New Scheduling Algorithms for Improving Performance and Resource Utilization in Hadoop YARN Clusters

Yi Yao, Han Gao, Jiayin Wang, Bo Sheng, and Ningfang Mi

Abstract—The MapReduce framework has become the defacto scheme for scalable semi-structured and un-structured data processing in recent years. The Hadoop ecosystem has evolved into its second generation, Hadoop YARN, which adopts fine-grained resource management schemes for job scheduling. Nowadays, fairness and efficiency are two main concerns in YARN resource management because resources in YARN are shared and contended by multiple applications. However, the current scheduling in YARN does not yield the optimal resource arrangement, unnecessarily causing idle resources and inefficient scheduling. It omits the dependency between tasks which is extremely crucial for the efficiency of resource utilization as well as heterogeneous job features in real application environments. We thus propose a new YARN scheduler which can effectively reduce the makespan (i.e., the total execution time) of a batch of MapReduce jobs in Hadoop YARN clusters by leveraging the information of requested resources, resource capacities and dependency between tasks. For accommodating heterogeneity in MapReduce jobs, we also extend our scheduler by further considering the job iteration information in the scheduling decisions. We implemented the new scheduling algorithm as a pluggable scheduler in YARN and evaluated it with a set of classic MapReduce benchmarks. The experimental results demonstrate that our YARN scheduler effectively reduces the makespans and improves resource utilizations.

Index Terms—MapReduce, Resource Management, YARN, Data Processing

1 Introduction

In the age of data explosion, an efficient parallel data processing scheme is essential to deal with massive volumes of data. MapReduce, proposed by Google [1], has soon emerged as a leading paradigm for big data processing due to its scalability and reliability. Its open source implementation, Apache Hadoop, has also been widely adopted in both academia and industry for big data processing and information analysis. Nowadays, the Hadoop ecosystem has evolved into its second generation, Hadoop YARN, which adopts fine-grained resource management schemes for job scheduling. When MapReduce is getting popular, fairness and efficiency become two main concerns in YARN because resources are shared and contended especially when a YARN cluster is serving a large set of jobs. However, the current scheduling in YARN does not yield the optimal resource arrangement, unnecessarily causing idle resources and inefficient scheduling. Given a limited set of resources in the cluster, when a batch of MapReduce jobs are launched, how to schedule their executions, i.e., allocating resources to jobs, becomes crucial to the performance. Without an appropriate management, the available resources may not be efficiently utilized leading to a prolonged finish time of the jobs.

This paper aims to develop efficient scheduling schemes in YARN clusters to improve resource utilization and reduce the makespan (i.e., the completion time) of a given set of jobs. The current widely adopted scheduling in YARN, such as FIFO scheduler, however, does not consider the optimal arrangement of cluster resources. For example, while it is desired to run cpu intensive jobs and memory intensive jobs simultaneously, the FIFO scheduler forces jobs to run sequentially which leads to unnecessary resource idleness. Moreover, the current resource sharing based schedulers, such as Fair and Capacity scheduler, omit the dependency between tasks. However, such dependency is crucial for the efficiency of resource utilization when we have multiple jobs running concurrently in cluster.

Therefore, in this work, we present HaSTE, a new Hadoop YARN scheduling algorithm based on task-dependency[1] and resource-demand. HaSTE aims to efficiently utilize the resources for scheduling map/reduce tasks in Hadoop YARN and improve the makespan of MapReduce jobs. Specifically, our solution dynamically schedules tasks

- *Y. Yao, H. Gao, and N. Mi are with the Department of Electrical and Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115 USA. E-mail: {yyao, hgao, ningfang}@ece.neu.edu.*
- *J. Wang is with the Department of Computer Science, Montclair State University, 1 Normal Ave, Montclair, NJ 07043 USA. E-mail: jiayin.wang@montclair.edu.*
- *B. Sheng is with the Department of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125 USA. E-mail: shengbo@cs.umb.edu.*

Manuscript received 30 Mar. 2016; revised 29 July 2018; accepted 19 Jan. 2019. Date of publication 23 Jan. 2019; date of current version 3 Sept. 2021. (Corresponding author: Han Gao.)
Recommended for acceptance by P. B. Gibbons.
Digital Object Identifier no. 10.1109/TCC.2019.2894779

1. In this work, we refer to task dependency as the data flow between phases in MapReduce.

for execution when resources become available based on each task's *fitness* and *urgency*. Fitness essentially refers to the gap between the resource demand of tasks and the residual resource capacity of nodes. This metric has been commonly considered in other resource allocation problem in the literature. The second metric, urgency, is designed to quantify the "importance" of a task in the entire process. It allows us to prioritize all the tasks from different jobs and more importantly, catches the dependency between tasks.

We further extend our new scheduling algorithm to dynamically determine the execution of tasks from multi-stage (or iterative) data processing applications. Nowadays, co-deploying multiple data processing frameworks (e.g., Spark [2], Storm [3]) in the same YARN cluster becomes a common practice. Many of these frameworks support multi-stage data processing applications. For example, MapReduce/Hadoop [1], [4] represents a typical two-stage process. Chained MapReduce jobs for SQL-on-Hadoop queries [2], [5], [6] and iterative machine learning algorithms (e.g., pagerank [7], k-means [8]) are also representative multi-stage applications. We found that without considering the iterative feature in the scheduling, the cluster resources cannot be efficiently utilized for executing iterative jobs, which thus incurs a long tail in the makespan. Therefore, we present an extended version of our new algorithm, named HaSTE-A, to further accommodate heterogeneous workloads with both iterative and non-iterative jobs. HaSTE-A differentiates iterative jobs from non-iterative ones by considering the third metric (i.e., *alignment*) to capture the number of iterations in an application and the runtime progress of iteration jobs. Coupled with fitness and urgency, HaSTE-A enforces both iterative and non-iterative jobs in a great alignment of their finished times such that the long tail in the makespan that was caused due to iterative jobs can be effectively reduced.

The rest of this paper is organized as follows. In Section 2, we briefly introduce the background of scheduling problem and existing scheduling policies in YARN. We formulate the scheduling problem of YARN system as resource constrained scheduling and propose our new scheduling policy HaSTE in Section 3. We present the extension of our scheduling algorithm to support iterative jobs in Section 4. The evaluation results of our scheduling algorithms are presented in Section 5. We describe the related works in Section 6 and conclude in Section 7.

2 Hadoop YARN Schedulers

In this section, we briefly introduce the scheduling process in a Hadoop YARN system and the schedulers that are currently used in YARN. A Hadoop YARN system consists of multiple worker nodes and the resources are managed by a centralized ResourceManager routine and multiple distributed NodeManager routines each running on a worker node. Compared to a classic Hadoop system, YARN features the following major differences in the design. First, unlike the JobTracker in Hadoop MapReduce, the ResourceManager no longer monitors the running status of each job. Instead, it launches an ApplicationMaster for each job on a worker node. Such an ApplicationMaster then generates resource requests, negotiates resources from the scheduler of ResourceManager and works with the NodeManagers to execute and monitor the corresponding job's map and reduce tasks. Furthermore, Hadoop YARN abandons the coarse-grained slot based resource management used in the old versions, but instead manages the system resources in a fine-grained manner such that each NodeManager needs to report the available memory and cpu cores of their worker node and each ApplicationMaster needs to specify the memory and cpu core demands for its tasks. The scheduler in Hadoop YARN will then allocate available resources to the waiting tasks based on a particular scheduling policy.

Each task request is a tuple $<p, \vec{r}, m, l, \gamma>$, where p represents the priority of a task, $\vec{r}$ gives the resource requirement vector of a task, m shows the total number of tasks which have the same resource requirements $\vec{r}$, l represents the location of a task's input data split, and γ is a boolean value to indicate whether a task can be assigned to a NodeManager that does not locally have its input data split. The scheduler also receives heartbeat messages from all active NodeManagers which report their current resource usage, including the capacity C and the current residual capacity R. If the current residual capacity R of a node is sufficient to accommodate at least one task and there are tasks waiting in the system, then the scheduler allocates tasks to that node according to a particular scheduling policy.

Unlike Hadoop MapReduce, YARN systems no longer explicitly distinguish map and reduce tasks such that other parallel data processing applications (such as Spark, Hive, Pig) can also be supported by YARN. In this work, we mainly focus on MapReduce applications running in YARN. Later, we show an extended solution to the problem of iterative job scheduling. The scheduling policies that are currently used in a Hadoop YARN system include FIFO, Fair, and Capacity.

- The FIFO policy sorts all waiting jobs in a nondecreasing order of their submission time. All task requests from each job will be further ordered by their priorities as well as their localities. Once ResourceManager receives a heartbeat message from a NodeManager, the first queuing task request that fits into the residual capacity of the corresponding node will be scheduled for service.
- Two Fair scheduling policies have been implemented in Hadoop YARN, i.e., Fair and Dominant Resource Fairness (DRF) [9]. The Fair policy only considers the memory usage of each job and attempts to assign equal share of memory to jobs, while the DRF policy aims to ensure all jobs to get on average an equal share on their dominant resource requirements (e.g., memory or cpu cores in the present YARN implementation).
- The Capacity policy works similar to the Fair policies. Under this policy, the scheduler attempts to reserve a guaranteed capacity for each job and orders these jobs by their deficit (i.e., the gap between a job's deserved capacity and actual occupied capacity).

Clearly, none of the above policies are designed for optimizing resource utilization and completion time of MapReduce jobs. Therefore, in this work, we design a new YARN scheduler to reduce the makespan of a batch of MapReduce jobs.

3 HaSTE

3.1 Problem Formulation

We consider that a set of n jobs $\{J_1, J_2, \ldots, J_n\}$ are submitted to a Hadoop YARN cluster consisting of m servers, $\{S_1, S_2, \ldots, S_m\}$. Each job consists of map tasks and reduce tasks. We consider all the tasks in all n jobs as a set T and assign each task a unique index number, i.e., t_i represents the ith task in the system. And then, each job J_i is represented by a set of tasks. We further define two subsets MT and RT to represent all the map tasks and reduce tasks respectively, i.e., $T = MT \cup RT$. $MT \cap J_i$ ($RT \cap J_i$) represents all the map (reduce) tasks of job J_i. In addition, assume that k types of computing resources are considered in the system, indicated by $r_1, r_2, \ldots, r_k$. Note that in the current YARN system, only two resources are included, memory and cpu. Here, we use k to define the problem with a general setting so that potential extensions can involve other types of resources, e.g., network bandwidth and disk I/O. In the rest of the paper, r_1 and r_2 represent memory and cpu resources, respectively. We use a two-dimensional matrix C to represent the resource capacity in the cluster. $C[i, j]$ indicates the amount of available resource r_j at server S_i, where $i \in [1, m]$ and $j \in [1, k]$. This matrix C is available to the scheduler after the cluster is launched and the values in C are updated during the execution of jobs upon each heartbeat message received from NodeManagers.

In Hadoop YARN, each task in a job can request user-specified resources for its execution. All map/reduce tasks share the same resource requirement. For a task $t_i \in T$, $\mathcal{R}[i, j]$ is defined to record the amount of resource r_j requested by t_i, where $\mathcal{R}[p, j] = \mathcal{R}[q, j]$ if t_p and t_q are the same type of tasks (either both map tasks or both reduce tasks) from the same job. The YARN scheduler can assign a task t_i to a work node S_j for execution as long as $\forall p \in [1, k], \mathcal{R}[i, p] \leq C[j, p]$. In this paper, given C and $\mathcal{R}$, our goal is to design an efficient scheduler that can help the cluster finish all the MapReduce jobs with the minimum time (i.e., minimize the makespan). More specifically, let st_i be the starting time of task $t_i \in T$, τ_i be the execution time of t_i. We notice that this scheduling problem is equivalent to the general resource constrained optimization problem which has been proved to be NP-complete [10].

Many heuristics have been proposed for solving the problem. Refs. [11], [12], [13] Most of them, however, are not practical to be directly implemented in the Hadoop YARN system. The main issue is that the processing time τ_i of each task t_i is required to determine the schedule in the conventional solutions. In practice, the value of τ_i cannot be known as a prior before its execution in the system. Profiling or other run time estimation techniques may be applied to roughly estimate the execution time of map tasks [14], [15]. However, it is extremely hard, if not impossible, to predict the execution times of reduce tasks in a cluster where multiple jobs could be running concurrently. In Hadoop YARN, the reduce tasks of a MapReduce job consist of two main stages, shuffle and reduce. In the shuffle stage, the output of each map task of the job is transferred to the worker nodes hosting the reduce tasks and computation in the reduce stage starts when all the input data are ready. Therefore, the execution time of a reduce task are dependent on several map-related factors, such as the execution times of all map tasks and the size of the intermediate output data. In this paper, we aim to develop a more practical heuristic that does not require any prior knowledge of task execution times.

3.2 Sketch of Our Solution HaSTE

We design a scheduler that consists of two components, *initial task assignment* (ITA) and *real-time task assignment* (RTA). ITA is executed when the cluster is just started and all ApplicationMasters have submitted the resource requests for their MapReduce tasks to the scheduler. The goal of ITA is to assign the first batch of tasks for execution while the rest of tasks remain pending in the system queue. Specifically, ITA algorithm needs to select a subset of pending tasks and select a hosting work node for each of them for execution. RTA, on the other hand, is launched during the execution of all the jobs when tasks are finished and the corresponding resources are released. When new resources become available at a worker node, the NodeManager will notify the scheduler through heartbeat messages. Then the scheduler will execute RTA to select one or more tasks from the pending queue and assign them to the worker node with new resources available. Compared to ITA, RTA is triggered by heartbeat messages with resource capacity update and only dispatches tasks to the hosting work node, i.e., the sender of the heartbeat message.

In our design, without prior knowledge of the execution time, we exploit the greedy strategy to develop both ITA and RTA algorithms. ITA is formulated as a variant knapsack problem and we use dynamic programming to derive the best task assignment in the beginning. RTA is a more complex problem involving the progress of all active tasks and the dependency between tasks. We develop an algorithm that considers *fitness* and *urgency* of tasks and determines the appropriate task to execute on-the-fly.

3.3 Initial Task Assignment

The objective of ITA is to select a set of tasks to start. Since the execution of each task is unknown, it is impossible to yield the optimal solution at this point. The information that can be leveraged by ITA only includes available resource capacity and resource demands. Therefore, we remark that the goal of the ITA algorithm is actually to avoid wasting any resources in the initial stage. To accomplish this goal, we adopt the greedy strategy and simplify our objective to be maximizing resource utilization after ITA. If there is only one type of resource, this problem is equivalent to the typical knapsack problem. Consider each worker node as a knapsack and the resource capacity refers to the knapsack capacity. Correspondingly, each task can be considered as an item and the requested resource amount is both the weight and the value of the item. The optimal solution to the converted knapsack problem will yield the maximized resource utilization in our problem setting. However, the Hadoop YARN system defines two resources (recall that we consider a general setting of k resources) in which case our problem cannot directly reduce to the knapsack problem. We thus need a quantitative means to compare different types of resources, e.g., "Is utilizing 100 percent cpu and 90 percent memory better than utilizing 90 percent cpu and 100 percent memory?". We then assume that the cluster

specifies a weight w_i for each resource r_i. The ITA problem can be formulated as follows:

$$\text{maximize: } \sum_{t_i \in T} \left(\sum_{j \in [1,m]} x_{ij} \cdot \sum_{p \in [1,k]} w_p \cdot \mathcal{R}[i,p] \right)$$
$$\text{s.t. } \sum_{j \in [1,m]} x_{ij} \leq 1, \forall t_i \in T;$$
$$\sum_{t_i \in T} x_{ij} \cdot \mathcal{R}[i,p] \leq C[j,p], \forall j \in [1,m], p \in [1,k].$$

We design an algorithm using dynamic programming to solve the problem. The details are illustrated in Algorithm 1. The main algorithm is simply a loop that assigns tasks to each of the m servers (lines 1–2). The core algorithm is implemented in the procedure $AssignTask(j,T)$, i.e., select tasks from T to assign to server S_j. We design a dynamic programming algorithm with two 2-dimensional matrices OPT and $\mathcal{L}$, where $OPT[a,b]$ is the maximum value of our objective function with a capacity $<a, b>$ and $\mathcal{L}$ records the list of tasks that yield this optimal solution. The main loops fill all the elements in OPT and $\mathcal{L}$ (lines 4-17). Eventually, the algorithm finds the optimal solution (line 18) and assigns the list of tasks to S_j (lines 19-23). When filling an element in the matrixes (lines 6-17), we enumerate all candidate tasks and based on the previously filled elements, we check: (1) if the resource capacity is sufficient to serve the task (lines 9-12); and (2) if the resulting value of the objective function is better than the current optimal value (lines 13-16). If both conditions are satisfied, we then update the matrices OPT (line 16) and $\mathcal{L}$ (line 17).

Algorithm 1. Initial Task Assignment (ITA)

```
   Data: C, T, R
   Result: x
 1 for j = 1 to m do
 2   AssignTask(j, T);
 3 Procedure AssignTask(j, T)
 4   for a = 1 to C[j,1] do
 5     for b = 1 to C[j,2] do
 6       for each t_i ∈ T do
 7         L = L[a − R[i,1], b − R[i,2]];
 8         if t_i ∈ L then Continue;
 9         if Σ_{t_p∈L} R[p,1] + R[i,1] > a then
10           Continue;
11         if Σ_{t_p∈L} R[p,2] + R[i,2] > b then
12           Continue;
13         V = w_1 · R[i,1] + w_2 · R[i,2];
14         tmp = OPT[a − R[i,1], b − R[i,2]] + V;
15         if OPT[a,b] < tmp then
16           OPT[a,b] = tmp; tmpL = L + {t_i};
17       L[a,b] = tmpL;
18   (x,y) = arg max_{a,b} OPT[a,b];
19   L = L[a,b];
20   T ← T − L;
21   for each t_i ∈ L do
22     x_ij = 1;
23   return;
```

3.4 Real-Time Task Assignment

RTA is the core component in our design of HaSTE as it is repeatedly conducted during the execution of all the jobs. The main goal of RTA is to select a set of tasks for being served on a worker node which has the newly released resources. Given the "snapshot" information only, it is difficult for the RTA algorithm to make the best decision for the global optimization, i.e., minimizing the makespan, especially considering the complexity of a MapReduce process. In this paper, we develop a novel algorithm that considers two metrics of each task, namely *fitness* and *urgency*. Our definition of *fitness* represents the resource availability in the system and resource demand from each task, while the *urgency* metric characterizes the dependency between tasks and the impact of each task's progress. In the rest of this section, we first describe the calculation of each metric and then present the overall algorithm of RTA.

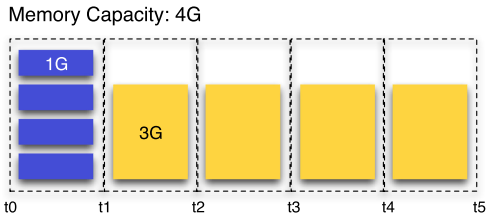
3.4.1 Fitness

Using *fitness* in our design is motivated by the greedy solution to the classic bin packing problem. We first note that some special cases of our problem are equivalent to the classic bin packing problem. Assume that all submitted jobs have only one type of tasks and all tasks are independent to each other. Also, assume that the execution times of all tasks are the same, say u time units. Our scheduling problem thus becomes packing tasks into the system for each time unit. The total resource capacity is considered as the bin size and the makespan is actually the number of bins. Thus, finding the optimal job scheduling in this setting is equivalent to minimizing the number of bins in the bin packing problem. The classic bin packing considers only one type of resource and has been proven to be NP-hard. A greedy heuristic, named First Fit Decreasing (FFD), is widely adopted to solve the problem because it is effective in practice and yields a $\frac{11}{9}OPT + 1$ worst case performance [16]. The main idea of FFD is to sort tasks in a descending order of the resource requirements and keep allocating the first fitted tasks in the sorted list to the bins. Fig. 1 illustrates how FFD can improve the makespan and the resource utilization when scheduling two jobs with different memory requirements.

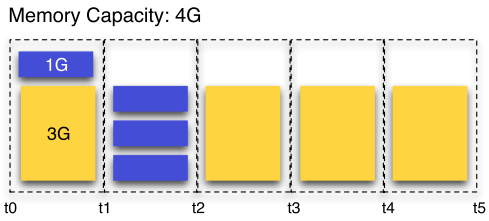
In fact, with two types of resources (memory and cpu) supported in Hadoop YARN, the simplified scheduling problem is equivalent to the vector bin packing problem in The literature [17], [18], [19] Different variants of FFD have been studied for solving the vector bin packing problem [18]. The FFD-DotProduct (dubbed as FFD-DP) has been shown to be superior under various evaluation sets. It provides relatively good performance compared with other heuristics for vector bin packing as shown in citations [18] and has negligible overhead which is important for online scheduling. Therefore, we adopt the FFD-DP method to schedule map and reduce tasks with two resource requirements. Specifically, we define *fitness* as

$$F_{ij} = \sum_{p \in [1,k]} \mathcal{R}[i,p] \cdot C[j,p] \cdot w_p. \quad (1)$$

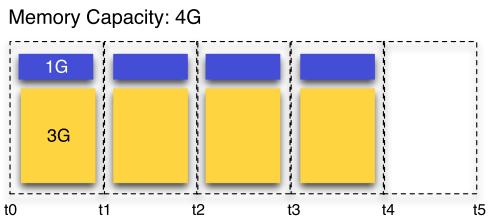
RTA uses Eq. (1) to calculate a fitness score for each pending task t_i when selecting tasks to be executed on the worker node S_j. Recall that for each resource r_p, $\mathcal{R}[i,p]$ is the requested amount from t_i, $C[j,p]$ is the resource capacity at S_j, and w_p is the weight of the resource. Intuitively, we prefer to select the task with the highest fitness score.



(a) FIFO Schedule



(b) Fair Schedule



(c) FFD Schedule

Fig. 1. Scheduling two jobs under (a) FIFO, (b) Fair and (c) FFD, where a worker node with 4G memory capacity is processing two jobs each with 4 tasks. Job 1 arrives first and each of its tasks requests 1G memory (blue blocks), while each task of Job 2 requests 3G memory, see yellow blocks. Assume that the execution time of each task is one time unit. Thus, the FFD scheduler uses 4 time units to finish both jobs while FIFO and Fair need 5 time units.

Therefore, RTA can sort all the pending tasks in the descending order of their fitness scores, and then assign the first task to the worker node S_j. After updating S_j's resource capacity, RTA will repeat this selection process to assign more tasks until there is no sufficient resource on S_j to serve any pending tasks. The FFD-DP algorithm works well with multiple resource types since it is aware of the skewness of resource requirements. For example, assume that there are two types of tasks with different resource requirements: one requests <1 GB, 3 cores$>$ and the other requests <3 GB, 1 core$>$; and RTA tries to assign tasks to a worker node with residual capacity of <1 0 GB, 6 cores$>$. The FFD-DP algorithm will choose 3 tasks of type II and 1 task of type I, which results in 100 percent resource utilization. The following table shows the fitness scores of these two types of tasks at each iteration of the algorithm.

Capacity	$<10,6>$	$<7,5>$	$<4,4>$	$<3,1>$
Type I <1 GB, 3 cores$>$	28	22	**16**	6
Type II <3 GB, 1 core$>$	**36**	**26**	16	**10**

3.4.2 Urgency

Scheduling in Hadoop YARN is more complex than the regular job scheduling problem due to the dependency between map and reduce tasks. Considering fitness alone may not always lead to good performance in practice. Although there has been previous work [20], [21], [22], [23] on job scheduling under the dependency constraints, their solutions cannot be directly applied to our problem because the dependency between map and reduce tasks is quite different from the dependency defined in [20], [21], [22], [23]. In traditional scheduling problems, a task t_j is said to be dependent on task t_i, i.e., $t_i \prec t_j$, if t_j cannot start before t_i has been completed. In the MapReduce framework, task dependency actually represents the data flow between phases, i.e., reduce tasks need to receive intermediate data from map tasks before they run. However, reduce tasks, although depend on the outputs of all map tasks, can start before the completion of all map tasks for retrieving the intermediate data from the completed map tasks. This *early* start is configured by a system parameter "slowstart" and renders a better performance in practice.

Consequently, the execution of reduce tasks are highly dependent on the execution of map tasks. Indeed, such dependency relationship has been known by ApplicationMasters when making reduce task requirements. A new metric, named "Ideal Reduce Memory Limit", is calculated as the product of the progress of map tasks and the total "available" memory for the corresponding job. The resource limit of reduce tasks increases gradually with the progress of map tasks. An ApplicationMaster sends new reduce task requests to the ResourceManager only when the present resource limit is enough for running more reduce tasks.

However, we observed that the current schedulers in Hadoop YARN, which are designed for more general task scheduling, fail to recognize the impact of dependency in MapReduce jobs and may lead to ineffective resource assignments and poor performance as well. For example, a job that has already launched many reduce tasks may not be able to have all its map tasks to be executed right away due to resource contention among other jobs; the launched reduce tasks will keep occupying the resources when waiting for the completion of all maps tasks of the same job. This incurs low utilization of resources that are allocated to those reduce tasks.

To address the above issue, HaSTE uses a new metric, named "urgency", to capture the performance impact caused by the dependency between map and reduce tasks of MapReduce jobs. Specifically, we have the following main scheduling rules associated with the *urgency*.

- R1: A job with more progress in its map phase, will be more urgent to schedule its map tasks. This rule can boost the completion of the entire map phase and further reduce the execution time of the launched reduce tasks.
- R2: A job with more resources allocated to its running reduce tasks will be more urgent to schedule its map tasks in order to avoid low resource utilization when its reduce tasks are waiting for the completion of map tasks.
- R3: Reduce tasks should be more urgent than map tasks of the same job if the ratio between resources occupied by currently running reduces and all currently running tasks is lower than the progress of map phase, vice versa.

In summary, R1 and R2 are used to compare the urgency between two different jobs while the urgency of map/reduce tasks from the same job is compared by R3. We calculate the map task urgency score (U_i^m) and reduce task urgency score (U_i^r) for job i as follows:

$$U_i^m = \frac{A_i^m}{T_i^m} \cdot (A_i^r \cdot R_i^r + A_i^{am} \cdot R_i^{am}), \tag{2}$$

$$U_i^r = U_i^m \cdot \frac{A_i^m}{T_i^m} \cdot \frac{O_i^m \cdot R_i^m + O_i^r \cdot R_i^r}{O_i^r \cdot R_i^r}. \tag{3}$$

Here, $A_i^m/A_i^r/A_i^{am}$ represents the number of map/reduce/ApplicationMaster tasks that have been assigned for job i, and $R_i^m/R_i^r/R_i^{am}$ represents the resource requirement of a single map/reduce/ApplicationMaster task, i.e., the weighted summation of memory and cpu requirements. T_i^m represents the total number of map tasks of job i. O_i^m/O_i^r represents the number of running map/reduce tasks of job i that are currently occupying system resources. All these metrics are accessible to the scheduler in the current YARN system. Therefore, we implemented our new scheduler as a pluggable component to YARN without any needs of changing other components.

3.4.3 HaSTE Scheduler

Now, we turn to summarize the design of HaSTE by integrating the two new metrics, i.e., fitness and urgency, into the scheduling decision.

Once a node update message is received from a NodeManager, the scheduler first creates a list of all resource requests that can fit the remaining resource capacity of that node. Meanwhile, the scheduler calculates the fitness and urgency scores of those chosen resource requests, and obtains the preference score for each request by summating the normalized fitness and urgency scores, see Eq. (4)

$$P_i = \frac{F_i - F_{min}}{F_{max} - F_{min}} + \frac{U_i - U_{min}}{U_{max} - U_{min}}, \tag{4}$$

where F_{max} and F_{min} (resp. U_{max} and U_{min}) record the maximum and minimum fitness (resp. urgency) scores among these requests.

Such preference scores are then used to sort all resource requests in the list. The resource request with the highest score will be chosen for being served. Note that each resource request can actually represent a set of task requests since tasks with the same type and from the same job usually have the same resource requirements. The scheduler will then choose a task that has the best locality (i.e., node local or rack local) and assign that task to the NodeManager. One special type of task request is the request for ApplicationMaster. Such requests always have the highest preference score in HaSTE due to its special functionality, i.e., submitting resource requirements and coordinating the execution of a job's tasks.

Finally, we remark that the complexity of our scheduling algorithm is $O(n \log n)$ which is determined by the sorting process. Here n is the number of running jobs rather than the number of running tasks since all tasks with the same type and from the same job could be represented in a single resource request and then have the same preference score. Therefore, HaSTE is a light-weighted and practical scheduler for the Hadoop YARN system.

4 HaSTE-A

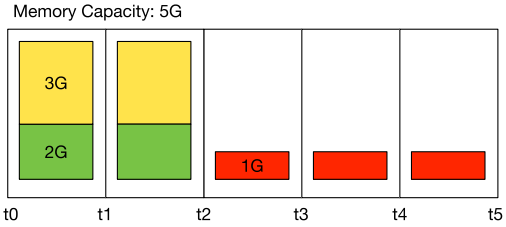
With the growth of applications in YARN systems, more and more iterative algorithms are adopted for the MapReduce paradigm. For example, the k-means algorithm [24] can be modeled as a set of identical MapReduce jobs such that each job's execution represents one iteration of the algorithm. Pagerank [7] is another example of iterative algorithms, which has multiple stages in each iteration and also needs to instantiate a sequence of jobs for each iteration. The iterative feature of these algorithms determines that a single round of the map-reduce procedure is not enough for processing data. Thus, these applications often submit more than one jobs to the YARN cluster. The number of jobs for an application depends on the number of its stages as well as its input dataset. For example, the stop condition for k-means is controlled by either the pre-defined maximum number of iterations or the pre-defined convergence threshold.

We observe that without considering the iterative feature, the current scheduling (even including HaSTE) cannot work well under the workloads with iterative applications. Two limitations can be found under those scheduling algorithms: (1) a long tail appears in the makespan due to the delayed execution of iterative algorithms, and (2) cluster resources (e.g., memory and cpu cores) cannot be fully utilized during the execution of those delayed iterative algorithms. In Fig. 2a, we provide a motivation example to illustrate the impact of iterative jobs on the scheduling performance. Assume that there are three jobs with different task numbers and resource requirements: Job 1 and Job 2 are non-iterative ones with two tasks each (see yellow and green blocks) while Job 3 is an iterative job with three tasks which need to be executed sequentially (see red blocks). The memory requirement for each job is labeled as well in the figure. One possible scheduling result under HaSTE is shown in Fig. 2a, where tasks in Job 1 and Job 2 fill the capacity first due to their large memory requirements and Job 3 can only start at the third time unit. In this case, we observe a long tail since the third time unit, which leads to low memory utilization and a long makespan.

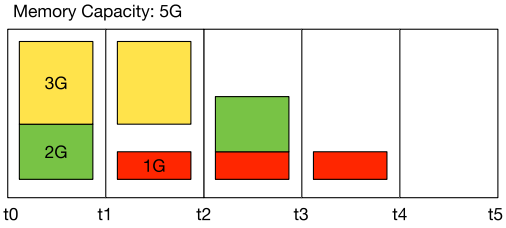
4.1 Alignment

In HaSTE, *fitness* represents the matching degree between resource requirement and current available capacity in the cluster, while *urgency* reflects the dependency between map and reduce tasks in one job and the relative rate among different jobs. To further identify the distinct nature of iterative applications, we introduce a new metric, called *alignment*, to capture the runtime process of iterative applications. Another three rules associated with the alignment metric are then defined as follows.

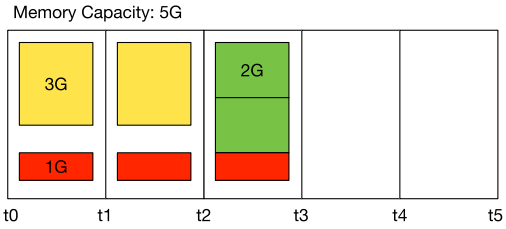
- R4. An iterative job should have a higher alignment score in order to run its map/reduce tasks earlier than other non-iterative jobs. This rule helps to align the processing of both non-iterative and iterative jobs and thus remove the long tail in the makespan that is caused by iterative ones.



(a) HaSTE with iterative job



(b) HaSTE-A less aggresive



(c) HaSTE-A more aggresive

Fig. 2. Scheduling three jobs with and without the alignment score. Assume that the execution time of each task is one time unit and cpu resouce is sufficient here. (a) HaSTE scheduler schedules Job 1 and Job 2 first based on the fitness and urgency scores. (b)(c) HaSTE-A further uses the alignment score to start Job 3 one or two time unit earlier. Thus, HaSTE scheduler uses 5 time units to finish all jobs while only cost three or four time units.

- R5. An iterative job with more stages should have a higher alignment score than other iterative jobs in order to shrink the intermediate lagging time among its stages. This rule can reduce the response time for such multi-stage jobs and also avoid low resource utilization at the end of the entire processing.
- R6. More resources should be allocated to jobs with the higher ratio between the number of finished stages and the number of total stages, vice versa. This rule can accelerate those jobs which are approaching the end of their execution.

R4 differentiates two types (i.e., iterative and regular) of jobs while R5 describes the relation between two iterative jobs. R6 further considers the dynamic runtime process of each job. In summary of these three rules, we set up a heuristic equation (Eq. (5)) for both map and reduce task's alignment $A_i^{m/r}$

$$A_i^{m/r} = \frac{I_i + I_i^{current}}{\sum_{j=1}^{m} I_j}. \tag{5}$$

Here, I_i represents the total number of stages or iterations[2] for job i and $I_i^{current}$ represents the current number of finished stages or iterations. We can see that the alignment score captures the iteration feature as well as the runtime process of these iterative jobs. Later, in Section 5.2.3, we show that the alignment score of iterative jobs increases across the runtime.

We further use our motivation example to illustrate how the alignment metric affects the task scheduling under a certain memory capacity in Figs. 2b and 2c. Under the consideration of alignment, our scheduler can schedule the iterative job (e.g., Job 3) earlier by starting that job's first task at the second unit time (see plot (b) in Fig. 2) or even more aggressively at the first time unit (see plot (c) in Fig. 2). As a result, the iterative job runs in parallel with the non-iterative jobs. The makespan is thus reduced by two time units and the memory resource is fully utilized under the aggressive way.

Another target in our design of alignment is to improve the average job response time. We define a job's response time from its submission to its finish. For example, the new scheduler using aggressive alignment can decrease the average response time of three jobs from 3 (see Fig. 2a) to 2.67 (see Fig. 2c) time units although Job 2's response time is increased.

4.2 HaSTE-A Scheduler

Now, we add *alignment* as the third part of the preference score and introduce the factor $\vec{\beta} = \{\beta_1, \beta_2, \beta_3\}$ to adjust the weights of each part of the preference score. The preference score for each request can be redefined by summating the normalized fitness, urgency, and alignment scores, see Eq. (6)

$$\begin{aligned} P_i = \beta_1 \cdot \frac{F_i - F_{min}}{F_{max} - F_{min}} + \beta_2 \cdot \frac{U_i - U_{min}}{U_{max} - U_{min}} \\ + \beta_3 \cdot \frac{A_i - A_{min}}{A_{max} - A_{min}}, \end{aligned} \tag{6}$$

where A_{max} and A_{min} record the maximum and minimum alignment scores among these requests. The value of each β_i can be pre-defined based on the proportion of iterative jobs in the cluster and how aggressive the user wants to execute iterative jobs. Intuitively, when we have a few iterative jobs simultaneously running with other non-iterative ones, a larger value will be used for β_3 (i.e., $\beta_3 > \beta_1$ and $\beta_3 > \beta_2$) such that HaSTE-A can aggressively accelerate the processing of those iterative jobs. However, if the majority of jobs are iterative, we can actually ignore the third factor in the preference score, i.e., setting β_3 as a very small value. HaSTE-A then simply treats all jobs as the same type and schedules them based on fitness and urgency only as HaSTE does.

5 Evaluation

In this section, we evaluate the performance of HaSTE and HaSTE-A by conducing experiments in a Hadoop YARN cluster. We implemented HaSTE, HaSTE-A and FFD-Dot-Product (abbrev. FFD-DP) schedulers in Hadoop YARN version 2.2.0 and compared them with the built-in schedulers (i.e., FIFO, Fair, Capacity, and DRF). The performance metrics considered in the evaluation include makespans of a batch of MapReduce jobs and resource usage of the Hadoop YARN cluster. For HaSTE-A, average response time is additional metric we considered.

2. In this work, we assume that there is a priori knowledge of the number of iterations for an iterative application. How to predict the number of iterations is out of this paper's scope and will be considered in our future work.

TABLE 1
Benchmark Descriptions

Benchmark	Description
WordMean	Calculate the mean length of words in the input data.
WordCount	Count the occurrence of each word in the input data, which are generated using RandomTextWriter.
Terasort	A popular benchmark to sort one terabyte of randomly distributed data.
PiEstimate	Estimate the value of π.
Pagerank	Pagerank is a link analysis and web ranking algorithm.
Kmeans	Kmeans is a clustering analysis algorithm for multi-dimensional numerical samples in data mining.
Scan	Scan is a SQL benchmark that creates two external tables and inserts the second table into the first one.
Dfsioe	Test the HDFS throughput and I/O rate of tasks by performing writes and reads simultaneously.

5.1 Resource Requests of MapReduce Jobs

In our experiments, we consider different resource requirements such that a job can be either memory intensive or cpu intensive. The resource requirements of map and reduce tasks of a MapReduce job can be specified by the user when that job is submitted. The user should set the resource requirements equal to or slightly more than the actual resource demands. Otherwise, a task will be killed if it needs more resources than its required resource amount.[3] Such a mechanism adopted in the YARN system can prevent malicious users from faking the resource requirements and thus from thrashing the system. On the other hand, it is not proper either to request much more than the actual demands. In such a case, the concurrency level of MapReduce jobs and the actual resource usage will be reduced and the performance will be degraded as well. We note that how to set appropriate resource requirements for each job is out of this paper's scope. In our experiments, we vary the resource requirements for different jobs in order to evaluate the schedulers under various resource requirements but keep the resource requirements configuration the same under different scheduling algorithms.

5.2 Batch Job Experiment Results

Here, we conduct three sets of experiments in a Hadoop YARN cluster with 8 nodes, each of which is configured with the capacity of 8 GB memory and 8 virtual cpu cores, i.e., $<$8G, 8cores$>$. The benchmarks we use in these experiments are summarized in Table 1.

5.2.1 Simple Workload

In the first set of experiments, we consider a simple workload which consists of four *Wordcount* jobs. Each job in this workload parses the same 3.5G wiki category links input file. Therefore, all the four jobs have the same number of map and reduce tasks. The map task number is determined by the input file size and the HDFS block size which is set to 64 MB in this experiment. As described in Section 5.1, for different jobs, we vary the resource requirements on a single type of resource for analyzing the impact of resource requirements on the scheduling performance. The configurations of each job and their resource requirements are shown in Table 2.

Fig. 3 shows the makespans and the average resource (mem and cpu) usage under different scheduling policies. Here, the memory/cpu usage is defined as the average amount of resources that are allocated for all running tasks during a specific time period. We observe that all the conventional schedulers (i.e., FIFO, Fair, and DRF) cannot efficiently utilize the system resources, e.g., under 60 percent cpu core usage and under 30 percent memory usage. Although these conventional schedulers obtain similar resource usage, FIFO outperforms Fair by 23.8 percent and DRF by 29.3 percent. That is because under Fair and DRF when multiple jobs are running concurrently in the cluster, their reduce tasks are launched and thus occupy most of the resources, which may dramatically delay the execution of map phases. Similarly, the makespan under the FFD-DP scheduling policy is 10 percent larger than under FIFO, although FFD-DP achieves the highest resource usage, e.g., 86.6 percent cpu cores usage in average. While, the new scheduler HaSTE solves this problem by considering the impacts of both resource requirements (i.e., fitness) and dependency between tasks (i.e., urgency) and thus achieves the best makespan, which is, for example, 27 and 44.6 percent shorter than FIFO and Fair, respectively.

5.2.2 Mixed Workload Case 1

To further validate the effectiveness of HaSTE, we conduct a more complex workload which is mixed with both cpu intensive and memory intensive MapReduce jobs. Table 3 shows the detailed workload configuration, where the input data for *Terasort* is generated through the *Teragen* benchmark, and the input for *Wordcount* and *Wordmean* is the wiki category links data. In this set of experiments, we set the HDFS block size equal to 128 MB.

Fig. 4 plots the makespans and the average resource usage under this mixed workload. Consistently, the three

TABLE 2
Simple Workload Configuration

Job ID	#Map	#Reduce	R^m	R^r
1	52	5	$<$1G, 2 cores$>$	$<$1G, 2 cores$>$
2	52	5	$<$1G, 3 cores$>$	$<$1G, 2 cores$>$
3	52	5	$<$1G, 4 cores$>$	$<$1G, 3 cores$>$
4	52	5	$<$1G, 5 cores$>$	$<$1G, 3 cores$>$

3. We note that the virtual cpu cores are not physically isolated from each task in the YARN system. While the number of virtual cpu cores requested for a task determines the priority of that task when competing for cpu times. Therefore, an inappropriate low request of virtual cpu cores is also not desired because it may lead to insufficient cpu times that a task can get and dramatically delay the execution of that task.

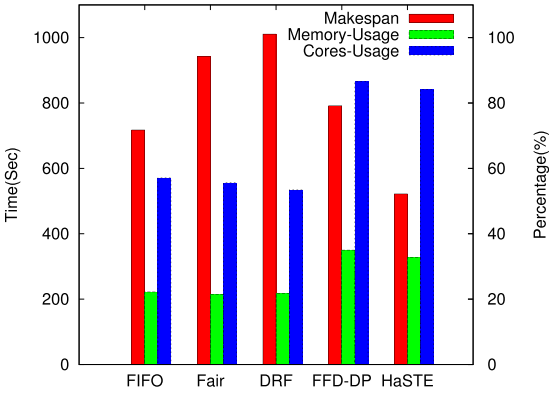


Fig. 3. Makespans and average resource usage under the workload of 4 *Wordcount* jobs. The left y-axis shows the makespans (sec.) while the right y-axis shows the cpu and memory resource usage (%).

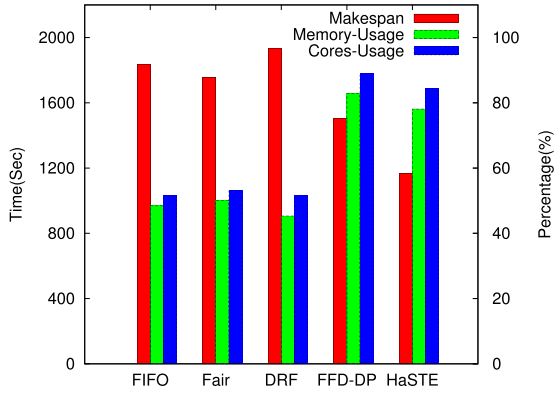


Fig. 4. Makespans and average resource usage under the mixed workload of four benchmarks. The left y-axis shows the makespans (sec.) while the right y-axis shows the cpu and memory resource usage (%).

conventional scheduling policies have similar average resource usage, e.g., around 50 percent for both cpu and memory. However, in this experiment, jobs experience similar makespans under the Fair and DRF policies as well as under FIFO. We interpret this by observing that the ApplicationMasters killed the running reduce tasks to prevent the starvation of map tasks when these reduce tasks occupy too many resources. On the other hand, both FFD-DP and HaSTE increase the average resource usage, e.g., to around 80 percent, through the resource-aware task assignment. FFD-DP also improves the makespan by 18.1 and 14.8 percent compared to FIFO and Fair, respectively. HaSTE further improves the performance in terms of makespan by 36.3 and 33.9 percent compared to FIFO and Fair, respectively.

To better understand how these scheduling policies work, we further plot the runtime memory allocations in Fig. 5. We observe that the precedence constraint of FIFO and the fairness constraint of Fair and DRF can both lead to inefficient resource allocation in the Hadoop YARN cluster. For example, when cpu intensive jobs are running under the FIFO policy, see jobs 3, 4, 6, 7 in Fig. 5a, the scheduler cannot co-schedule memory intensive jobs at the same time, and a large amount of memory resources in the cluster are idle for a long period. While, under the Fair and DRF policies, although all jobs share the resources, the fairness constraint, i.e., all jobs should get equal shares on average, in fact, hinders the efficient resource usage. For example, when a node has $<1\,\text{GB}, 4\,cores>$ available resources and two tasks t_1 and t_2 with $R_1 = <1\,GB, 4\,cores>$ and $R_2 = <1\,GB, 1\,core>$ are waiting for service, Fair may assign resources to t_2 if this tasks now deserves more share of resources, which will lead to a waste of 3 cpu cores on the node. We also observe that by tuning the resource shares among different jobs, the FFD-DP policy could achieve better resource usage across time. More importantly, HaSTE also achieves high or even slightly higher resource usage across time. This is because HaSTE allows jobs whose resource requirements can better fit the available resource capacities to have higher chance to get resources and thus improves the resource usage.

In summary, HaSTE achieves non-negligible improvements in terms of makespans and resource usage when the MapReduce jobs have various resource requirements. By leveraging the information of job resource requirements and cluster resource capacities, HaSTE is able to efficiently schedule map/reduce tasks and thus improve the system resource usage. In addition, the makespans of MapReduce jobs are further improved by taking the dependency between map and reduce tasks into consideration when multiple jobs are competing for resources in the YARN cluster.

5.2.3 Mixed Workload Case 2

We also conduct a mixed workload which consists of both iterative and non-iterative jobs to further evaluate the effectiveness of our HaSTE-A scheduler with the alignment metric in the scheduling. Table 4 summarizes the parameter configuration for each job in this workload. Specifically, we generate five jobs such that three of them are non-iterative MapReduce jobs (e.g., *WordCount*, *Terasort* and *Scan*) and the remaining two are iterative jobs such as *Pagerank* and *Kmeans*.

TABLE 3
Mixed Workload Case 1 Configuration

Job Type	Job ID	Input Size	#Map	#Reduce	R^m	R^r
Terasort	1	5 GB	38	6	$<3\,\text{GB}, 1\,core>$	$<2\,\text{GB}, 1\,core>$
	2	10 GB	76	12	$<4\,\text{GB}, 1\,core>$	$<2\,\text{GB}, 1\,core>$
WordCount	3	7 GB	52	12	$<2\,\text{GB}, 3\,cores>$	$<1\,\text{GB}, 2\,cores>$
	4	3.5 GB	26	6	$<2\,\text{GB}, 4\,cores>$	$<1\,\text{GB}, 2\,cores>$
WordMean	5	7 GB	52	8	$<2\,\text{GB}, 2\,cores>$	$<1\,\text{GB}, 1\,core>$
	6	3.5 GB	26	4	$<2\,\text{GB}, 1\,core>$	$<1\,\text{GB}, 1\,core>$
PiEstimate	7	-	50	1	$<1\,\text{GB}, 3\,cores>$	$<1\,\text{GB}, 1\,core>$
	8	-	100	1	$<1\,\text{GB}, 4\,cores>$	$<1\,\text{GB}, 1\,core>$

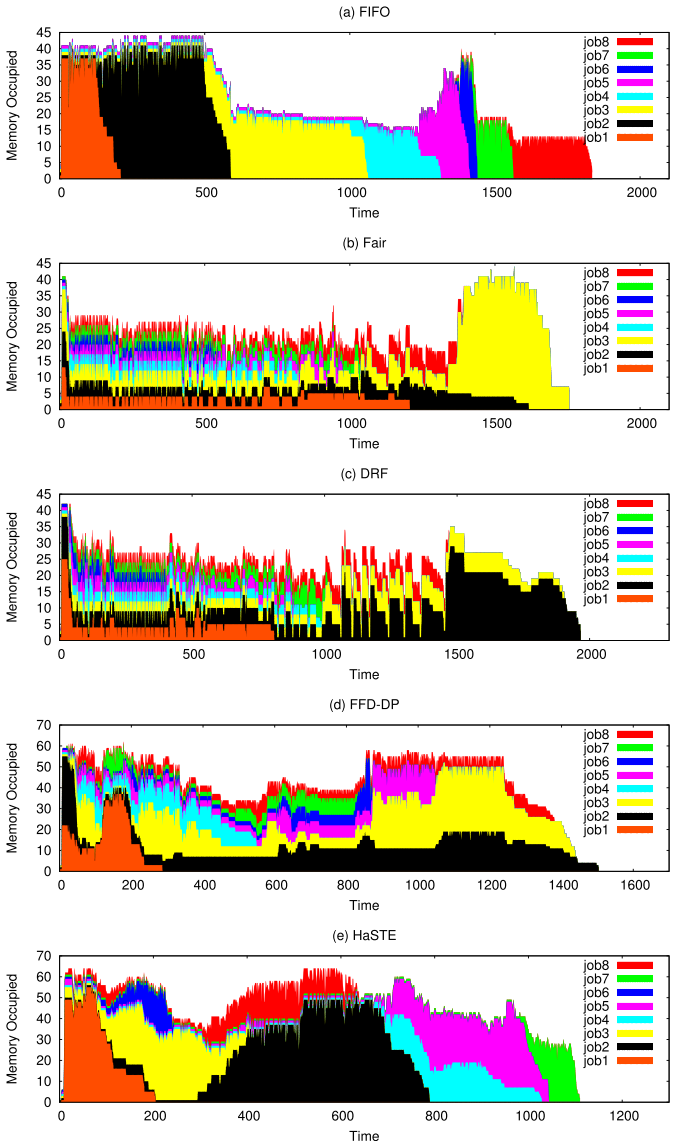


Fig. 5. Illustrating the memory resources that have been allocated to each job cross time under different scheduling policies.

In this set of experiments, we consider the makespan, the average job response time, and the average memory usage as the performance metrics to compare different scheduling policies. Fig. 6 shows the experimental results under the existing schedulers (i.e., FIFO and Fair, FFD-DP) and our new schedulers (i.e., HaSTE and HaSTE-A). Here, we set the factor $\overrightarrow{\beta}$ used in HaSTE-A as $\{0.2, 0.2, 0.6\}$ such that the preference score under HaSTE-A ranges from 0 to 1.0.

As shown in Fig. 6, the worst performance is found under the Fair scheduler. By considering the resource capacity and

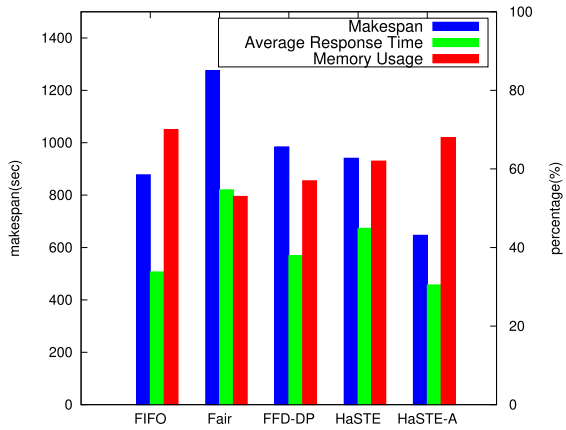


Fig. 6. Makespans, average response times and average memory usage under the mixed workload case 2 including three non-interative and two iterative benchmarks. The left y-axis shows the makespans (sec.) and the average response times (sec.), while the right y-axis shows the memory resource usage (%).

the dependency between tasks, HaSTE can reduce the makespan as well as the average response time by 26.7 and 17.9 percent, respectively, and increase the memory usage by 15 percent. However, we observe that such an improvement under HaSTE is not as significant as that under the workload with non-iterative jobs (see Section 5.2.2) and even diminishes compared to FIFO and FFD-DP. We interpret it by observing that HaSTE does not differentiate the iterative jobs by assigning them with high preference scores. Consequently, the non-iterative jobs (e.g., *Terasort* and *Wordcount*) that have higher fitness scores occupy the resources and even keep the resources because of their increasing urgency scores.

We also observe that HaSTE-A overcomes the limitation of HaSTE by further integrating *alignment* in the preference score and thus improves the performance for the workload with both non-iterative and iterative jobs. For example, the makespan under HaSTE-A is reduced by 26.4, 49.3, and 34.3 percent compared to FIFO, Fair, and FFD-DP, respectively. Additionally, the average response time is reduced as well by 9.1, 44.3 and 19.5 percent, respectively. These results demonstrate that HaSTE-A can effectively shorten the total execution time of a batch of jobs by boosting the scheduling of iterative jobs and meanwhile does not sacrifice the average performance, e.g., average job response time.

Fig. 7 depicts the amount of memory allocated to each of five jobs across time under different scheduling policies. We can see that after about 600 seconds, the memory usage drops to 50 percent (i.e., 30 GB out of 64 GB total capacity) and even 0 percent periodically under the FIFO, FFD-DP and HaSTE policies. Such a low memory usage is caused due to the delayed scheduling of iterative jobs (e.g.,

TABLE 4
Mixed Workload Case 2 Configuration

Job Type	#Iteration	Input Size	#Map[a]	#Reduce[a]	R^m	R^r
Terasort	1	5 GB	96	12	$<3\,\text{GB}, 1\,core>$	$<2\,\text{GB}, 1\,core>$
WordCount	1	10 GB	81	12	$<1\,\text{GB}, 3\,cores>$	$<1\,\text{GB}, 2\,cores>$
Scan	1	2.4 GB	20	6	$<1\,\text{GB}, 1\,cores>$	$<1\,\text{GB}, 1\,core>$
Pagerank	2	240 MB	37	12	$<2\,\text{GB}, 2\,cores>$	$<2\,\text{GB}, 2\,core>$
Kmeans	6	2 GB	120	5	$<2\,\text{GB}, 3\,cores>$	$<1\,\text{GB}, 3\,core>$

[a]*The map and reduce number of Pagerank and Kmeans are the sum of all iterations.*

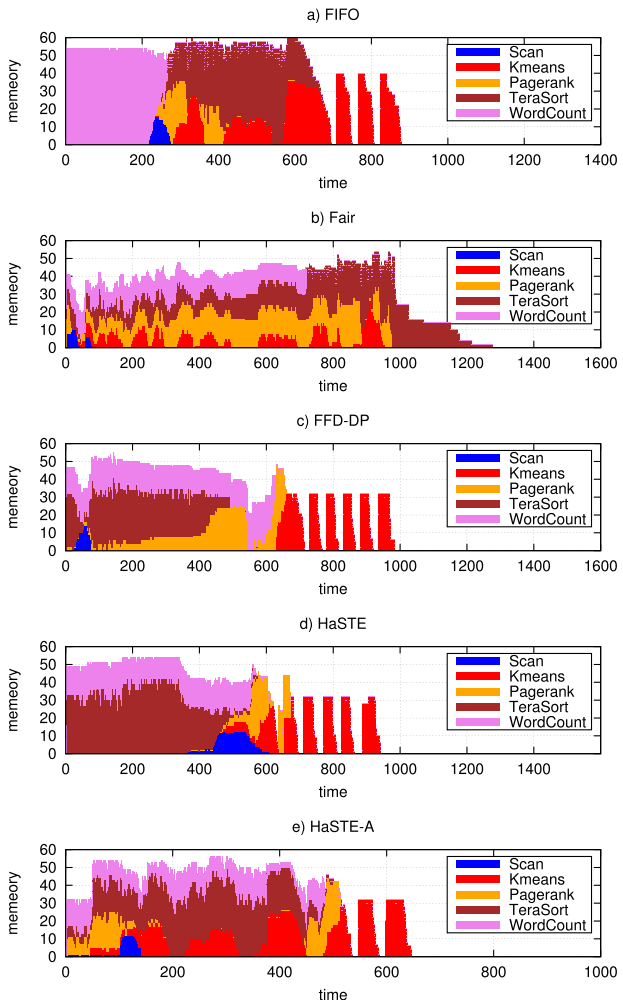


Fig. 7. Memory resource allocations for each job under different scheduling polices.

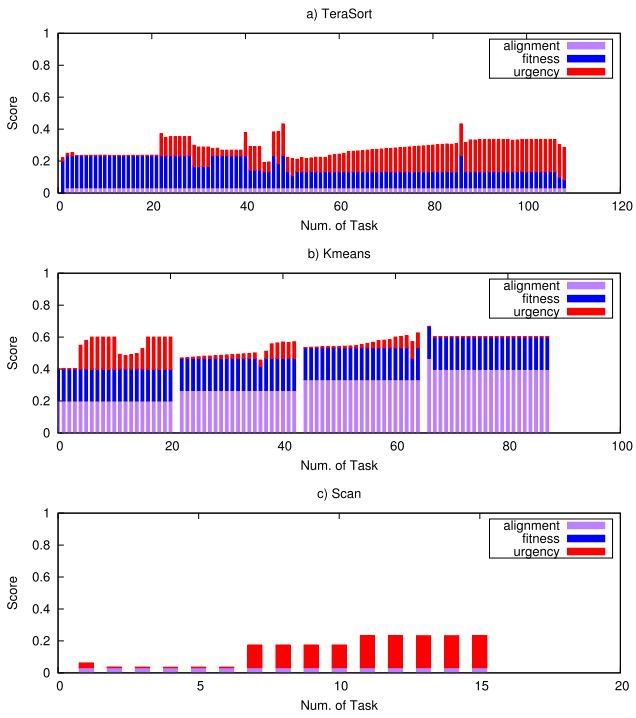


Fig. 8. The runtime scores of fitness, urgency and alignment scores for (a) *Terasort*, (b) *Kmeans* and (c) *Scan* under the scheduling policy HaSTE-A.

Kmeans) which run alone at the end of the overall processing in order to complete their iterations and thus lag the total completion time of the batch of five jobs. We also look closely at the preference scores of each job under HaSTE. We find that HaSTE treats iterative jobs (such as *Pagerank* and *Kmeans*) as non-iterative ones, neglecting their iteration feature and assigning them with a low urgency score.

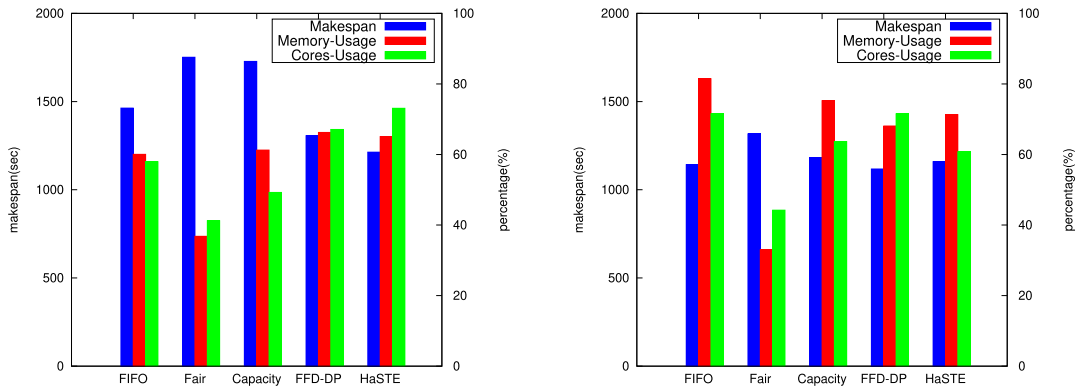
In contrast, our HaSTE-A scheduler makes its scheduling decisions using the combination of three factors (i.e., fitness, urgency and alignment). As shown in Fig. 7e, the resource requirements of tasks from the *WordCount* and *TeroSort* jobs best fit in the resource capacity and are thus scheduled first because of their high fitness scores. Moreover, HaSTE-A gives high aligment scores to two iterative jobs (i.e., *Pagerank* and *Kmeans*) such that these two jobs can get their required resources earlier and run their iterations in parallel with other non-iterative ones. As a result, HaSTE-A has been shown to be superior under the mixed workload with iterative jobs and achieve the best performance with the shortest makespan and the highest resource usage.

Fig. 8 further illustrates how the scores of fitness, urgency and alignment change across time for three representative jobs, i.e., *Terasort*, *Kmeans*, and *Scan*. Consistent with our discussion above, *Terasort* receives a high fitness score while the alignment score of *Kmeans* dominates this job's preference score across time and allows HaSTE-A to start the execution of its tasks earlier than that under the other scheduling algorithms. Moreover, tasks from *Scan* have the low resource (i.e., cpu and memory) requirements and thus receive a low fitness score across time. As a result, the *Scan* job is not able to obtain its required resources before 110 seconds (see the blue area in Fig. 7e). However, as the time passes, the urgency score of this job increases with the increasing of A_i^m and A_i^r (i.e., the number of map/reduce tasks that have been assigned for job i), see Eq. (3). The high urgency score then allows this *Scan* job to receive the resources and finish at 150 seconds.

To make a sum, HaSTE-A achieves the best makespan and average response time when the workload contains iterative jobs. The alignment metric, as the third component in our preference score, significantly overcomes the limitation of HaSTE by scheduling the iterative jobs early and aligning the execution of these iterative jobs with non-iterative ones.

5.3 Successive Job Experiment Results

In the previous experiments, we considered an extreme case that a batch of jobs arrives at the same time. Although this case is not common, it is difficult because all jobs compete for system resources simultaneously. Now, we further investigate a more general case that an open arrival process is used to generate and launch MapReduce jobs from different applications. In this set of experiments, we consider the successive job submission pattern in a heavy-loaded Hadoop YARN cluster. Fig. 9 shows the experimental results in terms of makespan and average memory/cpu resource usage under different scheduling schemems. Specifically, jobs are submitted with a random interval time between 0 to 60 seconds and 60 to 120 seconds, as shown in Figs. 9a and 9b, respectively. We further evaluate the



(a) Submit job successively with 0-60 seconds random interval

(b) Submit job successively with within 60-120 seconds random interval

Fig. 9. Makespans and average resource usage under the mixed workload case 2 with successive job submission pattern. The left y-axis shows the makespans (sec.) while the right y-axis shows the cpu and memory resource usage (%).

TABLE 5
Mixed Workload Case 3 Configuration

Job Type	Job Id	#Iteration	Input Size	#Map[a]	#Reduce[a]	R^m	R^r
Terasort	1	1	10 GB	24	6	$<3\,\text{GB}, 1\,core>$	$<4\,\text{GB}, 2\,core>$
	2	1	20 GB	96	6	$<5\,\text{GB}, 2\,core>$	$<6\,\text{GB}, 4\,core>$
WordCount	2	1	9 GB	36	12	$<2\,\text{GB}, 3\,cores>$	$<1\,\text{GB}, 3\,cores>$
	4	1	18 GB	162	12	$<4\,\text{GB}, 4\,cores>$	$<2\,\text{GB}, 5\,cores>$
Dfsioe	5	1	2.5 GB	64	1	$<1\,\text{GB}, 1\,cores>$	$<1\,\text{GB}, 3\,core>$
	6	1	4 GB	64	1	$<3\,\text{GB}, 3\,cores>$	$<2\,\text{GB}, 4\,core>$
Pagerank	7	2	0.5 GB	37	24	$<1\,\text{GB}, 3\,cores>$	$<2\,\text{GB}, 2\,core>$
	8	2	1 GB	37	24	$<2\,\text{GB}, 5\,cores>$	$<3\,\text{GB}, 4\,core>$
Kmeans	9	4	2 GB	16	3	$<2\,\text{GB}, 2\,cores>$	$<1\,\text{GB}, 3\,core>$
	10	5	10 GB	100	4	$<4\,\text{GB}, 4\,cores>$	$<2\,\text{GB}, 4\,core>$

[a]*The map and reduce number of Pagerank and Kmeans are the sum of all iterations.*

performance of the Capacity scheduler with a default configuration (i.e., all jobs are in the same queue) for comparison.

As shown in Fig. 9a, we can observe that HaSTE achieves the best performance (i.e., the shortest makespan) among all considered schedulers. Meanwhile, we notice that the Capacity scheduler with the default configuration has similar performance as Fair. The batch of jobs experience a long makespan under these two schedulers because both of them focus on fairness when allocating resources among jobs. As expected, when the interarrival time between jobs is longer, the performance improvement of HaSTE becomes less visible, see Fig. 9b. This is because resources competition is less intensive under this case even with the same workload and thus efficient scheduling algorithms becomes not critical. For example, when we increase the submission interval to 60-120 seconds, only one or two jobs run together during the most of execution period of time. All schedulers tend to obtain similar makepsan, as show in Fig. 9b.

5.4 Sensitive Analysis of Cluster Size

Finally, we investigate the effectiness of our new scheduler in a large cluster that contains more worker nodes and each node has larger capacity of cpu cores and memory. Specifically, we build a Hadoop YARN cluster in CloudLab [25] with 20 nodes, each of which is configured with 32 GB memory and 16 virtual cpu cores, i.e., $<32\text{G},16\text{cores}>$. The benchmark configuration is listed in Table 5, where we consider both non-iterative and iterative MapReduce jobs with different resource demands. Fig. 10 shows the experimental results (i.e., makespans and average resource usages) under five scheduling algorithms.

First of all, we notice that Fair and Capacity obtain better performance than FIFO, which is different from the previous experiments. We intepret that this large cluster with more resources actually experiences less pressure on resource contension and thus running more jobs simultaneously under Fair and Capacity can help make more efficient allocation decisions than under FIFO that only runs a job at one moment. More importantly, we can observe that HaSTE and HaSTE-A still achieve the best performance and HaSTE-A outperforms HaSTE because we have iterative jobs (e.g., Pagerank and Kmeans) in the benchmark.

6 Related Works

Improving the performance of Hadoop MapReduce systems has gained considerable research attention over the past few years. One important direction is the enhanced job scheduling. Zaharia et al. [26] proposed a delay scheduling policy to improve the performance of Fair scheduler by increasing the data locality of Hadoop. This work is compatible with both Fair scheduler and our proposed scheduling policies. Quincy [27] formulated the scheduling problem in Hadoop as a minimum flow network problem, and decided the slots

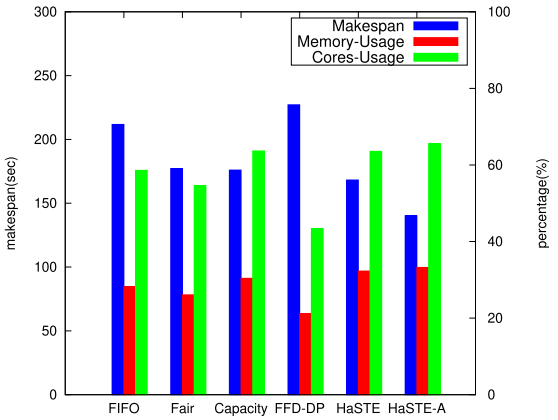


Fig. 10. Makespans and average resource usage under the mixed workload case 3 in a large cluster. The left y-axis shows the makespans (sec.) and the right y-axis shows the cpu and memory resource usage (%).

assignment that obeys the fairness and locality constraints by solving the minimum flow network problem. However, the complexity of this scheduler is high and it was designed for slot based scheduling in the first generation Hadoop. Verma et al. [28] introduced a heuristic method to minimize the makespan of a set of independent MapReduce jobs by applying the classic Johnson's algorithm. However, their evaluation is based on simulation only without real implementation in Hadoop. Wang et al. [29] proposed both static and dynamic slot configuration algorithms to balance the tradeoff between the overall fairness and the makespan for a batch of jobs. Dazhao et al. [30] proposed a self-adaptive task tuning system to automatically search the optimal configurations in the heterogeneous cluster. Our previous work [31] proposed a new scheme that uses the slot assignment as a tunable knob for reducing makespan of MapReduce jobs in Hadoop system. Refs. [28], [29], [30], [31], [32] were all based on the first-generation Hadoop scheme which utilize the slot concept for resource management.

Fine-grained resource management was also well studied for Hadoop systems. ThroughputScheduler [33] was proposed to improve the performance of heterogeneous Hadoop cluster. An explore stage was proposed to learn the resource requirement of tasks and the capabilities of nodes, and the best node was then selected to assign tasks in the scheduler. Polo et al. [34] leveraged job profiling information to dynamically adjust the number of slots on each node, as well as workload placement across nodes, to maximize the resource utilization of the Hadoop cluster. Our schedulers, however, do not require any learning phases or job profiles for scheduling. Wasi-ur Rahman et al. [35] is the first comprehensive study of intermediate data for YARN with Lustre and RDMA. Afrati et al. [36] mathematically investigates the scheduling problem which is assigning inputs with various sizes to a set of reducers with capacity. Rayon [37] is proposed to reserve resources for production jobs and best-effort jobs such that the SLAs for production jobs can be guaranteed and meanwhile the execution time of best-effort jobs can be reduced. We note that our HaSTE scheduler mainly focuses on how to allocate the reserved resources for best-effort jobs, which is complementary to Rayon in [37].

Although a bunch of previous works concentrates on the nature of the job, most of them classify the job into memory or cpu intensive. Some previous studies [38], [39] designed a modified framework to handle iterative jobs. However, we notice that none of these studies focuses on optimizing the scheduling for a MapReduce environment with iterative jobs. For example, Twister, proposed in [38], eliminates the disk read and write operations between map and reduce phases by differentiate static and variable data. While, our schedulers can be implemented as a plug-in module to the existing Hadoop YARN system without any modifications of those popular data processing frameworks, which presents high feasibility and flexibility in the scheduling.

7 Conclusion

In this paper, we presented two novel scheduling policies, named HaSTE and HaSTE-A, for Hadoop YARN systems. The primary goal of our new schedulers is to improve the usage of resources and reduce the makespan of a given set of MapReduce jobs. Based on each task's fitness and urgency, HaSTE dynamically schedules tasks for execution when resources become available. By further considering each task's alignment, our extended scheduler HaSTE-A effectively addresses the long tail issue caused by iterative jobs. We implemented both two schedulers in Hadoop YARN v.2.2.0 and evaluated them by running representative MapReduce benchmarks. The experimental results demonstrated that HaSTE and HaSTE-A improve the performance in terms of makespan under different workloads. In the future, we will extend our HaSTE scheduler to further allow resource allocation in YARN for other data-flow based frameworks, e.g., Spark. We also plan to derive the solution to an optimization problem for achieving an offline computed optimal makespan.

Acknowledgments

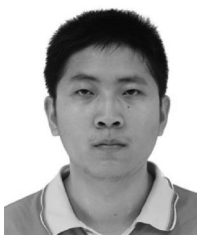
This work was partially supported by National Science Foundation Career Award CNS-1452751, CNS 109253 and AFOSR grant FA9550-14-1-0160.

References

[1] J. Dean, S. Ghemawat, and G. Inc, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Operating Syst. Des. Implementation*, 2004, pp. 137–150.
[2] Spark SQL. (2018). [Online]. Available: https://spark.apache.org/sql/
[3] Apache storm. (2018). [Online]. Available: http://storm.apache.org/
[4] Apache hadoop YARN. (2018). [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[5] Apache hive. (2018). [Online]. Available: https://hive.apache.org/
[6] Cloudera impala. (2018). [Online]. Available: http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html
[7] L. Page, "Method for node ranking in a linked database," U.S. Patent 6 285 999, Sep. 4, 2001.
[8] Apache mahout. (2018). [Online]. Available: https://mahout.apache.org/
[9] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 323–336.
[10] G. Ausiello, *Complexity and Approximability Properties: Combinatorial Optimization Problems and their Approximability Properties*. Berlin, Germany: Springer, 1999.

[11] P. Fattahi, M. S. Mehrabad, and F. Jolai, "Mathematical modeling and heuristic approaches to flexible job shop scheduling problems," *J. Intell. Manuf.*, vol. 18, no. 3, pp. 331–342, 2007.
[12] F. Pezzella, G. Morganti, and G. Ciaschetti, "A genetic algorithm for the flexible job-shop scheduling problem," *Comput. Operations Res.*, vol. 35, no. 10, pp. 3202–3212, 2008.
[13] M. Yazdani, M. Amiri, and M. Zandieh, "Flexible job-shop scheduling with parallel variable neighborhood search algorithm," *Expert Syst. Appl.*, vol. 37, no. 1, pp. 678–687, 2010.
[14] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic resource inference and allocation for MapReduce environments," in *Proc. 8th ACM Int. Conf. Autonomic Comput.*, 2011, pp. 235–244.
[15] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for MapReduce environments," in *Proc. IEEE Netw. Operations Manage. Symp.*, 2010, pp. 373–380.
[16] V. V. Vazirani, *Approximation Algorithms*. Berlin, Germany: Springer, 2001.
[17] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, 2015.
[18] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," *Research. Microsoft. Com*, 2011.
[19] D. S. Johnson, "Vector bin packing," in *Encyclopedia of Algorithms*. Berlin, Germany: Springer, 2014, pp. 1–6.
[20] J. Blazewicz, J. K. Lenstra, and A. Kan, "Scheduling subject to resource constraints: Classification and complexity," *Discrete Appl. Math.*, vol. 5, no. 1, pp. 11–24, 1983.
[21] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch, "Resource-constrained project scheduling: Notation, classification, models, and methods," *Eur. J. Oper. Res.*, vol. 112, no. 1, pp. 3–41, 1999.
[22] T. R. Browning and A. A. Yassine, "Resource-constrained multi-project scheduling: Priority rule performance revisited," *Int. J. Prod. Econ.*, vol. 126, no. 2, pp. 212–228, 2010.
[23] R. Kolisch and S. Hartmann, *Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*. Berlin, Germany: Springer, 1999.
[24] A. Sinha and P. K. Jana, "A novel MapReduce based k-means clustering," in *Proc. 1st Int. Conf. Intell. Comput. Commun.*, 2017, pp. 247–255.
[25] Cloud lab. (2018). [Online]. Available: https://www.cloudlab.us/
[26] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[27] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 261–276.
[28] 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 *Proc. IEEE 20th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2012, pp. 11–18.
[29] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, "Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters," in *Proc. IEEE 7th Int. Conf. Cloud Comput.*, 2014, pp. 761–768.
[30] D. Cheng, J. Rao, Y. Guo, C. Jiang, and X. Zhou, "Improving performance of heterogeneous MapReduce clusters with adaptive task tuning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 774–786, Mar. 2017.
[31] Y. Yao, J. Wang, B. Sheng, and N. Mi, "Using a tunable knob for reducing makespan of MapReduce jobs in a hadoop cluster," in *Proc. IEEE 6th Int. Conf. Cloud Comput.*, 2013, pp. 1–8.
[32] S. Tang, B.-S. Lee, and B. He, "Dynamic job ordering and slot configurations for MapReduce workloads," In *IEEE Trans. Services Computing*, vol. 9, no. 1, pp. 4–17, 2016.
[33] S. Gupta, C. Fritz, B. Price, R. Hoover, J. de Kleer, and C. Witteveen, "ThroughputScheduler: Learning to schedule on heterogeneous hadoop clusters," in *Proc. 10th ACM Int. Conf. Autonomic Comput.*, 2013, pp. 159–165.
[34] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, "Resource-aware adaptive scheduling for MapReduce clusters," in *Proc. 12th ACM/IFIP/USENIX Int. Conf. Middleware*, 2011, pp. 187–207.
[35] M. Wasi-ur Rahman, N. S. Islam, X. Lu, and D. K. D. Panda, "A comprehensive study of MapReduce over lustre for intermediate data placement and shuffle strategies on HPC clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 633–646, Mar. 2017.
[36] F. Afrati, S. Dolev, E. Korach, S. Sharma, and J. D. Ullman, "Assignment problems of different-sized inputs in MapReduce," *ACM Trans. Knowl. Discovery Data*, vol. 11, no. 2, 2016, Art. no. 18.
[37] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based scheduling: If you're late don't blame us!" in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.
[38] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative MapReduce," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 810–818.
[39] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endowment*, vol. 3, no. 1/2, pp. 285–296, 2010.

Yi Yao received the BS and MS degrees in computer science from the Southeast University, China, in 2007 and 2010, respectively. He is working toward the PhD degree in the Department of Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts. His current research interests include resource management, scheduling, and cloud computing.



Han Gao received the BS degree in communication engineering from Nankai University, China, in 2013, and the MS degree in electrical engineering from Pennsylvania State University, Pennsylvania, in 2015. He is working toward the PhD degree in the Department of Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts. His current research interests include distributed system, hadoop and spark scheduling, and cloud computing.

Jiayin Wang received the bachelor's degree in electrical engineering from Xidian University, China, in 2005, and the PhD degree from the University of Massachusetts Boston, in 2017. She is currently an assistant professor with the Computer Science Department, Montclair State University. Her research interests include cloud computing and wireless networks.

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

Ningfang Mi received the BS degree in computer science from Nanjing University, China, in 2000, the MS degree in computer science from the University of Texas at Dallas, Texas, in 2004, and the PhD degree in computer science from the College of William and Mary, Virginia, in 2009. She is an assistant professor with the Department of Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts. Her current research interests include performance evaluation, capacity planning, resource management, simulation, data center, and cloud computing.