Improving MapReduce Performance Using Smart Speculative Execution Strategy

Qi Chen, Cheng Liu, and Zhen Xiao, *Member, IEEE*

Abstract—MapReduce is a widely used parallel computing framework for large scale data processing. The two major performance metrics in MapReduce are job execution time and cluster throughput. They can be seriously impacted by straggler machines—machines on which tasks take an unusually long time to finish. Speculative execution is a common approach for dealing with the straggler problem by simply backing up those slow running tasks on alternative machines. Multiple speculative execution strategies have been proposed, but they have some pitfalls: i) Use average progress rate to identify slow tasks while in reality the progress rate can be unstable and misleading, ii) Cannot appropriately handle the situation when there exists data skew among the tasks, iii) Do not consider whether backup tasks can finish earlier when choosing backup worker nodes. In this paper, we first present a detailed analysis of scenarios where existing strategies cannot work well. Then we develop a new strategy, maximum cost performance (MCP), which improves the effectiveness of speculative execution significantly. To accurately and promptly identify stragglers, we provide the following methods in MCP: i) Use both the progress rate and the process bandwidth within a phase to select slow tasks, ii) Use exponentially weighted moving average (EWMA) to predict process speed and calculate a task's remaining time, iii) Determine which task to backup based on the load of a cluster using a cost-benefit model. To choose proper worker nodes for backup tasks, we take both data locality and data skew into consideration. We evaluate MCP in a cluster of 101 virtual machines running a variety of applications on 30 physical servers. Experiment results show that MCP can run jobs up to 39 percent faster and improve the cluster throughput by up to 44 percent compared to Hadoop-0.21.

Index Terms—MapReduce, straggler, speculative execution, cluster throughput, cost performance

1 Introduction

MapReduce [1] is proposed by Google in 2004 and has become a popular parallel computing framework for large-scale data processing since then. In a typical MapReduce job, the master divides the input files into multiple map tasks, and then schedules both map tasks and reduce tasks to worker nodes in a cluster to achieve parallel processing. When a machine takes an unusually long time to complete a task (the so-called *straggler machine*), it will delay the *job execution time* (the time from job initialized to job retired) and degrade the *cluster throughput* (the number of jobs completed per second in the cluster) significantly. This problem is handled via *speculative execution*—slow task is backed up on an alternative machine with the hope that the backup one can finish faster. Google simply backs up the last few running map or reduce tasks and has observed that speculative execution can decrease the job execution time by 44 percent [1]. Due to the significant performance gains, speculative execution is also implemented in Hadoop [2] and Microsoft Dryad [3] to deal with the straggler problem.

Hadoop is a widely used open-source implementation of MapReduce. The original speculative execution strategy used in Hadoop-0.20 (we call it Hadoop-Original) simply identifies a task as a straggler when the task's progress falls behind the average progress of all tasks by a fixed gap. Previous work [4] found that this can be misleading in heterogeneous environments and consequently proposed a new strategy called LATE [4], which is implemented in Hadoop-0.21 with some modifications (we call it Hadoop-LATE). Hadoop-LATE keeps the *progress rate* (*progress/time*) of tasks and estimates their *remaining time* (*progress left/progress rate*). When a job has only a few map or reduce tasks left in its computation, Hadoop-LATE will select the slow task with the longest remaining time to backup.

Microsoft Dryad is another parallel computing framework which supports MapReduce. Its original speculative execution strategy is similar to that in Google MapReduce [1]. Later, Mantri [5] proposes a new speculative execution strategy for Dryad. The main difference between LATE and Mantri is that Mantri uses the task's *process bandwidth* (*processed data/time*) to calculate the task's *remaining time* (*data left/process bandwidth*). In addition, Mantri considers saving cluster computing resource in its strategy.

However, the strategies mentioned above have some pitfalls in identifying slow tasks and choosing backup worker nodes. They use the average *process speed* (progress rate or process bandwidth) of a task to estimate the remaining time, which assumes that a task makes progress at a stable rate. However, this assumption can break down in practice for a variety of reasons. First, in MapReduce, a task is divided into multiple phases with a fixed ratio of progress for each. However, the actual duration ratio of those phases tend to vary in different jobs and deploy environments (shown in Fig. 2), leading to the progress rate fluctuation across different phases. Second, reduce tasks can be launched asynchronously before all map tasks complete.

• *The authors are with the Department of Computer Science, Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing 100871, P.R. China. Email: {chenqi, liucheng, xiaozhen}@net.pku.edu.cn.*

Manuscript received 5 Jul. 2012; revised 23 Nov. 2012; accepted 7 Jan. 2013; date of publication 24 Jan. 2013; date of current version 5 Mar. 2014.
Recommended for acceptance by D. Talia.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.15

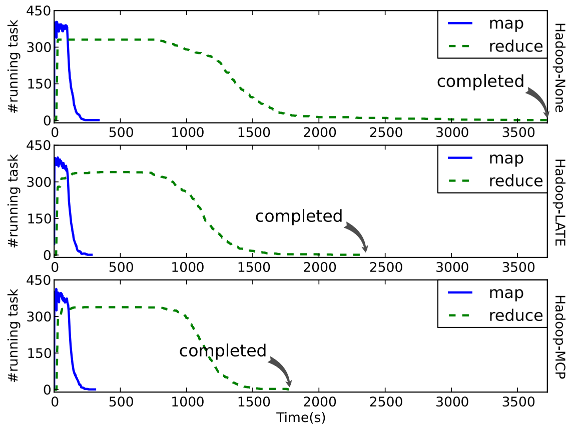


Fig. 1. Straggler phenomenon in different strategies.

In this case, reduce tasks that start later tend to have a higher process speed at the beginning because a certain amount of map outputs have been ready. However, after copying those ready map outputs, they will slow down to wait for new map outputs, leading to the drop of the process speed. Third, in a virtualized computing environment, such as Amazon EC2 [6], the I/O performance of worker nodes can be impacted significantly due to resource competition by co-hosted VMs as previously observed in [4]. As a result, the process speed of the tasks running inside the VM can vary. Finally, Hadoop and LATE do not consider whether backup tasks can finish earlier when choosing backup worker nodes, leading to some ineffective backups.

In this paper, we present a new set of speculative execution strategies called Maximum Cost Performance (MCP) to address the above issues successfully: i) Use both the progress rate and the process bandwidth within a phase to select slow tasks, ii) Use exponentially weighted moving average (EWMA) to predict process speed and calculate a task's remaining time, iii) Determine which task to backup based on the load of the cluster using a cost-benefit model, iv) Distinguish slow worker nodes by the process speed of map tasks completed on them, and v) Optimize data locality of map task backups. We have implemented MCP in Hadoop-0.21 (it can also fit into Hadoop-2.0) and call it Hadoop-MCP.

Fig. 1 compares the effectiveness of Hadoop-MCP with Hadoop-LATE and Hadoop-None (with speculative execution disabled). The experiment was conducted in a group of 100 VMs running on 30 Dell PowerEdge blade servers. We use the *Sort* benchmark in the standard Hadoop distribution on 90 GB of data. We start three sort jobs every 150 s and show how the number of running tasks varies with time in the first job. The figure shows that our algorithm (the bottom) can improve the job execution time by 25 percent over that of Hadoop-LATE (the middle) and by 51 percent over that of Hadoop-None (the top). More detailed evaluation can be found later in the paper.

In summary, we make the following contributions. First, we provide a detailed analysis of the pitfalls in the current speculative execution strategies. Second, we develop a new strategy named MCP on the basis of the analysis. Finally, we conduct an experiment on MCP and compare it with the current strategies. Experiment results show that MCP can run jobs up to 39 percent faster and improve the cluster throughput by up to 44 percent compared to Hadoop-0.21.

TABLE 1
Causes of Straggler

Internal factors	External factors
• heterogenous resource capacity of worker nodes • resource competition due to other MapReduce tasks running on the same worker node	• resource competition due to co-hosted applications • input data skew • remote input or output source being too slow • faulty hardware

The rest of the paper is organized as follows: Section 2 provides a background on the causes of stragglers and the existing speculative execution strategies. Section 3 analyzes the pitfalls of these strategies. Section 4 describes the details of our new algorithm. Section 5 evaluates its performance. Related work is described in Section 6. Section 7 concludes.

2 Background

In this section, we provide a description of MapReduce, give an overview of the causes of stragglers, and then describe the inner mechanisms of some widely used speculative execution strategies.

2.1 MapReduce Mechanisms

In a MapReduce cluster, after a job is submitted, a master divides the input files into multiple map tasks, and then schedules both the map tasks and the reduce tasks to worker nodes. A worker node runs tasks on its task slots and keeps updating the tasks' progress to the master by periodic heartbeat. Map tasks extract key-value pairs from the input, transfer them to some user defined map function and combine function, and finally generate the intermediate map outputs. After that, the reduce tasks copy their input pieces from each map task, merge these pieces to a single ordered (key, value list) pair stream by a merge sort, transfer the stream to some user defined reduce function, and finally generate the result for the job.

In general, a map task is divided into map and combine phases, while a reduce task is divided into copy, sort and reduce phases. Since Hadoop-0.20, reduce tasks can start when only some map tasks complete, which allows reduce tasks to copy map outputs earlier as they become available and hence mitigates network congestion. However, no reduce task can step into the sort phase until all map tasks complete. This is because each reduce task must finish copying outputs from all the map tasks to prepare the input for the sort phase.

2.2 Causes of Stragglers

We can categorize the causes for stragglers into internal and external reasons as shown in Table 1. Internal reasons can be solved by the MapReduce service provider, while external reasons cannot. For example, MapReduce clusters in the real world may be over-committed with multiple tasks running on the same worker node. This creates resource competition and may lead to heterogeneous performance. We can avoid this "internal reason" by limiting each worker node to run at most one task simultaneously or by only allowing tasks with different resource usage intensity to share the same worker node. However, we cannot avoid the resource competition due to co-hosted applications since we have no control over other users' VMs. Among these causes, the resource capacity heterogeneity of worker nodes is usually stable and foreseeable, while others are not. For most of these factors, speculative execution is an effective way to solve the straggler problem. However, *input data skew*—some tasks have to process a different amount of data from others, cannot be solved by simply duplicating the task on another worker node.

2.3 Previous Work

Several speculative execution strategies have been proposed in the literature, including MapReduce in Google [1], Hadoop [2], LATE [4], Dryad in Microsoft [3] and Mantri [5].

The original MapReduce implementation in Google and Dryad use the same speculative execution mechanism. They begin speculative execution only when the map or the reduce stage is close to completion. Then they select an arbitrary set of the remaining tasks to back up as long as slots are available, and mark a task as completed as soon as one of the task attempts completes. This strategy is very simple and intuitive. However, they do not consider the following questions: i) Are those remaining tasks really slow, or do they just have more data to process? ii) Whether the worker node chosen to run a backup task is fast or not? iii) Could the backup task complete before the original one?

Hadoop-Original improves this mechanism by using the progress of a task and starts the speculative execution when a job has no new map or reduce task to assign. It simply identifies a task as a straggler when the task's progress falls behind the average progress of all tasks by a fixed gap (i.e., 0.2). However, LATE finds that Hadoop-Original can be misleading in heterogeneous environments and thus makes some improvements. It keeps the progress rate of tasks and estimates their remaining time. Tasks with their progress rate below *slowTaskThreshold* are chosen as backup candidates, among whom the one with the longest remaining time is given the highest priority to be backed up. In addition, LATE considers a worker node to be slow if its *performance score* (the total progress or the average progress rate of all the succeeded and running tasks on it) is below the *slowNodeThreshold*. It will never launch any speculative task on these slow worker nodes. Moreover, LATE limits the number of backup tasks by *speculativeCap*. Compared with Hadoop-Original, it deals with the problems in question i) and ii), but still has some problems as we will discuss in Section 3. Hadoop-LATE is an implementation of the LATE strategy in Hadoop-0.21. It replace the *slowTaskThreshold* and *slowNodeThreshold* with the *std* (standard deviation) of all tasks' progress rate. The rationale is to let the *std* adjust the thresholds automatically. However, this may still cause misjudgment as we will see later.

The speculative execution strategy used in Mantri focuses more on saving cluster computing resource, i.e., task slots. It starts to back up outliers (e.g., abnormal tasks running slowly) when they show up instead of when all tasks have been assigned, which is much earlier than the other strategies. If a backup task has a high probability to complete earlier, Mantri will kill the original one when the cluster is busy (the so-called kill-restart scheme). On the other hand, it will simply duplicate the original one when the cluster is idle. Besides, Mantri estimates a task's remaining time based on the process bandwidth instead of the progress rate, which can avoid the unnecessary backups caused by input data skew. However, the kill-restart scheme is too radical as the new task is not guaranteed to complete earlier than the original one. Moreover, Mantri cares more about saving the cluster computing resource and less about reducing the job execution time for users. In addition, using the average process bandwidth can be misleading because some tasks do not always make progress at a stable rate as analyzed in the Section 1.

3 Pitfalls in the Previous Work

In this section, we analyze the pitfalls in existing speculative execution strategies.

3.1 Pitfalls in Selecting Backup Candidates

Hadoop-LATE and LATE use the average progress rate to select slow tasks and estimate their remaining time. They are based on the following assumptions:

- Tasks of the same type (map or reduce) process roughly the same amount of input data.
- Progress rate must either be stable or accelerate during a task's lifetime.

In the following, we present some scenarios where those assumptions break down.

3.1.1 Input Data Skew

As analyzed by Kwon et al. [7], tasks do not always process the same amount of data and may experience several types of data skew in MapReduce. When the input data has some big records that cannot be divided, the map tasks that process those records will process more data. Partitioning the intermediate data generated by the map tasks unevenly will also lead to the partition skew among the reduce tasks, typically when the distribution of keys in the input data set is skewed. Therefore, the first assumption can break down easily. This has also been analyzed in Mantri [5].

3.1.2 Phase Percentage Not Matching Corresponding Duration Ratio

The second assumption is made by LATE to simplify the calculation of a task's remaining time, but it may fail to hold in practice. For example, Hadoop allocates a fixed percentage of progress to each phase of a task, with 66.7 percent of map and 33.3 percent of combine for a map task, and 33.3 percent for each of the three phases (copy, sort, and reduce) in a reduce task. Such fixed settings make it easy to monitor a task's progress, but the progress rate calculated by $progress/time$ may be unstable. To demonstrate this, we analyze the time duration of all the phases defined in Hadoop for typical jobs, such as Grep, WordCount, and Sort, in our local cluster

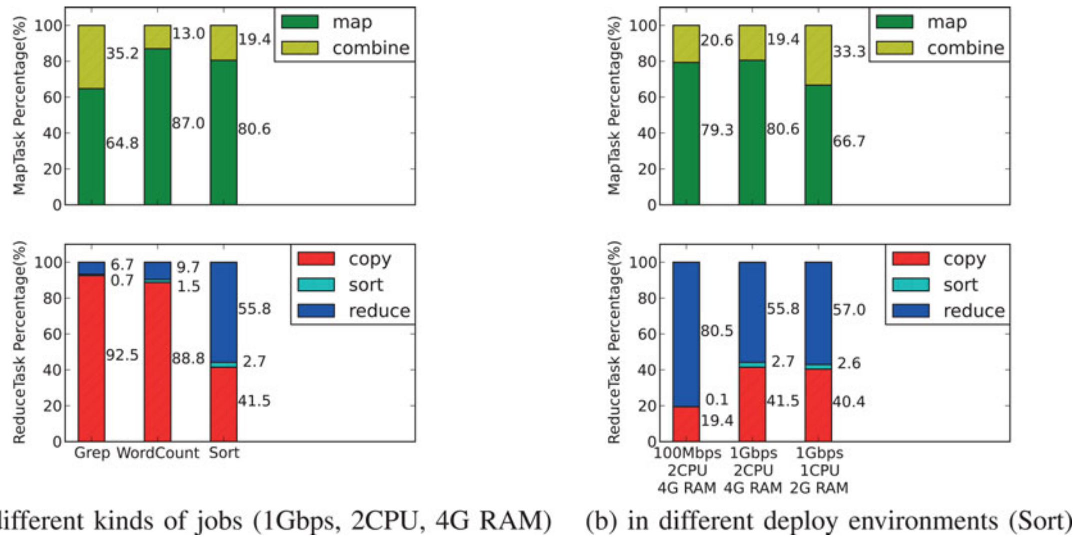


Fig. 2. Phases have different time duration ratios.

and show the results in Fig. 2. As we can see from the figure, the time duration ratio of phases varies a lot in different types of jobs and environments. In the three jobs, the sort phase consumes the shortest time and has the fastest progress rate in the reduce tasks. As a result, the progress rate of reduce tasks in those jobs will speed up from the copy to the sort phase and slow down from the sort to the reduce phase, which breaks the second assumption. In another experiment, we run grep on 6 GB of synthetic data and tune the regular expression so that its output sizes vary from 10 to 100 percent. We found that the ratio of the time duration of the three phases (copy, sort, reduce) in reduce tasks can vary substantially. Such fluctuation of progress rate will affect the estimation of a task's remaining time and mislead the judgement of straggler tasks.

Moreover, even if the progress rate always speeds up, LATE may still make mistakes. Take WordCount as an example (Fig. 3). Suppose we have two tasks running concurrently. One of them is a normal task (task 2), running in the map phase at 1 percent per second with 40 percent of the total task progress left, and the other is a slow task (task 1) running in the combine phase at 1 percent per second with 30 percent progress left. The remaining times of the normal and the slow tasks calculated by LATE will be $40\%/(1\%/\mathrm{s}) = 40$ s and $30\%/(1\%/\mathrm{s}) = 30$ s, respectively. In this case, LATE will give a higher priority to backup the normal task because it seems to have a longer remaining time. However, as the progress rate of the combine phase is three times that of the map phase in WordCount, the normal task (task 2) will be running at 3 percent per second once it enters the combine phase. In fact, the remaining time of the normal task (task 2) is $(40\% - 33\%)/(1\%/\mathrm{s}) + 33\%/(3\%/\mathrm{s}) = 18$ s due to the progress rate acceleration from the map phase to the combine phase.

3.1.3 Reduce Tasks Starting Asynchronously before All Map Tasks Complete

The second assumption may also fail to hold when the reduce tasks are launched asynchronously before all map tasks complete due to insufficient free slots in the cluster (shown in Fig. 4). In some jobs (e.g., Sort), the speed of the reduce tasks copying map outputs is much faster than the speed of the map tasks generating those outputs. In this case, if some reduce tasks are launched after a fraction of the map tasks complete, they will copy the generated map outputs at a high speed initially. But after that they will

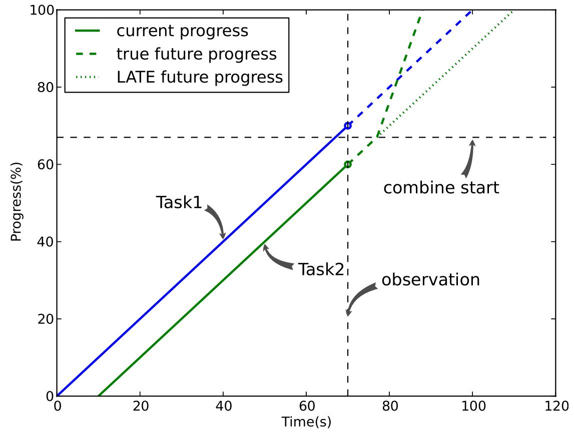


Fig. 3. A speed up situation where LATE will make a mistake.

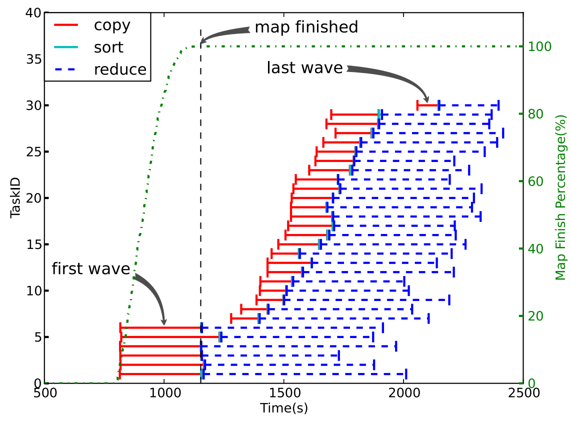


Fig. 4. Reduce tasks start in different groups.

have to wait for the map tasks to generate new outputs. This causes a slow down in the process speed of the copy phase and breaks the second assumption.

3.1.4 Using std as SlowTaskThreshold

Hadoop-LATE uses std as the slowTaskThreshold, which is not appropriate: sometimes it will misjudge too many stragglers, while other times it cannot identify any straggler at all. As an example, suppose that the reduce tasks are launched in two groups: g_1 and g_2, the number of the reduce tasks in g_2 is a times that of g_1, and the progress rate of reduce tasks in g_2 is b times that of g_1 ($b > 1$ for reduce tasks starting later will have a larger average progress rate). We can calculate the average progress rate of all running reduce tasks as follows:

$$progressRate_{avg} = \frac{1+ab}{1+a}. \tag{1}$$

Then the std of all reduce tasks can be calculated as

$$std = \sqrt{\frac{\left(1-\frac{1+ab}{1+a}\right)^2 + a\left(b-\frac{1+ab}{1+a}\right)^2}{1+a}} = \frac{\sqrt{a}(b-1)}{1+a}. \tag{2}$$

In Hadoop-LATE, a task is considered as a straggler when the following condition is satisfied:

$$progressRate_{avg} - progressRate_{task} > std. \tag{3}$$

Hence, we can calculate the difference between the average progress rate of all tasks and the progress rate of tasks in g_1 as follows:

$$\begin{aligned} progressRate_{avg} - progressRate_{g_1} &= \frac{1+ab}{1+a} - 1 \\ &= \frac{a(b-1)}{1+a}. \end{aligned} \tag{4}$$

In this case, when $a > 1$, $progressRate_{avg} - progressRate_{g_1} > std$ is always true. As a result, when the number of reduce tasks assigned in g_2 is larger than that in g_1, all the tasks in g_1 will be taken as stragglers even though they are running normally.

We can also calculate the difference between std and the average progress rate of all tasks as follows:

$$\begin{aligned} std - progressRate_{avg} &= \frac{1}{1+a}\left[\sqrt{a}(b-1) - (1+ab)\right] \\ &= \frac{-ba + (b-1)\sqrt{a} - 1}{1+a}. \end{aligned} \tag{5}$$

In this case, $std - progressRate_{avg} > 0$ is always true when $b = 6$ and $\frac{1}{9} < a < \frac{1}{4}$. Then $progressRate_{avg} - progressRate_{g_1}$ is always less than std which means that stragglers can never be identified.

This phenomenon shows up whenever tasks can be grouped into a fast set and a slow set, for example, data-local versus non-local map tasks (as explained in the Section 3.2.2) or small versus large tasks (i.e., tasks with different amount of data to process).

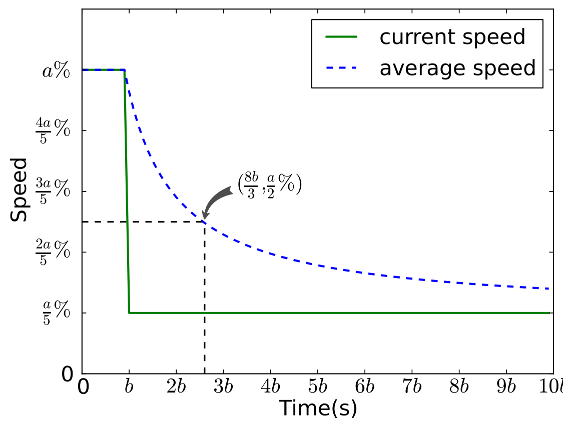


Fig. 5. Process speed falls down dramatically.

3.1.5 Long Time to Identify Stragglers

Since most of the MapReduce clusters are shared by users, the performance of worker nodes may degrade just because other users have launched some applications. This can cause some tasks to slow down dramatically. Unfortunately, it will take a long time for them to be identified as stragglers by using the average process speed. For example, suppose a task is running at the speed of a percent per second for b seconds when suddenly the speed falls to $\frac{a}{5}$ percent per second due to resource competition (shown in Fig. 5). The average process speed will fall down to $\frac{a}{2}$ percent per second only after $\frac{5}{3}b$ seconds. As a result, the remaining time estimated according to the average process speed will be much shorter than the actual value. Therefore, using the average progress speed cannot identify the straggler tasks in time and can even lead to some misjudgments.

3.2 Pitfalls in Selecting Backup Worker Nodes

3.2.1 Identifying Slow Worker Nodes Improperly

LATE and Hadoop-LATE use a threshold (e.g., *slowNodeThreshold*) to identity the straggler nodes. LATE uses the sum of progress of all the completed and running tasks on a worker node to represent the performance score of the node, while Hadoop-LATE uses the average progress rate of all the completed tasks on the node. They both consider a worker node as slow when the performance score of the node is less than the average performance score of all nodes by a threshold, and will never launch any speculative task on this slow node. However, some worker nodes may do more time-consuming tasks and get lower performance score unfairly. For example, they may do more tasks with a larger amount of data to process or they may do more non-local map tasks. As a result, such worker nodes are considered to be slow by mistake.

3.2.2 Choosing Backup Worker Nodes Improperly

Neither LATE nor Hadoop-LATE uses data locality to check whether backup tasks can finish earlier when choosing backup nodes. They assume that network utilization is sufficiently low during the map stage because most map tasks are data-local. As a result, they assume that non-local map tasks can run as fast as data-local map tasks. However, this assumption can break down easily: i) In a MapReduce cluster where multiple jobs are running simultaneously, the network bandwidth may be fully utilized because other jobs are busy copying map outputs to reduce tasks or writing the final outputs of reduce tasks to some stable file system, ii) Reduce tasks will copy map outputs concurrently along with the execution of map tasks, leading to bandwidth competition. In fact, we have observed that the execution time of a data-local map task can be over three times faster than that of a non-local map task, motivating us to consider data locality in our solution.

3.3 Summary

In this section, we have analyzed the speculative execution strategies proposed by previous work in detail and pointed out their pitfalls. These pitfalls may lead to misjudgement of straggler tasks, which wastes the cluster computing resources and degrades the efficiency of speculative execution. In fact, Yahoo! disabled speculative execution for some jobs due to the performance degradation. Facebook also disabled their speculative execution for reduce tasks [4]. The challenge here is how to back up only the right tasks on the right worker nodes at the right time. The next section presents our solution to this challenge.

4 Our Design

We propose a new speculative execution strategy named MCP for maximum cost performance. We consider the cost to be the computing resources occupied by tasks, while the performance to be the shortening of job execution time and the increase of the cluster throughput. MCP aims at selecting straggler tasks accurately and promptly and backing them up on proper worker nodes. To ensure fairness, we assign task slots in the order the jobs are submitted. Just like other speculative execution strategies, MCP gives new tasks a higher priority than backup tasks. In other words, MCP will not start backing up straggler map/reduce tasks until all new map/reduce tasks of this job have been assigned. MCP chooses backup candidates based on a prompt prediction of the tasks' process speed and an accurate estimation of their remaining time. Then, these backup candidates will be selectively backed up on proper worker nodes to achieve max cost performance according to the cluster load. In this section, we will present the implementation of MCP in detail.

4.1 Selecting Backup Candidates

4.1.1 Using EWMA to Predict Process Speed

In MCP, we predict tasks' process speed in the near future instead of simply using the past average rate. There are many prediction algorithms in the literature, such as exponentially weighted moving average and CUSUM [8]. In MCP, we choose the EWMA scheme which can be expressed as follows:

$$Z(t) = \alpha * Y(t) + (1 - \alpha) * Z(t-1), 0 < \alpha \leq 1, \quad (6)$$

where Z(t) and Y(t) are the estimated and the observed process speed at time t, respectively. α reflects a tradeoff between stability and responsiveness. In our implementation, we set α to be 0.2 according to the evaluation result in Section 5. To assure the accuracy of prediction, we will not start calculating tasks' process speed until it has executed for a certain amount of time (we call this period a speculative lag). To show how effective EWMA can be in predicting the process speed and the remaining time of a task, we run a Sort job in our cluster and then suddenly launch some I/O and CPU intensive workloads on one of the worker nodes. Fig. 6 shows the instantaneous speed and the remaining time of the reduce task running on this straggler node. We compare the EWMA scheme and the average scheme to the observed value. Results show that using EWMA scheme can predict the process speed and the remaining time of a task more accurately when a sudden speed drop occurs. EWMA is also effective in smoothing out speed oscillations when there is no sudden fluctuation in speed.

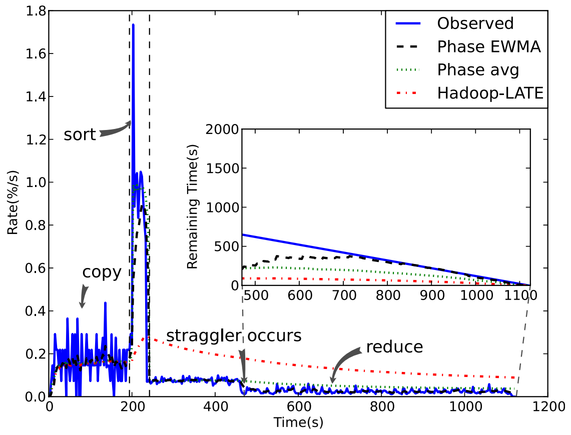


Fig. 6. Reduce task remaining time predicted by multiple methods.

4.1.2 Identifying Slow Tasks Using Per-Phase Process Speed

LATE identifies a task as slow when its progress rate is lower than the average progress rate by a fixed threshold. However, using the progress rate alone is insufficient to identify slow tasks. For example, large tasks which have more data to process may have a lower progress rate even though their processing bandwidth is normal. Using the process bandwidth alone is not sufficient either. Small tasks which have less data to process will be misjudged by the process bandwidth alone due to the impact of the constant task start up time. When we use both the progress rate and the process bandwidth together, we can avoid these misjudgments and achieve better precision.

Moreover, in Section 3.1.2, we have analyzed that a task's progress rate may vary greatly across different phases. Therefore, using the average rate may lead to misjudgement of slow tasks when the tasks are running in different phases. In our solution, we compare a task's process speed (both the process bandwidth and the progress rate) and estimate its remaining time in a smaller granularity. As said before, a map task is divided into map and combine phases, while a reduce task is divided into copy, sort and reduce phases. Therefore, we predict the process speed of each phase in a task by using the EWMA scheme, and use this per-phase process speed to identify slow tasks. Meanwhile, we estimate the remaining time of each phase in a task by using the process speed and the remaining data to process, and sum up the remaining time of all these phases to get the task's total remaining time.

4.1.3 Estimating Task Remaining Time and Backup Time

In MCP, a task that has the longest remaining time can get the highest priority to be backed up. As said before, a task's remaining time is estimated by the sum of the remaining time left in each phase. When a task is running in some phase cp (i.e., the current phase), the remaining time left in cp is estimated by the factors of the remain data and the process bandwidth in cp. However, the remaining time of the following phases fp is difficult to calculate since the task has not entered those phases yet. Therefore, we use the *phase average process speed* to estimate the remaining time of a phase est_time_p. The phase average process speed is the average process speed of tasks that have entered the phase. For those phases that no task has entered, we do not calculate their remaining time, which is fair to all tasks. Since tasks may process different amount of data, we adjust est_time_p by $factor_d$, which represents the ratio of the input size of this task to the average input size of all tasks. Now we can estimate the remaining time of tasks as follows:

$$\begin{aligned} rem\_time &= rem\_time_{cp} + rem\_time_{fp} \\ &= \frac{rem\_data_{cp}}{bandwidth_{cp}} + \sum_{p\ in\ fp} est\_time_p * factor_d, \end{aligned} \tag{7}$$

$$factor_d = \frac{data_{input}}{data_{avg}}. \tag{8}$$

To estimate the backup time of a slow task, we use the sum of est_time_p for each phase in this task as an estimation. Therefore, we can calculate the backup time as follows:

$$backup\_time = \sum_p est\_time_p * factor_d. \tag{9}$$

As we have analyzed in Section 3.1.3, when reduce tasks that start later are running in the copy phase, their process speed can be fast at the beginning and drop later. This will cause process speed fluctuation and impact the precision of time estimate. To avoid such impact, we estimate the remaining time of the copy phase as the time to copy all the completed map task outputs. We calculate the remaining time of the copy phase using the following equation:

$$rem\_time_{copy} = \frac{finish\_percent_{map} - finish\_percent_{copy}}{process\_speed_{copy}}. \tag{10}$$

In the equation above, the $process\_speed_{copy}$ is estimated by EWMA. The equation is reasonable because the process percentage of the copy phase in reduce tasks is limited by the percentage of completed map tasks.

Simply backing up a task that has the longest remaining time is not proper, as we can always get a task that has the longest remaining time. To make sure the task is slow enough, we consider whether backing up this task can effectively save cluster computing resources.

4.1.4 Maximizing Cost Performance of Cluster Computing Resources

Speculative execution has not only benefits, but also costs. In a Hadoop cluster, the cost of speculative execution is task slots, while the benefit is the shortening of the job execution time. We establish a cost-benefit model to analyze the tradeoff. In this model, the cost is represented as the time that the computing resources are occupied (represented as $slot\_number * time$), while the benefit is represented as the time saved by speculative execution. Backing up a task will occupy two slots for $backup\_time$ (both the original and the backup need to keep running until either completes) and save one slot $rem\_time - backup\_time$. In contrast, not backing it up will cost just one slot rem_time and benefit nothing. The profit of these two actions (backing it up or not) is the slot time that can be saved. Therefore, we can define the profit of the two actions as follows:

$$\begin{aligned} profit_{backup} = \alpha * (rem\_time - backup\_time) \\ - \beta * 2 * backup\_time, \end{aligned} \tag{11}$$

$$profit_{not\_backup} = \alpha * 0 - \beta * rem\_time, \tag{12}$$

where rem_time and $backup\_time$ are described in Section 4.1.3, and α and β are the weight of benefit and cost, respectively.

Then we will choose the action that gains more profit. If the profit of backing up this task outweighs that of not backing it up, we consider this task slow enough and select it as a backup candidate. Otherwise we will leave the task running normally

$$profit_{backup} > profit_{not\_backup} \Leftrightarrow \frac{rem\_time}{backup\_time} > \frac{\alpha + 2\beta}{\alpha + \beta}, \tag{13}$$

where $1 \le \frac{\alpha+2\beta}{\alpha+\beta} \le 2$. To simplify the formula above, we replace $\frac{\beta}{\alpha}$ with γ. Then the formula becomes $\frac{rem\_time}{backup\_time} > \frac{1+2\gamma}{1+\gamma}$. When a cluster is idle and has many free slots, the cost for speculative execution is less a concern, because it does not hurt other jobs' performance. On the other hand, when the cluster is busy and has many pending tasks of other jobs, the cost is an important consideration because backing up a task will delay other jobs' execution. We expect that γ varies with the load of the cluster: $\frac{1+2\gamma}{1+\gamma}$ gets its lowest value 1 ($\gamma = 0$) when the load of the cluster is low while reaches its highest value 2 ($\gamma = \infty$) when the load of the cluster is high. That is to say, γ should increase with the load of the cluster. Therefore, we set γ to the $load\_factor$ of the Hadoop cluster:

$$\gamma = load\_factor = \frac{number_{pending\_tasks}}{number_{free\_slots}}, \tag{14}$$

where $number_{pending\_tasks}$ is the number of pending tasks, and $number_{free\_slots}$ is the number of free slots in the cluster. When the cluster is idle and has many free slots, γ decreases to 0 indicating no cost for speculative execution and the backup condition becomes $rem\_time > backup\_time$. When the cluster is busy and has many pending tasks of other jobs, γ increases and $\frac{1+2\gamma}{1+\gamma}$ converges to 2 gradually. Then the backup condition becomes $rem\_time > 2 * backup\_time$. As a result, fewer tasks will be backed up. Therefore, using $load\_factor$ as γ fits our demand perfectly.

After iterating through all the running tasks, we will get a set of backup candidates. The candidate that has the longest remaining time will be backed up finally.

4.2 Selecting Proper Backup Worker Nodes

In order to achieve better performance, we should assign backup tasks to fast worker nodes. This requires an appropriate metric to measure the performance of worker nodes which varies a lot from time to time. For example, Microsoft witnessed that slow worker nodes vary over weeks due to changing data popularity [5]. As we have illustrated in Section 3.2.1, LATE and Hadoop-LATE do not evaluate the performance of worker nodes accurately. To tackle this problem, we use the moving average process bandwidth of data-local map tasks completed on a worker node to represent the node's performance.

In addition, we consider the data-locality of map tasks when making the backup decisions. As mentioned in Section 3.2.2, the process speed of data-local map tasks can be three times that of non-local map tasks. As a result, if we do not consider data-locality, backing up a map task may gain no benefit. For example, suppose the time of running a data-local, rack-local, and non-local map task is t, $2t$, and $3t$, respectively. Now we have a map task with $2t$ time left which needs to be backed up. If we back it up on a non-local worker node, it will take $3t$. In this case, the backup task will not complete before the original one. To solve this problem, we keep the process speed statistics of data-local, rack-local, and non-local map tasks for each worker node. For worker nodes that do not process any map task on a specific locality level, we use the average process speed of all worker nodes on this level as an estimate. When we select a map task to backup in MCP, we estimate how long it will take on the candidate worker node according to the locality level. We will assign the backup task to the node only if it is estimated to finish there faster.

4.3 Summary

In summary, a task will be backed up when it meets the following conditions:

- it has executed for a certain amount of time (i.e., the speculative lag),
- both the progress rate and the process bandwidth in the current phase of the task are sufficiently low,
- the profit of doing the backup outweighs that of not doing it,
- its estimated remaining time is longer than the predicted time to finish on a backup node,
- it has the longest remaining time among all the tasks satisfying the conditions above.

5 Evaluation

In this section, we evaluate the performance of our MCP strategy under both heterogeneous and homogeneous environments. We set up two scales of Hadoop clusters to show the scalability of MCP. To ensure the stable performance of MCP under various kinds of environments, we also analyze the sensitivity of the parameters used in MCP.

Our cluster consists of 30 physical machines. Each machine has dual-Processors (2.4 GHz Intel(R) Xeon(R) E5620 processor with 16 logic core), 24 GB of RAM and two 150 GB disks. These physical machines are organized in three racks connected by 1 Gbps Ethernet and managed by Open Stack Scalable Cloud Operating System [9] using KVM virtualization software [10]. Each virtual machine provided by OpenStack is in medium size with two virtual core, 4 GB RAM and 40 GB of disk space. We use such virtual machines to construct our experimental environment in two scales: small scale with 30 virtual machines on 15 physical machines, large scale with 100 virtual machines on 30 physical machines.

In our experiments, we keep the default configuration for HDFS, while configure each TaskTracker with four map slots and four reduce slots, as the capacity of our virtual machine is almost twice that of tiny virtual machine in EC2. We mainly use Hadoop Sort, WordCount, Grep, and Gridmix benchmark as our workloads because these benchmarks represent many different kinds of data-intensive and cpu-intensive jobs. For each test case, we run it at least five times. To reduce the influence of the variable environment, we demonstrate the performance of the average, the worst, and the best cases in our results. We compare MCP with Hadoop-LATE (Hadoop default configuration) and Hadoop-None to show our performance improvement and use the job execution time and the cluster throughput as our primary metrics. The improvement represented by the speedup of the job execution and the increase of the cluster throughput is calculated as follows:

$$Improvement = \frac{MCP - OtherStrategy}{OtherStrategy}. \tag{15}$$

Our results are summarized as follows:

- In a heterogenous cluster with about 30 nodes, MCP outperforms Hadoop-LATE on the job execution speed by 10, 19, 39, and 13 percent and cluster throughput by 5, 15, 38 and 15 percent when processing WordCount, Sort, Grep, and Gridmix benchmarks respectively.
- When data skew exists among the map or the reduce tasks in Sort jobs, MCP performs much better compared to Hadoop-LATE. The job execution speed and the cluster throughput are improved by 37 and 44 percent when map skew exists, and by 17 and 19 percent when reduce skew exists, respectively.
- When competing with other applications in Sort jobs, MCP handles slow tasks and slow nodes much better than Hadoop-LATE. MCP improves the job execution speed by 36 percent and the cluster throughput by 34 percent on average.
- The performance of MCP is demonstrated to be scalable. In a heterogenous cluster with about 101 nodes, MCP still improves Hadoop-LATE by 21 percent on the job execution speed and 16 percent on the cluster throughput when processing Sort jobs.

TABLE 2
Load of Each Host in Heterogeneous Environments

Load	Hosts	VMs
1VMs/host	3	3
2VMs/host	11	22
5VMs/host	1	5
Total	15	30

- In homogeneous environments with no straggler node, Hadoop-LATE may perform worse than Hadoop-None, while our MCP can still do better than Hadoop-None, indicating that MCP can also fit homogeneous environments very well.
- The overhead of MCP is very small: on average about 0.54 ms to handle a speculative request, in contrast to 0.74 ms for Hadoop-LATE.

5.1 Scheduling in Heterogeneous Environments

In heterogeneous environments, we run our experiments in a small scale cluster first. The distribution of virtual machines on physical servers in the small cluster is listed in Table 2. Under such an environment, the performance of the VMs can vary significantly due to the workloads from the co-hosted VMs. We run our experiments in two settings: without and with straggler nodes. Straggler nodes are created by running some I/O intensive applications on the physical machines. We show that MCP performs efficiently in both settings.

5.1.1 Working with Different Workloads

We run the WordCount benchmark first. The output of WordCount is the number of occurrence of words in the input files. The input files of WordCount in this experiment is about 8 GB with four map tasks per worker node. We submit three WordCount jobs one by one every 30 s. Fig. 7a shows the performance comparison of the three strategies. On average, MCP finishes jobs 10 percent faster than Hadoop-LATE and 10 percent faster than Hadoop-None. Moreover, MCP improves the throughput of the cluster by 5 percent compared with Hadoop-LATE and 6 percent compared with Hadoop-None. To understand why WordCount does not gain significant improvement, we take a closer look at the features of this workload. Due to the nature of most language files, the intermediate data and the final output of WordCount tend to be very small. Therefore, WordCount mostly executes in the map stage parsing text files, which is cpu-intensive. Once the map stage completes, the reduce stage is very short. Hence, the performance of WordCount is dominated by the execution of the map tasks. Since the average execution time for map tasks in WordCount is about 70 s (including the task setup time), the difference between running on a fast node and running on a slow node is small, leaving little room for improvement.

To explain why MCP outperforms Hadoop-LATE, we compare the precision, recall, and average find time of identifying straggler tasks in these two approaches. We run several WordCount jobs and gather their runtime statistics. After these jobs complete, we locate the occurrence of straggler tasks and calculate the precision, recall and average find time of identifying them. In this analysis, tasks that execute 1.5× slower than the average will be considered as stragglers. Table 3 gives the main result of this analysis. It shows that MCP identifies straggler tasks more accurately and promptly than Hadoop-LATE. In particular, MCP can improve the precision in identifying stragglers in reduce tasks by over 90 percent compared to Hadoop-LATE. To understand why Hadoop-LATE has such a low precision in identifying stragglers in reduce tasks, we categorize its

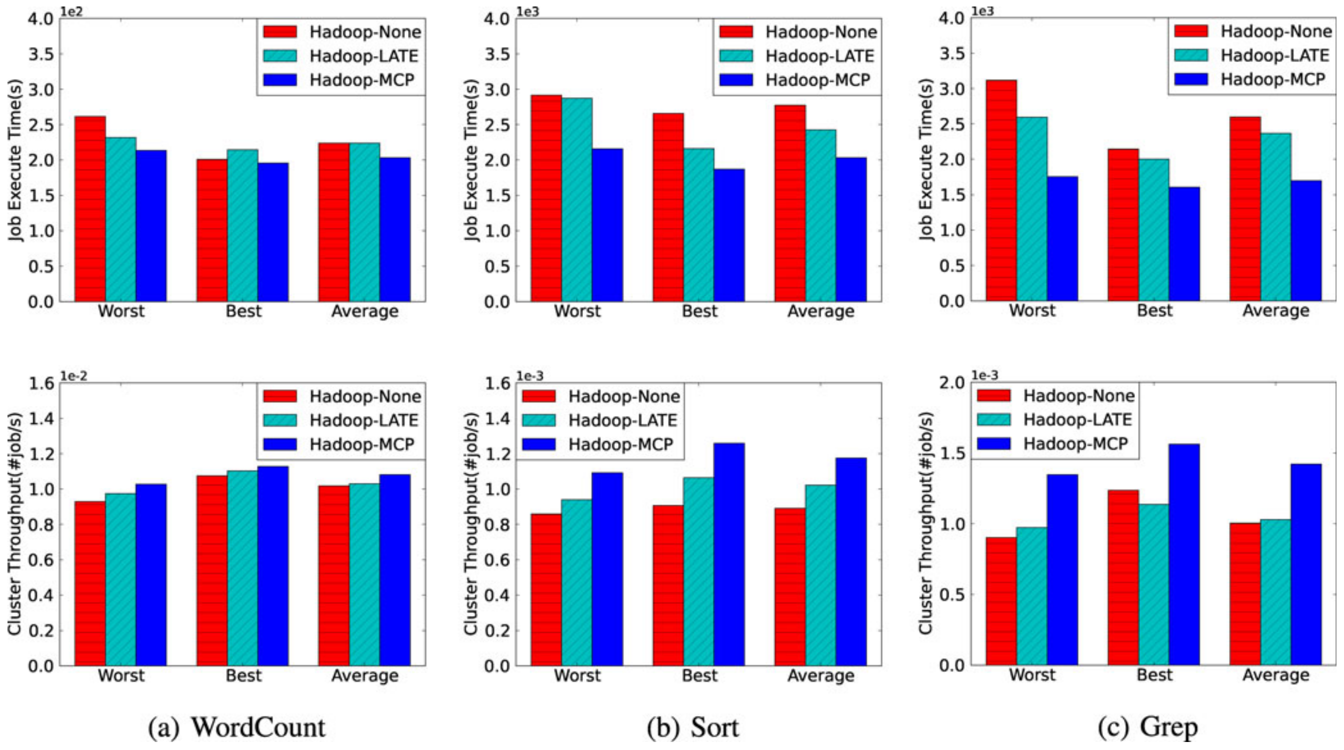


Fig. 7. Small scale cases without data skew.

TABLE 3
WordCount Analysis

Strategy	Precision		Recall		Average Find Time	
	map	reduce	map	reduce	map	reduce
Hadoop-LATE	37.6%	3%	100%	100%	70s	66s
Hadoop-MCP	45.2%	93.3%	87.1%	100%	56s	32s

mistakes according to our analysis in the previous section: due to data skew (described in Section 3.1.1), due to speed fluctuation across phases (described in Section 3.1.2), and due to normal slow down in the copy phase (described in Section 3.1.2). We find that they contribute to 19, 39, and 42 percent of the mistakes, respectively.

Sort is quite different from WordCount in that Sort writes a large amount of intermediate data and final output through the network and to the disks. Therefore, it is an I/O intensive application. We run sort jobs on a data set of 30 GB. Each sort job has 110 reduce tasks. We submit three sort jobs one by one every 150 s. Fig. 7b shows the job execution time and the cluster throughput of the three strategies. On average, MCP finishes jobs 19 percent faster than Hadoop-LATE and 37 percent faster than Hadoop-None. Moreover, MCP improves the throughput of the cluster by 15 percent over Hadoop-LATE and 32 percent over Hadoop-None. To explain why we can achieve a much bigger improvement for Sort than for WordCount, we give a further analysis. Since the reduce tasks in Sort jobs do much more work than its map tasks, the map stage makes up a very small part of the total execution time. As a result, in both the map and the reduce stages, MCP can have more opportunity to do effective speculative execution, leading to higher improvement.

The Grep benchmark searches a regular expression through input files and outputs the lines which contain strings matching the expression. Its behavior depends on how frequently the expression appears in the input file. When the expression appears frequently, it is I/O intensive just like Sort. When the expression appears rarely, it is CPU intensive just like WordCount. We launch Grep jobs to search the keyword 'the' in the data sets from Wikipedia, which is about 23 GB. We submit three Grep jobs one by one every 150 s. Fig. 7c shows the performance comparison of the three strategies. On average, MCP finishes jobs 39 percent faster than Hadoop-LATE and 53 percent faster than Hadoop-None. Moreover, MCP improves the throughput of the cluster by 38 percent over Hadoop-LATE and 42 percent over Hadoop-None. Since map tasks in Grep do more work (e.g., pattern matching) than in Sort, their average execution time is about 1.3× longer in Grep than in Sort. Therefore, speculative execution can be more effective for map tasks in Grep than in Sort, leading to bigger improvement.

To show the significance that each part of the MCP is, we give a deep analysis of the performance improvement achieved by each part of the MCP. We run three Grep jobs one by one the same as before. The results show that: i) only with the accurate straggler prediction, it finishes jobs 27 percent faster than Hadoop-LATE and improves the cluster throughput by 29 percent over Hadoop-LATE, ii) when adding cost performance model, it finishes jobs 31 percent faster and improves the cluster throughput by 32 percent, iii) when all parts are added (including the backup node selection), it finishes jobs 39 percent faster and improves the cluster throughput by 38 percent. From the result, we can find that accurate straggler prediction plays the most important role in the improvement of the MCP.

For a more complex and mixed workload, we run the Hadoop gridmix benchmark. This benchmark contains many kinds of jobs, such as streamSort, combiner, javaSort, monsterQuery, webdataScan, and webdataSort. We run it on a data set of 40 GB compressed and 10 GB uncompressed data and configure it with each kind of job running three small ones and three large ones. We set the number of reduce tasks to 20 for the small jobs and 110 for the large jobs. The job execution time of these jobs varies from 100 to 5,000 s. Fig. 8 shows the performance of the three strategies. On average, MCP finishes jobs 13 percent faster than Hadoop-LATE and 16 percent faster than Hadoop-None. Moreover, MCP improves the throughput of the cluster by 15 percent over Hadoop-LATE and 12 percent over Hadoop-None. From the result, we can also find that even though Hadoop-LATE can improve the average job execution time by launching more speculative execution, it decreases the cluster throughput due to too many wasted slots.

5.1.2 Handling Data Skew

To demonstrate that MCP can handle data skew effectively, we set up two environments with different kinds of data skew.

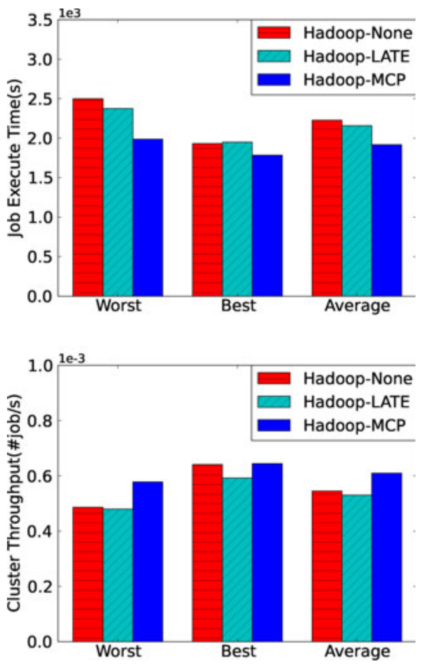


Fig. 8. Small scale Gridmix benchmark.

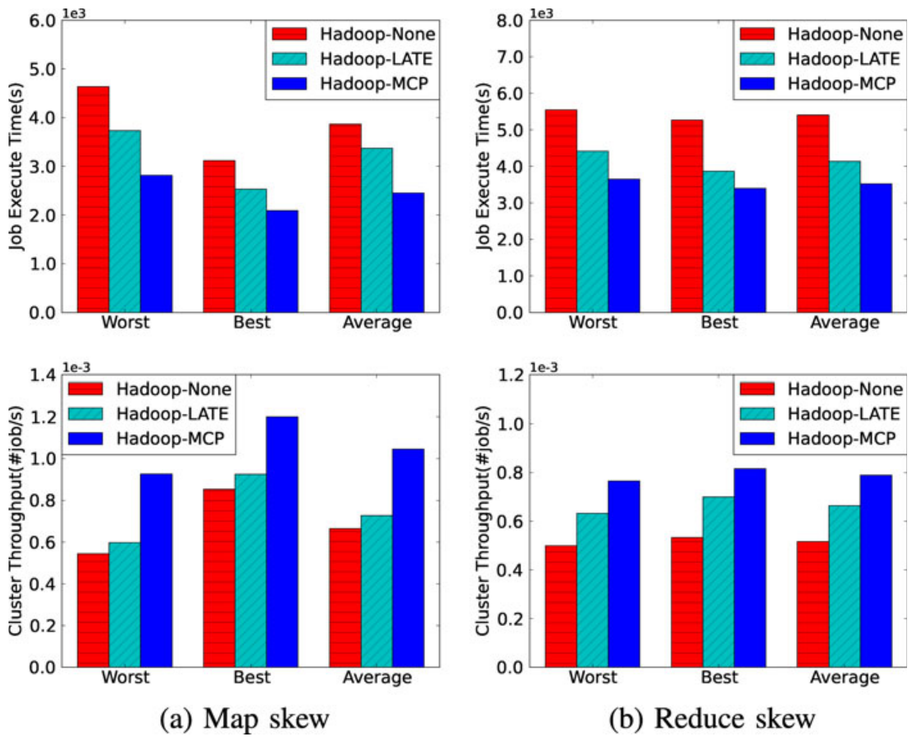


Fig. 9. Small scale cases with data skew.

First, we set up an environment that exhibits data skew among map tasks. We create a data set of 30 GB with the size of each input file about 100 MB. According to the split strategy in Hadoop, those input files will be divided into two parts: one is 64 MB and the other is 36 MB, which results in data skew among map tasks. We submit three sort jobs one by one every 150 s. Fig. 9a shows that MCP performs much better than Hadoop-LATE and Hadoop-None. On average MCP increases the job execution speed by 37 percent over Hadoop-LATE and 58 percent over Hadoop-None. Meanwhile, it improves the throughput of the cluster by 44 percent over Hadoop-LATE and 57 percent over Hadoop-None. MCP can achieve a much bigger improvement than Hadoop-LATE because Hadoop-LATE may conduct many unnecessary backups for the map tasks which occupies the precious slots for other jobs. As a result, the average delay of all jobs in Hadoop-LATE is much longer than that in MCP.

Reduce skew is likely to happen when the distribution of keys in the input data set is skewed and the map output is partitioned by some hash function. This kind of skew is also known as partition skew. Since many kinds of real world data set follow the Zipf distribution with σ parameter almost equals 1.0, such as word frequency in natural languages and website popularity, we create a data set of 30 GB which follows the Zipf distribution with σ equals 1.0 (σ is used to control the degree of the skew). We again run three sort jobs one by one every 150 s. Fig. 9b shows that on average MCP increases the job execution speed by 17 percent over Hadoop-LATE and by 53 percent over Hadoop-None. Meanwhile, it improves the throughput of the cluster by 19 percent over Hadoop-LATE and by 53 percent over Hadoop-None. MCP achieves less improvement over Hadoop-LATE for reduce skew than for map skew because unnecessary reduce backups do not affect the execution of map tasks from other jobs. It only delays the reduce tasks of other jobs. As we explained before, reduce tasks cannot enter the sort phase when there are still some map tasks running. Instead, they must wait for all map tasks to complete their outputs. Therefore, a small delay in launching reduce tasks will not affect the performance of other jobs significantly when those other jobs are still in the map stage.

5.1.3 Competing with Other Applications

To evaluate how MCP handles the competition of other applications, we run some I/O intensive processes (*dd* process which creates large files in a loop to write random data) on some of the physical machines to simulate the load of other applications. On straggler nodes, we start dd process at different time with various durations. Just like previous tests, we submit three sort jobs one by one every 150 s. Fig. 10 shows that on average, MCP can run 36 percent faster than Hadoop-LATE and 46 percent than Hadoop-None. MCP can also increase the throughput of the cluster by 34 percent over Hadoop-LATE and 41 percent over Hadoop-None.

5.1.4 Scalability

To evaluate the scalability of MCP, we run our experiment in a large cluster. The distribution of virtual machines on physical machines is listed in Table 4. We run Sort jobs on a data set of 90 GB. Each sort job has 340 reduce tasks. Each test case has three Sort jobs which are submitted one by one. Fig. 11 shows the job execution time and the cluster throughput. On average, MCP finishes jobs 21 percent faster than Hadoop-LATE and 26 percent faster than Hadoop-None. Moreover, MCP can improve the throughput of the cluster by 16 percent over Hadoop-LATE and by 22 percent

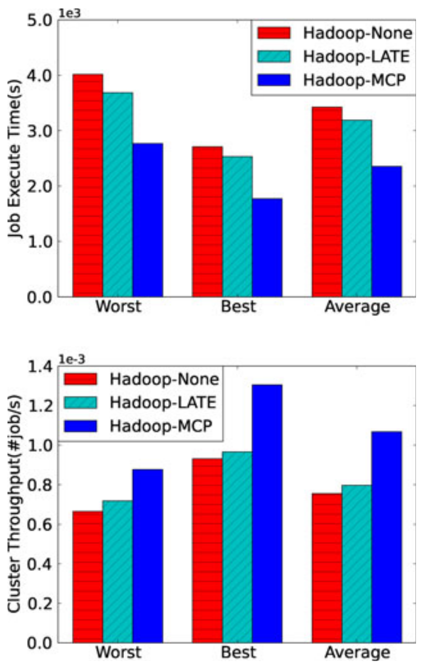


Fig. 10. Competing with other applications case.

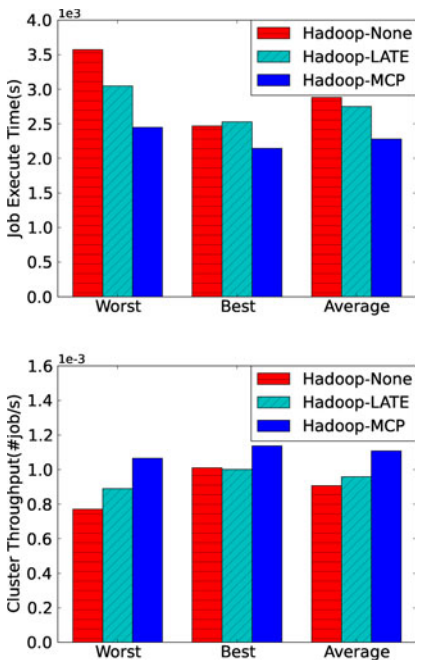


Fig. 11. Large scale case.

over Hadoop-None. We notice that in the 100 node cluster, the improvement of MCP over Hadoop-None is smaller than that in the 30 node cluster. This is because we construct the 30 node cluster on 15 physical machines while the 101 node cluster on 30 physical machines as shown in Tables 2 and 4. The 101 node cluster is obviously less heterogeneous than the 30 node cluster, leading to smaller probability of stragglers. Overall, the scalability of MCP is good due to its low scheduling cost.

5.2 Scheduling in Homogeneous Environments

In homogeneous environments, we evaluate MCP, Hadoop-LATE, and Hadoop-None in a small scale cluster with each host running two VMs. In this environment, the performance of the VMs is almost the same and there are not straggler nodes. We run sort jobs on a data set of 10 GB with each input file about 2 GB. We submit three sort jobs one by one every 150 s with 90, 70, and 50 reduce tasks, respectively. Fig. 12 shows the execution time of the jobs and the throughput of the cluster. In this test case, MCP finishes jobs 6 percent faster than Hadoop-LATE and 2 percent faster than Hadoop-None on average. Log analysis shows that Hadoop-LATE behaves worse than Hadoop-None due to too many unnecessary reduce task backups, while MCP achieves better results than Hadoop-LATE because we improve reduce backup precision by 40 percent. The reason why MCP can perform better than Hadoop-None is that we can achieve better data locality for map tasks through speculative execution.

TABLE 4
Load of Each Host in Scalability Environments

Load	Hosts	VMs
3VMs/host	27	81
5VMs/host	4	20
Total	31	101

5.3 Parameters Analysis

In Section 4.1.1, we propose to use EWMA to predict the process speed of tasks in order to find slow tasks or slow nodes in time. EWMA has a parameter α which reflects a tradeoff between stability and responsiveness. In this section, we evaluate the performance of various α values to see the variance trends in our small scale cluster. We run this experiment in heterogeneous environments with and without straggler nodes. We submit three sort jobs one by one on a data set of 30 GB. Each job has 20 reduce tasks. We run experiments with six α values from 0.1 to 0.6, repeating each one three times. Fig. 13 shows that we achieve best performance in

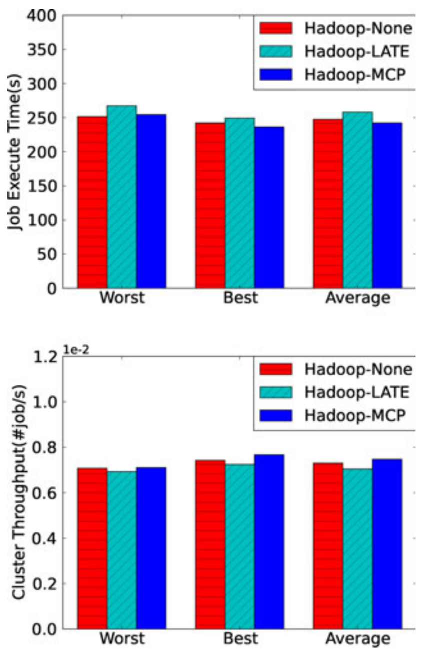


Fig. 12. Homogeneous environments case.

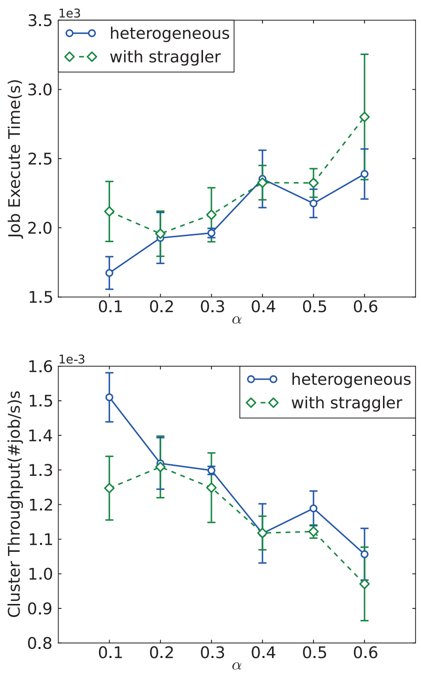


Fig. 13. Performance with different value of α.

heterogenous environment with α being 0.2. This is because when α is too low, it cannot find slow tasks and slow nodes in time while when it is high, it will cause too many oscillations which may lead to misjudgments.

5.4 Scheduling Cost

The master node is a potential bottleneck in a MapReduce cluster since it is responsible for resource allocation of all worker nodes and job scheduling for all jobs. Speculative execution will keep more data statistics and add additional computation to the master, which may affect its performance. We measure the average time that MCP and Hadoop-LATE spend on speculative scheduling in a job with about 350 map tasks and 110 reduce tasks. Log analysis shows that on average MCP spends about 0.54 ms on handling a speculative request, while Hadoop-LATE spends about 0.74 ms. This is because the time complexity of MCP is O(n), while the time complexity of LATE is O(nlogn) due to the sorting in its calculation. The result demonstrates that the scheduling cost of MCP is low.

6 Related Work

Following on Google's MapReduce paradigm [1], many works have been done to improve the performance of mapreduce.

SCOPE [11], Pig [12], and DryadLINQ [13] provide easy-to-use language (such as SQL-like language) to simplify the MapReduce programming. Work [14] and [15] optimize the MapReduce programming by selecting parameters automatically. Ganesha [16] offers a black-box method for diagnosis. MapReduce Online [17], MapReduce Merge [18] and Airavat [19] supply new features for MapReduce, such as pipelining, online aggregation, multi-jobs joining, security and privacy. All of them concern on how to improve the functionality of MapReduce to make it more convenient and available.

Towards performance improvement, there are Scarlett [20] and BlobSeer [21] optimizing the underlying data storage, [22] considering energy saving, [23] offering fast recovery, [24] concerning on the dynamic customizability, Fair Scheduling [2], Quincy [25] and Delay Scheduling [26] focusing on the task scheduling optimization to consider fairness and data locality, MapReduce in Google [1], Dryad in Microsoft [3], LATE [4], Hadoop [2] and Mantri [5] working on the speculative execution. Our work here also addresses on the issue of optimizing speculative exection to improve MapReduce performance.

Compared with current strategies [1], [3], [4], [2] and [5], MCP pays attention to the cost performance of cluster resources, and deals well with such scenarios as data skew, tasks that start asynchronously, improper configuration of phase percentage and abrupt resource competitions. MCP boosts the precision of speculative execution and improves not only the job execution time but also the cluster throughput.

Recently, the Hadoop community is developing a new version of Hadoop – Hadoop 2.0 [27]. In this version, the JobTracker in Hadoop 1.0 is replaced by the ResourceManager and per-application ApplicationMaster. The ResourceManager is responsible for computing resource allocation and the per-application ApplicationMaster is responsible for task scheduling and coordination. MCP focuses on the task scheduling in MapReduce Job, therefore it can be easily integrated into the ApplicationMaster of MapReduce and achieve performance improvement.

Our work also relates to the speculative execution for high performance computing (HPC) [28], distributed computing [29] and multiple processors [30], [31]. However, our work is different from them for the deploy environment is uncertain and varies over time and there are little communications between tasks. In multiple processors environment, it requires to arrange task assignments in advance. In MapReduce, we can schedule task assignments at run time according to the current condition of the environment. Bernardin et al. [29] uses mean speed, normalized mean, standard deviation and the ratio of waiting tasks to pending tasks of each job to identify slow tasks. We have demonstrated that most of these methods are not suitable for the uncertain and changeable environment. Star-MPI [28] dynamically adjusts the placement of tasks according to the performance observed over time. We only consider speculative execution at the end of a stage.

7 Conclusions

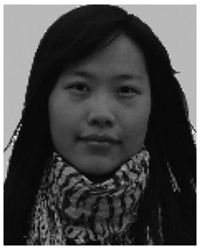
In this paper, we provide an analysis of the pitfalls of current speculative execution strategies in MapReduce. We present scenarios which affect the performance of those strategies: data skew, tasks that start asynchronously, improper configuration of phase percentage and abrupt resource competitions. Based on the analysis, we develop a new speculative execution strategy called MCP to handle these scenarios. MCP takes the cost performance of cluster computing resources into account, aiming at not only decreasing the job execution time but also improving the cluster throughput. Our experiments show that: MCP can achieve up to 39 percent improvements over Hadoop-LATE; MCP fits well in both heterogeneous and homogeneous environments; MCP can handle the data skew case well; MCP is quite scalable, which performs very well in both small clusters and large clusters; MCP has less overhead than Hadoop-LATE and can be easily implemented into new versions of Hadoop.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their invaluable feedback. This work was supported by the National Natural Science Foundation of China (Grant No. 61170056) and the National High Technology Research and Development Program ("863" Program) of China (Grant No. 2013AA013203). The contact author is Zhen Xiao.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Comm. ACM*, vol. 51, pp. 107-113, Jan. 2008.
[2] Apache hadoop, http://hadoop.apache.org/ 2013.
[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *Proc. Second ACM SIGOPS/EuroSys European Conf. Computer Systems (EuroSys '07)*, 2007.
[4] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, and I. Stoica, "Improving Mapreduce Performance in Heterogeneous Environments," *Proc. Eighth USENIX Conf. Operating Systems Design and Implementation, (OSDI '08)*, 2008.
[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-Reduce Clusters Using Mantri," *Proc. Ninth USENIX Conf. Operating Systems Design and Implementation, (OSDI '10)*, 2010.
[6] "Amazon Elastic Compute Cloud (EC2)," http://aws.amazon.com/ec2/, 2013.
[7] Y. Kwon, M. Balazinska, and B. Howe, "A Study of Skew in Mapreduce Applications," *Proc. Fifth Open Cirrus Summit*, 2011.
[8] P.H. Ellaway, "Cumulative Sum Technique and Its Application to the Analysis of Peristimulus Time Histograms," *Electroencephalography and Clinical Neurophysiology*, vol. 45, no. 2, pp. 302-304, 1978.
[9] "Open Stack Cloud Operating System," http://www.openstack.org/, 2013.
[10] K. Avi, K. Yaniv, L. Dor, L. Uri, and L. Anthony, "Kvm : The Linux Virtual Machine Monitor," *Proc. Linux Symp.*, 2007.
[11] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: Easy and Efficient Parallel Processing of Massive Data Sets," *Proc. VLDB Endowment*, vol. 1, pp. 1265-1276, Aug. 2008.
[12] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08)*, 2008.
[13] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P.K. Gunda, and J. Currey, "Dryadlinq: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," *Proc. Eighth USENIX Conf. Operating Systems Design and Implementation, (OSDI '08)*, 2008.
[14] H. Herodotou and S. Babu, "Profiling, Whatif Analysis, and Costbased Optimization of Mapreduce Programs," *Proc. VLDB Endow.*, vol. 3, pp. 1111-1122, Sept. 2011.
[15] S. Babu, "Towards Automatic Optimization of Mapreduce Programs," *Proc. First ACM Symp. Cloud Computing (SoCC '10)*, 2010.
[16] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Ganesha: Blackbox Diagnosis of Mapreduce Systems," *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 37, pp. 8-13, Jan. 2010.
[17] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce Online," *Proc. Seventh USENIX Conf. Networked Systems Design and Implementation (NSDI '10)*, 2010.
[18] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D.S. Parker, "Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters," *Proc. ACM SIGMOD Int'l Conf. Management of Data, (SIGMOD '07)*, 2007.
[19] I. Roy, S.T.V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and Privacy for Mapreduce," *Proc. Seventh USENIX Conf. Networked Systems Design and Implementation (NSDI '10)*, 2010.
[20] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters," *Proc. Sixth Conf. Computer Systems (EuroSys '11)*, 2011.
[21] B. Nicolae, D. Moise, G. Antoniu, L. Bouge, and M. Dorier, "Blobseer: Bringing High Throughput under Heavy Concurrency to Hadoop Map-Reduce Applications," *Proc. IEEE Int'l Symp. Parallel Distributed Processing (IPDPS)*, Apr. 2010.
[22] J. Leverich and C. Kozyrakis, "On the Energy (In)Efficiency of Hadoop Clusters," *ACM SIGOPS Operating Systems Rev.*, vol. 44, pp. 61-65, Mar. 2010.
[23] J.-A. Quiane-Ruiz, C. Pinkel, J. Schad, and J. Dittrich, "Rafting Mapreduce: Fast Recovery on the Raft," *Proc. IEEE 27th Int'l Conf. Data Eng. (ICDE)*, Apr. 2011.
[24] T. Sandholm and K. Lai, "Mapreduce Optimization Using Regulated Dynamic Prioritization," *Proc. 11th Int'l Joint Conf. Measurement and Modeling of Computer Systems, (SIGMETRICS '09)*, 2009.
[25] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles (SOSP '09)*, 2009.
[26] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," *Proc. Fifth European Conf. Computer Systems (EuroSys '10)*, 2010.
[27] "Apache hadoop 2.0," http://hadoop.apache.org/docs/r2.0.0-alpha/, 2013.
[28] A. Faraj, X. Yuan, and D. Lowenthal, "Star-Mpi: Self Tuned Adaptive Routines for Mpi Collective Operations," *Proc. 20th Ann. Int'l Conf. Supercomputing (ICS '06)*, 2006.
[29] J. Bernardin, P. Lee, and J. Lewis, "Using Execution Statistics to Select Tasks for Redundant Assignment in a Distributed Computing Platform," Patent 7 093 004, Aug. 2006.
[30] B. Ucar, C. Aykanat, K. Kaya, and M. Ikinci, "Task Assignment in Heterogeneous Computing Systems," *J. Parallel and Distributed Computing*, vol. 66, pp. 32-46, Jan. 2006.
[31] S. Manoharan, "Effect of Task Duplication on the Assignment of Dependency Graphs," *Parallel Computing*, vol. 27, pp. 257-268, Feb. 2001.



Qi Chen received the bachelor's degree from Peking University in 2010. She is currently a doctoral student at Peking University. Her current research focuses on cloud computing and parallel computing.

Cheng Liu received the bachelor's degree in 2009 and the master's degree in 2012 from Peking University. His current research focuses on the CDN network and cloud computing.

Zhen Xiao received the PhD degree from Cornell University in January 2001. After that he joined as a senior technical staff member at AT&T Labs - New Jersey and then a research staff member at IBM T.J. Watson Research Center. He is a professor in the Department of Computer Science at Peking University. His research interests include cloud computing, virtualization, and various distributed systems issues. He is a senior member of the ACM and the IEEE.