iMapReduce: A Distributed Computing Framework for Iterative Computation

Yanfeng Zhang*, Qixin Gao*, Lixin Gao†, Cuirong Wang*
*Northeastern University, China
†University of Massachusetts Amherst, USA
Email: threewells14@gmail.com, gaoqx@mail.neuq.edu.cn, lgao@ecs.umass.edu, wangcr@mail.neuq.edu.cn

***Abstract*—Relational data are pervasive in many applications such as data mining or social network analysis. These relational data are typically massive containing at least millions or hundreds of millions of relations. This poses demand for the design of distributed computing frameworks for processing these data on a large cluster. MapReduce is an example of such a framework. However, many relational data based applications typically require parsing the relational data iteratively and need to operate on these data through many iterations. MapReduce lacks built-in support for the iterative process. This paper presents iMapReduce, a framework that supports iterative processing. iMapReduce allows users to specify the iterative operations with map and reduce functions, while supporting the iterative processing automatically without the need of users' involvement. More importantly, iMapReduce significantly improves the performance of iterative algorithms by (1) reducing the overhead of creating a new task in every iteration, (2) eliminating the shuffling of the static data in the shuffle stage of MapReduce, and (3) allowing asynchronous execution of each iteration, *i.e.,* an iteration can start before all tasks of a previous iteration have finished. We implement iMapReduce based on Apache Hadoop, and show that iMapReduce can achieve a factor of 1.2 to 5 speedup over those implemented on MapReduce for well-known iterative algorithms.**

I. Introduction

With the success of Web 2.0 and the popularity of online social networks, a huge amount of relational data is collected everyday. These relational data typically contain millions or hundreds of millions records. Analyzing the massive relational data in short time becomes a daunting task. MapReduce [1] is a popular framework for data intensive computation in a large cluster environment. Since its introduction, MapReduce has become extremely popular for analyzing the large data sets. It provides a simple programming framework and is responsible for distributed execution of computation, fault tolerance, and load balancing. This enables programmers with no experience with distributed systems to exploit a large cluster of commodity hardware to perform data intensive computation.

However, MapReduce is designed for batch-oriented computations such as log analyzing and text processing. On the other hand, many relational data based applications [2], [3] require iterative processing. This includes algorithms for text-based search and machine learning. For example, the well known PageRank algorithm parses the web linkage graph many times for deriving page ranking scores. The huge amount of relational data present on these applications demands for a parallel programming model for implementing these algorithms. However, MapReduce lacks the support for iterative processing.

Further, implementing iterative computation in MapReduce usually requires users to design a series of jobs, which poses several performance penalties. First, the jobs in each iteration always perform the same function. Creating, scheduling, and destroying these jobs repeatedly wastes considerable resources and processing time. This is particularly true for light weighted jobs. Second, the same relational data is required in every iteration. It has to be loaded and shuffled for each iteration. Third, serial execution of MapReduce jobs requires finishing the previous iteration job to start a new iteration job. This can unnecessarily delay the process.

In this paper, we propose *iMapReduce* that explicitly supports the iterative processing of large relational data, and addresses all the issues in the MapReduce implementation of iterative processing. First, it provides a framework for programmers to explicitly model iterative algorithms. Second, it proposes the concept of persistent tasks to perform the iterative computation to avoid repeatedly creating, destroying, and scheduling tasks. Third, the input data are loaded to the persistent tasks once and do not need to be shuffled between map and reduce. This can significantly reduce the I/O and the network communication overhead and the processing time. Fourth, it facilitates asynchronous execution of tasks within the same iteration, to accelerate the processing speed.

We implement a prototype of iMapReduce based on Apache Hadoop. Our prototype is backward compatible to MapReduce in the sense that it supports any MapReduce job. Further, it explicitly supports the implementation of iterative algorithms. Programmers only need to specify the jobs required within an iteration only. We evaluate our prototype with several well-known iterative algorithms. Our performance evaluation shows that iMapReduce can speed up the process by a factor of 1.2 to 5, comparing with the MapReduce implementation.

The rest of the paper is organized as follows: We introduce MapReduce iterative algorithms in Section II. Section III describes iMapReduce design and implementation in detail. Evaluation results are provided in Section V. Section VI generalizes iMapReduce to any iterative algorithm. We review the related work in Section VII and conclude in Section VIII.

Work was done while Yanfeng Zhang was at University of Massachusetts Amherst

II. Iterative Algorithms

Many algorithms for data analysis and machine learning use an iterative process. In this section, we first provide two examples of iterative algorithms, and then summarize the limitations of implementing these algorithms in MapReduce.

A. Iterative Algorithm Examples

We present the Single Source Shortest Path (SSSP) and PageRank algorithms, along with their MapReduce implementations in this section.

1) Single Source Shortest Path: The shortest path problem is a classic example of iterative processing. It finds the minimum distance to every vertex starting from a single source. Formally, we describe the shortest path computation as follows. Given a weighted, directed graph $G = (V, E)$, with edge weight matrix W mapping edges to real-valued weights, for a source node s, find the minimum distance to any vertex v, $d(v)$, from s.

To perform the shortest path computation in the MapReduce framework, we can traverse the graph in a breadth-first manner. We start from source s, with the distance to source is 0, $d(s) = 0$, while any other node distance is initially set as ∞. The map function is applied on each node u. The input key as the node id, while the input value has two parts. The first part is the minimum distance from s to u, *i.e.*, $d(u)$, and the second part is node u's outgoing edges. The mapper outputs $\langle v, W(u, v) + d(u)\rangle$, where v is one of u's linked-to nodes and $W(u, v)$ is the edge weight from u to v. After the shuffling stage, a number of possible distances for the same node from different predecessors are gathered to reduce, where the reducer selects the minimum one as the reduce output. In the next iteration, which is expressed as another MapReduce job, the same processing is performed based on the previous MapReduce job's output. The iterative process converges when all the nodes' shortest distances are obtained.

2) PageRank: PageRank is an algorithm for computing the importance of vertices in a graph. It been widely used in applications such as web search, link prediction [2]. Similar algorithms such as rooted PageRank and the average algorithm have found applications in personalized news and video recommendation systems [3].

The PageRank vector PR is defined over a directed graph $G = (V, E)$. Each vertex v in the graph has a pagerank score $PR(v)$. The initial rank of each node is $\frac{1}{|V|}$. Each vertex v updates its rank iteratively as follow.

$$PR^{(i+1)}(v) = \frac{1-d}{|V|} + \sum_{u \in N^-(v)} \frac{d \cdot PR^{(i)}(u)}{|N^+(u)|}, \qquad (1)$$

where $N^-(v)$ is the set of vertices pointing to vertex v, $N^+(v)$ is the set of vertices that v points to, and d is a constant representing the damping factor. This iterative process continues for a fixed number of iteration or till $\sum_v |PR_i(v) - PR_{i-1}(v)| < \epsilon$.

In MapReduce, the map processes each node v, where the input key is the node id, and the input value also have two parts, node v's ranking score $PR(v)$ and node v's successors $N^+(v)$. The map derives the partial ranking score of w ($w \in N^+(v)$), which is $PR(v)/|N^+(v)|$. The reduce sums these partial ranking scores associated with the same node w from different predecessors and produces a new ranking score of w, *i.e.*, $\frac{1-d}{|V|} + sum([PR(w)])$. The iterative process is expressed as a series of MapReduce jobs, with the output of the previous job feeding to the next job, until PageRank algorithm converges.

B. Limitations of MapReduce Implementation

We have several observations from the MapReduce implementation of these iterative algorithms.

- The operation of each iteration is the same. Nevertheless, MapReduce implementation starts a new job for each iteration. Each new job needs to be initialized and loads its data. This can result in unnecessary overhead.
- The graph adjacency or weight matrix is shuffled in each iteration between mappers and reducers despite the fact that the matrix remains the same in all iterations. One possible solution to this problem is not to shuffle the graph adjacency or weight matrix but to perform a join operation on the graph matrix with the page ranking score or shortest distance in each iteration. However, it will require an additional MapReduce task in each iteration, and programmers have to explicitly specify the join operation.
- The map tasks in an iteration cannot be started before the finishing of all the reduce tasks in the previous iteration. The main loop in the MapReduce implementation dedicates the completion of previous iteration before the start of the next iteration. However, the map tasks should be able to start as soon as its input data is available. That is, as along as the reduce tasks from the previous iteration has completed deriving the input data required for a map task, the map task should be able to start.

iMapReduce aims to address these issues. In doing so, we make two observations about the graph-based iterative algorithms. First, both map and reduce operations use the same key. This enables an one-to-one mapping between map and reduce tasks. Second, each iteration contains only one MapReduce job. Although these two observations might not be true for all iterative algorithms, we note that it is indeed true for a large class of graph-based iterative algorithms. We will make these assumptions in presenting iMapReduce in Section III for the ease of exposition. In Section VI, we will describe how iMapReduce can implement iterative algorithms in which the above two assumptions do not hold, and demonstrate the performance improvement of iMapReduce in those cases.

III. iMapReduce

In this section we introduce iMapReduce. As described in Section II, MapReduce implementation of iterative algorithms needs to create new tasks for each iteration, shuffle the static graph in each iteration, and execute map tasks in each iteration synchronously. iMapReduce addresses these issues

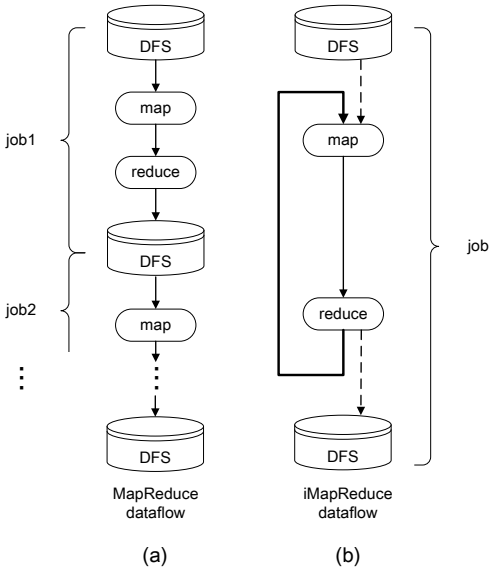


Fig. 1: (a) Dataflow of MapReduce. (b) Dataflow of iMapReduce.

by supporting iterative processes, eliminating the shuffling of the static data, and executing map tasks asynchronously. We will describe how iMapReduce implements these features in the next three subsections. In addition, we will describe the runtime support (including load balancing and fault tolerance mechanisms) of iMapReduce in Section III-D. Finally, we describe the application interface of iMapReduce in Section IV.

A. Supporting Iterative Processes

In the MapReduce implementation of graph-based iterative algorithms, a series of MapReduce jobs (consisting of maps and reduces) are scheduled. Figure 1(a) shows how the data flows in the MapReduce implementation. Each MapReduce job has to load data from Distributed File System (DFS) before the map operation. After the map operation operates to derive intermediate key value pairs, the reduce function operates on the intermediate data, and derives the output of the iteration, which is written to DFS. In the following iteration, map tasks load data from DFS and repeats the process. Finally, the iteration terminates when the termination condition is satisfied. We should note that these jobs including their component map/reduce tasks incur scheduling overhead. Additionally, this repeated DFS loading/dumping are expensive.

Note that every iteration performs the same operations. In other words, the series of jobs in iterative algorithms perform the same map and reduce functions. We exploit this property in iMapReduce by making map and reduce tasks *persistent*. That is, each map or reduce task is kept alive till the iteration is terminated. In order to feed the reduces' output to the maps' input, iMapReduce enables the reduce to pass its output directly to the map. Figure 1(b) shows the dataflow in iMapReduce. The dashed line indicates that the data loading from DFS happens just once in the initialization stage, and the data is written to DFS when the iteration terminates. During the iterative process, the reduce's output is directly sent to the map function for the next round of the iteration. As the implementation in MapReduce, the output of the maps are shuffled to feed to the reduces. In the following we describe how iMapReduce implements persistent tasks, and how the persistent tasks are terminated.

1) Persistent Tasks: In the MapReduce framework, each map or reduce task contains its portion of the input data and is assigned to a slave worker. The task runs by performing the map/reduce function on its input data records. Its life cycle ends when finishing processing all the input data records.

In contrast, each map and reduce task in iMapReduce is *persistent*. A persistent map or reduce task supports the iterative process by keeping alive during the whole iterative process. When all of the input data of a persistent task are parsed and processed, the task becomes dormant, waiting for the new updated input data. For a map task, it waits for the results from the reduce tasks and is activated to work on the new input records when the required data from the reduce tasks arrive. We will describe how the data is passed from the reduce tasks to the map tasks in Section III-B1. For the reduce tasks, they wait for the map tasks' output and are activated synchronously as in MapReduce.

To implement the persistent tasks, there should be enough *available task slots*. Available task slots indicate the number of tasks the system can accommodate (or allows to be executed) simultaneously. In MapReduce, the master splits a job into many small tasks, the number of tasks executed simultaneously cannot be larger than the available task slots. Once a slave worker completes an assigned task, it requests another one from the master. In iMapReduce, we need to set the granularity coarse enough so that there are sufficient available task slots for all the persistent tasks to start at the beginning. Clearly, this might make load balancing challenging. We will address this issue with a load balancing scheme in Section III-D1.

2) Termination Condition: Iterative algorithms typically terminate when a termination condition is met. Users terminate an iterative process in two ways; 1) Fixed number of iterations: Iterative algorithm stops after it iterates n times. 2) Bounding the distance between two consecutive iterations: Iterative algorithm stops when the distance is less than a threshold.

iMapReduce does the termination check after each iteration. To terminate the iterations by a fixed number of iterations, the persistent map/reduce task records its iteration number and terminates itself when the number exceeds a threshold. To bound the distance between the output from two consecutive iterations, the reduce tasks can save the output from two consecutive iterations and compute the distance. In order to get an overall distance from all the reduce tasks, the distances from the reduce tasks are merged by the master, and the master checks the termination condition to decide whether the iteration should be terminated. If the termination condition is satisfied, the master will notify all the map and reduce tasks to terminate their execution.

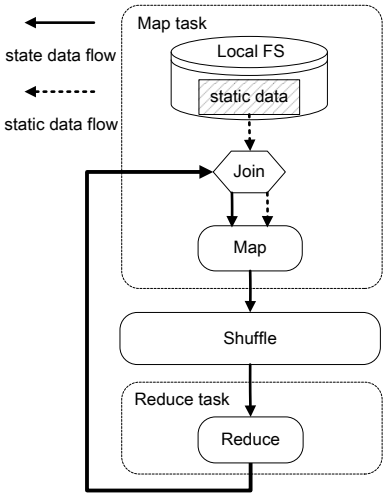


Fig. 2: Dataflow of the state data and the static data in iMapReduce.

B. Data Management

As described in Section II-B, the graph data does not change in each iteration. Nevertheless, the MapReduce implementation of these iterative algorithm has to shuffle the static graph. This can pose considerable overhead on I/O, network communication, and processing speed.

iMapReduce removes the shuffling of the graph data by dividing the input of the map task into the *static data* and the *state data*. The state data is updated after each iteration, while the static data remains the same after each iteration. For example, in the SSSP problem, the state data is the distance vector and the static data is the link list describing the input graph. Figure 2 shows the data flow in iMapReduce. As shown in the figure, the state data is passed from the reduce tasks to the map tasks and is joined with the static data in each iteration, and the static data is read from DFS only in the first iteration. Only the state data is shuffled in the shuffling stage.

In this section, we describe how the state data is passed from the reduce tasks to the map tasks, and how the state data and the static data are joined at the beginning of an iteration in iMapReduce.

1) Passing State Data from Reduce to Map: In MapReduce, the output of reduce is written to DFS and might be used later in the next MapReduce job. In contrast, iMapReduce should allow the state data to be passed from the reduce tasks to the map tasks so as to trigger the data join operation (between the static data and the state data) and start the map execution in the next iteration. To do so, iMapReduce builds persistent socket connections from the reduce tasks to the map tasks.

In order to simplify the process of building socket connections from the reduce tasks to the map tasks, we partition the graph nodes into subsets, and each subset is assigned to a map task as well as a reduce task. That is, each map task is assigned to the same subset of nodes with another reduce task. Therefore, there is a one-to-one correspondence between the map and the reduce tasks. Only one socket connection is needed for passing the state data from the reduce task to the corresponding map task.

Since the map task has a one-to-one correspondence with the reduce task, they hold the same node state. We can partition the static data using the same hash function as that used in shuffling the state data. In doing so, the state data is always shuffled to the map task where the corresponding static data is located. Further, in order to reduce the network resources needed for passing the state data, the task scheduler always assigns a map task and its corresponding reduce task to the same worker. This stays true even if task migration is performed as we will discuss in Section III-D1.

2) Joining State Data with Static Data: As shown in Figure 2, the map tasks take the input from both of the state data and the static data, while the reduce tasks take the input state data only and derive new state data. It is possible for the reduce tasks to take both the state and static data. However, this means that the shuffling process from map to reduce will need to be performed on both the static data and the state data. This can incur overhead on network communication as well as processing time, in particular, typically the static data is much larger than the state data.

iMapReduce takes the approach of splitting the static data and the state data before the shuffling. The reduce tasks take only the state data. Before the map tasks can be executed, a join operation between the state data and the static data has to be performed to ensure that each map task will contain the combined state and static data. iMapReduce automatically performs this operation without requiring users to write the corresponding MapReduce jobs. Since iMapReduce is supplied with its static and state data and the corresponding key, iMapReduce will automatically merge the state data and the static data for each map task before feeding the data into the map task.

C. Asynchronous Execution of Map Tasks

Since each map task needs only the state data from its corresponding reduce task, a map task can start its execution as soon as its state data arrives, without waiting for other map tasks. In iMapReduce, we schedule the execution of map tasks asynchronously.

To implement 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 produces a record, it is immediately sent back to its corresponding map task. On receiving the data from the reduce task, the map task starts processing the data. However, in order to be fault tolerant, we need to periodically save the state data in the local file system. In iMapReduce, we buffer a few reduce task output records and write them to a file before sending them to the map task through the socket. Further, eagerly triggering the map task will result in frequent context switches between reduce and map that impacts performance. Therefore, a buffer is designed at the reduce task. When the buffer grows to a size larger than a threshold, the data will be sent to the map tasks.

The join operation is performed as soon as the corresponding state data arrives from the reduce task of the previous iteration. Since the static data is always available, the join operation can be performed in an eager fashion. The map tasks start to execute immediately on getting the input data from the join operation. The reduce tasks cannot start until all the map tasks in the same iteration have been finished. In other words, the execution of the map tasks and the execution of the reduce tasks from the same iteration cannot be overlapped, as is the case in MapReduce. By enabling asynchronous execution of map tasks, the next iteration can start without waiting for the completion of previous iteration, which can further speed up the process.

D. 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 framework's success is its runtime support for load balancing and fault tolerance. In this section, we describe how iMapReduce supports load balancing and fault tolerance.

1) Load Balancing: In MapReduce, the master decomposes a submitted job into many small 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 the tasks are assigned to the workers in the beginning and executed simultaneously (since all the tasks are persistent as was discussed in Section III-A1). This one-time assignment conflicts with MapReduce's task scheduling strategy, so that we can not confer the benefit from MapReduce.

The less support of load balancing may lead to several problems: 1) Even though the initial input data is partitioned evenly among all the map tasks, 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 distributed evenly among tasks, it still cannot guarantee the best utilization of the computational resources, since a large cluster might consist of heterogeneous servers [4].

To address this problem, iMapReduce could perform task migration periodically (the implementation of the task migration mechanism is in progress). The master receives notification of completion sent by the reduce tasks after each iteration. By comparing the completion time, the master can differentiate stragglers from leaders. The straggler task indicates its own worker is heavy loaded, while the leader task's slave worker is light loaded. Thus, if there is a straggler, it is necessary to migrate this straggler task to a light loaded slave worker. In order to maintain the map and reduce task pair in the same slave worker, a map task and its corresponding reduce task are migrated together. The latest updated state data as well as static data for the map and reduce task pair are also migrated to the new slave worker before restarting the task pair.

2) Fault Tolerance: Fault Tolerance is important in a server cluster environment. iMapReduce relies on MapReduce mechanisms for fault-tolerance except that the state data is stored in DFS in order to keep the results from the previous iterations. In case there is a failure, iMapReduce returns to the last iteration with the state data, instead of starting the iterative process from the very beginning. The state data is usually small. Therefore, iMapReduce can maintain the data in local file system and DFS by saving the data after each iteration or a number of iterations. Note that the saving of the data to the local file system and DFS can be performed in parallel with the execution of the map/reduce tasks.

IV. Prototype and APIs

According to the design ideas we discussed above, we implement the iMapReduce prototype based on *Hadoop MapReduce* [5]. Any MapReduce jobs can run on our prototype. In addition, it supports iterative algorithms implementation. Users can turn on the functionality for implementing iterative algorithms, or turn them off for running MapReduce jobs as usual. iMapReduce extends MapReduce API as follows.

`void map(K, V1, V2)`. For the interface of map operation, in addition to the input key `K`, the value parameter consists of two parts: `V1` is the state data value and `V2` is the static data value. iMapReduce joins the state data and the static data internally, and users focus on describing the map computation.

`void setStatePath(Path)`. Users should specify the location of the initial state data.

`void setStaticDataPath(Path)`. Users should specify the location of the static data.

`void setMaxIterations(int)`. We can terminate iteration by checking how many iterations have been executed. A maximum iteration number can be set by this interface.

`void setDistanceThreshold(distanceFunc, double)`. Alternatively, the distance between the state data from two consecutive iterations are measured for termination check. iMapReduce compares the result of the distance measurement function and the threshold after each iteration for termination check.

`float distance(PrevState, CurrState)`. Users quantify the distance computation rule with the provided previous and current state information.

To show how to use iMapReduce prototype to implement iterative algorithms, an example of iMapReduce implementation for PageRank algorithm is given in Figure 3. In this example, the iterative processing is terminated by setting the distance threshold.

V. 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 prototype with that in Hadoop MapReduce [5]. Our experiments are performed on both a local cluster of commodity hardware and an Amazon EC2 cluster. The Hadoop Distributed File System's block size is 64MB, Hadoop heap size is set to 2GB. We describe the hardware environments as follows.

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

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

Distance
Input: PrevIterState R1, CurrIterState R2
 5: for node in R2 do
 6:     distance += |R2(node) – R1(node)|;
 7: end for
 8: output distance;

Main
 9: Job job = new Job();
10: job.setMap(Map);
11: job.setReduce(Reduce);
12: job.setStateDataPath("hdfs://…/initRankings");
13: job.setStaticDataPath("hdfs://…/graph");
14: job.setDistanceThreshold(Distance, 0.01);
15: job.submit();
```

Fig. 3: PageRank implementation in iMapReduce.

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

Amazon EC2 cluster: We build a test cluster on Amazon EC2 [6]. There are 80 small instances involved in our experiments. Each instance has 1.7 GB memory, Inter Xeon CPU E5430 2.66GHz, 146.77 GB instance storage and runs 32-bit platform Linux Debian 4.0 OS.

A. Data Sets

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

We first describe the graphs used for evaluating the SSSP algorithm as follows. 1) DBLP author cooperation graph. In DBLP graph, each node represents an author and a link between two nodes represents the cooperation relationship between the two authors. The link weight is set according to the cooperation frequency of the two linked authors. 2) The Facebook user interactions graph [7]. Facebook user is a node and the friendship between two users implies a link between them. The interaction frequency is used to assign weights to the user friendship links on this graph. 3) Synthetic graph. Based on [8], the power-law parameters on the link weight and the node out-degree are extracted from the above two real graphs. We then generate three synthetic graphs with 1 million, 10 million, and 50 million nodes. Table I shows the detail of these data sets.

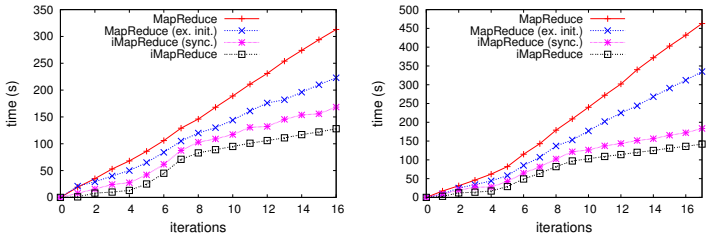
The data set used for PageRank is as follows. There are two real graphs: 1) Google webgraph [9] and 2) Berkley-Stanford webgraph [9]. 3) The power-law parameters on the node's out-degree are extracted from the above real graphs, and we generate 3 synthetic graphs with 1 million, 10 million, and 30 million nodes respectively. Data set statistics are shown in Table II.

TABLE I: SSSP data set statistics

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

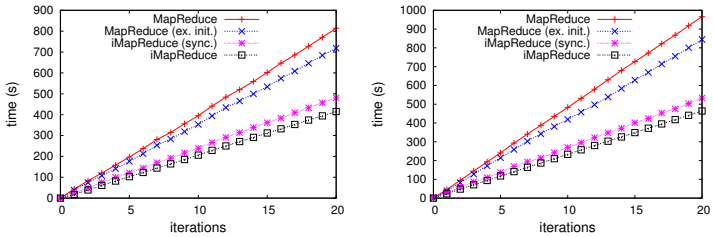
TABLE II: PageRank data set statistics

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



(a) (b)

Fig. 4: The running time of SSSP on local cluster. (a) DBLP author cooperation graph. (b) Facebook user interaction graph.



(a) (b)

Fig. 5: The running time of PageRank on local cluster. (a) Google webgraph. (b) Berkley-Stanford webgraph.

B. Local Cluster Experiments

We evaluate SSSP and PageRank under both MapReduce and iMapReduce. Since the real graphs are relatively small, we use real graphs as input to run experiments on our local cluster.

Figure 4 shows the running time of SSSP on MapReduce and iMapReduce with DBLP graph and Facebook graph as input, respectively. We can see that comparing with the MapReduce implementation, iMapReduce achieves a factor of 2-3 speedup. Figure 5 shows the running time of PageRank

on MapReduce and iMapReduce with google webgraph and berkley-stanford webgraph as input, respectively. Comparing with the MapReduce implementation, iMapReduce achieves about 2 times speedup.

As we described in Section III, there are three factors that help improve performance. 1) *Asynchronous map execution* eliminates the execution delay. 2) With the help of persistent map/reduce task, iMapReduce performs *one-time initialization* rather than spending time on initializing jobs/tasks for every iteration in MapReduce. 3) By managing static data locally, iMapReduce *eliminates static data shuffling*, which reduces the running time.

In order to investigate how these three factors improve performance, we use the following methods to measure each factor's contribution. 1) We first start MapReduce job and iMapReduce job respectively and get two running time results as two benchmarks, the gap between these two running time results indicates the benefit it gains from iMapReduce. 2) Based on iMapReduce, we let maps execute synchronously and record the running time, so that the running time difference from iMapReduce indicates the contribution of asynchronous map execution factor. 3) For measuring the job/task initialization time consumed in MapReduce implementations, we record the time interval between job submission and the first map execution for each map task, and average the time intervals information collected from all the map tasks. The initialization time in MapReduce is the initialization time for all the iteration jobs, while iMapReduce has one-time initialization that happens in the initial stage. We assume the estimated initialization time for MapReduce minus that of iMapReduce is the time saved by iMapReduce, which indicates the contribution of one-time initialization factor. 4) We accumulate iMapReduce running time with the above two factors' contribution time. The time difference between the accumulated time and the MapReduce running time indicates the performance contribution by eliminating static data shuffling.

Figure 4 and Figure 5 show the measurement results. For SSSP, we can see that asynchronous map task execution can reduce the running time by about 20%, and around 25% running time is saved from one-time job/task initialization. For PageRank, we can see asynchronous map execution reduces the running time by about 15%, while one-time job/task initialization contributes about another 20%.

C. Amazon EC2 Cluster Experiments

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

1) Running Time: We run SSSP over three synthetic graphs SSSP-s, SSSP-m, and SSSP-l on Amazon EC2 cluster (20 instances). We limit to 10 iterations and compare the running time on different synthetic graphs. Figure 6a shows the result. For the small graph, the iMapReduce implementation reduces the running time to 23.2% of that of MapReduce. iMapReduce reduces the running time to 37.0% and 38.6% of that of MapReduce for the medium graph and large graph, respectively. We can see that iMapReduce performs better when the input graph is small. Since for small graphs, there is relatively more time spent on job/task initialization, while iMapReduce does not need to perform these operations for each iteration.

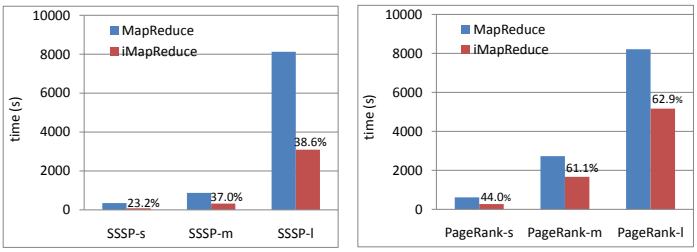


Fig. 6: The running time for the synthetic graphs on the Amazon EC2 cluster. (a) SSSP. (b) PageRank.

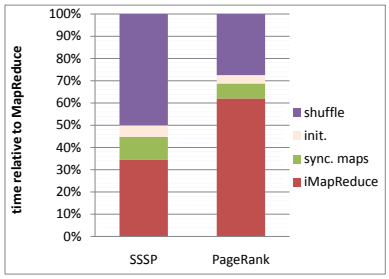


Fig. 7: Different factors' effects on running time reduction.

Similarly, PageRank is executed with 10 iterations on three synthetic graphs PageRank-s, PageRank-m, and PageRank-l on Amazon EC2 cluster (20 instances). The result is shown in Figure 6b. Similar to SSSP, more speedup for the PageRank-s graph is achieved, while a steady speedup factor is around 1.6.

To explore the performance improvement achieved by different factors, asynchronous map execution, one-time initialization, and static data shuffling elimination, we show in Figure 7 the running time reduction by these factors. SSSP and PageRank are both computed in 10 iterations on the SSSP-m graph and the PageRank-m graph respectively. We can see that the running time reduced by asynchronous map execution and one-time initialization remains constant independent from applications and graphs, and the time for shuffling the graph is proportional to the input graph file size (SSSP-m 958MB and PageRank-m 690MB).

2) Communication Cost: As we claimed, iMapReduce saves the network communication cost by eliminating static graph shuffling. A large amount of data is communicated between the map tasks and the reduce tasks. Reducing the amount of data communicated not only helps improve performance, but also saves communication resources. To quantify the amount of data communicated, we show in Figure 8 the total bandwidth used for running SSSP and PageRank both on the large synthetic graphs. As shown, iMapReduce significantly reduces communication cost.

3) Scalability: For scalability test, since our prototype is implemented based on Hadoop MapReduce, which scales well,

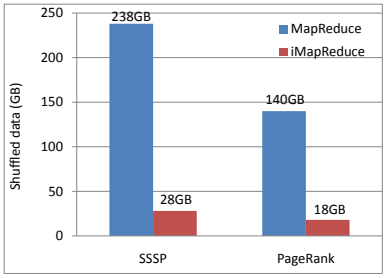
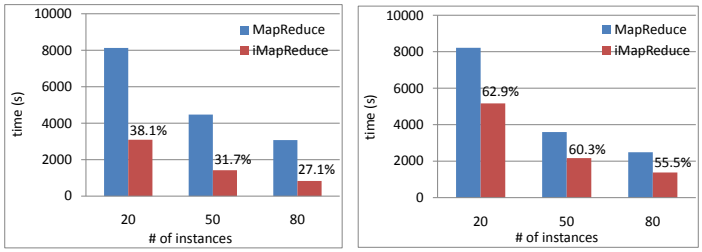


Fig. 8: Total communication cost.



(a) (b)

Fig. 9: The speedup over MapReduce implementations on Amazon EC2 cluster. (a) SSSP. (b) PageRank.

the scalability of iMapReduce prototype should meet most applications' and users' needs. Additionally, it is difficult to measure the maximum it can scale. Therefore we just scale our Amazon EC2 cluster to contain 50 nodes and 80 nodes, and run SSSP and PageRank on the SSSP-l graph and the PageRank-l graph, respectively. iMapReduce works fine without problems and it accelerates algorithms by using more computing resources. Moreover, we find that iMapReduce even performs better on a larger scale cluster.

Figure 9a shows the time consumed for running SSSP on different-size clusters. The figure shows the running time ratio of iMapReduce to MapReduce is reduced 11% when we scale from 20 instances to 80 instances. Figure 9b shows the scalability test result of PageRank. We can see that the running time ratio of iMapReduce to MapReduce is reduced 7% when we scale the cluster size. We explain these results as follows. The bigger the cluster, the more network communications would occur. Since our proposal aims at reducing network communications, it is more likely to exert its advantages on the bigger cluster.

VI. Extensions of iMapReduce

So far, we have focused on supporting graph-based iterative algorithms. iMapReduce can be extended to implement other iterative algorithms as well. In this section, we present extensions to iMapReduce that can support any iterative algorithm. As described in Section II-B, we make two assumptions in iMapReduce; (1) both map and reduce operations use the same key, and (2) each iteration consists of a single MapReduce job. We describe how to extend iMapReduce to support iterative algorithms where these assumptions do not hold in the next two subsections.

A. Accommodating Different Keys in Map and Reduce

In an iterative algorithm, it is possible that the keys used in map and reduce are different. For example, in the KMeans clustering algorithm, the keys in map and reduce are respectively node id and cluster id. We will describe how to extend iMapReduce to accommodate such an algorithm. First, we describe KMeans clustering algorithm.

1) KMeans Clustering Algorithm: KMeans is a commonly used clustering algorithm. With an input parameter k, the KMeans algorithm partitions n nodes into k clusters so that 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 nodes randomly as cluster centers, (2) Assign each node to its nearest cluster center, (3) Update the k cluster centers by "averaging" the nodes belonging to the same cluster center. Repeat steps (2) and (3) until the cluster assignment for all nodes is converged.

We can implement the KMeans algorithm in MapReduce as follows. The map function computes the similarity between every cluster center to a node and assigns the node to the most similar cluster center, so that all the nodes are grouped into clusters. The reduce updates the cluster center by averaging all the nodes assigned to the same cluster. The map needs all the cluster centers for a particular node to select the closest one to be assigned to. This means that the mapping from reduce tasks to map tasks is not one-to-one but one-to-all. That is, the state data (in this case, cluster center set) from each reduce task should be sent to all the map tasks.

Another difference between KMeans and SSSP/PageRank is that not only the map operation needs the static data (in this case, all coordinates of all nodes) for measuring distance, but also the reduce operation needs the static data for the averaging operation. That is, static data has to be shuffled between the map and reduce operations. Moreover, the map function's key is the node id while the reduce function's key is the cluster id. The keys for map and reduce operations are different.

2) iMapReduce Extensions: To support "KMeans-like" iterative algorithms, iMapReduce lets reduce tasks broadcast the updated state data to all the map tasks. Similarly, for the algorithms that have one-to-multiple mapping from reduce tasks to map tasks, we let reduce tasks multicast the updated state data to multiple map tasks.

For both one-to-all and one-to-multiple mapping, each map operation needs the state data from multiple reduce operations. Accordingly, we describe the mapping in the map interface by extending the input parameter *StateValue* (Section IV) to a list of state values. In the case of KMeans, the state value list is the the set of all the updated cluster centers.

When the reduce operation needs the static data, the static data have to be shuffled between the map and reduce operations. This is the case for the KMeans algorithm, where the node coordinates have to be shuffled. However, the join operation does not need to be performed in this case.

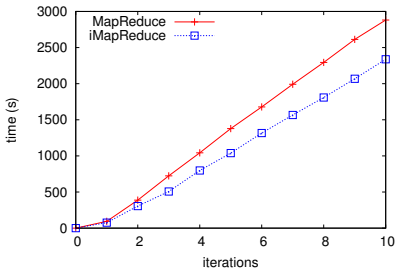


Fig. 10: Running time of KMeans for clustering Last.fm data on the local cluster.

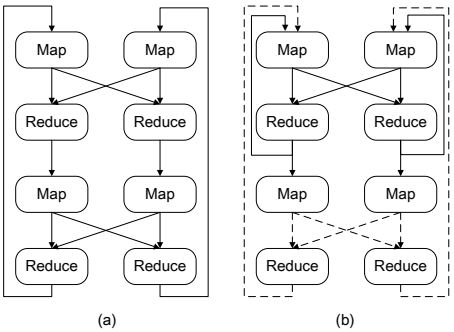


Fig. 11: The data flows with multiple MapReduce jobs.

Further, the map operation needs the output from a set of reduce tasks. The map operation can start only after its input data arrives. In the case of KMeans, the map operation cannot be started before all the updated cluster centers are collected. That is, map tasks cannot be executed asynchronously. Therefore, the option for map task synchronization should be turned on. In general, we can trigger the execution of map tasks when all reduce tasks that supply the input have completed.

3) Evaluation: We implement KMeans in iMapReduce and run the algorithm under the data set collected from Last.fm [10]. Last.fm is a popular music listening website. We use Last.fm's user listening history log for clustering the users based on their tastes. This log contains each user's artist preference information quantified by the times that music by the artist is listened. Sharing preferred artists indicates a common taste. Last.fm data set (1.5GB) has 359,347 user records and each user has 48.9 preferred artists on average.

As shown in Figure 10, iMapReduce achieves a factor of 1.2 speedup comparing with MapReduce for KMeans clustering algorithm. This is much less than that achieved for SSSP and PageRank. Nevertheless, this is expected since the implementation of KMeans needs to shuffle static data and has to execute the map tasks synchronously.

B. Accommodating Multiple MapReduce Jobs

Iterative algorithms might iterate on several MapReduce jobs. In this section, we discuss some potential extensions to iMapReduce to implement such iterative algorithms (work in progress). The key of these algorithms is to specify the mapping from that last reduce operation to the first map operation. Since the tasks in iMapReduce are persistent, the right connection directs dataflow from the reduce tasks to the right map tasks and let the iteration keep running. Figure 11(a) shows the data flow for the case that the multiple MapReduce jobs are executed serially. The second MapReduce job continues processing the first MapReduce job's output. And the second MapReduce job directs the output to the map of the first MapReduce job. In this case, each iteration consists of two MapReduce jobs.

Furthermore, it is possible to have multiple MapReduce jobs execute in parallel. Figure 11(b) shows an example of this scenario. The first MapReduce job will use the result of the second MapReduce job. The first MapReduce job is the main job which executes the iteration, and the second one is an auxiliary job which takes the main job's output and produces some auxiliary information for the main job. For example, users might design their own termination check with a MapReduce job. In this case, the auxiliary MapReduce job does termination check while the main job does iteration considering the auxiliary job's result on termination check.

VII. Related Work

MapReduce, a popular framework for performing data intensive computation on a large cluster of commodity hardware, has gained considerable attention over the past few years. The framework has been extended for diverse application requirements. MapReduce Online [11] pipelines and performs online aggregation to support efficient online queries. Their pipelining technique inspires our work on iterative processing data. There are a number of studies on improving the MapReduce framework for iterative processing [12], [13], [14], [15], [16], [17], [18], [19], [20], [21].

A class of these efforts targets on managing the static data efficiently. Design patterns for running efficient graph algorithms in MapReduce have been introduced in [19]. They partitioned the graph structure into n parts, and pre-stored the graph partitions on DFS. However, since the MapReduce framework arbitrarily assigns reduce tasks to workers, accessing graph vertex adjacency list will almost always involve remote reads. Therefore, this does not guarantee local access of the data for the graph. Most recently, HaLoop [21] was proposed aiming at iterative processing in a large cluster. HaLoop realizes the join of the static data and the state data by explicitly specifying an additional MapReduce job and relies on the task scheduler to maintain local access of the data. iMapReduce relies on persistent tasks to manage the static data and avoid tasks initialization, and we go further by allowing asynchronous execution of the map tasksp.

Some studies accelerate iterative algorithms by maintaining iteration state in memory. Twister [20] employs a lightweight MapReduce runtime system and uses publish/subscribe messaging based communication/data transfers instead of DFS. All the operations (including map and reduce) are performed in memory cache, by which it enhances the latency of accessing data. However, the dependence on memory does not make it

scale and fault tolerant, which is very important in a distributed large cluster environment. Spark [14] was developed recently to optimize iterative and interactive computation. It uses caching techniques to dramatically improve the performance for repeated operations. The main abstraction in Spark is *resilient distributed dataset* (RDD), which is maintained in memory across iterations and fault tolerant.

Some efforts focus on iterative graph algorithms, an important class of iterative algorithms. PEGASUS [12] modeles 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 5x faster performance over the regular job. Pregel [15] 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.

VIII. Conclusions and Future Work

In this paper, we propose iMapReduce that supports the implementation of iterative algorithms under a large cluster environment. iMapReduce extracts common features of iterative algorithms and provides support for these features. In particular, it proposes the concept of persistent tasks and persistent socket connections between tasks. It provides support for eliminating shuffling of static data among tasks, and for asynchronous execution of iterations when possible. We demonstrate our results in the context of three popular applications, Single Source Shortest Path, PageRank, and KMeans. Our results show a factor of ranging from 1.2 to 5 speedup can be achieved for these iterative algorithms. In addition, the data communication cost can be significantly reduced.

In future work, we plan to extend iMapReduce to more iterative algorithms and make it more general. On the other hand, more frameworks should be considered for performance comparison. For example, Dryad [22], [23], as another popular distributed computing framework, can represent more than one MapReduce stage at once, which could mitigate some of the serialization overhead.

Acknowledgment

The authors are grateful to the anonymous reviewers for their helpful comments and suggestions. 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] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, 2004.

[2] H. H. Song, T. W. Cho, V. Dave, Y. Zhang, and L. Qiu, "Scalable proximity estimation and link prediction in online social networks," in *IMC '09: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, 2009, pp. 322–335.

[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," in *WWW '08: Proceeding of the 17th international conference on World Wide Web*, 2008, pp. 895–904.

[4] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *OSDI '08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 29–42.

[5] Hadoop mapreduce. [Online]. Available: http://hadoop.apache.org/

[6] Amazon ec2. [Online]. Available: http://aws.amazon.com/ec2/

[7] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 205–218.

[8] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Rev.*, vol. 51, no. 4, pp. 661–703, 2009.

[9] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," Tech. Rep., Oct 2008. [Online]. Available: http://arxiv.org/abs/0810.1355

[10] Last.fm web services. [Online]. Available: http://www.last.fm/api/

[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10, 2010, pp. 21–21.

[12] U. Kang, C. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *ICDM '09: Proceedings of the 9th IEEE International Conference on Data Mining*, 2009, pp. 229 –238.

[13] D. G. Murray and S. Hand, "Scripting the cloud with skywriting," in *HotCloud '10: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 12–12.

[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *HotCloud '10: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pp. 6–6.

[16] P. Russell and L. Jinyang, "Piccolo: Building fast, distributed programs with partitioned tables," in *OSDI'10: Proceedings of the 9th conference on Symposium on Opearting Systems Design and Implementation*, 2010.

[17] P. Daniel and D. Frank, "Large-scale incremental processing using distributed transactions and notifications," in *OSDI'10: Proceedings of the 9th conference on Symposium on Opearting Systems Design and Implementation*, 2010.

[18] K. Karthik, R. Naresh, J. Suresh, and G. Ananth, "Asynchronous algorithms in mapreduce," in *IEEE Cluster'10: Proceedings of the 2010 IEEE International Conference on Cluster Computing*, 2010.

[19] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *MLG '10: Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, 2010, pp. 78–85.

[20] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *MAPREDUCE '10: Proceedings of the 1st International Workshop on MapReduce and its Applications*, 2010, pp. 810–818.

[21] B. Yingyi, H. Bill, B. Magdalena, and D. E. Michael, "Haloop: Efficient iterative data processing on large clusters," in *VLDB '2010: Proceedings of the 36th international conference on Very Large Data Bases*. VLDB Endowment, 2010.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007, pp. 59–72.

[23] Y. Yu, M. Isard, D. Fetterly, M. Budiu, lfar Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *OSDI '08: Proceedings of the 8th conference on Symposium on Opearting Systems Design and Implementation*, pp. 1–14.