J Supercomput (2013) 64:1038–1054
DOI 10.1007/s11227-011-0682-5

HAT: history-based auto-tuning MapReduce in heterogeneous environments

Quan Chen · Minyi Guo · Qianni Deng · Long Zheng · Song Guo · Yao Shen

Published online: 23 September 2011

Abstract In MapReduce model, a job is divided into a series of *map tasks* and *reduce tasks*. The execution time of the job is prolonged by some slow tasks seriously, especially in heterogeneous environments. To finish the slow tasks as soon as possible, current MapReduce schedulers launch a backup task on other nodes for each of the slow tasks. However, traditional MapReduce schedulers cannot detect slow tasks correctly since they cannot estimate the progress of tasks accurately (Hadoop home page http://hadoop.apache.org/, 2011; Zaharia et al. in 8th USENIX symposium on operating systems design and implementation, ACM, New York, pp. 29–42, 2008). To solve this problem, this paper proposes a History-based Auto-Tuning (HAT) MapReduce scheduler, which calculates the progress of tasks accurately and adapts to the continuously varying environment automatically. HAT tunes the weight of each phase of a map task and a reduce task according to the value of them in history tasks and uses

Q. Chen · M. Guo (✉) · Q. Deng · Y. Shen
Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China
e-mail: guo-my@cs.sjtu.edu.cn

Q. Chen
e-mail: chen-quan@sjtu.edu.cn

Q. Deng
e-mail: deng-qn@cs.sjtu.edu.cn

Y. Shen
e-mail: shen_yao@cs.sjtu.edu.cn

L. Zheng
Huazhong University of Science and Technology, Wuhan, China
e-mail: d8112104@u-aizu.ac.jp

L. Zheng · S. Guo
School of Computer Science and Engineering, The University of Aizu, Aizuwakamatsu, Japan

S. Guo
e-mail: sguo@u-aizu.ac.jp

the accurate weights of the phases to calculate the progress of current tasks. Based on the accurate-calculated progress of tasks, HAT estimates the remaining time of tasks accurately and further launches backup tasks for the tasks that have the longest remaining time. Experimental results show that HAT can significantly improve the performance of MapReduce applications up to 37% compared with Hadoop and up to 16% compared with LATE scheduler.

Keywords History-based auto-tuning · Scheduling algorithm · Heterogeneous environments · MapReduce

1 Introduction

Current data intensive applications need to process larger and larger data sets with the explosion of information. Due to the large data sets of applications, users prefer to use more and more processors or computers together to ensure the execution time of the applications reasonable and acceptable. This need has promoted the development of MapReduce, which is one of the most popular programming and scheduling models to process and generate large data sets [8]. MapReduce 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 [8]. MapReduce is used in Cloud Computing in the beginning [2, 3, 7, 23, 24]. It is initiated by Google, together with GFS [12] and BigTable [4] comprising backbone of Google's Cloud Computing platform. Apart from the Cloud Computing platform, MapReduce is also ported to work on GPU and multiprocessors. In addition, it is also extended to solve more loose-coupling problems [6, 9, 10, 18, 20, 22, 27].

For a MapReduce job (i.e., an application that is implemented based on the MapReduce programming model), its data set is divided into many small data sets. When a MapReduce system starts to execute a MapReduce job, the MapReduce scheduler[1] in the system launches a map task for each of the small data sets, and launches a group of reduce tasks to collect the results of all the map tasks. After the division, the MapReduce scheduler distributes these tasks onto different nodes according to the location of the tasks' data sets. In this way, all the nodes (called as *workers*) execute the tasks which are assigned to them in parallel. Since every node needs to execute many tasks, MapReduce scheduler launches a *task scheduler* for each node to manage tasks. Furthermore, since a MapReduce job is not completed until all the data is processed completely, the execution time of the MapReduce job is decided by the last finished tasks (i.e., the weakest link effect).

It is not a serious problem in homogeneous environments, since homogeneous nodes execute tasks with the same data set size in similar time. In heterogeneous environments, on the other hand, the execution time of a MapReduce job is prolonged by the last finished tasks (called as *straggler tasks*) seriously since workers require

[1] A MapReduce scheduler is a scheduler that schedules map and reduce tasks.

various time in accomplishing even the same tasks due to their differences, such as capacities of computation and communication, architectures, and memorizes.

One of the most popular solutions of this problem in MapReduce is launching backup tasks for straggler tasks on fast node. If a MapReduce scheduler launches a backup task γ_b for a straggler task γ, the small data set of γ is processed completely when either γ_b or γ finishes. In this case, if γ_b finishes before γ, the execution time of the job is reduced.

Although current MapReduce schedulers try to launch backup tasks for straggler tasks, they fail to detect straggler tasks correctly due to the wrong-estimated remaining time of all the tasks [13, 28]. The wrong detected straggler tasks cause at least two problems. First, launching backup tasks for these wrong straggler tasks cannot improve the performance of the MapReduce job since the real straggler tasks still prolong the execution time. Second, the backup tasks which are launched for the wrong straggler tasks waste system resources. The contention on the system resources even degrades the overall performance of the MapReduce job.

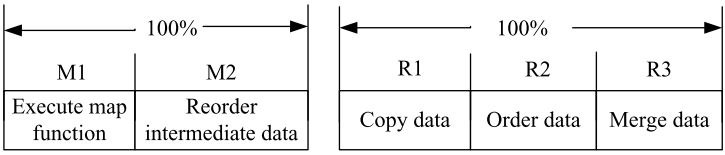
The wasting of system resources is one of the main problems of the backup strategy. Currently, MapReduce schedulers classify nodes into fast nodes and slow nodes, so that backup tasks can be launched on fast nodes. However, slow nodes can be further classified into *map slow nodes* and *reduce slow nodes* in a real system, since it is very possible that a node processes map tasks fast but processes reduce tasks slow and vice versa. We use map/reduce slow nodes to represent the nodes that execute map/reduce tasks slow than most of other nodes. The undistinguishing between map slow nodes and reduce slow nodes wastes system resources. Let us take a reduce task γ that needs a backup task for example. Current MapReduce schedulers will not launch the backup task on a slow node N_s. However, if N_s is only a *map slow node*, launching the backup task of γ on N_s can utilize resources on N_s efficiently and improve the overall performance, since N_s can process reduce tasks fast.

In order to detect straggler tasks and launch backup tasks efficiently, we propose a *History-based Auto-Tuning* (HAT) MapReduce scheduler. HAT incorporates historical information recorded on each node to tune parameters and detect straggler tasks dynamically. HAT further classifies slow nodes into map slow nodes and reduce slow nodes. In this way, HAT can launch backup tasks for map straggler tasks on reduce slow nodes and launch backup tasks for reduce straggler tasks on map slow nodes.

The most important contributions of this paper are three-fold:

- HAT detects straggler tasks correctly based on the historical information recorded on every node.
- HAT classifies slow nodes into map slow nodes and reduce slow nodes further.
- The experimental result shows that HAT scheduler can achieve a performance gain up to 37% for MapReduce applications.

The rest of this paper is organized as follows. Section 2 introduces the background information and the motivation of HAT MapReduce scheduler. Section 3 presents the HAT MapReduce scheduling algorithms. Section 4 reports the implementation details of HAT scheduler. Section 5 evaluates the performance of HAT. Section 6 describes the related works. Section 7 draws the conclusion with pointing out our future work.



(a) Map Task (b) Reduce Task

Fig. 1 Two phases of a map task and three phases of a reduce task

2 Background and motivation

Current MapReduce schedulers, such as Hadoop's scheduler [13, 25] and LATE [28], launch backup tasks for straggler tasks to improve performance of MapReduce applications. There are two main policies to detect straggler tasks: the least progress policy and the longest remaining time policy. For example, Hadoop's scheduler uses the least progress policy and LATE scheduler uses the longest remaining time policy. In the least progress policy, tasks with the least progress are the straggler tasks. In the longest remaining time policy, the tasks with the longest remaining time are the straggler tasks.

Both policies monitor the progress of every task using *progress score* (ranges from 0 to 1). In current MapReduce system, the execution of a map task comprises two phases and the execution of a reduce task comprises three phases as shown in Fig. 1. Therefore, the progress score of a task comprises from the progress score of every phase. Current MapReduce schedulers, such as Hadoop's scheduler and LATE, assume that $M1$, $M2$, $R1$, $R2$, and $R3$ are 1, 0, $\frac{1}{3}$, $\frac{1}{3}$, and $\frac{1}{3}$, respectively.

The progress of a phase, denoted by PS_{phase}, can be calculated in (1). In the equation, M is the number of key/value pairs that have been processed in the phase and N is the overall number of key/value pairs that needed to be processed in the phase.

$$PS_{\text{phase}} = \frac{M}{N}. \tag{1}$$

Take a task γ for example. If γ is a map task, since the first phase occupies the overall progress score, the progress score of γ is the progress $M1$. If γ is a reduce task and the first K phases of γ has finished, since each phase occupies $\frac{1}{3}$ of the progress score, PS_γ is calculated by adding the progress score of the finished phases and the progress score of the current phase. Therefore, the progress score of γ, denoted by PS_γ, is calculated in (2).

$$PS_\gamma = \begin{cases} PS_{\text{phase}} & \gamma \text{ is a map task,} \\ \frac{1}{3} * K + \frac{1}{3} * PS_{\text{phase}} & K \in (0, 1, 2), \gamma \text{ is a reduce task.} \end{cases} \tag{2}$$

For a MapReduce system with n running tasks $(\gamma_1, \gamma_2, \ldots, \gamma_n)$, the average progress score of the n running tasks, denoted by PS_{avg}, is calculated in (3).

$$PS_{\text{avg}} = \sum_{i=1}^{n} PS_i / n. \tag{3}$$

Suppose task γ_j's progress score is PS_j and it has run T_j seconds ($j \in (1, 2, \ldots, n)$). If the least progress policy is used to detect straggler tasks, γ_j is a straggler task only when $PS_j \leq PS_{\text{avg}} - 20\%$.

On the other hand, if the longest remaining time policy is used, the remaining time of all the n tasks needs to be calculated further. Then the scheduler chooses the tasks with the longest remaining time as straggler tasks. To calculate the remaining time of task γ_j, the progress rate of γ_j, denoted by PR_j, is calculated first in (4). Based on (4), the remaining time of γ_j, denoted by TTE_j, is calculated, (5).

$$PR_j = PS_j / T, \tag{4}$$

$$TTE_j = (1.0 - PS_j)/PR_j = T \times \frac{1.0 - PS_j}{PS_j}. \tag{5}$$

In most cases, the longest remaining time policy works better than the least progress policy [28]. This is because the tasks with the least progress score are not always the last finished tasks. For example, in a MapReduce system that has six tasks $(\gamma_1, \gamma_2, \ldots, \gamma_6)$, suppose their progress scores are 0.7, 0.5, 0.9, 0.9, 0.9, and 0.9, respectively. We further suppose that they need 100, 30, 10, 10, 10, and 10 seconds to finish their work. In this case, $PS_{\text{avg}} = (0.7 + 0.5 + 0.9 * 4)/6 = 0.8$. The least progress policy classifies γ_2 to be a straggler task. However, γ_1 is the real straggler task since γ_1 needs more time to finish its work.

On the other hand, if the progress score of all the tasks can be calculated accurately, the longest remaining time policy can always detect out the real straggler tasks. However, current MapReduce schedulers cannot calculate the progress score accurately, since $M1$, $M2$, $R1$, $R2$, and $R3$ are set to be constant values 1, 0, $\frac{1}{3}$, $\frac{1}{3}$, and $\frac{1}{3}$. In the real execution, the values of them are totally different for different hardware settings and different MapReduce applications.

For example, given a node with $R1$, $R2$, and $R3$ equal to 0.6, 0.2, and 0.2, respectively. Suppose a reduce task γ has finish the first phase and has run T seconds on the node, the remaining time of γ is $T * \frac{1-0.6}{0.6} = 0.67T$ seconds. However, since the values of $R1$, $R2$, and $R3$ are all $\frac{1}{3}$ in current schedulers, the calculated remaining time of γ is $T * \frac{1-1/3}{1/3} = 2T$ seconds. Based on the wrong remaining time, the longest remaining time policy cannot find real straggler tasks.

To this end, we propose HAT MapReduce scheduler, which uses a *history-based auto-tuning strategy*, to tune the values of $M1$, $M2$, $R1$, $R2$, and $R3$ based on the historical values of them in the completed tasks. Based on the specific values of them for the current hardware features and application features, HAT can estimate the progress scores of running tasks accurately, and hence can find real straggler tasks. HAT is implemented using the longest remaining time policy.

3 History-based auto-tuning MapReduce scheduling

This section presents HAT, a History-based Auto-Tuning MapReduce scheduler. First, we give an overview of HAT. Then we introduce a historical-based auto-tuning strategy for tuning runtime parameters used by HAT. Lastly, we present the detailed algorithms in HAT.

Algorithm 1 Runtime algorithm of HAT

Input: Key/Value pairs. Output: Statistical results.

Initialize the scheduler:

Step1: Every worker reads in historical information and tunes parameters using the history-based auto-tuning strategy as illustrated in Sect. 3.2.

Process tasks:

Step2: Every worker computes progress scores of all the running tasks using the progress monitoring algorithm as illustrated in Sect. 3.3.

Step3: HAT processes tasks and detects straggler tasks using the straggler detecting algorithm as illustrated in Sect. 3.4.

Step4: HAT detects slow nodes (either map slow nodes or reduce slow nodes) using the slow node detecting algorithm as illustrated in Sect. 3.5.

Step5: HAT launches backup tasks on appropriate nodes using the backup task launching algorithm as illustrated in Sect. 3.6.

Termination: HAT collects results and updates historical information on every node.

3.1 Overview of HAT

HAT detects straggler tasks based on the accurate progress score and achieves better performance compared with Hadoop and LATE scheduler. Algorithm 1 lists the runtime algorithm of HAT.

When HAT starts to execute a MapReduce job, each worker reads in the historical information from local node and sets them to be the default values of the parameters in this execution (to be described shortly). The historical information contains the values of $M1$, $M2$, $R1$, $R2$, and $R3$. Based on the dynamic-tuned $M1$, $M2$, $R1$, $R2$, and $R3$, HAT can compute progress scores of tasks more accurate, which is the basis of straggler task detecting. Meanwhile, HAT detects slow nodes according to the average progress rates of map tasks and reduce tasks on every node (to be described shortly). If there are any straggler tasks, HAT launches backup tasks for these straggler tasks. After all the data sets have been processed, HAT terminates the MapReduce job and reports the final result.

3.2 History-based auto-tuning strategy

As mentioned before, different worker process tasks in different speed. Therefore, each worker has different $M1$, $M2$, $R1$, $R2$, and $R3$. To obtain them for each worker accurately, HAT uses a history-based auto-tuning strategy to tune the value of them dynamically. In the strategy, each worker reads in the historical values of $M1$, $M2$, $R1$, $R2$, and $R3$ from the corresponding node as their default values when the worker was started. Once a map task finishes on the worker, $M1$ and $M2$ are updated. Once a reduce task finishes on the worker, $R1$, $R2$, and $R3$ are updated.

For any recorded weights of phases (i.e., $M1$, $M2$, $R1$, $R2$, $R3$), if the recorded value is V_{old} and the corresponding value of it in the just finished task is V_{finished}, HAT updates the recorded value to V_{new} that is calculated in (6). HP represents the weight of old values of $M1$ in the new value of it.

$$V_{\text{new}} = V_{\text{old}} * HP + V_{\text{finished}} * (1 - HP) \tag{6}$$

As shown in (6), if HP is too large (close to 1), V_{new} mostly depends on V_{old}. In this condition, V_{new} cannot reflect the up-to-date features of the current running tasks. On the other hand, if HP is too small (close to 0), the appropriate values of the weights may be destroyed by random factors, since V_{finished} is likely to be influenced by random events.

In addition, there is not any additional communication when a worker reads and updates historical information, since every worker reads and writes historical information from local node. So HAT scheduler is scalable.

3.3 Progress monitoring algorithm

During the execution of a MapReduce application, HAT computes the progress scores of all the running tasks periodically (every 100 ms). Given a task γ that is running in HAT. Suppose K phases of γ has been finished, (7) and (8) compute the progress score of γ, in which PS_{phase} is computed according to (1).

$$\text{For map task:} \quad PS_\gamma = \begin{cases} M1 \times PS_{\text{phase}} & \text{if } K = 0, \\ M1 + M2 \times PS_{\text{phase}} & \text{if } K = 1. \end{cases} \tag{7}$$

$$\text{For reduce task:} \quad PS_\gamma = \begin{cases} R1 \times PS_{\text{phase}} & \text{if } K = 0, \\ R1 + R2 \times PS_{\text{phase}} & \text{if } K = 1, \\ R1 + R2 + R3 \times PS_{\text{phase}} & \text{if } K = 2. \end{cases} \tag{8}$$

3.4 Straggler tasks detecting algorithm

Based on the accurate progress scores of tasks that are computed according to (7) and (8), HAT detects straggler tasks according to Algorithm 2. A task γ is a straggler task only when it fulfills the following two restrictions. First, γ is a slow task. Second, γ is one of the tasks with the longest remaining time.

Let PR_γ and PR_{avg} represent the progress rate of γ and the average progress rate. γ is a slow task only when its progress rate fulfills (9). *Task_Cap* is a cap of slow proportion in the equation.

$$PR_\gamma < (1.0 - \mathit{Task\_Cap}) \times PR_{\text{avg}}. \tag{9}$$

According to (9), if Task_Cap is too small (close to 0), HAT will classify some fast tasks into slow tasks. On the other hand, if Task_Cap is too large (close to 1), HAT will classify some slow tasks into fast tasks.

For all the slow tasks, HAT computes the remaining time of them according to (5). HAT chooses those slow tasks with the longest remaining time to be straggler tasks using the longest remaining time policy. To handle the fact that backup tasks of straggler tasks cost resources, HAT limits the number of straggler tasks and backup tasks. Therefore, a cap on the number of straggler tasks (i.e., the number of backup tasks since HAT only allows one backup task for each straggler task), denoted by *Strag_Cap*, is used. Suppose the number of the overall running tasks is *Task_Num*, the up-bound of the number of straggler tasks, denoted by *Strag_UB*, is

Algorithm 2 Straggler tasks detecting algorithm

While (the job is still running) {
Every worker computes progress rate of every tasks that are running on it.
HAT computes the average progress rate of all the running tasks.
Every worker determines slow tasks according to (9).
Every worker reports the list of slow tasks running on it.
HAT computes the remaining time for all the slow tasks according to (5) and orders the tasks in descending order according to the remaining time.
HAT computes the up-bound of the number of straggler tasks, Strag_UB.
If (the number of slow tasks $\leq$ Strag_UB)
All the slow tasks are detected as straggler tasks.
Else if (the number of slow tasks $>$ Strag_UB)
HAT selects Strag_UB slow tasks with the longest remaining time as straggler tasks.
HAT inserts all the straggler tasks into straggler map/reduce task list.
usleep(100000); //HAT detects straggler tasks every 100 ms.
}

Strag_Cap $*$ *Task_Num*. If Strag_Cap is too small (close to 0), some really straggler tasks is overlooked by HAT. On the other hand, if Strag_Cap is too large (close to 1), there can be too many straggler tasks. Too many backup tasks for these straggler tasks cost a lot of system resources. Algorithm 2 lists the detailed straggler detecting algorithm.

3.5 Slow nodes detecting algorithm

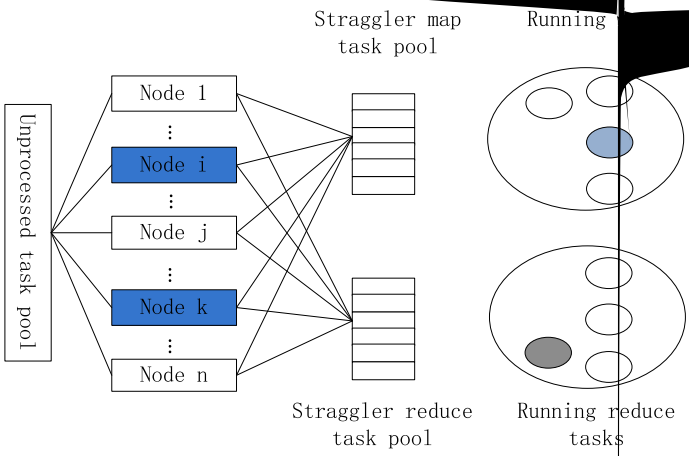
To detect slow nodes in the system, HAT uses the average progress rate of the running map/reduce tasks on a node to represent the map/reduce task progress rates of the node. The nodes with the smallest map/reduce task progress rate are map/reduce slow nodes. Given a node Φ with M map tasks and R reduce tasks. The map/reduce task progress rates of Φ, denoted by MR_Φ and RR_Φ, are calculated in (10). PR_i is the progress rate of the ith map/reduce task.

$$\begin{cases} MR_\Phi = \frac{\sum_{i=1}^{M} PR_i}{M}, \\ RR_\Phi = \frac{\sum_{i=1}^{R} PR_i}{R}. \end{cases} \tag{10}$$

For node Φ, if $MR_\Phi < (1 - Node\_Cap) * MR_{\text{avg}}$, it is a map slow node. If $RR_\Phi < (1 - Node\_Cap) * RR_{\text{avg}}$, it is a reduce slow node. MR_{avg} and RR_{avg} are the average map/reduce tasks progress rate of all the nodes. *Node_Cap* is a cap of slow proportion of the slow node.

Therefore, if *Node_Cap* is too small (close to 0), HAT will classify some fast nodes into slow nodes. On the other hand, if *Node_Cap* is too large (close to 1), HAT will classify some slow nodes into fast nodes.

To limit the number of slow nodes, a cap on the number of slow nodes, denoted by *SN_Cap*, is introduced in HAT. Suppose the number of the overall node


Straggler map task pool
Running
Unprocessed task pool
Node 1
Node i
Node j
Node k
Node n
Straggler reduce task pool
Running reduce tasks

```
<HAT>
        <MAP><M1>0.80</M1><M2>0.20</M2></MAP>
        <REDUCE><R1>0.59</R1><R2>0.19</R2><R3>0.22</R3></REDUCE>
</HAT>
```

Fig. 3 An example of $M1$, $M2$, $R1$, $R2$, and $R3$ that are recorded in XML format

the corresponding task pool. In this way, HAT always launches backup task for the straggler task which prolongs the execution time most serious first.

As mentioned before, every node records the values of $M1$, $M2$, $R1$, $R2$, and $R3$, which partly reflect the execution features of tasks on the node. For easy maintaining, the values are stored in XML format, as shown in Fig. 3. Every node uses a XML parser to read in the stored values and takes them as the default values of $M1$, $M2$, $R1$, $R2$, and $R3$ in the current execution.

In HAT, all the nodes prefer executing unprocessed tasks from the *unprocessed task pool* rather than launching backup tasks. Due to the large data set of tasks, all the nodes prefer to execute tasks whose data set is stored on local node.

5 Performance evaluation

This section evaluates the performance of HAT. Due to hardware limits, we use a cluster that comprises from five computers. In order to simulate heterogeneous environment, we have installed different number of virtual machines on the homogeneous computers. Each computer has 1 GB RAM and each virtual machine runs Linux 2.6.24. We implement HAT scheduler on Hadoop 0.19.1. Since Hadoop is implemented in Java, we use the latest JDK (i.e., J2SDK 1.6.0.10).

In order to demonstrate HAT scheduler can improve the performance of MapReduce applications in most of hardware scenarios, we simulate two heterogeneous environments with different settings (fast setting and slow setting). Table 1 lists the detailed settings. Note that, we run a CPU-intensive program on one of the computers that have two virtual machines to simulate two extremely slow nodes in the slow setting.

Two classic benchmarks, *Sort* and *WordCount*, are used to evaluate the performance of our HAT scheduler. The two benchmarks are always used to evaluate the performance of MapReduce schedulers, such as LATE [28] and Phoenix [18]. For each test, every benchmark is run ten times and the average execution time is used as the result.

5.1 Performance of HAT

Since slow setting provides more heterogeneity, we use slow setting as the main setting of our experiments. Experiments on the fast setting show similar result to the slow setting. Figure 4 show the performance of *Sort* and *WordCount* in HAT, Hadoop and LATE scheduler. We can see that HAT significantly improves the performance of *Sort*, with the performance gain up to 37%. Meanwhile, *WordCount* has achieved up to 16% performance gain with HAT compared with Hadoop. However, on the

Table 1 Setting of hardware system

Settings	VM/PC	No. of PC	No. of nodes	Write rate (MB/s)
Fast setting	1	1	1	2.87
	2	3	$2*3=6$	1.40
	bare Linux	1	1	3.43
Slow setting	1	1	1	2.87
	2	2	$2*3=4$	1.40
	2	1 extremely slow PC	2	1.34
	bare Linux	1	1	3.43

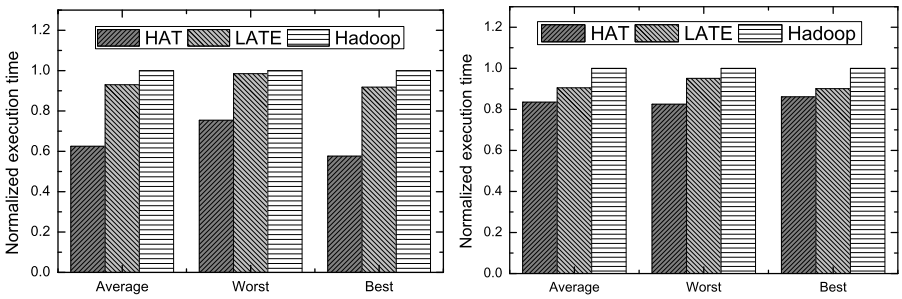


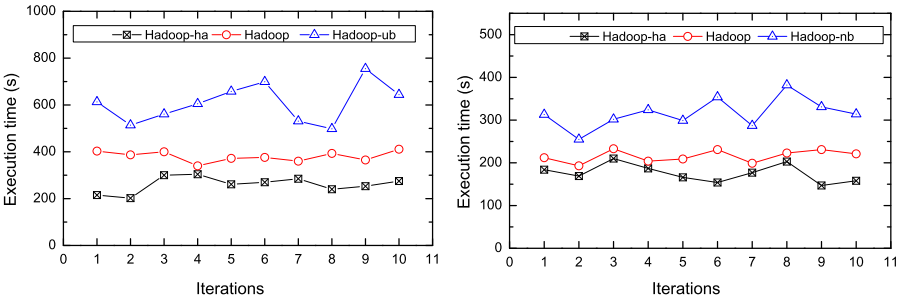
Fig. 4 Execution time of *Sort* and *WordCount* in the slow setting

other hand, both *Sort* and *WordCount* have achieved a slightly performance gain with the LATE scheduler, which uses the longest remaining time policy as well. The performance results of HAT are collected with best configured parameters (i.e., HP, Task_Cap, Node_Cap, SN_Cap, and Strag_Cap). We will describe the way to choose the appropriate values for them shortly.

As shown in Fig. 4, both benchmarks achieve a slightly better performance in LATE compared with Hadoop. The performance gains origin from the longest remaining time strategy in launching backup tasks [28]. Since HAT estimates the progress of tasks accurately, it can detect straggler tasks more accurate. Therefore, *Sort* and *WordCount* achieve better performance in HAT compared with LATE.

5.2 Effectiveness of backup strategy and history-based auto-tuning strategy

In order to evaluate the effectiveness of backup strategy and the history-based auto-tuning strategy in HAT, we compare the performance of *Sort* that is scheduled by Hadoop, Hadoop-nb (Hadoop without backup strategy) and Hadoop-ha (Hadoop with history-based auto-tuning strategy) on both the fast setting and the slow setting. *WordCount* shows similar results. Figure 5 shows the performance of *Sort* scheduled by the three schedulers on the two settings, respectively. From the figure, we can see that the backup strategy can significantly improve the performance of MapReduce



(a) Slow setting (b) Fast setting

Fig. 5 Execution time of *Sort* on the slow setting and the fast setting

Table 2 The recorded/real values of $M1$, $M2$, $R1$, $R2$, and $R3$

	Map task		Reduce task		
	$M1$	$M2$	$R1$	$R2$	$R3$
Node1	0.8/0.78	0.2/0.22	0.59/0.62	0.19/0.23	0.22/0.15
Node2	0.77/0.77	0.23/0.23	0.46/0.42	0.06/0.03	0.48/0.55
Node3	0.75/0.66	0.25/0.34	0.44/0.40	0.43/0.45	0.24/0.15
Node4	0.74/0.77	0.26/0.23	0.62/0.64	0.13/0.06	0.25/0.32
Node5	0.81/0.82	0.19/0.18	0.43/0.44	0.14/0.04	0.43/0.52
Node6	0.73/0.77	0.27/0.23	0.51/0.53	0.19/0.12	0.30/0.35
Node7	0.71/0.67	0.29/0.33	0.51/0.50	0.11/0.06	0.38/0.44
Node8	0.79/0.78	0.21/0.22	0.46/0.41	0.13/0.48	0.41/0.11

applications while the backup strategy can enhance the stability of execution time as well.

To demonstrate HAT can tune the values of $M1$, $M2$, $R1$, $R2$, and $R3$ accurately using the *history-based auto-tuning strategy*, Table 2 lists the recorded values and the real values of them on every node. For map tasks, the difference between real values and the recorded values are less than 5%. For reduce tasks, in most cases, the difference between real values and the recorded value are less than 10%. Both the recorded values and the real values are far from the constant values of them employed in Hadoop's scheduler and LATE scheduler (i.e., 1, 0, $\frac{1}{3}$, $\frac{1}{3}$, and $\frac{1}{3}$). Based on the accurate values of them, HAT detects the straggler tasks accurately. Therefore, HAT can improve the performance of MapReduce applications in heterogeneous environments.

5.3 Appropriate parameters in HAT

HAT uses five parameters (i.e., HP, Task_Cap, Node_Cap, SN_Cap, and Strag_Cap) to configure the scheduler for different hardware architecture and different applications. In order to find appropriate values for the parameters, we tune one parameter while keeping all the other parameters static. Following the runtime algorithm of HAT

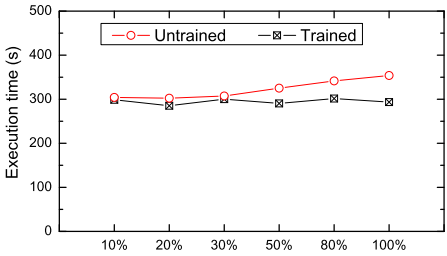


Fig. 6 Performance of *Sort* with different HP in the trained and untrained scenarios

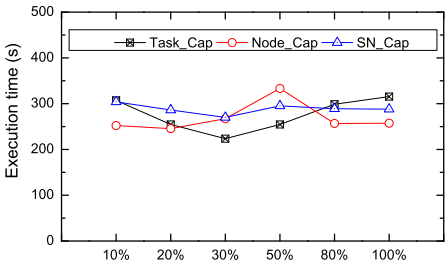


Fig. 7 Performance of *Sort* with different Task_Cap, Node_Cap and SN_Cap

that is proposed in Sect. 3.1, we evaluate the performance of *Sort* with different HP, Task_Cap, Node_Cap, SN_Cap, and Strag_ Cap. Experiment on *WordCount* shows similar results.

Figure 6 shows the performance of *Sort* with different HP in two scenarios: untrained scenario and trained scenario. HP is the weight of the recorded values of $M1$, $M2$, $R1$, $R2$, and $R3$ in the new values of them as defined in Sect. 3.2. We construct the trained scenario by executing *Sort* for two times before the current execution and construct the untrained scenario by setting the recorded $M1$, $M2$, $R1$, $R2$, and $R3$ to 1, 0, $\frac{1}{3}$, $\frac{1}{3}$, and $\frac{1}{3}$ manually just like the assumption in Hadoop and LATE scheduler.

From the figure, we can see that the value of HP does not affect the performance of *Sort* too much in the trained scenario. However, the performance of *Sort* degrades with the increasing of HP in the untrained scenario. The high and static performance of *Sort* in the trained scenario is resulted from the well-trained value of $M1$, $M2$, $R1$, $R2$, and $R3$. Figure 6 also suggests to use small HP if an application is executed for the first time. In this way, the value of $M1$, $M2$, $R1$, $R2$, and $R3$ can be tuned based on the current execution rapidly. Note that, if HP equals to 100%, the values of $M1$, $M2$, $R1$, $R2$, and $R3$ in the current job equal to the recorded value. In this case, HAT scheduler is the same to the LATE scheduler in the untrained scenario. In the following experiments, HP is 20%.

Figure 7 shows the performance of *Sort* with different Task_Cap, Node_Cap, and SN_Cap in the slow setting. From the figure, we can see that the best values of *Task_Cap*, *Node_Cap,* and *SN_Cap* are 30%, 20%, 30%, respectively.

Task_Cap is the percentile of speed below which a task will be considered too slow to be a slow task as defined in Sect. 3.4. As shown in Fig. 7, *Sort* gains best performance when Task_Cap is 30%. Deduced from (9), the smaller Task_Cap is the more tasks are classified to slow tasks. Therefore, if Task_Cap is smaller than 30%, some fast tasks are classified to slow tasks and even straggler tasks. In this case, the launching of backup tasks for these wrong-classified tasks consume a lot of system

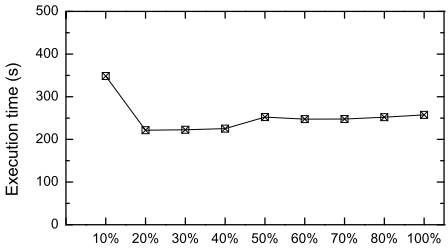


Fig. 8 The performance of *Sort* with different Strag_Cap

resources, so the overall execute time is prolonged. On the other hand, if Task_Cap is larger than 30%, some slow tasks and even straggler tasks are classified to fast tasks and none backup tasks are launched for them. These slow tasks will prolong the execute time as well.

Node_Cap is the percentile of speed below which a node will be considered too slow to be a map/reduce slow node as defined in Sect. 3.5. As shown in Fig. 7, *Sort* gains best performance when Node_Cap is 20%. If Node_Cap is small than 20%, some fast nodes are treated as map/reduce slow nodes by fault. In this case, the computing power of these wrong-classified nodes cannot be used to improve the performance by executing backup tasks for straggler tasks. On the other hand, if Node_Cap is larger than 20%, some map/reduce slow nodes are classified to fast nodes by fault. In the case, backup tasks may be launched on these slow nodes. Since the backup tasks on slow nodes will be finished later than the original straggler tasks, the overall execute time cannot be shortened.

SN_Cap is used to limit the maximum number of slow nodes as defined in Sect. 3.5. As shown in Fig. 7, *Sort* gains best performance when SN_Cap is 30%. SN_Cap is useful if Node_Cap has an unappropriate value, since SN_Cap limits the number of slow nodes. SN_Cap guarantees that there are not too many nodes are classified to slow nodes. If SN_Cap is smaller than 30%, some map/reduce slow nodes may be classified to fast nodes if Node_Cap is too small (e.g., smaller than 20%). On the other hand, if SN_Cap is larger than 30%, some fast nodes may be classified to slow nodes if Node_Cap is too large (e.g., larger than 30%). The wrong classifications lead to the poor performance as described in the last paragraph.

Strag_Cap is used to limit the maximum number of backup tasks as defined in Sect. 3.4. As shown in Fig. 8, *Sort* gains best performance when Strag_Cap equals to 30%. If Strag_Cap is smaller than 20%, HAT cannot launch backup tasks for all the straggler tasks because of the small number of backup tasks. On the other hand, if Strag_Cap is larger than 0.2, too many backup tasks will consume a lot of system resources, so the execution time of *Sort* is prolonged.

After a series of experiments, the best parameters for HAT are: HP = 20%, Task_Cap = 30%, Node_Cap = 20%, SN_Cap = 30%, Strag_Cap = 20% in our test bed for *Sort*. These parameters must be respecified for any new scenarios.

6 Related work

MapReduce is increasingly popular in large data set processing. There have been a lot of research works on its adaption and improvement [9, 20, 22, 27].

MapReduce scheduling has been extended to a great many of platforms, such as shared-memory multicore platform, Cell broadband engine platform, GPU, FPGA, and mobile platform. Phoenix [18] is a MapReduce framework on shared-memory multi-core architecture. Based on Phoenix, [26] and [14] optimized the performance of MapReduce on a multicore platform. De Kruijf and Sankaralingam [6] and Rafique et al. [17] implemented MapReduce frameworks for Cell broadband engine and Cell-based clusters. Mars [6] harnesses the GPU computation power and high memory bandwidth to accelerate MapReduce frameworks, such as Hadoop. In this case, MapReduce applications are executed on both CPUs and GPUs. Shan et al. [21] proposed FPMR for developers to create MapReduce programs on FPGA. Elespuru et al. [9] proposed a MapReduce framework on heterogeneous mobile platform.

A lot of efficient MapReduce scheduling algorithms have been proposed to improve the performance of MapReduce in many scenarios. Fischer et al. [11] proposes an idealized mathematic model to evaluate the cost of task assignments and develops a flow-based algorithm to optimally assign tasks. In [16], an infrastructure aware MapReduce scheduler is proposed. The scheduler monitors the tasks and evaluates the benefits of running each task on different nodes in real time. Based on the evaluation, the scheduler can decide the best distribution of tasks on nodes accordingly. In [5], Tiled-MapReduce scheduling algorithm is proposed. Tiled-MapReduce scheduler partitions a large MapReduce tasks into a number of small subtasks and iteratively processes one subtask at a time with efficient use of resources. In [27], a fair-sharing algorithm for a multiuser MapReduce system is proposed to arrange system resources (map/reduce task slots) for many users fairly. In [1], a MapReduce scheduling algorithm is proposed to minimize the execution time and improve the system resources utilization. The algorithm defines virtual machines (VM) and allocates the VMs to jobs, and to physical nodes. In [19], a Dynamic Priority (DP) parallel task scheduler is designed, which allows users to control their allocated capacity by dynamically adjusting their budgets. Note that our history-based auto-tuning strategy could be integrated into these MapReduce scheduling algorithms to improve the performance of MapReduce applications further.

Detecting straggler tasks in the execution of current job is another interesting issue. There are two policies to detect straggler tasks: the least progress policy and the longest remaining time policy. For example, Hadoop [13] uses the least progress policy while LATE [28] uses the longest remaining time policy to detect straggler tasks as mentioned in Sect. 2. Both policies need to estimate the progress of every map/reduce task accurately. ParaTimer [15] is a time-oriented progress indicator for parallel queries that ensembles of MapReduce jobs. However, the indicator can only estimate the progress of SQL queries. On the other hand, our HAT scheduler, which uses a history-based auto-tuning strategy, can estimate the progress of all the tasks accurately.

7 Conclusion

Traditional MapReduce schedulers cannot detect out straggler tasks accurately because the progress scores of tasks are calculated based on inaccurate weight of each

phase in the overall progress of a task. To address the problem, we have designed and implemented HAT: a History-based Auto-Tuning MapReduce scheduler. HAT estimates progress of a task accurately since it tunes the weight of each phase of a map task and a reduce task automatically according to the historical values of the weights. HAT further classifies slow nodes into map slow nodes and reduce slow nodes. In this way, HAT can launch backup tasks for reduce straggler tasks on map slow nodes and vice versa. Experimental results demonstrate that HAT can achieve up to 37% performance gain compared with Hadoop, and up to 16% performance gain compared with LATE scheduler.

One of our future works is to address the data locality problem when launching backup tasks, i.e., launching backup tasks on nodes with the corresponding data set of the straggler task. Another possible future direction is to profile a small-scale of the MapReduce application first before the real execution. In this way, the recorded values of $M1$, $M2$, $R1$, $R2$, and $R3$ are trained appropriately before the real execution. We will evaluate our HAT scheduler on more platforms, such as the Cell broadband engine platform and the rented Cloud Computing platform, to test the scalability of HAT.

References

1. Aboulnaga A, Wang Z, Zhang ZY (2009) Packing the most onto your cloud. In: Proceeding of the first international workshop on Cloud data management. ACM, New York, pp 25–28
2. Barroso LA, Dean J, Holzle U (2003) Web search for a planet: the Google cluster architecture. IEEE MICRO 23(2):22–28
3. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I (2009) Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener Comput Syst 25(6):599–616
4. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2006) Bigtable: a distributed storage system for structured data. In: Proceedings of the 7th USENIX symposium on operating systems design and implementation (OSDI 2006)
5. Chen R, Chen H, Zang B (2010) Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling. In: Proceedings of the 19th international conference on parallel architectures and compilation techniques. ACM, New York, pp 523–534
6. De Kruijf M, Sankaralingam K (2010) MapReduce for the cell broadband engine architecture. IBM J Res Dev 53(5):10
7. Dean J, Ghemawat S (2010) MapReduce: a flexible data processing tool. Commun ACM 53(1):72–77
8. Dean J, Ghemawat S (2004) Mapreduce: simplied data processing on large clusters. In: OSDI 2004: proceedings of 6th symposium on operating system design and implemention. ACM Press, New York, pp 137–150
9. Elespuru P, Shakya S, Mishra S (2009) Mapreduce system over heterogeneous mobile devices. In: Software technologies for embedded and ubiquitous systems, pp 168–179
10. Fang W, He B, Luo Q, Govindaraju NK (2010) Mars: accelerating MapReduce with graphics processors. IEEE Trans Parallel Distrib Syst
11. Fischer MJ, Su X, Yin Y (2010) Assigning tasks for efficiency in Hadoop. In: Proceedings of the 22nd ACM symposium on parallelism in algorithms and architectures. ACM, New York, pp 30–39
12. Ghemawat S, Gobioff H, Leung S-T (2003) The Google file system. In: SOSP 2003: proceedings of the 9th ACM symposium on operating systems principles. ACM, New York, pp 29–43
13. Hadoop (2011) Hadoop home page. http://hadoop.apache.org/
14. Jiang W, Ravi VT, Agrawal G (2010) A map-reduce system with an alternate API for multi-core environments. In: 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing. IEEE Press, New York, pp 84–93

15. Morton K, Balazinska M, Grossman D (2010) ParaTimer: a progress indicator for MapReduce DAGs. In: Proceedings of the 2010 international conference on management of data. ACM, New York, pp 507–518
16. Polo J, Carrera D, Becerra Y, Torres J, Ayguadé E, Steinder M, Whalley I (2010) Performance management of accelerated MapReduce workloads in heterogeneous clusters. In: 39th international conference on parallel processing (ICPP2010). San Diego, CA, USA
17. Rafique MM, Rose B, Butt AR, Nikolopoulos DS (2009) CellMR: a framework for supporting mapreduce on asymmetric cell-based clusters. In: IEEE international symposium on parallel & distributed processing. IPDPS 2009. IEEE Press, New York, pp 1–12
18. Ranger C, Raghuraman R, Penmetsa A, Bradski G, Kozyrakis C (2007) Evaluating mapreduce for multi-core and multiprocessor systems. In: HPCA 2007: proceedings of the 2007 IEEE 13th international symposium on high performance computer architecture. IEEE Computer Society, Washington, DC, pp 13–24
19. Sandholm T, Lai K (2010) Dynamic proportional share scheduling in hadoop. In: Job scheduling strategies for parallel processing. Springer, Berlin, pp 110–131
20. Schatz MC (2009) CloudBurst: highly sensitive read mapping with MapReduce. Bioinformatics 25(11):1363
21. Shan Y, Wang B, Yan J, Wang Y, Xu N, Yang H (2010) FPMR: MapReduce framework on FPGA. In: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays. ACM, New York, pp 93–102
22. Tian C, Zhou H, He Y, Zha L (2009) A dynamic MapReduce scheduler for heterogeneous workloads. In: Proceedings of the 2009 eighth international conference on grid and cooperative computing. IEEE Computer Society, Los Alamitos, pp 218–224
23. Vaquero LM, Rodero-Merino L, Caceres J, Lindner M (2008) A break in the clouds: towards a cloud definition. Comput Commun Rev 39(1):50–55
24. Varia J (2008) Cloud architectures. White paper of Amazon. jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf
25. Yahoo (2011) Yahoo! hadoop tutorial. http://developer.yahoo.com/hadoop/tutorial/
26. Yoo RM, Romano A, Kozyrakis C (2009) Phoenix rebirth: scalable MapReduce on a large-scale shared-memory system. In: IEEE international symposium on workload characterization. IISWC 2009. IEEE Press, New York, pp 198–207
27. Zaharia M, Borthakur D, Sarma JS, Elmeleegy K, Shenker S, Stoica I (2009) Job scheduling for multi-user mapreduce clusters. Technical report, UCB/EECS-2009-55, University of California at Berkeley
28. Zaharia M, Konwinski A, Joseph AD, Katz R, Stoica I (2008) Improving mapreduce performance in heterogeneous environments. In: 8th USENIX symposium on operating systems design and implementation. ACM, New York, pp 29–42