This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

Self-Adjusting Slot Configurations for Homogeneous and Heterogeneous Hadoop Clusters

Yi Yao[1] Jiayin Wang[2] Bo Sheng[2] Chiu C. Tan[3] Ningfang Mi[1]
1.Northeastern University, Boston, MA, USA
Email: {yyao, ningfang}@ece.neu.edu
2.University of Massachusetts Boston, Boston, MA, USA
Email: {jane, shengbo}@cs.umb.edu
3. Temple University, Philadelphia, PA, USA
Email: cctan@temple.edu

Abstract—The MapReduce framework and its open source implementation Hadoop have become the defacto platform for scalable analysis on large data sets in recent years. One of the primary concerns in Hadoop is how to minimize the completion length (i.e., makespan) of a set of MapReduce jobs. The current Hadoop only allows static slot configuration, i.e., fixed numbers of map slots and reduce slots throughout the lifetime of a cluster. However, we found that such a static configuration may lead to low system resource utilizations as well as long completion length. Motivated by this, we propose simple yet effective schemes which use *slot ratio* between map and reduce tasks as a tunable knob for reducing the makespan of a given set. By leveraging the workload information of recently completed jobs, our schemes dynamically allocates resources (or slots) to map and reduce tasks. We implemented the presented schemes in Hadoop V0.20.2 and evaluated them with representative MapReduce benchmarks at Amazon EC2. The experimental results demonstrate the effectiveness and robustness of our schemes under both simple workloads and more complex mixed workloads.

Index Terms—MapReduce jobs, Hadoop scheduling, reduced makespan, slot configuration

1 Introduction

MapReduce [1] has become the leading paradigm in recent years for parallel big data processing. Its open source implementation Apache Hadoop [2] has also emerged as a popular platform for daily data processing and information analysis. With the rise of cloud computing, MapReduce is no longer just for internal data process in big companies. It is now convenient for a regular user to launch a MapReduce cluster on the cloud, e.g., AWS MapReduce, for data-intensive applications. When more and more applications are adopting the MapReduce framework, how to improve the performance of a MapReduce cluster becomes a focus of research and development. Both academia and industry have put tremendous efforts on job scheduling, resource management, and Hadoop applications [3]–[11]. As a complex system, Hadoop is configured with a large set of system parameters. While it provides the flexibility to customize the cluster for different applications, it is challenging for users to understand and set the optimal values for those parameters. In this paper, we aim to develop algorithms for adjusting a basic system parameter with the goal to improve the performance (i.e., reduce the makespan) of a batch of MapReduce jobs.

This work was partially supported by the NSF grant (CNS-1251129), the AFOSR grant (FA9550-14-1-0160), and the AWS in Education Research Grant.

A classic Hadoop cluster includes a single master node and multiple slave nodes. The master node runs the *JobTracker* routine which is responsible for scheduling jobs and coordinating the execution of tasks of each job. Each slave node runs the *TaskTracker* daemon for hosting the execution of MapReduce jobs. The concept of "slot" is used to indicate the capacity of accommodating tasks on each node. In a Hadoop system, a slot is assigned as a map slot or a reduce slot serving map tasks or reduce tasks, respectively. At any given time, only one task can be running per slot. The number of available slots per node indeed provides the maximum degree of parallelization in Hadoop. Our experiments have shown that the slot configuration has a significant impact on system performance. The Hadoop framework, however, uses fixed numbers of map slots and reduce slots at each node as the default setting throughout the lifetime of a cluster. The values in this fixed configuration are usually heuristic numbers without considering job characteristics. Therefore, this static setting is not well optimized and may hinder the performance improvement of the entire cluster.

In this work, we propose and implement a new mechanism to dynamically allocate slots for map and reduce tasks. The primary goal of the new mechanism is to improve the completion time (i.e., the makespan) of a batch of MapReduce jobs while retain the simplicity in implementation and management of the slot-based

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

Hadoop design. The key idea of this new mechanism, named TuMM, is to automate the slot assignment ratio between map and reduce tasks in a cluster as a tunable knob for reducing the makespan of MapReduce jobs. The Workload Monitor (WM) and the Slot Assigner (SA) are the two major components introduced by TuMM. The WM that resides in the *JobTracker* periodically collects the execution time information of recently finished tasks and estimates the present map and reduce workloads in the cluster. The SA module takes the estimation to decide and adjust the slot ratio between map and reduce tasks for each slave node. With TuMM, the map and reduce phases of jobs could be better pipelined under priority based schedulers, and thus the makespan is reduced. We further investigate the dynamic slot assignments in heterogeneous environments, and propose a new version of TuMM, named H_TuMM, which sets the slot configurations for each individual node to reduce the makespan of a batch of jobs. We implemented the presented schemes in Hadoop V0.20.2 and evaluated them with representative MapReduce benchmarks at Amazon EC2. The experimental results demonstrate the effectiveness and robustness of our schemes under both simple workloads and more complex mixed workloads.

The rest of the paper is organized as follows. We explain the motivation of our work through some experimental examples in Section 2. We formulate the problem and derive the optimal setting for static slot configuration in a homogeneous cluster in Section 3. The design details of the dynamic mechanism for homogeneous clusters and heterogeneous clusters are presented in Section 4 and Section 5. Section 6 provides the experimental evaluation of the proposed schemes. Section 7 describes the related work of this work. We conclude in Section 8.

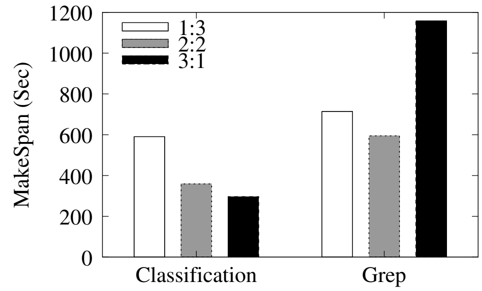


Fig. 1. The makespans of jobs under case 1 (i.e., *Classification*) and case 2 (i.e., *Grep*). The map and reduce slot ratios on each slave node are set to 1:3, 2:2, and 3:1.

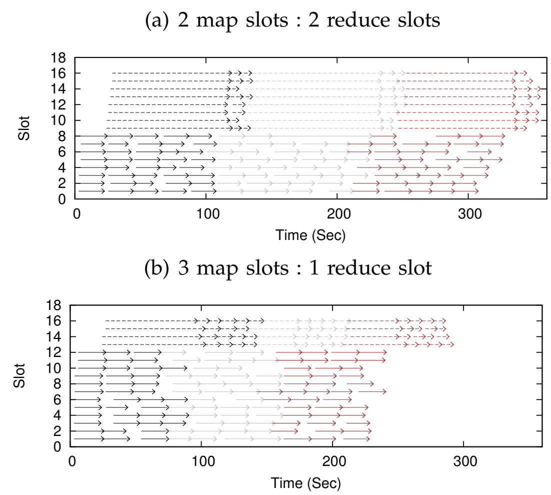


Fig. 2. Task execution times of three *Classification* jobs under different static slot configurations, where each node has (a) 2 map slots and 2 reduce slots, and (b) 3 map slots and 1 reduce slot. Each arrowed line represents the execution of one task, and the solid (resp. dashed) ones represent map (resp. reduce) tasks. In addition, we use three different colors to discriminate the three jobs.

2 Motivation

Currently, the Hadoop framework uses fixed numbers of map slots and reduce slots on each node throughout the lifetime of a cluster. However, such a fixed slot configuration may lead to low resource utilizations and poor performance especially when the system is processing varying workloads. We here use two simple cases to exemplify this deficiency. In each case, three jobs are submitted to a Hadoop cluster with 4 slave nodes and each slave node has 4 available slots. Details of the experimental setup are introduced in Section 6. To illustrate the impact of resource assignments, we also consider different static settings for map and reduce slots on a slave node. For example, when the slot ratio is equal to 1:3, we have 1 map slot and 3 reduce slots available per node. We then measure the overall lengths (i.e., makespans) for processing a batch of jobs, which are shown in Fig. 1.

Case 1: We first submit three *Classification* jobs to process a 10 GB movie rating data set. We observe that makespan is varying under different slot ratio settings and the best performance (i.e., shortest makespan) is achieved when each slave node has three map slots and one reduce slot, see the left column of Fig. 1.

To interpret this effect, we further plot the execution times of each task in Fig. 2. Clearly, *Classification* is a map-intensive application; for example, when we equally distribute resources (or slots) between map and reduce tasks, i.e., with the slot ratio of 2:2, the length of a map phase is longer than that of a reduce phase, see Fig. 2(a). It follows that each job's reduce phase (including shuffle operations and reduce operations) overlaps with its map phase for a long period. However, as the reduce operations can only start after the end of the map phase, the occupied reduce slots stay in shuffle for a long period, mainly waiting for the outputs from the map tasks. Consequently, system resources are underutilized.

For example, we tracked the CPU utilizations of each task in a slave node every 5 seconds and Table 1 shows part of the records in one of such overlapping periods. At each moment, the overall CPU utilization (i.e., the summation of CPU utilizations of the four tasks) is much

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

less than 400%, for a node with 4 cores. We then notice that when we assign more slots to map tasks, e.g., with the slot ratio of 3:1, each job experiences a shorter map phase and most of its reduce phase overlaps with the following job's map phase, see Fig. 2(b). The average CPU utilization is also increased by 20% compared to those under the the slot ratio of 2:2. It implies that for map-intensive jobs like *Classification*, one should assign more resources (slots) to map tasks in order to improve the performance in terms of makespan.

TABLE 1
Real time CPU utilizations of each task on a slave node in the overlapping time period of a job's map and reduce phases. The slot ratio per node is 2:2.

	ProcessId/TaskType			
Time(sec)	3522/map	3564/map	3438/reduce	3397/reduce
1	147%	109%	26%	0%
6	103%	93%	0%	4%
11	93%	99%	8%	0%
16	100%	100%	0%	0%
21	97%	103%	0%	0%

Case 2: In this case, we turn to consider reduce-intensive applications by submitting three *Grep* jobs to scan the 10 GB movie rating data. Similar to case 1, we also investigate three static slot configurations.

First, we observe that each job takes a longer time to process its reduce phase than its map phase when we have 2 map and 2 reduce slots per node, see Fig. 3(a). Based on the observation in case 1, we expect a reduced makespan when assigning more slots to reduce tasks, e.g., with the slot ratio of 1:3. However, the experimental results show that the makespan under this slot ratio setting (1:3) becomes even longer than that under the setting of 2:2, see the right column of Fig. 1. We then look closely at the corresponding task execution times, see Fig. 3(b). We find that the reduce tasks indeed have excess slots such that the reduce phase of each job starts too early and wastes time waiting for the output from its map phase. In fact, a good slot ratio should be set between 2:2 and 1:3 to enable each job's reduce phase to fully overlap with the following job's map phase rather than its own map phase.

In summary, in order to reduce the makespan of a batch of jobs, more resources (or slots) should be assigned to map (resp. reduce) tasks if we have map (resp. reduce) intensive jobs. On the other hand, a simple adjustment in such slot configurations is not enough. An effective approach should tune the slot assignments such that the execution times of map and reduce phases can be well balanced and the makespan of a given set can be reduced to the end.

3 System Model and Static Slot Configuration

In this section, we present a homogeneous Hadoop system model we considered and formulate the problem. In addition, we analyze the default static slot configuration in Hadoop and present an algorithm to derive the best configuration.

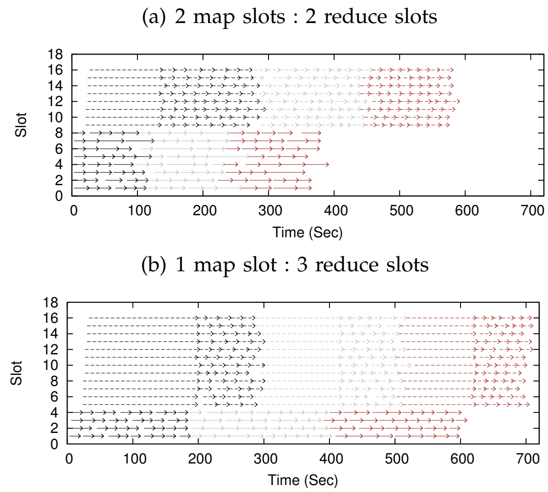


Fig. 3. Task execution times of a batch of *Grep* jobs under different static slot configurations, where each node has (a) 2 map slots and 2 reduce slots, and (b) 1 map slot and 3 reduce slots.

3.1 Problem Formulation

In our problem setting, we consider that a Hadoop cluster consisting of k nodes has received a batch of n jobs for processing. We use J to represent the set of jobs, $J = \{j_1, j_2, \ldots, j_n\}$. Each job j_i is configured with $n_m(i)$ map tasks and $n_r(i)$ reduce tasks. Let $st(i)$ and $ft(i)$ indicate the start time and the finish time of job j_i, respectively. The total slots number in the Hadoop cluster is equal to S, and let s_m and s_r be the number of map slots and reduce slots, respectively. We then have $S = s_m + s_r$. In this paper, our objective is to develop an algorithm to dynamically tune the parameters of s_m and s_r, given a fixed value of S, in order to minimize the makespan of the given batch of jobs, i.e., $minimize\{max\{ft(i), \forall i \in [1, n]\}\}$. Table 2 lists important notations that have been used in the rest of this paper.

TABLE 2
Notations used in this paper.

S, s_m, s_r	number of total/map/reduce slots of cluster;
$n_m(i), n_r(i)$	number of map/reduce tasks of job i;
$n'_m(i), n'_r(i)$	number of unscheduled map/reduce tasks of job i;
$\bar{t}_m(i), \bar{t}_r(i)$	average map/reduce task execution time of job i;
$w_m(i), w_r(i)$	total execution time of map/reduce tasks of job i;
$w'_m(i), w'_r(i)$	execution time of unscheduled tasks of job i;
$st(i), ft(i)$	start/finish time of job i;
rt_m, rt_r	number of currently running map/reduce tasks;

In a Hadoop system, makespan of multiple jobs also depends on the job scheduling algorithm which is coupled with our solution of allocating the map and reduce slots on each node. In this paper, we only consider

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

using the default FIFO (First-In-First-Out) job scheduler because of the following two reasons. First, given n jobs waiting for service, the performance of FIFO is no worse than other schedulers in terms of makespan. In the example of "Case 2" mentioned in Section 2, the makespan under FIFO is 594 sec while Fair, an alternative scheduler in Hadoop, consumes 772 sec to finish jobs. Second, using FIFO simplifies the performance analysis because generally speaking, there are fewer concurrently running jobs at any time. Usually two jobs, with one in map phase and the other in reduce phase.

Furthermore, we use execution time to represent the workload of each job. As a MapReduce job is composed of two phases, we define $w_m(i)$ and $w_r(i)$ as the workload of map phase and reduce phase in job j_i, respectively. We have developed solutions with and without the prior knowledge of the workload and we will discuss how to obtain this information later.

3.2 Static Slot Configuration with Workload Information

First, we consider the scenario that the workload of a job is available and present the algorithm for static slot configuration which is default in a Hadoop system. Basically, the Hadoop cluster preset the values of s_m and s_r under the constraint of $S = s_m + s_r$ before executing the batch of jobs, and the slot assignment will not be changed during the entire process. We have developed the following Algorithm 1 to derive the optimal values of s_m and s_r.

Our algorithm and analysis are based on an observation that the time needed to finish the workload of map or reduce phase is inversely proportional to the number of slots assigned to the phase in a homogeneous Hadoop cluster. Given s_m and s_r, the map (resp. reduce) phase of j_i needs $\frac{n_m(i)}{s_m}$ (resp. $\frac{n_r(i)}{s_r}$) rounds to finish. In each round, s_m map tasks or s_r reduce tasks are processed in parallel and the time consumed is equal to the execution time of one map or one reduce task. Let $\bar{t}_m(i)$ and $\bar{t}_r(i)$ be the average execution time for a map task and a reduce task, respectively. The workloads of map and reduce phases are defined as

$$w_m(i) = n_m(i) \cdot \bar{t}_m(i), w_r(i) = n_r(i) \cdot \bar{t}_r(i). \quad (1)$$

Algorithm 1 can derive the best static setting of s_m and s_r given the workload information. The outer loop (lines 1–10) in the algorithm enumerates the value of s_m and s_r (i.e., $S - s_m$). For each setting of s_m and s_r, the algorithm first calculates the workload ($w_m(i)$ and $w_r(i)$) for each job j_i in lines 3–5. The second inner loop (lines 6–8) is to calculate the finish time of each job. Under the FIFO policy, there are at most two concurrently running jobs in the Hadoop cluster. Each job's map or reduce phase cannot start before the precedent job's map or reduce phase is finished. More specifically, the start time of map tasks of job j_i, i.e., $st(i)$, is the finish time of j_{i-1}'s map phase, i.e., $st(i) = st(i-1) + \frac{w_m(i-1)}{s_m}$. Additionally, the start time of j_i's reduce phase should be no earlier than both the finish time of j_i's map phase and the finish time of j_{i-1}'s reduce phase. Therefore, the finish time of j_i is $ft(i) = \max(st(i) + \frac{w_m(i)}{s_m}, ft(i-1)) + \frac{w_r(i)}{s_r}$. Finally, the variables Opt_SM and Opt_MS keep track of the optimal value of s_m and the corresponding makespan (lines 9–10), and the algorithm returns Opt_SM and $S - Opt\_SM$ as the values for s_m and s_r at the end. The time complexity of the algorithm is $O(S \cdot n)$.

Algorithm 1 Static Slot Configuration

1: **for** $s_m = 1$ to S **do**
2:     $s_r = S - s_m$
3:     **for** $i = 1$ to n **do**
4:         $w_m(i) = n_m(i) \cdot \bar{t}_m(i)$
5:         $w_r(i) = n_r(i) \cdot \bar{t}_r(i)$
6:     **for** $i = 1$ to n **do**
7:         $st(i) = st(i-1) + \frac{w_m(i-1)}{s_m}$
8:         $ft(i) = \max(st(i) + \frac{w_m(i)}{s_m}, ft(i-1)) + \frac{w_r(i)}{s_r}$
9:     **if** $ft(n) < Opt\_MS$ **then**
10:         $Opt\_MS = ft(n)$; $Opt\_SM = s_m$
11: return Opt_SM and $S - Opt\_SM$

4 Dynamic Slot Configuration Under Homogeneous Environments

As discussed in Section 2, the default Hadoop cluster uses static slot configuration and does not perform well for varying workloads. The inappropriate setting of s_m and s_r may lead to extra overhead because of the following two cases:
(1) if job j_i's map phase is completed later than job j_{i-1}'s reduce phase, then the reduce slots will be idle for the interval period of $(st(i) + w_m(i)) - ft(i-1)$, see Fig. 4(a);
(2) if job j_i's map phase is completed earlier than the job j_{i-1}'s reduce phase, then j_i's reduce tasks have to wait for a period of $ft(i-1) - (st(i) + w_m(i))$ until reduce slots are released by j_{i-1}, see Fig. 4(b).

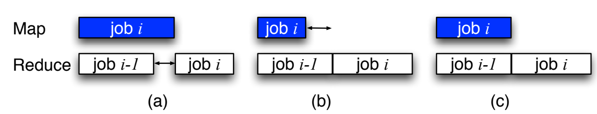


Fig. 4. Illustration of aligning the map and reduce phases. (a) and (b) are the two undesired cases mentioned above, and our goal is to achieve (c).

In this section, we present our solutions that dynamically allocate the slots to map and reduce tasks during the execution of jobs. The architecture of our design is shown in Fig. 5. In dynamic slot configuration, when one slot becomes available upon the completion of a map or reduce task, the Hadoop system will reassign a map or reduce task to the slot based on the current optimal values of s_m and s_r. There are totally $\sum_{i \in [1,n]}(n_m(i) + n_r(i))$ tasks and at the end of each task, Hadoop needs to decide the role of the available slot (either a map slot or a reduce slot). In this setting,

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

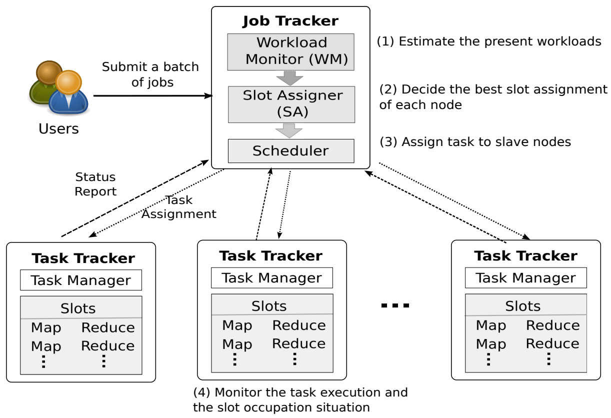


Fig. 5. The architecture overview of our design. The shade rectangles indicate our new/modified components in Hadoop.

therefore, we cannot enumerate all the possible values of s_m and s_r (i.e., $2^{\sum_i (n_m(i)+n_r(i))}$ combinations) as in Algorithm 1. Instead, we modify our objective in the dynamic slot configuration as there is no closed-form expression of the makespan.

Our goal now is, for the two concurrently running jobs (one in map phase and the other in reduce phase), to minimize the completion time of these two phases. Our intuition is to eliminate the two undesired cases mentioned above by aligning the completion of j_i's map phase and j_{i-1}'s reduce phase, see Fig. 4(c). Briefly, we use the slot assignment as a tunable knob to change the level of parallelism of map or reduce tasks. When we assign more map slots, map tasks obtain more system resources and could be finished faster, and vice versa for reduce tasks. In the rest of this section, we first present our basic solution with the prior knowledge of job workload. Then, we describe how to estimate the workload in practice when it is not available. In addition, we present a feedback control-based solution to provide more accurate estimation of the workload. Finally, we discuss the design of task scheduler in compliance with our solution.

4.1 Basic Sketch With Prior Knowledge of Workload

If the workload information is available, at the end of a task, Hadoop can obtain the value of the remaining workload for both map and reduce phases. Intuitively, we should assign more slots (resources) to the task type that has heavier remaining workload. Consider j_i and j_{i-1} are two active jobs and j_{i-1} is in reduce phase while j_i is in map phase. At the end of a task, we can get the number of remaining map tasks of j_i and remaining reduce tasks of j_{i-1}, indicated by $n'_m(i)$ and $n'_r(i-1)$. Let $w'_m(i)$ and $w'_r(i-1)$ represent the remaining workload of j_i's map phase and j_{i-1}'s reduce phase, we have:

$$w'_m(i) = n'_m(i) \cdot \bar{t}_m(i), \;\; w'_r(i-1) = n'_r(i-1) \cdot \bar{t}_r(i-1), \quad (2)$$

To align the completions of these two phases, the best parameters should satisfy the following condition:

$$\frac{n'_m(i)}{s_m} \cdot \bar{t}_m(i) = \frac{n'_r(i-1)}{s_r} \cdot \bar{t}_r(i-1) \;\Rightarrow\; \frac{w_m(i)'}{s_m} = \frac{w_r(i-1)'}{s_r} \quad (3)$$

Therefore, the number of map and reduce slots should be proportional to their remaining workloads as shown in Eq. 4-5,

$$s_m = \lfloor \frac{w'_m(i)}{w'_m(i) + w'_r(i-1)} \cdot S \rfloor, \quad (4)$$

$$s_r = S - s_m, \quad (5)$$

where s_m and s_r represent the target numbers of map and reduce slots respectively, and S is the total number of slots in the cluster which is configured based on system capacity. The floor function is used to ensure that the slot assignments are integer values. Furthermore, we introduce the upper bound s_m^h and the lower bound s_m^l for the map slots assignment. When the estimated value of s_m exceeds the bounds, we use the bound value as the new s_m value instead. In our design, s_m^l is set to be the number of nodes in the cluster (k) such that there is at least one map slot on each node. Similarly, s_m^h is set to be equal to $S - s_m^l$ such that the reduce slots number in each node is always greater than or equal to 1. The Hadoop system updates the values of s_m and s_r according to Eq. 4-5 every time a task is finished. If the current map slots are fewer than s_m, then the free slot will become a map slot and serve a map task. Otherwise, it turns to a reduce slot. With this setting, the current map and reduce phases could finish at approximately the same time with a high system resource utilization.

4.2 Workload Estimation

Our solution proposed above depends on prior knowledge of workload information. In practice, workload can be derived from job profiles, training phase, or other empirical settings. In some applications, however, workload information may not be available or accurate. In this subsection, we present a method that estimates the workload during the job execution without any prior knowledge.

We use w'_m and w'_r to represent the remaining workload of a map or reduce phase, i.e., the summation of execution time of the unfinished map or reduce tasks. Note that we only track the map/reduce workloads of running jobs, but not the jobs waiting in the queue. Basically, the workload is calculated as the multiplication of the number of remaining tasks and the average task execution time of a job. Specifically, when a map or reduce task is finished, the current workload information needs to be updated, as shown in Algorithm 2, where $n'_m(i)$/ $n'_r(i)$ is the number of unfinished map/reduce tasks of job j_i, and $\bar{t}_m(i)$/ $\bar{t}_r(i)$ means the average execution time of finished map/reduce tasks from j_i. Note that the execution time of each finished task is already collected and reported to the JobTracker in current Hadoop systems. In addition, we use the Welford's one

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

pass algorithm to calculate the average of task execution times, which incurs very low overheads on both time and memory space.

Algorithm 2 Workload Information Collector

if a map task of job j_i is finished **then**
  update the average execution time of a map task $\bar{t}_m(i)$
  $w'_m(i) = \bar{t}_m(i) \cdot n'_m(i)$
if a reduce task of job j_i is finished **then**
  update the average execution time of a reduce task $\bar{t}_r(i)$
  $w'_r(i) = \bar{t}_r(i) \cdot n'_r(i)$

4.3 Feedback Control-based Workload Estimation

The workload estimation scheme introduced in previous section works well under homogeneous system with fixed slots configuration. Under this case, all tasks from a job have similar execution time since they are processing the same amount of data with the same resource assignment. In our system design, however, the slots assignment is dynamically changed, which affects the per task execution time in practice. Assigning more slots to one type of tasks may cause the contention on a particular system resource and lead to an increased execution time of each following task in the same type. For example, in "Case 2" described in Section 2, when we use 1 map slot on each node, the average execution time of a map task is 18.5 sec. When we increase the number of map slots per node to 2, the average execution time of a map task becomes 23.1 sec with a 25% increase.

To overcome this issue, we have designed a feedback control based mechanism to tune the slots assignment. Under this mechanism, the slots assignment, s_m and s_r, is first calculated through Eq. 4-5. An additional routine is introduced to periodically update the workload information based on newly captured average task execution times. If the workloads have changed, then the slots assignment will also be updated according to Eq. 6-7.

$$s_m = s_m + \lfloor \alpha \cdot \left(\frac{w'_m}{w'_m + w'_r} - \frac{w_m}{w_m + w_r}\right) \cdot S \rfloor, \quad (6)$$
$$s_r = S - s_m. \quad (7)$$

When the new estimated workloads, i.e., w'_m and w'_r, differ from the previous estimation, an integral gain parameter α is used to control the new assignment of slots based on the new estimation. The Hadoop system will iteratively calculate s_m and s_r (Eq. 6-7) until there is no change on these two parameters. The value of α is set to be 1.2 in our system such that the slots assignment could converge quickly.

4.4 Slot Assigner

The task assignment in Hadoop works in a heartbeat fashion: the TaskTrackers report slots occupation situation to the JobTracker with heartbeat messages; and the JobTracker selects tasks from the queue and assigns them to free slots. There are two new problems need to be addressed when assigning tasks under TuMM. First, slots of each type should be evenly distributed across the slave nodes. For example, when we have a new slot assignment $s_m = 5, s_r = 7$ in a cluster with 2 slave nodes, a 2:3/4:3 map/reduce slots distribution is better than the 1:4/5:2 map/reduce slots distribution in case of resource contention. Second, the currently running tasks may stick with their slots and therefore the new slot assignments may not be able to apply immediately. To address these problems, our slot assignment module (SA) takes both the slots assignment calculated through Eq. 6-7 and the situation of currently running tasks into consideration when assigning tasks.

The process of SA is shown in Algorithm 3. The SA first calculates the map and reduce slot assignments of slave node x (line 1), indicated by $s_m(x)$ and $s_r(x)$, based on the current values of s_m and s_r and the number of running tasks in cluster. We use the floor function since slots assignments on each node must be integers. Due to the flooring operation, the assigned slots ($s_m(x)+s_r(x)$) on node x may be fewer than the available slots (S/k). In lines 3–6, we increase either $s_m(x)$ or $s_r(x)$ to compensate slot assignment. The decision is based on the deficit of current map and reduce slots (line 3), where s_m/ s_r represent our target assignment and rt_m/ rt_r are the number of current running map/reduce tasks. Eventually, we assign a task to the available slot in lines 7–10. Similarly, the decision is made by comparing the deficit of map and reduce tasks on node x, where $s_m(x)$/ $s_r(x)$ are our target assignment and rt_m(x)/ $rt_r(x)$ are the numbers of running tasks.

Algorithm 3 Slot Assigner

0: **Input:** Number of slave nodes in cluster: k
  Total numbers of running map/reduce tasks: rt_m, rt_r;
0: **When** receive heartbeat message from node x with the number of running map/reduce tasks on node x: $rt_m(x)$, $rt_r(x)$;
1: **Initialize** assignment of slots for node x:
  $s_m(x) \leftarrow \lfloor s_m/k \rfloor, s_r(x) \leftarrow \lfloor s_r/k \rfloor$;
2: **if** $(s_m(x) + s_r(x)) < S/k$ **then**
3:   **if** $(s_m - rt_m) > (s_r - rt_r)$ **then**
4:     $s_m(x) \leftarrow s_m(x) + 1$;
5:   **else**
6:     $s_r(x) \leftarrow s_r(x) + 1$;
7: **if** $(s_m(x) - rt_m(x)) > (s_r(x) - rt_r(x))$ **then**
8:   assign a map task to node x;
9: **else**
10:   assign a reduce task to node x.

5 Dynamic Slot Configuration Under Heterogeneous Environments

In the previous sections, we discussed about the static and dynamic slot configuration in a homogeneous Hadoop cluster environment, where all servers have the same computing and memory capacities. However, heterogeneous environments are fairly common in today's cluster systems. For example, system managers

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

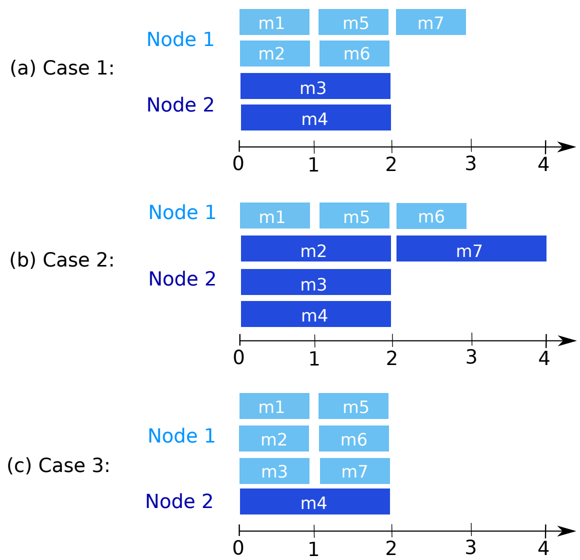


Fig. 6. Illustrating a Hadoop job with 7 map tasks running in a heterogeneous Hadoop cluster with 2 nodes and 4 map slots in total. The map phase of that job run faster when we have (c) 3 map slots on Node 1 and 1 map slot on Node 2, than when we have (a) 2 map slot on Node 1 and 2 map slots on Node 2, and (b) 1 map slot on Node 1 and 3 map slots on Node 2.

of a private data center could always scale up their data center by adding new physical machines. Therefore, physical machines with different models and different resource capacities can exist simultaneously in a cluster.

When deploying a Hadoop cluster in such a heterogeneous environment, tasks from the same job may have different execution times when running on different nodes. In this case, a task's execution time highly depends on a particular node where that task is running. A job's map tasks may run faster on a node which has faster cpu per slot while its reduce tasks may experience shorter execution times on the other nodes that have more memory per slot. Estimating the remaining workloads and deciding the slot configuration in this heterogeneous Hadoop cluster thus becomes more complex.

For example, consider a Hadoop job with 7 map tasks and a Hadoop cluster with two heterogeneous nodes such that node 1 is faster than node 2. Consider a cluster configured with 4 map slots in total, and one map task of that job takes 1 second and 2 seconds to finish on node 1 and node 2, respectively. We note that in this heterogeneous Hadoop cluster, various slot configurations will yield different performance (e.g., the execution time) of this job. As illustrated in Figure 6 case 1, the total execution time of the map phase takes 3 seconds if we set 2 map slots on node 1 and 2 map slot on node 2. However, the map phase execution time can be improved to 3 seconds if we change the slot configures on these two nodes, i.e., 3 map slot on node 1 and 1 map slots on node 2. This situation indicates that it is harder to predict the time needed to finish the map phase or reduce phase in the heterogeneous environment, and evenly distribute the map (or reduce) slot assignments across the cluster will no longer work well.

We thus argue that the centralized method (i.e., the algorithms described in Section 4 for a homogeneous Hadoop cluster) which utilizes the overall workload information to set the slot assignments over the entire cluster does not work well any more when the nodes in the cluster become heterogenous. Motivated by this, we present in this section a new version of TuMM, named H_TuMM, which dynamically sets the slot configurations for each node in a heterogeneous Hadoop cluster in order to reduce the makespan of Hadoop jobs.

5.1 Problem Formulation

The problem of finding the optimal slot assignment to map and reduce tasks in a heterogeneous Hadoop cluster that aligns the current running map and reduce workloads and minimizes the time required to finish current map and reduce workloads could be formulated as a linear programming problem as follows:

$$
\begin{aligned}
&Minimize \quad max \quad \{v_m^i * \overline{t_m^i}\}, \forall i \in I, && (8)\\
&subject\ to:\\
s_m^i + s_r^i \quad &= \quad S^i, \forall i \in I, && (9)\\
\sum v_m^i * s_m^i \quad &>= \quad n_m', \forall i \in I, && (10)\\
\sum v_r^i * s_r^i \quad &>= \quad n_r', \forall i \in I, && (11)\\
(v_m^j - 1) * \overline{t_m^j} \quad &<= \quad v_m^i * \overline{t_m^i},\\
&\quad \forall i, j \in I, \ if\ t_m^i < t_m^j, && (12)\\
v_m^i * \overline{t_m^i} \quad &<= \quad (v_m^j + 1) * \overline{t_m^j},\\
&\quad \forall i, j \in I, \ if\ t_m^i < t_m^j. && (13)\\
(v_r^j - 1) * \overline{t_r^j} \quad &<= \quad v_r^i * \overline{t_r^i},\\
&\quad \forall i, j \in I, \ if\ t_r^i < t_r^j, && (14)\\
v_r^i * \overline{t_r^i} \quad &<= \quad (v_r^j + 1) * \overline{t_r^j},\\
&\quad \forall i, j \in I, \ if\ t_r^i < t_r^j. && (15)\\
(v_r^j - 1) * \overline{t_r^j} \quad &<= \quad v_m^i * \overline{t_m^i},\\
&\quad \forall i, j \in I, \ if\ t_m^i < t_r^j, && (16)\\
v_m^i * \overline{t_m^i} \quad &<= \quad (v_r^j + 1) * \overline{t_r^j},\\
&\quad \forall i, j \in I, \ if\ t_m^i < t_r^j, && (17)
\end{aligned}
$$

Where, I represents the set of nodes in the cluster, $\overline{t_m^i}/\overline{t_r^i}$ represents the average map/reduce task execution time on node i, and n_m'/n_r' represents the remaining unassigned map/reduce tasks of jobs that are currently running under their map/reduce phases. Additionally, v_m^i/v_r^i denotes the waves of map/reduce tasks that have to run on node i before the finish time of current map/reduce phase, s_m^i/s_r^i represents the optimal slot assignment to map/reduce on node i, and S^i represents the constraint of total available slot number of node i. The target is to minimize the finish time of the current map phase under a set of constraints: Eq.(9) states that

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

the slots assigned to map or reduce tasks on each node should not exceed the pre-defined slot constraint of that particular node; Eq.s(10)-(11) state that all the remaining tasks of current running jobs need to be assigned across the cluster; Eq.s(12)-(13) state that the difference between the times each node takes to execute its assigned map tasks should not exceed the execution time of one task (this constraint is decided by the nature of the Hadoop scheduler); Eq.s(14)-(15), similarly, state that the time each node takes to execute its assigned reduce tasks should be roughly the same; and Eq.s(16)-(17) state that the finish time of map and reduce workloads that are dispatched to each node should also be aligned to avoid slot idleness.

However, it is quite time consuming to solve the above problem especially when the number of nodes in a Hadoop cluster is large. In order to make decisions for slot configurations instantly when the workloads change, we instead present a new algorithm which solves the problem by heuristically assigning slots for map and reduce tasks on each node in a heterogeneous Hadoop cluster.

5.2 Algorithm Design: H_TuMM

H_TuMM shares the similar idea of TuMM, i.e., dynamically assign slots to map and reduce tasks to align the process of map and reduce phase based on the collected workload information. The key difference of H_TuMM is to set the slot configurations for each node individually in a heterogeneous cluster, i.e., each of those nodes will have different slot assignment ratio between map and reduce tasks.

To accomplish it, H_TuMM collects the workload information on the entire cluster and on each individual node as well: when a map/reduce task is finished on node i, the workload collector updates (1) the average execution time of map/reduce tasks, i.e., $\overline{t_m}/\overline{t_r}$; and (2) the average execution of map/reduce tasks that ran on node i, i.e., $\overline{t_m^i}/\overline{t_r^i}$.

Based on the collected workload information, H_TuMM performs slot assignment for each node as shown in Algorithm 4. Once a slot in node i becomes available, H_TuMM first updates the slot assignments to map tasks (s_m^i) and reduce tasks (s_r^i) on node i. Such that the ratio of slot assignments (i.e., s_m^i/s_r^i) is equal to the ratio of remaining map and reduce workloads (i.e., $\frac{\overline{t_m^i}*n'_m}{\overline{t_r^i}*n'_r}$, see line 1-2 in Algorithm 4. Therefore, map and reduce phases running on that node are aligned. Similar to Algorithm 3, floor function is used to make sure that slots assignments are all integers. If there is one remaining slot, in this case, the free slot will be assigned to a map (resp. reduce) task if map (resp. reduce) tasks run relatively faster on this node compared to the average execution time across the entire cluster in order to improve the efficiency, see line 3-7 in Algorithm 4. When the slot assignment on the specific node is determined, the JobTracker can assign tasks based on the new slot configuration and the number of currently running tasks on that node (i.e., rt_m^i and rt_r^i), see line 8-11 in Algorithm 4.

Algorithm 4 Slot Assignment for Node i

0: **Input:** Average task execution time on node i and across the cluster, and the remaining task number of current running jobs;
0: **When** Node i has free slots and ask for new task assignment through the heartbeat message;
1: $s_m^i \leftarrow \lfloor S^i * \frac{\overline{t_m^i}*n'_m}{\overline{t_m^i}*n'_m+\overline{t_r^i}*n'_r} \rfloor$;
2: $s_r^i \leftarrow \lfloor S^i * \frac{\overline{t_r^i}*n'_r}{\overline{t_m^i}*n'_m+\overline{t_r^i}*n'_r} \rfloor$;
3: **if** $s_m^i + s_r^i \leq S^i$ **then**
4:     **if** $\frac{\overline{t_m^i}}{\overline{t_m}} > \frac{\overline{t_r^i}}{\overline{t_r}}$ **then**
5:         $s_r^i \leftarrow S^i - s_m^i$;
6:     **else**
7:         $s_m^i \leftarrow S^i - s_r^i$.
8: **if** $(s_m^i - rt_m^i) > (s_r^i - rt_r^i)$ **then**
9:     assign a map task to node i;
10: **else**
11:     assign a reduce task to node i;

6 EVALUATION

6.1 Experimental Setup and Workloads

6.1.1 Implementation

We implemented our new scheme (for homogeneous environment and heterogeneous environment) on the top of Hadoop Version 0.20.2. First, we added two new modules into the *JobTracker*: the **W**orkload Monitor (WM) that is responsible to collect past workload information such as execution times of completed tasks and to estimate the workloads of currently running map and reduce tasks and the **S**lot Assigner (SA) which uses the estimated information received from WM to adjust the slot ratio between map and reduce for each slave node. The *JobTracker* with these additional modules will then assign tasks to a slave node based on the adjusted slot ratio and the current slot status at that particular node. In addition, we modified the *TaskTracker* as well as the *JvmManager* that runs at each slave node to check the number of individual map and reduce tasks running on that node based on the new slot ratio received from the *JobTracker*. The architecture overview of this new Hadoop framework is shown in Fig. 5.

6.1.2 Benchmarks

We choose five representative data-analyzing Hadoop benchmarks from Purdue MapReduce Benchmarks Suite [12]:

- *Inverted Index*: take text documents as input and generate word to document indexing.
- *Histogram Rating*: take the movie rating data as input and calculate a histogram of input data.
- *Word Count*: take text documents as input and count the occurrence of each word.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

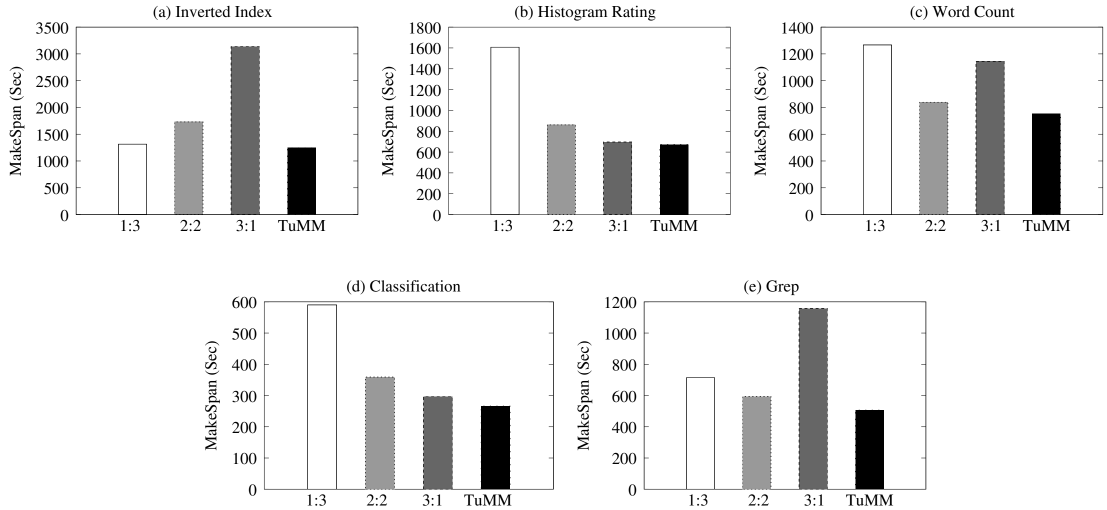


Fig. 7. Makespans of five Hadoop applications under TuMM and three static slot configurations.

- *Classification*: take the movie rating data as input and classify the movies into predefined clusters.
- *Grep*: take text documents as input and search for a pattern in the files.

In addition, we use different sizes of movie rating data [12] that consists of user ranking information and wiki category links data [13] that includes the information about wiki page categories, as the input to the above five benchmarks. A 10GB movie rating data and a 7GB wiki category data are used as input for experiments in the homogeneous cluster. And experiments under the heterogeneous cluster use a 8GB movie rating data and a 8GB wiki category data as inputs.

We further choose TPC-H [14] queries expressed as Pig programs [15] to validate the performance of H_TuMM under heterogeneous environments. A data generator in TPC-H can be used to create a database with the customized size. In such a database, there are totally eight tables, i.e., *customer, supplier, orders, lineitem, part, partsupp, nation,* and *region*. In our experiments, we generated a database with 4G data in total and selected three queries from the TPC-H benchmark to evaluate the performance of H_TuMM.

- *TPCH-Q15:* This query finds the supplier who contributed the most to the overall revenue for parts shipped during a given quarter of a given year.
- *TPCH-Q16:* This query counts the number of suppliers who can supply parts that satisfy a particular customer's requirements.
- *TPCH-Q18:* This query finds a list of the top 100 customers who have ever placed large quantity orders.The query lists the customer name, customer key, the order key, date and total price and the quantity for the order.

6.2 Evaluation in Homogeneous Environment

In this section, we evaluate the performance of TuMM in terms of the makespan of a batch of MapReduce jobs in a homogeneous environment. we launch a Hadoop cluster in the Amazon EC2 environment which consists of 5 *m1.xlarge* Amazon EC2 instances. Specifically, we have one master node and four slave nodes in the cluster. The number of slots which can be available on each slave node is set as 4 since an m1.xlarge instance at Amazon EC2 has 4 virtual cores.

We first consider simple workloads which consist of jobs from a single MapReduce benchmark and then validate the robustness of our approach with a mixed workload that is a combination of MapReduce benchmarks from Purdue MapReduce Benchmarks Suite.

6.2.1 Simple Workloads

We here conduct a set of experiments such that in each experiment 3 Hadoop jobs from one of the above benchmarks (see Section 6.1.2) are waiting for service. We remark that such a simple workload is often found in real systems as the same Hadoop jobs may be executed repeatedly to process similar or different input data sets. In our experiments, three Hadoop jobs use the same data set as the input. Furthermore, as the comparisons, we evaluate the performance under the static slot ratios for map and reduce. Since all the slave nodes normally have the same slot ratio in current Hadoop implementations, With our setting in the evaluation (i.e., total number of slots per node is 4), there are three static configuration alternatives, i.e., 1:3, 2:2 and 3:1, for a Hadoop cluster. So we enumerate all these three possible settings for the comparison with our solution.

Fig. 7 shows makespans (i.e., the completion lengths) of a given set under different slot configurations. We first observe that the performance varies a lot under three static slot settings. For example, the *Inverted Index* jobs experience the fastest makespan when the slot ratio is equal to 1:3. In contrast, the *Histogram Rating* jobs achieve better performance when we assign more slots to their map tasks, e.g., with slot ratio of 3:1. We also observe

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

that TuMM always yields the best performance, i.e., the shortest makespan, for all the five Hadoop benchmarks. We interpret this effect as the result of dynamic slot ratio adjustments enabled by TuMM.

Compared to the slot ratio of 2:2, our approach in average achieves about 20% relative improvement in the makespan. Moreover, such improvement becomes more visible when the workloads of map and reduce tasks become more unbalanced. For example, the makespan of the *Inverted Index* jobs is reduced by 28% where these jobs have their reduce phases longer than their map phases.

6.2.2 Mixed Workloads

In the previous experiments, each workload only contains jobs from the same benchmark. Now, we consider a more complex workload, which mixes jobs from different Hadoop benchmarks. Reducing the makespan for such a mixed workload thus becomes non-trivial. One solution to tackle this problem is to shuffle the execution order of these jobs. For example, the classic Johnson's algorithm [16] that was proposed for building an optimal two-stage job schedule, could be applied to process a set of Hadoop jobs and minimize the makespan of a given set as well. However, this algorithm needs to assume a priori knowledge of the exact execution times of each job's map and reduce phases, which unfortunately limits the adoption of this algorithm in real Hadoop systems. Moreover, for some cases, it may not be feasible to change the execution order of jobs, especially when there exists dependency among jobs or some of them have high priority to be processed first.

To address the above issues, our solution leverages the knowledge of the completed tasks to estimate the execution times of the currently running tasks and reduces the makespan of a set of jobs by dynamically adjusting the slot assignments for map and reduce tasks. As a result, TuMM does not need to change the execution order of jobs and does not need to know the exact task execution times in advance, either.

We generate the mixed workload for our experiments by randomly choosing 10 jobs from 5 different Hadoop benchmarks. In order to investigate the impact of job execution order, we also consider three different execution sequences, including (1) a sequence generated by Johnson's algorithm which can be considered as the optimal case in terms of the makespan; (2) a sequence that is inverse to the first one and can be considered as the worst case; and (3) a sequence that is random. Similarly, we evaluate the performance (i.e., makespan) under TuMM and three static slot configurations.

Fig. 9 shows the makespans of the 10 jobs in the mixed workload. We first observe that among three static settings, the slot ratio of 2:2 always achieves the best performance under three different execution orders. This is because the overall workloads of map tasks and reduce tasks from the 10 jobs are well balanced. We also notice that with a fixed number of slots per node, different job execution orders could yield different makespans. While our solution always achieves the best performance and the impact of execution sequence on our solution's performance becomes less visible. This means that no matter what the execution order is, TuMM can always serve the jobs with the shortest makespans. That is, our approach allows to improve the performance in terms of makespan without changing the execution order of jobs.

To better understand how TuMM uses the slot ratio as a tunable knob to improve the makespan, we further plot the task execution times for each job as well as the transient slot assignments in Fig. 8, where the plots in the first row depict the running period of each task from the 10 jobs while the plots in the second row illustrate how the slot assignments change across time. As shown in Fig. 8, TuMM dynamically adjusts the slot assignments to map and reduce tasks based on the estimated workload information. For example, in the first 1200 seconds of Fig. 8-(2), TuMM attempts to assign more slots to reduce tasks. Then, in the later 1200 seconds, TuMM turns to allow more available map slots on each node. This is because the Johnson's algorithm shuffles the order of 10 jobs such that all the reduce intensive jobs such as *Inverted Index* and *Grep* run before the map intensive jobs, e.g., *Histogram Rating* and *Classification*. The only exception is the first 100s where most of the slots are assigned to map tasks even though the running job actually has reduce intensive workloads. That is because TuMM does not consider the reduce workloads of this job in the first 100 seconds until its map tasks are finished. Fig. 8-(1) shows the corresponding task execution times under TuMM. It is obvious that each job's reduce phase successfully overlaps with the map phase of the following job and the makespan of 10 jobs is then shortened compared to the static settings.

In summary, TuMM achieves non-negligible improvements in makespan under both simple workloads and mixed workloads. By leveraging the history information, our solution accurately captures the changes in map and reduce workloads and adapts to such changes by adjusting the slot assignments for these two types of tasks. Furthermore, different job execution orders do not affect TuMM's performance. That is, our solution can still reduce the makespan without changing the execution order of a given set of jobs.

TABLE 4
Maximum and minimum task execution times (in seconds) of each job across Heter1 cluster.

	Map Tasks		Reduce Tasks	
Benchmarks	Min.	Max.	Min.	Max.
Classification	6.5	24.1	9.5	15.9
Histogram Rating	8.5	24.8	9.7	25.5
Inverted Index	5.1	17.4	16.5	48.1
Word Count	11.5	31.4	12.6	25.2
Grep	6.7	25.1	12.7	29.5

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

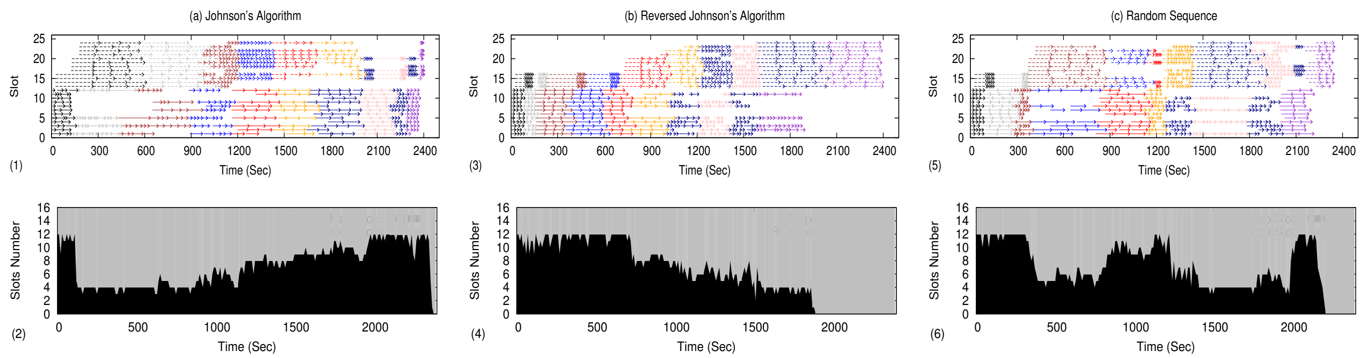


Fig. 8. Illustrating task execution times and slot assignments across time under TuMM, where the job execution sequence is (a) generated by Johnson's algorithm; (b) inverse to the first one; and (c) random. In the plots at the second row, black (resp. gray) areas represent the number of available map (resp. reduce) slots in the cluster.

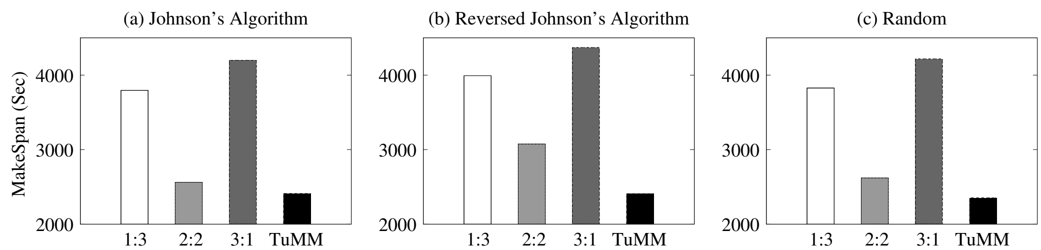


Fig. 9. Makespans of a mixed workload under TuMM and three static slot configurations. Three execution orders are also considered: (a) a sequence follows Johnson's algorithm, (b) a sequence with reversed order of Johnson's algorithm, and (c) a random sequence.

TABLE 3
Cluster configuration of two heterogeneous clusters, i.e., Heter1 and Heter2.

Cluster	Instance Type	Number of Slave Nodes	Number of Slots Per Node		Average Resources Per Slot	
			Map Slots	Reduce Slots	Compute Units	Memory
Heter1	m1.xlarge	3	1	2	2.67	5 GB
	m1.xlarge	3	2	2	2	3.75 GB
	m1.large	3	2	1	1.33	2.5 GB
Heter2	m1.xlarge	3	2	2	2	3.75 GB
	c1.xlarge	3	2	2	5	1.75 GB
	m2.xlarge	3	2	2	1.63	4.25 GB

6.3 Evaluation in Heterogeneous Environment

In this section, we evaluate the performance of H_TuMM in the heterogeneous environments. The mixed workloads introduced in previous section and the TPC-H benchmarks are used to validate the effectiveness and robustness of our scheme.

We build up two heterogeneous Hadoop clusters in the Amazon EC2 environment, i.e., Heter1 and Heter2. The detailed cluster configurations are shown in Table 3. Specifically, each cluster has one m1.xlarge type master node and 9 slave nodes. There are three different groups of slave nodes in each cluster, and slots in different groups have different physical resource capacities. We list the approximate number of compute units and memory sizes that shared by one slot in different node group in Table 3. It is clear that slots have equally scaled cpu and memory capacities in different node groups of Heter1, and skewed cpu and memory capacity ratios in different node groups of Heter2.

6.3.1 *Mixed Workloads*

We first conduct experiments using the mixed workload as described in Section 6.2.2, where the size of input data is 8GB and the data block size is set to 64MB such that each job has 128 map tasks. Additionally, the number of reduce tasks from each job is set to be 150 and 80 for the *Inverted Index* benchmark and the remaining benchmarks, respectively.

In order to investigate the impact of heterogeneous environments on Hadoop performance, we measured the maximum and minimum task execution times for each job across different slave nodes in Heter1 cluster. As shown in Table 4, each job's task execution times

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

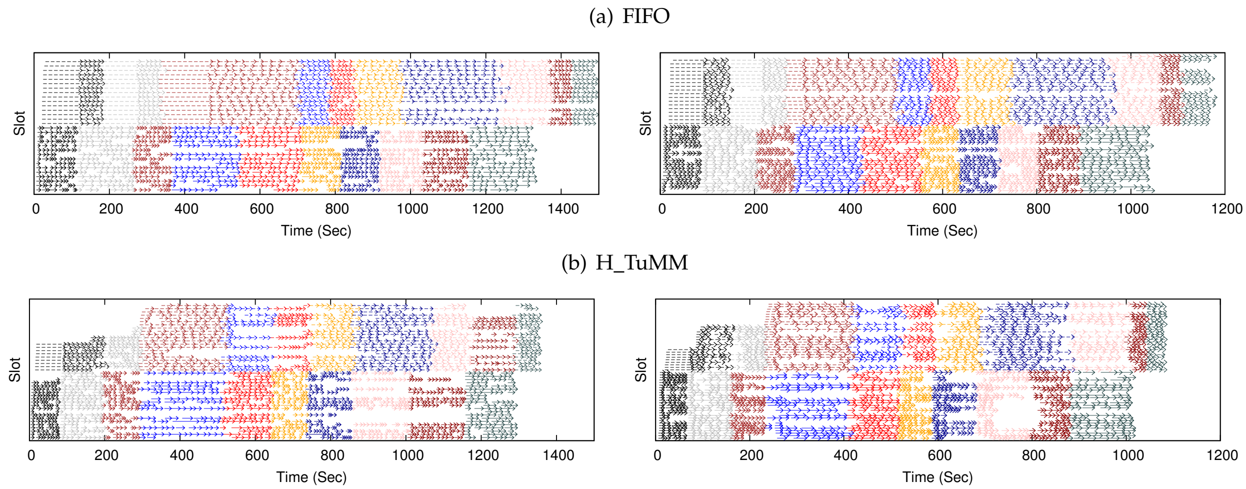


Fig. 10. Task execution times of a batch of mixed benchmarks under (a) FIFO and (b) H_TuMM. The plots in the left (resp. right) column show the results from Heter1 (resp. Heter2) cluster. There are in total 30 (resp. 36) slots across Heter1 (resp. Heter2) cluster, i.e., there are at most 30 (resp. 36) running tasks in Heter1 (resp. Heter2) cluster at any given time.

are no longer uniform, for example, the slowest map task(s) of a *Classification* job could almost run four times longer than the fastest one(s). We interpret this effect by observing the variance of resource capacity among the slots on different slave nodes.

Figure 10 illustrates task execution details (i.e., the running period of each task) of a batch of mixed benchmarks under both FIFO and H_TuMM scheduling policies. Plots in the left (resp. right) column show the results from Heter1 (resp. Heter2) cluster. We observe that in both clusters, our new H_TuMM policy dynamically change the slot assignment to map and reduce tasks over time while keep the number of total running tasks the same at any given time. Through tunning the slot assignments, H_TuMM successfully aligns each jobs reduce phase with the map phase of the following job and thus avoids the waste of slot resources. As a result, the makespan of 10 Hadoop jobs in the mixed workload becomes shorter under H_TuMM than under FIFO.

Figure 11 further depicts the number of map tasks that are dispatched by H_TuMM to each node over time in Heter1 cluster. Clearly, our H_TuMM dynamically sets the slot configurations for each node, such that the number of map tasks running on each node varies over time and each node is assigned with different number of map tasks (slots) at each moment.

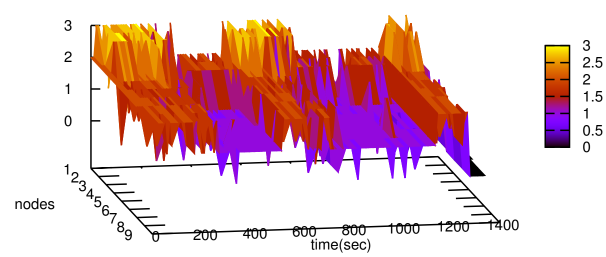


Fig. 11. Number of map tasks running on each node in Heter1 cluster under H_TuMM policy.

6.3.2 TPC-H Workloads

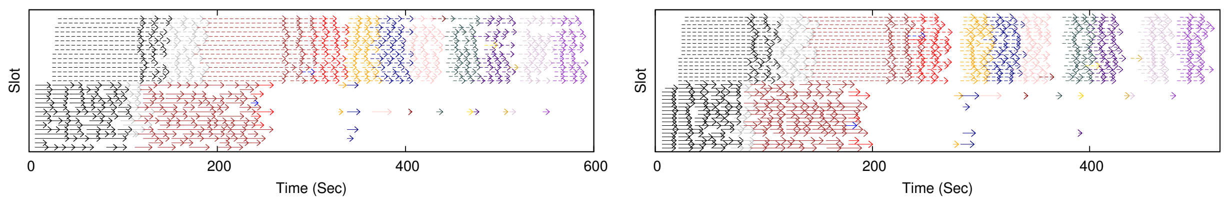
We now turn to the experiments which run the TPC-H benchmark in the heterogeneous clusters. As described in Section 6.1.2, we chose 3 different queries from TPC-H query set. Each of the three queries consists of 5 sub queries. A dependency chain exists between the sub queries from the same query, i.e., each sub query could start only after its precedent sub query completes. It follows that the 5 sub queries form the same query are indeed submitted and executed sequentially in the predefined order. Furthermore, the input data sizes of different sub queries vary even in the same query. Therefore, each sub query has different map task numbers. For example, in this set of experiments, the first sub query of all the three queries has the largest input data size and thus most map tasks are clustered in the first few sub queries, while the following sub queries have relatively small amount of map tasks.

We submit the 3 queries (i.e., TPCH-Q15, TPCH-Q16 and TPCH-Q18) to the cluster at the same time, such that the sub queries of each query could interleave with each other. The makespans of these three TPC-H queries in two heterogeneous clusters (i.e., Heter1 and Heter2) are shown in Table 5 and the execution details of these queries are further plotted in Figure 12. We observe that by dynamically adjusting slot assignments on each node, H_TuMM improves the performance (i.e., reducing the makespan) of all the three TPC-H queries when compared to FIFO. Such performance improvement can be consistently observed in both two heterogeneous clusters. Figure 12 further illustrates that the map and reduce phases are well aligned under the H_TuMM policy.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415802, IEEE Transactions on Cloud Computing

(a) FIFO



(b) H_TuMM

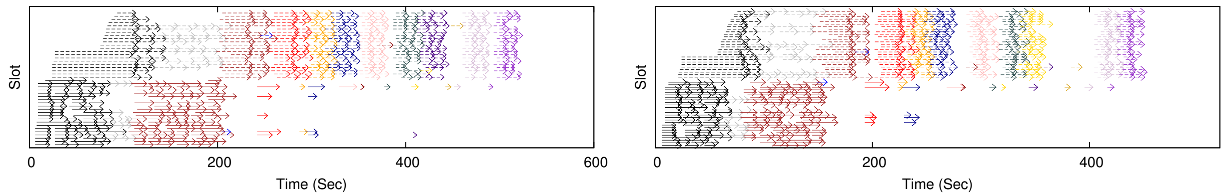


Fig. 12. Task execution times of three TPC-H queries under (a) FIFO and (b) H_TuMM. The plots in the left (resp. right) column show the results from Heter1 (resp. Heter2) cluster. Different colors represent different sub queries.

TABLE 5
Makespans of TPC-H queries under FIFO and H_TuMM in two heterogeneous clusters. The numbers in the parentheses are the relative improvements against FIFO.

Cluster	Query	FIFO	H_TuMM
Heter1	Q15	523.6	465.0 (11.1%)
	Q16	564.1	495.3 (12.2%)
	Q18	598.4	529.2 (11.5%)
Heter2	Q15	452.0	397.7 (12.0%)
	Q16	491.9	437.1 (11.1%)
	Q18	519.7	456.4 (12.4%)

7 Related Works

An important direction for improving the performance of a Hadoop system is job scheduling. The default FIFO scheduler does not work well in a shared cluster with multiple users and a variety of jobs. Fair [17] and Capacity [18] schedulers were proposed to ensure that each job can get a proper share of the available resources; and Quincy [5] addressed the scheduling problem with locality and fairness constraints. Zaharia et al. [3] proposed a delay scheduling to further improve the performance of the Fair scheduler by increasing data locality. Verma et al. [4] introduced a heuristic to minimize the makespan of a set of independent MapReduce jobs by applying the classic Johnson's algorithm.

Another category of schedulers further consider user-level goals while improving the performance. ARIA, a deadline aware scheduler, was recently proposed in [6], which always schedules a job with the earliest deadline and uses the Lagrange's method to find the minimum number of slots for each job in order to meet the predefined deadline. Similarly, Polo et al. [7] estimated the task execution times based on the average execution times of the completed tasks instead of the job profiles. Task execution times were then used to calculate the number of slots that a job needed to meet its deadline. Different deadline and locality aware scheduling algorithms are evaluated with empirical analysis for Hadoop system in [19]. Although these deadline aware schedulers support user-level goals, their techniques are still based on static slot configurations, i.e., having a fixed number of map slots and reduce slots per node throughout the lifetime of a cluster.

Fine-grained resource aware management is another important direction for improving performance in Hadoop. RAS [10] leverages existing profiling information to dynamically determine the number of job slots and their placement in the cluster. The goal of this approach is to maximize the resource utilization of the cluster and to meet job completion time deadlines. More recently, [11] introduces a local resource manager at each TaskTracker to detect task resource utilization and predict task finish time, and a global resource manager at the JobTracker to coordinate the resource assignments to each task; and [9] addresses the cluster resource utilization problem by developing a dynamic split model of resource utilization. Our work is complementary to the above techniques.

The Hadoop community recently released Next Generation MapReduce (YARN) [8], the latest architecture of Hadoop MapReduce, which replaces the fixed-size slot with a resource container that works in a fine-grained resource level. There is no longer map/reduce slots concept in YARN system. Specifically, YARN users need to specify requirements of cpu cores and memory size of each type of tasks, such as map, reduce and application master. Task assignment is then based on resource requirement of tasks and the residual resources of slave nodes. Consequently, resource management for YARN is quite different from the schemes that we proposed in this paper. We investigate the resource management problem in YARN system in our work [20]. The main objective of this paper is to reduce the completion length

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

(i.e., makespan) of a set of MapReduce jobs in slot based first generation Hadoop MapReduce system.

8 Conclusion

In this paper, we presented a novel slot management scheme, named TuMM, to enable dynamic slot configuration in Hadoop. The main objective of TuMM is to improve resource utilization and reduce the makespan of multiple jobs. To meet this goal, the presented scheme introduces two main components: *Workload Monitor* periodically tracks the execution information of recently completed tasks and estimates the present workloads of map and reduce tasks and *Slot Assigner* dynamically allocates the slots to map and reduce tasks by leveraging the estimated workload information. We further extended our scheme to manage resources (slots) for heterogeneous clusters. The new version of our scheme, named H_TuMM, reduces the makespan of multiple jobs by separately setting the slot assignments for the node in a heterogeneous cluster. We implemented TuMM and H_TuMM on the top of Hadoop v0.20.2 and evaluated both schemes by running representative MapReduce benchmarks and TPC-H query sets in Amazon EC2 clusters. The experimental results demonstrate up to 28% reduction in the makespans and 20% increase in resource utilizations. The effectiveness and the robustness of our new slot management schemes are validated under both homogeneous and heterogeneous cluster environments. In the future, we will further investigate the optimal total slot number configuration in the slot based Hadoop platform as well as the resource management policy in next generation Hadoop YARN platforms.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[2] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/
[3] M. Zaharia, D. Borthakur, J. S. Sarma *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys'10*, 2010.
[4] A. Verma, L. Cherkasova, and R. H. Campbell, "Two sides of a coin: Optimizing the schedule of mapreduce jobs to minimize their makespan and improve cluster performance," in *MASCOTS'12*, Aug 2012.
[5] M. Isard, Vijayan Prabhakaran, J. Currey *et al.*, "Quincy: fair scheduling for distributed computing clusters," in *SOSP'09*, 2009, pp. 261–276.
[6] A. Verma, Ludmila Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *ICAC'11*, 2011, pp. 235–244.
[7] J. Polo, D. Carrera, Y. Becerra *et al.*, "Performance-driven task co-scheduling for mapreduce environments," in *NOMS'10*, 2010.
[8] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.
[9] X. W. Wang, J. Zhang, H. M. Liao, and L. Zha, "Dynamic split model of resource utilization in mapreduce," in *DataCloud-SC '11*, 2011.
[10] J. Polo, C. Castillo, D. Carrera *et al.*, "Resource-aware adaptive scheduling for mapreduce clusters," in *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*, 2011.
[11] B. Sharma, R. Prabhakar, S.-H. Lim *et al.*, "Mrorchestrator: A fine-grained resource orchestration framework for mapreduce clusters," in *CLOUD'12*, 2012.
[12] Purdue mapreduce benchmarks suite. [Online]. Available: http://web.ics.purdue.edu/~fahmad/benchmarks.htm
[13] Wiki data sets. [Online]. Available: http://dumps.wikimedia.org/
[14] Tpc-h benchmark. [Online]. Available: http://www.tpc.org/tpch/
[15] Tpc-h benchmark on pig. [Online]. Available: https://issues.apache.org/jira/browse/PIG-2397
[16] S. M. Johnson, "Optimal two- and three-stage production schedules with setup times included," *Naval Research Logistics Quarterly*, vol. 1, no. 1, pp. 61–68, 1954.
[17] M. Zaharia, D. Borthakur, J. S. Sarma *et al.*, "Job scheduling for multi-user mapreduce clusters," University of California, Berkeley, Tech. Rep., Apr. 2009.
[18] Capacity scheduler. [Online]. Available: http://hadoop.apache.org/common/docs/r1.0.0/capacity_scheduler.html
[19] L. T. Phan, Z. Zhang, Q. Zheng, B. T. Loo, and I. Lee, "An empirical analysis of scheduling techniques for real-time cloud-based data processing," in *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1–8.
[20] Y. Yao, J. Wang, B. Sheng, J. Lin, and N. Mi, "Haste: Hadoop yarn scheduling based on task-dependency and resource-demand," in *IEEE International Conference on Cloud Computing (Cloud'14)*, 2014.

Yi Yao is a Ph.D. student at Northeastern University, Department of Electrical and Computer Engineering, Boston, MA. He received his M.S. and B.S. in Computer Science from the Southeast University, China, in 2010 and 2007. His current research interests are resource management, scheduling and cloud computing.

Jiayin Wang is currently a Ph.D. student at University of Massachusetts Boston, Department of Computer Science, Boston, MA. She received her Bachelor degree in Electrical Engineering from XIDIAN University, China in 2005. Her research interests include cloud computing and wireless networks.

Bo Sheng is an assistant professor in the Department of Computer Science at University of Massachusetts Boston. He received his Ph.D. in computer science from the College of William and Mary in 2010. His research interests include mobile computing, wireless networks, security and cloud computing.

Chiu C. Tan is an assistant professor in the Department of Computer and Information Sciences at Temple University. He received his PhD from the College of William and Mary in 2010. His current research interests are in cloud computing security, smarthealth systems, and wireless network security. He is also the director for the NSF/DoD REU Site program at Temple University.

Ningfang Mi is an Assistant Professor at Northeastern University, Department of Electrical and Computer Engineering, Boston, MA. She received her Ph.D. degree in Computer Science from the College of William and Mary, VA in 2009. She received her M.S. in Computer Science from the University of Texas at Dallas, TX in 2004 and her B.S. in Computer Science from Nanjing University, China, in 2000. Her current research interests are performance evaluation, capacity planning, resource management, simulation, data center and cloud computing.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.