iHadoop: Asynchronous Iterations for MapReduce

Eslam Elnikety*
Max Planck Institute for
Software Systems (MPI-SWS)
Saarbruecken, Germany
elnikety@mpi-sws.org

Tamer Elsayed*
Cairo Microsoft Innovation Lab
Cairo, Egypt
telsayed@microsoft.com

Hany E. Ramadan
King Abdullah University of
Science and Technology (KAUST)
Thuwal, KSA
hany.ramadan@kaust.edu.sa

***Abstract*—MapReduce is a distributed programming framework designed to ease the development of scalable data-intensive applications for large clusters of commodity machines. Most machine learning and data mining applications involve *iterative* computations over large datasets, such as the Web hyperlink structures and social network graphs. Yet, the MapReduce model does not efficiently support this important class of applications. The architecture of MapReduce, most critically its dataflow techniques and task scheduling, is completely unaware of the nature of iterative applications; tasks are scheduled according to a policy that optimizes the execution for a single iteration which wastes bandwidth, I/O, and CPU cycles when compared with an optimal execution for a consecutive set of iterations.**

This work presents iHadoop, a modified MapReduce model, and an associated implementation, optimized for iterative computations. The iHadoop model schedules iterations asynchronously. It connects the output of one iteration to the next, allowing both to process their data concurrently. iHadoop's task scheduler exploits inter-iteration data locality by scheduling tasks that exhibit a producer/consumer relation on the same physical machine allowing a fast local data transfer. For those iterative applications that require satisfying certain criteria before termination, iHadoop runs the check concurrently during the execution of the subsequent iteration to further reduce the application's latency. This paper also describes our implementation of the iHadoop model, and evaluates its performance against Hadoop, the widely used open source implementation of MapReduce. Experiments using different data analysis applications over real-world and synthetic datasets show that iHadoop performs better than Hadoop for iterative algorithms, reducing execution time of iterative applications by 25% on average. Furthermore, integrating iHadoop with HaLoop, a variant Hadoop implementation that caches invariant data between iterations, reduces execution time by 38% on average.

I. Introduction

We live in a digital world where data sizes are increasing exponentially. By 2007, the digital universe was estimated at 281 exabytes, and it is expected to experience more than a tenfold growth by 2011 [15]. Data are being generated at unprecedented scale and rate. This has a direct effect on the scale of datasets that need to be processed, with volumes varying from many terabytes to a few petabytes. For example, analyzing the hyperlink structure of the Web requires processing billions of Web pages, mining popular social networks involves millions of nodes and billions of edges, multi-dimensional astronomical data are collected at rates exceeding 100 GB/day [24], and Facebook is hosting over 260 billion images [2]. This explosion of available data has motivated the design of many scalable distributed frameworks [18], [12] to face the challenges presented when processing and analyzing those sizes. Distributed and parallel data analysis frameworks have several common traits, but most notably, they aggregate computing power from the available computing infrastructure to perform large scale data processing.

MapReduce is a large scale data-intensive processing framework that scales out to thousands of commodity machines [12]. With MapReduce, developers can focus on their domain-specific tasks, while the framework is responsible for low-level system issues such as job scheduling, load balancing, synchronization, and fault tolerance which is critical for long-running jobs. Developers need to implement two functions, *map* and *reduce*, and do not need to be concerned about the failures of unreliable commodity machines, or complicated parallelism and synchronization constructs. MapReduce is used in many applications such as indexing [23], data mining [30], [21], and machine learning [7], [37]. Compared to other approaches for large scale processing, e.g., parallel databases and grid computing, MapReduce is more scalable and more suitable for data-intensive processing [10]. Its ease of use is best reflected by its increasing popularity; Hadoop [16], an open source implementation of MapReduce, has been adopted by several enterprises such as Yahoo!, eBay, and Facebook [36]. It follows the basic architecture and programming model initially introduced by Google's MapReduce [12].

Iterative computations represent an important class of applications. They are at the core of several data analysis applications such as PageRank [3], k-means, Hyperlink-Induced Topic Search (HITS) [22], and numerous other machine learning and graph mining algorithms [31], [44], [34]. These applications process data iteratively for a pre-specified number of iterations or until a termination condition is satisfied. The output of an iteration is used as input to subsequent iterations.

Given the massive sizes of data and the importance of iterative computations, we need a scalable solution to efficiently apply iterative computations on large scale datasets. On one hand, the MapReduce framework scales to thousands of machines, but on the other hand, it was not designed to run iterative applications efficiently. Since a series of iterations can be represented as two or more back-to-back MapReduce jobs, developers can use a driver program to submit the necessary

*This work was performed while the first two authors were at King Abdullah University of Science and Technology (KAUST) in Thuwal, KSA.

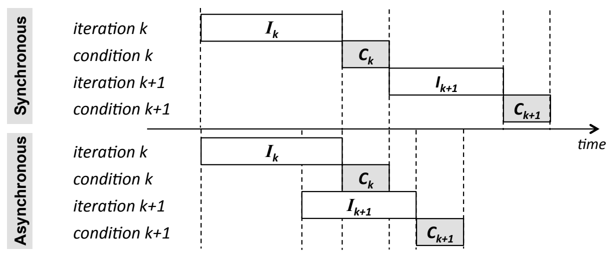


Fig. 1. Synchronous iterations versus asynchronous iterations

MapReduce jobs for each iteration. If the application has to meet certain criteria before stopping, the driver program is responsible for running the termination check after each iteration. The termination check itself can be a separate MapReduce job, or a separate program that is invoked by the driver. This approach has two significant limitations. An iteration must wait until:

- the previous iteration has finished completely and its output has been entirely written and committed to the underlying file system, and
- the termination check, if needed, has finished.

Since each iteration starts, usually, by reading from the file system what has just been written by the previous one, significant amounts of precious network bandwidth, I/O, and CPU cycles are wasted.

This work presents *iHadoop*, a modification of the MapReduce framework optimized for iterative applications. iHadoop modifies the dataflow techniques and task scheduling of the traditional MapReduce model, to make them aware of the nature of iterative computations. iHadoop strives for better performance by executing iterations *asynchronously*, where an iteration starts before its preceding iteration finishes. This is achieved by feeding the output of an iteration (as it progresses) to the following one, which allows both to process their data concurrently. Extra care is taken to preserve the MapReduce design principle of transparent fault tolerance, even with asynchronous iterations. iHadoop also modifies the task scheduler so as to schedule tasks that exhibit a producer/consumer relation of consecutive iterations on the same physical node *where* the cost of inter-iteration data transfer is minimal. Since iterations are running asynchronously with iHadoop, its task scheduler decides *when* it is optimal to schedule a certain task.

Some applications need to check the termination condition between every two consecutive iterations I_k and I_{k+1}. If the application converges after n iterations, the termination condition is satisfied only once out of n times. iHadoop runs this termination check in the background after starting iteration I_{k+1} asynchronously, speculating that the condition will not be satisfied after the completion of iteration I_k. If it turns out to be satisfied, iteration I_{k+1} is immediately terminated; this results in wasted computation, but it is insignificant compared to the performance gains achieved by asynchronous iterations. If the condition is not satisfied, I_{k+1} will have made considerable progress by the time the check concludes as we show experimentally in Section VI-B2. Figure 1 visually shows the difference between the execution pipelines of synchronous and asynchronous iterations, in the presence of termination checks.

This work makes the following main contributions:

- It introduces iHadoop, a modified version of the MapReduce model that allows iterations to run asynchronously. iHadoop also schedules the termination check, if needed, concurrently with the subsequent iteration. This results in greater parallelism at the application level and better utilization of resources.
- It modifies the MapReduce task scheduler to be aware of the asynchronous behavior. iHadoop takes advantage of inter-iteration locality and schedules the tasks that have strong dataflow interaction on the same physical node whenever possible.
- It presents an associated implementation of the iHadoop model, and describes issues encountered while tuning the performance of asynchronous iterations.
- It presents an experimental evaluation of iHadoop along with a performance comparison with Hadoop, using real and synthetic datasets.

The remaining of this paper is organized as follows. Section II gives an overview of the MapReduce framework–readers familiar with MapReduce can skip this section. Section III discusses the related work. We present the design of iHadoop in Section IV followed by a discussion of its implementation in Section V. We evaluate the performance of iHadoop in different settings in Section VI. Finally, we conclude in Section VII

II. MapReduce Framework

MapReduce, introduced by Dean and Ghemawat in 2004, is a framework for large scale data processing using commodity clusters [12]. Its simple programming interface can support a broad range of applications. The framework transparently distributes data, allocates tasks, and parallelizes computations across cluster nodes in a shared-nothing architecture. It is extensively used in large scale data centers such as those operated by Google, Facebook, and Amazon.com.

The programming model adopted by MapReduce is inspired by functional programming. The framework defines two primitives to be implemented by the user: *map* and *reduce*. These two primitives have the following signatures:

$$map : (k_1, v_1) \rightarrow [(k_2, v_2)],\ \ reduce : (k_2, [v_2]) \rightarrow [(k_3, v_3)]$$

The *map* function is applied on every (k_1, v_1) pair and produces a list of intermediate (k_2, v_2) pairs. The *reduce* function is applied on all intermediate pairs with the same key and outputs a list of (k_3, v_3) pairs.

A MapReduce execution passes through three phases; Map, Shuffle, and Reduce. In the Map phase, every map task applies the *map* function on its input split. There is no ordering requirements for the input of the Map phase, i.e., a map task can process any part of the input. In the Shuffle phase, the reduce tasks copy and sort their respective parts of the intermediate outputs produced by all map tasks. In the Reduce phase, the reduce tasks apply the *reduce* function on their inputs to produce the output of the MapReduce job. In contrast to the Map phase, the Reduce phase has ordering requirements, i.e., a reduce task cannot apply the *reduce* function on its input unless the respective parts of all the map tasks' outputs are ready and sorted.

III. Related Work

This section reviews models and systems related to iterative large scale processing.

A. Iterative MapReduce Implementations

There are many implementations of the MapReduce model [16], [11], [40], [27], [17], [39], [33], [6]. The following describes the MapReduce systems that optimize the framework for iterative applications.

HaLoop [4] is a modified version of Hadoop that supports iterative large scale applications. HaLoop caches and indexes loop-invariant data on local disks, for iterative applications that follow the construct $R_{i+1} = R_0 \cup (R_i \bowtie L)$, where R_0 is the initial result and L is an invariant relation. HaLoop modifies the task scheduler to reassign to each node the set of tasks that were assigned to it in earlier iterations. This reduces significantly the amount of data that need to be shuffled between the map and the reduce tasks of the same iteration. For those iterative applications that require a termination check, HaLoop introduces *fixpoint evaluation* by comparing, in a distributed manner, the current iteration output with the cached output of the previous iteration. However, this evaluation is limited only to those iterative algorithms where the convergence criteria depend only on the output of the two most recent iterations.

Twister [14] is a stream-based MapReduce implementation that supports iterative applications. It uses a publish/subscribe messaging infrastructure configured as a broker network for sending/receiving data to/from tasks and daemons. Twister configures the tasks to load the data from disks once whenever they start. To avoid repeatedly reloading data in every iteration, Twister uses long running map/reduce tasks, i.e., it does not initiate new map and reduce tasks for every iteration. In case of failures, the entire computation is rolled back few iterations to the last saved state. Twister is based on the assumption that datasets and intermediate data can fit into the distributed memory of the computing infrastructure, which is not the case for clusters of commodity machines where each node has limited resources.

iMapReduce [43] also provides a framework that can model iterative algorithms. It uses *persistent tasks* and keep all the tasks alive during the whole iterative process. iMapReduce keeps a one-to-one mapping between the map and the reduce tasks and its task scheduler assigns statically every map task and its corresponding reduce to the same worker. The reduce tasks only produce dynamic data, which are afterwards joined with the static data by the map tasks. This allows iMapReduce to run the map tasks of the subsequent iteration asynchronously while the reduce tasks has to wait until all the map tasks of the same iteration are finished. iMapReduce's static scheduling policy and coarse task granularity can lead to an unoptimized resource utilization and load unbalancing. In case of a failure, the computation is rolled back to the last saved state. iMapReduce optimizations are limited to those applications where every iteration corresponds to exactly one MapReduce job/step. In cases where the map and reduce functions have different keys, iMapReduce has to start the map tasks of the following iteration synchronously.

Unlike the previous systems that use a static mapping between tasks and nodes, which can lead to load unbalancing, iHadoop uses a dynamic scheduling policy. The use of persistent/long-running tasks, like in Twister and iMapReduce, limits some resources of the infrastructure to a certain job even if they remain idle. This requires that the computing infrastructure can accommodate all the persistent tasks simultaneously, and puts a limit on the number of the MapReduce jobs that can run concurrently. iHadoop does not persist tasks for the following iteration; since with large scale datasets, the runtime optimizes the task granularity so that the time it takes to create, destroy, and schedule tasks is insignificant compared to the time required by each task to process its input. iHadoop is the first to our knowledge to execute asynchronously the map and reduce tasks of the following MapReduce job in an iterative process without any limitations on the supported iterative applications: Any iterative MapReduce application can run over iHadoop with the asynchronous iterations optimization.

B. Frameworks Based on MapReduce and Alternatives

Many systems have been built on top of Hadoop, such as HadoopDB [1], Hive [35], Pig [29], Pegasus [21] and Apache Mahout[25]. Similarly, FlumeJava [5] builds an optimized execution plan using Google's MapReduce implementation. Unlike iHadoop, these systems do not leverage the possible inter-iteration optimizations that can be applied to iterative algorithms.

Kambatla et al. [20] worked around the global synchronization overhead between the Map and the Reduce phases in the MapReduce model. They extended the model with the notion of partial synchronization, where they apply locality-enhancing partitioning scheme and relax the strict synchronization on inter-components edges. Their technique is targeted at the class of computations where violating global synchronization will not threaten the correctness of the computation.

The dataflow between successive iterations in this work is similar to the technique presented by MapReduce Online [8] which adds a pipelining functionality to Hadoop to allow "early" output estimation. However, in MapReduce online, the data that are being passed from a MapReduce job to the next represent snapshots of the computation and the second MapReduce job has to be recomputed from scratch once the first job finishes.

Pregel [26] is a distributed system designed for processing large graph datasets but it does not support the broad class of iterative algorithms that iHadoop does. Spark [42] supports iterative and interactive applications while providing scalability

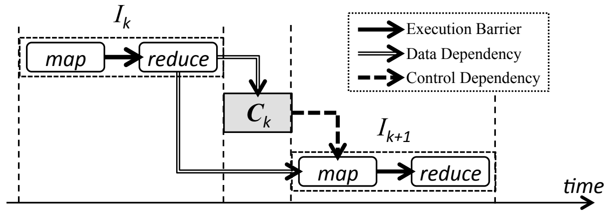


Fig. 2. Different dependencies in iterative MapReduce

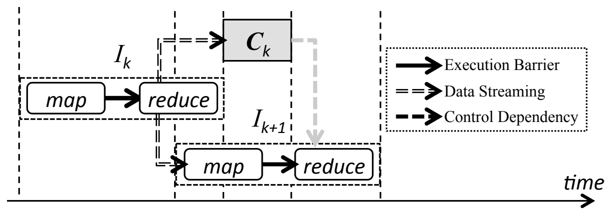


Fig. 3. Asynchronous pipeline

and fault tolerance similar to those of MapReduce. However, Spark does not support parallel reductions as in MapReduce. CIEL [28] can make control-flow decisions enabling iterative and recursive computations but does not exploit the possible asynchronous execution of iterations.

Dryad [18] is a distributed framework based on a directed acyclic graph (DAG) execution model. Dryad is more expressive and flexible than MapReduce but not as straightforward to use. DryadLINQ [41] provides a high-level programming API for Dryad graphs. It supports iterative computation using standard loops. Ekanyake et al. [13] show that even with loop unrolling, DryadLINQ still has programming model overhead when performing iterative computations.

Parallel database systems are able to exploit parallelism by partitioning and replicating their storage and optimizing query execution plans accordingly [9], [32], [19]. Although they can scale linearly for tens of nodes, there are few deployments with more than one hundred nodes. Parallel databases often run on expensive homogeneous architectures and assume that failures are rare — which is not the case with large commodity clusters. We are not aware of any published parallel database deployment that was able to scale to thousands of machines.

IV. iHadoop Design

This section examines the various dependencies present during the execution of iterative algorithms with MapReduce. Having identified the points where additional parallelism can be safely exploited, we then describe the asynchronous iterations model, followed by a description of concurrent termination checking. Introducing asynchronous iterations has implications for task scheduling, fault tolerance, and load balancing. We address these issues in Sections IV-D–IV-F.

A. Dependencies in MapReduce processing

Figure 2 shows some of the dependencies observed in the execution of a typical iterative application in MapReduce. Only the dependencies relevant to our work are shown.

Execution Barrier: where execution of a phase has to wait until the execution of an earlier phase is *completely* finished (synchronization). For example, there is an execution barrier between Map and Reduce phases in every MapReduce step.

Data Dependency: where the output of one phase is delivered as input to the next phase. There is a data dependency between the Reduce phase of iteration I_k and the Map phase of iteration I_{k+1}. Similarly, when the application requires a termination check, there is a data dependency between the Reduce phase of iteration I_k and the termination check C_k.

Control Dependency: which determines if the computation should proceed. There is a control dependency between the termination check C_k and the Map phase of iteration I_{k+1}.

B. Asynchronous Iterations

Every map task is associated with an input split stored in the distributed file system. Across a sequence of iterations, the input split for each map task can be either a part of the output of the immediately preceding iteration, or a part of static data or the output of an earlier iteration (not the immediately preceding).

For an input split of the later case, the corresponding map task can start, with no changes in the MapReduce dataflow, asynchronously with the previous iteration since its data are already available in the distributed file system (DFS). The following discussion is concerned with those map tasks that consume parts of the output of the previous iteration as their input splits. With the typical MapReduce dataflow, those map tasks will have to wait until their input splits are available in the DFS, i.e., until the previous iteration is completely finished.

A mapper of I_{k+1} can process any part of the input data in any order. There is *no* execution barrier between the Reduce phase of I_k and the Map phase of I_{k+1} despite the data dependency; there is nothing in theory that prevents mappers of I_{k+1} from progressing whenever input data are available, even while the reducers of I_k are still processing their inputs and emitting their outputs. Therefore, we propose that the output of I_k can be fed as input to the mappers of I_{k+1}. Figure 3 shows how the data dependency between the Reduce phase of iteration I_k and the Map phase of iteration I_{k+1} is converted into a communication channel.

With asynchronous iterations, every reducer opens two streams: (1) a stream to the underlying file system, similar to MapReduce, to maintain the same fault tolerance model of MapReduce, and (2) a socket stream to mappers of a following iterations, used for direct transfer of the inter-iteration data.

C. Concurrent Termination Checking

Figure 2 shows that a termination check at the end of iteration k, C_k, is involved in two types of dependencies:

- *Data dependency* with the Reduce phase of the previous iteration I_k; this data dependency can be handled in a way similar to the data dependency between that Reduce phase of I_k and the Map phase of I_{k+1}.

- *Control dependency* with iteration I_{k+1}.

If the iterative application will eventually run for n iterations, the termination condition will not be satisfied for the the first $n-1$ iterations, and will only be satisfied at the last one. iHadoop speculates that the termination condition will not be satisfied for *every* iteration, and thus the next iteration can immediately start without delay. In the asynchronous setup, described earlier and shown in Figure 3, iteration I_{k+1} ignores the control dependency and starts even while iteration I_k is running. Whenever C_k is checked and found satisfied (i.e., iterative computation has to stop), the iHadoop framework sends a termination signal to I_{k+1} cleaning up all the tasks belonging to it.

iHadoop runs the termination check in the background concurrently with asynchronous iterations. For $n-1$ out of n iterations, iHadoop takes advantage of the correct speculation that allows the next iteration to make significant progress upon the completion of the earlier iteration. Although the following iteration will be useless if the condition is satisfied, this is insignificant compared to the time savings asynchronous iterations can achieve. Section V discusses when precisely the termination condition gets checked.

D. Task Scheduling

Every reduce task of iteration I_k feeds its output to one or more map tasks of iteration I_{k+1}. The iHadoop task scheduler takes advantage of this inter-iteration locality by scheduling tasks that have a direct data dependency on the same physical machine. Map tasks of the first iteration are scheduled using MapReduce's default task scheduler. For subsequent iterations, the iHadoop task scheduler tries to schedule a reduce task and its associated set of map tasks (those tasks that exhibit a producer/consumer relation) on the same physical machine.

While the MapReduce task scheduler tries to schedule map tasks on a machine that has a local replica of its assigned input split, the iHadoop task scheduler achieves the same goal since a map task will receive its input split streamed from a reduce task running on the same physical machine. With large clusters, network bandwidth is a relatively scarce resource and the iHadoop task scheduler strives to reduce the network traffic and hence the overall execution time. If iteration I_{k+1} has parts of its input that are not produced by iteration I_k (like static data in the DFS), iHadoop task scheduler schedules the map tasks that will be processing these data according to the default MapReduce policy.

MapReduce implementations have traditionally considered where computations (tasks) should be executed (i.e., on which node). Since iHadoop supports asynchronous iterations, it raises a new factor that should be taken into consideration by implementations, namely *when* to schedule tasks for optimal execution. We discuss this in more detail, in the context of our implementation, in Section V-A.

E. Fault Tolerance

For systems with many commodity machines, failures should be treated as the normal case rather than the exception [38], and thus MapReduce was designed to tolerate machine failures gracefully.

iHadoop builds on top of MapReduce's fault tolerance mechanism; it handles the following two additional failure scenarios in the context of the asynchronous iterations:

- failures of in-progress reduce tasks that are streaming their outputs to one or more map tasks of the immediately following iteration, and
- failures of map tasks that are consuming their input directly from a reducer of the immediately preceding iteration.

Upon failure of an in-progress reduce task, iHadoop restarts in-progress map tasks that are receiving input from the failed reducer, whereas completed map tasks do not have to be re-started. Since the reduce operator is deterministic, we only need a deterministic reduce-to-map partitioning scheme to ensure that each map task of the subsequent iteration will process the same input in case of a reduce task restart.

Every reduce task in asynchronous iterations sends its output to the set of linked mappers of the following iteration and to the distributed file system. Upon failure of a map task that is consuming its input directly from a reducer that did not fail, iHadoop restarts the map task but its input is retrieved in this case from the distributed file system if it is ready. Otherwise, the map task waits until the input is ready in the distributed file system.

F. Load Balancing

Load balancing in MapReduce depends on two design issues. (1) Task granularity: MapReduce launches a number of map and reduce tasks that far exceeds the number of worker nodes in the cluster. This improves the dynamic load balancing by being able to allocate an approximate equal share of the work to each node. (2) Dynamic scheduling: Each worker updates the master with its current load and available resources periodically. The master uses this information to schedule tasks dynamically wherever there are available resources.

iHadoop load balancing inherits its behavior from MapReduce. iHadoop does not require any changes to the task granularity. However, the iHadoop task scheduler tries to schedule every pair of tasks that exhibit a producer/consumer relation together on the same physical machine. Ideally, every reduce task of iteration I_k will have an equal share of the work and will be producing an equal share of the output, which will result in an equal share of work to the map tasks of iteration I_{k+1}. As an extension to improve the load balancing for an iterative application over iHadoop, a map task of iteration I_{k+1} can be scheduled to run on any less loaded worker node. But in the later case, its input will be streamed over the network rather than the fast local inter-process transfer.

V. iHadoop Implementation

iHadoop is built on top of Hadoop [16]. This section discusses some optimizations and issues encountered when implementing iHadoop.

```
00:33:16 JobClient: I:9 ( map 100% reduce 32%) I:10 ( map 0% reduce 0%)
00:33:42 JobClient: I:9 ( map 100% reduce 55%) I:10 ( map 2% reduce 0%)
00:33:43 JobClient: I:9 ( map 100% reduce 56%) I:10 ( map 4% reduce 0%)
00:33:55 JobClient: I:9 ( map 100% reduce 77%) I:10 ( map 27% reduce 3%)
00:34:20 JobClient: I:9 ( map 100% reduce 91%) I:10 ( map 74% reduce 18%)
00:34:59 JobClient: I:9 ( map 100% reduce 99%) I:10 ( map 99% reduce 31%)
00:35:01 JobClient: I:9 ( map 100% reduce 100%) I:10 ( map 100% reduce 32%)
00:35:02 JobClient: I:10 ( map 100% reduce 32%) I:11 ( map 0% reduce 0%)
00:35:08 JobClient: I:10 ( map 100% reduce 53%) I:11 ( map 0% reduce 0%)
00:35:15 JobClient: I:10 ( map 100% reduce 67%) I:11 ( map 1% reduce 0%)
00:35:48 JobClient: I:10 ( map 100% reduce 72%) I:11 ( map 14% reduce 0%)
00:36:46 JobClient: I:10 ( map 100% reduce 78%) I:11 ( map 34% reduce 5%)
```

Fig. 4. A sample iHadoop execution log of asynchronous iterations

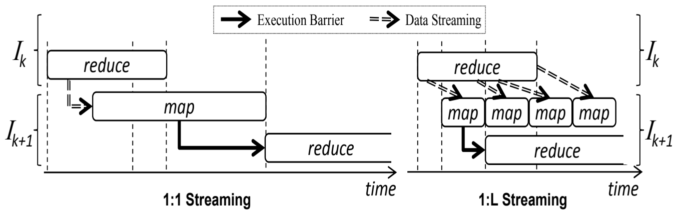


Fig. 5. Reducer to mappers streaming

A. Reducer Early Start

iHadoop's task scheduler is optimized to launch tasks in a manner which achieves better utilization of resources. We denote a reducer from iteration I_k that is in the phase P by $R_{k,P}$. A reducer $R_{k,Reduce}$ applying the *reduce* function on its input will be streaming its output to mappers of iteration I_{k+1}. In this scenario, a reducer $R_{k+1,Shuffle}$ of iteration I_{k+1} can start collecting the intermediate outputs of the mappers of iteration I_{k+1} as they finish. We call this optimization *reducer early start* where reducers of two consecutive iterations I_k and I_{k+1} are processing and shuffling, respectively, their input at the same time.

Figure 4 shows this optimization in action. As reducers of iteration I_9 are streaming their outputs to mappers of iteration I_{10}, the reducer early start will try to schedule reducers of I_{10} to start collecting intermediate output from the mappers of I_{10} as they finish. Note that by the time I_9 finishes, the mappers of I_{10} have made significant progress and reducers of I_{10} were able to collect much of the intermediate output.

A reducer starts the Shuffle phase by copying the intermediate output corresponding to its partition of the key space from all the map tasks. For reducers of iteration I_{k+1} to be able to shuffle the outputs of the mappers of the same iteration efficiently, each reducer of iteration I_k should be streaming its output to several map tasks of iteration I_{k+1}. We investigate this in more detail in the next subsection.

B. Reducer to Mappers Streaming

Figure 5 illustrates how streaming is implemented in iHadoop. The left hand side shows "1:1 streaming" where a reducer of I_k is streaming all its emitted output to a single mapper of I_{k+1}. The reducer of I_{k+1} has to wait until this mapper finishes before it can shuffle its intermediate output.

To mitigate the potentially long delay, iHadoop implements a "1:L streaming" mechanism where the reducer streams its intermediate output to L *linked* mappers of I_{k+1} in succession, each of which will receive an equal portion of that output and thus can potentially finish in L times less than the case for "1:1 streaming". This allows a reducer of I_{k+1} to start collecting its input earlier. Notice that the reducer of I_k still streams to one mapper at a time. The number of linked mappers, L, has to be chosen carefully so that the overhead of creating that number of mappers and new connection streams does not degrade overall performance; we investigate the effect of varying L in Subsection VI.

C. Mapper JVM Reuse

To decrease the overhead of creating new connection streams between a reducer and the corresponding linked mappers of the following iteration, iHadoop reuses the same Java Virtual Machine (JVM) for these mappers. Before a reducer of iteration I_k starts the Reduce phase, iHadoop's task scheduler ensures that there is a mapper of iteration I_{k+1} running on the same physical machine waiting to establish a socket connection with that reducer. Once a connection is established, the JVM will manipulate that connection as a shared object between subsequent map tasks. When the map task finishes, the JVM promptly requests the next map task to run, which will use the shared connection stream to read its input from the reducer.

After applying the *map* function on their inputs, mappers need to create partitions for every reducer and commit them to disk. These operations can take a long time and delay the flow of data from a reducer to the next linked mapper. To speed up this transition, every map task, after applying the *map* function, forks two threads: the first is a low priority thread that commits the intermediate output to local disk, and the other of higher priority is to run the next map task.

D. Concurrent Termination Check

A termination check C_k typically runs after the conclusion of each iteration I_k. If this check is implemented as a MapReduce job, then a similar streaming technique (to the one discussed earlier from reducers of I_k to mappers of I_{k+1}) can be adopted. This enables concurrent execution of an iteration and its termination check.

C_k can generally be any user-defined program (i.e., not necessarily a MapReduce job), so the current implementation of iHadoop only launches the termination check C_k after iteration I_k commits its output to HDFS. In Subsection VI-B2 we measure the performance gains achieved from running asynchronous iterations while the termination check executes concurrently.

VI. Experimental Evaluation

This section evaluates the performance of iHadoop, compares it to other variants that have different optimizations enabled, and discusses the experimental results.

A. Evaluated Systems and Setup

The implementation of iHadoop is based on Hadoop 0.20.2 which is the baseline to all the evaluated systems. `Hadoop`,

TABLE I
SYSTEMS COMPARED IN THE EXPERIMENTS

System	Iterative API	Asynchronous	Caching	Note
Hadoop	-	-	-	Baseline
$\texttt{iHadoop}_a$	✓	✓	-	
$\texttt{iHadoop}_c$	✓	-	✓	$\equiv$ HaLoop
$\texttt{iHadoop}_{ac}$	✓	✓	✓	

the baseline, uses a driver program that controls the execution of a sequence of iterations. If a termination check is required, this check is executed as a separate MapReduce job that is launched after the completion of each iteration. The x in $\texttt{iHadoop}_x$ indicates features denoted by single letters: 'a' for 'asynchronous' and 'c' for 'caching invariant data' as listed in Table I. $\texttt{iHadoop}_a$ uses asynchronous iterations and concurrent termination check. $\texttt{iHadoop}_c$ caches invariant data between iterations. We used HaLoop to obtain results for this optimization. However, we keep the name '$\texttt{iHadoop}_c$' for the sake of consistency. Since the optimizations introduced in this work are orthogonal to the caching optimization, $\texttt{iHadoop}_{ac}$ is our implementation of the asynchronous iterations support while caching and indexing invariant data between iterations based on the HaLoop caching technique.

We ran the experiments on the two clusters described in Table II. Each cluster has a separate NameNode/JobTracker machine with 2.67 GHz Intel® Xeon® X5550 processor and 24GB of RAM. We used the two datasets described in Table III: LiveJournal[1], and WebGraph[2]. All the results presented in this section are obtained without any node/task failures and no other user processes/jobs were concurrently running with our jobs on the clusters. By default, we set the number of reducers R to the number of the nodes in the cluster where we run the experiment. A maximum of 6 map tasks and 2 reduce tasks were configured per node. Unless otherwise stated, the output of each iteration is written to HDFS with replication factor of 3 upon its completion, the number of iterations is pre-specified with no termination check, and $\texttt{iHadoop}_a$ and $\texttt{iHadoop}_{ac}$ have the number of linked mappers L set to 5.

TABLE II
CLUSTER CONFIGURATIONS

	Cluster_{12}	Cluster_6
Nodes	12	6
Total Cores	48	24
Threads/Core	2	
RAM/Node	4GB	
Processor	Intel ® Core™ i7-2600	
Network	1Gbit	

Three iterative applications were used in the evaluation: PageRank (PR), Descendant Query (DQ), and Parallel Breadth-First Search (PBFS). These applications represent two different classes of iterative algorithms; PR and PBFS represent those algorithms where the input of an iteration is mainly the output of the preceding one, while DQ represents those where the input of an iteration depends on previous iterations along with the immediately preceding one. PR is a link analysis algorithm. Each iteration is represented as (i) a join operation between ranks and the linkage structure, followed by (ii) a rank aggregation step. The input for the join step is the output of the previous aggregation step and some (cachable) invariant data (e.g., linkage structure). The input for the aggregation step is the output of the previous join step only. PBFS is a graph search algorithm. Each iteration is represented by a single MapReduce job. The mappers of this job read the output of the earlier iteration, update the distance of each node, and emit the updated records to the reducers. For each node, the reducers set the smallest distance discovered so far. DQ is a social networking algorithm, each iteration is represented as (i) a join operation between vertices discovered from the previous iteration and the linkage structure, followed by (ii) a duplicate elimination step. The input for the join step is the output of the previous duplicate elimination step and some invariant metadata. The input for the duplicate elimination step is the output of the previous join step and the outputs of *all* previous duplicate elimination steps.

[1]http://snap.stanford.edu/data/soc-LiveJournal1.html
[2]http://lemurproject.org/clueweb09.php/

TABLE III
DATASETS USED IN THE EXPERIMENTS

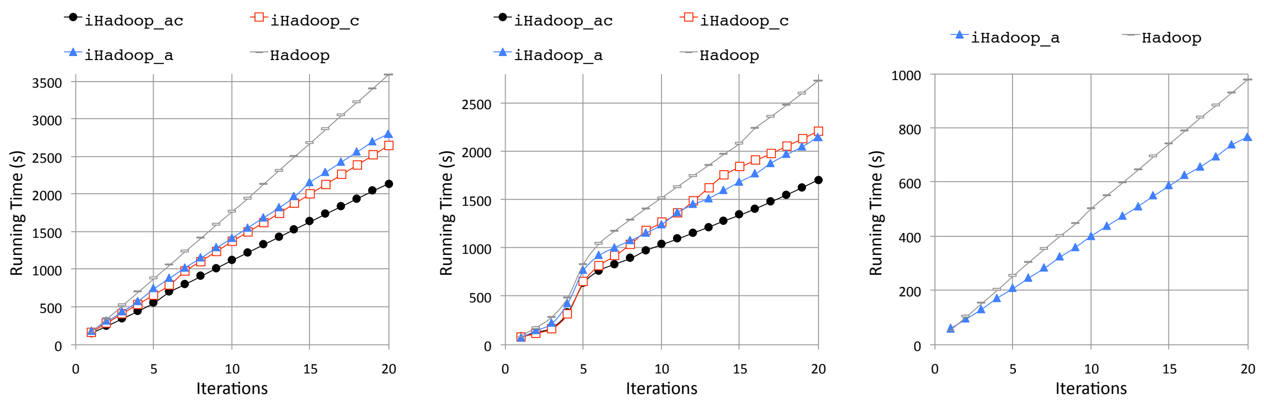
Dataset	Nodes	Edges	Description
LiveJournal	4,847,571	68,993,773	Social network
WebGraph	50,013,241	151,173,117	First English segment of ClueWeb09

B. Results

1) Asynchronous Iterations Evaluation: **PageRank.** We ran PR on Cluster_{12} for 20 iterations using the WebGraph dataset; Figure 6(a) represents the overall running time of the entire job, each iteration is comprised of both the join and aggregation steps of PR. Figure 7(a) illustrates the average execution time per iteration, normalized to the baseline.

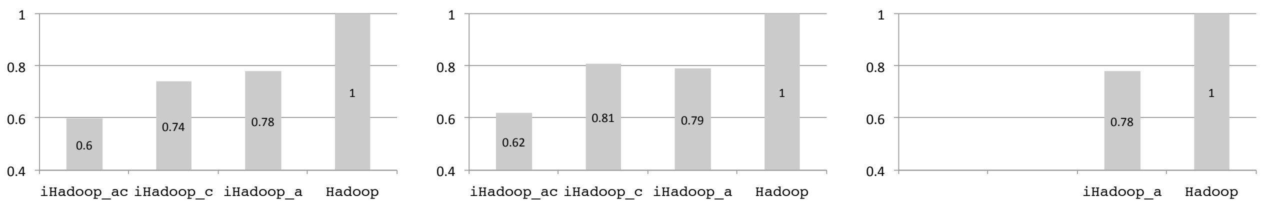
The results indicate that $\texttt{iHadoop}_a$ performs significantly better than Hadoop, reducing the average running time to 78% due to using asynchronous iterations. $\texttt{iHadoop}_c$ adopting the caching of invariant data, is able to reduce the running time to 74%. The combination of asynchronous iterations and caching, featured in $\texttt{iHadoop}_{ac}$, exhibits the best performance of 60% of the baseline execution time. The asynchronous behavior of $\texttt{iHadoop}_{ac}$ reduces the running time to 81% when compared to $\texttt{iHadoop}_c$.

Descendant Query. Figures 6(b)–7(b) show the performance of running DQ on Cluster_6 for 20 iterations using the LiveJournal dataset. $\texttt{iHadoop}_a$ and $\texttt{iHadoop}_{ac}$ had L set to 4 in this experiment since the amount of output that each reducer produces in DQ are relatively smaller. $\texttt{iHadoop}_a$ reduces the total running time to 79% compared with Hadoop. $\texttt{iHadoop}_c$ achieves a similar improvement of 81% over Hadoop. Moreover, $\texttt{iHadoop}_{ac}$ reduces the total run time to 62% of the baseline.



(a) PageRank (b) Descendant Query (c) Parallel Breadth-First Search

Fig. 6. Total running time of 20 iterations



(a) PageRank (b) Descendant Query (c) Parallel Breadth-First Search

Fig. 7. Normalized average iteration time

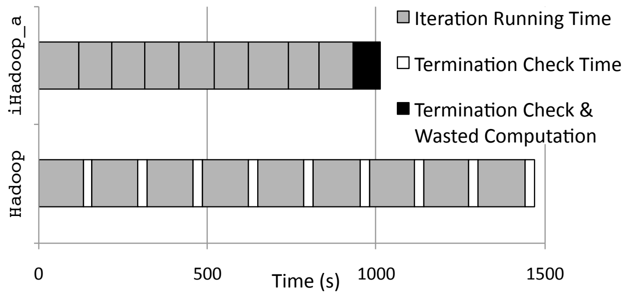


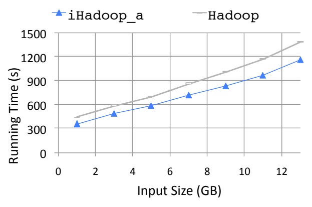
Fig. 8. Concurrent vs. synchronous termination condition check

In DQ, as the number of iterations increases, the input size for one iteration is dominated by the outputs of *earlier* previous iterations rather than the immediately proceeding iteration. For iteration I_{k+1} in $\texttt{iHadoop}_a$ and $\texttt{iHadoop}_{ac}$, the ratio of data that is being transferred via local streaming (the output of iteration I_k) compared to the data that is being transferred from HDFS (the outputs of iterations I_{k-1}, I_{k-2}, ..) is getting smaller as the iteration number increases. This mitigates the effect of the fast local data transfer. This demonstrates that starting an iteration earlier has a larger effect on the time savings achieved by $\texttt{iHadoop}_a$ and $\texttt{iHadoop}_{ac}$ compared to the fast local data transfer between a reducer and its linked mappers.

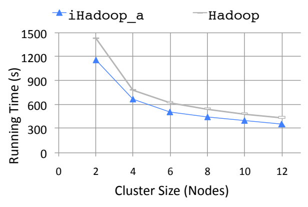
Parallel Breadth-First Search. Figure 6(c)– 7(c) show the performance of PBFS on Cluster_{12} for 20 iterations using the WebGraph dataset. $\texttt{iHadoop}_a$ reduces the total running time to 78% compared with `Hadoop`. This experiment is missing the values for $\texttt{iHadoop}_c$ and $\texttt{iHadoop}_{ac}$, since we cannot apply the caching techniques to a "one-step" PBFS since each iteration reads the whole graph and generates an updated version of the graph as a whole.

Overall, the results for these applications show clearly that asynchronous iterations significantly improve the performance of iterative computations, under different configurations that use a variety of optimization settings.

2) Concurrent Termination Check: To measure the performance gain that `iHadoop` achieves when termination checking is performed concurrently with the subsequent iteration, we ran PR on Cluster_6 using the LiveJournal dataset with a user-defined termination condition that was satisfied at the end of the ninth iteration. Figure 8 compares the performance of `Hadoop` and $\texttt{iHadoop}_a$. The shaded blocks indicate the execution time of the iterations while the white ones indicate the execution time of the termination check. Since the check is always performed concurrently with the iterations, it was never featured in $\texttt{iHadoop}_a$. As a side effect of the concurrent execution, a portion of the tenth iteration was already performed when the condition was finally satisfied. We call this portion *wasted computation*. The black block in Figure 8 represents the time $\texttt{iHadoop}_a$ spent after the ninth iteration in (a) the termination check and (b) wasted computation. We quantify the wasted computation by subtracting the average termination check time from the total time spent after the ninth iteration. However, the wasted computation is less than 5% of the total running time. Running the termination check

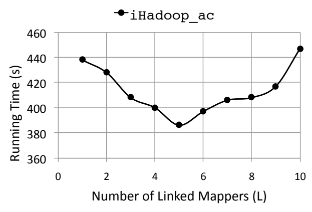


(a) Variable input size

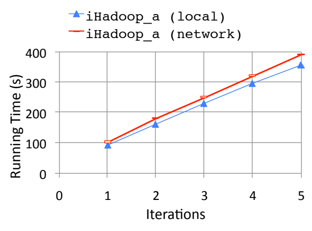


(b) Variable cluster size

Fig. 9. Scalability tests



(a) Varying L



(b) Network streaming

Fig. 10. Performance tuning

concurrently with the asynchronous iterations of $\mathtt{iHadoop}_a$ introduced an overhead of 7% as measured by comparing this execution to the execution of $\mathtt{iHadoop}_a$ without a termination check. Whereas the termination check for `Hadoop` introduced 24% overhead to the running time.

Overall, the experiment shows that asynchronous iterations and concurrent termination checking in $\mathtt{iHadoop}_a$ reduce the overall execution time to 69%, achieving an average speed up of 1.45 per iteration over `Hadoop`.

3) iHadoop Scalability Tests: To examine the scalability of the asynchronous behavior of $\mathtt{iHadoop}_a$, we used PR over the LiveJournal dataset for 5 iterations in the following set of experiments. In Figure 9 (a), we compare the performance of $\mathtt{iHadoop}_a$ and `Hadoop` when varying the input size. We replaced the identifiers in LiveJournal with longer strings to generate several sizes of the dataset. In Figure 9 (b), we compare the performance of $\mathtt{iHadoop}_a$ and `Hadoop` when varying the number of nodes in the cluster. For both, $\mathtt{iHadoop}_a$ and `Hadoop`, we set the number of reducers to the number of nodes. These experiments show that $\mathtt{iHadoop}_a$ scales as well as `Hadoop`.

4) iHadoop Performance Tuning: **Number of Linked Mapper.** As discussed in Section V-B, the number of linked mappers per reducer (L) can be critical to iHadoop performance. While increasing L will increase the overhead of creating multiple mappers, it will also help reducers (of the same iteration of the linked mappers) collect intermediate data faster as mappers finish. Figure 10(a) illustrates the effect of varying L on the total execution time of 5 iterations of PR using the LiveJournal dataset on Cluster_6. From the figure, $\mathtt{iHadoop}_{ac}$ performs best at $L = 5$ for PageRank with the LiveJournal dataset. The optimal value for L depends on the amount of data every reducer generates.

Network vs. Local Streaming. iHadoop's task scheduler exploits inter-iteration locality by scheduling tasks that exhibit a producer/consumer relationship on the same physical machine. This local data transfer is faster and saves bandwidth. In Figure 10(b), we compare, using PR on Cluster_{12} over the LiveJournal dataset, the performance of $\mathtt{iHadoop}_a$ when every reduce task and its set of linked mappers are scheduled on the same physical machine (local) versus the performance of $\mathtt{iHadoop}_a$ when forcing every reduce task to run on a machine other than those running its set of linked map tasks (network). In $\mathtt{iHadoop}_a$ (network), every reduce task sends its output to its linked map tasks over the network. Overall, $\mathtt{iHadoop}_a$ (local) was 9% faster due to the faster data transfer between reducers and their linked mappers and the overall savings in bandwidth.

Skipping HDFS. Every MapReduce step in $\mathtt{iHadoop}_a$ and $\mathtt{iHadoop}_{ac}$ writes its output to HDFS, and concurrently streams this output to the mappers of the following iteration. For PR and BPFS, with pre-specified number of iterations, outputs are never read back from HDFS in case of no failures. Skipping writing iteration's output to HDFS can significantly improve the performance. However, it has implications on the fault tolerance model of iHadoop. Checkpointing the output of the computation to HDFS every n iterations is one way to speed up an iterative application running time, mitigating the negative effect on fault tolerance. The experiments show that for PR, skipping writing to HDFS makes an iteration of $\mathtt{iHadoop}_a$ 15% faster on average. This reduces the normalized average iteration time of $\mathtt{iHadoop}_a$ presented in Figure 7(a) from 0.78 to 0.67. In $\mathtt{iHadoop}_{ac}$ the reduction reaches 31% per iteration. This reduces the normalized average execution from 0.60 to 0.43. Modifications are required to roll back iterative computations to the last saved state (i.e., the last committed output to HDFS) if a task fails. This kind of deployment can be practical for clusters with high reliability, or those that wish to minimize disk space usage or disk power consumption.

`iHadoop` applies the optimizations presented in this work to iterative applications only. We ran several non-iterative applications (e.g., word count, Pi Estimator, and grep) using `iHadoop` and compared the performance to `Hadoop`. The overhead of our implementation on non-iterative algorithms is negligible.

VII. Conclusion

MapReduce does not support iterative algorithms efficiently. Dataflow and task scheduling are unaware, and thus do not take advantage of, the structured behavior of iterative algorithms. iHadoop optimizes for iterative algorithms by modifying the dataflow techniques and task scheduling to allow iterations to run asynchronously. The MapReduce framework takes advantage of a large scale partitioned parallelism as it divides the input data into many input splits that are processed in parallel. Asynchronous iterations achieve significant performance gains since they introduce more parallelism on top of that already exploited by MapReduce; namely, they allow more than one iteration to run concurrently.

References

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment*, 2, August 2009.

[2] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[3] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proceedings of the seventh international conference on World Wide Web (WWW)*, 1998.

[4] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment*, 3, September 2010.

[5] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on PLDI*, 2010.

[6] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-Mapreduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[7] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In *NIPS*, 2007.

[8] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[9] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment*, 3, September 2010.

[10] Grzegorz Czajkowski. Sorting 1PB with MapReduce. http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html, 2008.

[11] Marc de Kruijf and Karthikeyan Sankaralingam. MapReduce for the Cell B.E. Architecture. Technical Report TR1625, Department of Computer Sciences, The University of Wisconsin-Madison, Madison, WI, 2007.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[13] Jaliya Ekanayake, Thilina Gunarathne, Geoffrey Fox, Atilla Soner Balkir, Christophe Poulain, Nelson Araujo, and Roger Barga. DryadLinq for Scientific Analyses. In *Proceedings of the Fifth IEEE International Conference on e-Science (E-SCIENCE)*, 2009.

[14] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a Runtime for Iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[15] John F. Gantz. The Diverse and Exploding Digital Universe, 2008.

[16] Hadoop. http://hadoop.apache.org/, May 2011.

[17] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[18] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2007.

[19] Matthias Jarke and Jurgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16, June 1984.

[20] Karthik Kambatla, Naresh Rapolu, Suresh Jagannathan, and Ananth Grama. Asynchronous Algorithms in MapReduce. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, 2010.

[21] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the Ninth IEEE ICDM*, 2009.

[22] Jon M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 46, September 1999.

[23] Jimmy Lin, Donald Metzler, Tamer Elsayed, and Lidan Wang. Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search. In *Proceedings of the 18th Text REtrieval Conference (TREC)*, 2009.

[24] LSST. http://www.lsst.org/lsst, May 2011.

[25] Mahout. http://mahout.apache.org/, May 2011.

[26] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.

[27] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. A Peer-to-Peer Framework for Supporting MapReduce Applications in Dynamic Cloud Environments. In *Cloud Computing: Principles, Systems and Applications*. 2010.

[28] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: a Universal Execution Engine for Distributed Data-Flow Computing. In *Proceedings of the 8th USENIX Conference on NSDI*, 2011.

[29] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a Not-So-Foreign Language for Data Processing. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, 2008.

[30] Spiros Papadimitriou and Jimeng Sun. DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining. In *Proceedings of the Eighth IEEE ICDM*, 2008.

[31] Sayan Ranu and Ambuj K. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In *Proceedings of the IEEE International Conference on Data Engineering*, 2009.

[32] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, 1979.

[33] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. FPMR: MapReduce Framework on FPGA. In *Proceedings of the 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2010.

[34] Nisheeth Shrivastava, Anirban Majumder, and Rajeev Rastogi. Mining (Social) Network Graphs to Detect Random Link Attacks. In *Proceedings of the IEEE 24th ICDE*, 2008.

[35] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2, August 2009.

[36] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.

[37] Abhishek Verma, Xavier Llorà, David E. Goldberg, and Roy H. Campbell. Scaling Genetic Algorithms Using MapReduce. In *Proceedings of the Ninth International Conference on Intelligent Systems Design and Applications (ISDA)*, 2009.

[38] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing Cloud Computing Hardware Reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[39] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, 2007.

[40] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[41] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLinq: a System for General-Purpose Distributed Data-Parallel Computing using a High-Level Language. In *Proceedings of the 8th OSDI*, 2008.

[42] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[43] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. iMapReduce: A Distributed Computing Framework for Iterative Computation. In *Proceedings of the 1st International Workshop on Data Intensive Computing in the Clouds*, 2011.

[44] Feida Zhu, Xifeng Yan, Jiawei Han, and Philip S. Yu. gPrune: a Constraint Pushing Framework for Graph Pattern Mining. In *Proceedings of the 11th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD)*, 2007.