J Grid Computing (2012) 10:47–68
DOI 10.1007/s10723-012-9204-9

iMapReduce: A Distributed Computing Framework for Iterative Computation

Yanfeng Zhang · Qixin Gao · Lixin Gao · Cuirong Wang

Received: 15 July 2011 / Accepted: 1 March 2012 / Published online: 25 March 2012
© Springer Science+Business Media B.V. 2012

Abstract Iterative computation is pervasive in many applications such as data mining, web ranking, graph analysis, online social network analysis, and so on. These iterative applications typically involve massive data sets containing millions or billions of data records. This poses demand of distributed computing frameworks for processing massive data sets on a cluster of machines. MapReduce is an example of such a framework. However, MapReduce lacks built-in support for iterative process that requires to parse data sets iteratively. Besides specifying MapReduce jobs, users have to write a driver program that submits a series of jobs and performs convergence testing at the client. This paper presents iMapReduce, a distributed framework that supports iterative processing. iMapReduce allows users to specify the iterative computation with the separated map and reduce functions, and provides the support of automatic iterative processing within a single job. More importantly, iMapReduce significantly improves the performance of iterative implementations by (1) reducing the overhead of creating new MapReduce jobs repeatedly, (2) eliminating the shuffling of static data, and (3) allowing asynchronous execution of map tasks. We implement an iMapReduce prototype based on Apache Hadoop, and show that iMapReduce can achieve up to 5 times speedup over Hadoop for implementing iterative algorithms.

Keywords Iterative computation · iMapReduce · Distributed computing framework · Hadoop

Y. Zhang (✉)
School of Information Science and Engineering,
Northeastern University, 11 Wenhua Road,
Shenyang, Liaoning 110819, China
e-mail: yanfengzhang@ecs.umass.edu

Q. Gao · C. Wang
Department of Electrical and Information Engineering,
Northeastern University at Qinhuangdao,
143 Taishan Road, Qinhuangdao,
Hebei 066000, China

Q. Gao
e-mail: gaoqx@neuq.edu.cn

C. Wang
e-mail: wangcr@neuq.edu.cn

L. Gao · Y. Zhang
Department of Electrical and Computer Engineering,
University of Massachusetts Amherst,
151 Holdsworth Way,
Amherst, MA 01002, USA

L. Gao
e-mail: lgao@ecs.umass.edu

1 Introduction

With the success of Web 2.0 and the popularity of online social networks, massive amounts of data are being collected, such as Facebook activities, Flickr photos, Web pages, eBay sales records, and cell phone records. The collected data typically contain millions or billions of records. For the purpose of processing large-scale data sets, Google, Microsoft and Yahoo! have built their own data centers [11]. Additionally, Amazon and Microsoft provide commodity hardware for public usage [1, 28]. Users pay to lease a number of virtual server instances to customize their own clusters.

To analyze the massive data sets, a distributed computing framework is needed on top of a cluster of servers. MapReduce [12] is a framework proposed for data intensive computation in a large cluster environment. Since its introduction, MapReduce, in particular its opensource implementation, Hadoop [15], has become extremely popular for analyzing large data sets. It provides a simple programming model and takes care of distributed execution, distributed storage, and fault tolerance, which enable programmers with no experience on distributed systems to exploit a large cluster of commodity machines to perform data intensive computation.

MapReduce is designed for batch-oriented computation such as log analysis and text processing. However, many data analysis applications [2, 3, 7, 10, 25, 32, 35] require iterative processing of the data, which includes algorithms for text-based search and machine learning. For example, the well-known PageRank algorithm parses the web linkage graph iteratively for deriving pages' ranking scores. The huge amount of data presented in these applications demands for an efficient distributed framework to implement these iterative algorithms. Nevertheless, MapReduce lacks the built-in support for iterative processing.

Further, implementing iterative algorithms in MapReduce usually requires users to design a chain of jobs, which poses several performance penalties. First, it wastes considerable resources on launching, scheduling, and cleaning up these jobs, and as result leads to longer processing time, even though these jobs perform the same operations. Second, the data used for iterative computation are loaded and shuffled repeatedly in each iteration job, even though they are not changed during iterations. Third, the synchronous batched executions of these MapReduce jobs require finishing the previous iteration job before starting the next iteration job, which can unnecessarily delay the process.

In this paper, we propose *iMapReduce*. To address the inefficiencies of iterative algorithm implementations in MapReduce, iMapReduce provides an efficient iterative processing framework while reserving the similar MapReduce programming interfaces at the same time. First, it proposes the concept of persistent tasks to avoid repeated task scheduling. Second, the input data are loaded to local file system only once instead of shuffled among workers many times, which can significantly reduce I/O and network communication overhead. Third, it facilitates asynchronous execution of map tasks within the same iteration to break synchronization barrier between MapReduce jobs.

We implement a prototype of iMapReduce based on Apache Hadoop [15] and Hadoop Online Prototype (HOP) [16]. Our prototype is backward compatible to Hadoop MapReduce in the sense that it supports any Hadoop MapReduce job. We evaluate our prototype with several well-known iterative algorithms, including Shortest path, PageRank and K-means. Our experimental results show that iMapReduce can accelerate the iterative process significantly, which is up to 5 times faster than the Hadoop implementations.

The rest of the paper is organized as follows. In Section 2, we introduce two typical iterative algorithms' MapReduce implementations and summarize their limitations. Section 3 describes the design and implementation of iMapReduce. Evaluation results are provided in Section 4. Section 5 generalizes iMapReduce to any iterative algorithm. We review the related work in Section 6 and conclude the paper in Section 7.

2 Iterative Algorithms

Many data mining algorithms have an iterative process to operate data recursively. In this section, we first provide two examples of iterative algorithms, and describe their MapReduce implementations. Then we summarize the limitations of implementing these algorithms in MapReduce.

2.1 Iterative Algorithm Examples

We describe Single Source Shortest Path (SSSP) and PageRank along with their MapReduce implementations in this section.

2.1.1 Single Source Shortest Path

Single Source Shortest Path (SSSP) [10] is a classical problem that derives the shortest distance from a source node s to all other nodes in a graph. Formally, we describe the shortest path computation as follows. Given a weighted, directed graph $G = (V, E)$, with link weight matrix W mapping edges to real-valued weights, for a source node s, find the minimum distance to any other vertex v, $d(v)$.

To perform the shortest path computation in the MapReduce framework, we can traverse the graph in a breadth-first manner. Starting from a source s, with the distance to the source being initialized as 0, i.e., $d(s) = 0$, and any other node's minimum distance being initially set to be ∞, the map function is applied on each node u. The map input key is a node id, and the map input value is composed of two parts. The first part is the minimum distance from s to u, i.e., $d(u)$, and the second part is the set of node u's outgoing links' weight values, i.e., $W(u, *)$. Based on these two types of input, the map function generates the output key-value pair $\langle v, d(u) + W(u, v)\rangle$, where v is any of u's outbound neighboring nodes and $W(u, v)$ is the link weight from u to v. At meantime, to maintain its current shortest distance and the link weight information, the mapper also produces $\langle u, [d(u), W(u, *)]\rangle$ pair.

The map output key-value pairs are shuffled to the reducers. In each reducer, a number of possible distance values for a node v from different predecessors u are gathered, and the minimum distance value is selected to update v's shortest distance only if it is shorter than v's original shortest distance. Node v's shortest distance combined with its outbound link weight information composes the reduce output value, which is written to DFS for the next iteration MapReduce job. In the next iteration, the same map/reduce operations are performed on the previous MapReduce job's output, which is the set of updated shortest distance values. For the sake of terminating the iterative process, an additional MapReduce job following each iteration job is launched to check the termination condition. Finally, the iterative process terminates when all the nodes' shortest distance values are not changed.

The map and reduce operations can be summarized as follows.

Map For each node u, based on its outgoing links' weight values $W(u, *)$, output key-value pairs $\langle v, d(u) + W(u, v)\rangle$, where $W(u, v) \in W(u, *)$, and output its current shortest distance value as well as its outgoing links' weights, $\langle u, [d(u), W(u, *)]\rangle$.

Reduce For each node v, select the minimum value among $d(v)$ and $d(u) + W(u, v)$ received from any u to update $d(v)$, and output key-value pair $\langle v, [d(v), W(u, *)]\rangle$.

2.1.2 PageRank

PageRank [3, 6, 32] is a popular algorithm initially proposed for ranking web pages. Later on, it has been used in a wide range of applications, such as link prediction [23, 36] and recommendation systems [2, 17, 37].

The PageRank vector R is defined over a directed graph $G = (V, E)$. Each node v in the graph is associated with a PageRank score $R(v)$. The initial rank of each node is $\frac{1}{|V|}$. Each node v updates its rank iteratively as follows:

$$R^{(k+1)}(v) = \frac{1-d}{|V|} + \sum_{u \in N^-(v)} \frac{d \cdot R^{(k)}(u)}{|N^+(u)|}, \tag{1}$$

where $N^-(v)$ is the set of nodes pointing to node v, $N^+(v)$ is the set of nodes that v points to, k is the iteration number, and d is a constant representing the damping factor. This iterative process continues for a fixed number of iterations or till the difference between the resulting PageRank scores of two consecutive iterations is smaller than a threshold.

In MapReduce, the map function is applied on each node u, where the input key is the node id and the input value contains node u's ranking score $R(u)$ as well as node u's outbound neighbors set $N^+(u)$. The mapper on node u derives the partial ranking score of v, $v \in N^+(u)$, i.e., $d\frac{R(u)}{|N^+(u)|}$, that will be shuffled to node v. Meanwhile, the retained PageRank score $\frac{1-d}{|V|}$ and the outbound neighbors set $N^+(u)$ are shuffled to itself. The reducer on node v accumulates these partial ranking scores and the retained ranking score to produce a new ranking score of v. The updated ranking score of v along with the outbound neighbors set is written to DFS for feeding the next iteration MapReduce job. To stop the iterative process, users have to perform another MapReduce job after each iteration to measure the difference from the last iteration's result.

The map and reduce operations can be summarized as follows.

Map For each node u, output key-value pairs $\langle v, d\frac{R(u)}{|N^+(u)|}\rangle$, where $v \in N^+(u)$, and output the retained ranking score and its outbound neighbors set, $\langle u, [\frac{1-d}{|V|}, N^+(u)]\rangle$.

Reduce For each node v, sum $\frac{1-d}{|V|}$ and $d\frac{R(u)}{|N^+(u)|}$ received from any u to update $R(v)$, and output key-value pair $\langle v, [R(v), N^+(v)]\rangle$.

2.2 Limitations of MapReduce Implementation

As described above, MapReduce can be used to implement iterative algorithms. However, we list three limitations for implementing iterative algorithms in MapReduce.

1. The operations in each iteration are the same. Nevertheless, MapReduce implementation starts a new job for each iteration, which involves repeated task initializations and cleanups. Moreover, these jobs have to load the input data from DFS and dump the output data to DFS repeatedly. These result in the unnecessary **scheduling overhead**.
2. The adjacency information data is shuffled in each iteration between map and reduce, despite the fact that it remains the same during all iterations. An alternative approach is to run an additional MapReduce job to perform a join operation between the static adjacency data and the dynamic iterated data before each iteration. Both of these approaches result in the unnecessary **communication overhead**.
3. The map tasks in an iteration cannot be started before finishing all the reduce tasks in the previous iteration. The main loop in the MapReduce implementation requires the completion of previous iteration job before starting the next iteration job. However, the map tasks should be started as soon as their input data are available. This limitation results in the unnecessary **synchronization overhead**.

iMapReduce aims to address these limitations and provides an efficient distributed computing framework for implementing iterative algorithms. In doing so, we have two observations about these iterative algorithms. First, the processing unit in both map function and reduce function is node, which enables a one-to-one mapping between mappers and reducers. Second, each iteration is expressed by only one MapReduce job. Although these two observations might not be true for all iterative algorithms, we note that they are indeed true for a large class of graph-based iterative algorithms. We will make these assumptions in presenting iMapReduce in Section 3 for ease of exposition. In Section 5, we will describe how iMapReduce supports implementing iterative algorithms where the above two assumptions do not hold.

3 iMapReduce

In this section we propose iMapReduce. iMapReduce provides support of iterative processing (Section 3.1), efficient data management (Section 3.2), and asynchronous map task execution (Section 3.3). In addition, we will describe the runtime support including load balancing and fault tolerance mechanisms in Section 3.4 and framework APIs in Section 3.5.

3.1 Iterative Processing

In the MapReduce implementations of iterative algorithms, a series of MapReduce jobs consisting of map operation and reduce operation are scheduled. Figure 1a shows the dataflow in the MapReduce implementation. Each MapReduce job has to load the input data from DFS before the map operation. After the map operation derives the intermediate key-value pairs, the reduce function operates on the intermediate data and derives the output of this iteration, which is written to DFS. In the next iteration, the map function loads the iterated data from DFS again and repeats the process. These MapReduce jobs including their component map/reduce tasks incur unnecessary scheduling overhead. Additionally, the repeated DFS loading/dumping is expensive, even though Hadoop provides locality optimization that reduces the remote communication.

We note that each iteration performs the same operations. In other words, the series of jobs perform the same map and reduce functions. We exploit this property in iMapReduce by making map/reduce tasks *persistent*. That is, the map/reduce operations in map/reduce tasks are kept executing till the iteration is terminated. Further, iMapReduce enables the reduce's output to be passed to the map for the next round iteration.

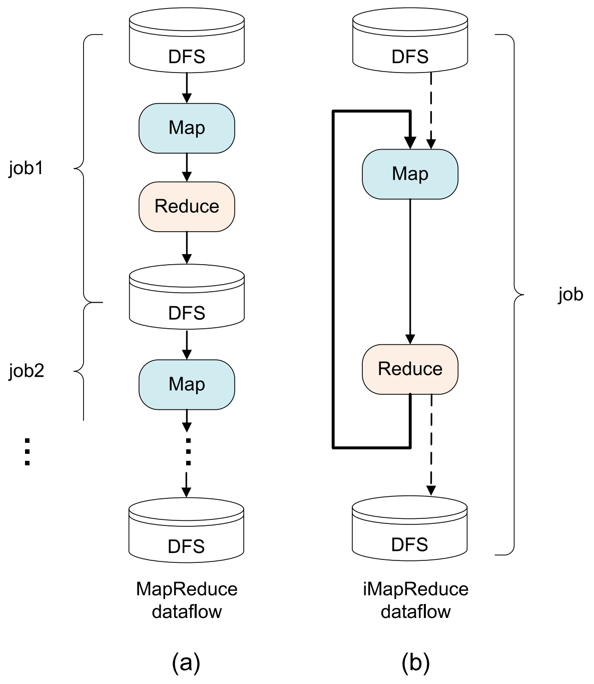

Fig. 1 **a** Dataflow of MapReduce. **b** Dataflow of iMapReduce

Figure 1b shows the dataflow in iMapReduce. The dashed line indicates that the data loading from DFS happens only once in the initialization stage, and the output data are written to DFS only once when the iteration terminates. In the following we describe how iMapReduce implements the persistent tasks and how the persistent tasks are terminated.

3.1.1 Persistent Tasks

A map/reduce task is a computing process with a specified map/reduce operation corresponding to a subset of data records. In the Hadoop MapReduce framework, each map/reduce task is assigned to a slave worker processing a subset of the input/shuffled data, and its life cycle ends when completing processing the assigned data records.

In contrast, each map/reduce task in iMapReduce is persistent. A persistent map/reduce task keeps alive during the entire iterative process. When all the assigned data of a persistent task are parsed, the task becomes dormant, waiting for the new input/shuffled data. For a map task, it waits for the results from the reduce task, and is reactivated to work on the new input data records when the required data arrive from the reduce task. We will describe how the data are passed from the reduce tasks to the map tasks in Section 3.2.1. For the reduce task, it waits for all the map tasks' output and is reactivated synchronously as in MapReduce.

To implement persistent tasks, there should be enough *available task slots*. The number of available map/reduce task slots is the number of map/reduce tasks that the framework can accommodate (or allows to be executed) simultaneously. In Hadoop MapReduce, the master splits a job into many small map/reduce tasks, the number of map/reduce tasks executed simultaneously cannot be larger than the number of the available map/reduce task slots (the default number in Hadoop is two per slave worker). Once a slave worker completes an assigned task, it requests another one from the master. In iMapReduce, we need to guarantee that there are sufficient available task slots for all the persistent tasks to start at the beginning. This means that the task granularity should be set coarser to have less tasks. Clearly, this might make load balancing challenging. We will address this issue with a load balancing scheme in Section 3.4.2.

3.1.2 Iteration Termination

Iterative algorithms typically stop when a termination condition is met. Users terminate an iterative process in two ways: (1) Fixed number of iterations: Iterative algorithm stops after a fixed number of iterations. (2) Bounding the distance between two consecutive iterations: Iterative algorithm stops when the difference between two consecutive iterations is less than a threshold.

iMapReduce performs the termination check after each iteration. To terminate the iterative computation by a fixed number of iterations, it is straightforward that we record the iteration number in each task and terminate it when the number exceeds a threshold. To bound the distance between two consecutive iterations, the reduce tasks save the output from two consecutive iterations and calculate the distance. Users should specify the distance measurement method, e.g., Euclidean distance, Manhattan distance, etc. In order to obtain an global distance value from all reduce tasks, the local distance values from the reduce tasks are merged by the master, and the master checks the termination condition to decide whether to terminate or not. If the termination condition is satisfied, the master will notify all the persistent map/reduce tasks to terminate their executions.

3.2 Data Management

As described in Section 2.2, even though the graph adjacency information is unchanged from iteration to iteration, the MapReduce implementations reload and reshuffle the unchanged graph data in each iteration, which poses considerable communication overhead.

To avoid the shuffling of unchanged graph data, iMapReduce differentiates the *static data* from the *state data*. The state data are updated in each iteration, while the static data remain the same across iterations. For example, in the SSSP example, the nodes' shortest distance values as the state data are updated at each iteration, while the link weights as the static data are unchanged. In PageRank, the pages' ranking scores as the state data are updated iteratively, while the adjacency lists as the static data are unchanged.

Figure 2 shows the state/static data flow in an iMapReduce worker. Within a worker, the initial state data and the static data are loaded to local FS from DFS in the initial stage. Before each mapreduce iteration starts, the iterated state data are joined with the local static data for map operation. Then the state data produced by map are shuffled to reduce, and the updated state data from reduce are passed to map to start another iteration. In the following, we describe how the state data are passed from reduce to map and how to join the state data with the static data.

3.2.1 Passing State Data from Reduce to Map

In MapReduce, the output of a reduce task is written to DFS and might be used later in the next MapReduce job. In contrast, iMapReduce allows the state data to be passed from the reduce task to the map task directly, so as to trigger the join operation with the static data and to start the map execution for the next iteration. To do so, iMapReduce builds persistent socket connections from the reduce tasks to the map tasks.

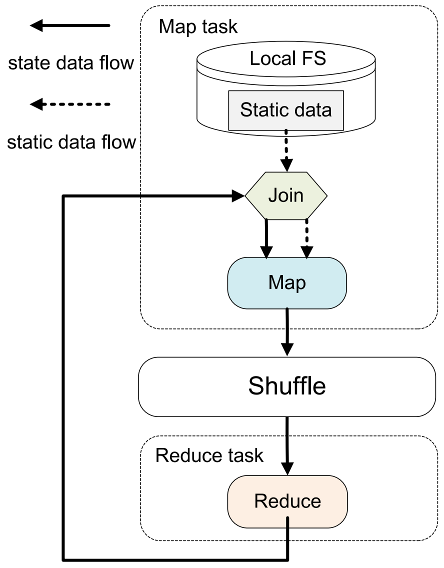

Fig. 2 Dataflow of the state data and the static data in an iMapReduce worker

In order to reduce the number of the socket connections, we partition the set of graph nodes into multiple subsets. Each map task accompanied with a reduce task is assigned with a subset of nodes. In other words, there is a one-to-one correspondence between the map tasks and the reduce tasks. Only one socket connection is needed for passing the state data from a reduce task to the corresponding map task, which processes the same subset of nodes. Otherwise, we have to perform another shuffling process from the reduce tasks to the map tasks, where the number of persistent connections grows exponentially as the number of tasks increases.

Since the map task has a one-to-one correspondence with the reduce task, they hold the same subset of nodes. We can partition the static data using the same partition function as that is used in the shuffling of the state data. In doing so, the state data are always shuffled to the exact reduce task where its connected map task is holding the corresponding static data. Further, in order to avoid the network communication needed for passing the state data, we prefer a local connection, so that the task scheduler maintains the mapping information and always assigns the map task and its corresponding reduce task to the same worker.

3.2.2 Joining State Data with Static Data

As shown in Fig. 2, iMapReduce separates the static data from the state data, so that we only update the state data iteratively. The static data are located in map tasks and are joined with the iterated state data for map operation. Therefore, the map operation takes input from both the state data and the static data, while the reduce operation only takes input from the state data. Before executing the map operation, a join operation between the state data and the static data is performed to obtain the combined state and static data records, which are used in the map operation.

iMapReduce automatically performs this join operation without requiring users to write an additional MapReduce job (Some previous work has focused on optimizing this additional joining MapReduce job [5]). We assign the same key to the static data record and the state data record that will be joined together. For the PageRank example, a node's static adjacency list and its iteratively updated ranking score are indexed by the same node id. The static data records and the state data records are sorted in the natural order of their keys respectively. In order to implement the join operation, we read one static data record and read one state data record correspondingly, so they are with the same key. The framework will join the static data and the state data before feeding the joined data to the map operation. The joined state data record and static data record are provided to map operation as the input parameters, so that users can concentrate on implementing the map operation, without worrying about the maintenance of the static data in their iterative algorithm implementations.

3.3 Asynchronous Execution of Map Tasks

In MapReduce iterative algorithm implementations, two synchronization barriers are existed, between maps-reduces and between MapReduce jobs, respectively. Due to the synchronization barrier between jobs, the map tasks of the next iteration job cannot start before the completion of the previous iteration job, which requires all the reduce tasks' completion. However, since the map task needs only the state data from its corresponding reduce task, a map task can start its execution as soon as its input state data arrives, without waiting for the other reduce tasks' completion. In iMapReduce, we schedule the execution of map tasks asynchronously. By enabling the asynchronous execution, the synchronization barriers between MapReduce jobs are eliminated, which can further speed up the iterative process.

To implement the asynchronous execution, we build a persistent socket connection from a reduce task to its corresponding map task. In a naive implementation, as soon as the reduce task produces a data record, it is immediately sent back to its corresponding map task. Upon receipt of the data from the reduce task, the map task performs the map operation on it immediately. However, the eager triggering in the native implementation will result in frequent context switches between reduce operation and map operation that impacts performance. Thus, a buffer is designed in each reduce task. As the buffer size grows larger than a threshold, the data are sent to the corresponding map task.

In each map task, the join operation is performed as soon as the state data from the corresponding reduce task arrive. Since the static data are always ready in the map task, the join operation can be performed in an eager fashion. The map operation is executed immediately after the join operation produces the joined static and state data record. Note that the reduce tasks cannot start until all the map tasks in the same iteration have been completed. In other words, the execution of map tasks and the execution of reduce tasks cannot be overlapped in the same iteration, as is the case in MapReduce.

3.4 Runtime Support

The runtime support for load balancing and fault tolerance is essential for a distributed computing framework. As we know, one of the key reasons for MapReduce's success is its simple and efficient runtime support. In the following, we describe how iMapReduce supports load balancing and fault tolerance.

3.4.1 Fault Tolerance

Fault Tolerance is important in a server cluster environment. MapReduce splits a job into multiple fine-grained tasks and reschedule the failed task whenever a task failure is detected. Moreover, MapReduce provides speculative execution [40] that is designed on clusters of heterogeneous hardware. Speculative execution starts another concurrent task to process the same data block if extra resources are available, where the first completed task's output is preferred.

iMapReduce relies on checkpointing mechanism for fault-tolerance. For each map task, the static data block has a replica on DFS, while for each reduce task, the output state data as the checkpoint are dumped to DFS every few iterations. In case there is a failure, iMapReduce recovers from the most recent checkpoint iteration, instead of starting the iterative process from scratch. Since the state data are relatively small, it is expected to consume little time for dumping these data to DFS (i.e., make several file copies on several other machines for data redundancy). Note that the checkpointing process is performed in parallel with the iterative process.

3.4.2 Load Balancing

In MapReduce, the master decomposes a submitted job into multiple tasks. The slave worker completes one task followed by requesting another one from the master. This "complete-and-then-feed" task scheduling mechanism makes good use of computing resources. In iMapReduce, all tasks are assigned to slave workers in the beginning at one time, since tasks are persistent in iMapReduce. This one-time assignment conflicts with MapReduce's task scheduling strategy, so that we cannot confer the benefit from the original MapReduce framework.

Lack of load balancing support may lead to several problems: (1) Even though the initial input data are evenly partitioned among workers, it does not necessarily mean that the computation workload is evenly distributed due to the skewed degree distribution. (2) Even though the computation workload is evenly distributed among workers, it still cannot guarantee the best utilization of computing resources, since a large cluster might consist of heterogeneous servers [40].

To address these problems, iMapReduce performs *task migration* whenever the workload is unbalanced among workers. After each iteration, each reduce task sends an iteration completion report to the master, which contains the reduce task id, the iteration number, and the processing time for that iteration. Upon receipt of all the reduce tasks' reports, the master calculates the average processing time for that iteration excluding longest and shortest, based on which the master calculates the time deviation of each worker and identifies the slower workers and the faster workers. If the time deviation is larger than a threshold, the reduce task along with its connected map task in the slowest worker is migrated to the fastest worker in the following three steps. The master (1) kills a map-reduce task pair in the slow worker, (2) launches a new map-reduce task pair in the fast worker, and (3) sends a rollback command to the other map/reduce tasks. All the map/reduce tasks that receive the rollback command skip their current work. The rolled back map tasks reload the latest checkpointed state data from DFS and proceed the iterative computation, and the new launched map tasks load not only the state data but also the corresponding static data from DFS.

However, we notice that 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 break it into several small sub-partitions distributed to different workers.

3.5 APIs

According to the design ideas, we implement a prototype of iMapReduce [18] based on Hadoop [15] and Hadoop Online Prototype (HOP) [16]. Our prototype supports any Hadoop job. In addition, it supports the implementation of iterative algorithms. Users can turn on iterative processing functionalities for implementing iterative algorithms, or turn them off for implementing MapReduce jobs as usual.

To implement an iterative algorithm in iMapReduce, users should implement the following interfaces:

– `void map(Key, StateValue, StaticValue).`
  The map interface in iMapReduce has one input key `Key` and two input values: state data value `StateValue` and static data value `StaticValue`. Both of the values are associated with the same input key. iMapReduce framework joins the state data and the static data automatically, so that users can focus on describing the map computation.
– `void reduce(Key, StateValue).`
  The reduce interface in iMapReduce is the same as that in MapReduce, with an input key and an input value. Note that the input value is state data that has been separated from static data.
– `float distance(Key, PrevState, CurrState).`
  Users implement this interface to specify the distance measurement using a key's previous state value and its current state value. The returned float values for different keys are accumulated to obtain the distance value between two consecutive iterations' results. For example, Manhattan distance and Euclidean distance can be used to quantify the difference.

In addition, iMapReduce provides the following job parameters (i.e., JobConf's parameters) to help users specify iterative computation:

– `job.set("mapred.iterjob.statepath", path).`
  Set the DFS path of the initial state data.
– `job.set("mapred.iterjob.staticpath", path).`
  Set the DFS path of the static data.
– `job.setInt("mapred.iterjob.maxiter",` n`).`
  Set the maximum iteration number n to terminate an iterative computation.
– `job.setFloat("mapred.iterjob.disthresh",` ϵ`).`
  Set the distance threshold as ϵ to terminate an iterative computation, which is used together with `distance` interface.

To show how to implement iterative algorithms in iMapReduce, an example of PageRank algorithm implementation code is given in Fig. 3. In the map function (Line 1–4), each page's PageRank score is evenly distributed to its neighbors and retaining $\frac{(1-d)}{N}$ by itself, where N is the total number of pages in the graph. In the reduce function (Line 5), for each page, it combines the partial PageRank scores from its incoming neighbors and its own retained score. For the distance measurement, we calculate the Manhattan distance (Line 6). Additionally, we should specify the location of the initial state data (Line 11), as well as the location of the static data (Line 12). iMapReduce 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. By using the distance-based

```
Map
Input: Key n, StateValue R(n), StaticValue adj(n)
 1: for link in adj(n) do
 2:     output( link.endnode, (d × R(n)) / |adj(n)| );
 3: end for
 4: output(n, (1-d)/N);

Reduce
Input: Key n, Set <values>
 5: output(n, sum(<values>) );

Distance
Input: Key n, PrevState R1(n), CurrState R2(n)
 6: return abs(R1(n) -R2(n) );

Main
 7:  Job job = new Job();
 8:  job.setMap(Map);
 9:  job.setReduce(Reduce);
10:  job.setDistance(Distance);
11:  job.set("mapred.iterjob.statepath", "hdfs://…/initRankings");
12:  job.set("mapred.iterjob.staticpath", "hdfs://…/adjacencylists");
13:  job.setFloat("mapred.iterjob.disthresh", 0.01);
14:  job.submit();
```

Fig. 3 PageRank implementation in iMapReduce

termination check method, the iterative computation will terminate when the distance measured by the distance function is smaller than 0.01 (line 13).

4 Evaluation

In this section, we evaluate iMapReduce. Two typical graph based iterative algorithms are considered: SSSP and PageRank. We compare the performance of the two algorithms implemented in iMapReduce with that implemented in Hadoop [15].

4.1 Experiment Setups

4.1.1 Cluster Environment

Our experiments are performed on both a local cluster of commodity machines and a cluster built on Amazon EC2 cloud [1]. Hadoop's block size is set to be 64 MB, and Hadoop's heap size is set to be 2 GB. We describe the cluster settings as follows.

Local Cluster A local cluster containing 4 nodes is used to run experiments. Each node has Intel (R) Core(TM)2 Duo E8200 dual-core 2.66 GHz CPU, 3 GB of RAM, 160 GB storage, and runs 32-bit Linux Debian 4.0 OS. These four nodes are locally connected by a switch with communication bandwidth of 1 Gbps.

Amazon EC2 cluster We build a test cluster on Amazon EC2. There are 80 small instances involved in our experiments. Each instance has 1.7 GB memory, Inter Xeon CPU E5430 2.66 GHz, 146.77 GB instance storage and runs 32-bit platform Linux Debian 4.0 OS.

4.1.2 Data Sets

We implement SSSP and PageRank under iMapReduce and evaluate the performance on real graphs and synthetic graphs. We generate the synthetic graphs in order to evaluate iMapReduce on graphs of different sizes.

We first describe the graphs used for running SSSP algorithm as follows. (1) DBLP author cooperation graph. Each node in DBLP graph represents an author, and each link between two nodes represents the cooperation relationship between the two authors. The link weight is set according to the cooperation frequency between the two linked authors. (2) Facebook user interactions graph [38]. Each node in the graph represents a Facebook user, and each link between two nodes implies that the two users are friends. The users interaction frequency is used to assign weight to the user friendship links. (3) Synthetic graph. The log-normal parameters of the link weight distribution (i.e., shape parameter $\sigma = 1.2$, scale parameter $\mu = 0.4$) and that of the node out-degree distribution (i.e., shape parameter $\sigma = 1.0$, scale parameter $\mu = 1.5$) are extracted by average from the above two real graphs [8]. Based on these log-normal parameters, we then generate three log-normal synthetic graphs, containing 1 million, 10 million, and 50 million nodes, respectively. Table 1 shows the brief description of these data sets.

The data set used for PageRank is described as follows. Two real graphs are considered: (1) Google webgraph [22] and (2) Berkley-Stanford webgraph [22]. Unlike SSSP's weighted graphs, these graphs are unweighted. (3) The log-normal parameters (i.e., shape parameter $\sigma =$

Table 1 SSSP data sets statistics

Graph	# of nodes	# of edges	File size
DBLP	310,556	1,518,617	16 MB
Facebook	1,204,004	5,430,303	58 MB
SSPP-s	1M	7,868,140	87 MB
SSPP-m	10M	78,873,968	958 MB
SSPP-l	50M	369,455,293	5.19 GB

2, scale parameter $\mu = -0.5$) of the node's out-degree distribution are extracted from the above real graphs to generate three synthetic graphs, containing 1 million, 10 million, and 30 million nodes, respectively. Table 2 shows the brief description of these data sets.

4.2 Local Cluster Experiments

On our local cluster, we use real graphs as input to evaluate SSSP and PageRank under both MapReduce and iMapReduce. As discussed in Section 3, there are three factors that help reduce running time. (1) With the help of iterative processing support, iMapReduce performs *one-time initialization* rather than spending time on multiple job/task initializations in each iteration in Hadoop. (2) By maintaining static data locally, iMapReduce *avoids static data shuffling*. (3) By *asynchronous map execution*, iMapReduce eliminates execution delay. Besides measuring the running time of MapReduce and iMapReduce, we measure how much these factors help improve performance.

In order to investigate how much these three factors affect on performance improvement, we measure each factor's contribution by the following steps. (1) We first record MapReduce job's running time and iMapReduce job's running time as two extreme reference points. The time difference between the two indicates the reduced time achieved by iMapReduce. (2) By using iMapReduce, we let the map tasks execute synchronously and record the running time, so that the running time difference from iMapReduce indicates the contribution from the asynchronous map execution factor. (3) In order to measure the job/task initialization time consumed in the Hadoop implementations, we record the initialization time, which is the time period from the job submission to the averaged time point when these map tasks start performing map operations. The time spent on job/task initialization in Hadoop implementations is the summation of the initialization time from all iteration jobs. iMapReduce has one-time initialization that only happens in the initialization stage. The estimated initialization time of multiple Hadoop jobs excluding that of the iMapReduce job is the time saved by the factor of one-time initialization. (4) The contribution from avoiding static data shuffling can be estimated by subtracting the saved time measured by (2) and (3) from the referenced Hadoop running time.

Figure 4 shows the running time of SSSP on DBLP author cooperation graph. Figure 5 shows that on Facebook user interaction graph. We can see that by iMapReduce it can achieve a factor of 2–3 speedup over the Hadoop implementation. The plot labeled with MapReduce (ex. init.) shows the MapReduce running time excluding the initialization time in each job, and the plot labeled with iMapReduce (sync.) shows the synchronous iMapReduce running time, where we let the map tasks execute synchronously. We can see that by means of one-time initialization we can reduce the

Table 2 PageRank data sets statistics

Graph	# of nodes	# of edges	File size
Google	916,417	6,078,254	49 MB
Berk-Stan	685,230	7,600,595	57 MB
PageRank-s	1M	7,425,360	61 MB
PageRank-m	10M	75,061,501	690 MB
PageRank-l	30M	224,493,620	2.26 GB

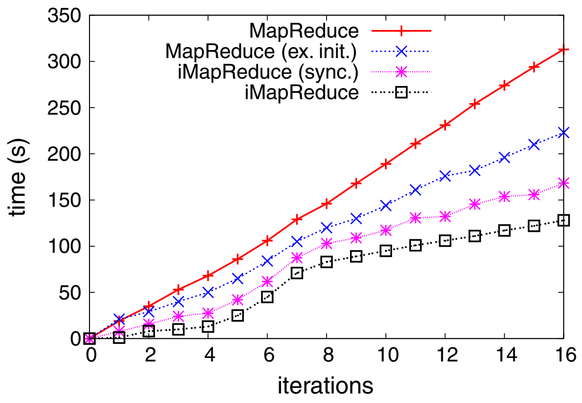

Fig. 4 The running time of SSSP on DBLP author cooperation graph

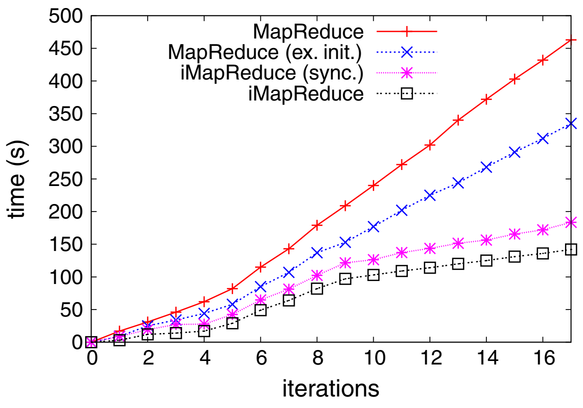

Fig. 5 The running time of SSSP on Facebook user interaction graph

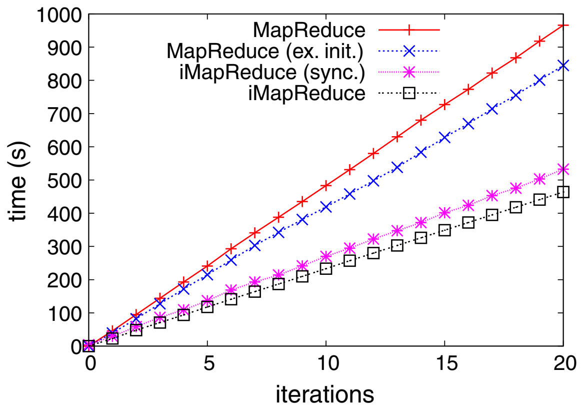

Fig. 7 The running time of PageRank on Berkley-Stanford webgraph

running time by about 20%, which is illustrated by the plot labeled with MapReduce (ex. init.). Further, another 15% running time is saved by asynchronous map execution, which is illustrated by the plot labeled with iMapReduce (sync.). Then, the remaining improvement is achieved by avoiding static data shuffling, which saves about 20% running time.

Figure 6 shows the running time of PageRank on Google webgraph, and Fig. 7 shows that on Berkley-Stanford webgraph. Comparing with the Hadoop implementation, iMapReduce achieves about 2 times speedup. In addition, we can see that 10% running time is saved by one-time initialization, 30% running time is saved by avoiding static data shuffling, and another 10% running time is saved by asynchronous map execution.

4.3 EC2 Cluster Experiments

We deploy iMapReduce prototype on Amazon EC2 cluster and perform experiments on the synthetic graphs.

4.3.1 Running Time

We run SSSP on the three synthetic graphs SSSP-s, SSSP-m, and SSSP-l on Amazon EC2 cluster (20 small instances). We limit to ten iterations and compare running time on the synthetic graphs with different sizes. Figure 8 shows the result. The iMapReduce implementation reduces running time to 23.2%, 37.0% and 38.6% of that of Hadoop for SSSP-s, SSSP-m, and SSSP-l, respectively. We can see that iMapReduce performs better when the input is small. The main iterative

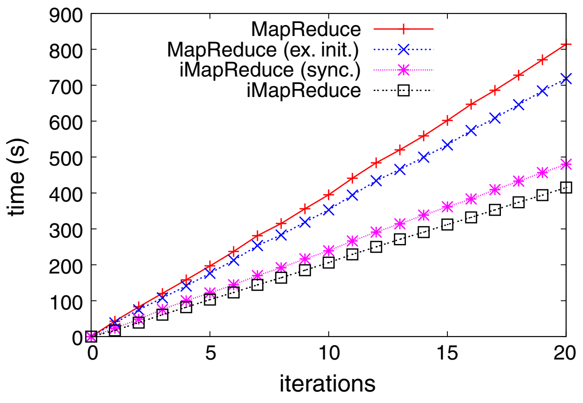

Fig. 6 The running time of PageRank on Google webgraph

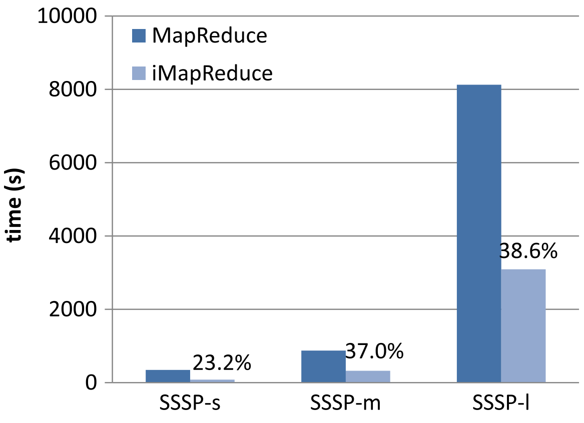

Fig. 8 The running time of SSSP on the synthetic graphs

computation takes less time on the small input, so that there is relatively more time spent on job/task initialization. While iMapReduce does not perform job/task initialization for each iteration, its one-time initialization property can reduce the total running time significantly.

Similarly, PageRank is executed with ten iterations on the three synthetic graphs PageRank-s, PageRank-m, and PageRank-l on Amazon EC2 cluster (20 small instances). The results are shown in Fig. 9. More significant speedup is achieved for the PageRank-s graph, where the running time is reduced to 44% comparing with Hadoop implementation. For the PageRank-m graph and the PageRank-l graph, iMapReduce reduces the running time to about 60%.

To explore the contributions from different factors, we show in Fig. 10 the time portion reduction by the these factors, i.e., one-time initialization, static data shuffling avoidance, and asynchronous map execution. SSSP and PageRank are both computed 10 iterations on the SSSP-m graph and on the PageRank-m graph, respectively. We can see that the running time reduced by one-time initialization and asynchronous map execution are both around 5–10%. The time reduced by avoiding static data shuffling is proportional to the input static data size (SSSP-m 958 MB and PageRank-m 690 MB). It takes more time on shuffling the bigger-size static data, so we can save relatively more time when the static data size is larger.

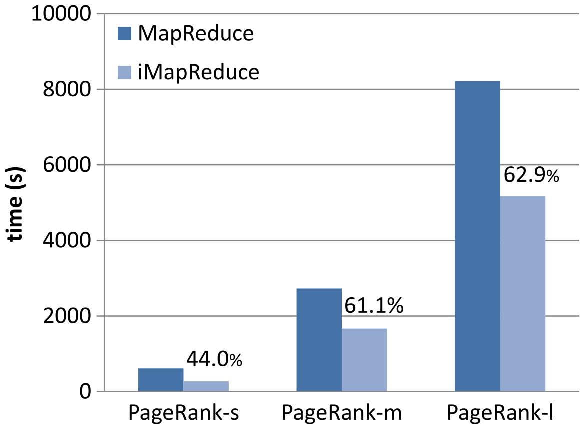

Fig. 9 The running time of PageRank on the synthetic graphs

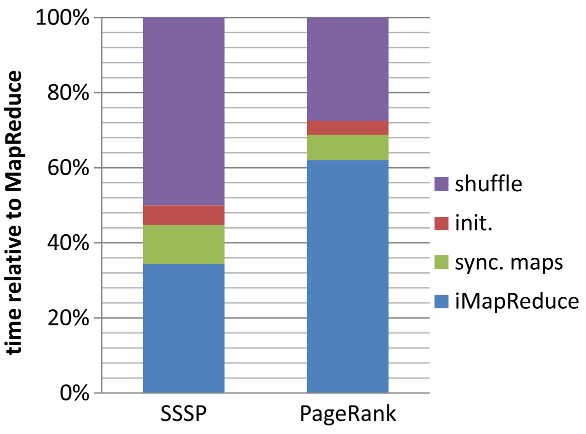

Fig. 10 Different factors' effects on running time reduction

4.3.2 Communication Cost

In Hadoop MapReduce implementations, large amounts of data are communicated between map tasks and reduce tasks. Reducing the amount of data communicated not only helps improve performance but also saves communication resources. iMapReduce saves network communication cost by avoiding static graph shuffling. To quantify the saving of communication, we measure the total amount of data exchanged when performing the iterative algorithms on the SSSP-l graph and on the PageRank-l graph. As shown in Fig. 11, iMapReduce significantly reduces the communication cost. The amount of required data exchange is reduced to only about 12%.

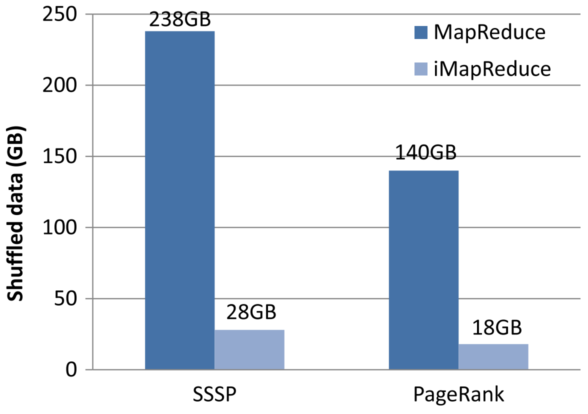

Fig. 11 Total communication cost

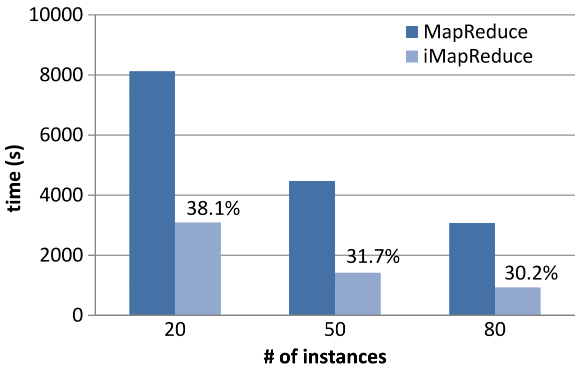

Fig. 12 The speedup over the Hadoop implementations for running SSSP when scaling cluster size from 20 to 80

4.3.3 Scaling Performance

Since our prototype is implemented on top of Hadoop MapReduce, which scales well, the scalability of iMapReduce prototype should meet most applications' needs. We scale our Amazon EC2 cluster to contain 50 instances and 80 instances, and run SSSP and PageRank on the SSSP-l graph and on the PageRank-l graph, respectively. iMapReduce accelerates algorithm convergence when using more computing resources as expected. Moreover, iMapReduce performs even better on a larger scale cluster.

Figure 12 shows the running time of SSSP when we scale the cluster size. The figure shows that the running time ratio of iMapReduce to MapReduce is reduced by 8% when we scale cluster from 20 instances to 80 instances. Figure 13 shows the running time result of PageRank when we scale the cluster size. We can see that the running time ratio of iMapReduce to MapReduce is reduced by 7% when we scale the cluster from 20 instances to 80 instances. We explain these results as follows. The bigger the cluster is, the more network communications would occur. Since iMapReduce aims at reducing network communications, it is more likely to exert its advantages on the bigger cluster.

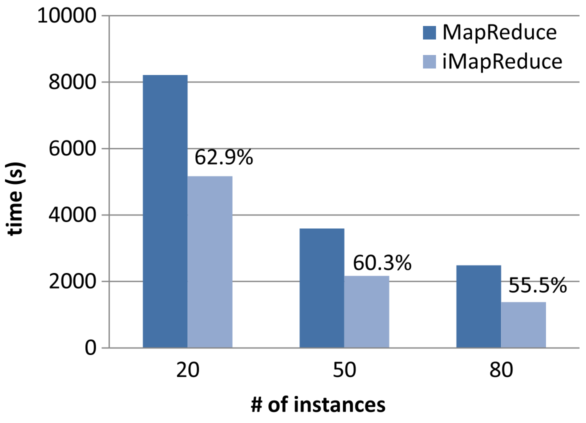

Fig. 13 The speedup over the Hadoop implementations for running PageRank when scaling cluster size from 20 to 80

4.3.4 Parallel Efficiency

We measure the parallel efficiencies of iMapReduce and MapReduce in the context of SSSP and PageRank, The *parallel efficiency* is defined as

$$\text{Parallel efficiency} = \frac{T_*}{T_n \times n}, \tag{2}$$

where T_* is the running time of an application in a single EC2 instance (no communication and no synchronization between machines), n is the number of instances, and T_n is the running time of an application in an n-instance cluster. We first run an application in a single EC2 instance to obtain T_*, where the partition number is one. Then we run the application on a cluster of machines with cluster size of 20, 50 and 80 respectively.

Figure 14 shows the parallel efficiencies for SSSP computation and PageRank computation. We can see that iMapReduce yields higher parallel efficiencies than Hadoop MapReduce in both SSSP and PageRank. For SSSP, performance slowdown could be around 60% by MapReduce when scaling cluster size to 80. While by iMapReduce, the performance slowdown could

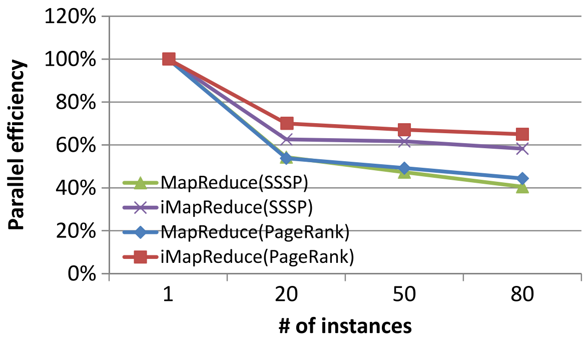

Fig. 14 Parallel efficiencies of iMapReduce and MapReduce for SSSP and PageRank

be around 43%. For PageRank, the parallel efficiency of iMapReduce is also much better than MapReduce.

5 Extensions of iMapReduce

So far, we have focused on supporting graph-based iterative algorithms in iMapReduce. However, iMapReduce can be extended to implement other iterative algorithms as well. In Section 2.2, we make two assumptions in iMapReduce: (1) there is one-to-one correspondence from reducers to mappers, and (2) each iteration is described as a single MapReduce job. In this section, we present how to extend iMapReduce to support the iterative algorithms which do not hold these two assumptions.

5.1 Accommodating Broadcast from Reduce to Map

In an iterative algorithm, it is possible that a mapper needs all reducers' output. For example, to implement the *Jacobi method* [4] in MapReduce, according to $x^{(k+1)} = D^{-1}(b - Rx^{(k)})$, each reducer calculates a part of the iterated vector, and all mappers need the intact vector x for computation, where D is a diagonal matrix. Another famous iterative algorithm, K-means clustering algorithm, also requires all reducers' output for map operation. iMapReduce accommodates broadcast from reduces to maps as shown in Fig. 15. Each reduce task can send its output key-value pairs to all map tasks. Note that, the broadcast from reduces to maps is different from the shuffle from maps to reduces. By broadcasting, each reduce task sends n copies of its output to n map tasks (each map task receives a whole output), while by shuffling, each map task splits its output into n partitions and sends the n partitions to n reduce tasks (each reduce task receives a part of the output), where n is the number of map/reduce tasks. In the following, we will present the K-means algorithm in iMapReduce to show how iMapReduce accommodates broadcasting.

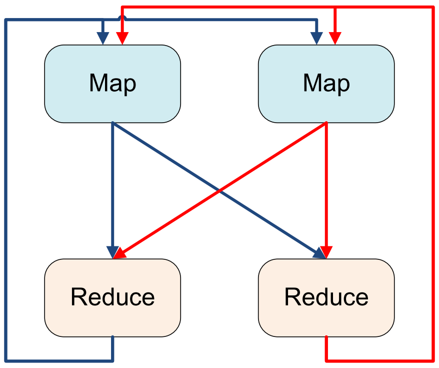

Fig. 15 Broadcast from reduces to maps

5.1.1 K-means Clustering Algorithm

K-means [25] is a commonly used clustering algorithm, which partitions n nodes into k clusters so that the nodes in the same cluster are more similar than those in other clusters. We describe the algorithm briefly as follows. (1) Start with selecting k random nodes as cluster centroids, (2) Assign each node to the nearest cluster centroid, (3) Update the k cluster centroids by "averaging" the nodes belonging to the same cluster centroid. Repeat steps (2) and (3) until convergence has been reached.

The map and reduce operations for the K-means algorithm can be described as follows.

Map For each key *nid* (node id), compute the distance from the node to any cluster centroid, and output the closest cluster id *cid* along with the node coordinate information *ncoord*, i.e., output key-value pair $\langle cid, ncoord \rangle$.

Reduce For each key *cid* (cluster id), update the cluster centroid coordinate by averaging all the nodes' coordinates that belong to *cid*, and output *cid* along with the cluster's updated centroid coordinate *ccoord*, i.e., output key-value pair $\langle cid, ccoord \rangle$.

The map function needs all the cluster centroids in order to find the closest one for a certain node. This means that the mapping from reduce to map is not one-to-one but one-to-all. The state data, i.e., cluster centroids set, should be sent from each reduce task to all map tasks. Another difference between K-means and the graph-based iterative algorithms described in Section 2 is that, not only the map operation needs the static data (i.e., the coordinates of all nodes) for measuring the distance to the centroids, but also the reduce operation needs the static data for the averaging operation. Therefore, the coordinates of nodes have to be shuffled from map to reduce.

5.1.2 iMapReduce Implementation

To support "K-means-like" iterative algorithms, iMapReduce lets reduce tasks broadcast the updated state data to all map tasks. For the one-to-all mapping, the map function operates on a static data record with multiple state data records. Accordingly, users should specify the mapping type by setting

- `job.set("mapred.iterjob.mapping", "one2all")`.

Otherwise, the framework will use `"one2one"` by default. In addition, the map function's input parameter `StateValue` (Section 3.5) that originally contains a single state value are extended to a list of state values. In the case of K-means, the state values list is the set of all cluster centroids.

Further, the map operation needs the output from multiple reduce tasks. The map operation cannot be started before all its input data arrive. In the case of K-means, the map operation cannot be started before all the updated cluster centroids are collected. That is, the map tasks cannot be executed asynchronously. Therefore, the option for synchronous map execution should be turned on by setting

- `job.setBoolean("mapred.iterjob.sync", true)`.

In general, we trigger the map operations synchronously when all the reduce operations that supply the input have completed.

5.1.3 Experimental Results

We implement K-means in iMapReduce and run the algorithm on the data set collected from Last.fm [21]. The users' listening history log is used to cluster users based on their tastes. This log contains each user's artist preference information quantified by the times the artist is listened. Sharing a preferred artist indicates a common taste. Last.fm data set (1.5 GB) has 359,347 user records, and each user has 48.9 preferred artists on average.

Figure 16 shows the K-means running time limited in ten iterations, which are performed on our local cluster. iMapReduce achieves about 1.2× speedup over Hadoop. This is less significant than that achieved for the SSSP and PageRank algorithms. Nevertheless, this is under our expectation since the implementation of K-means needs to shuffle static data and has to execute map operations synchronously.

It is also possible to reduce the shuffling of static data by using `Combiner` provided by Hadoop MapReduce. `Combiner` performs local aggregation at the map side before shuffling the intermediate data in order to reduce the amount of shuffled data. We also perform the K-means experiments with `Combiner` (other experiment configurations are the same). As expected, the running time is reduced in both Hadoop and iMapReduce. For Hadoop implementation, the running time is reduced from 2881 s to 2226 s (23% reduction), while for iMapReduce implementation, the running time is reduced from 2338 s to 1733 s (26% reduction).

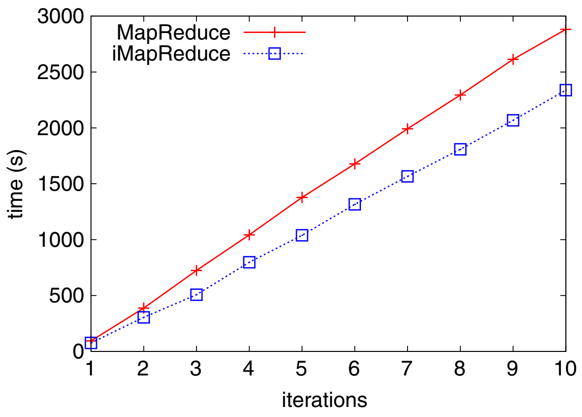

Fig. 16 Running time of K-means for clustering Last.fm data on the local cluster

5.2 Accommodating Multiple Map-Reduce Phases

In reality, it is common that a single map-reduce pass of the input is not sufficient to express algorithm logic. To solve this problem, we can perform multiple map-reduce steps in each iteration. In this section, we discuss the extensions of iMapReduce that support multiple map-reduce phases for each iteration.

Figure 17 shows an example of two-phase case. The state data are parsed by two map-reduce phases in each iteration, and the static data can be joined with the state data at two different map steps respectively. To join the state data with their corresponding static data, the key is to specify the mapping from the reducers of the previous phase to the mappers of the next phase, as well as the mapping from the reducers of the last phase to the mappers of the first phase. In the following, we give an example of matrix power computation to illustrate the multiple map-reduce phases case.

5.2.1 Matrix Power Computation

Square matrices can be multiplied by themselves repeatedly. This repeated multiplication can be described as a power of the matrix, i.e., $M^k = \prod_1^k M$. The operation of each iteration is matrix multiplication, i.e., $M^k = M \times N$, where $N = M^{k-1}$. If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product $P = MN$ is the matrix P with element p_{ik} in row i and column k, where $p_{ik} = \sum_j m_{ij} n_{jk}$.

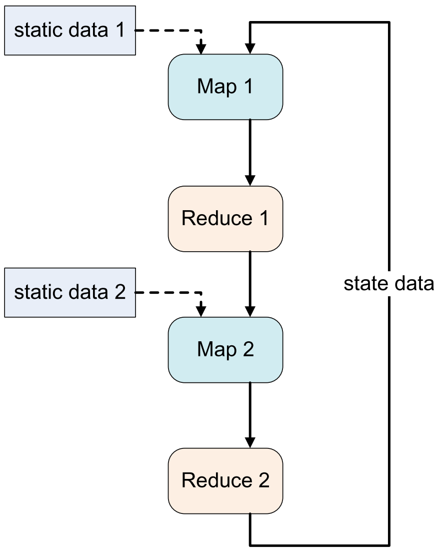

Fig. 17 Two map-reduce phases for each iteration in iMapReduce

In the MapReduce framework, we use two map-reduce phases to perform the matrix multiplication [29], which forms each iteration. In the first phase, the map extracts the columns of M and the rows of N, and the reduce joins column j of M and row j of N together. In the second phase, the map multiplies a column vector with the joined row vector to obtain an matrix, and the reduce sums these matrices to obtain the final result matrix. The map and reduce operations in each iteration are described as follows.

Map 1 For each key (i, j), send each matrix element m_{ij} of M to the key-value pair $\langle j, (M, i, m_{ij})\rangle$. For each key (j, k), send each matrix element n_{jk} of N to the key-value pair $\langle j, (N, k, n_{jk})\rangle$.

Reduce 1 For each key j, collect its list of associated values $\langle j, [(M, i_1, m_{i_1 j}), (M, i_2, m_{i_2 j}), \ldots, (N, k_1, n_{jk_1}), (N, k_2, n_{jk_2}), \ldots]\rangle$.

Map 2 Take the output key-value pair of Reduce 1. For each value that comes from M, say (M, i, m_{ij}), and each value that comes from N, say (N, k, n_{jk}), produce the key-value pair $\langle (i, k), m_{ij} n_{jk}\rangle$. It will output all the permutations $\langle (i_1, k_1), m_{i_1 j} n_{jk_1}\rangle$, $\langle (i_1, k_2), m_{i_1 j} n_{jk_2}\rangle$, ..., $\langle (i_2, k_1), m_{i_2 j} n_{jk_1}\rangle$, ...

Reduce 2 For each key (i, k), produce the sum of the list of values associated with this key. The result key-value pair is $\langle (i, k), p_{ik}\rangle$, where $p_{ik} = \sum_j m_{ij} n_{jk}$.

5.2.2 iMapReduce Implementation

In iMapReduce, we connect Reduce 1 to Map 2 such that both of them operate on the same key j, and connect Reduce 2 to Map 1 such that both of them operate on the same key (i, k). Since the connected reduce and map operate on the same type of keys, we can connect them according to one-to-one mapping. In order to avoid static data shuffling, we note that M as a static multiplicator is used in each iteration, so that we make M as the static data, which are joined with N at Map 2. While in the first map-reduce phase, there is no join operation, so that the static data are not set.

Accordingly, besides setting the first map-reduce phase by configuring a `JobConf` job1, users should set the second map-reduce phase by configuring another `JobConf` job2. These two jobs are chained together by setting

- `job1.addSuccessor(job2)`
- `job2.addSuccessor(job1).`

Otherwise, if a single map-reduce phase is sufficient to execute an iteration, the job will set the successor to be itself by default. The framework connects these map/reduce functions according to the chained order.

5.2.3 Experimental Results

We perform matrix power computation for a synthetic matrix (1000 × 1000) for five iterations in iMapReduce as well as in Hadoop, which is conducted on our local cluster. In the matrix power computation, the major portion of running time is consumed by the intermediate data shuffling between Map 2 and Reduce 2, which is ineluctable. However, as shown in Fig. 18, iMapReduce can still achieve about 10% speedup over Hadoop by various framework optimizations.

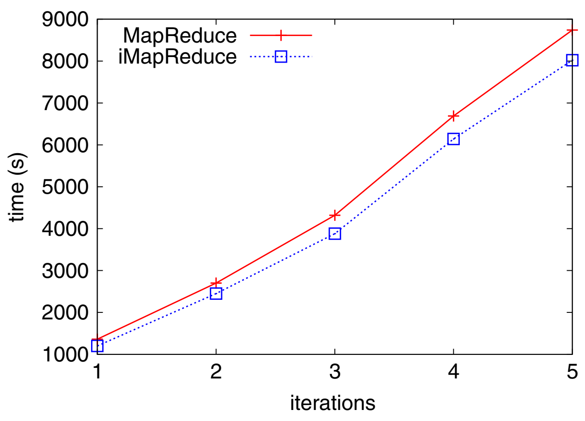

Fig. 18 Running time of matrix power computation on the local cluster

5.3 Accommodating Auxiliary Map-Reduce Phase

Some applications require running an auxiliary map-reduce phase that generates some auxiliary information. The main map-reduce phase performs iterative computation, and the auxiliary map-reduce phase also takes the main phase's output and produces some auxiliary information. Figure 19 shows a typical structure of this case. The execution of the main phase takes consideration of the result of the auxiliary phase, and the auxiliary phase generates the auxiliary information without pausing active computation in the main phase. In the following, we illustrate a common usage of the auxiliary map-reduce phase for detecting convergence of an iterative computation.

5.3.1 Convergence Detection of K-means

iMapReduce can terminate iterative computation based on the maximum iteration number or based on the difference between two consecutive

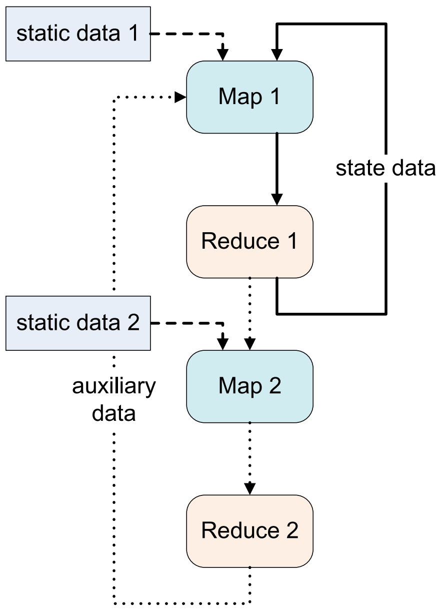

Fig. 19 Auxiliary map-reduce phase in iMapReduce

iterations' results. However, iMapReduce only provides a limited number of choices on measuring the difference (based on Manhattan distance or Euclidean distance), which is not sufficient to detect convergence for a broad class of algorithms. For example, the K-means algorithm is considered converged when the number of nodes that move between clusters is less than a threshold. The main map-reduce phase (Map 1 and Reduce 1) for computing K-means has been discussed in Section 5.1. We use another auxiliary map-reduce phase to detect convergence. The map and reduce operations in the auxiliary map-reduce phase are described as follows.

Map 2 For each cluster *cid*, compute the number of nodes that do appear in cluster *cid* in the previous iteration, say *num_stay*, and output $\langle \mathbf{0}, num\_stay \rangle$, where $\mathbf{0}$ is a unique key, so that all the mappers' outputs are shuffled to a unique reducer.

Reduce 2 In the sole reducer assigned for key $\mathbf{0}$, sum all *num_stay* from different mappers to retrieve the total number of nodes that move between clusters, which is $num\_move = |S| - num\_stay$, where $|S|$ is the total number of input nodes, and broadcast the termination signals to all mappers in Map 1 step if *num_move* is less than a threshold.

5.3.2 iMapReduce Implementation

In iMapReduce, these two map-reduce phases are running in parallel. As long as the map tasks in the main phase receive the termination signals from Reduce 2, they will terminate themselves and notify the master. To perform an auxiliary map-reduce phase in iMapReduce, users should set

- `job1.addAuxiliray(job2)`

where job1 corresponds to the main map-reduce phase, and job2 corresponds to the auxiliary map-reduce phase. In addition, the one-to-all mappings from Reduce 2 to Map 1 should be specified.

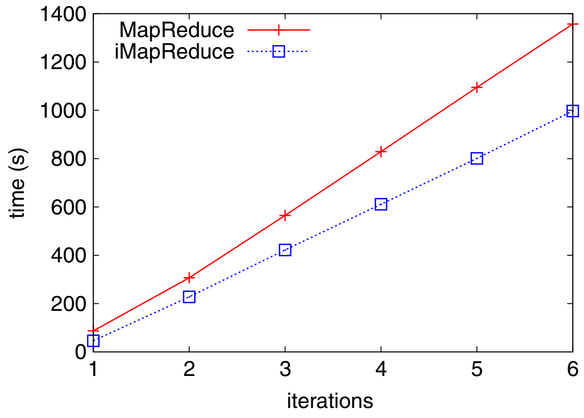

Fig. 20 Running time of K-means with convergence detection

5.3.3 Experimental Results

Based on the experiment setup mentioned in Section 5.1.3, we perform K-means with an auxiliary phase for convergence detection in iMapReduce. For comparison with MapReduce implementation, we also implement an additional Hadoop job for convergence detection, which is executed between two K-means computation jobs. While in iMapReduce, the two map-reduce phases are running in parallel (asynchronously) that could help reduce the running time. The results are shown in Fig. 20, where the algorithm terminates after six iterations. We can see that 25% running time is reduced, which is mainly resulted from the elimination of the synchronously executed auxiliary job.

6 Related Work

MapReduce, as a popular distributed framework for data intensive computation, has gained considerable attention over the past few years [12]. The framework has been extended for diverse application requirements. MapReduce Online [9] pipelines map/reduce operations and performs online aggregation to support efficient online queries, which directly inspires our work.

To support implementing large-scale iterative algorithms, there are a number of studies proposing new distributed computing frameworks for iterative processing [5, 13, 19, 20, 24, 27, 30, 33, 34, 39, 41].

A class of these efforts targets on managing static data efficiently. Design patterns for running efficient graph algorithms in MapReduce have been introduced in [24]. They partition the static graph adjacency list into n parts and prestore them on DFS. However, since the MapReduce framework arbitrarily assigns reduce tasks to workers, accessing the graph adjacency list can involve remote reads. This cannot guarantee local access to the static data. HaLoop [5] was proposed aiming at iterative processing on a large cluster. It realizes the join of the static data and the state data by explicitly specifying an additional MapReduce job, and relies on the task scheduler and caching techniques to maintain local access to static data. While iMapReduce relies on persistent tasks to manage static data and to avoid tasks initialization.

Some studies accelerate iterative algorithms by maintaining the iterated state data in memory. Spark [39] was developed to optimize iterative and interactive computation. It uses caching techniques to dramatically 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. [26] 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 [31] supports data-dependent iterative or recursive algorithms by building an abstract dynamic task graph. Piccolo [34] 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. PrIter [42] enables prioritized iteration, which exploits the dominant property of some portion of the data and schedules them first for computation, rather than blindly performs computations on all data. This is realized by maintaining a state table and a priority queue in memory. Our iMapReduce framework is built on Hadoop, the iterated state data as well as the static data are maintained in files but not in memory. Therefore, it is more scalable and more resilient to failures.

Some other efforts focus on graph-based iterative algorithms, an important class of iterative algorithms. PEGASUS [20] models those seemingly different graph iterative algorithms as a generalization of matrix-vector multiplication (GIM-V). By exploring matrix property, such as block multiplication, clustered edges and diagonal block iteration, it can achieve 5× faster performance over the regular job. Pregel [27] chooses a pure message passing model to process graphs. In each iteration, a vertex can, independently of other vertices, receive messages sent to it in the previous iteration, send messages to other vertices, modify its own and its outgoing edges' states, and mutate the graph's topology. By using this model, processing large graphs is expressive and easy to program. iMapReduce exploits the property that the map and reduce functions operate on the same type of keys, i.e., node id, to accelerate graph-based iterative algorithms.

The most relevant work is that of Ekanayake et al., who proposed Twister [13, 14], which employs stream-based MapReduce implementation that supports iterative applications. Twister employs novel ideas of loading the input data only once in the initialization stage and performing iterative map-reduce processing by long running map/reduce daemons. iMapReduce differs from Twister mainly on that Twister stores intermediate data in memory, while iMapReduce stores intermediate data in files. Twister loads all data in distributed memory for fast data access and grounds on the assumption that datasets and intermediate data can fit into the distributed memory of the computing infrastructure. iMapReduce aims at providing a MapReduce based iterative computing framework running on a cluster of commodity machines where each node has limited memory resources. In iMapReduce, the intermediate data, including the intermediate results, the shuffled data between map and reduce, and the static data, are all stored in files. This key difference results in different implementation mechanisms, including different data transfers and different joining techniques of static data and state data. Furthermore, iMapReduce supports asynchronous map execution, which further improves performance. Besides, iMapReduce is implemented based on Hadoop MapReduce. The iterative applications in Hadoop can be easily modified to run on iMapReduce.

7 Conclusions

In this paper, we propose iMapReduce that supports the implementation of iterative algorithms under a large cluster environment. iMapReduce extracts the common features of iterative algorithms and provides the built-in support for these features. In particular, it proposes the concept of persistent tasks to reduce the job/task initialization overhead, provides efficient data management to avoid the shuffling of static data among tasks, and allows asynchronous map task execution when possible. Accordingly, the system performance is greatly improved through these optimizations. For clarity, we first present iMapReduce for supporting graph-based iterative algorithms. Then, the original framework is slightly extended to support more iterative applications, which makes our framework more general on supporting iterative implementations. We demonstrate our results in the context of various applications, which show up to 5 times speedup over the traditional Hadoop MapReduce.

Acknowledgements 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.

References

1. Amazon ec2: http://aws.amazon.com/ec2/. Accessed 2011
2. Baluja, S., Seth, R., Sivakumar, D., Jing, Y., Yagnik, J., Kumar, S., Ravichandran, D., Aly, M.: Video suggestion and discovery for Youtube: taking random walks through the view graph. In: Proceedings of the 17th International Conference on World Wide Web (WWW '08), pp. 895–904 (2008)
3. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Comput Networks ISDN **30**, 107–117 (1998)
4. Bronshtein, I.N., Semendyayev, K.A.: Handbook of Mathematics, 3rd edn. Springer, London (1997)
5. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: Haloop: efficient iterative data processing on large clusters. Proc. VLDB Endow. **3**, 285–296 (2010)
6. Chakrabarti, S.: Dynamic personalized pagerank in entity-relation graphs. In: Proceedings of the 16th International Conference on World Wide Web (WWW '07), pp. 571–580 (2007)
7. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G.R., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. In: Proceedings of the 20th Neural Information Processing Systems (NIPS '06), pp. 281–288 (2006)
8. Clauset, A., Shalizi, C.R., Newman, M.E.J.: Power-law distributions in empirical data. SIAM Rev. **51**(4), 661–703 (2009)
9. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: Mapreduce online. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI '10), pp. 21–21 (2010)
10. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms, 2nd edn. McGraw-Hill Higher Education (2001)
11. Data center wiki page: http://en.wikipedia.org/wiki/Datacenter. Accessed 2011
12. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**, 107–113 (2008)
13. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.H., Qiu, J., Fox, G.: Twister: a runtime for iterative mapreduce. In: Proceedings of the 1st International Workshop on MapReduce and its Applications (MAPREDUCE '10), pp. 810–818 (2010)
14. Ekanayake, J., Pallickara, S., Fox, G.: Mapreduce for data intensive scientific analyses. In: Proceedings of the 4th IEEE International Conference on eScience (eScience '08), pp. 277–284 (2008)
15. Hadoop mapreduce: http://hadoop.apache.org/. Accessed 2011
16. Hadoop online prototype: http://code.google.com/p/hop/. Accessed 2011
17. Herlocker, J.L., Konstan, J.A., Terveen, L.G., Riedl, J.T.: Evaluating collaborative filtering recommender systems. ACM Trans. Inf. Syst. **22**, 5–53 (2004)
18. imapreduce on Google code: http://code.google.com/p/i-mapreduce/. Accessed 2012
19. Kambatla, K., Rapolu, N., Jagannathan, S., Grama, A.: Asynchronous algorithms in mapreduce. In: Proceedings of the 2010 IEEE International Conference on Cluster Computing (Cluster '10), pp. 245–254 (2010)
20. Kang, U., Tsourakakis, C., Faloutsos, C.: Pegasus: a peta-scale graph mining system implementation and observations. In: Proceedings of the 9th IEEE International Conference on Data Mining (ICDM '09), pp. 229–238 (2009)
21. Last.fm web services: http://www.last.fm/api/. Accessed 2011
22. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Statistical properties of community structure in large social and information networks. In: Proceedings

of the 17th International Conference on World Wide Web (WWW '08), pp. 695–704, (2008)
23. Liben-Nowell, D., Kleinberg, J.: The link-prediction problem for social networks. J. Am. Soc. Inf. Sci. Technol. **58**, 1019–1031 (2007)
24. Lin, J., Schatz, M.: Design patterns for efficient graph algorithms in mapreduce. In: Proceedings of the 8th Workshop on Mining and Learning with Graphs (MLG '10), pp. 78–85 (2010)
25. Lloyd, S.P.: Least squares quantization in pcm. IEEE Trans. Inform. Theory **28**, 129–136 (1982)
26. Logothetis, D., Olston, C., Reed, B., Webb, K.C., Yocum, K.: Stateful bulk processing for incremental analytics. In: Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC '10), pp. 51–62 (2010)
27. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC '09), pp. 6–146 (2009)
28. Microsoft windows azure platform: http://www.microsoft.com/windowsazure/. Accessed 2011
29. Mining of massive datasets: http://infolab.stanford.edu/ullman/mmds/book.pdf. Accessed 2010
30. Murray, D.G., Hand, S.: Scripting the cloud with skywriting. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10), p. 12 (2010)
31. Murray, D.G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., Hand, S.: Ciel: a universal execution engine for distributed data-flow computing. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11), p. 9 (2011)
32. Page, L., Brin, S., Motwani, R., Terry, W.: The pagerank citation ranking: bringing order to the web. In: Proceedings of the 9th International Conference on World Wide Web (WWW '98) (1998)
33. Peng, D., Dabe, F.: Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of the 9th Conference on Symposium on Opearting Systems Design and Implementation (OSDI '10), pp. 1–15 (2010)
34. Power, R., Li, J.: Piccolo: building fast, distributed programs with partitioned tables. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10), OSDI'10, pp. 1–14 (2010)
35. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multicore and multiprocessor systems. In: Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA '07), pp. 13–24 (2007)
36. Sarukkai, R.R.: Link prediction and path analysis using markov chains. Comput. Netw. **33**, 377–386 (2000)
37. Takács, G., Pilászy, I., Németh, B., Tikk, D.: Scalable collaborative filtering approaches for large recommender systems. J. Mach. Learn. Res. **10**, 623–656 (2009)
38. Wilson, C., Boe, B., Sala, A., Puttaswamy, K.P., Zhao, B.Y.: User interactions in social networks and their implications. In: Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09), pp. 205–218 (2009)
39. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10), p. 10 (2010)
40. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving mapreduce performance in heterogeneous environments. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08), pp. 29–42 (2008)
41. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Imapreduce: a distributed computing framework for iterative computation. In: Proceedings of the 1st International Workshop on Data Intensive Computing in the Clouds (DataCloud '11), p. 1112 (2011)
42. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Priter: a distributed framework for prioritized iterative computations. In: Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11), pp. 13:1–13:14 (2011)