitions, the snapshot generation interval, and the termination condition. For better understanding, we walk through how prioritized PageRank is implemented in PrIter. Figure 4 shows the pseudo-code of this implementation. Basically, prioritized PageRank implementation follows the algorithm logic that we illustrated in Section 2.2. 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).

4. DISCUSSION

In this section, we first discuss the optimal queue size. We then consider an extension to PrIter that stores the StateTable on disk. This will address the scalability of PrIter.

4.1 Optimal Queue Size

As shown in Section 2, 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). 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:

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

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 5.1. Figure 5 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 StateTable size N. Given the queue size linear in $\sqrt{N}$, we can explore different settings of $\frac{q}{\sqrt{N}}$ to select the optimal queue size. In PrIter, we set $\frac{q}{\sqrt{N}}$ to be 100 by default. As will be shown in Section 5, the default setting gives competitive performance. Additionally, we will compare the different settings of $\frac{q}{\sqrt{N}}$ to see the performance difference in Section 5.5.

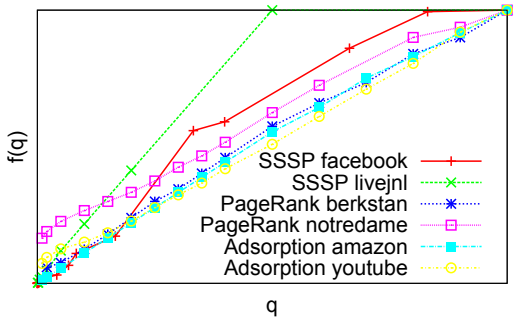


Figure 5: Function f for real graphs.

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 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.

4.2 On-Disk Extension

Although PrIter's StateTable is maintained in memory, it is possible to maintain the StateTable on disk. Suppose the number of the cluster workers is fixed, if the input graph becomes huge, the StateTable with billions of records cannot be loaded into memory. Hence, the on-disk extension will be helpful to scale PrIter.

The key problem of on-disk StateTable is the lack of random access support, such that it is not practical to update a specified StateTable entry in the UpdateState function. However, we are inspired by the same idea of MapReduce to solve this problem, where MapReduce realizes grouping keys by pre-sorting. In the map output files, the ⟨key, value⟩ pairs are sorted in the natural order of keys. Then, through a single pass of these map output files, the ⟨key, value⟩ pairs from various map output files are grouped by key for the reduce operation.

The on-disk version of PrIter can be implemented as shown in Figure 6. The StateTable is stored in a file with all the ⟨nodeid, priority, iState, cState⟩ tuples sorted in the nature order of nodeids. Meanwhile, the ⟨nodeid, value⟩s in the map output file are also sorted in the same order. As all the map output files are collected, we move the cursors (top-down) in these map output files and in the StateTable file to match the tuples that have the same nodeid. The tuples with the same nodeid are grouped together and passed to the UpdateState/DecidePriority function to update state and to decide priority value. The updated ⟨nodeid, priority, iState,

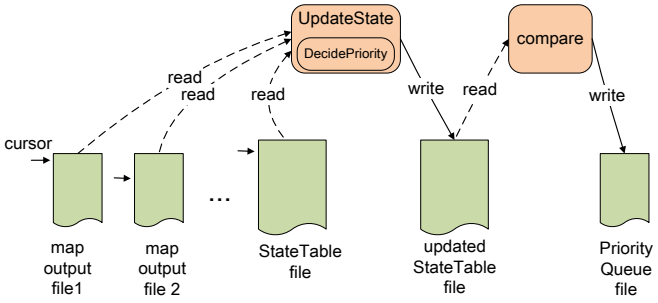


Figure 6: The data flow of the on-disk version of PrIter.

cState⟩ tuples are written to another file storing the updated StateTable. Note that, during the StateTable update, the sampling process for estimating the enqueue threshold (Section 3.3) proceeds simultaneously. Given the enqueue threshold, the updated StateTable file is then parsed by a compare function, by which the records whose priority values are larger than the enqueue threshold are written to the priority queue file for the next subpass.

Basically, comparing with MapReduce, there are two additional read passes of the StateTable file and one additional write of the priority queue file. We expect these additional local disk I/Os would not downgrade the performance seriously. The significant benefit from prioritized execution easily compensates for the performance losses. Therefore, this method can be feasible to scale PrIter.

5. EVALUATION

In this section, we evaluate our prototype implementation of PrIter. Our prototype is implemented based on Hadoop [2]. We have made our implementation available at [3]. Any Hadoop program can be implemented with PrIter. Users have the option to turn on/off the prioritized execution. In order to see the performance improvement from prioritized execution, we compare the priority-on PrIter with the priority-off PrIter. We also compare PrIter implementations with Hadoop implementations.

5.1 Experiment Setup

The experiments are performed on a cluster of local machines, and a large cluster of 100 medium instances on Amazon EC2 Cloud [1]. The local cluster consisting of 4 commodity machines is used to run small-scale experiments. Each machine has Intel E8200 dual-core 2.66GHz CPU, 3GB of RAM, and 160GB storage. These 4 machines are connected through a switch with bandwidth of 1Gbps.

Four iterative algorithms described in Section 2 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 [4]. 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 [28], where the weight reflects user interaction frequency. The web graphs for the PageRank computation are directed and unweighted. Note that, we perform Personalized PageRank on these real graphs, and 100 featured nodes in these graphs are randomly selected for computing Personalized PageRank. 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.

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 large-scale experiments, we generate a large synthetic web graph for PageRank computation. 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.

Table 1: Data Sets Summary

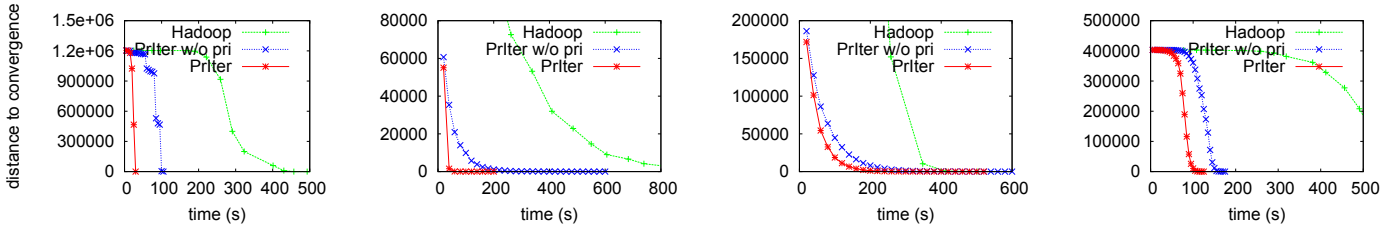
Algorithm	Graph	Nodes	Edges
SSSP	Facebook	1,204,004	20,492,148
	LiveJournal	4,847,571	68,993,773
	roadCA	1,965,206	5,533,214
PageRank	Berk-Stan	685,231	7,600,595
	Google	875,713	5,105,039
	Notredame	325,729	1,497,134
	Synth_WEB	100,000,000	1,216,907,427
Adsorption	Facebook	1,204,004	20,492,148
	Youtube	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

5.2 Convergence Speed

PrIter prioritizes the computation by performing update on the dominant data that contribute to the convergence the most. As a result, the iterative algorithms approach to the convergence point with less node activations, which means less computation workload and less amount of network communication. Therefore, the algorithms implemented with prioritization will converge faster than that without prioritization.

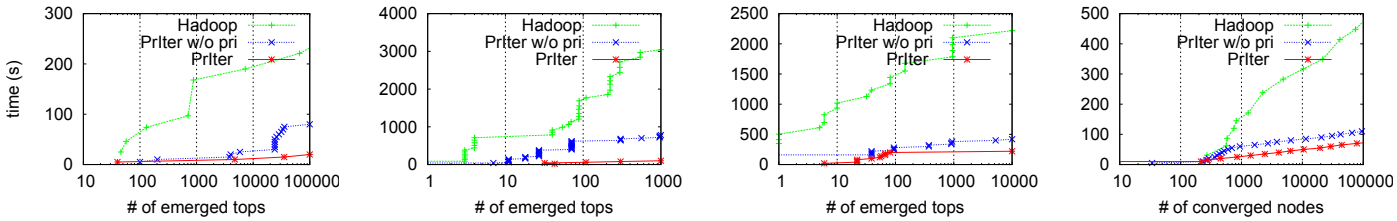
To evaluate the effect of prioritized execution, we compare the convergence rate by turning on and off the prioritized execution. Additionally, we also compare PrIter with the traditional Hadoop. We let PrIter generate a result snapshot every few seconds, and calculate its distance to the final result, which has been pre-computed offline. For the Hadoop implementations, we record the snapshot after completing each job and measure the distance to the convergence point. The distance in SSSP/ConnComp is defined as the number of nodes that have not yet finalized their shortest distances/component ids. In PageRank/Adsorption, we use L1-Norm distance between the current PageRank/label distribution vector and the final converged vector.

We perform the convergence speed experiment on our local cluster. The experiment results are shown in Figure 7. We can see that the PrIter implementations with prioritized execution converge faster than that without prioritized execution. Overall, the prioritized execution of PrIter speeds



(a) SSSP (Facebook) (b) PageRank (BerkStan) (c) Absorption (Facebook) (d) ConnComp (Amazon)

Figure 7: Convergence speed.



(a) SSSP (Facebook) (b) PageRank (BerkStan) (c) Adsorption (Facebook) (d) ConnComp (Amazon)

Figure 8: Top record emergence time.

up the convergence by a factor of 2 to 8. Further, the convergence speed of the PrIter implementations is much faster than that of the Hadoop implementations, where a more significant speedup ranging from 5x-50x is achieved.

5.3 Top Record Emergence Time

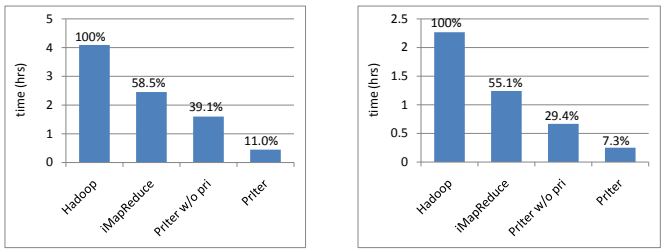
In reality, peoples are always interested in a small collection of more attractive records. For example, users always care about the closest nodes in the shortest path problem, and they are only interested in the first few pages of Google search results. The time it takes to derive the top records is 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 8 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.

5.4 Large Scale Experiment

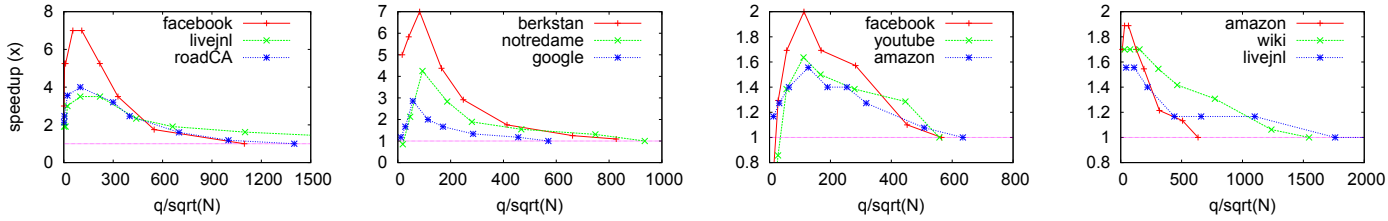
To evaluate the performance of PrIter in a large scale environment, we deploy PrIter on our Amazon EC2 Cluster that involves 100 medium instances. We run PageRank and Personalized PageRank algorithms for Synthetic_WEB graph with PrIter, Hadoop, and iMapReduce [32]. iMapReduce stores the intermediate iteration state in files rather than store it in memory, and it supports iterative processing and improves the performance mainly by eliminating the shuffling of the static data.



(a) PageRank (b) Personalized PageRank

Figure 9: The convergence time on Amazon EC2.

Figure 9 shows the convergence time of PageRank and Personalized PageRank, by using Hadoop, iMapReduce, priority-off PrIter, and priority-on PrIter. We take the convergence time of Hadoop as a point of reference. iMapReduce reduces



(a) SSSP (b) PageRank (c) Adsorption (d) ConnComp

Figure 10: The effect of priority queue size (different settings of $\frac{q}{\sqrt{N}}$).

the convergence time by 45%, which is achieved mainly by the avoidance of shuffling static data. The priority-off PrIter reduces the convergence time to about 35%, and it also improves iMapReduce by about 40% primarily because of the elimination of sort phase, which has been illustrated in previous work [24]. Moreover, the priority-on PrIter reduces the convergence time of the priority-off PrIter by an additional factor of 3 to 4, and the running time is reduced to only 10% of the Hadoop's running time. Additionally, since Personalized PageRank performs random walk from 100 randomly selected starts (*i.e.*, a smaller number of dominant nodes with large initial ΔR), which is much smaller than that in the standard PageRank (start from all nodes), the prioritization should have more positive effect on promoting convergence. As expected, the results show better performance on Personalized PageRank than on Pagerank.

5.5 Optimal Queue Size

As mentioned in Section 4.1, there is an optimal priority queue size that results in the best performance, and it depends on several factors. Since only the StateTable size N is easy to obtain, we will set the queue size based on the factor of the StateTable size. In other words, PrIter sets the queue size q in proportion to $\sqrt{N}$. To see the effects of different settings of $\frac{q}{\sqrt{N}}$, we perform a series of experiments on our local cluster, running each algorithm on three different real graphs.

Figure 10 shows the convergence speedup results varying algorithms and inputs. We can see that most of them reach their optimal points when $\frac{q}{\sqrt{N}}$ is set to be between 50 and 200, which provides positive support to our default selection ($\frac{q}{\sqrt{N}} = 100$).

6. RELATED WORK

With the increasing popularity of MapReduce [10, 22] and its open source implementation Hadoop [2], a series of distributed computing frameworks have been proposed these years, such as Dryad/DryadLINQ [13, 29], Hive [27], Pig [21], and Comet [12]. These efforts directly promote the development of cloud computing. However, these proposed frameworks unanimously embraced a batch processing model, which limits their potential to efficiently implement iterative algorithms. To address this problem, there are a number of efforts targeted on providing efficient frameworks for the distributed implementations of iterative algorithms [7, 32, 24, 30, 11, 19, 14, 17].

Shuffling static data during the iterative process incurs significant communication overhead. HaLoop [7] 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 locate. iMapReduce [32] 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 3.1, PrIter integrates iMapReduce to avoid the unnecessary static data shuffling.

Recently, a series of frameworks that maintain the iteration state in memory have been proposed for iterative computations. Piccolo [24] 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. Percolator [23] provides distributed transaction support for the BigTable datastore [8], which is used by Google to enable incremental processing of web indexes. Spark [30] 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. [17] 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 [20] supports data-dependent iterative or recursive algorithms by building an abstract dynamic task graph. Twister [11] 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.

Pregel [19] is another in-memory system, which provides an expressive model for programming graph-based algorithms. It uses a pure message passing model to process graphs, and the iterative algorithms in Pregel are expressed as a sequence of supersteps. Basically, a node performs message passing and votes to halt after finishing its computation in a superstep. While the idea of prioritized execution can be integrated into Pregel as well.

GraphLab [18] improves upon MapReduce abstraction by compactly expressing asynchronous iterative algorithms with sparse computational dependencies. The GraphLab data model relies on shared memory and provides a shared data table (SDT) to maintain vertex state and edge state, while PrIter addresses distributed commodity machines exchanging information through network message passing. In addition, the asynchronous programming model in GraphLab requires data consistency models to prevent data-races, and the asynchronous model can also lead to non-deterministic behavior, which depends largely on the asynchronous update order. PrIter proposes rearranging the update order considering node state instead of blindly processing nodes in a synchronous manner, which can be adopted by GraphLab to support efficient asynchronous update.

PrIter accelerates the convergence of iterative algorithms by the prioritized execution of iterative updates. A priority value is assigned to each data point (represented by a key), and only the high priority data points are executed in each iteration. To the best of our knowledge, this is the first work that supports the prioritized execution for iterative computations.

7. CONCLUSIONS

Parsing massive data set iteratively is a time-consuming process. In this paper, we argue that the prioritized execution of iterative computations accelerates iterative algorithms. To support prioritized iteration, we propose PrIter, a distributed framework for fast iterative computation running on a large cluster of commodity machines. Experiments are performed in the context of various applications to evaluate PrIter. The experimental results show that PrIter significantly improves performance over that achieved by Hadoop.

The key idea of PrIter is that, it enables selecting a subset of data to perform updates in each pass of the data, rather than performing updates on all data. In particular, PrIter selects a subset of data that ensures fast convergence of the iterative process. Therefore, each data point is given a priority value to indicate the importance of the update operation on that data point. PrIter extracts the subset of data that have higher priority values to perform the iterative updates. We show not only that PrIter is feasible for iterative algorithms, but also that it is effective in a large distributed environment.

PrIter reduces the amount of expensive data access by performing much cheaper data operations. With the development of powerful multi-core CPU and high-capacity RAM memory, processing data is much faster than before, while disk/network I/O becomes the bottleneck of computations nowadays, especially under a distributed environment. In our experimental EC2 cluster, we has the measurement that the time spent on accessing 10^{10} bytes data (including network I/O and disk I/O) is about 10000 times the time spent on 10^{10} double format multiplications. Moreover, some industry colleagues complain the networking bandwidth bottleneck in production clusters as well. PrIter is proposed to balance the amount of data accesses and the amount of data operations in a reasonable manner, which spends more on data operations (extracting priority) to reduce the remote data accesses (data shuffles), and the balance can be tuned by controlling the size of the priority queue. We consider that it should be a correct direction to shift more resources from data accesses to data operations.

Acknowledgment

This work is partially supported by U.S. NSF grants CCF-1018114. Yanfeng Zhang was a visiting student at UMass Amherst, supported by China Scholarship Council, when this work was performed. We would like to thank Samamon Khemmara, Renjie Zhou, Jiangtao Yin, as well as the anonymous reviewers, for their insightful comments.

8. REFERENCES

[1] Amazon ec2. http://aws.amazon.com/ec2/.

[2] Hadoop. http://hadoop.apache.org/.

[3] Priter project. http://code.google.com/p/priter/.

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

[5] 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 *WWW '08*, pages 895–904, 2008.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW '98*, pages 107–117, 1998.

[7] Y. Bu, B. Howe, M. Balazinska, and D. M. Ernst. Haloop: Efficient iterative data processing on large clusters. In *VLDB '10*, 2010.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[9] 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 *NIPS*, pages 281–288, 2006.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*, pages 10–10, 2004.

[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *MapReduce '10*, pages 810–818, 2010.

[12] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *SoCC '10*, pages 63–74, 2010.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07*, pages 59–72.

[14] U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM '09*, pages 229 –238, 2009.

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

[16] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci. Technol.*, 58(7):1019–1031, 2007.

[17] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SoCC '10*, pages 51–62, 2010.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD '10*, pages 135–146, 2010.

[20] 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 *NSDI'11*, 2011.

[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110, 2008.

[22] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09*, pages 165–178, 2009.

[23] D. Peng and F. Dabe. Large-scale incremental processing using distributed transactions and notifications. In *OSDI '10: Proceedings of the 9th conference on Symposium on Opearting Systems Design and Implementation*, pages 1–15, 2010.

[24] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI'10*, 2010.

[25] N. Slonim, N. Friedman, and N. Tishby. Unsupervised document classification using sequential information maximization. In *SIGIR '02*, pages 129–136, 2002.

[26] H. H. Song, T. W. Cho, V. Dave, Y. Zhang, and L. Qiu. Scalable proximity estimation and link prediction in online social networks. In *IMC '09*, pages 322–335, 2009.

[27] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. In *VLDB '09*, pages 1626–1629, 2009.

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

[29] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08*, pages 1–14, 2008.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud'10*, pages 10–10, 2010.

[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI '08*, pages 29–42, 2008.

[32] Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. In *DataCloud '11*, 2011.

[33] T. Zhou, Z. Kuscsik, J.-G. Liu, M. Medo, J. R. Wakeling, and Y.-C. Zhang. Solving the apparent diversity-accuracy dilemma of recommender systems. *Proceedings of the National Academy of Sciences*, 107(10):4511–4515, March 2010.

APPENDIX

A. SELECTIVE INCREMENTAL PAGERANK

In this section, 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

$$\sum_{l=0}^{\infty}(1-d)d^lW^lE = (1-d)E/(I-dW), \qquad (12)$$

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:

$$R_{inc}^{(k)}(j) = \Delta R_{inc}^{(0)}(j) + \sum_{l=1}^{k} \left\{ d^l \sum_{\{i_0,\ldots,i_{l-1},j\} \in P(j,l)} \frac{\Delta R_{inc}^{(0)}(i_0)}{\prod_{h=0}^{l-1} deg(i_h)} \right\}, \quad (13)$$

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), we have

$$R_{inc}^{(k)} = \Delta R_{inc}^{(0)} + dW\Delta R_{inc}^{(0)} + \ldots + d^k W^k \Delta R_{inc}^{(0)}. \quad (14)$$

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:

$$R_{sel}^{(k)}(j) = \Delta R_{sel}^{(0)}(j) + \sum_{l=1}^{k} \left\{ d^l \sum_{\{i_0,\ldots,i_{l-1},j\} \in P'(j,l)} \frac{\Delta R_{sel}^{(0)}(i_0)}{\prod_{h=0}^{l-1} deg(i_h)} \right\}, \quad (15)$$

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). □

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,* $R_{sel}^{(\infty)} = R_{inc}^{(\infty)}$.

B. PRIORITIZED INCREMENTAL PAGERANK

In this section, 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.

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,

$$\frac{||\Delta R_{sel}(S)||_1}{|S|} > ||\Delta R_{sel}(V-S)||_1. \quad (16)$$

That is,

$$\max_{j \in S}(\Delta R_{sel}(j)) > \max_{j \in V-S}(\Delta R_{sel}(j)). \quad (17)$$

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* $R_{inc}^{(\infty)}$.