PrIter: A Distributed Framework for Prioritizing Iterative Computations

Yanfeng Zhang, Qixin Gao, Lixin Gao, *Fellow, IEEE*, and Cuirong Wang

pervasive:普遍的 re：推荐系统 essential最本质的 discrimination无差别的

Abstract—Iterative computations are pervasive among data analysis applications, including web search, online social network analysis, recommendation systems, and so on. These applications typically involve data sets of massive scale. Fast convergence of the iterative computations on the massive data set is essential for these applications. In this paper, we explore the opportunity for accelerating iterative computations by prioritization. Instead of performing computations on all data points without discrimination, we prioritize the computations that help convergence the most, so that the convergence speed of iterative process is significantly improved. We develop a distributed computing framework, PrIter, which supports the prioritized execution of iterative computations. PrIter either stores intermediate data in memory for fast convergence or stores intermediate data in files for scaling to larger data sets. We evaluate PrIter on a local cluster of machines as well as on Amazon EC2 Cloud. The results show that PrIter achieves up to 50 × speedup over Hadoop for a series of iterative algorithms. In addition, PrIter is shown better performance for iterative computations than other state-of-the-art distributed frameworks such as Spark and Piccolo.

priter:基于R(k)=R(k-1)+!R(k)，通过!R的大小，选择部分node（priority scheduling）进行迭代
基于mapreduce，并没有涉及到asynchronous

Index Terms—PrIter, prioritized iteration, iterative algorithms, MapReduce, distributed framework

idea：在迭代计算中，有先后次序的执行各个操作。（即优先执行那些有利于计算收敛的那些计算）

myriad无数的

1 Introduction

Iterative computations are common in myriad of data mining algorithms. PageRank [2], as a well-known iterative algorithm, has been widely used in web search engines. Other iterative algorithms such as Adsorption [3] and Expected Hitting Time [4] are applied to the problem domains such as link prediction [4] and recommendation systems [5]. In computational biology, iterative algorithms such as K-means clustering algorithm [6] 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 [7], which powered cloud computing, is such a framework that supports data processing of massive scale. Dryad/DryadLINQ [8], [9], Hadoop [10], Pig [11], Hive [12], and Pregel [13] have been proposed as well. In particular, all of the previously proposed frameworks assume that the iterative update is equally important for all data points.

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. PrIter allows either to store data in memory for better performance or to store data in files for better scalability. 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 PageRank on Amazon EC2 Cloud [14] as well as on a local cluster. Our experimental results show that PrIter significantly speeds up the convergence of the iterative computations, which achieves up to 50× 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 2 × -8× speedup over that without prioritization. In addition, the file-based PrIter is shown to have competitive performance, which is only 2 times slower than the memory-based PrIter, but it can scale to much larger data sets.

- *Y. Zhang is with the Computing Center, Northeastern University, No. 11, Lane 3, WenHua Road, HePing District, Shenyang, Liaoning 110819, China. E-mail: zhangyf@cc.neu.edu.cn.*
- *Q. Gao and C. Wang are with Northeastern University at Qinhuangdao, 143 Taishan Road, Qinhuangdao, Hebei 066004, China. E-mail: {gaoqx, wangcr}@mail.neuq.edu.cn.*
- *L. Gao is with Department of Electrical and Computer Engineering, University of Massachusetts Amherst, 151 Holdsworth Way, Amherst, MA 01003. E-mail: lgao@ecs.umass.edu.*

Manuscript received 3 June 2012; revised 26 Aug. 2012; accepted 4 Sept. 2012; published online 11 Sept. 2012.
Recommended for acceptance by J. Wang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-06-0529.
Digital Object Identifier no. 10.1109/TPDS.2012.272.

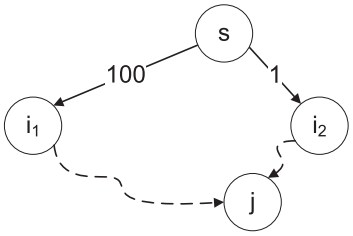


Fig. 1. A SSSP example.

The rest of the paper is organized as follows: In Section 2, we illustrate the benefit of prioritized iteration through two example algorithms. Section 3 presents the system design of PrIter, which maintains intermediate data in memory. In Section 4, we extend PrIter to maintain intermediate data in files, so that it is able to scale to much larger data sets. 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. Comparing to the conference version [1], this paper improves the scalability of PrIter by proposing file-based PrIter and adds more experimental results.

2 Motivating Examples

In this section, we describe two well-known iterative algorithms that benefit from the prioritized execution. More motivating examples can be found in Section 1 of the supplementary file, can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.272.

2.1 Single Source Shortest Path (SSSP)

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:

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

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.

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 Fig. 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. 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 webpages. Later on, these algorithms have found a wide range of applications, such as link prediction [4], [15]. PageRank ranks webpages 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 webpages, 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 a 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

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

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 nonzero. Below we will focus on discussing the prioritized iteration for PageRank. However, similar argument follows for Personalized PageRank.

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 (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 $\mathbf{0}$. In Section 2.1 of the supplementary file, available online, we have shown that as long as each node is activated an infinite number of times, 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 (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,

$$\left\|R_{sel}^{(k)}\right\|_1 = \left\|R_{sel}^{(k-1)}\right\|_1 + d\left\|\Delta R_{sel}^{(k-1)}(S^k)\right\|_1.$$

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 $R_{sel}^{(k-1)}$ toward $R_{sel}^{(\infty)}$.

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 (4), which is always selecting nodes in S^* to activate in each subpass but ignoring the nodes in $V - S^*$. That is, 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 Section 2.2 of the supplementary file, available online).

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.

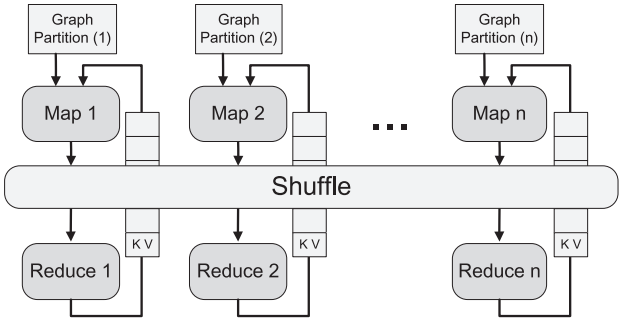


Fig. 2. Iterative processing structure.

3 PrIter Design

In this section, we propose PrIter, a distributed framework for prioritized iterative computations, which is implemented based on Hadoop MapReduce [10]. 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. 支持迭代计算，并进行迭代间的信息传递
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. 中间状态的维护（正确性）
3. The framework needs to support prioritized execution. That is, an efficient selection of the high-priority data should be provided. 支持优先操作的选择

PrIter provides the functionalities of iterative processing (Section 3.1), state maintenance (Section 3.2), prioritized execution (Section 3.3), and online top-k query (Section 3.4). PrIter also supports termination check (Section 3.1 of the supplementary file, available online), load balancing, and fault-tolerance (Section 3.2 of the supplementary file, available online). We describe PrIter's API and show a representative PageRank implementation example with PrIter in Section 4 of the supplementary file, available online.

3.1 Iterative Processing

PrIter incorporates the support of iMapReduce [16] 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). Fig. 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 (nid), 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 $nids$. A hash function F applied on the keys/nodes is used to split the static graph data and the initial node state data evenly into n partitions according to $pid = F(nid, n)$, 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 map output $\langle \text{key}, \text{value} \rangle$/ $\langle \text{node}, \text{state} \rangle$ pairs are shuffled to the reduce tasks according to the same hash function, $rid = F(nid, n)$, where rid is the reduce task id and n is the number of reduce tasks. (Note that, PrIter requires the number of reduce tasks to be equal to the number of map tasks.) Accordingly, a reduce task is connected to the map task with the same 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 $\langle \text{node}, \text{state} \rangle$ pairs are directed to the connected map as the map's input. By using the same hash function $F(nid, n)$ 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.

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 is described in Section 3.2 of the supplementary file, available online.) 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 traditional iterative computations), 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 nid.

In MapReduce, the output $\langle \text{key}, \text{value} \rangle$ pairs of various maps are sorted according to the natural order of keys, then the reduce function is performed on the grouped $\langle \text{key}, \text{values list} \rangle$ 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 [17]. Moreover, we start the reduce operation immediately upon receiving a map's output. In other words, the "reduce" function is applied on $\langle \text{key}, \text{value} \rangle$ rather than $\langle \text{key}, \text{values list} \rangle$. It updates the corresponding entry in the StateTable according to a received value, rather than performing a reduce function on all the received values associated with the same key. We replace the reduce function by an *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 $\langle \text{key}, \text{value} \rangle$ pairs as input. Users can specify the update rules to achieve their goals.

3.3 Prioritized Execution

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.

Fig. 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 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 $\langle \text{node}, \text{iState} \rangle$ 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 nid are copied in the priority queue, and accordingly its iState in the StateTable is reset.

The size of the priority queue demonstrates the tradeoff 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. (Discussion of the optimal queue size can be found in Section 4 of the supplementary file, available online.) However, the prioritized iteration is shown to improve the performance over a wide range of queue size settings as shown in Section 6.4 of the supplementary file, available online.

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 exact 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 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 approximated to be the $(\frac{q \cdot s}{N})$th record's priority value in the sorted samples set. Intuitively, the more samples the more accuracy of this approximation is obtained but the more time is consumed. In Section 6.3 of the supplementary file, available online, we show the effectiveness of the sampling approach. 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 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 online.

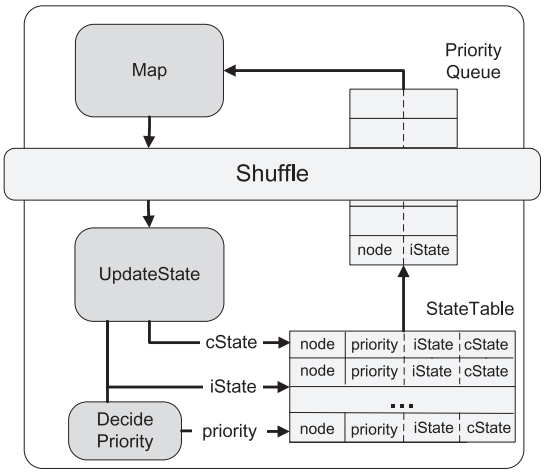


Fig. 3. Data flow in a PrIter MRPair.

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 ($\langle \text{node}, \text{cState} \rangle$ 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 premerge operation alleviates the merge worker's workload significantly.

4 File-Based PrIter

PrIter relies on in-memory StateTable to proceed prioritized iteration. Suppose the number of workers is fixed, if the input graph becomes huge, the StateTable with billions of records cannot be loaded into memory. In this section, we extend PrIter to store data in files so as to scale to much larger data sets.

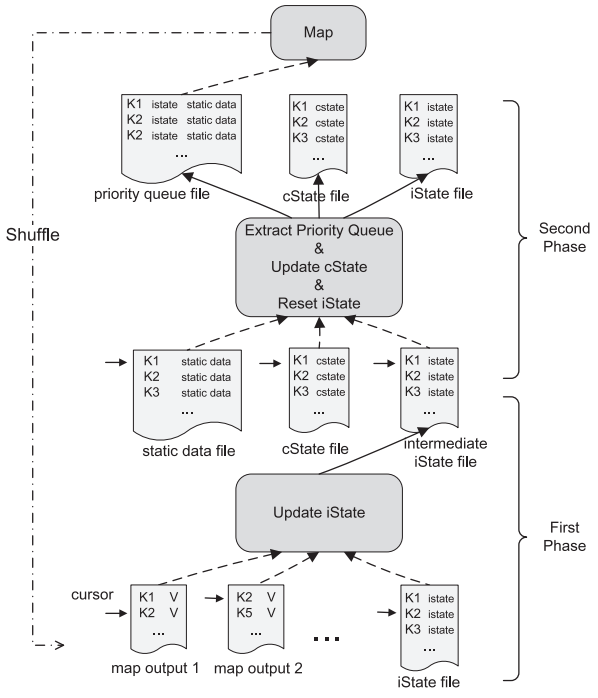


Fig. 4. The data flow in the file-based PrIter.

4.1 Data in Sorted Files

The key problem of file-based system is the lack of fast 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 idea of MapReduce to solve this problem, where MapReduce realizes grouping keys by presorting. In the map output files, the $\langle \text{key}, \text{value} \rangle$ pairs are sorted in the natural order of keys. In the reduce phase, by sequential access to multiple files in a single pass, the $\langle \text{key}, \text{value} \rangle$ pairs from various map output files are grouped by key for the reduce operation.

The file-based PrIter stores every key/node's information (line by line) in local files and in the order of their keys/nids. In each MRPair, iState and cState are stored in separated files, i.e., *iState file* and *cState file*. The static data, for example, the linklist of each node, are also stored in a local file in the order of their keys (i.e., *static data file*). Since the computation for deciding each key's priority is lightweight, we will compute the priority value when needed, so the priority values are not stored in file to avoid the relatively more expensive disk I/Os. As these files are sorted in the same order of keys, when querying every node's information, we can sequentially parse these separated files (line by line) in the same pace.

4.2 Two-Phase Update

As presented in Section 3.2, PrIter starts to update the in-memory StateTable immediately upon receiving a map output file, and it extracts a priority queue after receiving all map output files. However, in the file-based PrIter, we start to perform update after receiving all map output files. Basically, given the map output files and the initial iState/cState/static data files, the file-based PrIter performs update and extracts priority queue by two phases of parsing these files. As shown in Fig. 4, the first phase aggregates the map output files and the iState file to produce an *intermediate iState file*, and the second phase reads the intermediate iState file, cState file, and static data file to produce the updated iState and cState files and the *priority queue file*.

In the first phase, we leverage Hadoop's merging mechanism, which groups the key-value pairs with the same key by parsing the sorted map output files. We add the sorted iState file in this merge phase. We move the line cursors (top down) in these map output files and in the iState file to match the tuples that have the same key. The values in the matched tuples are aggregated, and the aggregated result is the new iState, which is written to the intermediate iState file. Along with the aggregation, we also perform the sampling process (Section 3.3) to retrieve a priority threshold for priority queue extraction in the second phase.

In the second phase, by parsing the iState values in the intermediate iState file, we decide each key's priority value. With the already retrieved priority threshold in the first phase, the keys with priority values higher than the threshold will be extracted. As a high-priority key is selected, the key along with its iState and its static data are written to the priority queue file. Note that, because the map operation will use the static data (Section 3.1), we also retrieve the static data from the static data file. In the meantime, the selected key's cState is updated, and its iState is reset to the default value. Accordingly, we need to read the old cState file and output the updated cState/iState. If a key is not selected, its cState and iState are unchanged to be written to the new cState and iState files, respectively.

After these files have been parsed, we generate a priority queue file, in which each line contains the high-priority key, iState, and static data information. It then notifies the local map task to process the priority queue file to start the next iteration. Based on the iState and the static data, the map produces output files and shuffles them to reduce tasks to continue the iterative process.

Comparing with the memory-based PrIter, the file-based PrIter requires additional local disk I/Os for updating state and generating priority queue. However, it can scale to much larger data sets. In Section 5, we will show that the file-based PrIter can give competitive performance.

5 Evaluation

We implement a prototype of PrIter based on Hadoop [10]. Any Hadoop program can be implemented with PrIter. We also have made our implementation available at [18]. In this section, we evaluate our prototype implementation of PrIter. Note that, besides the results shown in this section, we also include more additional results on the top record emergence time, the effectiveness of sampling queue extraction, and the impact of priority queue size in Section 6 of the supplementary file, available online.

5.1 Environment Setup

The experiments are performed on Amazon EC2 Cloud [14] and on a cluster of local machines. The experiment on EC2 involves 100 medium instances, each with 1.7-GB memory and five EC2 compute units. The local cluster consisting of four commodity machines is used to run small-scale experiments. Each machine has Intel E8200 dual-core

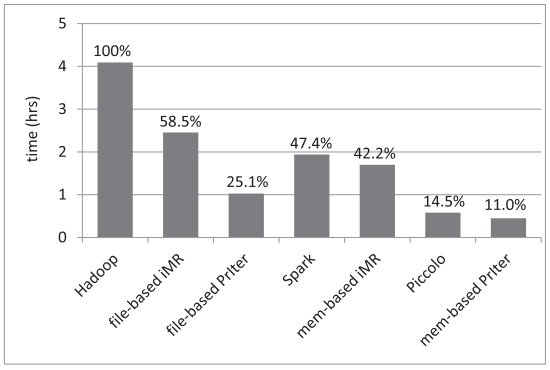


Fig. 5. The convergence time of PageRank (100M node synthetic graph) on Amazon EC2.

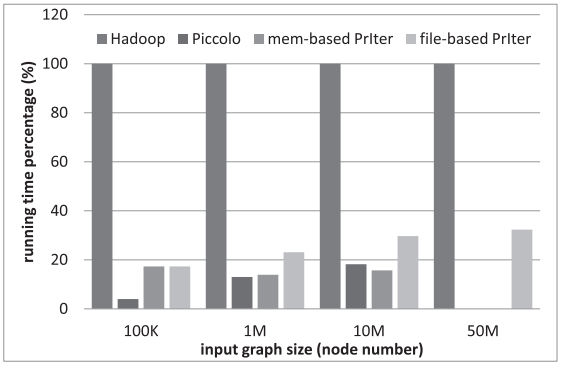


Fig. 6. The running time of PageRank on local cluster for synthetic graphs with different graph sizes.

2.66-GHz CPU, 3-GB of RAM, and 160-GB storage. These four machines are connected through a switch with bandwidth of 1 Gbps. The experiment data sets, including a number of real data sets and synthetic data sets, are described in Section 6.1 of the supplementary file, available online.

5.2 Overall Performance

To evaluate the performance of PrIter in a large scale environment, we deploy PrIter on Amazon EC2 Cloud involving 100 medium instances. We run PageRank on a very large synthetic webgraph (containing 100,000,000 nodes) with memory-based PrIter, file-based PrIter, memory-based iMapReduce, file-based iMapReduce, Piccolo, Spark and Hadoop. iMapReduce [16], an extended version of Hadoop, supports iterative processing by launching persistent tasks and maintaining the intermediate iteration state in local files (file-based iMapReduce) or in local memory (memory-based iMapReduce), and it improves the performance mainly by eliminating the shuffling of the static data. Piccolo [17], which is implemented with C++, allows to operate a global table stored in distributed machines, with each machine holding a part of the table in local memory. Users can implement iterative algorithms by accessing in-memory distributed tables. In addition, Piccolo uses MPI to communicate between distributed workers to improve performance. Spark [19], [20] is another state-of-the-art cloud-based framework that supports large-scale iterative computations. Spark utilizes *resilient distributed data set* (RDD), which is a read-only collection of objects maintained in memory across iterations. RRD makes three replicas for each piece of data to support fault tolerance but sacrifices performance.

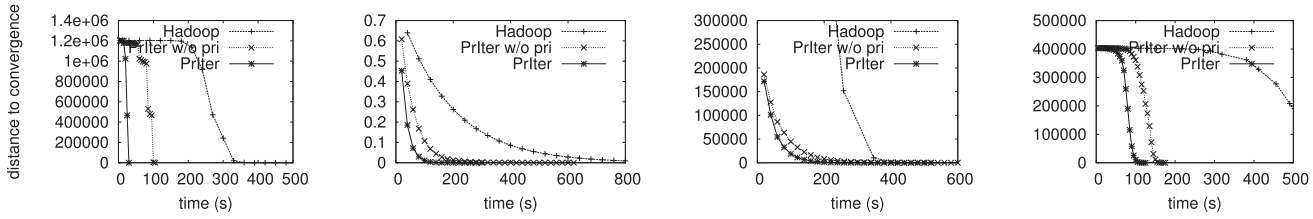
Fig. 5 shows the convergence time of PageRank, by using Hadoop, file-based iMapReduce, file-based PrIter, Spark, memory-based iMapReduce, Piccolo, and memory-based PrIter. We take the convergence time of Hadoop as a baseline, which takes 100 percent time to convergence. The file-based iMapReduce reduces the convergence time by about 42 percent, which is achieved mainly by the avoidance of static data shuffling. The file-based PrIter reduces the Hadoop convergence time to 25 percent by prioritized computation. These three frameworks store intermediate data in files, which can scale to much larger data sets. Since disk access is much slower than memory access, the file-based iMapReduce and the file-based PrIter are expected to be slower than their in-memory versions. However, we can see that the file-based PrIter only takes around two times longer time than the memory-based PrIter, but it allows to scale to much larger data sets. Spark, famous as a fault tolerant memory-based cloud system, sacrifices performance for data redundancy. As depicted in the figure, Spark only reduces Hadoop convergence time by 53 percent (Spark should perform better with controlled partitioner [20]). Piccolo also stores data in memory and leverages MPI for high performance communication, which reduces Hadoop convergence time to 14 percent.

The memory-based PrIter performs the best, which reduces Hadoop convergence time to only 11 percent. Since PrIter incorporates iMapReduce for efficient iterative processing (see Section 3.1), the performance improvement by the memory-based PrIter results from 1) the optimizations from the memory-based iMapReduce and 2) the prioritized computation. As shown in the figure, the memory-based iMapReduce reduces Hadoop running time to 42.2 percent, which is mainly achieved by avoiding static data shuffling [16] and eliminating sort phase [16], [17]. Thanks to the prioritized computation, the memory-based PrIter further reduces the memory-based iMapReduce running time to almost 1/4 (from 42.2 to 11 percent).

5.3 Varying Input Size

To illustrate PrIter's performance with different input size, we, respectively, generate synthetic graphs with 100,000 nodes, 1,000,000 nodes, 10,000,000 nodes, and 50,000,000 nodes and run PageRank on our local cluster with the file-based PrIter, memory-based PrIter, Piccolo, and Hadoop.

We record the convergence time of these frameworks on different-size graphs. For the same size graph, we consider Hadoop's running time as 100 percent of the running time. As shown in Fig. 6, the memory-based PrIter and the file-based PrIter always converge faster than Hadoop, which only take 20-30 percent of the Hadoop's running time. The file-based PrIter requires additional local disk I/Os, which impacts the performance. Therefore, the file-based PrIter performs a little worse (around two times slower) than the memory-based PrIter for 100K-node graph, 1M-node graph, and 10M-node graph. On the other hand, Piccolo performs well on small-size graph. But as the graph size increases, Piccolo is not as good as the memory-based PrIter. In addition, Piccolo and the memory-based PrIter cannot process large graphs due to memory space limitation, while the file-based PrIter stores intermediate data in files, which can scale to much larger data sets.



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

Fig. 7. Convergence speed.

5.4 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. To illustrate the effect of prioritized iteration in more general sense, we perform the experiments in the context of four popular applications using the real data sets described in Table 1 of the supplementary file, available online. We let PrIter generate a result snapshot every few seconds, and calculate its distance to the final result, which has been precomputed 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 Fig. 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 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 $5\times - 50\times$ is achieved.

6 Related Work

With the increasing popularity of MapReduce [7], [21] and its open source implementation Hadoop [10], a series of distributed computing frameworks have been proposed these years, such as Dryad/DryadLINQ [8], [9], Hive [12], MapReduce Online [22], and Pig [11]. 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. These include Twister [23], HaLoop [24], iMapReduce [16], Piccolo [17], Spark [19], [20], CIEL [25], and so on.

Among these works, Pregel [13] 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. The idea of prioritized execution can be integrated into Pregel as well. GraphLab [26] improves upon MapReduce abstraction by compactly expressing asynchronous iterative algorithms with sparse computational dependencies. The GraphLab data model relies on shared memory to maintain vertex state and edge state. Most recently, the authors extend the GraphLab framework to the substantially more challenging distributed setting while preserving strong data consistency guarantees [27]. However, the asynchronous programming model in GraphLab requires data consistency models to prevent data races, and the asynchronous model can also lead to nondeterministic 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.

Besides these recently proposed large scale distributed frameworks, Jack Dongarra et al. have contributed a number of open source software packages for parallel sparse matrix vector iterative computation, including BLAS [28] and LAPACK [29]. PrIter's idea of selective update in each iteration differs from the traditional synchronous iteration in these previous works and can be an optimization to be applied in these softwares. In addition, many early stage studies laid the foundation of asynchronous iteration and have proved its effectiveness and convergence [30], [31], [32]. Our work provides a new selective update iteration scheme, which explores the priority property to accelerate distributed iterative computation.

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 sets iteratively is a time-consuming process. In this paper, we argue that the prioritized execution of iterative computations accelerates iterative algorithms. Prioritized iteration enables selecting a subset

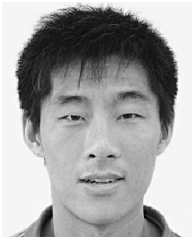
of data that ensure fast convergence to perform updates, rather than performing updates on all data. We formally prove that the prioritized iteration converges to the correct results. With prioritized iteration, the iterative process proceeds more effectively. To support prioritized iteration in a distributed environment or in a cloud, we propose PrIter. For better scalability, we also propose file-based PrIter that stores all data in files. 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.

ACKNOWLEDGMENTS

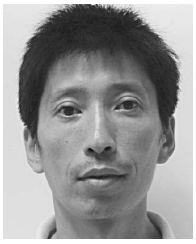
This work was partially supported by US National Science Foundation (NSF) grants (CCF-1018114, CNS-1217284), Fundamental Research Funds for the Central Universities (N120416001, N120816001, N100704001), China Mobil Labs Fund (MCM20122051), MOE-Intel Special Fund of Information Technology (MOE-INTEL-2012-06), and National Natural Science Foundation of China (61272179). A preliminary version [1] appeared in the Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11), 2011.

REFERENCES

[1] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A Distributed Framework for Prioritized Iterative Computations," *Proc. ACM Symp. Cloud Computing (SOCC '11),* 2011.
[2] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Proc. Seventh Int'l Conf. World Wide Web (WWW '98),* pp. 107-117, 1998.
[3] 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," *Proc. Int'l Conf. World Wide Web (WWW '08),* pp. 895-904, 2008.
[4] D. Liben-Nowell and J. Kleinberg, "The Link Prediction Problem for Social Networks," *Proc. ACM Conf. Information and Knowledge Management (CIKM '03),* pp. 556-559, 2003.
[5] 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," *Proc. Nat'l Academy of Sciences of USA,* vol. 107, no. 10, pp. 4511-4515, Mar. 2010.
[6] N. Slonim, N. Friedman, and N. Tishby, "Unsupervised Document Classification Using Sequential Information Maximization," *Proc. 25th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval,* pp. 129-136, 2002.
[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Proc. USENIX Symp. Operating Systems Design & Implementation (OSDI '04),* p. 10, 2004.
[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *ACM SIGOPS Operating Systems Rev.,* vol. 41, pp. 59-72, Mar. 2007.
[9] 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," *Proc. USENIX Symp. Opearting Systems Design & Implementation (OSDI '08),* pp. 1-14, 2008.
[10] Hadoop, http://hadoop.apache.org/, 2013.
[11] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," *Proc. ACM SIGMOD Int'l conf. Management of Data,* pp. 1099-1110, 2008.
[12] 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," *Proc. Int'l Conf. Very Large Database (VLDB '09),* pp. 1626-1629, 2009.
[13] 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," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 135-146, 2010.
[14] Amazon ec2, http://aws.amazon.com/ec2/, 2013.
[15] H.H. Song, T.W. Cho, V. Dave, Y. Zhang, and L. Qiu, "Scalable Proximity Estimation and Link Prediction in Online Social Networks," *Proc. Int'l Conf. Internet Measurement (IMC '09),* pp. 322-335, 2009.
[16] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Imapreduce: A Distributed Computing Framework for Iterative Computation," *Proc. IEEE Int'l Workshop Data Intensive Cloud Computing (DataCloud '11),* 2011.
[17] R. Power and J. Li, "Piccolo: Building Fast, Distributed Programs with Partitioned Tables," *Proc. USENIX Symp. Opearting Systems Design and Implementation (OSDI '10),* 2010.
[18] Priter project, http://code.google.com/p/priter/, 2013.
[19] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," *Proc. USENIX Workshop Hot Topics in Cloud Computing (HotCloud '10),* 2010.
[20] 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," *Proc. Ninth USEINX Symp. Networked Systems Design and Implementation (NSDI '12),* 2012.
[21] 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," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 165-178, 2009.
[22] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce Online," *Proc. USEINX Symp. Networked Systems Design and Implementation (NSDI '10),* 2010.
[23] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for Iterative Mapreduce," *Proc. IEEE Int'l Workshop MapReduce (MapReduce '10),* pp. 810-818, 2010.
[24] Y. Bu, B. Howe, M. Balazinska, and D.M. Ernst, "Haloop: Efficient Iterative Data Processing on Large Clusters," *Proc. Int'l Conf. Very Large Database (VLDB '10),* 2010.
[25] D.G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: A Universal Execution Engine for Distributed Data-Flow Computing," *Proc. USEINX Symp. Networked Systems Design and Implementation (NSDI '11),* 2011.
[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J.M. Hellerstein, "Graphlab: A New Framework for Parallel Machine Learning," *Proc. Int'l Conf. Uncertainty in Artificial Intelligence (UAI '10),* 2010.
[27] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J.M. Hellerstein, "Distributed Graphlab: A Framework for Machine Learningand Data Mining in the Cloud," *Proc. Int'l Conf. Very Large Database (VLDB '12),* 2012.
[28] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. Math. Software,* vol. 14, no. 1, pp. 1-17, Mar. 1988.
[29] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J.D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK's User's Guide.* SIAM, 1992.
[30] D. Chazan and W. Miranker, "Chaotic Relaxation," *Linear Algebra and Its Applications,* vol. 2, no. 2, pp. 199-222, 1969.
[31] G.M. Baudet, "Asynchronous Iterative Methods for Multiprocessors," *J. ACM,* vol. 25, pp. 226-244, Apr. 1978.
[32] D.P. Bertsekas, "Distributed Asynchronous Computation of Fixed Points," *Math. Programming,* vol. 27, pp. 107-120, 1983.



Yanfeng Zhang received the BSc, MSc, and PhD degrees in computer science from Northeastern University, China, in 2005, 2008, and 2012, respectively. He had been a visiting PhD student in University of Massachusetts Amherst during August 2009 to April 2012. He is currently with the Computing Center at Northeastern University, China. His current research consists of large scale data mining, distributed systems, and cloud computing. He has published many technical papers in the above areas. His paper in ACM cloud computing 2011 was honored with "Paper of Distinction."



Qixin Gao received the PhD degree from Institute of Computer Science and Engineering, Northeastern University, Shenyang, China, in 2008. He is currently with Northeastern University at Qinhuangdao, China. His current research interests include image processing, visual perception, and massive data processing.

Lixin Gao received the PhD degree in computer science from the University of Massachusetts at Amherst in 1996. She is a professor of electrical and computer engineering at the University of Massachusetts at Amherst. Her research interests include social networks, Internet routing, network virtualization and cloud computing. Between May 1999 and January 2000, she was a visiting researcher at AT&T Research Labs and DIMACS. She was an Alfred P. Sloan fellow between 2003 and 2005 and received the National Science Foundation (NSF) CAREER Award in 1999. She won the best paper award from IEEE INFOCOM 2010, and the test-of-time award in ACM SIGMETRICS 2010. Her paper in ACM Cloud Computing 2011 was honored with "Paper of Distinction." She received the Chancellor's Award for Outstanding Accomplishment in Research and Creative Activity in 2010, and is a fellow of the IEEE.

Cuirong Wang received the PhD degree from Northeastern University, Shenyang, China, in 2003. She is currently a professor with the Computer Science Department, Northeastern University at Qinhuangdao, China. Her current research interests include data center networks, cloud computing, and wireless sensor networks. She has been a main researcher in several National Nature Science Foundation research projects of China. She is an advanced member of China Computer Federation.