Scheduling Heterogeneous MapReduce Jobs for Efficiency Improvement in Enterprise Clusters

Yi Yao
Northeastern University
Email: yyao@ece.neu.edu

Jianzhe Tai
Northeastern University
Email: jtai@ece.neu.edu

Bo Sheng
University of Massachusetts Boston
Email: shengbo@cs.umb.edu

Ningfang Mi
Northeastern University
Email: ningfang@ece.neu.edu

***Abstract*—The MapReduce paradigm and its open source implementation Hadoop are emerging as an important standard for large-scale data-intensive processing in both industry and academia. A MapReduce cluster is typically shared among multiple users with different types of workloads. When a flock of jobs are concurrently submitted to a MapReduce cluster, they compete for the shared resources and the overall system performance might be seriously degraded. Therefore, one challenging issue is to efficiently schedule all the jobs in such a shared MapReduce environment. However, we find that prior scheduling algorithms supported by Hadoop cannot guarantee good performance under different workloads. In this paper, we propose a new Hadoop scheduler, which leverages the knowledge of workload patterns to improve the system performance by dynamically tuning the resource shares among users and the scheduling algorithms for each user. Experimental results from Amazon EC2 cluster show that our scheduler reduces the average MapReduce job response times under a variety of workloads compared to the existing FIFO and Fair schedulers.**

I. Introduction

Nowadays MapReduce [1] has become an important paradigm for parallel data-intensive cluster programming due to its simplicity and flexibility. Essentially, it is a software framework that allows a cluster of computers to process a large set of data in parallel. MapReduce, namely, consists of two main steps, "map" and "reduce", where the "map" step dispatches the huge data set to individual computers in the cluster for processing and "reduce" combines the intermediate data from "map" and derives the final output.

In a MapReduce system such as Hadoop, scheduling policy is a critical factor for the performance. There are two tiers of scheduling in a Hadoop system which is shared by multiple users: Tier 1 scheduling is to assign free slots (resources) to active users; and Tier 2 scheduling is to arrange jobs for each individual user. By default, Hadoop uses a FIFO (First-In-First-Out) scheduler which ignores the tier 1 scheduling and does not perform well in practice. A small job submitted after a large job will get stuck for a long period and the system performance might be seriously degraded. The *Fair* scheduler [2] solved this problem by sharing cluster slots fairly among users (i.e., tier 1) as well as among jobs from a single user (i.e., tier 2). Consequently, each job has an equal chance to get slots and small jobs will not be stuck behind large ones. However, we observe that when job sizes of various users are not uniform, the *Fair* policy becomes inefficient. We thus argue that a good scheduler should discriminately assign slots to users in tier 1 according to their average job sizes, aiming to improve the overall performance in terms of job execution times. We also find that, in tier 2 scheduling, the *Fair* scheduler outperforms FIFO when high variance exists in job sizes of a user; otherwise, *Fair* loses its advantages.

To address the above issues, we develop a novel Hadoop scheduler, called LsPS, which aims to improve overall performance by leveraging the present job size patterns to tune its scheduling schemes among multiple users and for a single user. Specifically, we first develop a lightweight information collector that tracks important statistic information of the recently finished jobs for each user. We then propose a self-tuning scheduling policy which consists of the scheduling at two levels: the resource shares *across multiple users* are assigned based on the estimated job size of each user; and the job scheduling *for each individual user* is further adjusted to accommodate to that user's job size distribution. Experimental results in a real-world Hadoop cluster environment confirm the effectiveness and the robustness of our solution. We show that our scheduler improves the performance in terms of average job execution times under a variety of system workloads. The reminder of this paper presents our results in detail.

II. Algorithm Description

Considering the dependency between map and reduce tasks, the Hadoop scheduling can be formulated as a two-stage multi-processor flow-shop problem. However, finding the optimal solution with the minimum execution times is NP-hard [3]. Therefore, we propose LsPS, an adaptive scheduling algorithm which leverages the knowledge of workload characteristics to dynamically adjust the scheduling schemes with the goal of improving efficiency in terms of job response times in systems, especially under heavy-tailed workloads [4].

The overview of LsPS is presented in Algorithm 1. Briefly, LsPS consists of the following three components:

- Workload information collection: monitor the execution of each job, and gather the workload information.
- Scheduling among multiple users: allocate slots (both map and reduce slots) for users according to their workload characteristics, i.e., scheduling at Tier 1.
- Scheduling for a single user: tune the scheduling schemes for jobs from each user based on that user's job size distribution, i.e., scheduling at Tier 2.

In this section, we present the detailed implementation of the above three components. Table I lists some notations used

This work was partially supported by the National Science Foundation (NSF) grant CNS-1251129, an IBM Faculty Award and an AWS in Education Research Grant.

978-3-901882-50-0

in the rest of this paper.

U / u_i	number of users / i-th user, $i \in [1, U]$
J_i / $job_{i,j}$	set of all user i's jobs / j-th job of user i. $job_{i,j} \in J_i$.
$\bar{t}^m_{i,j}$ / $\bar{t}^r_{i,j}$	average map/reduce task execution time of $job_{i,j}$
$\bar{t}^m_i$ / $\bar{t}^r_i$	average map/reduce task execution time of jobs from u_i
$n^m_{i,j}$ / $n^r_{i,j}$	number of map/reduce tasks in $job_{i,j}$
$\bar{S}^*_i$	average map/reduce phase size of current jobs from u_i
$s_{i,j}$	size of $job_{i,j}$, i.e., total exe. time of map and reduce tasks
$\bar{S}_i$	average size of completed jobs from u_i
CV_i/CV'_i	coefficient of variation of completed/current job sizes of u_i
SU_i / $SJ_{i,j}$	the slot share of u_i / the slot share of $job_{i,j}$
AS_i	the slot share that u_i actually received

TABLE I
NOTATIONS USED IN THE ALGORITHM.

Algorithm 1 Overview of the LsPS scheduler

1. When a new job from user i is submitted
   a. Estimate avg. phase size $\bar{S}^*_i$ of user i using Eq. 7;
   b. Adjust slot shares among all active users, see Section II-B;
   c. Tune the job scheduling scheme for user i, see Section II-C;
2. When a task of job j from user i is finished
   a. Update the estimated average task execution time $\bar{t}^*_{i,j}$;
3. When the j-th job from user i is finished
   a. Measure avg. map/reduce task execution time $t^m_{i,j}$ / $t^r_{i,j}$ and map/reduce task number $n^m_{i,j}$ / $n^r_{i,j}$;
   b. Update history info. of user i, i.e., $\bar{t}_i, \bar{S}_i, CV_i$, using Eq.(1-6);
4. When a free slot is available
   a. Sort users in a non-increasing order of deficits $AS_i - SU_i$;
   b. Assign the slot to the first user u_{i^*}in the sorted list;
   c. Increase num. of actual received slots AS_{i^*} by 1;
   d. Choose a job from user u_{i^*} to get service based on the current scheduling scheme.

A. Workload Information Collection

In this subsection, we first introduce a light-weighted history information collector in LsPS that gathers the important statistical information of jobs and users. Basically, we collect and update the information of each job's map and reduce tasks separately by the same means. To avoid redundant description, we use the general term *task* to represent both types of tasks and the term *phase size* to represent size of either map phase or reduce phase of each job.

In LsPS, the workload information that needs to be collected for each user u_i includes average task execution time $\bar{t}^m_i$ (or $\bar{t}^r_i$), average job size $\bar{S}_i$, and the coefficient of variation CV_i of job sizes. The Welford's one-pass algorithm [5] is used to on-line update these statistics.

$$\bar{t}^m_i = \bar{t}^m_i + (\bar{t}^m_{i,j} - \bar{t}^m_i)/j, \quad (1)$$

$$\bar{t}^r_i = \bar{t}^r_i + (\bar{t}^r_{i,j} - \bar{t}^r_i)/j, \quad (2)$$

$$s_{i,j} = \bar{t}^m_{i,j} \cdot n^m_{i,j} + \bar{t}^r_{i,j} \cdot n^r_{i,j}, \quad (3)$$

$$\bar{S}_i = \bar{S}_i + (s_{i,j} - \bar{S}_i)/j, \quad (4)$$

$$v_i = v_i + (s_{i,j} - \bar{S}_i)^2 \cdot (j-1)/j, \quad (5)$$

$$CV_i = \sqrt{v_i/j}/\bar{S}_i, \quad (6)$$

where $\bar{t}^m_{i,j}$ (resp. $\bar{t}^r_{i,j}$) represents the measured average map (resp. reduce) task execution time of $job_{i,j}$, $n^m_{i,j}$ (resp. $n^r_{i,j}$) indicates the number of map (resp. reduce) tasks of $job_{i,j}$, and $s_{i,j}$ denotes the size of the j-th completed job of user u_i (i.e., $job_{i,j}$). The average job size $\bar{S}_i$ (resp. map and reduce task execution time, $\bar{t}^m_i$ and $\bar{t}^m_i$) of user u_i is set to be zero initially and updated once a new job's size information is collected.

Upon each job's completion, LsPS collects its task execution time and updates the workload statistics for the corresponding user using the above equations, i.e., Eq.(1-6). The statistical information will then be utilized by LsPS to tune the schemes for scheduling the jobs arriving in the next time period.

B. Scheduling Among Multiple Users

In this subsection, we present our algorithm for scheduling among multiple users, i.e., assigning slots to all users. Although there are two types of slots, i.e., map slots and reduce slots, in a Hadoop system, we use the same strategy to allocate them to different users. For simplicity, we present a general algorithm in the rest of this subsection which can be applied to both types of slots. Note that a job could get different slot assignments for its map and reduce tasks if they have different workload characteristics.

Specifically, LsPS adaptively adjusts the slot shares among all active users such that the share ratio is inversely proportional to the ratio of their average job phase sizes. Consequently, LsPS implicitly gives higher priority to users with smaller jobs. We expect that LsPS can avoid small jobs waiting behind large ones and thus improve the overall performance.

One critical issue that needs to be addressed is how to correctly measure the phase sizes of the jobs that are currently running or waiting for the service. In Hadoop systems, it is not easy to obtain the exact execution times of job's tasks before they are finished. In this paper, we instead estimate a job's phase size, as the production of its task number and the average task execution time. In particular, the total task number of $job_{i,j}$, i.e., $n_{i,j}$, can be obtained immediately when the job is submitted, and the execution times of tasks from the same job can be assumed to be close to each other [6]. Therefore, if $job_{i,j}$ is running, we use the average execution time of the finished tasks of $job_{i,j}$, e.g., $\bar{t}^*_{i,j}$, to represent its overall average task execution time $\bar{t}_{i,j}$. For those jobs from user u_i that are still waiting for service or currently running but have no finished tasks, we use the average task execution times $\bar{t}_i$ of jobs from the same user to approximate their average task execution times $\bar{t}_{i,j}$.

Therefore, user u_i's average map/reduce phase size of jobs is calculated as follows,

$$\bar{S}^*_i = \frac{1}{|J_i|} \cdot \sum_{j=1}^{|J_i|} n_{i,j} \cdot \bar{t}_{i,j}, \quad (7)$$

where J_i represents the set of jobs from user u_i that are currently running or waiting for service.

Once a new job arrives, LsPS updates the average size of that job's owner (see Section II-A), and then adaptively adjusts the map and the reduce slot shares (SU_i) among all active

users using Eq.8.

$$SU_i = SU_i^* \cdot (\alpha \cdot U \cdot \frac{\frac{1}{S_i^*}}{\sum_{i=1}^{U} \frac{1}{S_i^*}} + 1 - \alpha), \quad (8)$$

$$\forall i, SU_i > 0, \quad (9)$$

$$\sum_{i=1}^{U} SU_i = \sum_{i=1}^{U} SU_i^*, \quad (10)$$

where SU_i^* represents the slot shares for user u_i under the *Fair* scheme, i.e., equally dispatching the slots among users, U indicates the number of active users, and α is a tuning parameter within the range from 0 to 1. Parameter α in Eq.8 is used to control how aggressively LsPS biases towards the users with smaller job phase sizes: when α is close to 0, our scheduler increases the degree of fairness among all users, performing similar as *Fair*; and when α is increased to 1, LsPS gives the strong bias towards the users with small jobs in order to improve the efficiency. In the remainder of the paper, we set α to 1 if there is no explicit specification. We remark that through setting parameter α, one can tune LsPS to meet different predefined targets, e.g., fairness or efficiency. We also remark that it is guaranteed no active users gets starved for slots, see Eq.9, and all available slots in the system are fully distributed to active users, see Eq.10.

The new slot shares (i.e., SU_i) will then be used to determine which user can receive the slot that just became available for redistribution. Specifically, LsPS sorts all active users in a non-increasing order of their deficits, i.e., the gap between the expected assigned slots (SU_i) and the actual received slots (AS_i), and then dispatchs that particular slot to the user with the largest deficit. Additionally, some users might have high deficits but their actual demands on map/reduce slots are less than the expected shares. In such a case, LsPS re-dispatches the extra slots to those users who have less deficits but need more slots for serving their jobs.

C. Scheduling for A Single User

The second design principle used in LsPS is to dynamically tune the scheduling scheme for jobs from an individual user by leveraging the knowledge of job size distribution.

Our algorithm considers the CV of total job sizes, i.e., map size plus reduce size, of each user to determine which scheme should be used for jobs from the same user. In order to accurately estimate CV' of each user's current job sizes, we combine the history information of recently finished jobs and the estimated size distribution of current jobs that are running or waiting in the system. The past CV_i of user u_i can be provided by the information collector described in Section II-A. The current CV_i' can be calculated using Eq.(3-6) but replacing average task execution times $\bar{t}_{i,j}$ with average task execution times $\bar{t}_i$ of jobs from the same user as described in Section II-B. If both CV values of a user are smaller than 1, then the LsPS scheme schedules the current jobs from that user in the order of their submission times. Otherwise the user level scheduler fairly assigns slots among jobs. It is also possible that the two values are conflicting, i.e., $CV_i > 1$ and $CV_i' < 1$ or vise versa, which means the user's workload pattern changes. Under such case, the fair scheme is adopted to assign slots to that user's jobs. Meanwhile, the history information is reset by starting a new collection window.

III. Experimental Results in Amazon EC2

We implement and evaluate the LsPS scheduler in Amazon EC2, a cloud platform that provides pools of computing resources to developers for flexibly configuring and scaling their compute capacity on demand.

1) Experimental Setting: In our experiments, we lease a m1.large instance as master node to perform *heartbeat* and *jobtracker* routines for job scheduling. Additionally, we use another 11 m1.large instances to launch slave nodes, each of which is configured with two map slots and two reduce slots. As the Hadoop project [7] provides an API to support pluggable schedulers, we implement LsPS in Hadoop by extending the *TaskScheduler* interface and then change *mapred.jobtracker.taskScheduler* in the Hadoop configuration file to plug our scheduler into the Hadoop system.

Four representative MapReduce applications are used for scheduling evaluation, i.e., `WordCount`, `PiEstimator`, `Grep` and `Sort`. In addition, the `randomtextwriter` program is used to generate random files as the input to WordCount and Grep applications. We also run the `RandomWriter` application to generate 10G random data as the input in Sort applications. For PiEstimator applications, we set the sample space at each map task as 100 million random points, and the map task number is set to be 20.

2) Performance Evaluation: We conduct experiments with the mixed MapReduce applications, aiming to evaluate LsPS performance in a diverse environment of both CPU-bound applications, such as PiEstimator, and IO-bound applications, e.g., WordCount and Grep. In our experiments, there are four different users each of which submits a set of jobs for one type of applications above. Different distributions are introduced in both inter-arrival times and job sizes for each user, see Table II.

The experimental results are shown in Figure 1. The relative performance improvement against FIFO is also plotted in the figure. We first observe that all the users experience worst performance under FIFO when the workload is a complex mixture of demands from multiple users. The performance degradation comes mainly from the fact that the extremely large jobs from user 4, i.e., `Sort`, take over all the resources in the cluster and stuck all the following small jobs from other users. We also observe that, under the FIFO policy, all the users tend to have similar average job execution time despite of their different job size patterns. On the other hand, *Fair* could improve the performance by allowing jobs from all users to get resource shares. Therefore, small jobs gain more benefits under *Fair* policy compared with FIFO by avoiding waiting for large ones. As a result, the average execution times of the first three users, which in average submit small jobs, are improved by a factor of 2.

User	Job Type	Avg Input Size	Input Size Pattern	Job Arrival Pattern	Avg Inter-arrival	Submission Number
1	WordCount	100MB	Exponential	HeavyTail	20 sec	150
2	PiEstimator	-	-	Uniform	30 sec	100
3	Grep	500MB	HeavyTail	Exponential	100 sec	30
4	Sort	2GB	Exponential	Exponential	600 sec	5

TABLE II
EXPERIMENTAL SETTINGS FOR FOUR USERS IN AMAZON EC2.

We further observe that better performance is achieved under LsPS such that the overall performance is improved by a factor of 3.5 and 1.8 over FIFO and *Fair*, respectively. We interpret it as an outcome of setting suitable scheduling algorithms for each user based on their corresponding workload features. The largest performance improvements come from the first two users. LsPS significantly reduces the average job execution time of user 1 by assigning more resources to it. For user 2, when LsPS detects that this user keeps submitting jobs with similar sizes, it turns to schedule the jobs from this user based on their submission times because Fair scheduling now cannot achieve any benefits due to the uniform job size distribution. Moreover, compared to FIFO, the performance of user 4 is not sacrificed although the LsPS policy discriminately gives it less resources due to its large jobs.

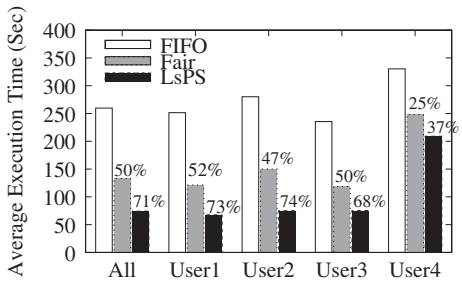

Fig. 1. Average job execution time, i.e., the duration between job submission and completion, for all users and schedulers. The relative improvements over FIFO are plotted on the bars of *Fair* and LsPS.

IV. RELATED WORKS

Scheduling in Hadoop systems has already received a lot of attention recently. An early work of Matei Zaharia et al. [2] proposed the Fair scheduler which fairly assigns resources to each user and provides performance isolation among users. However, the objective of this scheduler is not to optimize system performance. In [8], a delay scheduler was proposed to improve the performance of Fair scheduler by increasing data locality. It simply delays the task assignment for a while if the task's data is not local. This improvement is achieved at task level, and can be combined with our proposed job scheduling policy. Considering the similar direction, the quincy scheduler [9] addressed the problem of scheduling with fairness and data locality by formulating and solving a minimum flow network problem. However, computation complexity of this scheduler is high. In [10], Sandholm et. al. considered the profit of the service provider and proposed a scheduler that splits slots to users according to the bids they pay instead of fair share. The efficiency of scheduler is not considered in their work.

Another major direction of improving the Hadoop scheduling policy is considering the deadline or SLA of jobs. A deadline based scheduler was proposed in [11], which utilizes earliest deadline first policy to sort jobs and the lagrange optimization method to find out the minimum number of map and reduce slots aiming to meet the predefined deadline. This solution required a detailed profile for each job to provide execution times of map and reduce tasks. Jorda Polo et al. [6] estimated task execution times based on the average execution times of already finished tasks, and calculated the number of slots that a job needs to meet its deadline. In this paper, we partly adopt the method to aid in estimating each user's job size.

V. CONCLUSION

In this paper, we have proposed LsPS, an adaptive scheduling technique for improving the efficiency of Hadoop systems that process heterogeneous MapReduce jobs. MapReduce workloads of contemporary enterprise clients have revealed the diversity of job sizes, ranging from seconds to hours and having varying distributions as well. Our new LsPS policy online captures the present job size patterns of each user and leverages this knowledge to dynamically adjust the slot shares among all active users and to further on-the-fly tune the scheme for scheduling jobs for a single user. Experiments in Amazon EC2 have shown that LsPS consistently improves the performance in terms of job execution times under a variety of system workloads. In the future, we will extend our policy to schedule Hadoop jobs from users by taking account of the factor of priority into the share assignment.

REFERENCES

[1] J. Dean, S. Ghemawat, and G. Inc, "Mapreduce: simplified data processing on large clusters," in *OSDI'04*, 2004.

[2] M. Zaharia, D. Borthakur, J. S. Sarma *et al.*, "Job scheduling for multi-user mapreduce clusters," Univ. of California, Berkeley, Tech. Rep., 2009.

[3] B. Moseley, A. Dasgupta, R. Kumar *et al.*, "On scheduling in map-reduce and flow-shops," in *SPAA'11*, 2011.

[4] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *CCGRID'10*, 2010.

[5] B. P. Welford, "Note on a method for calculating corrected sums of squares and products," in *Technometrics*, 1962, pp. 419–420.

[6] J. Polo, D. Carrera, Y. Becerra *et al.*, "Performance-driven task co-scheduling for mapreduce environments," in *NOMS'10*, 2010.

[7] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/

[8] 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.

[9] M. Isard, Vijayan Prabhakaran, J. Currey *et al.*, "Quincy: fair scheduling for distributed computing clusters," in *SOSP'09*, 2009.

[10] Thomas Sandholm and K. Lai, "Mapreduce optimization using regulated dynamic prioritization," in *SIGMETRICS '09*, 2009, pp. 299–310.

[11] A. Verma, Ludmila Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *ICAC'11*, 2011, pp. 235–244.