2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)

SAMR: A Self-adaptive MapReduce Scheduling Algorithm In Heterogeneous Environment

Quan Chen Daqiang Zhang Minyi Guo Qianni Deng
Department of Computer Science
Shanghai Jiao Tong University, Shanghai, China
{*chen-quan, zhangdq*}*@sjtu.edu.cn* ,
{*guo-my, deng-qn*}*@cs.sjtu.edu.cn*

Song Guo
School of Computer Science and Engineering,
The University of Aizu, Japan
sguo@u-aizu.ac.jp

***Abstract*—Hadoop is seriously limited by its MapReduce scheduler which does not scale well in heterogeneous environment. Heterogenous environment is characterized by various devices which vary greatly with respect to the capacities of computation and communication, architectures, memorizes and power. As an important extension of Hadoop, LATE MapReduce scheduling algorithm takes heterogeneous environment into consideration. However, it falls short of solving the crucial problem – poor performance due to the static manner in which it computes progress of tasks. Consequently, neither Hadoop nor LATE schedulers are desirable in heterogeneous environment. To this end, we propose *SAMR: a Self-Adaptive MapReduce scheduling algorithm*, which calculates progress of tasks dynamically and adapts to the continuously varying environment automatically.**

When a job is committed, *SAMR* splits the job into lots of fine-grained map and reduce tasks, then assigns them to a series of nodes. Meanwhile, it reads historical information which stored on every node and updated after every execution. Then, *SAMR* adjusts time weight of each stage of map and reduce tasks according to the historical information respectively. Thus, it gets the progress of each task accurately and finds which tasks need backup tasks. What's more, it identifies slow nodes and classifies them into the sets of slow nodes dynamically. According to the information of these slow nodes, *SAMR* will not launch backup tasks on them, ensuring the backup tasks will not be slow tasks any more. It gets the final results of the fine-grained tasks when either slow tasks or backup tasks finish first. The proposed algorithm is evaluated by extensive experiments over various heterogeneous environment. Experimental results show that *SAMR* significantly decreases the time of execution up to 25% compared with Hadoop's scheduler and up to 14% compared with LATE scheduler.

***Keywords*-MapReduce, Scheduling algorithm, Heterogeneous environment, Self-adaptive**

I. Introduction

MapReduce is a programming model and an associated implementation for processing and generating large data sets [1]. It enables users to specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all the intermediate values associated with the same intermediate key [1]. MapReduce is used in Cloud Computing in the beginning [2], [3], [4], [5], [6]. It is initiated by Google, together with GFS [7] and BigTable [8] comprising backbone of Google's Cloud Computing platform. MapReduce has achieved increasing success in various applications ranging from horizontal and vertical search engines to GPU to multiprocessors, e.g, [9], [10], [11], [12], [13], [14], [15], [16], [17].

The open-source project Hadoop is the most well-celebrated MapReduce framework, which processes vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable and fault-tolerant manner [18], [19]. Its MapReduce scheduler just considers scheduling in homogeneous environment, and fails to find tasks which prolonging execution time. Therefore it is inefficient, especialy in heterogeneous environment.

As an important extension of Hadoop, LATE scheduling algorithm tries to improve Hadoop by attempting to find real slow tasks by computing remaining time of all the tasks. It selects a set of tasks which have longer remaining time than those of all other nodes, and considers the set of tasks are slow tasks. However it does not compute the remaining time for tasks correctly, and can not find real slow tasks in the end.

To this end, we share the similar idea to LATE scheduling algorithm in this paper and propose *SAMR: a Self-Adaptive MapReduce scheduling algorithm*. *SAMR* significantly improves MapReduce in terms of saving time of execution as well as system resources. *SAMR* is inspired by facts that slow tasks prolong the execution time of the whole job and nodes requires various time in accomplishing the same tasks due to their differences, such as capacities of computation and communication, architectures, memorizes and power. Although Hadoop and LATE also launch backup tasks for slow tasks, they cannot find the appropriate tasks which are really prolong the execute time of the whole job because the two scheduler always use a static way to find slow tasks. On the contrary, *SAMR* incorporates historical information recorded on each node to tune parameters and find slow tasks dynamically. *SAMR* can find slow tasks which need

978-0-7695-4108-2/10 $26.00 © 2010 IEEE
DOI 10.1109/CIT.2010.458

backup task really. In order to save system resources, *SAMR* classifies slow nodes into map slow nodes and reduce slow nodes further. *SAMR* defines fast nodes and slow nodes to be nodes which can finish a task in a shorter time and longer time than most other nodes. Map/reduce slow nodes means nodes which execute map/reduce tasks using a longer time than most other nodes. In this way, *SAMR* launches backup map tasks on nodes which are fast nodes or reduce slow nodes.

The most important contributions of this paper are threefold:

- *SAMR* uses historical information recorded on every node to tune weight of each stage dynamically.
- *SAMR* takes the two stages characteristic of map tasks into consideration for the first time.
- *SAMR* classifies slow nodes into map slow nodes and reduce slow nodes further.

The rest of this paper is organized as follows. Section II introduces basic conceptions and analyzes the drawbacks of existing MapReduce scheduling algorithms. Section III introduces the *SAMR* and reports the implementation details. Section IV describes the experimental results. Section V draws the conclusion with pointing out our future work.

II. Related Work

In order to understand the programming model that is the basis of *SAMR*, this section provides a brief view of MapReduce. It first introduces the preliminary knowledge about MapReduce and then overviews the related work.

A. Basic conceptions in MapReduce

MapReduce is a programming model enabling a great many of nodes to handle huge data by cooperation. In traditional MapReduce scheduling algorithm, a MapReduce application needs to be run on the MapReduce system is called a *"job"*. A *job* can be divided into a series of *"Map tasks"(MT)* and *"Reduce tasks"(RT)*. The tasks which execute map function are called *"Map tasks"*, and which execute reduce function are called *"Reduce tasks"*.

In a cluster which runs MapReduce, nodes were classified into *"NameNode"* and *"DataNode"* from data storage aspect. There is only one *NameNode*, which records all the information of where data is stored. there are lots of *DataNodes* which store data. There are only one *"JobTracker"(JT)* and a series of *"TaskTracker"(TT)*. *JobTracker* is a process which manages jobs. *TaskTracker* is a process which manages tasks on the corresponding nodes. Table I lists all the conceptions and notions used in this paper.

MapReduce scheduling system involves six steps when executing a MapReduce job, illustrated in Figure 1 [1].

First, user program forks the MapReduce job. Second, master distributes *MT* and *RT* to different workers. Third, *MT* reads in the data splits, and runs map function on the data which is read in. Fourth, these *MT* write intermediate key/value pairs into local disks. Fifth, *RT* read the intermediate results remotely, and run reduce function on the intermediate results which is read in. At last, these *RT* write the final results into the output files.

Table I
Conceptions and notions in this paper

Name	Description
NameNode	Records where data is stored
DataNode	Stores data
JobTracker(JT)	Manages MapReduce jobs
TaskTracker(TT)	Manages tasks
Job	MapReduce application
Map tasks(MT)	Tasks which run map function
Reduce tasks(RT)	Tasks which run reduce function
ProgressScore(PS)	Process score of a task
ProgressRate(PR)	Progress rate of a task
TimeToEnd(TTE)	Remaining time of a task
TrackerRate(TrR)	Progress rate of a TaskTracker
HISTORY_PRO(HP)	Weight of historical information
SLOW_TASK_CAP(STaC)	Parameter used to distinguish slow tasks
SLOW_TRACKER_CAP(STrC)	Parameter used to distinguish slow TTs
SLOW_TRACKER_PRO(STrP)	Maximum proportion of slow TTs
BACKUP_PRO(BP)	Maximum proportion of backup tasks
M1	Weight of first stage in MTs
M2	Weight of second stage in MTs
R1	Weight of coping data in RTs
R2	Weight of sorting in reduce tasks
R3	Weight of merging in reduce tasks

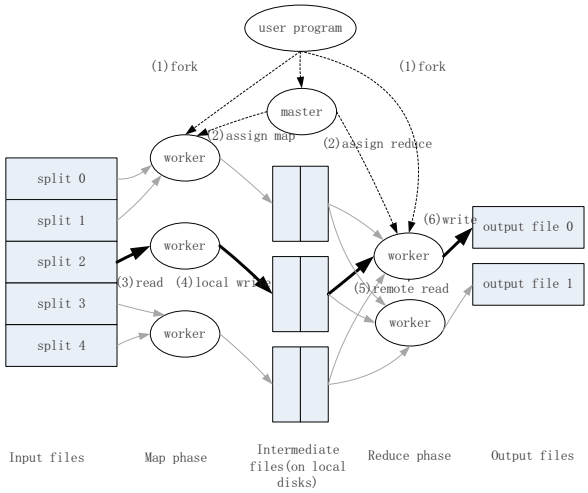

Figure 1. Overview of a MapReduce job

B. MapReduce scheduling algorithm in Hadoop

The most famous implementation of MapReduce, Hadoop, suffers from a problem that it cannot distinguish tasks which need backup tasks on fast nodes correctly.

Hadoop monitors the progress of tasks using *"Progress Score(PS)"* (range from 0 to 1). The average progress score is PS_{avg}. The *Progress Score* of the i^{th} task is *PS[i]*. Suppose the number of tasks which are being executed is *T*, the number of key/value pairs need to be processed in a task

is *N*, the number of key/value pairs have been processed in the task is *M*, and the task has finished *K* stages (only for reduce task. There are three stages in a reduce task: *copy data phase, sort phase* and *reduce phase*). Hadoop gets *PS* according to the Eqs. 1 and 2 and launches backup tasks according to the Eq. 3.

$$PS = \begin{cases} M/N & \text{For } \textit{MT}, \\ 1/3 * (K + M/N) & \text{For } \textit{RT}. \end{cases} \quad (1)$$

$$PS_{avg} = \sum_{i=1}^{T} PS[i]/T \quad (2)$$

$$\text{For task } T_i\text{: } PS[i] < PS_{avg} - 20\% \quad (3)$$

If the Eq. 3 is fulfilled, T_i needs a backup task. The main shortcomings of this method include:

(1) In Hadoop, the values of *R1, R2*, *R3*, *M1, M2* are 0.33, 0.33. 0.34, 1 and 0 respectively. However *R1, R2, R3, M1* and *M2* are dynamic when tasks running on different nodes, especially in heterogeneous environment.

(2) Hadoop always launches backup tasks for those tasks of which *PS*s are less than $PS_{avg} - 20\%$. In this case, Hadoop may launch backup tasks for wrong tasks. For example, task *Ti*'s *PS* is 0.7 and needs 100 seconds to finish its work, while another task *Tj*'s *PS* is 0.5, but only needs 30 seconds to finish its work. Suppose the average progress score PS_{avg} is 0.8, the method will launch a backup task for *Tj* according to the Eq. 3. If we launch a backup task for *Ti* instead of *Tj*, it will save more time. What's more, tasks that *PS* is larger than 0.8 will have no chance to have a backup task, even though they are very slow and need a very long time to finish. This is because PS_{avg} will never be larger than 1.

(3) Hadoop may launch backup tasks for fast tasks. For example, there are 3 *RT Ri, Rj, Rk*, and their *PSs* are 0.33, 0.66 and 0.66. In this case, PS_{avg} = $(0.33+0.66+0.66)/3 = 0.55$. According to the Eq. 3, we should launch a backup task for *Ri*. However, the second stage, *sort stage*, only needs a very short time in a real system. It is unnecessary to launch a backup task for *Ri*.

C. LATE MapReduce scheduling algorithm

LATE MapReduce scheduling algorithm always launches backup tasks for those tasks which have more remaining time than other tasks. Suppose a task *T* has run *Tr* seconds. Let *PR* denotes the progress rate of *T*, and *TTE* denotes how long time remaining until T was finished. LATE MapReduce scheduling algorithm computes *PR* and *TTE* according to the Eqs. 4 and 5. *PS* in the Eq. 4 is computed according to the Eq. 1.

$$PR = PS/Tr \quad (4)$$

$$TTE = (1.0 - PS)/PR \quad (5)$$

Although LATE uses an efficient strategy to launch backup tasks, it often launches backup tasks for inappropriate tasks. This is because LATE cannot find *TTE* for all the running tasks correctly.

One shortcoming of LATE, which is same to Hadoop, is the values of *R1, R2*, *R3*, *M1, M2* are 0.33, 0.33. 0.34, 1 and 0 respectively. This setting may lead to the wrong *TTE*. Suppose *R1, R2 and R3* are 0.6, 0.2 and 0.2 respectively in a real system. When the first stage finishes in *Tr* seconds, the reduce task still needs $(1-0.6)*(Tr/0.6) = 0.67Tr$ seconds to finish the whole task. However, the *TTE* computed in LATE scheduling algorithm is $(1-0.33)*(Tr/0.33) = 2*Tr$ seconds.

Another shortcoming of LATE MapReduce scheduling algorithm is it does not distinguish map slow nodes and reduce slow nodes. One node may executes *MT* quickly, but executes *RT* slower than most of other nodes. LATE just considers one node either fast node or slow node, does not classify slow nodes further.

III. *SAMR*: Self-adaptive MapReduce Scheduling Algorithm

SAMR is developed with a similar idea to LATE MapReduce scheduling algorithm. However, *SAMR* gets more accurate *PSs* of all the tasks by using historical information. By using accurate *PSs*, *SAMR* finds real slow tasks and decreases more execute time comapared with Hadoop and LATE. Algorithm 1 illustrates the process of *SAMR*. Subsection III-A describes how to reading historical information and tune parameters using it. Subsection III-B describes how to find slow tasks. Subsection III-C describes how to find slow *TTs*. Subsection III-D describes when *SAMR* launches backup tasks. Subsection III-E describes the implementation details of *SAMR*.

Algorithm 1 SAMR algorithm

```
1: procedure SAMR
2:     input: Key/Value pairs
3:     output: Statistical results

4:     Reading historical information and tuning parameters using it
5:     Finding slow tasks
6:     Finding slow TaskTrackers
7:     Launching backup tasks
8:     Collecting results and updating historical information
9: end procedure
```

A. Reading historical information and tuning parameters using historical information

TTs read historical information(*R1, R2, R3, M1, M2*) that recorded on the disks from nodes where the *TTs* are running. The historical information is stored on every node in xml format. The values of *R1, R2*, *R3*, *M1, M2* are 0.33, 0.33. 0.34, 1 and 0 by default. Algorithm 1 updates the values after every execution(line 8 in Algorithm 1). In order to get

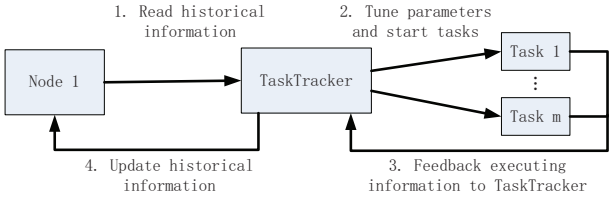

Figure 2. The way to use and update historical information

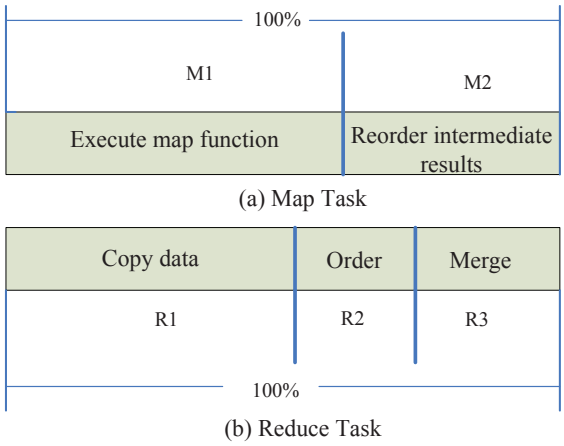

Figure 3. The two stages of *MT* and three stages of *RT*

the correct remaining time of tasks, *SAMR* computes *PS* by tuning parameters which used to compute *PS*.

SAMR uses a parameter, *HISTORY_PRO(HP)*(range from 0 to 1) to tune weight of historical information in setting parameters. If *HP* is too large (close to 1), parameters in the current tasks are depend on historical information seriously. Thus specific situation of current job is overlooked by *SAMR*. Meanwhile, if *HP* is too small (close to 0), the parameters of current task are depend on the finished tasks which in the same job seriously. *SAMR* uses historical information following 4 steps, illustrated in Figure 2.

First, *TTs* read historical information from the nodes where they are running on. The historical information includes historical values of *M1, M2, R1, R2* and *R3*. Then, *TTs* tune *M1, M2, R1, R2* and *R3* according to historical information, *HP*, and information collected from the current running system. Consequently, *TTs* collect values of *M1, M2, R1, R2* and *R3* according to the real running information after the tasks finished. Finally, *TTs* write these historical information back to the xml stored on the nodes (line 8 in algorithm 1).

In addition, every *TT* read historical information from node which it is running on. There is not any additional communication is needed when read and update historical information, So *SAMR* is scalable.

Figure 3 illustrates the divisions and weights of stages in *MT* and *RT*. In Hadoop and LATE, *M1* of *MT* is 1.0, *M2* of *MT* is 0, *R1*, *R2* and *R3* of *RT* are all 1/3.

B. Finding slow tasks

SLOW_TASK_CAP(STaC)(range from 0 to 1) is used to classify tasks into fast and slow tasks. If the progress rate of task *Ti*, PR_i and the average progress rate of all the running tasks, *APR* fulfill the Eq. 6, *Ti* is judged to be slow task. Suppose the number of task running now is *N*, *APR* is computed according to the Eq. 7.

$$PR_i < (1.0 - STaC) * APR \tag{6}$$

$$APR = \sum_{j=1}^{N} PR_j / N \tag{7}$$

According to the Eq. 6, if *STaC* is too small (close to 0), *SAMR* will classify some fast tasks into slow tasks. If *STaC* is too large (close to 1), *SAMR* will classify slow tasks into fast tasks.

C. Finding slow TaskTrackers

SLOW_TRACKER_CAP(STrC)(range from 0 to 1) is used to classify *TaskTrackers* into fast *TTs* and slow *TTs*. One *TT* only runs on one node, so slow nodes are the same as slow *TTs*.

Suppose there are *N TTs* in the system. The rate of the i^{th} *TaskTracker, TTi,* for *MT* is TrR_{mi}, for *RT* is TrR_{ri}, the average rate of all the *TTs* for *MT* is $ATrR_m$, for *RT* is $ATrR_r$. If there are *M MT* and *R RT* run on *TTi*, TrR_{mi}, TrR_{ri}, $ATrR_m$ and $ATrR_r$ can be computed according to the Eqs. 8, 9, 10 and 11.

$$TrR_{mi} = \sum_{i=1}^{M} PR_i / M \tag{8}$$

$$TrR_{ri} = \sum_{i=1}^{R} PR_i / R \tag{9}$$

$$ATrR_m = \sum_{i=1}^{N} TrR_{mi} / N \tag{10}$$

$$ATrR_r = \sum_{i=1}^{N} TrR_{ri} / N \tag{11}$$

For *TTi*, if it fulfills the Eq. 12, it is a slow map *TT*. If it fulfills the Eq. 13, it is a slow reduce *TT*.

$$TrR_{mi} < (1 - STrC) * ATrR_m \tag{12}$$

$$TrR_{ri} < (1 - STrC) * ATrR_r \tag{13}$$

According to the Eqs. 11 and 12, if *STrC* is too small (close to 0), *SAMR* will classify some fast *TTs* into slow *TTs*. If *STrC* is too large (close to 1), *SAMR* will classify some slow *TTs* into fast *TTs*.

SLOW_TRACKER_PRO(STrP)(range from 0 to 1) is used to define the maximum proportion of slow *TTs* in all the

TTs. Suppose the number of slow *TTs* is *SlowTrackerNum*, the number of all the *TTs* is *TrackerNum*. The Eq. 15 must be fulfilled in the system.

$$SlowTrackerNum < STrP * TrackerNum \quad (14)$$

If the Eqs. 12 and 14 are fulfilled at the same time, *SAMR* views the *TT* as a map slow *TT*.

D. Launching backup tasks

If there are slow tasks, and Eq. 15 is fulfilled, a backup task can be launched when some of *TTs* are free. *BACKUP_PRO(BP)*(range from 0 to 1) is used to define the maximum proportion of backup tasks in all the tasks. Suppose the number of backup tasks is *BackupNum*, the number of all the running tasks is *TaskNum*. The Eq. 15 must be fulfilled in the system.

$$BackupNum < BP * TaskNum \quad (15)$$

E. Implementation of SAMR

This subsection introduces implementation details of *SAMR*. *SAMR* supposes the weight of "Execute map function" is M1, and the weight of "Reorder intermediate results" is M2, the weights of "copy data", "sort", and "merge" are R1, R2 and R3 respectively, which are illustrated in Figure 3. Therefore, $M1 + M2 = 1$ and $R1 + R2 + R3 = 1$. *M1, M2, R1, R2* and *R3* are computed according to method illustrated in Figure 2.

Suppose the number of key/value pairs which have been processed in a task is N_f, the number of overall key/value pairs in the task is N_a, the current stage of processing is S (limited to be 0, 1, 2), the progress score in the stage is $SubPS$.

The $SubPS$ in the stage can be computed according to the Eq. 16. *PS* of tasks is computed according to the Eqs. 17 and 18.

$$SubPS = N_f / N_a \quad (16)$$

$$\text{For } MT\text{: } PS = \begin{cases} M1 * SubPS & \text{if } S = 0, \\ M1 + M2 * SubPS & \text{if } S = 1. \end{cases} \quad (17)$$

$$RT\text{: } PS = \begin{cases} R1 * SubPS & \text{if } S = 0, \\ R1 + R2 * SubPS & \text{if } S = 1, \\ R1 + R2 + R3 * SubPS & \text{if } S = 2. \end{cases} \quad (18)$$

SAMR computes *PS* more accurate than Hadoop and LATE, Because *M1, M2, R1, R2* and *R3* are tailored according to historical information. However, in Hadoop and LATE, *M1, M2, R1, R2* and *R3* are 1, 0, 0.33, 0.33, 0.34 respectively, which cannot adaptive to different environment. After getting exact *PS*, *SAMR* computes the remain time of all the running tasks, TTE, according to the Eq. 5. By this way, *SAMR* finds real slow tasks and launches backup tasks for these slow tasks on fast nodes of this kind of tasks consequently.

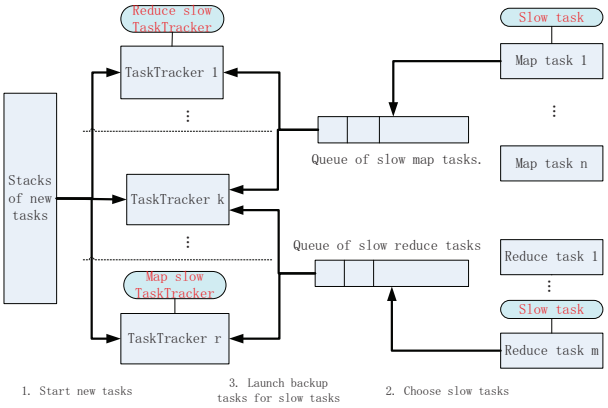

Figure 4. Overview on *SAMR*, *TTs* tries to launch new tasks first. If stack of new tasks is empty, they try to launch backup tasks for tasks in queue of slow tasks

SAMR can launch a backup task on *TaskTracker* TT_j for task T_i only when T_i is a slow task which fulfills the Eq. 6, TT_j is not a slow *TT*, according to the Eqs. 12 and 13, and the number of backup tasks is less than the maximun number of backup tasks, which was got according to the Eq. 15.

SAMR schedules tasks in the following 3 steps, illustrated in Figure 4.

First, all the *TTs* obtain new tasks from stack of new tasks according to data locality property. Then, the *TTs* compute *PR* and *TTE* for all the tasks running on it. Next, the algorithm finds which tasks are slow *MT* or slow *RT*. Consequently, these slow tasks are inserted into correspondent queue of slow tasks (queue of slow *MT* or queue of slow *RT*). Meanwhile, if stack of new tasks is empty, the *TT* tries to find slow task in the queue of slow tasks, and launches backup task. Only when the *TT* is not a map/reduce slow *TT*, it can launch backup tasks for *MT*/*RT*.

IV. Evaluation

In order to verify the effectiveness of *SAMR*, we carry a series of experiments. In particular, we try to answer 3 questions bellow:

- What's the best parameters for *SAMR*?
- Is the historical information recorded correct in *SAMR*?
- What is the performance of *SAMR* in heterogeneous environment?

A. Experimental environment

We establish experimental environment by using virtual machines on five personal computers. All the virtual machines use Ubuntu operating system. The version of JDK is 1.6.0.10, and the version of Hadoop is 0.19.1. The *SAMR* is implemented based on Hadoop 0.19.1. Because we cannot get the primary version of LATE MapReduce scheduling algorithm, so we implement a new one, according

Figure 5. Affection of "HP" on the execute time, "HP" does not affect execute time too much

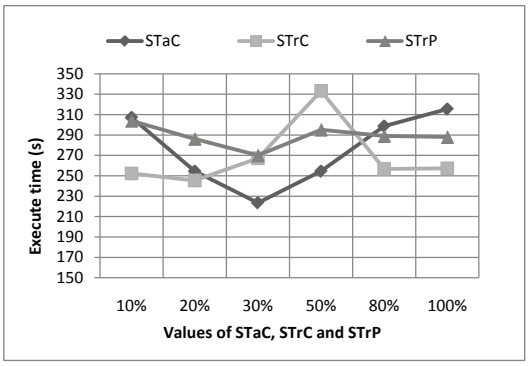

Figure 6. Affection of "STaC", "STrC", and "STrP" on the execute time. The time of execution first decreases then increases with the increase of *STaC*. It is shortest when *STrC* is 0.2, and decreases then increases with the increase of *STrP*.

to the method mentioned in [20]. The detail experimental environment are showed in table II and III.

Table II
DETAIL ENVIRONMENT OF EXPERIMENTS

No. of vm(VM/PC)	No. of PC	No. of nodes	write rate(MB/S)
1	1	1	2.87
2	3	6	1.40
real pc	1	1	3.43

Table III
PROFILES OF EXPERIMENTS

No. of copies	*TT*/node	m/r slots on *TTs*	benchmarks
2	1	2/2	*"Sort"*,*"WordCount"*

The benchmarks used in these experiments are examples in Hadoop: *"Sort"* and *"WordCount"*, because LATE also uses the two programs as benchmarks. The two benchmarks show key characteristic of MapReduce clearly.

B. Best parameters in SAMR

Before evaluating the performance of *SAMR*, we should select a best combination of parameters in *SAMR*. Definitions of these parameters are shown in table I.

In order to select the best parameters in *SAMR*, we changes one parameter while keeping all the other parameters constant. In the experiments, We run *"Sort"* and *"WordCount"* benchmarks ten times each for every setting. We only post the results of *"Sort"*. Results of *"WordCount"* are similar to the results of *"Sort"*.

1) HP: *HP* means weights of historical information in setting weights of each stage in *MT* and *RT*. As illustrated in Figure 5, the time of execution does not change dramatically while *HP* changing. The *HP* tunes historical information close to the real world when *SAMR* run for first several times. But, *HP* is important when different types of jobs run on the same system. Because different jobs usually have different *R1, R2, R3, M1, M2* when running on the same system.

2) STaC: *STaC* is a parameter used to find slow tasks according to the Eq. 6. In Figure 6, the time of execution first decreases then increases with the increase of *STaC*. This is because *SAMR* judges fewer tasks to be slow tasks with the increase of *STaC* according to the Eq. 6. When *STaC* is smaller than 0.3, *SAMR* judges several fast tasks to be slow tasks. Backup tasks of these fast tasks consume a great many of system resources. So the execute time is prolonged. On the other hand, when *STaC* is larger than 0.3, some real slow tasks are judged to be fast tasks. These slow tasks will prolong the execute time. We set *STaC* to be 0.3 in the following experiments.

3) STrC: *STrC* is a parameter used to find slow *TTs* according to the Eqs. 12 and 13.

As shown in Figure 6, the time of execution is shortest when *STrC* is 0.2. This is because *SAMR* judges fewer *TTs* to be slow *TTs* with the increase of *STrC* according to the Eqs. 12 and 13. When *STrC* is small than 0.2, *SAMR* judges several fast *TTs* to be slow ones. In this condition, the system resources can be used are decreased. When *STrC* is larger than 0.2, some slow *TTs* are judged to be fast ones. In this condition, backup tasks may run on these slow *TTs*, so the execute time cannot be shorten. We set *STaC* to be 0.3 in the following experiments.

4) STrP: *STrP* is a parameter used to limit the maximum number of slow *TTs* according to the Eq. 14. As shown in Figure 6, the time of execution first decreases then increases with the increase of *STrP*. This is because the maximum number of slow *TTs* becomes more with the increasing of *STrP*. when *STrP* is smaller than 0.3 and *STrC* is larger than 0.2, some real slow *TTs* are judged to be fast *TTs*. Backup tasks can run on these slow nodes, the system resources competition on slow nodes will prolong the execute time. On the other hand, if *STrP* is larger than 0.3, and *STrC* is smaller than 0.2, *SAMR* judges several fast *TTs* to be slow

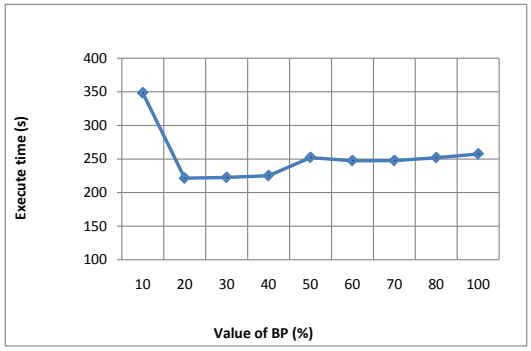

Figure 7. Affection of "BP" on the execute time. The time of execution first decreases then increases with the increase of *BP*

	Historical information/ Real information				
	Map		Reduce		
	FirsSpan	SecondSpan	FirstSpan	SecondSpan	ThirdSpan
Node 1	0.8/0.78	0.2/0.22	0.59/0.62	0.19/0.23	0.22/0.15
Node 2	0.77/0.77	0.23/0.23	0.46/0.42	0.06/0.03	0.48/0.55
Node 3	0.75/0.66	0.25/0.34	0.44/0.4	0.32/0.45	0.24/0.15
Node 4	0.74/0.77	0.26/0.23	0.62/0.64	0.13/0.06	0.25/0.3
Node 5	0.81/0.82	0.19/0.18	0.43/0.44	0.14/0.04	0.43/0.52
Node 6	0.73/0.77	0.27/0.23	0.51/0.53	0.19/0.12	0.3/0.35
Node 7	0.71/0.67	0.29/0.33	0.51/0.5	0.11/0.06	0.38/0.44
Node 8	0.79/0.78	0.21/0.22	0.46/0.41	0.13/0.48	0.41/0.11

Figure 8. Historical information and real information recorded on all the 8 nodes. Historical information is little different from real information

ones. System resources can be used is decreased. We set *STaC* to be 0.3 in the following experiments.

5) BP: *BP* is a parameter used to limit the maximum number of backup tasks according to the Eq. 15. As shown in Figure 7, the time of execution first decreases then increases with the increase of *BP*. This is because the maximum number of backup tasks become larger with the increasing of *BP*. When *BP* is smaller than 0.2, *SAMR* cannot launch backup tasks for all the slow tasks because of the limitation of the number of backup tasks. On the other hand, when *BP* is larger than 0.2, backup tasks will consume a lot of system resources, so the time of execution is prolonged. We set *STaC* to be 0.3 in the following experiments.

After a series of experiments, the best parameters for *SAMR* are: *HP* = 0.2, *STaC* = 0.3, *STrC* = 0.2, *STrP* = 0.3, *BP* = 0.2. These parameters must be re-specified in new environment.

C. Correctness of historical information

In order to verify the correctness of historical information used in *SAMR*, we list historical information and information collected from the real system in Figure 8. For either *MT* or *RT*, the weights of stages recorded in the historical information are not far from the weights collected from the real system. The weights of all the stages are far from the constant weights in Hadoop and LATE.

D. Performance of SAMR

In order to evaluate performance of *SAMR*, We compare performance of five different MapReduce scheduling

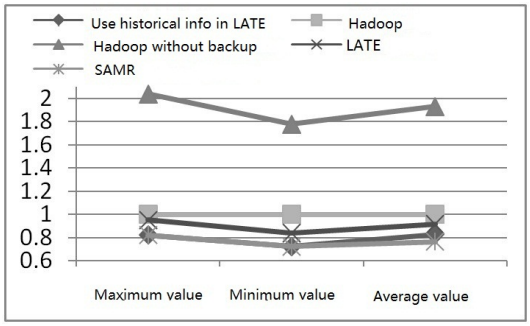

Figure 9. The execute results of "Sort" running on the experiment platform. Backup mechanism and Historical information are all very useful in "Sort". *SAMR* decreases time of execution about 24% compared to Hadoop.

algorithm by running *Sort* and *WordCount* ten times each. The five algorithms are Hadoop without backup mechanism, scheduling algorithm in Hadoop, LATE, LATE using historical information and *SAMR*. Parameters used in *SAMR* are gotten from experiments in subsection IV-B.

Figure 9 shows the efficiency of *SAMR* when running *Sort* benchmark. We uses the execute time of Hadoop as the baseline, and finds that Hadoop without backup mechanism spends about double time in executing the same job. LATE decreases about 7% execute time, LATE using historical information mechanism decreases about 15% execute time, *SAMR* decreases about 24% execute time compared to Hadoop. This is because *RT* spend a long time in *Sort*. Backup tasks on fast nodes for slow *RT* can finish in a shorter time than the primary slow *RT*, and hence saving a lot of time. By finding real slow tasks and launching backup tasks on fast nodes, *SAMR* has archived better performance than all the other MapReduce schedulers.

In Figure 10, Hadoop without backup mechanism spends just a little more time than Hadoop when running *WordCount* benchmark. LATE scheduling algorithm can decrease about 8% , LATE scheduling algorithm with historical information mechanism can decrease about 13% execute time, *SAMR* can decrease about 17% execute time compared to Hadoop. This is because *RT* spends a shorter time in *WordCount* than *Sort* benchmark and backup tasks for slow tasks cannot save too much time.

V. Conclusion

In this paper, we have proposed *SAMR*: a Self-adaptive MapReduce scheduling algorithm, which uses historical information and classifies slow nodes into map slow nodes and reduce slow nodes. Experimental results have shown the effectiveness of the self-adaptive MapReduce scheduling algorithm. The algorithm decreases the execution time of MapReduce jobs, especially in heterogeneous environments. The algorithm selects slow tasks and launch backup tasks accordingly while classifying nodes correctly, and saving a lot of system resources.

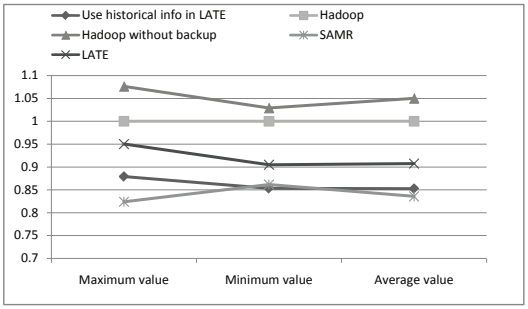

Figure 10. The execute results of "WordCount" running on the experiment platform. Backup tasks and Historical information are not very useful in "WordCount". *SAMR* decreases time of execution about 17% compared to Hadoop

However, the proposed algorithm could be further improved in term of several aspects. First, this algorithm will focus on how to account for data locality when launching backup tasks, because data locality may remarkably accelerate the data load and store. Second, *SAMR* is considering a mechanism to incorporate that tune the parameters should be added. Third, *SAMR* will be evaluated on various platforms by first evaluated on rented Cloud Computing platform.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplied data processing on large clusters," in *OSDI 2004: Proceedings of 6th Symposium on Operating System Design and Implemention*, (New York), pp. 137–150, ACM Press, 2004.

[2] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[3] J. Varia, "Cloud architectures," *White Paper of Amazon, jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf*, 2008.

[4] L. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[5] L. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2008.

[6] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP 2003: Proceedings of the 9th ACM Symposium on Operating Systems Principles*, (New York, NY, USA), pp. 29–43, ACM, 2003.

[8] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, 2006.

[9] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *HPCA 2007: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), pp. 13–24, IEEE Computer Society, 2007.

[10] M. de Kruijf and K. Sankaralingam, "Mapreduce for the cell b.e. architecture," tech. rep., Department of Computer Sciences, University of WisconsinCMadison, 2007.

[11] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT 2008: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, (New York, NY, USA), pp. 260–269, ACM, 2008.

[12] M. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce," *Bioinformatics*, vol. 25, no. 11, p. 1363, 2009.

[13] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng, "Spatial Queries Evaluation with MapReduce," in *Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing-Volume 00*, pp. 287–292, IEEE Computer Society, 2009.

[14] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," tech. rep., Technical Report UCB/EECS-2009-55, University of California at Berkeley, 2009.

[15] C. Tian, H. Zhou, Y. He, and L. Zha, "A dynamic MapReduce scheduler for heterogeneous workloads," in *Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing-Volume 00*, pp. 218–224, IEEE Computer Society, 2009.

[16] P. Elespuru, S. Shakya, and S. Mishra, "MapReduce system over heterogeneous mobile devices," in *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, pp. 168–179, Springer, 2009.

[17] C. Jin and R. Buyya, "MapReduce programming model for .NET-based distributed computing," in *Proceedings of the 15th European Conference on Parallel Processing (Euro-Par 2009)*, Citeseer, 2009.

[18] Yahoo, "Yahoo! hadoop tutorial." http://public.yahoo.com/gogate/hadoop-tutorial/start-tutorial.html.

[19] Hadoop, "Hadoop home page." http://hadoop.apache.org/.

[20] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *8th Usenix Symposium on Operating Systems Design and Implementation*, (New York), pp. 29–42, ACM Press, 2008.