Int J Parallel Prog (2016) 44:832–866
DOI 10.1007/s10766-015-0395-0

CrossMark

MapReduce Parallel Programming Model: A State-of-the-Art Survey

Ren Li[1] · **Haibo Hu**[2] · **Heng Li**[2] · **Yunsong Wu**[3] · **Jianxi Yang**[1]

Received: 20 October 2014 / Accepted: 20 October 2015 / Published online: 29 October 2015
© Springer Science+Business Media New York 2015

Abstract With the development of information technologies, we have entered the era of Big Data. Google's MapReduce programming model and its open-source implementation in Apache Hadoop have become the dominant model for data-intensive processing because of its simplicity, scalability, and fault tolerance. However, several inherent limitations, such as lack of efficient scheduling and iteration computing mechanisms, seriously affect the efficiency and flexibility of MapReduce. To date, various approaches have been proposed to extend MapReduce model and improve runtime efficiency for different scenarios. In this review, we assess MapReduce to help researchers better understand these novel optimizations that have been taken to address its limitations. We first present the basic idea underlying MapReduce paradigm and describe several widely used open-source runtime systems. And then we discuss the main shortcomings of original MapReduce. We also review these MapRe-

✉ Ren Li
renli@cqjtu.edu.cn

Haibo Hu
hbhu@cqu.edu.cn

Heng Li
liheng@cqu.edu.cn

Yunsong Wu
yswu@cqu.edu.cn

Jianxi Yang
yjx@cquc.edu.cn

[1] College of Information Science and Engineering, Chongqing Jiaotong University, Chongqing, China

[2] School of Software Engineering, Chongqing University, Chongqing, China

[3] College of Computer Science, Chongqing University, Chongqing, China

duce optimization approaches that have recently been put forward, and categorize them according to the characteristics and capabilities. Finally, we conclude the paper and suggest several research works that should be carried out in the future.

Keywords MapReduce · Hadoop · Cloud computing · Big data · Scalability

1 Introduction

The rapid development of the Internet has led to massive volumes and variety of data becoming available, as well as the rate at which the data are being generated is increasing exponentially [1]. These characteristics are acknowledged as essential of Big Data [2]. The computing capabilities of multi-core computers have become remarkably sophisticated, but inevitable bottlenecks in their performance and scalability still limit the possibilities for handling large-scaled data in a centralized context. The "scale-up" optimization strategy, in which computations performed by a single machine should be changed to a distributed computing context with a "scale-out" feature, has therefore been comprehensively acknowledged. It has now become particularly important to determine the effective way of achieving efficient and scalable storage and computation of big data, and this work represents a considerable challenge [3].

In recent years, cloud computing technologies have received a great deal of attention from researchers in the information technology industry and academia. The US National Institute of Standards and Technology has defined cloud computing as "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [4]. The development of cloud computing has been evolutionary rather than revolutionary, via a merging of several conventional technologies, including virtualization, grid computing, utility computing, and autonomic computing [5].

Of the cloud computing technologies that are currently in use, Google's MapReduce paradigm [6], which is built on the Google File System (GFS) [7], has become the prominent parallel programming model in the cloud computing community because of its simplicity, scalability, and fault tolerance [8]. Several open-source frameworks have been released to make MapReduce available for the public. And one of these, Hadoop MapReduce [9] well improves scalability of large-scaled data processing. It also enables programmers to focus on computational logic rather than those low-level programming details, such as data partitioning, task scheduling, load balance, and fault tolerance. Nowadays, MapReduce is extensively used to solve data-intensive problems in various fields, lots of algorithms that were originally designed for the single machine context have been parallelized to be suitable for MapReduce. For example, Yahoo uses MapReduce to meet its big data analysis requirements [10]. Apache Mahout Project [11] provides several MapReduce-based scalable machine learning libraries. Urbani et al. [12] developed a MapReduce-based inference engine WebPIE to implement scalable ontology reasoning for the Semantic Web [13]. Work [14] described the efficient Skyline query processing of massive data, and the method they used was also

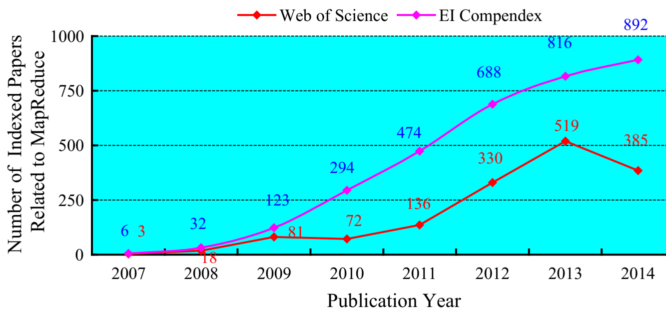

Fig. 1 The Number of indexed papers related to MapReduce from 2007 to 2014

based on MapReduce. Figure 1 shows the number of indexed papers related to the term "MapReduce" in Web of Science and EI Compendex databases where the X-axis denotes publication years from 2007 to 2014, the Y-axis represents the corresponding indexed number.

However, several inherent limitations seriously affect the efficiency of original MapReduce. For instance, the fundamental computing unit in MapReduce, a job, is performed in two phases: map and reduce, but it is difficult to implement multi-iteration algorithms in a single job. Hadoop provides a job chain mechanism, but running multiple MapReduce jobs is still computationally expensive. In addition, the batch processing characteristic of original MapReduce is notoriously difficult to use when handling large-scale data in real time and interactive context. Numerous optimization approaches have been released to tackle the challenges in recent years. Therefore, a systematic review of these novel MapReduce optimization solutions is much in demand.

Until now, several relevant works that survey MapReduce have been presented. For example, Doulkeridis et al. [15] reviewed the state-of-the-art in improving the performance of parallel query processing using MapReduce, although some important optimizations like hardware acceleration and performance tuning of MapReduce were not listed in detail. Li et al. [16] surveyed the distributed data management and processing approaches using MapReduce. However, their work focused on reviewing the high level languages and database-related operators for MapReduce, some extensions of MapReduce programming model were not discussed. In 2011, Lee et al. [17] gave a survey of MapReduce data processing, but some novel improvements and features of MapReduce runtimes such as YARN resource management framework were not described.

Different from above works, we intend to focus on the efficiency and flexibility improvements of MapReduce, and provide a more comprehensive review of state-of-the-art methods for optimizing the MapReduce programming model and its runtime system. We will first identify the major drawbacks of original MapReduce paradigm. We will then provide an in-depth analysis of the optimization approaches with aspect to their different objectives and capabilities such as job scheduling optimization, programming model extension and hardware-based acceleration, etc.

The remainder of this paper is organized as follows. Section 2 gives an overview of the basics of Google's MapReduce paradigm, its high-level abstractions, as well

as some well-known open-source implementations. Section 3 contains a discussion and analysis of the major drawbacks to MapReduce that affect its performance and efficiency. Based on that, Sect. 4 contains a detailed review of the state-of-the-art improvements that have been undertaken to MapReduce related to these drawbacks. In Sect. 5, we conclude the whole paper and suggest the future work that are required.

2 Basics of MapReduce Paradigm

As the basis of other runtime implementations, Google's MapReduce programming model was designed to process huge amounts of data in a cluster of commodity machines [18]. The key ideas behind the design of this model are the principle of "divide and conquer", and the strategy of "moving computing instead of moving data". To help programmers focus on implementing computational logic in a large distributed cluster, researchers also developed distributed and parallel computing functions in their MapReduce runtime that hide the programming details of load balancing, network communication and fault tolerance. However, the source code of their runtime system is not available to the public because of Google's privacy policy.

In this section, we first describe the basics of Google's MapReduce programming model and introduce some existing open-source implementations. We then present a short review of some well- known high-level abstractions of MapReduce. The characteristics of these MapReduce systems are also compared to help users to choose the appropriate tool for their particular requirements.

2.1 MapReduce Programming Model

The basic design idea of Google's MapReduce is inspired by the Map and Reduce functions in classical functional programming languages like Lisp [19]. In its master–slave architecture, data stored in the GFS are converted into key–value pairs and processed in a job consisting of a map phase and a reduce phase. A batch of jobs can also be formed into a chain to cope with complex computing tasks. Data must meet a basic requirement to be suitable for MapReduce computations, that the datasets should be decomposable into many small independent sub-datasets for processing in a particular task.

In a MapReduce job, the master node first partitions input data into M independent chunks (where M is the number of Map tasks) and passes them to the mapper nodes. Each map task is independently executed in a mapper node. Afterwards, in the map phase, each mapper accepts data chunks and then generates a series of intermediate key–value pairs according to a user-defined Map function. The MapReduce runtime system then automatically sorts and merges these intermediate key–value pairs depending on the key. The intermediate data with the same key are divided into R segments (where R is the number of reducer nodes) using a hash function. Finally, after being notified of the location of the intermediate data in the reduce phase, each reducer accepts a set of intermediate key–value pairs and merges all the data with the same key value, then generates a series of key–value pairs according to a user-defined Reduce function. Figure 2 illustrates the processes involved in one MapReduce job.

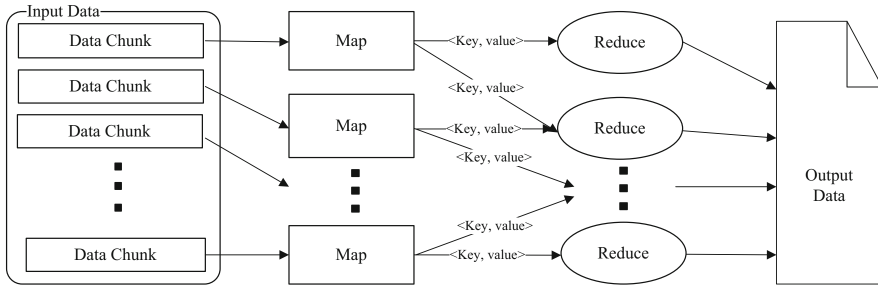

Fig. 2 Google's MapReduce paradigm

Table 1 Sample of word count pseudo-code that can be used in MapReduce

Map function pseudo-code	Reduce function pseudo-code
Map(String *key*, String *value*):	Reduce(String *key*, Iterator *values*):
//key: file name	//key: one word
//value: contents in one row	//values: a list of counts
for each word w in *value*:	int *result* = 0;
Emit_Intermediate(w, "1");	for each v in values:
	result += StringToInt(v);
	Emit(*key*, IntToString(*result*));

The formal expressions describing the Map and Reduce functions are given below, and in the $[v_i]$ represents a list of values with respect to k_2, $i \geq 0$. Programmers can follow the pseudo-code of Map and Reduce functions as shown in Table 1 to count the number of occurrences of words in a collection of files using MapReduce.

$$\text{Map} : \langle k_1, v_1 \rangle \rightarrow [\langle k_2, v_2 \rangle]$$
$$\text{Reduce} : \langle k_2, [v_i] \rangle \rightarrow [\langle k_3, v_3 \rangle]$$

As shown in Table 1, the Map function running in one map task takes a particular file or a portion of multiple files as input. Each time map task encounters word w, it produces intermediate key–value pairs in which w is the key, and the value (assigned to "1" in Table 1) is the number of occurrences of word$w$. The MapReduce runtime system then automatically divides these intermediate data into several partitions according to the number of reduce tasks, and each partition is assigned to the appropriate reduce task after the data are sorted to group all of the key–value pairs with the same key. Finally, the Reduce function takes the sorted data as its input, and for each value v it encounters, it sums all the counts produced for word w.

2.2 Open-Source Implementations of MapReduce

To date, several open-source runtimes that implement Google's MapReduce model have been presented to solve data-intensive computing problems for different usage scenario or deployment environments.

QTConcurrent [20] is a C++ library for multi-threaded applications within the QT project [109], providing functional programming style APIs for parallel processing, including a MapReduce implementation for shared-memory systems. Different from Google's MapReduce, QT Concurrent can only be executed in non-distributed environment. The number of threads used in a QT program is automatically adjusted depending on the number of processor cores available.

Phoenix [21] implements MapReduce by using multi-core chips and shared-memory multi-processors. It comprises of a parallel programming API and an efficient runtime that automatically manages thread creation, dynamic task scheduling, data partitioning, and fault tolerance for all of the processors. A C++ re-implementation of Phoenix, called Phoenix++ [22] has recently been released.

Disco [23], which utilizes Erlang language, is a lightweight, open-source framework for distributed computing, and is also built on top of Google's MapReduce paradigm. Profiling and debugging MapReduce jobs and random access to petabyte-scale data and auxiliary results are well supported with the help of the Disco Distributed File System. Many companies, including Nokia Research Center, employ Disco for large-scaled log analyses, probabilistic modeling, data mining, and full-text indexing [23].

Skynet [24] is a Ruby-based open-source implementation of Google's MapReduce framework. It is an adaptive, self-upgrading, fault-tolerant, and fully distributed system with no single point at which failure might occur. Unlike master–slave architecture used in other systems, Skynet has no special master server, and all the workers use the "peer recovery" strategy and message queue architecture to monitor the progress of the other worker-computers.

GridGain [25] provides a Java-based middleware for the in-memory processing of big data in distributed context. GridGain allows real-time in-memory processing of both transactional and non-transactional live data with very low latencies. Several novel features, such as distributed task sessions, checkpoints for long running tasks, early and late load balancing, and affinity co-location with data grids, have been integrated in GridGain, in which a task is split into multiple sub-tasks assigned to the available cluster nodes. The results of the sub-tasks are then aggregated and sent back to the user.

Another open-source MapReduce runtime, Twister [26], aims to provide efficient iteration computation in distributed in-memory environment. Twister uses publish/subscribe messaging infrastructure to handle communications, including transferring intermediate data between map and reduce tasks in distributed memory. It also uses a configuration phase for any proposed long-running map and reduce tasks, to eliminate the need for reloading static data for each iteration. A combine operation produces a collective output from all the reduce tasks. Moreover, Twister contains a task pool to avoid the need for tasks to initialize repeatedly.

Distinctively, Misco [27] implements a Python-based MapReduce framework for mobile systems. It has a polling-based task assignment mechanism and uses the HTTP protocol to transfer data between the Misco server and the workers. An Earliest Deadline First Scheduler and a Sequential Scheduler are used to schedule its applications and tasks, respectively. Furthermore, a UDP Server module contained in the Misco Server is used to monitor incoming worker logs using the UDP transport protocol.

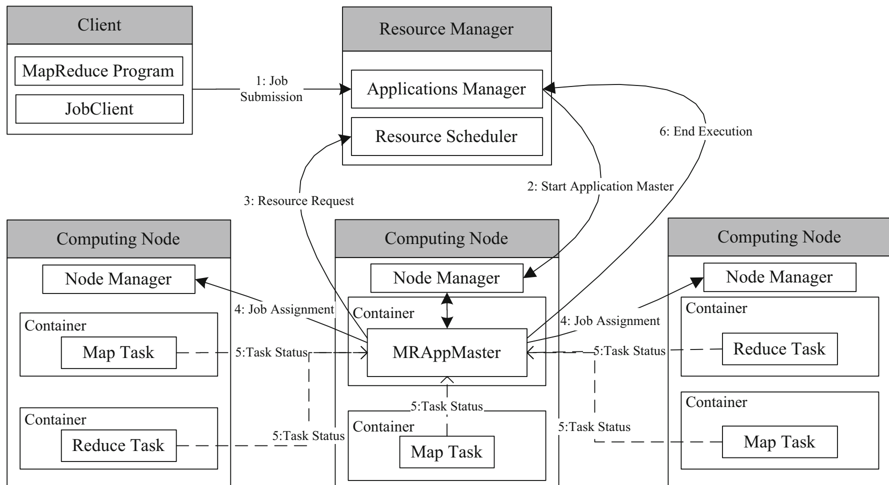

Fig. 3 Architecture of YARN-based Hadoop MRv2

Apache Hadoop is the dominant open-source MapReduce framework [9] which allows distributed processing of large datasets across clusters of commercial computers. The core components of Hadoop ecosystem include Hadoop Common, Hadoop File System (HDFS), Hadoop YARN, and Hadoop MapReduce. Hadoop Common provides the functional infrastructure for Hadoop framework. HDFS is an open-source implementation of GFS, and provides scalable distributed file system support. Hadoop YARN is a novel general framework for job scheduling and cluster resource management. YARN can serve as a runtime environment not only for MapReduce but also for Spark [65]. Generally, MapReduce is called MRv2 in Hadoop YARN framework.

In YARN, although the MapReduce programming model is used unchanged, the JobTracker and TaskTracker in Hadoop 1.0 are divided into two novel components. These are Resource Manager, which allocates global resources for all of the applications, and Application Master, which manages the job execution process for one particular application. Figure 3 illustrates the overall architecture and workflow of Hadoop MRv2. First, when YARN receives the submitted MapReduce application, the Application Manager component in Resource Manager calls the corresponding Node Manager to start Application Master (denoted as MRAppMaster in Fig. 3) in a Container. Then, Application Master requests computational resources from the Resource Scheduler component in Resource Manager, and asks the corresponding Node Mangers to start the jobs for this application. During executing, the task in each Container sends its running status to Application Master via RPC protocols.

2.3 High-Level Abstractions of MapReduce Model

When implementing complicated data analysis tasks in MapReduce frameworks mentioned above, programmers have to design several Map and Reduce functions separately, and determine how to group these functions to form a series of MapRe-

duce jobs. In order to further hide the programming detail of MapReduce, researchers have proposed several high-level abstractions, such as Sawzall [105], Apache Pig [102], Cascading [103] and Scalding [104].

Sawzall [105] is an abstracted type-safe script language designed for analysing massive individual log records in Google's MapReduce clusters. A general Sawzall script can be abstracted into two phases: a filtering phase and an aggregation phase. After the Sawzall interpreter is instantiated for each piece of input data, the filtering phase evaluates the analysis on each input record individually and emits the intermediate results to the aggregation phase. Then, based on a set of predefined aggregators (also called tables), the aggregation phase collate and reduce the intermediate to create the final results. Through using Sawzall, the resulting code is much simpler and shorter, by a factor of ten or more, than the corresponding C++ code in MapReduce. In 2010, the runtime of Sawzall which runs a Sawzall script once over a single input was open-sourced.

Motivated by Sawzall, Apache develops an open-source large-scaled data analysis platform Pig [102] which adopts a high-level SQL-like language Pig Latin. Pig provides a default MapReduce-based deployment model that executes ad-hoc computing tasks in a Hadoop cluster and HDFS installation after the Pig Latin programs being compiled into a series of MapReduce jobs. Furthermore, Pig Latin can handle nested data models and also is able to use operations that are commonly found in databases, such as *Group By*, *Join*, *Filter*, *Union* and *Foreach* etc.

Table 2 presents the Pig Latin code of word count sample. First, the Pig interpreter parses the Pig Latin command and verifies that the input are valid. Second, the Pig compiler builds a logical plan for the computational logic that the programmer defined. And then, the Pig compiler converts the logical plan into a physical plan to determine MapReduce job following lazy execution strategy. Finally, the corresponding jobs are executed automatically in Hadoop context. The computation flow of Pig's word count program is illustrated in the Fig. 4.

Cascading [103] is another notable high level abstraction framework for building data application on the top of Hadoop MapReduce. Cascading hides the topological structure and configuration of MapReduce from programmers to improve efficiency of complex business logic development. Cascading follows the "pipe and filter" strategy to define computational workflow before compiling and executing into Hadoop MapReduce.

Twitter's Scalding API [104] extends Java-based Cascading to enable MapReduce application development with promising Scala language which combines the features

Table 2 Sample of word count pseudo-code that can be used in Apache Pig

Pig Latin Code of Word Count using Apache Pig
A = load *Input_Data* as line:chararray;
B = foreach *A* generate flatten (TOKENIZE(line)) as word;
C = group *B* by word;
D = foreach *C* generate group, count(*B*)
dump *D*;

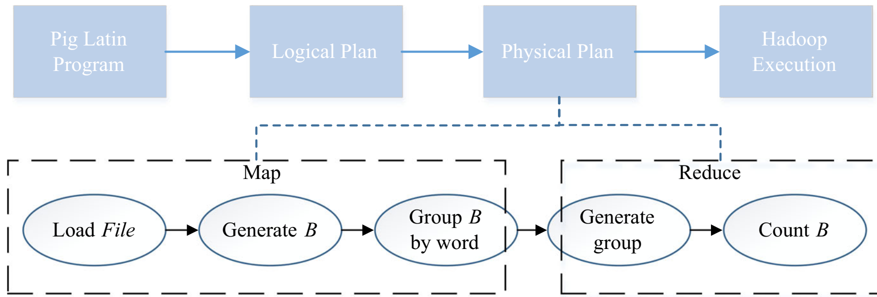

Fig. 4 The computation flow of word count in Apache Pig

of functional programming and object-oriented programming. Recently, Twitter also open-sources the Summingbird [106] library that allows programmers write MapReduce programs that look like native Scala or Java collection transformations and execute them on Scalding platform.

The main characteristics of these open-source MapReduce runtimes and high-level abstractions mentioned above are summarized in Table 3. The characteristics include the development and programming language, the supported operating system, and the deployment environment.

3 Shortcomings of Original MapReduce

As mentioned, MapReduce has been widely used to solve data-intensive problems in various fields because of its simplicity, fault tolerance and scalability. However, there is still some debates amongst academic researchers about its efficiency, even MapReduce has been criticised as a "major step backwards" in parallel data processing, compared with traditional RDBMSs [28]. Dean et al. addressed misconceptions about Google's MapReduce and discussed ways of handling the pitfalls related to, for example, heterogeneous systems, indices, and structured data and schemes [18]. But it is obviously that several pitfalls of MapReduce need to be addressed, as described below.

1. An efficient scheduling mechanism is critical to MapReduce runtime system because it often runs multiple jobs in a distributed context. However, the default scheduler in original Hadoop MapReduce assumes that the computing nodes in the clusters are homogeneous and that the estimated straggler tasks can be speculatively copied and re-executed by other idle nodes [29]. This means that a heavy I/O transfer load is inevitable, particularly when working under heterogeneous clusters. In addition, within a single MapReduce job, the reduce phase does not begin executing until all the map tasks are completed. This synchronization barrier seriously affects its performance. In the context of cloud computing data center, it is quite common for the computing resources available to MapReduce to be shared with multiple users. But, the Hadoop MapReduce assumes that all the computing

Table 3 Features of open-source implementations and abstractions of MapReduce

	Development language	Programming language	Deployment environment	Operating system
QT Concurrent [20]	C++	C++	Share–Memory System	Linux, Windows, Mac OS
Phoenix [21]/ Phoenix++ [22]	Java, C++	Java, C++	Share–MemorySystem	Linux
Disco [23]	Erlang	Python	Master–Slave Clusters	Linux, Mac OS
GridGain [25]	Java	Java	Master–Slave Clusters	Windows, Max OS, Linux
Skynet [24]	Ruby	Ruby	Peer–to–Peer Clusters	Linux
Twister [26]	Java	Java	Share–Memory System	Linux
Misco [27]	Python	Python	Master–Slave Mobile Systems	Linux
Hadoop MapReduce [9]	Java	Java, C++	Master–Slave Clusters	Linux
Apache Pig [102]	Java	Pig Latin	Master–Slave Clusters	Linux
Cascading [103]	Java	Java	Master–Slave Clusters	Linux
Scalding [104]	Scala	Scala	Master–Slave Clusters	Linux

resources are assigned to a single user, so the original job scheduling mechanism can scarcely adapt to shared MapReduce environments.

2. The simplicity of MapReduce programming model makes it relatively easy for programmers to use. Many complex computational tasks, especially the iterative computing algorithms used in data mining and graph analyses, are difficult to implement in a single MapReduce job, but running multiple jobs in Hadoop is computationally expensive [30]. Therefore, an extension of the MapReduce programming model and runtime is in great demand.
3. With the development of the Internet of Things, large-scale stream data will be generated rapidly by a range of sensors. Real-time computing requirements for high speed data streams pose significant challenges. The original MapReduce was designed for batch-oriented offline processing, which means that the data must all be copied to the distributed file system or distributed databases at the beginning or during the computing process. Therefore, original MapReduce model barely satisfies upcoming real-time or interactive processing requirements.
4. Commercial computers almost all currently have multi-core CPUs or high performance GPUs. However, all map and reduce tasks in the original MapReduce are linearly executed, so the hardware computing capability of each node is not completely used. On the other hand, most MapReduce implementations are also designed to be executed in single cluster environments, but many cloud computing data centers have been established around the world. Research is therefore required on how the original MapReduce paradigm can be extended to make it suitable for processing large-scale distributed data across multiple clusters.
5. MapReduce applications are typically used in a cluster environment consisting of large numbers of commercial computers. The complex configuration parameters and cluster setup details involved pose challenges to MapReduce participators. Therefore, some optimizations, such as those that simulate MapReduce contexts, are in great demand allowing the performance of MapReduce to be tuned and its dynamic running behaviour to be analysed.
6. In practice, MapReduce clusters are mostly deployed in cloud computing data centers in which data and computing resources are shared by multi-tenants. However, original MapReduce runtime only provides Token-based and Kerberos-based authentication mechanisms. Therefore, how to further protect large-scaled sensitive data and provide stronger authentication and authorization should be considered seriously. On the other hand, energy consumption often holds a large fraction of the total cost of MapReduce clusters. This brings a critical and challenging issue about how to decrease the usage of power and maintain the capability of MapReduce at same time.

4 Improvements of MapReduce

In this section, we review the state-of-the-art optimization approaches that have been proposed to address the aforementioned issues of MapReduce programming model and its runtime systems. We classify these approaches into following six aspects according to their different characteristics.

4.1 Optimizations of Job Scheduling

Although MapReduce model provides a simple programming interface for users, it does not contain any execution plan that specifies how jobs are executed in the nodes. The job scheduling mechanism of particular MapReduce runtime system seriously affects efficiency of MapReduce since all the functions such as task partitioning and resource allocation are integrated in. The default FIFO job scheduler in Hadoop MapReduce runtime assumes that the job submitted is executed sequentially under homogeneous cluster. However, it is very common that MapReduce is being deployed in heterogeneous environment, and the computing and data resources are shared by multiple users and applications. Recently, researchers have made lots of progress in the area of MapReduce job scheduling optimizations with respect to these two scenarios: shared and heterogeneous MapReduce environment.

4.1.1 Shared MapReduce Environment

Under shared MapReduce environment, how to allocate limited data and computational resources should be primarily considered. Assigning these resources equally to each user or application is the straightforward way. However, diversity of multiple jobs and features such as data locality are not well considered by this kind of fairness-based method. Therefore, Zaharia et al. [35] proposed a delay scheduling algorithm for addressing the conflict between data locality and fairness in a shared MapReduce cluster. The basic idea of the proposed algorithm is that a job that should be scheduled according to fairness is delayed for a small amount of time when it cannot be executed locally, letting other jobs launch instead. The delay scheduling algorithm is utilized in the Hadoop Fair Scheduler.

Besides taking advantage of data locality, Seo et al. [32] proposed a High Performance MapReduce Engine (HPMR) to reduce the amount of intermediate output of multiple jobs to shuffle as well. HPMR contains two optimization schemes, pre-fetching and pre-shuffling, to improve the degree of data locality and efficiency of data shuffling, respectively, for shared MapReduce environments. The prefetching schema is classified into types: the intra-block prefetching and the inter-block prefetching. In intra-block pre-fetching, only an input split or an intermediate output is pre-fetched, while the whole candidate data block is pre-fetched in inter-block pre-fetching. During pre-shuffling, HPMR looks over an input split before the map phase begins and predicts the target reducer where the key-value pairs are partitioned.

Meanwhile, some works improve the efficiency of job scheduling by estimating the execution status of multiple MapReduce applications. Sandholm et al. [31] introduced a total system efficiency metric for shared clusters on a Xen-virtualized infrastructure. This work dynamically adjusts the allocation of resources based on the average actual application performance ratio in shared environment. Sandholm et al. incorporated three user-assigned prioritization strategies into Hadoop MapReduce as well, one that prioritizes entire workflows, one that prioritizes different stages of a single workflow and one that detects and prioritizes bottleneck components within a workflow stage. As a result, users can change the priority of an application over time and assign different priority to different components under their limitation of total priority.

For the similar purpose, Polo et al. [33] presented a performance-driven co-scheduling mechanism for shared MapReduce environments that dynamically predicts the performance of parallel MapReduce jobs and adjusts the resource allocation. A job performance estimation algorithm for dynamically estimating the completion time of a MapReduce job was proposed. The allocation algorithm used in the task scheduler assigns free slots to each job depending on the results of the estimation, and organizes multiple jobs into an ordered queue based on the proposed scheduling policy. The authors also presented an application-centric MapReduce multi-job task scheduler to meet user-defined high level performance goals by exploiting the capabilities of a hybrid system [34]. By developing a prototype in a cluster of Cell/BE blades, this scheduler is capable of dynamically allocating resources to co-located MapReduce jobs based on their completion time goals.

LsPS [36] is an adaptive scheduling algorithm that uses knowledge of workload characteristics to tune the scheduling schemes for Heterogeneous MapReduce cluster, with important statistical information on job workloads for each user, including average task execution time, average job size and the coefficient of variation of job sizes, being monitored and gathered by the light-weighted historical information collector developed. When scheduling for multiple users, LsPS allocates slots according to their workload characteristics, and when scheduling for a single user, LsPS tunes the scheduling schemes for jobs based on that user's job size distribution.

4.1.2 Heterogeneous Environment

The slow tasks which are denoted as stragglers often prolong the execution time of MapReduce jobs. Effectively identifying stragglers in the heterogeneous cluster and speculatively scheduling them to other idle nodes is the straightforward way to decrease response time. Until now, several works, such as [37], [38], [39] and [29], improve the efficiency of job scheduling in heterogeneous context by making good use of the speculation strategy.

The Longest Approximate Time to End (LATE) scheduling algorithm [37] defines some static parameters like *SpeculativeCap* and *SlowTaskThreshold* to denote the number of speculative tasks that can be running at one time, and to determine whether a target task is slow enough to prevent needless speculation, respectively. This algorithm estimates the progress rate and the time to completion of each task according to the static parameters and the amount of time that the task has been running. By using LATE, when a node asks for a new task and the number of speculative tasks that are running is less than the denoted total number *SpeculativeCap*, the request is ignored if the node's total progress is below the proposed parameter *SlowNodeThreshold*. Otherwise, the tasks that are currently running but are not being speculated are ranked by their estimated completion times and a copy of the highest ranked task with a progress rate below *SlowTaskThreshold* is launched.

Inspired by an idea similar to the LATE algorithm, Chen et al. developed a Self-Adaptive MapReduce scheduling algorithm (SAMR) [38] and a History-based Auto-Tuning (HAT) MapReduce scheduler [39] for heterogeneous environments. Instead of using static parameters to find stragglers (as LATE does), Chen et al. utilized historical information updated after every execution to adjust the time weight of each stage of

the Map and Reduce tasks when estimating tasks execution times. In addition, SAMR dynamically identifies slow nodes and classifies them into sets of map slow nodes and reduce slow nodes. As a result, the backup map tasks are launched on the nodes which are fast nodes or reduce slow nodes.

As the SAMR algorithm does not take into account the fact that different types of jobs may have different map and reduce stage weights, Sun et al. [29] developed an Enhanced Self-Adaptive MapReduce scheduling algorithm (ESAMR) to improve the speculative re-execution of slow tasks. ESAMR differentiates historical stage weight information on each node and divides them into k clusters using a k-means clustering algorithm. ESAMR estimates the execution time of the running tasks according to the cluster's weights being classified.

In addition, because of the limited network bandwidth in clusters, some studies tried to optimize job scheduling by enhancing data locality for heterogeneous MapReduce clusters. For example, Zhang et al. [40] developed a data-locality-aware scheduling method that addresses the data locality problem that occurs in heterogeneous MapReduce environments. After receiving a request from a requesting node, this method preferentially schedules a task with input data stored on the requesting node. If no such task exists, the method selects a task with input data stored nearest to the requesting node, and then decides whether to schedule the task to the requesting node or to reserve the task for the node storing the input data by making a trade-off between waiting time and transmission time at runtime.

MapReduce with Access Patterns [43] is a combination of data access semantics and the MapReduce programming framework that has been used in High Performance Computing (HPC) analytics applications. It contains a data-centric scheduler to increase the performance of the HPC analytics MapReduce programs by maintaining data locality. In this scheduling algorithm, a virtual split-based approach is used to assign all the independent data chunks on a data node to local tasks and to avoid data transfers completely. A weighted-set cover-based approach was also designed to select data nodes for scheduling Map tasks with multiple dependent chunks.

Bu et al. [45] developed a task scheduling system to mitigate interference while preserving the task data locality in a virtual MapReduce cluster. In order to estimate the task slowdown caused by interference in virtual MapReduce cluster, an exponential interference prediction model is presented and used in their Dynamic Threshold policy. In addition, Bu et al. [35] improved the Delay Scheduling algorithm by adjusting delay intervals of ready-to-run jobs in proportion to the input size.

To take full advantage of the feature of data replication in HDFS, Yang et al. [42] developed a Data-replica scheduler that improves the performance of MapReduce task scheduling and data allocation in heterogeneous clusters. In this approach, after detecting the location of a free slot, the scheduler determines whether the data node is fast or slow according to the proposed fast-datanode queue. If the node is considered to be fast, an unprocessed block is read, otherwise the scheduler reads the data block from the head of the proposed undo-blockInfo table. The fast-datanode queue, the undo-blockInfo table, and the slow-data-etime queue (which is ordered by the predicted completion time for the processing tasks) are updated after the free slot has been processed.

Ahmad et al. [41] analysed the key reasons for MapReduce's poor performance on heterogeneous clusters and proposed a load balance optimization approach called Tarazu which consists of three components: Communication-Aware Load Balancing (CALB), Communication-Aware Scheduling (CAS), and Predictive Load Balancing (PLB). CALB regulates the use of remote Map tasks depending on whether Map or Shuffle is likely to be in the critical path, CAS determines how many remote tasks are needed and when to execute them in the task-steal mode, and PLB achieves better load balance in the Reduce phase by skewing the intermediate key distribution among the Reduce tasks depending on the types of nodes on which the reduce tasks run.

Table 4 summarizes above job scheduling optimizations according to their scenarios and used optimization strategies. We observe that data locality, speculative execution and dynamic performance estimation are the most commonly used job scheduling optimization strategies for both shared and heterogeneous MapReduce clusters.

4.2 Optimizations of MapReduce Programming Model

The simplicity of Google's MapReduce programming model makes it quite prevalent in the area of big data processing. However, programmers have to define a series of Map and Reduce functions to implement complex computational logic due to the limited operators and fixed workflow of original MapReduce. In this section, we review these flexibility optimizations with respect to extension of MapReduce programming model and discuss these workflow improvements for iteration computing.

4.2.1 Extension of MapReduce Programming Model

The Map–Reduce–Merge framework [46] extends the original MapReduce model with a Merge phase to support relational algebra and to implement several join algorithms for multiple heterogeneous datasets. This framework uses a coordinator to manage two sets of mappers and reducers. After these tasks are completed, the coordinator launches a set of mergers that read the output from selected reducers and merges them with user-defined logic. The novel programming model used in the Map–Reduce–Merge framework is shown below, in which α, β, and γ are dataset lineages, k is a key, and v represents the value entities.

$$\begin{aligned}
&\text{Map} : \langle k_1, v_1\rangle_\alpha \rightarrow [\langle k_2, v_2\rangle]_\alpha\\
&\text{Reduce} : \langle k_2, [v_2]\rangle_\alpha \rightarrow \langle k_3, [v_3]\rangle_\alpha\\
&\text{Merge} : \langle\langle k_2, [v_3]\rangle_\alpha, \langle k_3, [v_4]\rangle_\beta\rangle \rightarrow [\langle k_4, v_5\rangle]_\gamma
\end{aligned}$$

Map–Join–Reduce [47] extends original MapReduce model to meet the demand for efficiently joining multiple datasets. To add to the mapper and reducer, the authors of [47] developed a novel filtering-join-aggregation model that adds a third joiner executing function to a MapReduce job. A one-to-many shuffling strategy that efficiently shuffles each intermediate key–value pair generated by the mapper to many joiners at one time was also introduced.

Table 4 Summarization of job scheduling optimizations for shared and heterogeneous MapReduce

	Scenarios		Used optimization strategies and methods				
	Shared	Heterogeneous	Locality	Shuffling	Speculation	Exec. Estimation	Others
Sandholm et al. [31]	✓					✓	Prioritization
Seo et al. [32]	✓		✓	✓			
Polo et al. [33,34]	✓					✓	
Zaharia et al. [35]	✓		✓				Fairness
Yao et al. [36]	✓	✓				✓	
Zaharia et al. [37]		✓			✓		Static Param.
Chen et al. [38,39]		✓			✓	✓	
Sun et al. [29]		✓			✓	✓	
Zhang et al. [40]		✓	✓				
Ahmad et al. [41]		✓					Load Balance
Yang et al. [42]		✓					Data-Replica
Sehrish et al. [43]		✓	✓				
Bu et al. [45]		✓	✓			✓	

Tuple MapReduce [48] is a model that extends MapReduce to improve parallel data processing tasks using compound records, optional in-reduce ordering, or inter-source datatype joins. Instead of computing a key–value pair in the same way as the original MapReduce, Tuple MapReduce processes a raw n-sized tuple and includes a group-by clause before the reduce phase. The Map function in the Tuple MapReduce model takes a tuple as an input of types $(i_1, \ldots, i_m)$ and produces a list of other tuples as the output $list(v_1, \ldots, v_n)$, where the first g fields in the *list* are used to group-by and the first s fields are used to sort the fields. The Reduce function then takes as its input a tuple of size g, $(v_1, \ldots, v_g)$, and a list of tuples $list(v_1, \ldots, v_n)$. Finally, a list of tuples $list(k_1, \ldots, k_l)$ is produced by the Reduce function.

Vu et al. [49] proposed cHadoop, which supports continuous MapReduce jobs. A *carry* operation is added to the reduce phase to re-inject the data generated as the output of the map phase to the next execution. In the cHadoop framework, two components, named the Continuous Job Tracker and the Continuous NameNode, which were extended from the Job Tracker and the Name Node in Hadoop MapReduce, respectively, were developed to manage and execute continuous jobs, respectively.

Another MapReduce model extension is XMR [50], which features a hierarchical reduce phase. In order to tune the performance, the authors of [50] studied the parameters used in MapReduce and set minimum, maximum, and average values for each parameter. They also developed a data redistribution algorithm to identify the high-performing nodes and to reorganize the HDFS file fragments according to the computing ratios. In this method, the number of map and reduce slots is effectively managed to decrease the execution time. A hybrid routing schedule shuffle phase to define the scheduler tasks, to decrease the level of memory management required, is developed as well.

4.2.2 Improvements of Iterative Computing

As mentioned above, the fixed workflow and HDFS-based intermediate data storage mechanism of original Hadoop MapReduce cause serious performance degradation when executing a series of jobs iteratively. Several studies as described below try to solve the bottlenecks of iterative computing.

Haloop [51] is a modified version of Hadoop MapReduce designed to support iterative computing. Besides using several novel programming interfaces, Haloop contains a new loop control module in the master node, which iteratively starts MapReduce jobs in a loop until a fixed point has been reached. It also contains a novel task scheduler that takes advantage of data locality and provides a data caching and indexing mechanism to improve the iterative computing performance.

In order to overcome global synchronization overheads in large-scale iterative computing contexts, Kambatla et al. [52] developed an extension of MapReduce model by adapting locality-enhanced partitioning, partial synchronization, and eager scheduling optimization techniques.

iHadoop [53] schedules iterations asynchronously and connects the output of one iteration to the next, allowing both to process data concurrently. In addition, iHadoop concurrently checks the terminations that occur in the background against the asynchronous iterations.

Pipelined-MapReduce [54] improves MapReduce programming model that allows direct data transfer through a pipeline between n Map and Reduce tasks. Instead of determining intermediate data transfers through Task Trackers, the mappers in Pipelined-MapReduce determine which reduce tasks the intermediate data should be sent to, and send the data through a TCP socket directly.

MapCombine [55] contains a novel controller component to schedule the iterations efficiently and avoid re-initializing the runtime environment. The combine phase of the original MapReduce model is modified so that the static data can be cached to balance the workload of a computing node and solve the problem of data reloading. Moreover, MapCombine contains an interaction layer that is responsible for fault tolerance, downtime recovery, and communication between the controller and the combiners.

The iMapReduce framework [56] is another distributed computing framework for implementing iterative algorithms. This system follows a persistent task strategy, and the entire iterative iMapReduce process is implemented in one single job, with all Map and Reduce tasks continuing to be executed until the master has checked that the termination conditions are satisfied. iMapReduce uses a one-to- one socket connection to directly pass status data from the Reduce task to the Map task when starting the next iteration, to avoid the unnecessary shuffling of data in the iterative computing system. Map tasks in iMapReduce can also be asynchronously executed because the Reduce tasks immediately send the data they have produced through the persistent socket connection.

Twister4Azure [58] is a distributed decentralized iterative MapReduce runtime for the Windows Azure Cloud. It uses a Merge task after the reduce phase to determine the terminations of loops that take all of the Reduce outputs and broadcast data for the current iteration as inputs. To avoid unnecessary static data reloads and transfers between iterations, Twister4Azure supports three types of data caching approaches, instance storage based caching, direct in-memory caching, and memory- mapped-file based caching. The authors also developed a cache-aware scheduling algorithm to schedule new MapReduce jobs through Azure queues.

It is noteworthy that, Spark [65], as an emerging distributed in-memory computing framework, improves the efficiency of iterative computing through building on top of a high level abstraction called Resilient Distributed Dataset (RDD) [57]. Moreover, Spark provides more than 80 operators such as *map*, *filter*, *reduceByKey*, *join*, *reduce* and *collect* and classifies them as Transformations and Actions operators. Instead of storing intermediate data into distributed file systems, the runtime of Spark efficiently manages the lineage of read-only RDDs in memory and allows to pipeline several transformations based on the Lazy computing strategy.

Currently, Spark has become the top level project in Apache. Programmers can write Spark applications in Java, Scala or Python and deploy them on Hadoop, Mesos, or in the cloud to access diverse data sources like HDFS, Cassandra, HBase and S3. Within Spark, some high-level tools such as Spark SQL, GraphX and Spark Streaming are provided to implement SQL query integration, graph and stream processing of big data, respectively.

Table 5 shows the summarization of MapReduce programming model extension and iteration computing optimization.

Table 5 Summarization of optimizations of MapReduce Programming Model

	Data model	Operations	Key optimizations
Map–Reduce–Merge [46]	KV pairs	Map, Reduce, Merge	Add a Coordinator module
Map–Join–Reduce [47]	KV pairs	Map, Join, Reduce	Filtering-join-aggregation model
Tuple MapReduce [48]	N-sized tuple	Map, groupBy, Reduce	Novel groupBy scheduling mechanism
cHadoop [49]	KV pairs	Map, Reduce, Carry	Continuous Job Tracker & NameNode
XMR [50]	KV pairs	Map, Hierarchical reduce	Data redistribution & Routing schedule
Haloop [51]	KV pairs	Map, Reduce	Loop control & data caching, indexing
Kambatla et al. [52]	KV pairs	Map, Reduce	Locality-enhanced partitioning, partial synchronization, and eager scheduling
iHadoop [53]	KV pairs	Map, Reduce	Schedules iterations asynchronously
Pipelined-MapReduce [54]	KV pairs	Map, Reduce	Direct data transfer via TCP socket
MapCombine [55]	KV pairs	Map, Reduce	Novel controller component
iMapReduce [56]	KV pairs	Map, Reduce	Direct one-to-one socket connection
Twister4Azure [58]	KV pairs	Map, Reduce, Merge	Merge mechanism and data caching
Spark [65]	RDD	Transformations and Actions	In-memory process of RDD DAG

4.3 Real-Time Support

Since original MapReduce model is couple with distributed file system, such as GFS or HDFS, it lacks efficient support mechanism for real-time processing. In recent years, several solutions which take high-speed stream as input data source have been released to deal with real-time processing issues of original MapReduce.

The Hadoop Online Prototype (HOP) [59] extended original MapReduce to support long-running jobs through online aggregation of, and continuous queries, on streaming data. Within a job, HOP directly pipelines the output of the Map function to the Reduce task using a TCP socket when the intermediate data buffer of the Map function reaches a threshold size. Similarly, the output of a job in HOP can be directly pipelined to the Map task of the next job. Using this pipelining strategy and the job progress metric

developed, HOP gives an approximate answer in online aggregation and continuous query situations.

Böse et al. [60] developed an online MapReduce framework using shared-memory architecture, to allow the efficient mining of large data streams. In this framework, the Map tasks read input data from a file or from data streams and generate output pairs periodically and as events. A generated pair is assigned to the input queue of the corresponding reducer using a hash map mechanism. A collector component is utilized to compute the preliminary or final result used in convergence estimations and there is a visualization option.

By constructing an experimental evaluation of Hadoop MapReduce in the Amazon EC2 cloud, Phan et al. [61] identified key factors that affect real-time scheduling in MapReduce, and formulated the problem as a constraint satisfaction model. The authors of [61] developed an enhanced MapReduce execution model and a range of heuristic techniques for online scheduling using hierarchical scheduling, real-time virtual machines, and probabilistic models, based on this model.

MiscoRT [62] is a mobile MapReduce framework that was developed to support the execution of distributed applications with real-time response requirements on smart phone networks. It extends the Misco system with two levels of schedulers, called the Application Scheduler and the Task Scheduler. The Application Scheduler determines the order in which the applications are run, based on the urgencies and time constraints that are applicable, and it estimates the execution times using an analytical model. The Task Scheduler schedules tasks dynamically and uses the measured laxity values of the tasks to adjust the scheduling order. Additionally, node failures are also considered in both components.

Peng et al. [63] extended the original MapReduce model to support real-time analytics processing. In their proposed framework, the shuffle and sort phases of MapReduce are removed. A timestamp is integrated into the key part of the key–value model input to support the necessity of a real-time data stream. The authors used a Chord based on the JOL rule language to manage the input data stream. Cassandra was chosen for persistent key–value storage in the system.

RTMR [64] is a MapReduce-based large-scale data processing approach for high-speed data streams. RTMR pre-processes historical data to generate intermediate results that are distributed (cached) to the local disk of each working node according to the hash results for the key, to decrease the overheads involved in repeated data loading and computing. Based on this, the map phase in RTMR distributes the input data stream to the appropriate worker node and uses a local pipeline strategy, which is controlled by the system parameters and thread pools to optimize the CPU utilization rate, to transfer the intermediate data asynchronously between the Map and Reduce tasks. The authors also modified the in-memory and disk storage data structures and improved the intermediate results read/write strategy using the overhead estimation and replacement algorithm they developed, to optimize the access performance of local intermediate data.

On the other hand, to optimize the efficiency of incremental processing that is maintaining a very larger repository of Web documents and processing small updates concurrently, Peng et al. [44] from Google designed the Percolator system which provides two main abstractions: ACID transactions over a random-access repository and

observers, a way to organize the workflow of incremental computation. The structure of Percolator system in the cluster consists of three components: a Percolator worker, a BigTable tablet server and a GFS chunk server. During computing, all observers are running in the Percolator worker which scans BigTable for changed columns and invokes the corresponding observers as a function call in the worker process to perform transactions by sending read/write RPCs to BigTable tablet servers. Moreover, two services: the timestamp oracle and the lightweight lock service are integrated in Percolator to implement snapshot isolation and improve efficiency of notifications.

Similarly, in order to provide low-latency querying service as Google's Zeitgeist system does, Akidau et al. [107] designed another programming model and distributed framework called MillWheel with the capability of scalable and fault-tolerant stream processing. The basic workflow of MillWheel can be seen as a directed compute graph consisting of several user-defined transformations on input data. Each of these transformations, which are also called *computations*, can be parallelized across an arbitrary number of machines based on (key, value, timestamp) triples. Specifically, since data arrival time does not strictly correspond to its generation time in a distributed system with inputs from all over the world, MillWheel includes a timestamp-based Low Watermark mechanism to distinguish the latency and completeness of incoming data.

4.4 Hardware Acceleration

As discussed in Sect. 3, original MapReduce is designed for distributed computing in a cluster of commodity computers, the computing capabilities of hardware such as multi-core CPUs and GPUs are not adequately utilized. This section reviews these optimization approaches which extend original MapReduce to various hardware architectures. We classify them into two aspects: processor-level and cross-clusters hardware acceleration according to their different characteristics.

4.4.1 Processor-Level Hardware Acceleration

Employing the computing capabilities of multi-core CPUs is the most common way for parallel computing. Some works, such as [21], [67] and [73], implemented Google's MapReduce programming model in multi-core CPUs context.

In addition to the features offered by Phoenix [21], MATE [67] provides a generalized reduction API for the MapReduce model. Both Map and Reduce phases were combined into a Reduction function that processes split data, defined by the splitter function, and updates the intermediate results to give reduction objects. A Combination function was also developed to combine the final results from multiple copies of the reduction objects into one object. The MATE runtime, which has the same scheduling strategy and fault tolerance mechanism as Phoenix, has two types of temporary buffers, a reduction-object buffer and a combination buffer, which allocate each thread using different splits and store the intermediate output results from each stage.

Tiled-MapReduce [73] is an extended MapReduce model for shared memory multicore platforms that is based on a tiling strategy. It decomposes a large MapReduce

job into a number of independent small subjobs that are then iteratively processed all at one time. In so doing, execution workflow in the general MapReduce model is modified to give four phases, Map, Combine, Reduce, and Merge. The Map and Combine phases within a subjob are iteratively executed to generate partial results. The Reduce phase is then used to process the partial results of the iterations rather than the intermediate data. An Iteration Window approach is used in the runtime of the Tiled-MapReduce model to exploit the cache hierarchy in a multi-core platform efficiently. Optimized systems were also developed to improve the memory, cache, and CPU efficiency.

Comparing with multi-core CPUs, GPUs have an order of magnitude higher computation power and memory bandwidth. Several GPUs-based MapReduce implementations and optimizations have been proposed to take full advantage of the performance of GPUs.

For example, Stuart et al. [68] developed a multi-GPUs MapReduce implementation for efficient volume rendering. The workflow in this system takes advantage of NVIDIA GPUs, and consists of Map, Partition, Sort, and Reduce stages, the partial reduce and combine phases being omitted. To overcome the I/O transfer bottleneck, the runtime handles volume data in a streaming manner rather than by storing the intermediate key–value pairs and final values on a disk.

StreamMR [70] is an OpenCL-based MapReduce framework for AMD GPUs. Since atomic operations can cause severe performance degradation in AMD GPUs, StreamMR uses an atomic-free mechanism that maintains a separate output buffer for each workgroup in global memory to allow efficient output and pre-processing to be achieved. StreamMR also maintains a hash table for each wavefront, to group the intermediate results.

Chen et al. [71] developed an accelerated GPU-based MapReduce implementation. To avoid storing intermediate data in the memory in the device, they used a reduction-based approach that encapsulates intermediate key–value pairs into reduction objects and performs the reduction process in the shared GPU memory. The size of the shared memory space is often insufficient for this to be achieved, so a memory hierarchy was developed for storing the reduction objects on both the shared memory and the memory of the device. To balance the memory overhead and the locking costs, the authors further developed the multi-group scheme that partitions each thread block into multiple groups, so that each group has its own copy of the reduction object.

Grex [72] is a MapReduce framework designed to allow general purpose GPUs to be used for parallel data processing. Its workflow consists of five execution stages, parallel split, map, boundary merge, group, and reduce. In the first parallel split stage, instead of splitting the input data sequentially, Grex assigns one unit of data to each thread and uses the parallel prefix sum algorithm to compute the token for each key in parallel, based on the temporary data structure KeyRecord developed. To overcome problems with data skewing and load balancing, the split data are distributed to the subsequent Map tasks in parallel, and each working thread computes a unique address to write the intermediate key–value pairs to, so that the use of locks and atomic operations to synchronize the thread with other threads can be avoided. Grex assigns an equal number of intermediate data to reduce tasks, again to overcome problems with data skewing and load balancing. It uses a lazy emit strategy to decrease the

overhead involved in handling intermediate data in terms of computation and global memory usage, and uses shared memory, texture cache, and constant memory in the GPU memory hierarchy to decrease delays accessing memory.

Besides the above approaches, some works implement MapReduce model in coupled CPU and GPU architecture. For instance, MapCG [66] allows source code level portability between a CPU and a GPU. It consists of a MapReduce-based high-level programming language and a runtime for specific hardware architectures. Intermediate key–value pairs in MapCG are copied values and are never sorted, unlike what happens in Phoenix [21]. Two lightweight memory allocators on the CPU and GPU were developed to deal with the performance bottleneck in Phoenix caused by massive requests for malloc() functions. A hash table structure in MapCG groups the key–value pairs on the GPU.

Mars [69] is a MapReduce framework that can run on NVIDIA GPUs, AMD GPUs, multi-core CPUs, and Hadoop-based distributed systems. Mars workflow consists of three loosely coupled stages, Map, Group, and Reduce. The Group stage is designed to group the Map outputs by key. The Mars scheduler that runs on the CPU schedules Map and Reduce tasks to GPU threads. The sizes of the outputs from the Map and Reduce stages are unknown, and conflict may occur when multiple threads each try to write the results to the shared output array, so a lock-free scheme was developed and applied to both the Map and Reduce stages. The Map–Reduce–Count step in this scheme computes the number of intermediate results, the total sizes of the intermediate keys, and the corresponding values. Based on this, a prefix sum and an array of writing locations are generated to represent the location of the output array for each thread. A Rapid Group and a skew handling scheme were also developed, to optimize the sort operation in the Group stage and the workload distribution in the Reduce stage, respectively. Furthermore, Mars enables the integration of GPU-accelerated code to distributed environment, like Hadoop, with the least effort.

Cell processor is another hardware architecture used for MapReduce optimization. A MapReduce runtime for the Cell/BE architecture was developed by the study [74]. Unlike the original MapReduce model, this runtime consists of five stages, Map, Partition, Quick-sort, Merge–sort, and Reduce. The Map stage executes the user-specified Map function in the SPEs, and produces a set of keys containing pointers to the corresponding values. The Partition, Quick-sort, and Merge–sort stages then group identical keys together to produce a set of partitions sorted by the keys, on both the PPE and the SPEs. Finally, the Reduce stage applies the user-defined Reduce function on the SPEs to produce one logical output array of key–value pairs. In this implementation, the whole computational procedure in the runtime is designed to execute in 20 threads, with one PPE main thread spawning all of the other threads, eight SPE worker threads performing the five stages described above, eight PPE scheduler threads notifying the appropriate SPE when a work unit is ready for prefetching, two PPE worker threads performing the input data partitioning and final merge–sort processes, and one PPE event thread responding to output buffer memory allocation requests and controlling the execution flow.

Rafique et al. [75] developed a MapReduce system for asymmetric clusters of asymmetric multi-core processors and general-purpose processors. A general-purpose server with multi-core x86 processors and a large amount of memory was designed

to be the manager of their proposed architecture, and this server is responsible for the dynamic scheduling of jobs, the distribution of data, and allocating work to the Cell-based drivers or compute nodes, such as SONY PS3 and IBM QS20 systems equipped with the cell processors. The authors also developed a streaming approach for splitting data into work units, to fit the in-cores in the computing nodes, using various optimization techniques, such as prefetching, double buffering, and asynchronous I/O.

The work in [76] proposed another MapReduce runtime system based on Cell architecture, and this was derived from the system developed by [74]. The modified execution scheme consists of four stages, Map, quick-sort, merge–sort, and Reduce, with partitioning being performed implicitly in the Map stage. The modified run-time uses the main thread and the scheduler thread in an event-driven controller to achieve task initiation, scheduling, and workflow control on the PPE instead of using 12 threads. The performance and scalability of the original system were also improved in several additional ways, including adding implicit partitioning, quick sorting, memory management, and execution schemes.

In order to take advantage of the computational performance of Intel Xeon Phi coprocessor, Lu et al. [108] presented a MapReduce optimization named MRPhi sharing similar idea with Phoenix++. Within this system, two technologies from on Phoenix++ are adopted, which are efficient combiners and different container structures. As Xeon Phi features with wide 512-bit VPUs on each core which doubles the vector width compared with other Intel Xeon CPUs, MRPhi implements map phase in a vectorization friendly way to assist the auto-vectorization and take advantage of VPUs in Xeon Phi. In addition, since the auto-vectorization often fails due to the complex logic, Lu et al. employs the SIMD parallelism to implement hash computation manually. To better utilize the hyper-threading capability of Xeon Phi and improve the resource utilization, user-defined map and reduce phases are pipelined by using the MIMD threads. Furthermore, to deal with the issue of large local arrays, Lu et al. employs low overhead atomic operations on the global array, and the cache efficiency can be improved as well because of the coherent L2 caches with ring interconnection.

4.4.2 Clusters-Based Hardware Acceleration

Hadoop On the Grid (HOG) [77] extended Hadoop MapReduce to execute on the Open Science Grid in the United States. HOG consists of a grid submission and execution component, an across grid HDFS, and an across grid MapReduce. Grid submission and execution component, which is based on Condor and GlideinWMS, is used to manage the submission and execution of the Hadoop worker nodes and to allocate the nodes at remote sites. The master server in the HOG system resides on a stable central server. Failures on the grid are common, so the time between the heartbeat messages is lower in this system than in other systems, and the rack awareness strategy of HDFS was extended to become the site awareness in HOG. The MapReduce job tracker is deployed on the central server, and its communication with task trackers is based on HTTP over the WAN, to make it suitable for the grid context.

Heintz et al. [78] developed a cross-phase MapReduce system to overcome the limitations of Hadoop MapReduce when processing large-scale distributed data and using large amounts of computational resources. They proposed a Map-aware Push

technique to hide latency and enable dynamic feedback between the push and map phases. The better performing nodes and those with faster links can process more data in the runtime using this method than using other methods. To eliminate the impacts of mapper–reducer link bottlenecks, the authors also developed a Shuffle-aware Map approach that includes a shuffle-aware scheduler to feed back the cost of a downstream shuffle into the map process, to allow the map phase to be altered in an appropriate way.

G-Hadoop is another MapReduce framework that allows large-scale distributed computing across multiple clusters [79]. This system replaces the native HDFS the Gfarm globally distributed file system, and uses the Torque Resource Manager as the distributed resource management system for each High End Computing cluster, to allow datasets to be shared across multiple sites. G-Hadoop architecture is also based on the master–slave model. The master node, which is installed at a central organization, is responsible for job splitting, task assignment, and metadata management, and consists of a Metadata server from the Gfarm system and a modified JobTracker based on the data-aware scheduling system. The slave node, which is sent to each participating cluster, is designed to perform the TaskTracker, JobTracker, I/O Server, and Network share functions. A novel job execution flow, based on the system just described, was also developed for G-Hadoop.

In order to implement efficient data-intensive computing in distributed clusters, Mantha et al. [80] proposed Pilot-MapReduce which is an extension of the MapReduce runtime framework based on the Pilot abstraction model and enables the separation of resource management and the application of MapReduce in general-purpose distributed infrastructures. The architecture of the proposed Pilot-MapReduce framework consists of the MapReduce-Manager, the Pilot API, and several Data Pilot and Compute Pilot on different resources. Among them, the MapReduce-Manager is responsible for orchestrating the resource pool and managing the entire MapReduce computing process. The Pilot API is used as an abstraction for compute and data resources, as well as managing Data Units and Compute Units where the Map and Reduce tasks worked. In addition, the Pilot-MapReduce supports three different distributed MapReduce topologies: local, distributed, and hierarchical.

Instead of adapting the master–slave architecture, as in original MapReduce, P2P-MapReduce [81] is a peer-to-peer MapReduce framework that provides a more reliable solution for managing node churn, master failure, and job recovery in dynamic cloud infrastructures. Master nodes and slave nodes in P2P-MapReduce form logical peer-to-peer networks in which master nodes are responsible for receiving job requests from user nodes, managing and executing recovery jobs, and coordinating masters and slaves. Using the proposed Task Managers, each slave executes the tasks assigned according to the lowest workload strategy.

Table 6 shows the corresponding hardware architecture used for improving the efficiency of original MapReduce.

Table 6 Summarization of hardware-based acceleration for MapReduce

	Multi-core CPUs	GPUs	Co-processors	Cell	P2P	Grid	Clusters
Phoenix [21]	✓						
MATE [67]	✓						
Tiled-MapReduce [73]	✓						
Stuart et al. [68]		✓					
StreamMR [70]		✓					
Chen et al. [71]		✓					
Grex [72]		✓					
MapCG [66]	✓	✓					
Mars [69]	✓	✓					
de Krujif et al. [74]				✓			
Rafique et al. [75]				✓			
Papagiannis et al. [76]				✓			
Lu et al. [108]			✓				
HOG [77]						✓	
Heintz et al. [78]							✓
G-Hadoop [79]							✓
Mantha et al. [80]							✓
P2P-MapReduce [81]					✓		

4.5 Performance Tuning of MapReduce

Performance Tuning of practical MapReduce clusters is a challenging work because of the number of computing nodes and complexity of configuration parameters. Hence, several works tried to build the simulation context for MapReduce performance modelling and optimization.

MRPerf [82] is a sub-phase level simulation tool for modeling the performance of MapReduce applications on large clusters. Taking the user-defined cluster topology specification as its input, MRPerf simulates the inter- and intra-rack network communications performed by MapReduce, relying on the ns-2 network simulator. Similarly, according to the user-defined application job specification, MRPerf captures the computation time for each data-size-dependent sub-phase within a Map task using cycle–bye parameters. MRPerf also simulates disk I/O time using a disk simulator and the Data layout files.

MRSim [83] uses GidSim to simulate the network topology and communications, and uses the SimJava discrete event engine to model the other components of the system. MRSim takes a cluster topology file and a job specification file as its input, and provides simulation services for shared multi-core CPUs, hard disk drives, network traffic and memory buffers, merge parameters, parallel copy and sort parameters, combiners, and other components.

Kolberg et al. [88] developed the MSRG simulator for the MapReduce environment on top of the SimGrid simulation toolkit. The MSRG system provides a speculative execution mechanism and a data replication function, as well as several API functions that allow a user to define task costs and intermediary data.

Liu et al. [89] developed HSim, which is another MapReduce simulator for Hadoop applications. HSim models several Hadoop parameters (including node parameters, cluster parameters, Hadoop system parameters, and HSim simulator parameters) to provide highly accurate and dynamic behaviour simulation services. In the HSim architecture, the Cluster Reader component reads the cluster parameters from the Cluster Spec to create a simulated Hadoop cluster environment. The Job Spec is then processed by the Job Reader Component and the jobs are submitted to HSim to be simulated. However, neither the combiner nor the load balancing mechanism is considered in the current versions of HSim.

SimMapReduce [85] is a GridSim-based simulator designed to manage resources and to evaluate the scheduling performance of MapReduce. Its architecture has four layers, SimJava, GridSim, SimMapReduce, and User Definition. The discrete event processes are modelled in SimJava, the basic system component provisions are supported by Gridsim, the different cluster configurations are simulated in SimMapReduce, and the multi-layer scheduling algorithms are supported using predefined XML configuration files in the User Definition layer.

On the other hand, some studies improve the performance of MapReduce clusters by tracing the runtime or optimizing the configuration parameters.

MR-Scope [84] is a real-time tracing tool for MapReduce. It is composed of three layers, the Hadoop Layer, the RPC Layer, and the Client Layer. The Hadoop layer dynamically traces the node status in Hadoop using the heartbeat and observation point features. The RPC layer is responsible for establishing communication between the client and Hadoop utilizing a collection of protocols, with the internal RPC mechanism of Hadoop being changed to a non-blocking way. The Client layer can visualize the distribution of the HDFS blocks and their replicas, and display the ongoing processes in each running task using three different but interconnected perspectives, the HDFS perspective, the MapReduce task scheduling perspective, and the on-going MapReduce distributed perspective.

Predator [86] is an experience-guided configuration optimizer for MapReduce. It is based on a Hadoop configuration model in which the parameters are divided into four groups according to the general configuration experience, information on the cluster CPUs and memory, the input information for a job, and the results of the Grid Hill Climbing (GHC) algorithm developed for this system. The GHC algorithm, which is based on an objective function developed to estimate the job execution time, randomly generates sampling points that divide a parameter into equal subspaces and search the optimized configuration parameters.

In order to identify the relationships among the workload characteristics, Hadoop configurations, and workload performance, and accurate performance prediction of Hadoop cluster, Yang et al. [87] proposed a statistical analysis approach for Hadoop MapReduce. By applying the principal component analysis approach, they first transform the critical metrics of Hadoop workflow and framework configuration parameters into a small set of independent principal components. Then, they use cluster analysis

methods to identify groups of workloads with similar behaviours. In addition, they propose a regression model to predict relative performance of workloads under different Hadoop configurations.

Vianna et al. [90] developed an analytical model for estimating the MapReduce workload performance, particularly focusing on the intra-job pipeline parallelism between the Map and Reduce tasks. Based on the reference model which is designed to predict the performance of parallel computations, the authors represent the intra-job pipeline as a precedence tree. And then, the approximate Mean Value Analysis is applied to predict mean job response time, throughput and resource utilization of Hadoop.

4.6 Security and Energy Optimization of MapReduce

Security and power saving are two key factors that should be seriously considered for cloud computing center owners. Lots of research works have been proposed to improve the security and power efficiency for various distributed environments from grid computing to cloud computing. In addition, a few studies tried to improve the energy efficiency of cloud computing data centers. We review these typical solutions specific to MapReduce as the following.

4.6.1 Security Optimizations of MapReduce

In cloud computing context, users often submit individual data to the clusters, but they do not know where and how the data is being stored and processed since the operational details inside the cloud are invisible to data contributors. Some studies have been proposed to enhance the data privacy. Airavat [96] is a MapReduce-based distributed system that provides end-to-end confidentiality, integrity and privacy guarantees for sensitive data in cloud computing context. To prevent information leaks through system resources, it adds a SELinux-like mandatory access control to the MapReduce distributed file system. Moreover, Airavat adopts differential privacy strategy to ensure the output of aggregate computations does not violate the privacy of individual inputs.

For similar purpose, Guo et al. [99] extended MapReduce with novel integration of access control via attributed-based encryption and privacy-preserving aggregate computation via homomorphic encryption technologies. Within the proposed approach of [98], a transform module is integrated before the reduce phase to find the same key on ciphertext for the encrypted intermediate data.

Apart from the above solutions, several solutions extend the MapReduce model and runtime to optimize security for various usage scenarios. For example, in order to realize privacy data protection efficiently in hybrid clouds, Han et al. [100] presented a hierarchical control architecture based multi-cluster MapReduce programming model named HMR. Within HMR, data isolation and placement among private cloud and public clouds according to the data privacy characteristic is implemented by the control center. A Map- Reduce-GlobalReduce scheduling process is also designed to perform the corresponding distributed parallel computation correctly under the multi-clusters mode.

In order to support secure computing with mixed-sensitivity data on hybrid clouds, Zhang et al. [101] proposed tagged-MapReduce which extends the original key-value model with a sensitivity tag. Moreover, Zhang et al. designed several scheduling strategies that can exploit properties of the map and reduce functions to rearrange the computation for greater efficiency under these constraints while maintaining MapReduce correctness.

SecureMR [97] is a decentralized replication-based integrity verification scheme for MapReduce. It enhances the Hadoop MapReduce with five logical security components: Secure Manager, Secure Scheduler, Secure Task Executor, Secure Committer and Secure Verifier. Among them, Secure Manger and Secure Scheduler are deployed in the master node for task duplication, secure task assignment and commitment-based consistency checking. Secure Task Executor which runs in both mappers and reducers is used to prevent Dos and replay attacks that exploit fake or old task assignments. In addition, Secure Committer deployed in mappers generates commitments for the intermediate data and sends them to Secure Manager to complete the commitment-based consistency checking. Security Verifier running in a reducer collaborates with Secure Manager to verify the intermediate results from mappers. During the process, two protocols: Commitment protocol and Verification protocol are utilized to implement security communication among these components.

Different from above solutions, Yoon et al. [95] designed a black box approach, with no modification of the original MapReduce operations or introduction of extra operations, to detect the attacks launched by malicious or misconfigured nodes, which may tamper with the ordinary functions of the MapReduce framework. To achieve this goal, Yoon et al. collected the low-level system calls and traces of running Hadoop logs through dynamic instrumentation. Moreover, Yoon et al. identified a set of invariants to form the basic execution behaviour of both the Hadoop framework and the applications. Based on that, Yoon et al. detected the malicious nodes by matching these correlated Hadoop logs and system calls against the identified invariants.

4.6.2 Energy Optimizations for MapReduce

An effective way to reduce energy consumption of MapReduce clusters is to selectively power down idle nodes. Following this idea, the covering subset [91] scheme keeps one replica of each block within a small subset of machines to remain fully powered to preserve data availability, and the other nodes are powered down during low-utilization. On the contrary, the All-In strategy [92] powers down the entire cluster during periods of inactivity, and runs at full capacity more effectively.

On the other hand, Green HDFS [93] logically divides the HDFS cluster into disjoint hot and cold zones. The frequently accessed data is placed in the hot zone, which is always powered. Green HDFS fills the cold zone using one powered on machine at a time.

In order to improve the energy efficiency for time-sensitive, interactive data analysis workflows, Chen et al. [94] developed a workload manager Berkeley Energy Efficient MapReduce (BEEMR). The basic idea of BEEMR is firstly classifying each MapReduce job into one of three classes: interactive job, batch-able job and interruptible job, based on several empirical parameters. After that, the interactive jobs are serviced in

the proposed interactive zone and retained priority to set in a full-power ready state as this kind of job processes data in a low latency way. For all batch-able and interruptible jobs whose latency is not a concern, BEEMR put them in a wait queue of batch zone. Energy savings come from aggregating jobs in the batch zone to achieve high utilization, executing them in regular batches and transitioning machines in the batch zone to a low-power state when jobs complete.

5 Conclusions and Future Work

Continuous growth in the amount of data produced has led to the emergence of many scalability and performance challenges for traditional single-machine-based computing environments. The MapReduce paradigm has become the de facto standard for data-intensive computing in the cloud computing field because of its simplicity, scalability, and fault tolerance. Several approaches to optimize MapReduce have been developed recently, and these are aimed at improving the programming model and decreasing the occurrence of bottlenecks.

In this paper, we first described the basic idea behind the MapReduce model and discussed some of the shortcomings of the original model. We then assessed the methods designed to address these shortcomings in terms of their abilities to optimize the efficiency and flexibility of different aspects of MapReduce like optimizing job scheduling, extending the programming model, accelerating hardware performance, tuning the runtime, as well as security and energy optimization. These methods are each focused on optimizing a specific aspect of the model, and improve the performance of MapReduce and make it more suitable for use in dealing with complex computational logic. Research on the aspects mentioned above should be continued and combined. We also suggest some other problems that will also need to be studied, and these are listed below.

1. Optimize the MapReduce programming model for use in Hadoop YARN.

The novel runtime environment Hadoop YARN is currently considered to be the next generation of the MapReduce open source framework. However, the programming model used in MRv2 is not well improved, and most of the MapReduce optimizations being developed by researchers in both industry and academia are focused on improving performance of the original MapReduce model. More research is required to determine how to integrate currently available MapReduce optimizations (e.g., iterative MapReduce models, more efficient scheduling algorithms, and hardware acceleration solutions) into YARN-based MRv2. In addition, how to integrate the features of other YARN supported programming model such as Spark into MapReduce should be further researched.

2. Research into the parallelization of classical algorithms based on the optimized MapReduce.

In the context of big data processing, classical data mining algorithms, such as the k-means, SVM, and Naive Bayes algorithms, are frequently used to extract useful information. However, most of the current parallelization approaches using these algorithms are based on the original MapReduce programming model. The optimized MapReduce

models described above, which perform better and are more flexible than the original MapReduce model, are not often used. Research is therefore required to determine how these classical algorithms can be modified to make them more suitable for use in optimized MapReduce models.

3. Combining MapReduce with the Internet of Things.

The Internet of Things is developing rapidly, and many smart devices have been deployed. However, most of these are only used to gather and transfer data to a cloud computing center through the Internet, which means that the computing and storage capabilities of the devices are not exploited well. A considerable amount of research is therefore required to determine how the MapReduce model can be extended or restructured to meet the demands of the Internet of Things.

Acknowledgments This work was supported by the Scientific and Technological Research Program of Chongqing Municipal Education Commission under Grant KJ1500538, and the Fundamental Research Funds for the Central Universities of China under Grant 106112014CDJZR098801.

References

1. Shamsi, J., Khojaye, M.A., Qasmi, M.A.: Data-intensive cloud computing: requirement, expectations, challenges and solutions. J. Grid Comput. **11**(2), 281–310 (2013)
2. Meng, X., Ci, X.: Big data management: concepts, techniques and challenges. J. Comput. Res. Dev. **50**(1), 146–169 (2013)
3. Wang, S., Wang, H., Qin, X., Zhou, X.: Architecting big data: challenges, studies and forecasts. Chin. J. Comput. **34**(10), 1741–1752 (2013)
4. Mell, P., Grance, T.: Definition of Cloud Computing. Technical report, National Institute of Standards and Technology (NIST) (2009)
5. Sakr, S., Liu, A., Batista, D.M., Alomari, M.: A survey of large scale data management approaches in cloud environments. IEEE Commun. Surv. Tutor. **13**(3), 311–336 (2011)
6. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–133 (2008)
7. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google file system. In: Proceedings of 19th ACM Symposium on Operating Systems Principles, pp. 29–43. ACM (2003)
8. Mika, P., Tummarello, G.: Web semantics in the clouds. IEEE Intell. Syst. **23**(5), 82–87 (2008)
9. Apache Hadoop. http://hadoop.apache.org/. Accessed Oct. 2014
10. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: PigLatin: a not-so-foreign language for data processing. In: Proceedings of ACM SIGMOD International Conference of Management of Data (2008)
11. Mahou. http://mahout.apache.org/. Accessed Oct 2014
12. Urbani, J., Kotoulas, S., Maassen, J., Harmelen, F.V., Bal, H.: WebPIE: a web-scale parallel inference engine using mapreduce. J. Web Semant. **10**, 59–75 (2012)
13. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Sci. Am. **284**(5), 34–43 (2001)
14. Ding, L., Xin, J., Wang, G., Huang, S.: Efficient skyline query processing of massive data based on MapReduce. Chin. J. Comput. **34**(10), 1785–1796 (2011)
15. Doulkeridis, C., Nørvåg, K.: A survey of large-scale analytical query processing in MapReduce. VLDB J. **23**(3), 355–380 (2014)
16. Li, F., Ooi, B.C., Özsu, M.T., Wu, S.: Distributed data management using MapReduce. ACM Comput. Surv. **46**(3), 31 (2014)
17. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. ACM SIGMOD Rec. **40**(4), 11–20 (2011)
18. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
19. Van Biema, M.: Parallelism in Lisp. In: IJCAI87, pp. 56–61 (1987)

20. QT Concurrent. http://doc.qt.io/qt-5/qtconcurrent-index.html. Accessed Oct 2015
21. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for multi-core and multiprocessor systems. In: Proceedings of IEEE 13th International Symposium on High Performance Computer Architecture, pp. 13–24 (2007)
22. Talbot, J., Yoo, R.M., Kozyrakis, C.: Phoenix++: modular mapreduce for shared-memory systems. In: Proceedings of 2nd International Workshop on MapReduce and Its Applications, pp. 9–16 (2011)
23. Disco massive data—minimal code. http://discoproject.org/. Accessed Oct 2014
24. Geni, Skynet A Ruby MapReduce Framework. http://skynet.rubyforge.org/. Accessed Oct 2014
25. GridGain. http://www.gridgain.com/. Accessed Oct 2014
26. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G.: Twister: a runtime for iterative MapReduce. In: Proceedings of First International Workshop on MapReduce and Its Applications of ACM, pp. 810–818. ACM (2010)
27. Dou, A.J., Kalogeraki, V., Gunopulos, D., Mielikainen, T., Tuulos, V.: Misco: a MapReduce Framework for mobile systems. In: Proceedings of 3rd International Conference on Pervasive Technologies Related to Assistive Environment (2010)
28. DeWitt, D., Stonebraker, M.: MapReduce: a major step backwards. The Database Column **1** (2008)
29. Sun, X., He, C., Lu, Y.: ESAMR: an enhanced self-adaptive MapReduce scheduling algorithm. In: Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems, pp. 148–155 (2012)
30. Husain, M.F., McGlothlin, J., Masud, M.M., Khan, L.R., Thuraisingham, B.: Heuristics-based query processing for large RDF graphs using cloud computing. IEEE Trans. Knowl. Data Eng. **23**(9), 1312–1327 (2011)
31. Sandholm, T., Lai, K.: MapReduce optimization using regulated dynamic prioritization. In: Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems, pp. 299–310 (2009)
32. Seo, S., Jang, I., Woo, K., Kim, I., Kim, J.-S., Maeng, S.: HPMR: prefetching and pre-shuffling in shared MapReduce computation environment. In: Proceedings of 2009 IEEE International Conference on Cluster Computing and Workshops (2009)
33. Polo, J., Carrera, D., Becerra, Y., Torres, J., Ayguade, E., Steinder, M., Whalley, I.: Performance-driven task co-scheduling for MapReduce environments. In: Proceedings of 12th IEEE/IFIP Network Operations and Management Symposium, pp. 373–380 (2010)
34. Polo, J., Carrera, D., Becerra, Y., Beltran, V., Torres, J., Ayguade, E.: Performance management of accelerated MapReduce workloads in heterogeneous clusters. In: Proceedings of 39th International Conference on Parallel Processing, pp. 653–662 (2010)
35. Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I.: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: EuroSyst, pp. 265–278 (2010)
36. Yao, Y., Tai, J., Sheng, B., Mi, N.: Scheduling heterogeneous MapReduce jobs for efficiency improvement in enterprise clusters. In: Proceedings of 2013 IFIP/IEEE International Symposium on Integrated Network Management, pp. 872–875 (2013)
37. Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., Stoica, I.: Improving MapReduce performance in heterogeneous environments. In: Proceedings of 8th USENIX Symposium on Operating System Design and Implementation, pp. 29–42 (2008)
38. Chen, Q., Zhang, D., Guo, M., Deng, Q., Guo, S.: SAMR: A self-adaptive MapReduce scheduling algorithm in heterogeneous environment. In: Proceedings of 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, 10th IEEE International Conference on Scalable Computing and Communications, pp. 2736–2743 (2010)
39. Chen, Q., Guo, M.Y., Deng, Q.N., Zheng, L., Guo, S., Shen, Y.: HAT: history-based auto-tuning MapReduce in heterogeneous environments. J. Supercomput. **64**(3), 1038–1054 (2013)
40. Zhang, X., Feng, Y., Feng, S., Fan, J., Ming, Z.: An effective data locality aware task scheduling method for MapReduce framework in heterogeneous environments. In: Proceedings of 2011 International Conference on Cloud and Service Computing, pp. 235–242 (2011)
41. Ahmad, F., Chakradhar, S.T., Raghunathan, A., Vijaykumar, T.N.: Tarazu: optimizing MapReduce on heterogeneous clusters. In: Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 61–74 (2011)

42. Yang, Y., Shi, B., Jiang, B., Xiang, L.: Data-replicas scheduler for heterogeneous MapReduce cluster. J. Netw. **8**(5), 1096–1103 (2013)
43. Sehrish, S., Mackey, G., Shang, P.J., Wang, J., Bent, J.: Supporting HPC analytics applications with access patterns using data restructuring and data-centric scheduling techniques in MapReduce. IEEE Trans. Parallel Distrib. Syst. **24**(1), 158–169 (2013)
44. Peng, D., Dabek, F., Inc, G.: Large-scale incremental processing using distributed transactions and notifications. In: Usenix Symposium on Operating Systems Design & Implementation, pp. 4–6 (2010)
45. Bu, X., Rao, J., Xu, C.: Interference and locality-aware task scheduling for MapReduce applications in virtual clusters. In: Proceedings of the 22nd ACM International Symposium on High-Performance Parallel and Distributed Computing, pp. 227–238 (2013)
46. Yang, H.-C., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-Reduce-Merge: simplified relational data processing on large clusters. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 1029–1040 (2007)
47. Jiang, D., Tung, A.K.H., Chen, G.: Map–Join–Reduce: toward scalable and efficient data analysis on large clusters. IEEE Trans. Knowl. Data Eng. **23**(9), 1299–1311 (2011)
48. Ferrera, P., de Prado, I., Palacios, E., Fernandez-Marquez, J.L., Serugendo, G.D.: Tuple MapReduce: beyond classic MapReduce. In: Proceedings of 12th IEEE International Conference on Data Mining, pp. 260–269 (2012)
49. Vu, T.-T., Huet, F.: A lightweight continuous jobs mechanism for MapReduce frameworks. In: Proceedings of 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pp. 269–276 (2013)
50. Premchaiswadi, W., Pomsaiyud, W.: Optimizing and tuning MapReduce jobs to improve the large-scale data analysis process. Int. J. Intell. Syst. **28**(2), 185–200 (2013)
51. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: Haloop: efficient iterative data processing on large clusters. Proc. VLDB Endow. **3**(1), 285–296 (2010)
52. Kambatla, K., Rapolu, N., Jagannathan, S., Grama, A.: Asynchronous algorithms in MapReduce. In: Proceedings of 2010 IEEE International Conference on Cluster Computing, Cluster, pp. 245–254 (2010)
53. Elnikety, E., Elsayed, T., Ramadan, H.E.: IHadoop: asynchronous iterations for MapReduce. In: Proceedings of 2011 3rd IEEE International Conference on Cloud Computing Technology and Science, pp. 81–90 (2011)
54. Wang, L., Ni, Z., Zhang, Y., Wu, Z., Tang, L.: Pipelined-MapReduce: an improved MapReduce parallel programming model. In: Proceedings of 4th International Conference on Intelligent Computation Technology and Automation, pp. 871–874 (2011)
55. Xu, W., Gong, X.J., Li, X.Y.: MapCombine: a lightweight solution to improve the efficiency of iterative MapReduce. In: Proceedings of iCETS 2012, pp. 444–456
56. Zhang, Y.F., Gao, Q.X., Gao, L.X., Wang, C.R.: iMapReduce: a distributed computing framework for iterative computation. J. Grid Comput. **10**(1), 47–68 (2012)
57. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., Mccauley, M., Franklin, M., Shenker, S., Stoica, I.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In: USENIX Symposium on Networked Systems Design and Implementation, vol. 70, pp. 141–146 (2012)
58. Gunarathne, T., Zhang, B.J., Wu, T.L., Qiu, J.: Scalable parallel computing on clouds using Twister4Azure iterative MapReduce. Future Gener. Comput. Syst. **29**(4), 1035–1048 (2013)
59. Condie, T., Conway, N., Alvaro, P., Hellerstein, M., Elmeleegy, K., Sears, R.: MapReduce Online. EECS Department, University of California, Berkeley. Tech. Rep. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-136.html
60. Böse, J. H., Andrzejak, A., Hogqvist, M.: Beyond online aggregation: Parallel and incremental data mining with online Map-Reduce. In: Proceedings of 2010 Workshop on Massive Data Analytics on the Cloud, MDAC 2010, in Association with the 19th Annual World Wide Web Conference (2010)
61. Phan, L.T.X., Zhang, Z., Loo, B.T., Lee, I.: Real-Time MapReduce Scheduling. Technical Report, University of Pennsylvania (2010)
62. Dou, A.J., Kalogeraki, V., Gunopulos, D., Mielikainen, T., Tuulos, V.: Scheduling for real-time mobile MapReduce systems. In: Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems, pp. 247–258 (2011)

63. Peng, C.-Z., Jiang, Z.-J., Cai, X.-B., Zhang, Z.-K.: Real-time analytics processing with MapReduce. In: Proceedings of 2012 International Conference on Machine Learning and Cybernetics, vol. 4, pp. 1308–1311 (2012)
64. Qi, K., Zhao, Z., Fang, J., Ma, Q.: Real-time processing for high speed data stream over large scale data. Chin. J. Comput. **35**(3), 477–490 (2012)
65. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (2010)
66. Hong, C., Chen, D., Chen, W., Zheng, W., Lin, H.: MapCG: Writing parallel program portable between CPU and GPU. In: Proceedings of 19th International Conference on Parallel Architecture and Compilation Techniques, pp. 217–226 (2010)
67. Jiang, W., Ravi, V.T., Agrawal, G.: A Map-reduce system with an Alternate API for multi-core environments. In: Proceedings of 10th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pp. 84–93 (2010)
68. Stuart, J.A., Chen, C.-K., Ma, K.-L., Owens, J.D.: Multi-GPU volume rendering using MapReduce. In: Proceedings of 19th ACM International Symposium on High Performance Distributed Computing, pp. 841–848 (2010)
69. Fang, W., He, B., Luo, Q., Govindaraju, N.K.: Mars: accelerating MapReduce with graphics processors. IEEE Trans. Parallel Distrib. Syst. **22**(4), 608–620 (2011)
70. Elteir, M., Lin, H., Feng, W.-C., Scogland, T.: StreamMR: an optimized MapReduce framework for AMD GPUs. In: Proceedings of 17th IEEE International Conference on Parallel and Distributed Systems, pp. 364–371 (2011)
71. Chen, L., Agrawal, G.: Optimizing MapReduce for GPUs with effective shared memory usage. In: Proceedings of 21st ACM Symposium on High-Performance Parallel and Distributed Computing, pp. 199–210 (2012)
72. Basaran, C., Kang, K.D.: Grex: an efficient MapReduce framework for graphics processing units. J. Parallel Distrib. Comput. **73**(4), 522–533 (2013)
73. Chen, R., Chen, H.: Tiled-mapreduce: efficient and flexible mapreduce processing on multicore with tiling. Trans. Archit. Code Optim. **10**, 1 (2013)
74. de Krujif, M., Sankaralingam, K.: MapReduce for the Cell B.E. architecture. IBM J. Res. Dev. **53**(5), 10:1–10:12 (2009)
75. Rafique, M.M., Rose, B., Butt, A.R., Nikolopoulos, D.S.: Supporting MapReduce on large-scale asymmetric multi-core clusters. Oper. Syst. Rev. **43**, 25–34 (2009)
76. Papagiannis, A., Nikolopoulos, D.S.: Rearchitecting MapReduce for heterogeneous multicore processors with explicitly managed memories. In: Proceedings of 39th International Conference on Parallel Processing, pp. 121–130 (2010)
77. He, C., Weitzel, D., Swanson, D., Lu, Y.: HOG: Distributed Hadoop MapReduce on the grid. In: Proceedings of 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 1276–1283 (2012)
78. Heintz, B., Wang, C., Chandra, A., Weissman, J.: Cross-phase optimization in mapreduce. In: Proceedings of 1st IEEE International Conference on Cloud Engineering, pp. 338–347 (2013)
79. Wang, L., Tao, J., Ranjan, R., Marten, H., Streit, A., Chen, J., Chen, D.: G-Hadoop: MapReduce across distributed data centers for data-intensive computing. Future Gener. Comput. Syst. **29**(3), 739–750 (2013)
80. Mantha, P.K., Luckow, A., Jha, S.: Pilot-MapReduce: an extensible and flexible MapReduce implementation for distributed data. In: Proceedings of 2012 3rd International Workshop on MapReduce and Its Applications, pp. 17–24 (2012)
81. Marozzo, F., Talia, D., Trunfio, P.: P2P-MapReduce: parallel data processing in dynamic cloud environments. J. Comput. Syst. Sci. **78**(5), 1382–1402 (2012)
82. Wang, G., Butt, A.R., Pandey, P., Gupta, K.: Using realistic simulation for performance analysis of MapReduce setups. In: Proceedings of 1st ACM Workshop on Large-Scale System and Application Performance, pp. 16–29 (2009)
83. Hammoud, S., Li, M., Liu, Y., Alham, N.K., Liu, Z.: MRSim: a discrete event based MapReduce simulator. In: Proceedings of 2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery, pp. 2993–2997 (2010)

84. Huang, D., Shi, X., Ibrahim, S., Lu, L., Liu, H., Wu, S., Jin, H.: MR-Scope: a real-time tracing tool for MapReduce. In: Proceedings of 19th ACM International Symposium on High Performance Distributed Computing, pp. 849–855 (2010)
85. Teng, F., Yu, L., Magoules, F.: SimMapReduce: a simulator for modeling MapReduce framework. In: Proceedings of the 2011 5th FTRA International Conference on Multimedia and Ubiquitous Engineering, pp. 277–282 (2011)
86. Wang, K., Lin, X., Tang, W.: Predator—an experience guided configuration optimizer for Hadoop MapReduce. In: Proceedings of 4th IEEE International Conference on Cloud Computing Technology and Science, pp. 419–426 (2012)
87. Yang, H.L., Luan, Z.Z., Li, W.J., Qian, D.P.: MapReduce workload modeling with statistical approach. J. Grid Comput. **10**(2), 279–310 (2012)
88. Kolberg, W., Marcos, P.D., Anjos, J.C.S., Miyazaki, A.K.S., Geyer, C.R., Arantes, L.B.: MRSG—a MapReduce simulator over SimGrid. Parallel Comput. **39**(4–5), 233–244 (2013)
89. Liu, Y., Li, M.Z., Alham, N.K., Hammoud, S.: HSim: a MapReduce simulator in enabling cloud computing. Future Gener. Comput. Syst. **29**(1), 300–308 (2013)
90. Vianna, E., Comarela, G., Pontes, T., Almeida, J., Almeida, V., Wilkinson, K., Kuno, H., Dayal, U.: Analytical performance models for MapReduce workloads. Int. J. Parallel Prog. **41**(4), 495–525 (2013)
91. Leverich, J., Kozyrakis, C.: On the energy (In) efficiency of Hadoop Clusters. In: Proceedings of HotPower (2009)
92. Lang, W., Patel, J.: Energy management for MapReduce clusters. In: VLDB (2010)
93. Kaushik, R.T., et al.: Evaluation and analysis if GreenHDFS: a selft-adaptive. Energy-conserving variant of the Hadoop distributed file system. In: Proceedings of IEEE CloudCom (2010)
94. Chen, Y., Alspaugh, S., Borthakur, D., Katz, R.: Energy efficiency for large-scale MapReduce workloads with significant interactive analysis. In: Proceedings of EuroSys (2012)
95. Yoon, E., Squicciarini, A.: Toward detecting compromised MapReduce workers through log analysis. In: Proceedings of 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (2014)
96. Roy, I., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: security and privacy for MapReduce. In: NSDI (2010)
97. Wei, W., Du, J., Yu, T., Gu, X.: SecureMR: a service integrity assurance framework for MapReduce. In: Proceedings of 2009 Annual Computer Security Applications Conference (2009)
98. Chen, X., Huang, Q.: The data protection of MapReduce using homomorphic encryption. In: Proceedings of 4th IEEE International Conference on Software Engineering and Service Science, pp. 419–421 (2013)
99. Guo, Z., Zhu, X., Guo, L., Kang, S.: Design of a security framework On MapReduce. In: Proceedings of 5th International Conference on Intelligent Networking and Collaborative Systems, pp. 139–145 (2013)
100. Han, H., Zheng, W.: A privacy data-oriented hierarchical MapReduce programming model. Telkomnika Indones. J. Electr. Eng. **11**(8), 4587–4593 (2013)
101. Zhang, C., Chang, E.-C., Yap, R.H.C.: Tagged-MapReduce: a general framework for secure computing with mixed-sensitivity data on hybrid clouds. In: Proceedings of 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (2014)
102. Apache Pig. http://pig.apache.org/. Accessed May 2015
103. Cascading. http://www.cascading.org/. Accessed May 2015
104. Scalding. http://www.cascading.org/projects/scalding/. Accessed May 2015
105. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: parallel analysis with Sawzall. Sci. Program. **13**(4), 277–298 (2005)
106. Summingbird. https://github.com/twitter/summingbird. Accessed May 2015
107. Akidau, T., Balikov, A., Chernyak, S., Haberman, J., Lax, R., Mcveety, S., Mills, D., Nordstrom, P., Whittle, S.: MillWheel: faulttolerant stream processing at Internet scale. In: Proceedings of the 39th International Conference on Very Large Data Bases, VLDB, vol. 6 (2013)
108. Lu, M., Zhang, L., Huynh, H.P., Ong, Z., Liang, Y., He, B., Goh, R.S.M., Huynh, R.: Optimizing the MapReduce framework on Intel Xeon Phi coprocessor. In: Proceedings of IEEE International Conference on Big Data (2013)
109. QT. http://www.qt.io/download/. Accessed Oct 2015