HaSTE: Hadoop YARN Scheduling Based on Task-Dependency and Resource-Demand

Yi Yao* yyao@ece.neu.edu
Jiayin Wang† jane@cs.umb.edu
Bo Sheng† shengbo@cs.umb.edu
Jason Lin* jacks953107@ece.neu.edu
Ningfang Mi* ningfang@ece.neu.edu

*Department of Electrical and Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115
†Department of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125

***Abstract*—The MapReduce framework has become the de facto 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. One of the primary performance concerns in YARN is how to minimize the total completion length, i.e., makespan, of a set of MapReduce jobs. However, the precedence constraint or fairness constraint in current widely used scheduling policies in YARN, such as FIFO and Fair, can both lead to inefficient resource allocation in the Hadoop YARN cluster. They also omit the dependency between tasks which is crucial for the efficiency of resource utilization. We thus propose a new YARN scheduler, named HaSTE, which can effectively reduce the makespan of MapReduce jobs in YARN by leveraging the information of requested resources, resource capacities, and dependency between tasks. We implemented HaSTE as a pluggable scheduler in the most recent version of Hadoop YARN, and evaluated it with classic MapReduce benchmarks. The experimental results demonstrate that our YARN scheduler effectively reduces the makespans and improves resource utilization compare to the current scheduling policies.**

I. 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 [2], has also been widely adopted in both academia and industry for big data processing and information analysis. When MapReduce is getting popular, how to improve its performance becomes critical especially when a MapReduce cluster is serving a large number of jobs. Given limited resources in the cluster, when a batch of MapReduce jobs cannot all be executed, how to schedule their executions, i.e., allocate 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 an efficient scheduling scheme for YARN MapReduce to improve resource utilizations and reduce the makespan of a given set of jobs. We focus on the new generation of Hadoop system, YARN MapReduce [3]. Compared to the classic Hadoop MapReduce, YARN adopts a completely different design for resource management. In YARN, there is no "slot" which is the building block in the old versions, and the system no longer distinguishes map and reduce tasks when allocating resources. Instead, each task specifies a resource request in the form of <2G,1core> (i.e., requesting 2G memory and 1 cpu core), and it will be assigned to a node with sufficient capacity.

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 a new Hadoop YARN scheduling algorithm, named HaSTE, which aims at efficiently utilizing the resources for scheduling map/reduce tasks in Hadoop YARN and improving the makespan of MapReduce jobs. HaSTE meets these goals by leveraging the requested resources, resource capacities, and the dependency between tasks. Specifically, our solution dynamically schedules tasks 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 problems in the literatures. 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. Furthermore, we develop an aggregation function that combines the fitness and urgency to compare all candidate tasks and select the best one for execution.

The rest of this paper is organized as follows. In Section II, we briefly introduce the background of the scheduling problem and existing scheduling policies in YARN. In Section III, we formulate the scheduling problem of YARN as resource constrained scheduling, and propose our new scheduling policy HaSTE. Evaluation results of our proposed scheme are presented in Section IV. We describe the related works in Section V and conclude in Section VI.

This work was partially supported by the National Science Foundation grant CNS-1251129, and the AWS in Education Research Grant.

II. HADOOP YARN SCHEDULER

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 classic Hadoop, 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 its worker node and each ApplicationMaster needs to specify the resource 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 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 classic Hadoop, YARN systems no longer explicitly distinguish map and reduce tasks such that other parallel data processing applications can also be supported by YARN. In this work, we mainly focus on MapReduce applications running in YARN. The scheduling policies that are widely 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. The first queuing task request that fits into the residual capacity of the worker node will be scheduled for service at each iteration.
- Two Fair scheduling policies have been implemented in Hadoop YARN, i.e., Fair and Dominant Resource Fairness (DRF) [4]. Fair policy only considers the memory usage of each job and attempts to assign equal shares of memory, while the DRF policy aims to ensure all jobs to get on average an equal share on their dominant resource requirements.
- The Capacity policy works similar as the Fair policies. The scheduler attempts to reserve a guaranteed capacity for each job queue and orders jobs by their deficit between their deserved capacity and the actual occupied capacity.

We argue that the above policies are not designed for optimize resource utilization and completion time of MapReduce jobs in YARN. Therefore, in this work, we strive to design a new YARN scheduler which attempts to minimizing the makespan of a batch of MapReduce jobs.

III. HASTE

A. 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 n jobs as a set T and assign each task a unique index number, i.e., t_i represents the i-th task in the system. Each job J_i is then 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 YARN, each task can request for user-specified resources. All map/reduce tasks in a job 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 Hadoop 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, and x_{ij} indicate the association between t_i and S_j, i.e., x_{ij} is 1 if task t_i is assigned to worker node S_j. Then our scheduling problem is to derive st_i and x_{ij} in order to

$$\text{minimize: } max\{st_i + \tau_i\}, \forall i \in T$$

$$\text{s.t.} \sum_{j \in [1,m]} x_{ij} = 1, \forall t_i \in T; \quad (1)$$

$$\sum_{t_i \in A(\theta)} x_{ij} \mathcal{R}[i, p] \leq C[j, p], j \in [1, m], p \in [1, k], \theta > 0; \quad (2)$$

$$x_{ij} \in \{0, 1\}, st_i \geq 0, \ \forall i, j.$$

Here time is measured as a discrete value which is multiple of the *time unit*. θ represents a particular time point, and $A(\theta)$ is defined as the set of *active* tasks at time θ, $A(\theta) = \{t_i \in T | st_i \leq \theta \leq st_i + \tau_i\}$. Therefore, constraint (1) specifies that each task could be assigned to exactly one NodeManager, and constraint (2) requires that the resources consumed by all active tasks at a particular worker node S_j cannot exceed its resource capacity.

Assume τ_i is available and each map/reduce task is independent, our scheduling problem is equivalent to general resource constrained scheduling problem which has been proved to be NP-complete [5]. Many heuristics have been proposed for solving the problem. 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 [6], [7]. 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 which host the reduce tasks, while 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.

B. Sketch of Our Solution HaSTE

We design a scheduler that consists of two components, *initial task assignment* and *real-time task assignment*. First, *initial task assignment* 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 *initial task assignment* is to assign the first batch of tasks for execution while the rest of tasks remain pending in the system queue. Specifically, *initial task assignment* algorithm needs to select a subset of pending tasks and select a hosting work node for each of them for execution. On the other hand, *real-time task assignment* 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 *real-time task assignment* to select one or more tasks from the pending queue and assign them to the worker node with new resources available. Compared to *initial task assignment*, *real-time task assignment* 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 *initial task assignment* and *real-time task assignment* algorithms. *Initial task assignment* is formulated as a variant of the knapsack problem. We then use dynamic programming to derive the best task assignment in the beginning. *Real-time task assignment* 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 for determining the appropriate task to execute on-the-fly.

C. Initial Task Assignment

The objective of this component 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. Therefore, we adopt the greedy strategy and simplify our objective to be maximizing the resource utilization after *initial task assignment*. If there is only one type of resource, then this problem is equivalent to the typical knapsack problem. Consider each worker node as a knapsack, 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% cpu and 90% memory better than utilizing 90% cpu and 100% memory? We then assume that the cluster specifies a weight w_i for each resource r_i. The *initial task assignment* problem can be formulated as follows:

$$\text{maximize: } \sum_{t_i \in T} \Big(\sum_{j \in [1,m]} x_{ij} \cdot \sum_{p \in [1,k]} w_p \cdot \mathcal{R}[i,p] \Big)$$

$$\text{s.t. } \sum_{j \in [1,m]} x_{ij} \leq 1, \forall t_i \in T;$$

$$\sum_{t_i \in T} x_{ij} \cdot \mathcal{R}[i,p] \leq C[j,p], \forall j \in [1,m], p \in [1,k].$$

We design an algorithm using dynamic programming to solve the problem. The details are illustrated in the following 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 $\mathcal{M}$ and $\mathcal{L}$, where $\mathcal{M}[a,b]$ 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 $\mathcal{M}$ 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 matrices (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 update the matrices $\mathcal{M}$ and $\mathcal{L}$ (line 16-17).

Algorithm 1: Initial Task Assignment

```
   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 = M[a − R[i,1], b − R[i,2]] + V;
15 |  |  |  |  if M[a,b] < tmp then
16 |  |  |  |  |  M[a,b] = tmp;  tmpL = L + {t_i};
17 |  |  |  L[a,b] = tmpL;
18 |  |  (x,y) = argmax_{a,b} M[a,b];
19 |  |  L = L[a,b];
20 |  |  T ← T − L;
21 |  |  for each t_i ∈ L do
22 |  |  |  x_ij = 1;
23 |  |  return;
```

D. Real-time Task Assignment

Real-time task assignment is the core component in our design of HaSTE as it is repeatedly conducted during the execution of all the jobs. The main goal 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 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 subsection, we first describe the calculation of each metric and then present the overall algorithm of *real-time task assignment*.

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 [8]. 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. Figure 1 illustrates how FFD can improve the makespan and resource utilization when scheduling two jobs with different memory requirements.

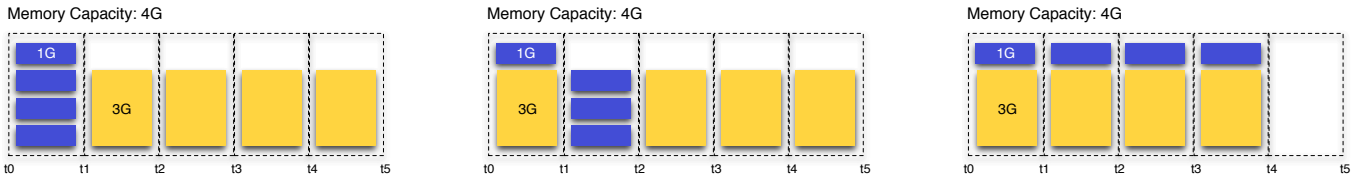
In fact, with two types of resources (memory and cpu) supported in Hadoop YARN, the simplified scheduling problem is equivalent to the vector bin packing problem in the literature. Different variants of FFD have been studied for solving the vector bin packing problem [9]. The FFD-DotProduct (dubbed as FFD-DP) method has been shown to be superior under various evaluation sets. Therefore, we adopt the FFD-DP method to schedule map and reduce tasks with two resource requirements. Specifically, we define *fitness* as:

$$F_{ij} = \sum_{p \in [1,k]} \mathcal{R}[i,p] \cdot C[j,p] \cdot w_p. \tag{3}$$

Real-time task assignment uses Eq.(3) 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. Therefore, *real-time task assignment* 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, *real-time task assignment* 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 two types of tasks with different resource requirements: one requests <1 GB, 3 cores> and the other requests <3 GB, 1 core>; and *real-time task assignment* tries to assign tasks to a worker node with residual capacity of <10 GB, 6 cores>. The FFD-DP algorithm will choose 3 tasks of type II and 1 task of type I, which results in 100% 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**

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



(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 task 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.

there has been previous work [10]–[13] 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 [10]–[13]. 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. However, in the MapReduce framework, 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 the 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 have the following equations to calculate the urgency score U_i^m (U_i^r) for each map (reduce) task from job i:

$$U_i^m = \frac{A_i^m}{T_i^m} \cdot (A_i^r \cdot R_i^r + A_i^{am} \cdot R_i^{am}), \quad (4)$$

$$U_i^r = U_i^m \cdot \frac{A_i^m}{T_i^m} \cdot \frac{O_i^m \cdot R_i^m + O_i^r \cdot R_i^r}{O_i^r \cdot R_i^r}. \quad (5)$$

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) 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.(6).

$$P_i = \frac{F_i - F_{min}}{F_{max} - F_{min}} + \frac{U_i - U_{min}}{U_{max} - U_{min}}, \quad (6)$$

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.

IV. Evaluation

In this section, we evaluate the performance of HaSTE by conducing experiments in a Hadoop YARN cluster. We implemented both HaSTE and FFD-DotProduct (abbrev. FFD-DP) schedulers in Hadoop YARN version 2.2.0 and compared HaSTE with three built-in schedulers (i.e., FIFO, Fair and DRF) and FFD-DP. The performance metrics considered in the evaluation include makespans of a batch of MapReduce jobs and resource utilizations of the Hadoop YARN cluster.

A. 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[1]. 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 because the concurrency level of MapReduce jobs and the actual resource utilizations 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.

[1]We note that the virtual cpu cores are not physically isolated for 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.

B. Experiment Results

Here, we conduct two sets of experiments in a Hadoop YARN cluster with 8 nodes, each of which is configured with the capacity of 8GB memory and 8 virtual cpu cores, i.e., <8G, 8cores> .

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 64MB in this experiment. As described in Section IV-A, 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 I.

TABLE I: Simple Workload Configuration.

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

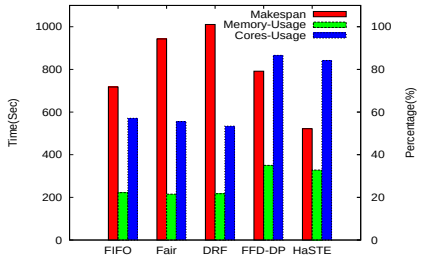


Fig. 2: Makespans and average resource utilizations under the workload of 4 *Wordcount* jobs. The left y-axis shows the makespans (sec.) while the right y-axis shows the cpu and memory resource utilizations (%).

Figure 2 shows the makespans and the average resource (mem and cpu) utilizations under different scheduling policies. We observe that all the conventional schedulers (i.e., FIFO, Fair, and DRF) cannot efficiently utilize the system resources, e.g., under 60% cpu core utilization and under 30% memory utilization. Although these conventional schedulers obtain similar resource utilizations, FIFO outperforms Fair by 23.8% and DRF by 29.3%. 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% larger than under FIFO, although FFD-DP achieves the highest resource utilizations, e.g., 86.6% cpu cores utilization 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% shorter than FIFO and Fair, respectively.

2) Mixed Workload: To further validate the effectiveness of HaSTE, we conduct a more complex workload which is mixed with both cpu and memory intensive MapReduce jobs. Table II 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 128MB.

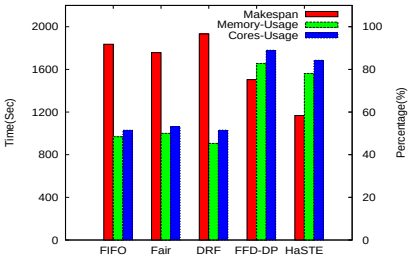


Fig. 3: Makespans and average resource utilizations under the mixed workload of four benchmarks. The left y-axis shows the makespans (sec.) while the right y-axis shows the cpu and memory resource utilizations (%).

Figure 3 plots the makespans and the average resource utilizations under this mixed workload. Consistently, the three conventional scheduling policies have similar average resource utilizations, e.g., around 50% 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 utilizations , e.g., to around 80%, through the resource-aware task assignment. FFD-DP also improves the makespan by 18.1% and 14.8% compared to FIFO and Fair, respectively. HaSTE further improves the performance in terms of makespan by 36.3% and 33.9% compared to FIFO and Fair, respectively.

To better understand how these scheduling policies work, we further plot the runtime memory allocations in Figure 4. We observe that the precedence constraint of FIFO policy and the fairness constraint of Fair and DRF policies 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 job 3,4,6,7 in Figure 4(a), 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 utilizations. For example,

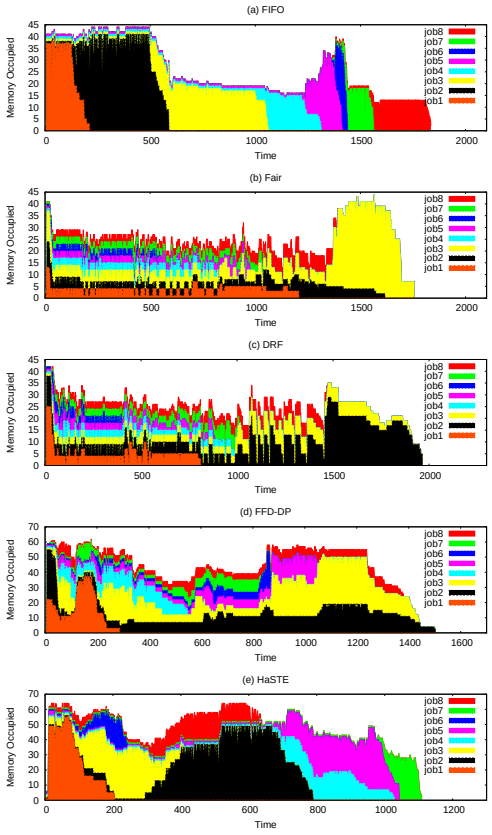


Fig. 4: Illustrating the memory resources that have been allocated to each job cross time under different scheduling policies.

when a node has $< 1\,GB, 4\,cores >$ 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 utilizations across time. More importantly, HaSTE also achieves high or even slightly higher resource utilizations 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 utilizations.

In summary, HaSTE achieves non-negligible improvements in terms of makespans and resource utilizations 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 utilization. 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.

V. Related Work

One important direction is the enhanced job scheduling. Zaharia et al. [14] proposed a delay scheduling policy to

TABLE II: Mixed Workload Configuration.

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

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 [15] formulated the scheduling problem in Hadoop as a minimum flow network problem, and decided the slots assignment that obey 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. [16] introduced a heuristic to minimize the makespan of a set of independent MapReduce jobs by applying the classic Johnson's algorithm. Our previous work [17] proposed a new scheme that uses the slot assignment as a tunable knob for reducing the makespan of MapReduce jobs in a Hadoop system. These two works were both based on the first generation Hadoop, which adopts the slot concept for resource management.

Fine-grained resource management was also well studied for Hadoop systems. ThroughputScheduler was proposed by Gupta et al. [18] to improve the performance of heterogeneous Hadoop clusters. An explore stage was proposed to learn the resource requirement of tasks and the capabilities of nodes, and the scheduler could then select the best node to assign tasks. Polo et al. [19] 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 scheduler, however, does not require any learning phases or job profiles for scheduling. Therefore, HaSTE is more lightweight and practical.

VI. Conclusion

In this paper, we presented a novel scheduling policy (HaSTE) for Hadoop YARN systems. The primary goal of HaSTE is to improve the resource utilization 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. We implemented HaSTE in Hadoop YARN v.2.2.0 and evaluated this scheme by running representative MapReduce benchmarks. The experimental results demonstrated that HaSTE improves the performance in terms of both resource utilization and makespan under different workloads. In the future, we plan to extend our design of scheduling policies for other cloud computing paradigms such as Storm and Spark.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[3] V. K. Vavilapalli, A. C. Murthy, C. Douglas *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013.

[4] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: fair allocation of multiple resource types," in *USENIX NSDI*, 2011.

[5] G. Ausiello, *Complexity and Approximability Properties: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 1999.

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

[7] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*. IEEE, 2010, pp. 373–380.

[8] V. V. Vazirani, *Approximation algorithms*. springer, 2001.

[9] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," *research. microsoft. com*, 2011.

[10] J. Blazewicz, J. K. Lenstra, and A. Kan, "Scheduling subject to resource constraints: classification and complexity," *Discrete Applied Mathematics*, vol. 5, no. 1, pp. 11–24, 1983.

[11] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch, "Resource-constrained project scheduling: Notation, classification, models, and methods," *European Journal of Operational Research*, vol. 112, no. 1, pp. 3–41, 1999.

[12] T. R. Browning and A. A. Yassine, "Resource-constrained multi-project scheduling: Priority rule performance revisited," *International Journal of Production Economics*, vol. 126, no. 2, pp. 212–228, 2010.

[13] R. Kolisch and S. Hartmann, *Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis*. Springer, 1999.

[14] M. Zaharia, D. Borthakur, J. Sen Sarma *et al.*, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.

[15] M. Isard, V. Prabhakaran, J. Currey *et al.*, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 261–276.

[16] 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 *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*. IEEE, 2012, pp. 11–18.

[17] Y. Yao, J. Wang, B. Sheng, and N. Mi, "Using a tunable knob for reducing makespan of mapreduce jobs in a hadoop cluster," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 1–8.

[18] S. Gupta, C. Fritz, B. Price, R. Hoover, J. de Kleer, and C. Witteveen, "Throughputscheduler: Learning to schedule on heterogeneous hadoop clusters," in *Proceedings 10th ACM International Conference on Autonomic Computing (ICAC'13)*. ACM.

[19] J. Polo, C. Castillo, D. Carrera *et al.*, "Resource-aware adaptive scheduling for mapreduce clusters," in *Middleware 2011*. Springer, 2011, pp. 187–207.