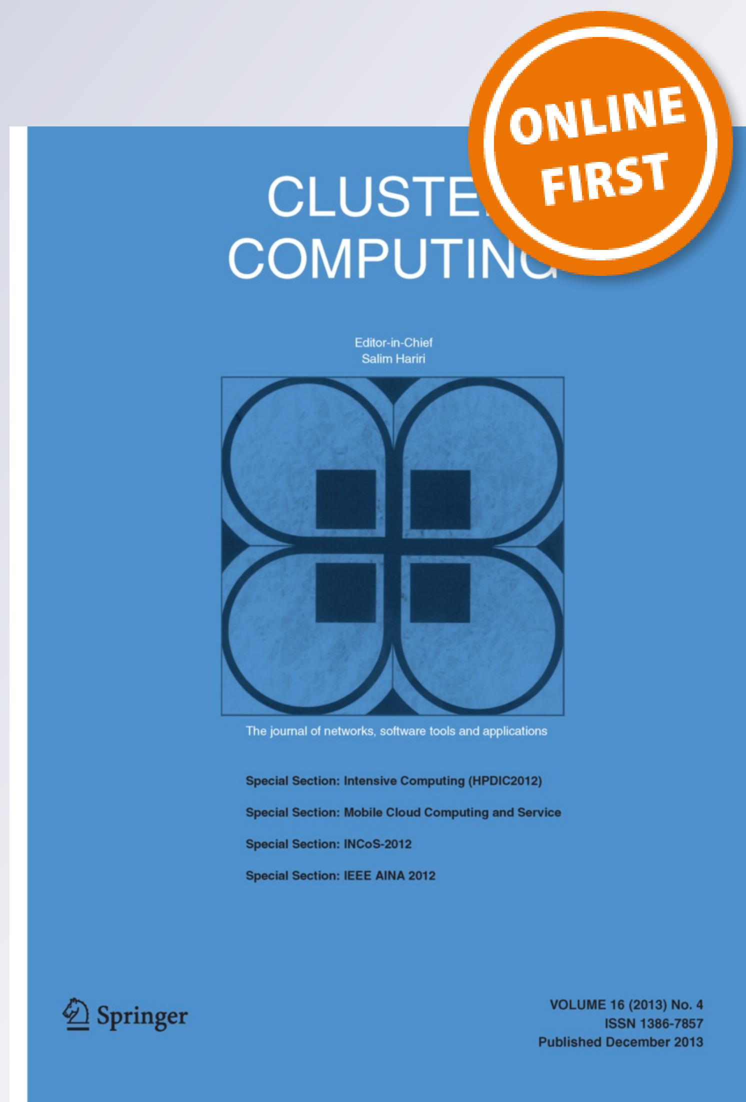
Adaptive Combiner for MapReduce on cloud computing

Tzu-Chi Huang, Kuo-Chih Chu, Wei-Tsong Lee & Yu-Sheng Ho

Cluster Computing
The Journal of Networks, Software Tools and Applications

ISSN 1386-7857

Cluster Comput
DOI 10.1007/s10586-014-0362-3



Springer

Your article is protected by copyright and all rights are held exclusively by Springer Science +Business Media New York. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Springer

Author's personal copy

Cluster Comput
DOI 10.1007/s10586-014-0362-3

Adaptive Combiner for MapReduce on cloud computing

Tzu-Chi Huang · Kuo-Chih Chu · Wei-Tsong Lee · Yu-Sheng Ho

Received: 21 May 2013 / Revised: 30 October 2013/ Accepted: 18 February 2014
© Springer Science+Business Media New York 2014

Abstract MapReduce is a programming model to process a massive amount of data on cloud computing. MapReduce processes data in two phases and needs to transfer intermediate data among computers between phases. MapReduce allows programmers to aggregate intermediate data with a function named combiner before transferring it. By leaving programmers the choice of using a combiner, MapReduce has a risk of performance degradation because aggregating intermediate data benefits some applications but harms others. Now, MapReduce can work with our proposal named the Adaptive Combiner for MapReduce (ACMR) to automatically, smartly, and trainer for getting a better performance without any interference of programmers. In experiments on seven applications, MapReduce can utilize ACMR to get the performance comparable to the system that is optimal for an application.

Keywords MapReduce · Combiner · Cloud computing · ACMR

T.-C. Huang (✉) · K.-C. Chu
Department of Electronic Engineering, Lunghwa University of Science and Technology, Taoyuan, Taiwan
e-mail: tzuchi.phd@gmail.com; tzuchi@mail.lhu.edu.tw

K.-C. Chu
e-mail: kcchu@mail.lhu.edu.tw

W.-T. Lee · Y.-S. Ho
Department of Electrical Engineering, Tamkang University, Taipei, Taiwan
e-mail: wtlee@mail.tku.edu.tw

Y.-S. Ho
e-mail: clare.ysho@gmail.com

1 Introduction

MapReduce [1–4] is a programming model proposed by Google to process a massive amount of data. MapReduce can distribute jobs over a huge number of computers in a cloud [5–9]. For example, MapReduce has been used to help Google process more than terabyte data everyday. Most importantly, MapReduce allows a programmer who has no experience in developing parallel programs to easily create a program capable of using computers in a cloud. MapReduce can automatically distribute the program over computers in a cloud to simultaneously process data just with two functions developed by the programmer, i.e., the Map function (a.k.a. Mapper) and the Reduce function (a.k.a. Reducer).

When MapReduce works, it partitions input data into data blocks and distributes them over Mappers running in computers. After Mappers process data blocks and generate outputs called intermediate data, MapReduce forwards the intermediate data to its corresponding Reducer. MapReduce needs a programmer to develop a Mapper that generates intermediate data composed of pairs of a key and a value after processing a data block. MapReduce automatically groups values with the same key and processes the values with their corresponding Reducer. Finally, MapReduce collects outputs of all Reducers as the final result. When forwarding intermediate data from Mappers to Reducers, however, MapReduce may have different performances according to whether intermediate data is processed by a combiner or not.

A combiner [1] is a mechanism supposed to handle intermediate data in a Mapper before intermediate data is delivered to a Reducer. Although a combiner technically can be considered a function having code identical to a Reducer, we simply refer to it as "combiner" instead of calling it "the Reducer running in the computer at where a Map-

Published online: 11 March 2014

Author's personal copy

per resides". A combiner can aggregate intermediate data to decrease its size. Accordingly, a combiner can conserve network bandwidth and shorten the delay stemming from the propagation of intermediate data between Mappers and Reducers. Although a combiner may improve performances by bandwidth conservation and delay shortening in networks, it trades the processing of intermediate data among Reducers for intermediate data aggregation performed at Mappers. A combiner increases Mapper workloads and conversely may degrade performances because of idling Reducers. Besides, a combiner needs a sophisticated design because intermediate data aggregation uses much memory in Mappers. Because a combiner has the pros (i.e., bandwidth conservation and delay shortening in networks) and the cons (i.e., Mapper overload and design difficulty) in intermediate data aggregation, the decision of using a combiner currently is left to programmers.

Using a combiner in MapReduce is not an easy task. Using a combiner requires considering features of applications, types and quantity of intermediate data, numbers of Mappers and Reducers at run time, computation power and memory of computers, and network bandwidth among computers in a cloud. Without a careful design, using a combiner may incur performance degradation due to computation overload and memory exhaustion in Mappers. Besides, using a combiner burdens a programmer with extra tasks (e.g. design issues of a combiner) that can prevent him or her from focusing on application designs. Accordingly, using a combiner should be automatically, smartly, and transparently finished to surely improve performances without burdening a programmer.

In this paper, Adaptive Combiner for MapReduce(ACMR) is proposed to facilitate the use of a combiner in MapReduce. ACMR can automatically determine the situation of using a combiner according to behaviors of an application at run time. ACMR uses the Single Exponential Smoothing (SES) algorithm to smartly predict workloads of Mappers and the quantity of intermediate data. According to workloads and computation power of Mappers, ACMR adjusts the use of combiners in Mappers without overloading Mappers and idling Reducers. ACMR uses a combiner for various applications on demand and gets the better performance. ACMR transparently works in a MapReduce system to serve various applications without any interference of programmers. For a proof of concept, ACMR currently is implemented in a MapReduce system written in PHP (short for Hypertext Preprocessor) [10–12], a widely-used general-purpose scripting language. In experiments, ACMR has tests in performance impacts of α chunk size, and change interval with three applications. In comparison with systems of always using a combiner and using no combiner, ACMR has the performance comparable to the system that is optimal for an application.

We organize this paper as follows. We introduce a background in Sect. 2. In Sect. 3, we address Adaptive Combiner for MapReduce (ACMR). We implement ACMR in Sect. 4 and have performance observations in Sect. 5. In Sect. 6, we review some related works. Finally, we conclude this paper in Sect. 7.

Mapper(Key1, Value1) => List(Key2, Value2)
Reducer(Key2, List(Value2)) => List(Key3, Value3)

Fig. 1 Processing data with mapper and reducer

2 Background

2.1 MapReduce

MapReduce is a programming model for processing a huge amount of data. Because of simplicity, MapReduce merely needs a programmer to focus on the development of the Map function (a.k.a. Mapper) and the Reduce function (a.k.a. Reducer). Accordingly, MapReduce divides the processing of data into the Map phase and the Reduce phase and respectively calls the Mapper and the Reducer developed by a programmer in the corresponding phase. When shifting from the Map phase to the Reduce phase, MapReduce automatically synchronizes operations of the data processing by forwarding outputs generated by Mappers (i.e. intermediate data) to the corresponding Reducers. In the Reduce phase, MapReduce collects outputs of Reducers as the final result.

A programmer develops a Mapper and a Reducer as shown in Fig. 1. A programmer develops a Mapper: (1) to process input data composed of pairs of a key and a value, and (2) to generate intermediate data composed of different pairs of a key and a value (i.e. Key2 and Value2 in Fig. 1). Because the system automatically handles intermediate data by grouping values according to their keys and forwards intermediate data to the corresponding Reducer, a programmer should develop a Reducer to process intermediate data that is separated into groups according to different keys (i.e. the list of Value2 in Fig. 1). Finally, a programmer develops a Reducer to both process intermediate data and generate different pairs of a key and a value (i.e. a list of Key3 and Value3 in Fig. 1) that will be collected by the system as the final result.

When the system gets intermediate data composed of pairs of a key and a value generated by Mappers, it applies a partition function (e.g. a hash function receiving a key as the input) to intermediate data and determines its corresponding Reducer according to the output of the partition function. When forwarding intermediate data to a Reducer, the system may consume much bandwidth and have delays in networks to degrade performances. To reduce bandwidth consumption and delays in networks, the system may aggregate intermediate data in Mappers to decrease its size before forwarding it to Reducers. In intermediate data aggregation, however, the system needs to consume computation power and memory in Mappers. Without a careful design, the system may have per-

Author's personal copy

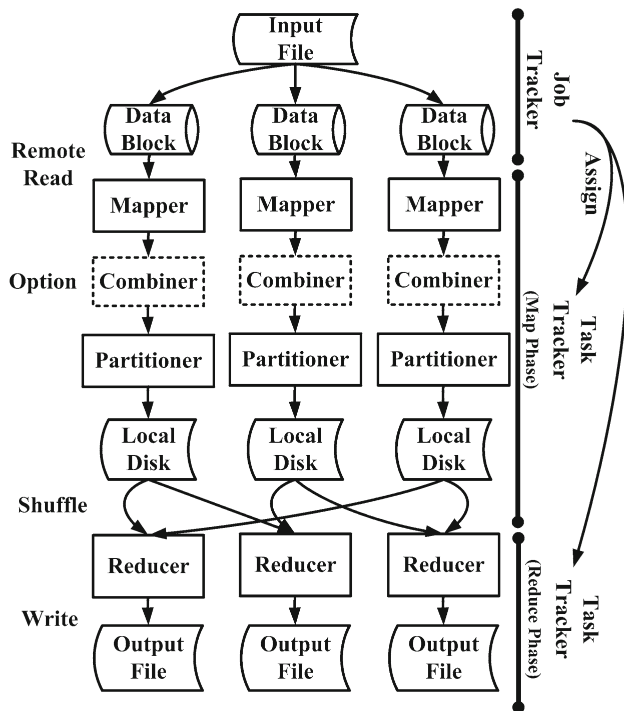


Fig. 2 MapReduce working principle in Hadoop

formance degradation due to intermediate data aggregation. Accordingly, the system has a tough problem of handling intermediate data, which is the focus of this paper.

2.2 Hadoop

Hadoop [13, 14] is an open-source MapReduce system implemented by the community of the Apache Software Foundation according to Google MapReduce [1]. Hadoop has been widely used by Yahoo, Facebook, and Amazon, so it should be mentioned here. Although Hadoop includes not only a MapReduce system but also a Hadoop Distributed File System (HDFS) [15] and a Hadoop database (HBase) [16], we merely discuss its MapReduce system and refer to the MapReduce system as Hadoop. Hadoop has the MapReduce working principle in Fig. 2.

Hadoop considers an application a job and divides the job into several tasks that can be distributed over computers in a cloud. Hadoop uses a Job Tracker and several Task Trackers to manage computation resources used by the application. Meanwhile, Hadoop uses a Name Node and several Data Nodes to manage input data of an application. Hadoop partitions input data of an application into data blocks of 64 MB and distributes the data blocks over Mappers in a cloud. After a Mapper processes a data block and generates intermediate data, Hadoop stores intermediate data in the disk of the computer on where the Mapper runs. Next, Hadoop enters the Shuffle phase and moves intermediate data to computers on where the corresponding Reducers run.

When a Mapper generates intermediate data, Hadoop does no operation about intermediate data except saving it to disks. Hadoop merely provides programmers with a Java class named Combiner that can be inherited to implement the function of intermediate data aggregation. However, Hadoop burdens programmers with the design. Hadoop allows programmers to enable the function of using a combiner but does not provide them with certain information such as system run-time information and services. Working with a poorly designed combiner, Hadoop may exhaust computation resources and memory of computers running Mappers but idle those running Reducer to degrade performances. Working without a combiner, conversely, Hadoop may consume bandwidth and have delays in networks to degrade performances as well when forwarding intermediate data from Mappers to Reducers. Like other systems, Hadoop disables the function of a combiner by default and leaves the use of Class Combiner to programmers.

3 Adaptive Combiner for MapReduce (ACMR)

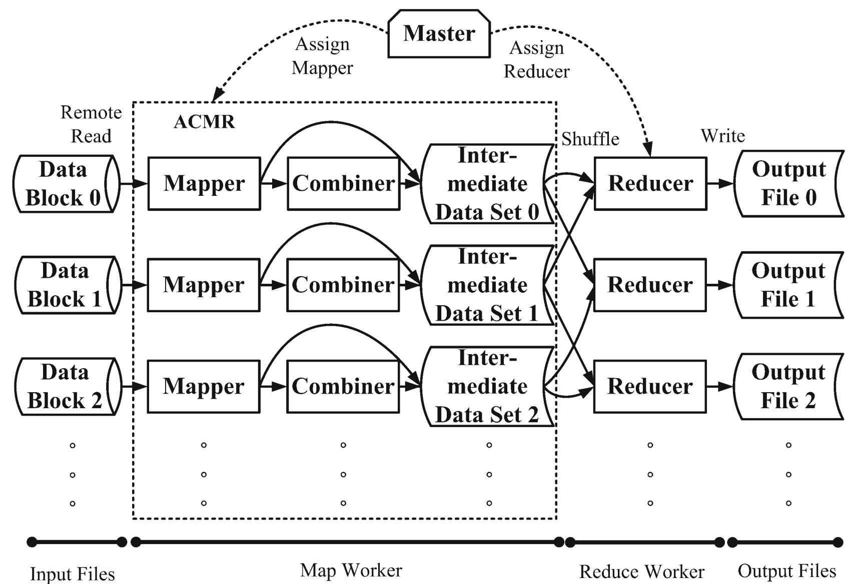
Adaptive Combiner for MapReduce (ACMR) can facilitate the use of a combiner in MapReduce. Briefly, ACMR has features of (1) determining to use a combiner automatically according to behaviors of an application at run time, (2) predicting workloads of Mappers and the quantity of intermediate data, (3) adjusting the use of combiners in Mappers without overloading Mappers and idling Reducers, (4) using a combiner for various applications on demand to get the better performance, and 5) working in a MapReduce system transparently to serve various applications without any interference of programmers. In this section, we detail the ACMR design including the ACMR overview, the ACMR working principle, Task Data Sample Module in ACMR, Aggregation Decision Module in ACMR, and Performance Prediction Module in ACMR.

3.1 ACMR overview

ACMR is a mechanism based on performance history of using a combiner and not using a combiner in order to determine whether the system will use a combiner or not. In Fig. 3, ACMR works in the Map phase and deduces whether using a combiner or not for incoming input data. ACMR is designed for input data in a regular format because most applications in a cloud serve input data in a certain regular format, e.g. a web log having URLs in each record and a document having words separated by spaces. In the Map phase, ACMR uses Task Data Sample Module to split input data into data blocks. In the initial time of running an application, ACMR lets intermediate data of one data block (after the processing of a Mapper) go through a combiner but that of another data

Author's personal copy

Fig. 3 ACMR overview



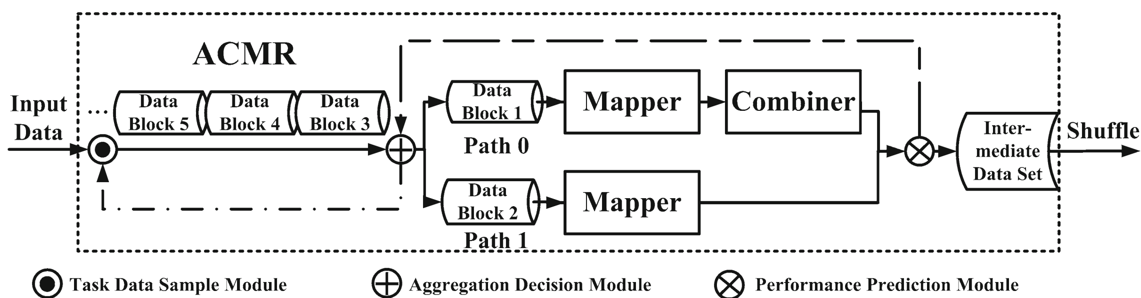


Fig. 4 ACMR working principle

block (after the processing of a Mapper) bypass the combiner. Next, ACMR respectively records performance metrics of processing the two data blocks in a Mapper, i.e. the processing time and the intermediate data size. After that, ACMR uses Performance Prediction Module to predict performance metrics of processing the next data block (i.e. the third data block) in a Mapper with and without a combiner. ACMR uses Aggregation Decision Module to handle performance metrics for determining whether intermediate data of the next data block (i.e. the third data block) should go through a combiner or not. No matter intermediate data of a data block (e.g. the third data block) goes through a combiner or not, ACMR always records performance metrics of processing the data block and applies Aggregation Decision Module to the performance metrics in order to determine whether intermediate data of the next data block (e.g. the fourth data block) should go through a combiner or not.

3.2 ACMR working principle

ACMR roughly consists of 3 procedures, i.e. performance history observation, performance prediction, and aggregation decision. ACMR works in the computer running a Mapper in order to control the further processing of intermediate data generated by the Mapper as shown in Fig. 4. In each computer running a Mapper, ACMR uses Task Sample Data Module to split input data into data blocks. After that, ACMR uses Aggregation Decision Module to determine whether intermediate data of a data block processed by a Mapper should go through a combiner or not. In the initial time of running an application, ACMR lets intermediate data of a data block (after the processing of a Mapper) go through a combiner (i.e. Data Block 1 in Fig. 4) but that of another data block (after the processing of a Mapper) bypass the combiner (i.e. Data Block 2 in Fig. 4). When a data block is processed by a Mapper with or without a combiner, ACMR records the processing time and the intermediate data size of a data block and refers to them as performance metrics. According to performance metrics of the two data blocks (i.e. Data Blocks 1 and 2 in Fig. 4), ACMR uses Performance Prediction Module to predict performance metrics of the next data block (i.e. Data Block 3 in Fig. 4) with the processing of a combiner (i.e. Path 0 in Fig. 4) and those of the next data block without the processing of a combiner (i.e. Path 1 in

Author's personal copy

Fig. 4). Next, ACMR uses Aggregation Decision Module to determine whether intermediate data of the next data block (i.e. Data Block 3) should go through a combiner (i.e. Path 0 in Fig. 4) or not (i.e. Path 1 in Fig. 4). Meanwhile, ACMR notifies Task Data Sample Module of the decision made by Aggregation Decision Module, so Task Data Sample Module can dynamically adjust the size used to split incoming input data. Eventually, ACMR repeats the procedures of observing performance history, predicting performances of the next data block processed by a Mapper with and without a combiner, and determining whether the next data block should be processed by a Mapper with or without a combiner.

3.3 Task Data Sample Module

Task Data Sample Module is responsible for splitting input data into data blocks because ACMR uses a data block as a unit to determine whether its intermediate data should go through a combiner or not. However, Task Data Sample Module needs to choose the appropriate data block size because the data block size can affect performances. Task Data Sample Module should adjust the data block according to the variability of data format in input data, so the system can determine the appropriate way to process intermediate data of new incoming input data.

If Task Data Sample Module splits input data into few large data blocks, ACMR may not quickly respond to the change of format in input data, which may degrade performances. When a half of input data is processed by a Mapper to generate intermediate data having many repeated keys (suitable for intermediate data aggregation) but another half is not, for example, Task Data Sample Module should split input data into at least two data blocks because intermediate data of the input data should be handled in different ways. If Task Data Sample Module splits input data into many small data blocks, conversely, ACMR wastes computation power to observe performance history, predict performances, and make aggregation decision if input data does not have much change in format. When input data does not have a change in format, Task Data Sample Module should increase the current block size because the way to process intermediate data is usually the same. When input data has a change in format, Task Data Sample Module should decrease the current block size to alleviate negative impacts of the temporarily inappropriate way to process intermediate data.

$$s(t+1) = \begin{cases} s(t)+1 & if\ intermediate\ data\ processing\ way\ is\ not\ changed, \\ ceil(\frac{s(t)}{2}) & if\ intermediate\ data\ processing\ way\ is\ changed, \end{cases} \quad \text{where } s(t) \geq 1 \tag{1}$$

Task Data Sample Module uses a chunk size ranging from 16 MB to 64 MB (configurable by programmers) as a unit to adjust the data block. Referring to the widely-used flow control mechanism in TCP [17], Task Data Sample Module uses the Additive Increase/Multiplicative-Decrease (AIMD) algorithm [18] to dynamically adjust the data block. According to Eq. 1, Task Data Sample Module increases the current data block size by a chunk size if Aggregation Decision Module decides not to change the way of processing intermediate data (e.g. keeping on intermediate data aggregation). Conversely, Task Data Sample Module decreases the current data block size by a half if Aggregation Decision Module decides to change the way of processing intermediate data (e.g. switching to intermediate data aggregation). Through the AIMD algorithm, Task Data Sample Module can efficiently increase the data block size without increasing overheads of ACMR when input data has the identical data format. When input data has a change in format, Task Data Sample Module can quickly decrease the data block size for responding to the change.

3.4 Aggregation Decision Module

Aggregation Decision Module is used to determine the way to process intermediate data of the next data block, i.e. aggregating intermediate data of the next data block or not. To this end, Aggregation Decision Module makes the decision according to performance metrics of the next data block deduced by Performance Prediction Module. Because the deductive performance metrics have the processing time and the intermediate data size of the next data block processed by a Mapper with and without a combiner, Aggregation Decision Module can easily determine which way benefits performances by comparing the deductive performance metrics.

Aggregation Decision Module uses Eqs. 2 and 3 to determine the way to process intermediate data of the next data block. Aggregation Decision Module assumes that a Mapper processes the next data block with a combiner to cost T_0 seconds and generate S_0 byte intermediate data while a Mapper processes the next data block without a combiner to cost T_1 seconds and generate S_1 byte intermediate data. Obviously, Aggregation Decision Module should determine to process intermediate data of the next data block with a combiner if both T_0 and S_0 are smaller than T_1 and S_1, but determine to process intermediate data of the next data block without a combiner if both T_1 and S_1 are smaller than T_0 and S_0. If applying T_0, T_1, S_0, and S_1 to Eq. 2 can make W(T_0,T_1) equal to W(S_0,S_1), in other words, Aggregation Decision Module can get 0 in P to choose Path 0 in Fig. 4 but get 1 in P to choose Path 1 in Fig. 4 according to Eq. 3. In Eq. 3, Aggregation Decision Module determines to process intermediate data of the next data block with a combiner if getting 0 in P, but determines to process intermediate data of the next data block without a combiner if getting 1 in P.

$$w(x, y) = \begin{cases} 0\ if & x - y < 0 \\ 1\ otherwise \end{cases} \tag{2}$$

Author's personal copy

$$P = \begin{cases} w(T_0, T_1) & if \ w(T_0, T_1) \equiv w(S_0, S_1) \\ w(N_r, N_d) & otherwise \end{cases} \quad (3)$$

Aggregation Decision Module may have a situation that can not determine the way to process intermediate data of the next data block according to performance metrics, for example, when T_0 is smaller than T_1 but S_0 is larger than S_1. At that time, Aggregation Decision Module simply refers to the number of intermediate data files (denoted by N_d) and the number of current idle Reducers (denoted by N_r), because an intermediate data file is usually delivered to a single Reducer as input. Aggregation Decision Module determines to aggregate intermediate data of the next data block in order to alleviate workloads of Reducers if intermediate data files are more than current idle Reducers. Conversely, Aggregation Decision Module determines not to aggregate intermediate data of the next data block if intermediate data files are less than current idle Reducers. After applying N_d and N_r to Eq. 3, Aggregation Decision Module can get 0 in P to choose Path 0 in Fig. 4 but get 1 in P to choose Path 1 in Fig. 4.

3.5 Performance Prediction Module

Performance Prediction Module can predict performance metrics of the next data block when it is processed by a Mapper with and without a combiner. Performance Prediction Module uses the Single Exponential Smoothing (SES) algorithm [19] to predict performance metrics of the next data block and delivers the performance metrics to Aggregation Decision Module for determining the way to process intermediate data of the next data block. To this end, Performance Prediction Module records the last data block size, the last processing time, and the last intermediate data size due to the processing of a data block by a Mapper with and without a combiner. With the knowledge of costs in processing a recent data block with and without a combiner (i.e. Path 0 and Path 1 in Fig. 4), Performance Prediction Module can easily predict performance metrics of the next data block having a different size.

$$P_{t+1} = \alpha R_t + (1 - \alpha) P_t \; 0 \leq \alpha \leq 1, \quad (4)$$

where R_t is the current value, P_t is the previously predicted value, and P_{t+1} is the predicted value.

$$T = \frac{T_{t+1}}{LastDataBlockSize}, \quad S = \frac{S_{t+1}}{LastDataBlockSize}, \quad (5)$$

where T is the predicted processing time normalized by LastDataBlockSize and S is the predicted intermediate data size normalized by LastDataBlockSize.

Performance Prediction Module uses Eqs. 4 and 5 to predict performance metrics of the next data block processed by a Mapper with and without a combiner. Firstly, Performance Prediction Module uses Eq. 4 to respectively predict the processing time and the intermediate data size of the next data block. Next, Performance Prediction Module uses Eq. 5 to normalize performance metrics with the last data block size because the last data block respectively processed by a Mapper with and without a combiner may have a different size. After predicting performance metrics, Performance Prediction Module delivers performance metrics to Aggregation Decision Module for determining the way to process intermediate data of the next data block.

In Eq. 4, Performance Prediction Module can be sensitive to the change of performance metrics if α is small. If α is large, conversely, Performance Prediction Module can be stable to the temporal change of performance metrics. Although Performance Prediction Module can merely leave the value of α to the ACMR developer according to the definition of the SES algorithm, we implement Performance Prediction Module to automatically correct α based on the real performance metrics of the last data block. In other words, Performance Prediction Module is implemented in the prototype to adjust α each time performance metrics of a data block are recorded.

4 Implementation

4.1 MapReduce in PHP (PHPMR)

Although Hadoop [13,14] is a well-known MapReduce system, we implement ACMR in a MapReduce system written in PHP (short for Hypertext Preprocessor) [10–12] instead of implementing it in Hadoop because: (1) PHP is a general-purpose scripting language widely used on the web design [12] with features of portability and object-oriented programming, which is comparable to Java [11,20] used in Hadoop; (2) Hadoop includes a database (HBase) and a distributed file system (HDFS), which may prevent us from clearly observing ACMR performances; (3) PHP works in a PHP engine widely supported by most modern operating systems such as Unix and Windows while Hadoop currently only works in Unix. We install an Apache web server [21] and a PHP engine [10] in Windows 7 and refer to the MapReduce system written in PHP as PHPMR in the following text of this paper.

PHPMR uses a base class to implement most functions of a MapReduce system such as loading input data, splitting input data into data blocks, running Mappers or Reducers in computers in a cloud, forwarding intermediate data from Mappers to Reducers, and collecting outputs of Reducers as the final result. Besides, PHPMR allows a programmer to configure parameters for an application, e.g., target IP addresses of computers in a cloud, the initial data block size, numbers of Mappers in specific computers, numbers of Reducers in specific computers, and file names having input data. PHPMR

Author's personal copy

Table 1 PHPMR API

API name	Description
Function Mapper (string $Filename, string Chunk	A Map Function to Receive a File Name and a Chunk as Parameters
Function Reducer (string $Key, string $Value)	A Reduce Function to Receive a Key and an Array of Value as Parameters
Function Emit (string $Key, string $Value)	A Emit Function Capable of Sending Intermediate Data or Result

needs a programmer to develop a class that inherits the base class and implements a Mapper and a Reducer as member functions that can be called at run time. Besides all APIs (short for Application Programming Interfaces) [22] of the PHP engine available to a programmer, PHPMR provides a programmer with APIs related to MapReduce, e.g. API Emit for outputting intermediate data in a Mapper and the results in a Reducer.

In PHPMR, A programmer can develop applications according to APIs in Table 1. A programmer needs to develop a function named Mapper that accepts a file name of input data and a data block (i.e. a part of input data in the file) as the parameters. In the Mapper (i.e. API Mapper in Table 1), a programmer can process the data block and use API Emit to output pairs of a key and a value that will be saved in temporary files later by PHPMR. Besides, a programmer needs to develop a function named Reducer that accepts a key and an array of the associated values as the parameters. In the Reducer (i.e. API Reducer in Table 1), a programmer can use API Emit to output the results after processing values associated with the key. When developing applications in PHPMR, a programmer neither cares about how to distribute or run Mappers and Reducers in a cloud nor manually moves intermediate data from a Mapper to a Reducer, because PHPMR can automatically deal with the issues on behalf of the programmer. Accordingly, a programmer can easily develop applications in PHPMR as if he or she were developing applications in other MapReduce systems.

4.2 ACMR in PHPMR

We implement ACMR in PHPMR as shown in Fig. 5. In PHPMR, we read input data from input files and make input data go through Task Data Sample Module and Aggregation Decision Module. With such a design, we can split input data into data blocks according to the data block size configured by Task Data Sample Module. Besides, we can configure a Mapper whether to deliver intermediate data to a combiner or not according to the decision of Aggregation Decision Module. Currently, we simply read a global variable set by Aggregation Decision Module to determine whether intermediate data should go through a combiner or not. We deploy Performance Prediction Module at the outputs of a Mapper and a Combiner in order to record the processing time and the intermediate data size. By deploying Performance Prediction Module at the location, we not only can deliver performance metrics to Aggregation Decision Module but also can save intermediate data in temporary files.

Because the same code typically can be used in a combiner and a Reducer [1], we do not implement a combiner with the code different to a Reducer for an application. We know that a Mapper eventually needs to call API Emit to emit intermediate data to the system, so we merely change the way of processing outputs when API Emit is called in a Mapper. If the ACMR determines to process intermediate data of a data block with a combiner, we do a workaround in API Emit by delivering intermediate data to the processing of a Reducer locally and then emit its outputs to temporary files storing intermediate data.

ACMR can be easily implemented at other MapReduce systems if the system can offer the following services and information. First, the system should have a function capable of splitting input data into data blocks according to Task Data Sample Module before delivering the data blocks to a Mapper. Second, the system should call Aggregation Decision Module for determining whether intermediate data of the next data block should go through a combiner or not. Third, the system should notify Task Data Sample Module of the decision made by Aggregation Decision Module about whether intermediate data of the next data block should go through a combiner or not. Forth, the system should record the last data block size, the last processing time, and the last intermediate data size of a data block processed by a Mapper (with or without a combiner according to Aggregation Decision Module), and then applies them to Performance Prediction Module for predicting performance metrics of the next data block. Fifth, the system should deliver performance metrics deduced by Performance Prediction Module to Aggregation Decision Module for determining the way to process intermediate data of the next data block.

5 Performance observation

5.1 Experiment environment

We construct a cloud composed of 22 identical computers where one computer plays the role of Master responsible for accepting commands from a user, dispatching tasks to Map-

Author's personal copy

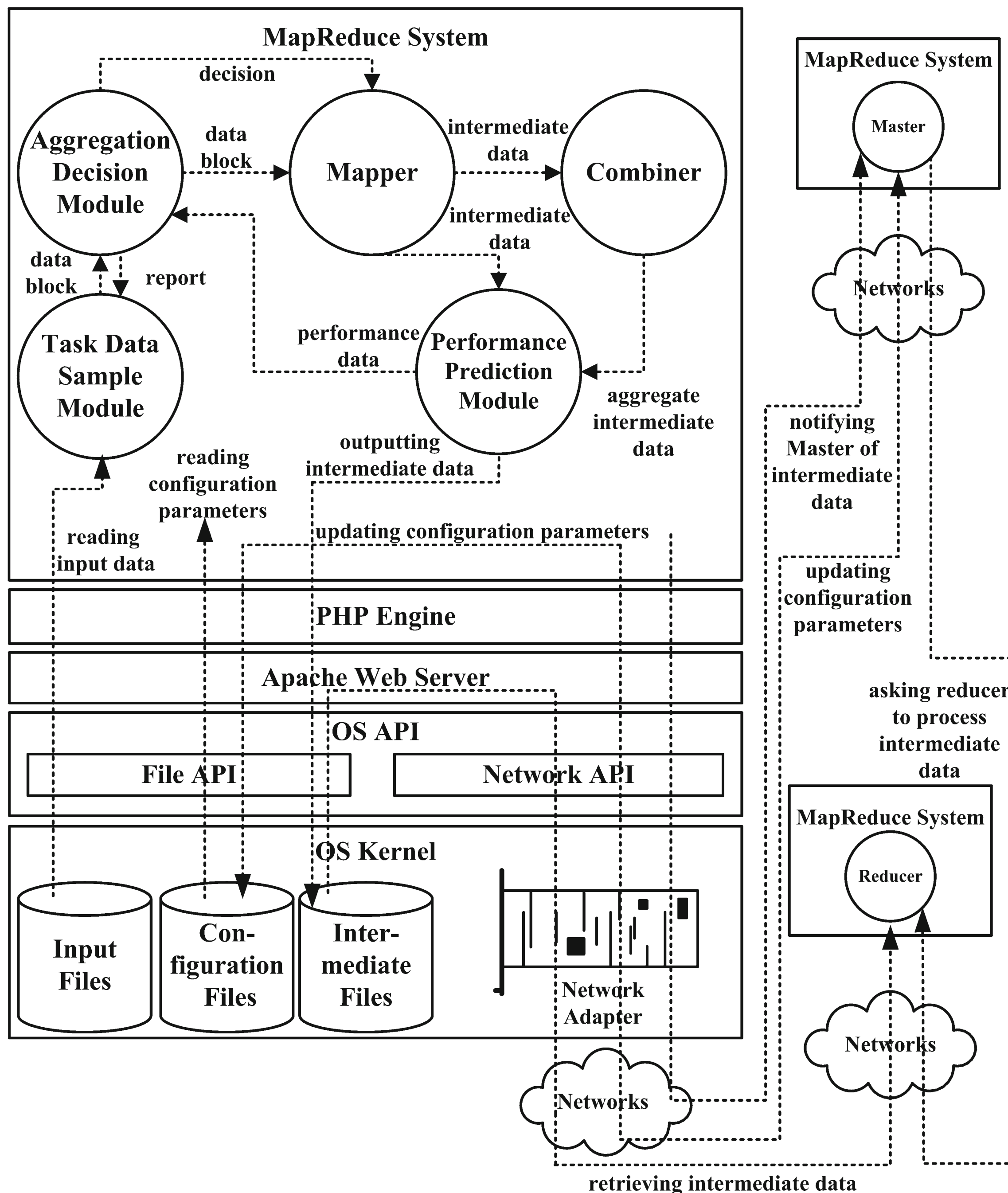
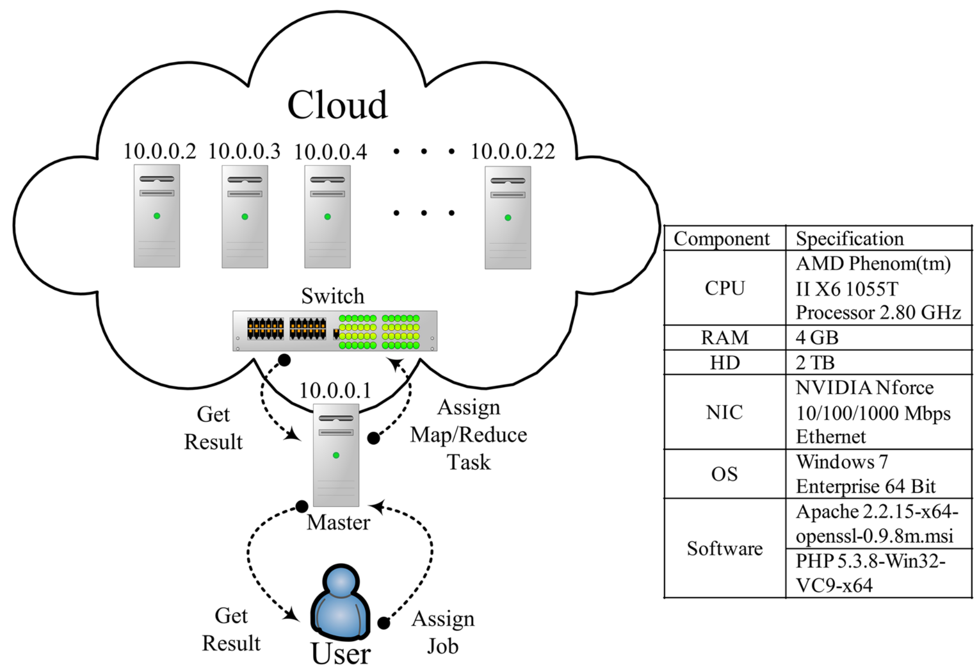


Fig. 5 ACMR in PHPMR

pers or Reducers in other computers, and collecting outputs of Reducers as the final result. We use a Gigabyte Ethernet switch to connect the computers in Fig. 6 and install Windows 7, an Apache web server, and a PHP engine to all computers. We detail hardware and software specifications of the computers at the right side of Fig. 6.

Besides, we install PHPMR in all computers in Fig. 6 and conduct tests with ACMR in PHPMR. First, we observe performance impacts of α in Eq. 4 used by Performance Prediction Module and chunk size used by Task Data Sample Module with different change intervals. Second, we observe execution time of seven applications [23] in ACMR and other systems that always use a combiner or no combiner. We simply enable or disable a combiner in PHPMR for emulating other systems so that the performance comparisons are made at the same base. In all experiments, we feed applications with input data stored in files in a computer, i.e. Master in Fig. 6.

Author's personal copy



Component	Specification
CPU	AMD Phenom(tm) II X6 1055T Processor 2.80 GHz
RAM	4 GB
HD	2 TB
NIC	NVIDIA Nforce 10/100/1000 Mbps Ethernet
OS	Windows 7 Enterprise 64 Bit
Software	Apache 2.2.15-x64-openssl-0.9.8m.msi
	PHP 5.3.8-Win32-VC9-x64

Fig. 6 Experiment network topology

5.2 Performance impact of α chunk size, and change interval

We observe impacts of α chunk size, and change interval on performances of three applications, i.e., Word Count [1], Bubble Sort [24], and K-Means [25], because: (1) Word Count should aggregate intermediate data with many repeated words in input data, (2) Bubble Sort should distribute intermediate data over Reducers as soon as possible without aggregating intermediate data, and (3) K-Means does most tasks in Mappers to generate intermediate data that is close to the final result, which does not matter with the use of a combiner. We do the tests with 3 computers, i.e. a Master, a Mapper, and a Reducer. For observing performance impacts of α, we set α to 0.1, 0.3, 0.6, 0.9, and self-correcting. By setting α to self-correcting, i.e. the default implementation of current ACMR prototype, we make Performance Prediction Module automatically correct α based on the real performance metrics of the last data block.

Because the AIMD algorithm in Task Data Sample Module increases a data block by a chunk size or decreases a data block by a half, we observe impacts of different chunk sizes on performances. We use 4 chunk sizes as units for Task Data Sample Module to adjust the data block size. We make the three applications handle 48 MB input data. However, we prepare input data with 4 change intervals (i.e. the size to keep a specific data format) for each of the three applications. In each test, we feed an application with a specific change interval. In Word Count, we make input data change between having low-repeated words and having high-repeated words. In 2 MB change interval, for example, we observe Word Count to handle input data that has 2 MB low-repeated words followed by 2 MB high-repeated words, and so on. In Bubble Sort (sorting numbers by an increasing sequence), we make input data change between a decreasing sequence and an increasing sequence. In 2 MB change interval, for example, we observe Bubble Sort to handle input data that has a 2 MB decreasing sequence followed by a 2 MB increasing sequence, and so on. In K-Means, we make input data change between having low-repeated numbers and having high-repeated numbers. In 2 MB change interval, for example, we observe K-Means to handle input data that has 2 MB low-repeated numbers followed by 2 MB high-repeated numbers, and so on.

According to Fig. 7, ACMR hardly has a performance difference with a different α in Word Count because using a combiner to aggregate intermediate data having high-repeated keys can get a much better performance than bypassing a combiner. At the premise of having a great performance distance between using a combiner and using no combiner, ACMR always deduces performance metrics of the next data block to indicate that aggregating intermediate data is better, no matter what value α has in the SES algorithm used by Performance Prediction Module. When a chunk size is larger, ACMR has a slightly worse performance because one of the first two data blocks is processed by a Mapper without a combiner in the initial time to degrade the overall performance, even though subsequent data blocks are destined to a Mapper with a combiner. When the change interval is longer, ACMR has a slightly worse performance as well because the front part of input data has more low-repeated words to not only invalidate the effect of intermediate data

Author's personal copy

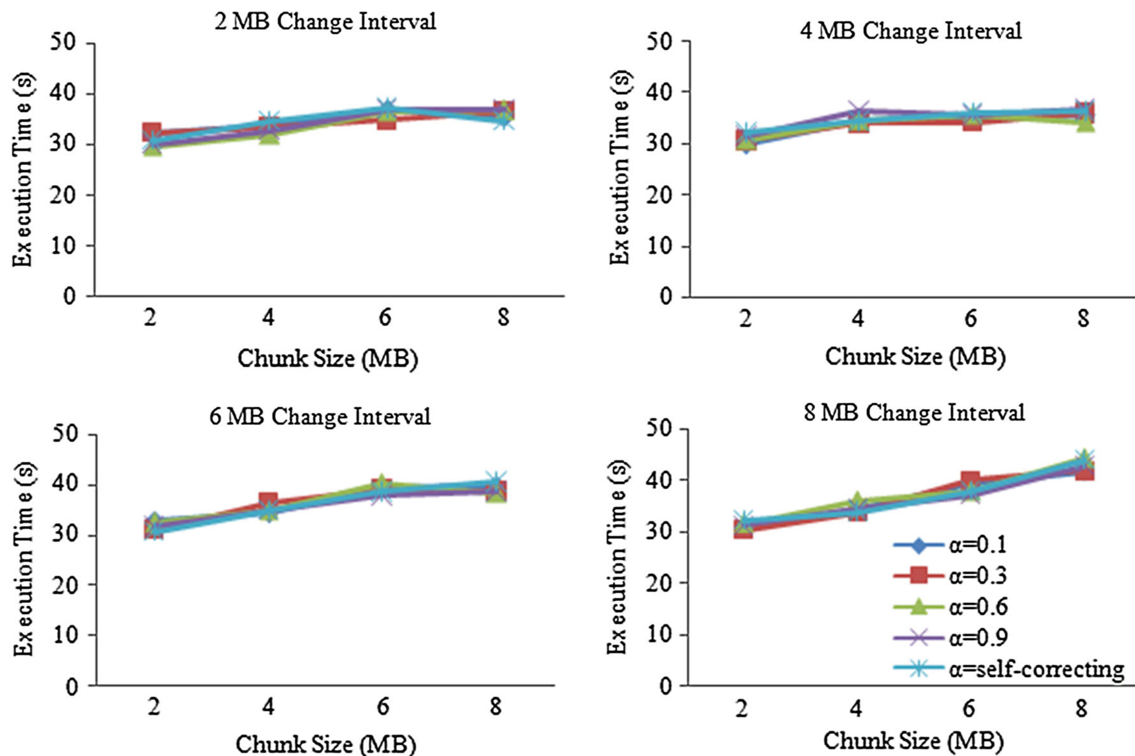


Fig. 7 Performance impact of α chunk size, and change interval on word count

aggregation but also incur an extra overhead from a combiner.

According to the curves overlapping with each other in Fig. 8, ACMR has no performance difference with a different α in Bubble Sort. Because Bubble Sort does not generate intermediate data having high-repeated keys, ACMR gets a great difference in observing performance metrics of the first two data blocks in the initial time. No matter what value α has, ACMR always predicts that performance metrics of the next data block without a combiner are much better than those of the next data block with a combiner. Accordingly, ACMR keeps choosing to bypass the processing of a combiner for subsequent data blocks. With a larger chunk size, ACMR has a worse performance because one of the first two data blocks is processed by a Mapper with a combiner in the initial time to greatly degrade the overall performance, especially for the CPU-bound application that greatly relies on the computation of Mappers. When the change interval is longer, ACMR has a worse performance as well because the front part of input data has more decreasing sequence to not only invalidate the effect of intermediate data aggregation but also incur an extra overhead from a combiner.

Similar to the previous experiments, ACMR does not get much performance difference with a different α in Fig. 9 because K-Means does most tasks in Mappers to generate intermediate data that is close to the final result, which does not matter with the use of a combiner. However, ACMR still needs α for the SES algorithm used by Performance Prediction Module. Besides, ACMR does not show any performance difference on K-Means with the change of chunk size and interval, because performances of K-Means have more relation with the first point randomly chosen as the group center than the way to process intermediate data. Accordingly, ACMR does not get a curve that is linear or in proportion to the change of α Chunk Size, and Change Interval.

Although we conclude in previous experiments that performances are hardly affected by α, we have a further observation on behaviors of ACMR with the three applications. We use the same experiment configuration and show the results in Fig. 10. In the y-axis, we use 1 to denote that intermediate data of a data block is going through a combiner, but use 0 to denote that intermediate data of a data block is bypassing a combiner. We use the x-axis to denote the execution time of an application in ACMR.

In Word Count, we observe that ACMR uses no combiner to process intermediate data of the first data block (i.e. the first point having 0 at y-axis) and then uses a combiner to process intermediate data of the second data block (i.e. the second point having 1 at y-axis) in the initial time. After that, we observe that ACMR keeps using a combiner to process

Author's personal copy

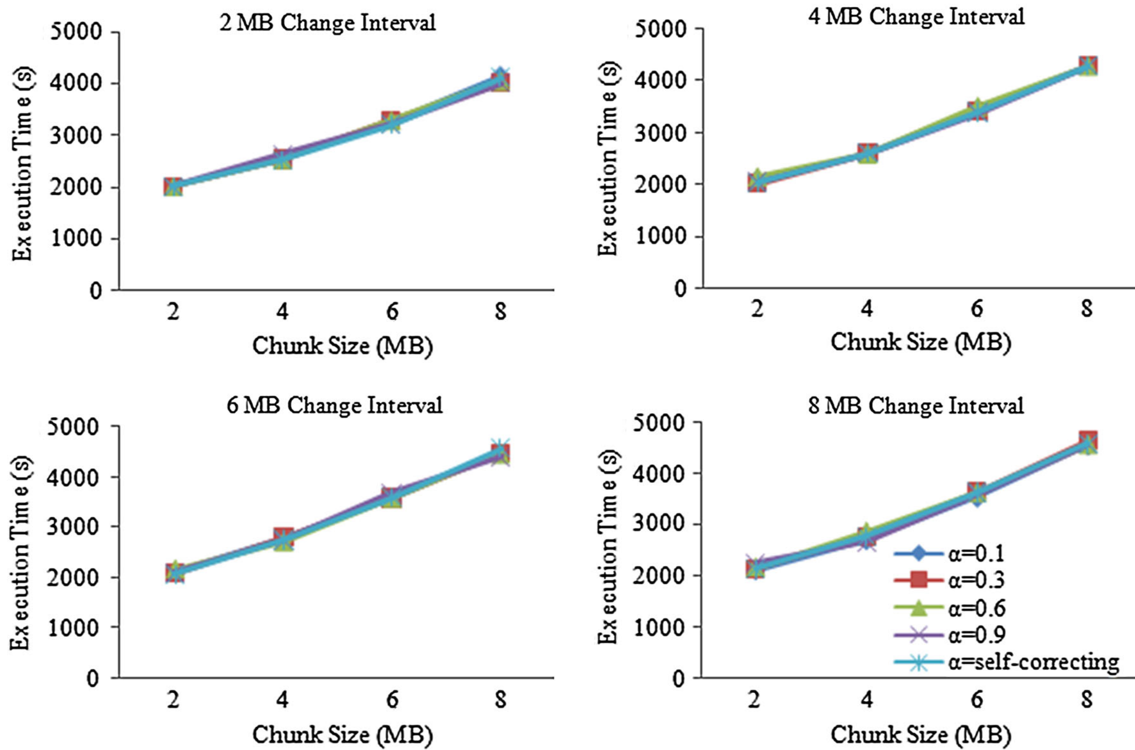


Fig. 8 Performance impact of α chunk size, and change interval on Bubble Sort

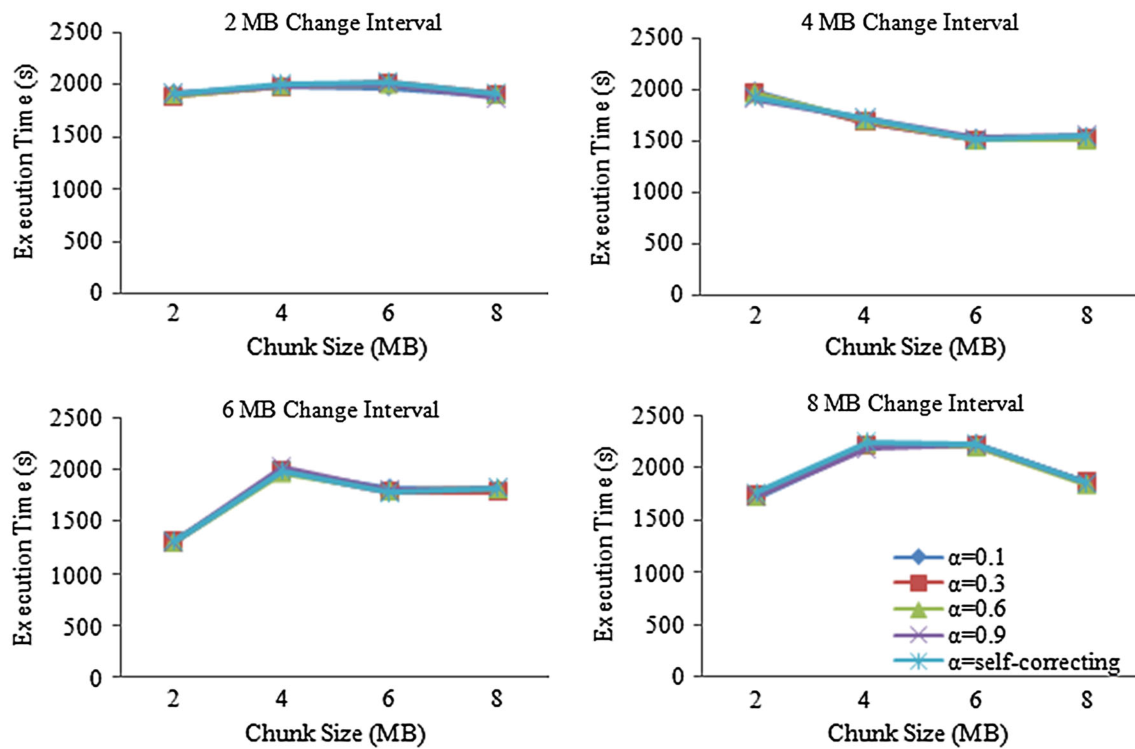


Fig. 9 Performance impact of α chunk size, and change interval on K-Means

Author's personal copy

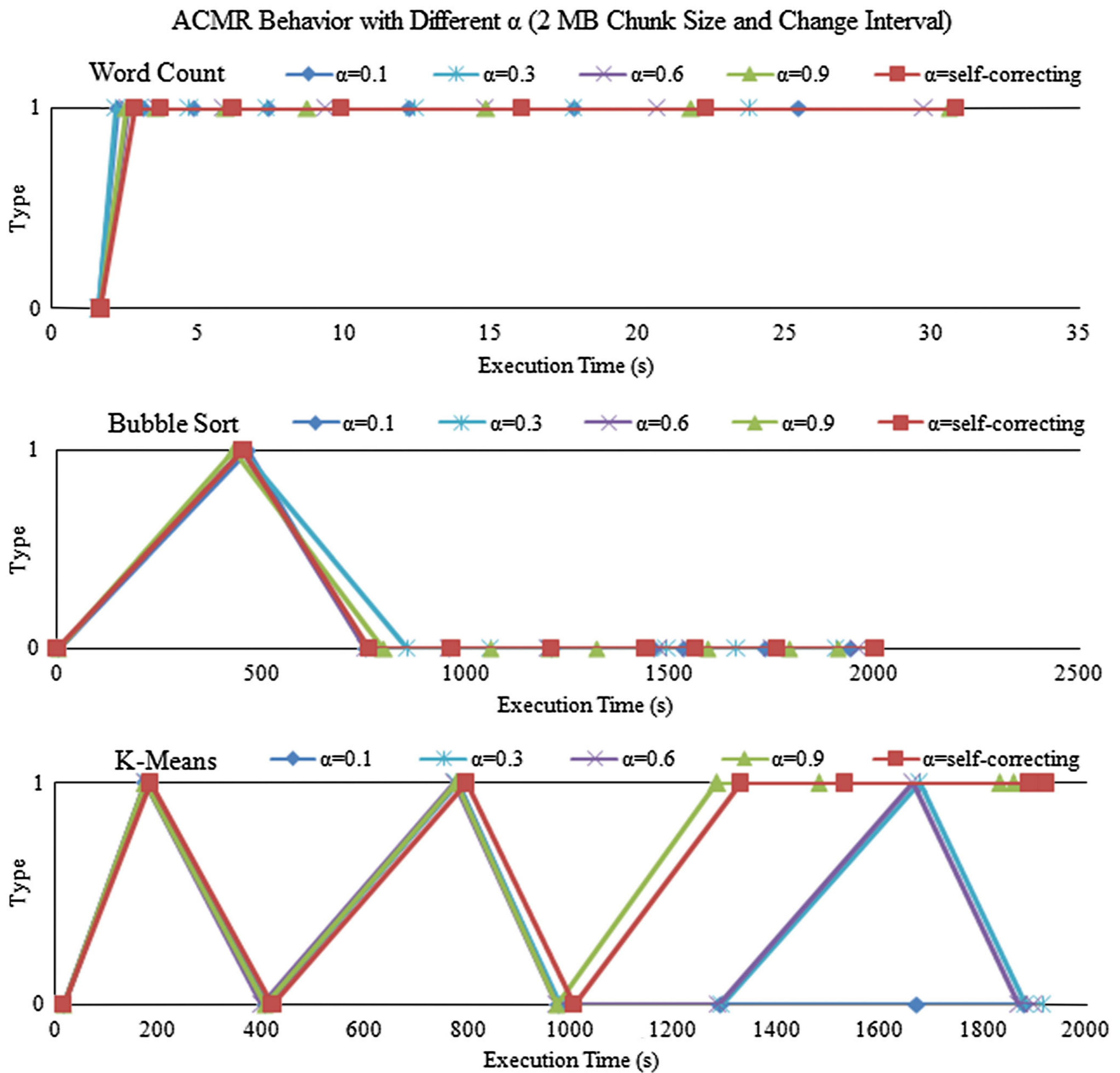


Fig. 10 ACMR behavior with different α (2 MB chunk size and change interval)

intermediate data of subsequent data blocks (i.e. the subsequent points having 1 at y-axis), no matter what value α has. In Bubble Sort, we observe that ACMR uses no combiner to process intermediate data of the first data block (i.e. the first point having 0 at y-axis) and then uses a combiner to process intermediate data of the second data block (i.e. the second point having 1 at y-axis) in the initial time. After that, we observe that the curves go down to indicate processing intermediate data of subsequent data blocks without a combiner (i.e. the subsequent points having 0 at y-axis), which corresponds to the feature of Bubble Sort whose intermediate data should bypass a combiner. In K-Means, we observe that ACMR does not have the consistent behavior with a different α because performances of using a combiner are very close to those of using no combiner, which also implies that using a combiner or not does not matter. According to the experiments, we show that ACMR can choose the right way to process intermediate data for applications in MapReduce.

5.3 Performance of word count

In the following experiments, we program several applications with the MapReduce programming model and observe their performances in a cloud. We use the applications to

Author's personal copy

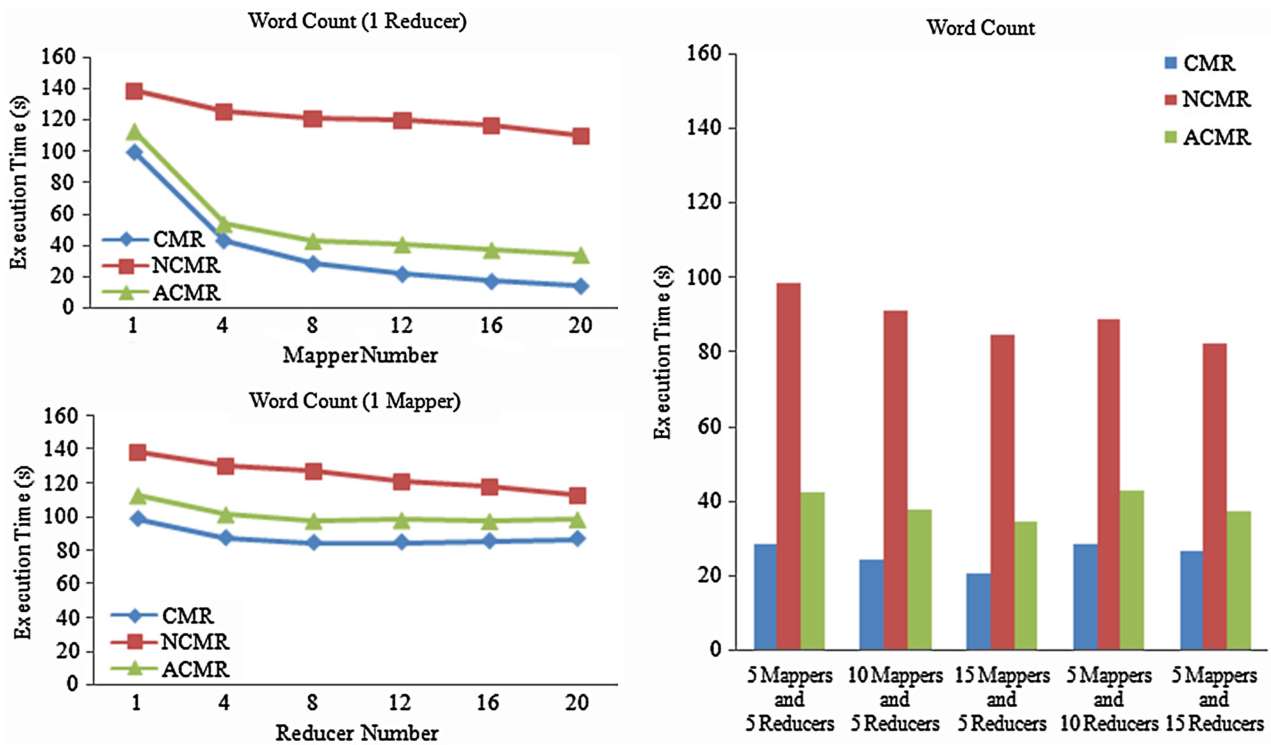


Fig. 11 Performance of word count

test ACMR and take other MapReduce systems for comparisons. We refer to the system that always uses a combiner as CMR and refer to the system that always uses no combiner as NCMR. We design CMR and NCMR to process input data with a fixed 64 MB data block size without any capability like ACMR to dynamically change both the data block size and the way of processing intermediate data. We use the 2 MB chunk size and self-correcting α in ACMR. In each experiment, we detail performances of the 3 systems that use 1 Mapper and 1 Reducer respectively. Then, we verify performances of the 3 systems that use multiple Mappers and Reducers.

Word Count [1] can calculate the number of words separated by a space or newline character in a file. Word Count uses a Mapper to parse data and considers a word to be a key having a value of 1 by generating intermediate data in the format composed of pairs of a word and “1”. Word Count relies on the system to forward intermediate data to different Reducers according to the result of a hash function with the key as the input. Word Count uses a Reducer to merge values associated with the same key by counting them. Word Count may use a combiner to merge values in a Mapper to conserve network bandwidth. In the test, Word Count processes a 128 MB text file.

In the left part of Fig. 11, we observe that increasing Mappers or Reducers can short the processing time. Especially in NCMR, we observe that distributing intermediate data over Reducers can efficiently improve performances because NCMR generates a huge amount of intermediate data. Because intermediate data aggregation can greatly improve performances by reducing network bandwidth consumption, we observe that CMR and ACMR both outperform NCMR in the test. Because CMR and ACMR can alleviate Reducer workloads by aggregating most intermediate data in Mappers, we see in the lower-left part of Fig. 11 that increasing Reducers in CMR and ACMR does not have more performance improvement than increasing Reducers in NCMR. We observe that the systems with multiple Mappers and Reducers have performances in the right part of Fig. 11 corresponding to those in the left part of Fig. 11.

ACMR can get performances close to CMR because ACMR learns to aggregate intermediate data for getting a better performance after comparing performance metrics of the first two data blocks in the initial time. After that, ACMR predicts performance metrics of subsequent data blocks and increases the data block size accordingly. Gradually, ACMR adapts itself to behaving like CMR and has performances much better than NCMR and close to CMR.

5.4 Performance of Grep

Grep [26] can search a line of data for a specific string according to a regular expression. Grep can output the string and the number of the string appearing in the line of data. In the experiment, Grep is implemented to locate each word in a file according to the regular expression “.+”. Grep uses Map-

Author's personal copy

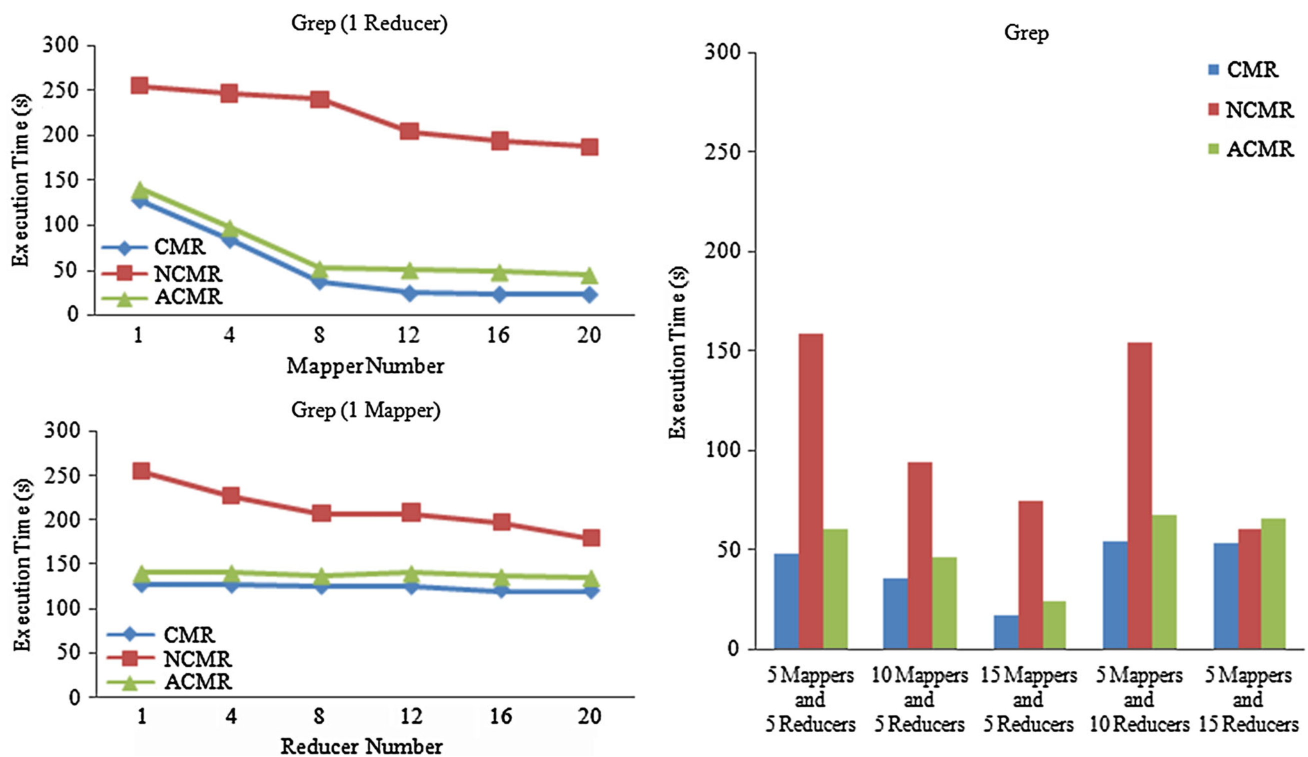


Fig. 12 Performance of grep

pers to parse data with the regular expression and generate intermediate data like Word Count does. Next, Grep uses Reducers to merge intermediate data and output the number of different words. Although being like Word Count, Grep needs more computation power in Mappers than Word Count. In the experiment, Grep processes a 128 MB text file.

We observe that Grep in Fig. 12 has performances similar to Word Count in Fig. 11. Because Grep uses more computation power than Word Count in Mappers for parsing data, we notice that increasing Mappers in Grep can get more performance improvement than Word Count by comparing curves in upper-left parts of Figs. 11 and 12. Like Word Count, we observe that aggregating intermediate data can benefit performances of Grep by decreasing the intermediate data size. We see that CMR and ACMR both outperform NCMR in the experiment. Because ACMR processes the first two data blocks with different ways in the initial time for collecting performance metrics, we observe that the performance of ACMR is slightly worse than that of CMR but is much better than NCMR.

5.5 Performance of Radix Sort

Radix Sort [27] can sort numbers with multiple computers simultaneously after partitioning and grouping them according to their digits. Radix Sort uses Mappers to parse data and group numbers into different sets according to their digits. Radix Sort generates a series of numbers separated by a space character as intermediate data that will be saved into different temporary files according to the digit of the number. If Radix Sort generates "123" and "234" as intermediate data, for example, the system saves the three-digit numbers in the same file separated by a space character. Radix Sort uses Reducers to classify numbers by buckets according to the remainder of each number after divided by ten to the power of its digit minus 1. Finally, Radix Sort outputs numbers in buckets in order as the final result. Radix Sort may use a combiner to sort intermediate data for alleviating Reducer workloads, but has no way to decrease the intermediate data size. In the test, Radix Sort processes a 64 MB text file.

We show the results in Fig. 13. Due to the contribution of sorting intermediate data in a combiner to alleviate Reducer workloads, we see in the upper-left part of Fig. 13 that CMR and ACMR both can outperform NCMR in the experiment of increasing Mappers. However, we notice that increasing Mappers hardly improves performances in all systems because the single Reducer is easily overloaded by intermediate data from Mappers and becomes the performance bottleneck.

When a single Mapper cooperates with multiple Reducers, we observe in the lower-left part of Fig. 13 that CMR outperforms NCMR until that the system has 8 Reducers, because the gain of intermediate data aggregation in a Map-

Author's personal copy

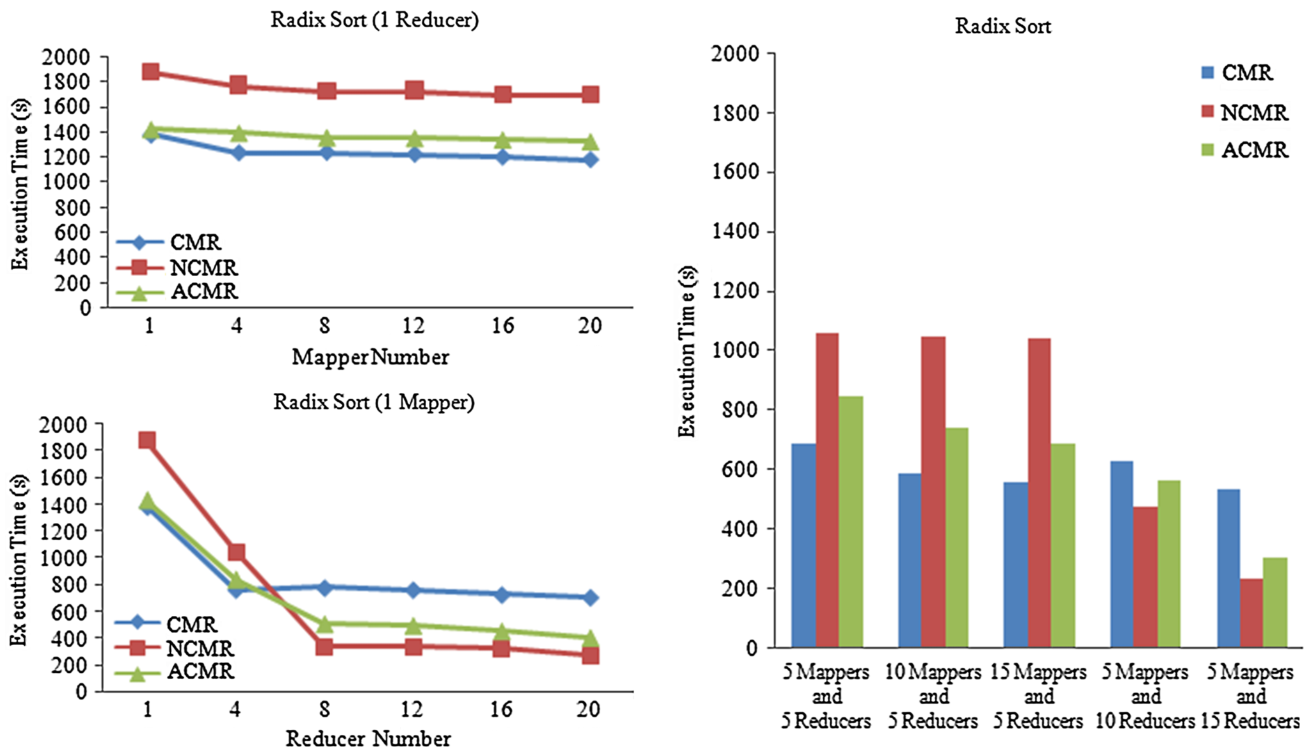


Fig. 13 Performance of Radix Sort

per is more than the delay of distributing intermediate data over Reducers. Once increasing Reducers to 8 instances, conversely, we observe that NCMR outperforms CMR, because the Reducers are enough to consume intermediate data from a Mapper and alleviate Mapper workloads. When Mappers are less than Reducers, we observe that aggregating intermediate data may not always benefit performances due to the underutilization of Reducers. Despite the change of Mapper and Reducer numbers in the experiment, we observe that ACMR still can adjust itself to the change for earning the performance between CMR and NCMR. According to the right part of Fig. 13, we confirm the capability of ACMR with multiple Mappers and Reducers.

5.6 Performance of Bubble Sort

Like Radix Sort, Bubble Sort [24] can sort numbers with multiple computers simultaneously after partitioning and grouping them according to their digits. In Mappers, Bubble Sort can do exactly as Radix Sort. Instead of sorting numbers with buckets, however, Bubble Sort uses double loops to swap numbers over each other in Reducers which have more complexity and cost more computation power in comparison to Radix Sort. Finally, Bubble Sort collects outputs of Reducers as the final result. Bubble Sort can sort numbers in a combiner to alleviate Reducer workloads, but can not decrease the intermediate data size. In the test, Bubble Sort processes a 64 MB text file.

In the upper-left part of Fig. 14, we observe that increasing Mappers can improve performances of CMR and ACMR because CMR and ACMR both have combiners in Mappers to alleviate Reducer workloads. We notice that increasing Mappers hardly has a positive effect on performances of NCMR because NCMR has no combiner in Mappers and easily overloads a single Reducer with intermediate data from the Mappers. By comparing upper-left parts of Figs. 13 and 14, we notice that Bubble Sort can get more performance improvement than Radix Sort with the increase of Mappers because Bubble Sort uses more computation power than Radix Sort in combiners that can be greatly beneficial to performances. In the experiment of increasing Mappers, we see that ACMR can get the performance close to CMR and much better than NCMR.

When increasing Reducers in the lower-left part of Fig. 14, we observe that CMR can not get performance improvement because most Reducers are idled to wait intermediate data from the single Mapper, a performance bottleneck in the system. Conversely, we notice that NCMR can get benefits from the increase of Reducers because the single Mapper can share workloads with Reducers by distributing intermediate data over Reducers as soon as possible. Nevertheless, we still observe that ACMR can get the performance close to NCMR and better than CMR with the increase of Reducers. Accord-

Author's personal copy

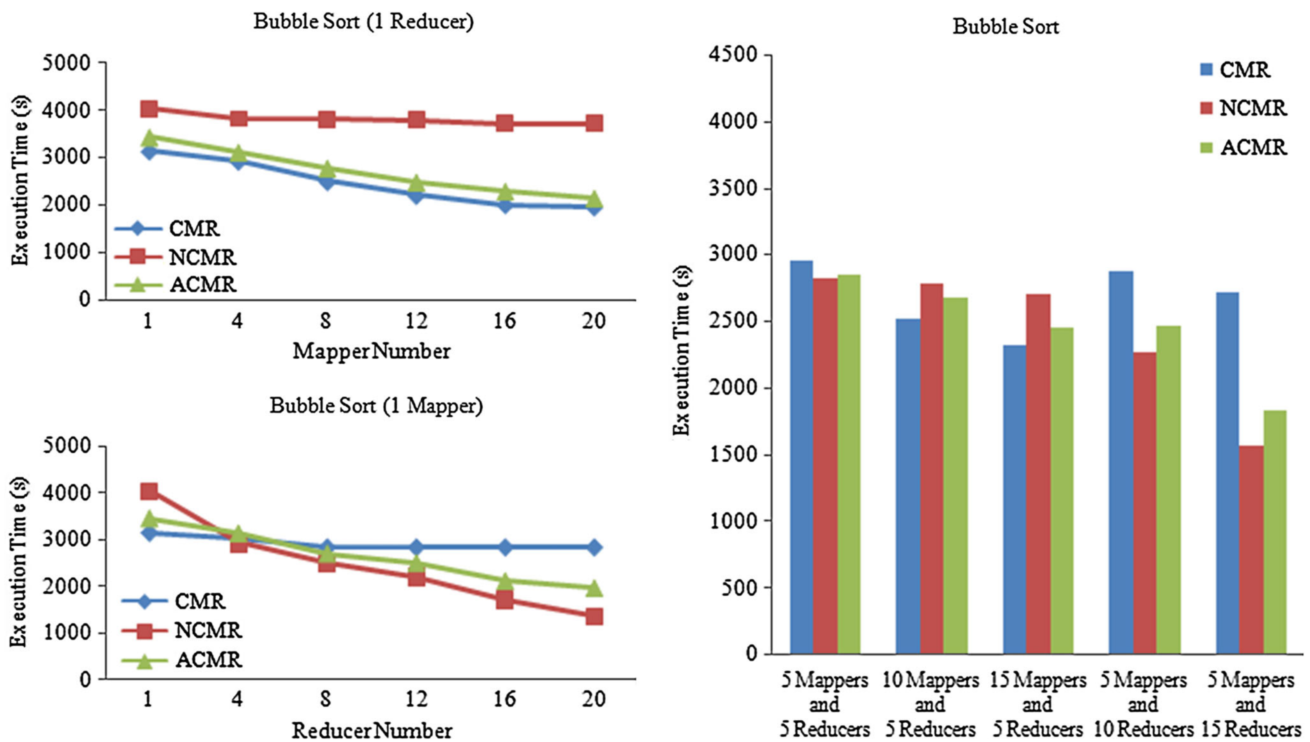


Fig. 14 Performance of Bubble Sort

ing to the right part of Fig. 14 where a system has multiple Mappers and Reducers, we can get the same conclusion as well.

5.7 Performance of K-Means

K-Means [25] can partition numbers into k clusters. In the experiment, K-Means randomly selects k numbers from input data as initial means in Mappers. Next, K-Means repeats the procedures of calculating the distances between a number and each of the means, associating a number to a cluster according to the nearest mean, and replacing a mean with the centroid of the cluster, until that all numbers are classified into clusters without any change. In Mappers, K-Means generates pairs of the cluster identification and the number belonging to the cluster as intermediate data. Finally, K-Means uses Reducers to merge intermediate data and output the final result. In the experiment, K-Means partitions numbers into clusters corresponding to the number of Reducers in a cloud. K-Means may merge intermediate data in a combiner to slightly reduce intermediate data and alleviate some Reducer workloads. In the experiment, K-Means processes a 64 MB text file

In the upper-left part of Fig. 15, we observe that increasing Mappers can improve performances for all systems until 16 Mappers. When a system has more than 16 Mappers, we see no performance improvement because the single Reducer is too busy to consume intermediate data from Mappers. Because merging intermediate data in a combiner can only slightly reduce intermediate data and alleviate some Reducer workloads, we observe that CMR and ACMR do not outperform much NCMR. Because a single Mapper is easily overloaded to become a performance bottleneck, we can not improve performances by increasing Reducers for all systems in the lower-left part of Fig. 15. We get the similar observations in the right part of Fig. 15 where a system has multiple Mappers and Reducers. However, we observe that ACMR still has performances comparable to CMR and NCMR in the experiment.

5.8 Performance of Inverted Index

Inverted Index [28] can locate all URLs in a document. In MapReduce, Inverted Index uses Mappers to parse data and generate a series of pairs of a URL and a position as intermediate data. Usually, Inverted Index costs much computation power in Mappers to search input data for URLs according to a complicated regular expression. In Reducers, Inverted Index can merge intermediate data by gathering and sorting positions of the same URL as the final result. Inverted Index may use a combiner to greatly reduce intermediate data because using URLs as keys usually occupies much space. In the test, Inverted Index processes a 128 MB text file

In the upper-left part of Fig. 16, we observe that performances of a system can be improved with the increase of

Author's personal copy

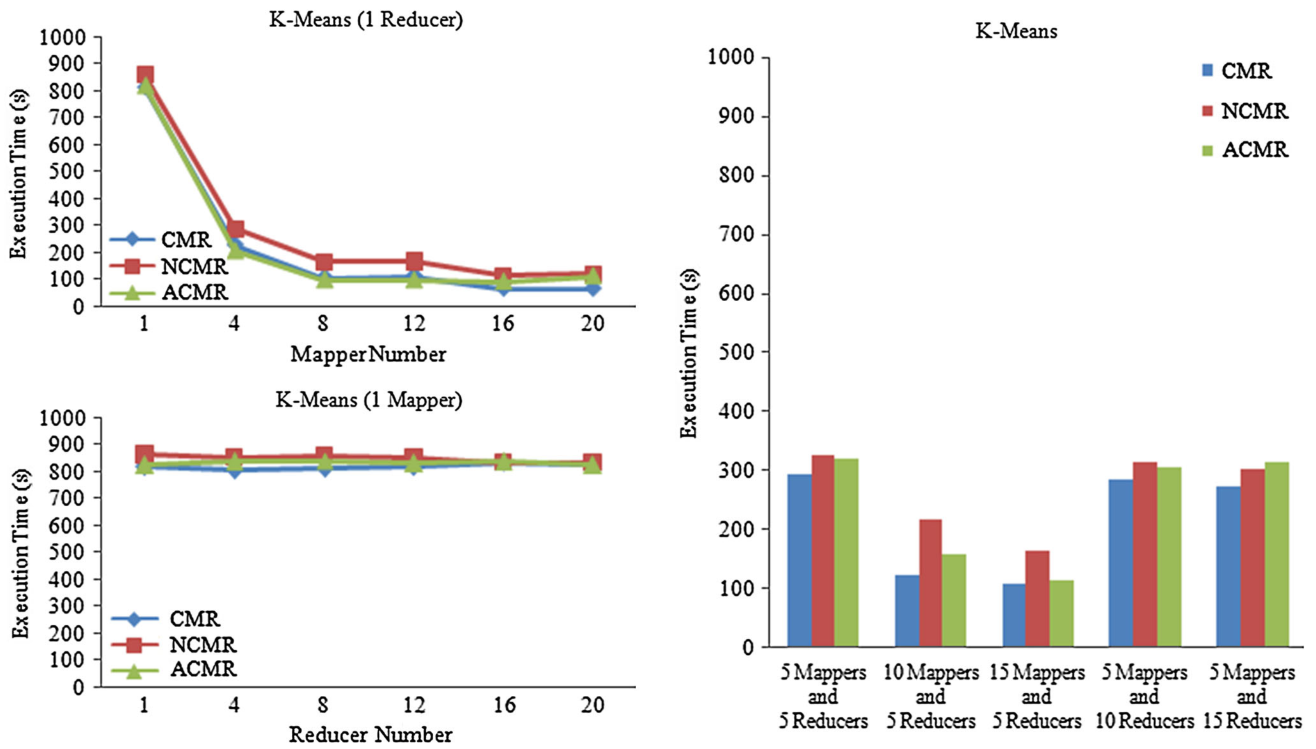


Fig. 15 Performance of K-Means

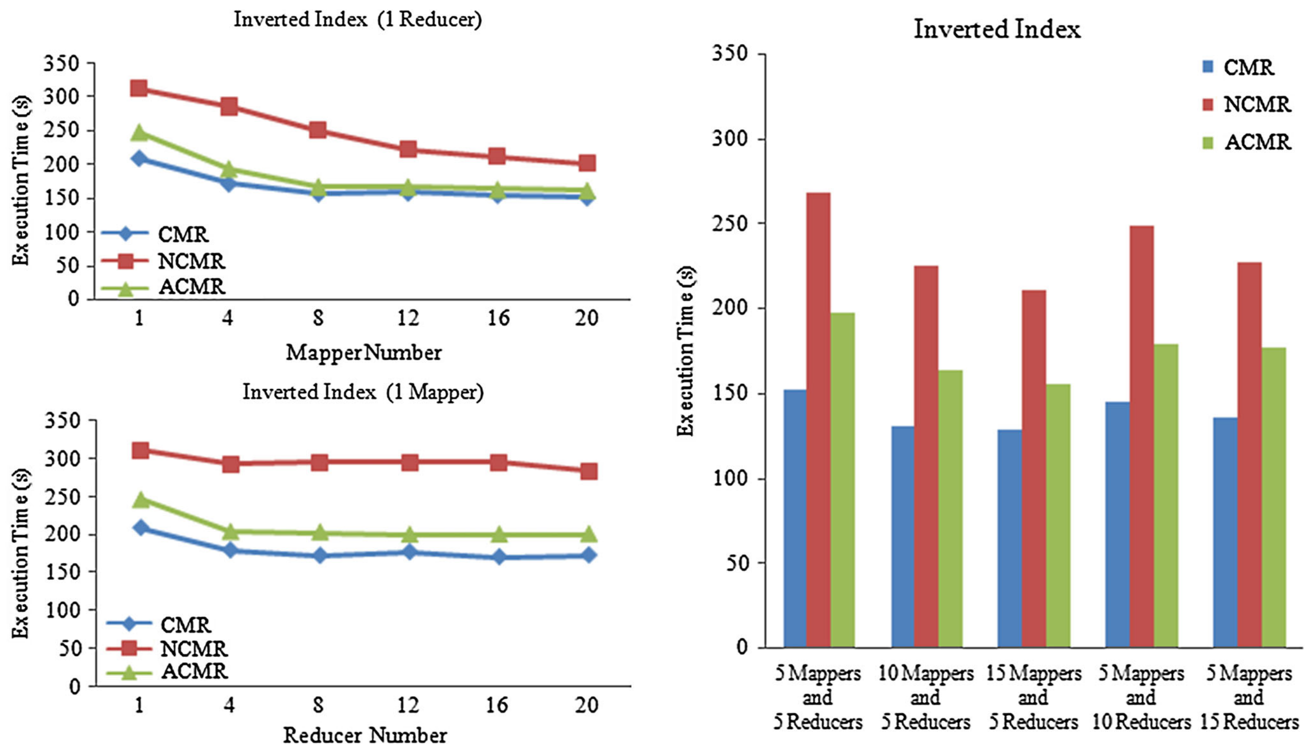


Fig. 16 Performance of Inverted Index

Author's personal copy

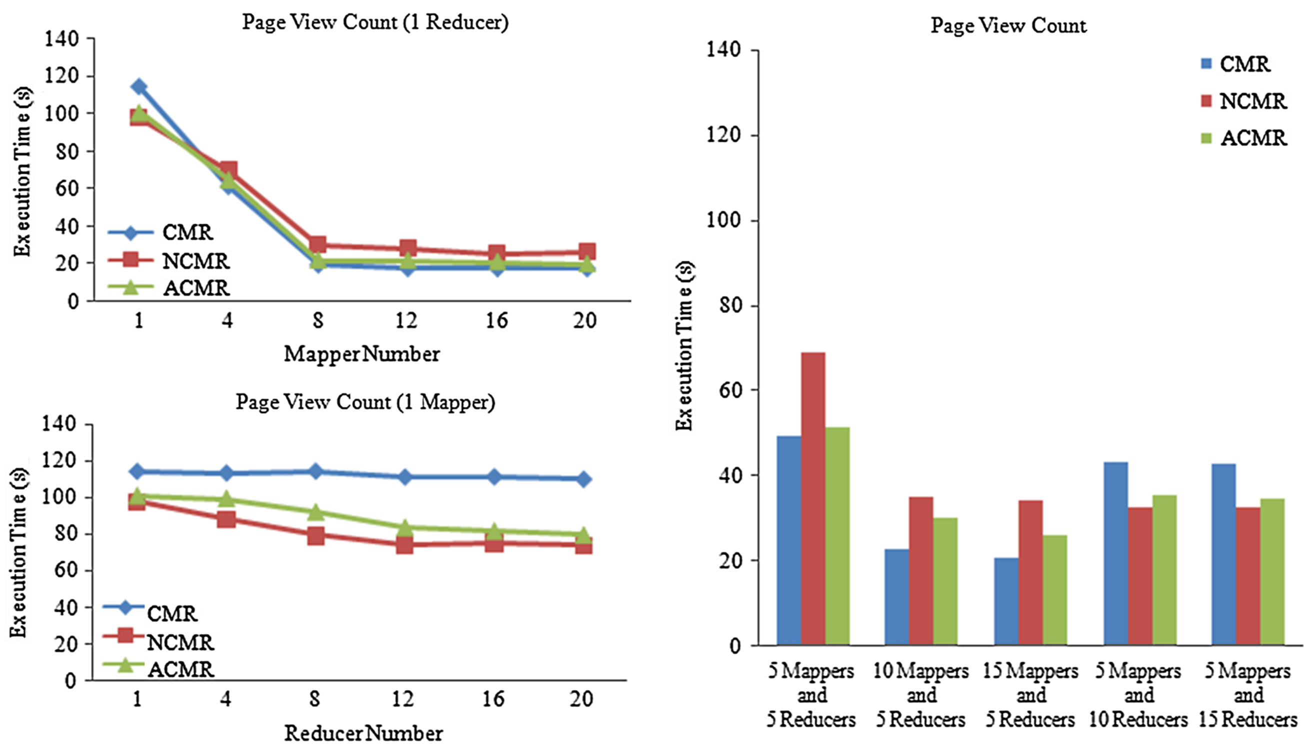


Fig. 17 Performance of Page View Count

Mappers until 12 Mappers are used. When a system has 12 Mappers, we see that the single Reducer is too busy to consume intermediate data generated by Mappers. Because intermediate data aggregation can remove redundant keys in intermediate data to greatly conserve network bandwidth, we observe that CMR and ACMR both strongly outperform NCMR. Similar to the contribution of intermediate data aggregation in conserving network bandwidth, we observe that CMR and ACMR both strongly outperform NCMR too in the lower-left part of Fig. 16 where all systems have Reducers to consume intermediate data from a single Mapper. In the experiment, we show that ACMR can adjust itself to the application and earn performances comparable to CMR and much better than NCMR. When a system has multiple Mappers and Reducers in the right part of Fig. 16, we still have the same observation.

5.9 Performance of Page View Count

Page View Count [28] can analyze a web log file to get the statistic of each URL. Similar to Word Count in counting data, Page View Count counts URLs instead of words. Because of using Mappers to parse data and generate a series of pairs of an URL and an IP address accessing the URL as intermediate data, Page View Count needs one regular expression for locating an URL and another one for locating an IP address in input data. Accordingly, Page View Count costs much more computation power than Word Count in Mappers. Page View Count uses Reducers to count the times of accessing a URL. Page View Count may aggregate intermediate data with a combiner to greatly decrease the intermediate data size because repeated URLs usually occupy much space. In the test, Page View Count processes a 128 MB text file

In the upper-left part of Fig. 17, we observe that all systems can have performance improvement with the increase of Mappers until 8 Mappers are used. When using more than 8 Mappers, we observe that a single Reducer can not consume intermediate data generated by Mappers and the performance can not be further improved with the increase of Mappers. Although aggregating intermediate data can remove redundant URLs in intermediate data to greatly conserve network bandwidth, we do not observe the obvious performance difference between CMR and NCMR because a combiner has to compete with a Mapper for computation power and because the gain of network bandwidth conservation is neutralized by the delay of locating URLs and IP addresses in input data.

In the lower-left part of Fig. 17, we notice an interesting phenomenon that NCMR outperforms CMR. Because aggregating intermediate data in a combiner and locating URLs and IP addresses in a Mapper both need much computation power, we observe that the single Mapper is terribly overloaded by the high computation tasks to become a performance bottleneck. Accordingly, we observe that the CMR performance lags the NCMR performance in the exper-

Author's personal copy

iment. Because CMR always enables combiners in Mappers, we observe that increasing Reducers hardly has performance improvement in CMR. Conversely, we see that increasing Reducers can slightly benefit the NCMR performance by alleviating Mapper workloads. In the right part of Fig. 17, we have the same observations on CMR and NCMR. Even though intermediate data aggregation may have a negative impact on performances in the experiment, we still observe that ACMR can adjust itself to the application and get the performance close to the better system.

6 Related works

Currently, several proposals are related to the use of a combiner in MapReduce. However, they need a programmer to manually enable or disable a combiner. They are not like our ACMR capable of automatically, smartly, and transparently determining to use a combiner for improving performances at run time. Accordingly, they have a risk of performance degradation once a combiner is not appropriately enabled or disabled for an application.

Google MapReduce [1] and Hadoop MapReduce [13,14] both provide a programmer with a combiner inside the systems and leave the use of combiner to a programmer. The systems implement a combiner with code of a Reducer and append the combiner to a Mapper for processing intermediate data generated by the Mapper. Because leaving the use of combiner to a programmer, the systems have a risk of performance degradation resulting from the inappropriate use of a combiner. If a programmer always enables a combiner, the systems may have a performance bottleneck in a computer running a combiner because the combiner competes with a Mapper for computation power. If a programmer always disables a combiner, the systems may have a performance bottleneck in a computer running a Reducer because the Reducer is overloaded by intermediate data from Mappers.

Dryad [29] can be revised to provide the MapReduce functions and become a MapReduce system. Dryad combines concepts of Google MapReduce and Relation Algebra [30] to create a new architecture for distributed and parallelized processing. Dryad can update the dataflow processing graph at run time by connecting computers to each other through TCP pipes. Dryad may use combiners to handle intermediate data outputted by Mappers for constructing an aggregation tree capable of reducing intermediate data and alleviating Reducer workloads. However, Dryad still relies on a programmer to manually deploy a combiner in the dataflow processing graph and can not guarantee that the deployment of the combiner improves performances.

Twister [31] is a hierarchical MapReduce system mainly designed for applications that need the iterative processing of data. Because the iterative processing of data goes through several iterations of Map and Reduce phases, Twister utilizes memory cache in computers to avoid high overheads of file I/O. Besides, Twister uses the architecture of publishing and subscribing messages to communicate and transfer data between computers. Differing from other systems, Twister forwards intermediate data from a Mapper directly to a Reducer and uses a combiner to handle outputs from a Reducer before delivering them to a Mapper at the next phase. Twister uses a combiner mainly for aggregating Reducer outputs on behalf of a Mapper at the next phase. Although using a combiner is necessary, Twister allows a programmer to disable it in the initial time. By leaving the choice of using a combiner to a programmer, Twister suffers a risk of performance degradation like other systems.

MapReduce Online [32] revises the way of intermediate data delivery between a Mapper and a Reducer in Hadoop. Like our ACMR, MapReduce Online uses either a Reducer to process intermediate data directly from a Mapper without a combiner or a combiner in a Mapper to aggregate intermediate data. MapReduce Online monitors the memory buffer used by a Mapper to hold intermediate data until a threshold is reached, and then forwards intermediate data to a Reducer. If network I/O is blocked, MapReduce Online uses a combiner to aggregate intermediate data from a Mapper locally. MapReduce Online is not like our ACMR capable of automatically predicting performances and dynamically determining the way to process intermediate data, so it may suffer a risk of performance degradation. With the design of using a combiner once network I/O is blocked, for example, MapReduce Online may further overload a busy Mapper to degrade performances by running a CPU-bound combiner.

Kambatla et al. [33] propose a concept of local MapReduce, an extra iterative MapReduce procedure that works like a combiner to process intermediate data in a Mapper before delivering the intermediate data to a Reducer. They revise the traditional MapReduce programming model to avoid a global synchronization point between the Map phase and the Reduce phase. They design the local MapReduce to output intermediate data associated with the same key destined to a certain Reducer. However, they merely show the benefits of intermediate data aggregation in experiments. They do not test their system with applications that may have the performance penalty due to intermediate data aggregation.

Distributed Aggregation [34] proposes six partial aggregation ways for Dryad [29]. Distributed Aggregation provides an application with six aggregation trees and evaluates their performances. According to simulation results, Distributed Aggregation shows that different aggregation trees can benefit performances of different applications. Unlike our ACMR, Distributed Aggregation can not help a programmer to automatically choose an aggregation tree for an application. Accordingly, Distributed Aggregation has a risk of per-

Author's personal copy

Table 2 Comparisons of related works

Study name	Using combiner in mapper	Using combiner in reducer	Using combiner in mapper or reducer	Combiner implementation	MapReduce application type
ACMR			V	S	General
Google MapReduce [1]	V			M	General
Hadoop MapReduce [13] [14]	V			M	General
Dryad [29]	V			M	General
Twister [31]		V		C, M	Iterative
MapReduce Online [32]	V			M	General
Local MapReduce [33]	V			M	Iterative
Distributed Aggregation [34]	V			M	General
Map-Join-Reduce [35]		V		C, M	General

C: Manually Enabled or Disabled by User at Initial Time M: Manually Decided by User at Application Development Time S: Automatically Decided by System at Run Time

formance degradation due to leaving the choice of an aggregation tree to a programmer.

Map-Join-Reduce [35] is proposed to handle heterogeneous data sets by a filtering-join-aggregation programming model. Map-Join-Reduce needs a programmer to develop a Join function named Joiner to co-exist with a Reducer for collecting intermediate data associated with the same key. Map-Join-Reduce allows a programmer to enable a combiner in a Mapper for conserving network bandwidth consumption, and then dispatches intermediate data to a Joiner before delivering it to a Reducer locally. Although introducing a Joiner to a system can speed up the processing of intermediate data in a Reducer, Map-Join-Reduce still leaves the choice of using a combiner to a programmer and can not avoid a risk of performance degradation.

Finally, we use Table 2 to briefly compare methods of processing intermediate data in the related works as a summary. Among the studies, we can observe that ACMR has the unique feature of automatically determining to use a combiner in a Mapper or a Reducer according to the behavior of an application at run time.

7 Conclusions

In this paper, Adaptive Combiner for MapReduce (ACMR) is proposed to facilitate the use of a combiner in MapReduce, because using a combiner has either advantages of bandwidth conservation and short delay in networks or disadvantages of Mapper overload and design difficulty. ACMR can automatically determine the situation of using a combiner according to the behavior of an application at run time. ACMR can use the Single Exponential Smoothing (SES) algorithm to smartly predict Mapper workloads and the intermediate data size. According to workloads and computation power of Mappers, ACMR can adjust the use of combiners in Mappers without overloading Mappers and idling Reducers. ACMR can use combiners for various applications on demand and get the better performance. ACMR can transparently work in a MapReduce system to serve various applications without any interference of programmers. For a proof of concept, ACMR currently is implemented in a MapReduce system written in PHP (short for Hypertext Preprocessor), a widely-used general-purpose scripting language.

ACMR has tests in performance impacts of α chunk size, and change interval with Word Count, Bubble Sort, and K-Means. In the three applications, ACMR does not show much performance difference with a different α, but α is required by the SES algorithm. In Word Count and Bubble Sort, ACMR has a slightly worse performance with the increase of chunk size and change interval, because one of the first two data blocks is processed by a Mapper with a wrong way (i.e. using no combiner in Word Count and using a combiner in Bubble Sort) in the initial time to degrade the overall performance. In K-Means, ACMR is hardly affected by the increase of chunk size and change interval because the K-Means performance has more relation with the first point randomly chosen as the group center in the application than the way to process intermediate data.

Besides, we test the ACMR performance with seven applications and take systems of always using a combiner and using no combiner for comparisons. In Word Count, Grep, Radix Sort, Bubble Sort, and Inverted Index, we observe that a system with a combiner can greatly outperform a system without a combiner, because a combiner can alleviate Reducer workloads. In K-Means and Page View Count, we notice that a system with a combiner hardly has a better performance than a system without a combiner, because a combiner may further increase Mapper workloads to make a Mapper the performance bottleneck. We observe that increas-

Author's personal copy

ing Mappers in a system basically can benefit an application, but may not have more performance improvement in Radix Sort and Page View Count when a certain Mapper number is reached due to the overload of a Reducer. Moreover, we notice that increasing Reducers in a system can greatly benefit an application such as Radix Sort and Bubble Sort only if the application can quickly distribute intermediate data over Reducers to alleviate Mapper workloads. When a Mapper is busy in computation without generating much intermediate data, we observe that increasing Reducers in a system hardly has a help to performances because of idling the Reducers. When a Mapper is overloaded in an application such as Radix Sort, Bubble Sort, and Page View Count, furthermore, we observe that using no combiner can get a better performance than using a comber in a system.

Although a different application may need a different way to process its intermediate data for getting a better performance, i.e. aggregating intermediate data or not, we observe that ACMR always can get the performance comparable to the system that is optimal for an application. Since a system hardly satisfies all applications with or without a combiner, we show that ACMR is the outstanding solution capable of automatically, smartly, and transparently determining the suitable way to process intermediate data for getting a better performance on behalf of programmers.

Acknowledgments In this paper, we introduce the achievement of Project NSC 100-2628-E-262-001-MY2 supported by National Science Council at Taiwan. We appreciate the cooperation of colleagues at Tamkang University. We thank Lunghwa University of Science and Technology for greatly approving of our cloud computing research and kindly offering us experiment devices. Besides, we thank the editor and reviewers for their valuable comments on this paper.

References

1. Dean, J. and Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of the 6th symposium on operating systems design and implementation (OSDI), pp. 137–150, Dec 2004
2. Dede, E., Govindaraju, M., and Ramakrishnan, L.: Benchmarking MapReduce implementations for application usage scenarios. In: Proceedings of the IEEE/ACM international conference on grid computing (GRID), pp. 90–97, Sept 2011
3. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. J. ACM SIGMOD **40**(4), 11–20 (Dec. 2011)
4. Mazur, E., Li, B., Diao, Y., and Shenoy, P.: Towards scalable one-pass analytics using MapReduce. In: Proceedings of IEEE international symposium on parallel and distributed processing workshops and Phd Forum (IPDPSW), pp. 1102–1111, May 2011
5. Li, K., Yang, L.T., Lin, X.: Advanced topics in cloud computing. J. Netw. Comput. Appl. **34**(4), 1033–1034 (2011)
6. Zhou, M., Mu, Y., Susilo, W., Yan, J., Dong, L.: Privacy enhanced data outsourcing in the cloud. J. Netw. Comput. Appl. **35**(4), 1367–1373 (2012)
7. Wu, T.L., Qiu, J., and Fox, G.: MapReduce in the clouds for science. In: Proceedings of IEEE second international conference on cloud computing technology and science (CloudCom), pp. 565–572, Dec 2010
8. Prodan, R., Sperk, M., Ostermann, S.: Evaluating high-performance computing on google app engine. IEEE Softw. **29**(2), 52–58 (2012)
9. Huang, T.C.: Program ultra-dispatcher for launching applications in a customization manner on cloud computing. J. Netw. Comput. Appl. (JNCA) **35**(1), 423–446 (2012)
10. PHP: Hypertext Preprocessor, http://www.php.net/
11. Suzumura, T., Trent, S., Tatsubori, M., Tozawa, A. and Onodera, T.: Performance comparison of web service engines in PHP, Java and C. In: Proceedings of IEEE international conference on web services (ICWS), pp. 385–392, Sept 2008
12. Yu, X. and Yi, C.: Design and implementation of the website based on PHP & MYSQL. In: Proceedings of international conference on E-product E-service and E-entertainment (ICEEE), pp. 1–4, Nov 2010
13. White, T.: Hadoop: the definitive guide. ISBN: 978-0-596-52497-4, O'Reilly Media, Yahoo! Press, June 5, 2009
14. Duan, A.: Research and application of distributed parallel search hadoop algorithm. In: Proceedings of international conference on systems and informatics (ICSAI), pp. 2462–2465, May 2012
15. Shvachko, K., Kuang, H., Radia, S. and Chansler, R.: The Hadoop distributed File system. In: Proceedings of 2010 IEEE 26th symposium on mass storage systems and technologies (MSST), pp. 1–10, May 2010
16. Taylor, R.C.: An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. J. BMC Bioinforma. **11**(12), S1 (2010)
17. Wright, G.R. and Stevens, W.R.: TCP/IP illustrated: the protocols. ISBN: 0-201-63346-9, Vol. 2: The Implementation. Addison-Wesley, 1995
18. Yang, Y.R and Lam, S.S.: General AIMD congestion control. In: Proceedings of ICNP, pp. 187–198, Nov 2000
19. Everette, S., Gardner, J.: Exponential smoothing: the state of the art. J. Forecast. **4**(1), 1–28 (1985)
20. Gosling, J., Joy, B., and Steele, G.L.: The Java Language Specification, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA (1996). ISBN:0201634511
21. The Apache Software Foundation, http://www.apache.org/
22. Lee, W.M.: Recommending proper API code examples for documentation purpose. In: Proceedings of 18th Asia Pacific software engineering conference (APSEC), pp. 331–338, 2011
23. Yang, G.: The application of MapReduce in the cloud computing. In: Proceeding of 2th international symposium on intelligence information processing and trusted computing (IPTC), pp. 154–156, Oct 2011
24. Astrachan, O.: Bubble sort: an archaeological algorithmic analysis. In: Proceedings of the 34th SIGCSE technical symposium on computer science education, pp. 1–5, 2003
25. Inaba, M., Katoh, N., and Imai, H.: Applications of weighted voronoi diagrams and randomization to variance-based k-clustering. In: Proceedings of 10th annual ACM symposium computational geometry, pp. 332–339, June 1994
26. Bull, R.I., Trevors, A., Malton, A.J., Godfrey, M.W.: Semantic grep: regular expressions + relational abstraction. In: Proceedings of ninth working conference on reverse, engineering (WCRE'02), pp. 267–276, Oct 2002
27. Zhu, S., Zhiwei, X., Haibo, C., Rong, C., Weihua, Z., and Binyu, Z.: Evaluating SPLASH-2 applications using MapReduce. In: Proceedings of APPT'09, pp. 452–464, 2009
28. He, B., Fang, W., Luo, Q., Govindaraju, N.K., and Wang, T.: Mars: a MapReduce framework on graphics processors. In: Proceedings

Author's personal copy

of the 17th international conference on parallel architectures and compilation, techniques, pp. 260–269, 2008
29. Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of European conference on computer systems (EuroSys), pp. 59–72, 2007
30. Smith, J.M., Chang, P.Y.T.: Optimizing the performance of a relational algebra database interface. J. ACM **18**(10), 568–579 (Oct. 1975)
31. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.H., Qiu, J., and Fox, G.: Twister: a runtime for iterative MapReduce. In: Proceedings of the first international workshop on MapReduce and its applications(HPDC'10), pp. 810–818, 2010
32. Condie, T., Conway, N., Alvaro, P., and Hellerstien, J.M.: MapReduce online. In: Proceedings of 7th USENIX conference on networked systems design and implementation (NSDI), pp. 12–21, 2010
33. Kambatla, K., Rapolu, N., Jagannathan, S., and Grama, A.: Asynchronous algorithms in mapreduce. In: Proceedings of IEEE CLUSTER, pp. 245–254, 2010
34. Yu, Y., Gunda, P.K., and Isard, M.: Distributed aggregation for data-parallel computing: interfaces and implementations. In: Proceedings of ACM symposium on operating systems principles (SOSP), pp. 247–260, 2009
35. Jiang, D., Tung, A.K.H., Chen, G.: Map-join-reduce: towards scalable and efficient data analysis on large clusters. J. IEEE Trans. Knowl. Data Eng. **23**(9), 1299–1311 (2011)

Tzu-Chi Huang received his B.S., M.S., and Ph.D. degrees in Electrical Engineering from National Cheng Kung University in 1997, 1999, and 2008 respectively. He was a system program engineer responsible for the development of network device drivers and related protocols in Silicon Integrated Systems (SiS) Corp. Now, he is an assistant professor in the Department of Electronic Engineering at Lunghwa University of Science and Technology. His research interests include cloud computing, mobile computing, and network protocols. He has much experience in system design and protocol engineering.

Kuo-Chih Chu received the B.S. degree in Information Engineering and Computer Science from Feng-Chia University, Taichung, Taiwan, R.O.C., in 1996 and received M.S. and Ph.D. degrees in Electrical Engineering from National Cheng Kung University, Taiwan, R.O.C., in 1998 and 2005, respectively. He is currently an associate professor with the Department of Electronic Engineering, Lunghwa University of Science and Technology, Taoyuan, Taiwan.

Wei-Tsong Lee received B.S., M.S., and Ph.D. degrees in Electrical Engineering from National Cheng Kung University, Tainan, Taiwan. In 2003, he joined the department members of Electrical Engineering of Tamkang University as associate professor, and reached professor in 2007. From 2010, he is the chairman of Electrical Engineering Department. His research interests are computer architecture, micro-processor interface and computer network.

Yu-Sheng Ho received his M.S. degree from Lunghwa University of Science and Technology in 2011. He is currently a Ph.D. candidate in the Institute of Department of Electrical Engineering at Tamkang University. His research interests include cloud computing, and parallel and distributed file systems.