Dependency-aware and Resource-efficient Scheduling for Heterogeneous Jobs in Clouds

Jinwei Liu* and Haiying Shen†
*Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634, USA
†Department of Computer Science, University of Virginia, Charlottesville, VA 22904, USA
jinweil@clemson.edu, hs6ms@virginia.edu

***Abstract*—Data analytics frameworks shift towards larger degrees of parallelism. Efficient scheduling of data-parallel jobs (tasks) is critical for improving job performance such as response time, and resource utilization. It is an important challenge for large scale data analytics frameworks in which jobs are more complex and have diverse characteristics (e.g., diverse resource requirements). Prior work on scheduling cannot achieve low response time and high resource utilization simultaneously because they cannot accurately estimate the durations of tasks in the queue of a worker machine by using sampling-based approach (including sampling with late binding) for task placement, and thus they fail to place tasks at the best possible worker machine. Also, they do not sufficiently consider the diverse resource requirements of jobs (tasks) for placing tasks on worker machines. To address this challenge, we propose a Dependency-aware and Resource-efficient Scheduling (DRS) to achieve low response time and high resource utilization. DRS takes into account task dependency and assigns tasks that are independent of each other to different worker machines. Also, DRS considers tasks' resource requirements and packs complementary tasks whose resource demands on multiple resources are complementary to each other to increase the resource utilization. In addition, DRS uses the mutual reinforcement learning to estimate the task's waiting time (the duration of tasks in the queue of a worker), and assigns tasks to workers with the consideration of tasks' waiting time to reduce the response time. Extensive experimental results based on a real cluster and experiments using real-world Amazon EC2 cloud service show that DRS achieves low response time and high resource utilization compared to previous strategies.**

I. Introduction

Cloud frameworks tailored for managing and analyzing big datasets are powering ever larger clusters of computers. Parallel processing frameworks have become a key computing platform. Providing low response time and high resource utilization for parallel jobs that run on thousands of machines poses a challenge for scheduling. Usually, parallel jobs consist of hundreds or thousands of concurrent tasks, and the response time is determined by the tail task. Therefore tasks should be efficiently scheduled. The response time will be increased even if a single task placed on a contended machine. Also, jobs (tasks) become more complex and have diverse characteristics (e.g., diverse resource requirements). Existing schedulers [1]–[3] cannot well handle this challenge because they cannot accurately estimate the durations of tasks in the queue of a worker machine and the sampling-based approach used for task placement, is not accurate and sometimes is even opportunistic, which prevents the schedulers from placing tasks at best possible worker machines.

9781-5090-1445-3/16$31.00

In parallel jobs, the dependency between sub-second concurrent tasks (e.g., 100ms tasks) is common. The existing scheduling approaches [2], [4], [5] try to address some of the difficulties of job scheduling but ignore task dependency while scheduling tasks. The work [4] designs a new scheduling algorithm, Longest Approximate Time to End (LATE), that is both simple and highly robust to heterogeneity for Hadoop MapReduce environment. However, it cannot handle sub-second parallel jobs with task dependency constraints. The work [5] proposes a new TAGS (Task Assignment based on Guessing Size) algorithm, and it is counterintuitive in many respects, including load unbalancing, non-work conserving, and fairness, but it assumes tasks are independent, and it thus cannot handle tasks with dependency constraints. The work [2] presents a distributed, low latency scheduler Sparrow with placement constraints for heterogeneous cloud environment. However, Sparrow does not consider the dependency relations among tasks when assigning tasks to workers, and it thus may violate the dependency constraints exiting among tasks in parallel jobs. Although the work [6] considers task dependency, it does not consider the deadline constraints of individual jobs. In contrast to existing scheduling approaches, our proposed DRS takes into account task dependency and assigns tasks that are independent of each other to different workers (or different cores of a worker) so that the tasks can run in parallel, and thus further reduce the response time and increase the throughput.

In parallel processing framework, the workloads/jobs become more and more diverse. The work [7] shows that more than 50% of the jobs at Google have constraints about the machines that they can run on. Different jobs may have requirements on the hardware or software. For example, a job might require a machine with special hardware (i.e., GPUs, amount of memory). These are simple constraints. In addition to simple constraints, combinatorial constraints are also possible [8], i.e., requiring two tasks to be placed on two distinct machines. Efficient resource allocation with the consideration of jobs' resource constraints thus attracts many interests, and becomes a hot topic [9]–[13]. Several existing works [3], [9], [10], [14] on scheduling aim to achieve high resource utilization. The work [3] proposes a hybrid centralized/distribued scheduler called Hawk for job scheduling. Long jobs are scheduled using a centralized scheduler, and short jobs are scheduled in a fully distributed way. Hawk uses a randomized work-stealing algorithm to improve the resource utilization. The work [9] leverages the information provided

by the developed reservation system about jobs' deadlines and estimated runtimes to plan ahead which jobs to defer so that the right sources can be assigned to the right jobs. The work [14] introduces opportunistic scheduling, which classifies tasks into two types: regular tasks and opportunistic tasks. The work [14] ensures low latency for regular tasks, but uses the opportunistic tasks for high utilization to fill the slack left by regular tasks. However, the above works cannot fully utilize the resource because they do not consider tasks' diverse resource requirements on different resource types and leverage the complementarity of tasks' resource requirements to increase the resource utilization. Unlike previous works, our proposed DRS assigns tasks to workers with the consideration of satisfying tasks' constraints on resources and minimizing the resource consumption, and DRS thus can achieve high resource utilization with low cost.

To address the challenge and handle the issues in previous works, in this paper, we propose a Dependency-aware and Resource-efficient Scheduling (DRS) to achieve low response time and high resource utilization. DRS is more advantageous than previous schedulers in that DRS can achieve low response time of jobs and high resource utilization simultaneously by utilizing the dependency information and the mutual reinforcement learning to accurately estimate the duration of tasks in the waiting queue of a worker and leveraging the complementarity of tasks' diverse resource requirements on different resource types to increase the resource utilization. We summarize the contribution of this work below.

- We build a linear programming model that aims to minimize the resource cost and increase the resource utilization and present a cost-efficient scheduling system for processing heterogeneous jobs in clouds.
- DRS takes into account task dependency and assigns tasks that are independent of each other to different workers so that the response time of jobs can be reduced. Also, DRS introduces the concept of scheduler domains and uses the Gossip protocol for the communication between the scheduler managers in different domains based on Geometric Random Graph (GRG) to reduce the communication overhead.
- DRS leverages tasks' diverse resource requirements on different resource types and packs workers' complementary tasks whose resource demands on multiple resource types are complementary to each other to increase the resource utilization.
- DRS uses the mutual reinforcement learning to estimate the tasks' waiting time in the queue of workers, and assigns tasks to workers with the consideration of tasks' waiting time in the queue of workers so that the response time can be reduced.
- We have conducted extensive experiments on both a real cluster and Amazon EC2 to demonstrate the advantages of DRS in reducing the resource cost and time overhead and increasing the throughput.

The remainder of this paper is organized as follows. Section II describes the system model used in this paper. Section III presents design for our cost-efficient scheduling system. Section IV presents the performance evaluation for DRS. Section V reviews the related work. Section VI concludes this paper with remarks on our future work.

II. System Model

In this section, we first introduce some concepts and assumptions, then we introduce our proposed model for minimizing the resource cost with the consideration of tasks' resource constraints.

A. Concepts and Assumptions

In a distributed system, there are workers and schedulers. Workers are used to execute tasks, and schedulers are used to assign tasks to workers. A job is supposed to be split into m tasks, and the tasks are allocated to workers based on the dependency relations among tasks and their requirements on resources. We assume jobs can be handled by any scheduler and tasks are run by workers in a fixed number of slots. Each worker has a buffer queue[1] which is used for queueing tasks when a worker is allocated to more tasks than it can run concurrently.

Definition 1: Waiting time: The time from when a task is submitted to the worker machine until when the task starts executing.

Definition 2: Service time: The time from when a task starts executing until when the task finishes executing on a worker machine.[2]

Definition 3: Job response time: The time from when a job is submitted to the scheduler until the tail task of the job finishes executing.

Definition 4: Throughput: The total number of jobs completing their executions within the job deadlines per unit of time.

Problem Statement: Given a set of jobs consisting of tasks, constraints of tasks (i.e., per-task constraints, resource constraints, response time constraints of jobs) and a set of heterogeneous worker machines, what is the minimum *resource cost*? Then, how to schedule these jobs so that the resource cost is minimized while the overhead is reduced?

We consider a scheduling problem with the objective of scheduling multiple jobs onto multiple heterogeneous workers to minimize resource cost and increase resource utilization while satisfying jobs' constraints. We present an integer linear programming (ILP) model to minimize resource cost and increase resource utilization while satisfying jobs' constraints. We introduce the details of the model as follows. For easy

TABLE I: Notations

n	Total # of workers	L	A set of workers' queue length
$\mathbf{J}$	A set of jobs	$f(k)$	Proc. rate func. of worker k
a	# of jobs in $\mathbf{J}$	s_c^k	CPU size of worker k
J_i	The ith job in $\mathbf{J}$	s_m^k	Mem. size of worker k
t_{ij}	The jth task of J_i	M	Resource matrix
t_{ij}^b	t_{ij}'s starting time	R_r	Type r resource
t_{ij}^e	t_{ij}'s ending time	l	# of resource types
S_{ij}	Task t_{ij}'s size	c_r	Cost of R_r per unit time
C_{cos}	Resource cost	t_{ij}^s	t_{ij}'s service time
m	# of tasks / job	$d_{ij,r}$	t_{ij}'s demand on R_r

[1] Although some cluster schedulers do not support buffer queues, we assume a worker has a buffer queue [2], [3], [14], [15] for the purpose of utilizing task dependency information to increase throughput in scheduling of cloud-scale computing clusters [14].

[2] In the paper, we currently do not consider preemption due to the complexity of modeling the optimization problem with multiple constraints, and we leave it to our future work.

reference, Table I shows the main notations used in this paper. Suppose a jobs ($\mathbf{J} = \{J_1, ..., J_a\}$) are submitted at time t. The deadlines on job response time are represented by $t_1^d, t_2^d, ..., t_a^d$. Denote $t_{ij}^b(u)$ as the starting time of the uth task on the chain C_l belonging to job J_i running on a worker machine, and $t_{ij}^e(u)$ as the ending time (completion time) of the uth task running on a worker machine. Define $f(k)$ $(k \in \{1, ..., n\})$ as a function of processing rate of the kth worker, which is the million instructions per second (MIPS) speed, and $f(k)$ is related to the CPU size s_c^k and memory size s_m^k of the worker k [16]. The larger the CPU size, the higher the processing rate; the larger the memory size, the higher the processing rate. Specifically, the function $f(k)$ is expressed as follows

$$f(k) = \alpha s_c^k + \beta s_m^k \tag{1}$$

where $k \in \{1, ..., n\}$, and α and β are weights such that $\alpha + \beta = 1$. Let S_{ij} be the size of task t_{ij} belonging to job J_i in terms of Millions of Instructions (MI) [17], $t_{ij,k}^s$ be the service time of task t_{ij} assigned to worker k. Thus, we have $t_{ij,k}^s = \frac{S_{ij}}{f(k)}$ $(i \in \{1, ..., a\}, j \in \{1, ..., m\})$. Let $y_{ij,k} = \{0, 1\}$ $(i \in \{1, ..., a\},\ j \in \{1, ..., m\}, k \in \{1, ..., n\})$ be a binary variable representing if task t_{ij} is assigned to worker k, $y_{ij,k} = 1$ if t_{ij} is assigned to worker k, otherwise $y_{ij,k} = 0$. Let $x_{ij,uv,k} = 1$ if task t_{ij} precedes task t_{uv} on worker machine k, otherwise $x_{ij,uv,k} = 0$.

B. Resource Cost Minimization

The workers in a distributed system have different resource types (e.g., CPU, memory, GPU). For analytical tractability, we use a matrix M to represent the resources of workers

$$M = \begin{bmatrix} m_{11} & m_{12} & \cdots m_{1l} \\ m_{21} & m_{22} & \cdots m_{2l} \\ \vdots & \vdots & \vdots \\ m_{n1} & m_{n2} & \cdots m_{nl} \end{bmatrix} \tag{2}$$

where each column represents one resource type, and each row represents the resource types of a worker. Different tasks may have different requirements on different resource types. Some tasks are CPU intensive, and some tasks are memory intensive. It is of importance to reduce the resource cost while satisfying tasks' requirements on resources for task assignment [18]–[21]. Represent C_{cos} as the cost for all jobs' (a jobs submitted at time t) resource consumption. Let c_r be the cost of using resource type R_r per time unit $(r \in \{1, ..., l\})$. Let $d_{ij,r}$ be task t_{ij}'s demand on resource type R_r (the number of units on resource type R_r required by t_{ij}) where $i \in \{1, ..., a\}, j \in \{1, ..., m\}, r \in \{1, ..., l\}$. We assume that the resource allocated to the tasks cannot be reallocated to other tasks until the tasks finish execution. Thus, we formalize our problem as the following constrained optimization problem:

$$min \quad C_{cos} = \sum_{i=1}^{a} \sum_{j=1}^{m} \sum_{r=1}^{l} \sum_{k=1}^{n} (d_{ij,r} \cdot c_r \cdot t_{ij,k}^s \cdot y_{ij,k}) \tag{3}$$

$$s.t. \ \sum_{i=1}^{a} \sum_{j=1}^{m} d_{ij,r} \sum_{k=1}^{n} t_{ij,k}^s y_{ij,k} - \sum_{i=1}^{a} \sum_{j=1}^{m} m_{kr} \sum_{k=1}^{n} t_{ij,k}^s y_{ij,k} \le 0 \ (r \in \{1, ..., l\}) \tag{4}$$

$$(t_{ij}^b + t_{ij,k}^s) \cdot y_{ij,k} \le (t_{uv}^b + (1 - x_{ij,uv,k}) \cdot E) \cdot y_{uv,k} \quad (\forall u \in \{1, ..., a\}, v \in \{1, ..., m\}) \tag{5}$$

$$t_{ij}^b + \sum_{k=1}^{n} t_{ij,k}^s \cdot y_{ij,k} \le t_i^d \ (t_{ij} \in C_l) \tag{6}$$

$$t_{ij}^b + \sum_{k=1}^{n} t_{ij,k}^s \cdot y_{ij,k} \le t_{iq}^b \ (q \in \{1, ..., m\}) \tag{7}$$

$$x_{ij,uv,k} + x_{uv,ij,k} = 1 \tag{8}$$

$$x_{ij,uv,k} \in \{0, 1\} \tag{9}$$

$$y_{ij,k} \in \{0, 1\} \tag{10}$$

$$\sum_{k=1}^{n} y_{ij,k} = 1 \tag{11}$$

where $t_{ij,k}^s = \frac{S_{ij}}{f(k)}$, and E in Formula (5) is a very large positive real number. The constraint (4) is to ensure that the cumulative resource usage of R_r on a worker machine m_k does not exceed the capacity of this resource type during the period of task execution. The constraint (5) is to ensure the execution order of t_{ij} and t_{uv} on worker k. The constraint (6) is to ensure jobs can complete within their specified deadlines. The constraint (7) ensures the dependency relation between t_{ij} and t_{iq}. The constraint (11) is to ensure that a task can be assigned to only one worker machine. k $(k|_{y_{ij,k}=1} \in \{1, ..., n\})$ represents a set of workers derived from the model.

We use the CPLEX linear program solver [22] to solve the large-scale linear optimization problem. Given a set of jobs consisting of tasks, constraints of jobs and tasks, and the resource information of a number of workers, by the resource cost minimization model, we know the target workers for all the tasks, namely, the schedulers can assign tasks to different workers and minimize the resource cost while satisfying the constraints of jobs and tasks.

III. System Design

In this paper, we aim to reduce the resource cost and the time overhead while reducing the response time of jobs and increasing the throughput. In Section II, we introduce the system model for resource cost minimization. In this section, we describe the design of our cost-efficient scheduling system. Though Sparrow [2] and Hawk [3] can provide low latency scheduling, they do not consider reducing the resource cost. Also, Sparrow assigns tasks to workers based on batch sampling with late binding, which can incur time overhead and thus increase the response time of jobs because the sampling based approach cannot accurately learn the duration of tasks in workers' queues. To handle this problem, we present DRS that assigns tasks to workers with the consideration of reducing resource cost. Also, DRS takes into account the communication overhead, and it utilizes the scheduler domains and uses the Gossip protocol for the communication between the scheduler managers in different domains based on the GRG to reduce the communication overhead [23], [24]. By using scheduler managers to manage the information of schedulers, DRS can help balance the load of schedulers and thus avoid hot spots, which is neglected in Sparrow [2] and Hawk [3]. In addition, DRS uses the mutual reinforcement learning to estimate tasks' waiting time in queues of workers for task placement so that the response time can be reduced.

By the ILP model for resource cost minimization, the target workers for all the tasks can be obtained. Algorithm 1 shows the pseudocode of DRS scheduling. DRS first assigns jobs

to schedulers based on the geographic distance between the jobs and schedulers and schedulers' loads. Also, DRS reduces the communication overhead between schedulers using Gossip protocol for the communication between the scheduler managers in different domains rather than the communication between different schedulers. Second, the schedulers split jobs into tasks and assign tasks to workers based on the ILP model to reduce the resource cost. We present the details for the design of DRS in the following.

A. Job Submission and Job Assignment

In DRS, schedulers are scattered in a distributed system. When users submit their jobs, the jobs are delivered to the schedulers near them. If the scheduler is heavily loaded, the job will be delivered to the lightly loaded neighbor of the heavily loaded scheduler to achieve load balance.

Algorithm 1: Pseudocode for Job_Processing()

```
A Job is submitted
sort(s[]) //Sort schedulers by distances between schedulers and the user submitting the job
for i ← 1 to Len(s) do
    if s[i] is lightly loaded then
        Assign the job to s[i]
        s[i] split the job into m tasks
        Call Task_Scheduling() //Call Algorithm 2
    if i = Len(s) + 1 then
        i ← i mod Len(s)
return
```

Algorithm 2: Task_Scheduling()

```
for i ← 1 to n do
    if m_i satisfy the resource constraints of the tasks then
        if m_i has multi-processors then
            Fetch independent tasks from the queue and assign them to different processors
return
```

Each scheduler has a unique key, and each scheduler has the resource information of several workers associated with the scheduler. The workers periodically report their resource information to the scheduler, and the scheduler periodically updates a table which contains the resource information of each worker associated with the scheduler. To make the job scheduling more efficient, we not only make the resource information of workers available to the scheduler near those workers, but also classify the schedulers into several categories (i.e., CPU intensive, memory intensive, GPU intensive, etc.). The CPU intensive schedulers have the resource information of CPU intensive workers near those schedulers, similarly, the memory intensive schedulers have the resource information of memory intensive workers near those schedulers, etc. When a job arrives, if the job is CPU intensive, the job will be assigned to a CPU intensive scheduler; if it is memory intensive, the job will be assigned to a memory intensive scheduler, etc.

Based on the resource cost minimization model, we can obtain the target workers for all tasks. Next we describe the communication between scheduler managers, and how to assign tasks to different workers.

B. Scheduler Manager Communication

To reduce the communication overhead, we introduce the scheduler managers and scheduler domains that split the schedulers into different sets. Each set is a scheduler domain, and there is a scheduler manager in each domain, maintaining the information of schedulers in that domain.

To quickly spread different schedulers' information and reduce the communication overhead, we use the Gossip protocol for the communication between the scheduler managers in different domains rather than the communication between different schedulers. First, the schedulers in each domain report their load information to the scheduler manager in the same domain. Second, the scheduler manger uses Gossip protocol to communicate with its neighbor based on GRG. Specifically, the scheduler manager in each domain randomly chooses one of its neighbors and sends its information to the neighbor. According to [25], the average time that every scheduler manager receives messages is $O(r^{-1}logN)$, where r is the constant transmission radius [26], and N is the number of scheduler managers in the distributed system. In our design, we choose the constant transmission radius r such that each scheduler manager has at least one neighbor. Compared to the approach using Gossip protocol for the communication between different schedulers, the average time that every scheduler manager receives messages in DRS is much less, because the number of scheduler managers is much less than that of schedulers in the whole distributed system.

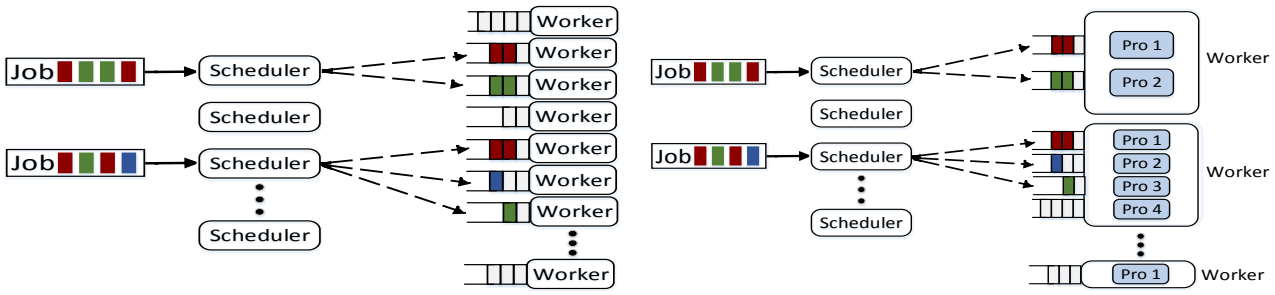
C. Resource Allocation

To achieve high resource utilization, we classify the workers into several categories based on their resources, and make the schedulers take into account resource utilization based on the resource requirements when the schedulers assign tasks to worker machines. On the other hand, DRS tries to place different intensive tasks (i.e., tasks with complementary resource demands) that are independent of each other in a worker with multi-processors so that the tasks can run in parallel achieving high resource utilization.

Figure 1(a) shows that the schedulers run tasks that are independent of each other in parallel on single processor workers. In Figure 1(a), the tasks of a job that do not depend on each other are marked in different colors, and the schedulers assign them to different workers. Figure 1(b) shows that the schedulers run tasks that do not depend on each other in parallel on multiprocessor workers, the independent tasks of a job are marked in different colors, and the schedulers assign them to different processors of a worker. The workers release the resources that are allocated to the tasks after the tasks complete. The workers periodically report their resource information to the corresponding schedulers, and the schedulers update the workers' resource information.

D. Task Scheduling

1) Fine-grained Waiting Time Prediction: Many previous works using sample-based techniques to place tasks based on the queue length at worker machines [2]. In previous work [27], schedulers place tasks only based on the queue length at worker machines. However, queue length is not



(a) Running independent tasks in parallel on multiple single processor workers

(b) Running independent tasks in parallel on multiprocessor workers

Fig. 1: Running independent tasks in parallel.

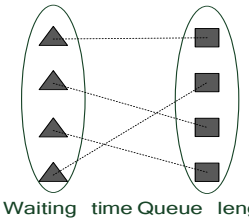


Fig. 2: Bipartite graph for waiting time and queue length.

enough for learning about the future waiting time of a task that will be assigned to a worker because queue length provides only a coarse-grained prediction of waiting time. For example, the scheduler tries to place a task on one of two workers, one of which has two 100ms tasks queued and the other of which has one 400ms task queued. The scheduler will place the task in the queue with only one task though that queue will lead to a 200ms longer waiting time. To handle this issue, we propose a fine-grained (reinforcement learning-based) approach to predict the waiting time of tasks.

Reinforcement Learning-based Approach: In order to predict the waiting time of tasks, we use the mutual reinforcement learning to accurately predict task duration [28].

We use a graph based model to learn the task duration. We model the relation between the workers and their resources by using a bipartite graph (see Figure 2). In the model, each edge linking a waiting time and queue length (expected queue length) represents the waiting time mapped to a worker with the queue length, and each rectangle represents a queue length. Let T be the set of tasks that are going to be put at the end of queues of workers. Let L be the set of queue lengths of different workers. The ith task of the set T is T^i, and the length of the ith queue in the set L is L^i. Suppose the T^i is going to be put at the end of the ith queue.

Given some known (labeled) examples of T and L. The following equation can be used to estimate the waiting time from their neighbors and queue length:

$$T_{c+1} = \gamma T_c + (1-\gamma) L_{c+1} \tag{12}$$

Correspondingly, the equation below can be used to estimate the queue length from their neighbors and waiting time:

$$L_{c+1} = \theta L_c + (1-\theta) T_{c+1} \tag{13}$$

Repeating h (the number of iterations) times, all tasks' waiting times can be estimated. The steps for iteratively finding waiting time and queue length are shown in Algorithm 3. The algorithm first checks if it is convergent (line 2). If it is not convergent, then the algorithm propagates tasks' waiting time by estimating tasks' waiting time from their neighbors and queue lengths (line 3). Then, the algorithm propagates queue length by estimating queue length from their neighbors and tasks' waiting time (line 4). Next, the algorithm clamps the labeled data of tasks' waiting time and queue length (line 5). The algorithm can estimate all tasks' waiting time and workers' queue length after repeating a certain number of times.

TABLE II: Parameter settings.

Parameter	Meaning	Setting	Parameter	Meaning	Setting
n	# of servers	10-50	α	Weight for CPU size	0.5
a	# of jobs	50-1000	β	Weight for Mem. size	0.5
m	# of tasks / job	10-20	θ	Weight for waiting time	0.5
l	# of resc. types	2	γ	Weight for queue length	0.5

Algorithm 3: Pseudocode for iteratively finding waiting time and queue length

Input: waiting time feature vector T_0, queue length feature vector L_0, weighting coefficients γ, θ, some manual labels of T_0 and/or L_0
Output: Waiting time T and queue length L.

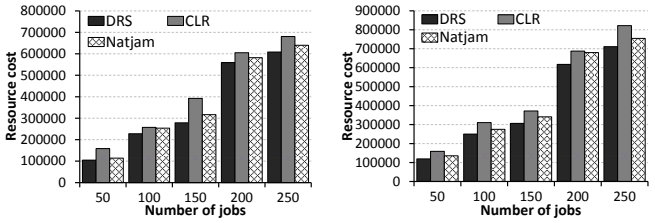
1 Set c=0
2 **while** *not convergence* **do**
3 Propagate waiting time. $T_{c+1} \leftarrow \gamma T_c + (1-\gamma) L_c$ // Formula (12)
4 Propagate queue length. $L_{c+1} \leftarrow \theta L_c + (1-\theta) T_{c+1}$ // Formula (13)
5 Clamp the labeled data of T_{c+1} and L_{c+1}
6 Set $c = c+1$
7 **return** L, T

IV. Performance Evaluation

We first conducted testbed experiments in a large-scale real cluster Palmetto [29], which is Clemson University's primary high-performance computing (HPC) resource. We tested various evaluation metrics and compared our method with four other methods. To further evaluate the performance of our method, we conducted experiments on the real-world Amazon EC2 [30]. In the experiment, the dependency relationship among tasks are created based on the starting time and ending time of tasks in the Google cluster trace [31]. In the following, we introduce our real testbed experimental results and experimental results on Amazon EC2, respectively.

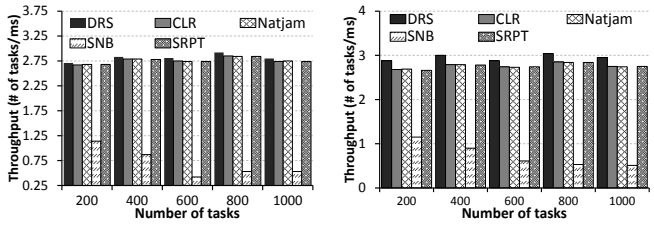
A. Experimental Results on the Real Cluster

We deployed our testbed in a large-scale cluster located in our university. We implemented our method and other four methods in our testbed, and we evaluated their performance with 50 servers. We compared the results of our method and the other four methods CLR [32], Natjam [33], SRPT [34] and SNB [35] in various scenarios. We used up to 1000 heterogeneous short jobs which have different resource requirements [36]. Each scheduler divides the job into tasks and



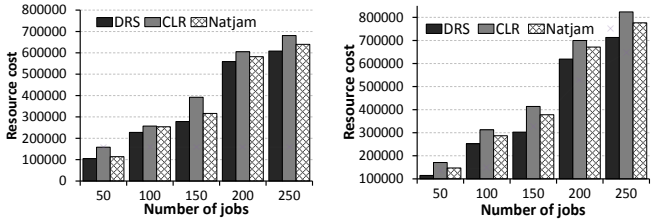
(a) Resource cost vs. number of jobs with 15 tasks/job (b) Resource cost vs. number of jobs with 20 tasks/job

Fig. 3: Performance of various methods on resource cost.



(a) Throughput vs. number of tasks with job submission rate 5 jobs/sec (b) Throughput vs. number of tasks with job submission rate 50 jobs/sec

Fig. 4: Performance of various methods on throughput on a real cluster.



(a) Resource cost for various methods with 15 tasks per job with 20 workers (b) Resource cost for various methods with 20 tasks per job with 30 workers

Fig. 5: Performance of various methods on resource cost on Amazon EC2.

forwards tasks to workers for processing. Table II shows the parameter settings in our analysis unless otherwise specified. Initially, we submitted jobs at a fixed rate and collected the results, then we varied the job submission rate by ten times of initial rate and conducted the experiments again to conclude the effects of changing job submission rate on schedulers.

Figure 3(a) shows the relationship between the resource cost and the number of jobs with 15 tasks per job. In Figure 3(a), we observe that the resource cost increases as the number of jobs increases. This is because the more the jobs, the more resources are needed for running the tasks of the jobs, and therefore the resource cost increases. In addition, we also see that the resource costs of Natjam, CLR and DRS follow DRS<Natjam<CLR. The reason is that DRS considers improving resource utilization by minimizing the resource cost, however Natjam and CLR allocate resource to tasks without the consideration of reducing resource cost. Figure 3(b) shows the relationship between the resource cost and the number of jobs with 20 tasks per job. From Figure 3(b), we see the similar results due to the same reasons. By examining Figure 3(a) and Figure 3(b), we find that the resource cost increases as the number of tasks per job increases. This is because the more tasks per job, the more likely a job consumes more resources, and thus the resource cost increases.

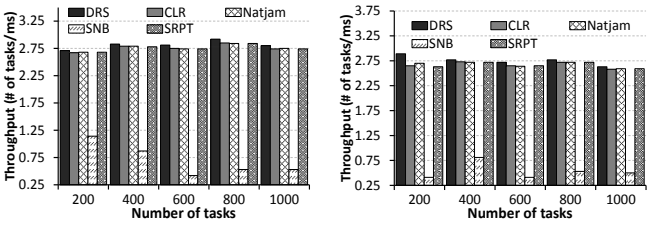
Figure 4(a) shows the relationship between throughput (# of tasks/ms) and the number of tasks with job submission rate 5 jobs per second. From Figure 4(a), we see that the throughput follows SNB<SRPT<CLR≈Natjam<DRS. This is because DRS considers the dependencies among tasks and schedules tasks that are independent of each other to different worker machines so that the tasks can run in parallel, which reduces response time and increases the throughput. Also, DRS uses the mutual reinforcement learning to accurately estimate tasks' waiting time in the queues of workers, and assigns tasks to workers that have less waiting time, which also reduces response time and increases the throughput. Figure 4(b) shows the relationship between throughput and the number of tasks with job submission rate 50 jobs per second. In Figure 4(b), we see the similar results due to the same reasons explained in Figure 4(a). By examining Figure 4(a) and Figure 4(b), we find that the throughput slightly increases as job submission rate increases. Both Figure 4(a) and Figure 4(b) suggest that DRS outperforms the other four methods.

B. Experimental Results on Amazon EC2

To fully test the performance of our method, we also conducted experiments on the real-world Amazon EC2. We evaluated the performance of DRS and the other methods with 30 servers. We used the similar heterogeneous short jobs with four workers for each scheduler. Each worker consists of 12GB memory and six 2.5 GHz processors. Then, DRS first submitted the jobs to schedulers and then the schedulers assigned jobs to worker machines. We used the average values of each metric as the performance on the metric.

Figure 5(a) shows the relationship between resource cost and number of jobs on Amazon EC2 with 20 workers. In Figure 5(a), we also see that the resource cost increases the number of jobs increases. Also, we find that the cost of DRS is lower than Natjam, and the cost of Natjam is slightly lower than that of CLR. This is because DRS assigns tasks to workers with the consideration of reducing the resource cost. Figure 5(b) shows the relationship between resource cost and number of jobs on Amazon EC2 with 30 workers. From Figure 5(b), we also observe the resource cost increases the number of jobs increases and the resource cost of DRS is lower than Natjam, and the resource cost of Natjam is slightly lower than that of CLR.

Figure 6(a) shows the relationship between throughput and the number of tasks with job submission rate 5 jobs per second. In Figure 6(a), we observe that the throughput follows SNB<SRPT<Natjam<CLR<DRS, which is consistent with our real testbed results. Figure 6(b) shows the relationship between throughput and the number of tasks with job submission rate 50 jobs per second. In Figure 6(b), we observe the throughput follows SNB<SRPT<CLR<Natjam<DRS, which is also consistent with our real testbed results. Comparing Figure 6(a) with Figure 4(a), and Figure 6(b) with Figure 4(b), we find that the throughput in Amazon EC2 is relatively lower than that in cluster. This is because the communication overhead in Amazon EC2 is relatively higher than that in the local cluster, and the number of servers is larger than that in Amazon EC2, which increases the chance that more tasks can find idle worker machines. From these figures, we find both



(a) Throughput vs. number of tasks with job submission rate 5 jobs/sec (b) Throughput vs. number of tasks with job submission rate 50 jobs/sec

Fig. 6: Performance of various methods on throughput on Amazon EC2.

our testbed results and experimental results on EC2 show that our method DRS outperforms the other methods.

V. Related Work

We first review existing works on improving throughput in scheduling. Then, we describe previous works on resource utilization related to job (task) scheduling. Finally, we indicate the advantages of our proposed DRS compared to the previous works.

A. Improving Throughput

Many works focus on maximizing throughput in scheduling [2], [37]–[40]. Zaharia *et al.* [37] proposed a delay scheduling: when job that should be scheduled next according to fairness cannot launch a local disk, it waits for a small amount of time, letting other jobs launch tasks instead. The work [37] shows delay scheduling achieves nearly optimal data locality in a variety of workloads and can increase throughput by up to 2x while preserving fairness. Raicu *et al.* [38] developed Falkon, a Fast and Light-weight task execution framework which integrates multi-level scheduling to separate resource acquisition from dispatch, and a streamlined dispatcher, and they showed Falkon throughput (487 tasks/sec) and scalability (to 54,000 executors and 2,000,000 tasks processed in just 112 minutes) are one to two orders of magnitude better than other systems used in production Grids. Wang *et al.* [39] struck the right balance between data-locality and load-balancing to maximize throughput. They presented a new queueing architecture and proposed a map task scheduling algorithm constituted by the Join the Shortest Queue policy together with the Max Weight policy. Bar-Noy *et al.* [40] proposed 2 and $(2+\varepsilon)$-ratio approximation algorithms to maximize the throughput of real time scheduling. However, all of the above works neglect dependency in scheduling to increase throughput. Unlike previous works, we consider task dependency and assign tasks that are of independent of each to different workers so that the independent tasks can run in parallel and thus increase throughput.

There is a large body of work on scheduling in distributed systems. Many existing works focused on independent tasks, while many tasks are not independent such as MapReduce tasks. Harchol-Balter [5] analyzed several natural task assignment policies and proposed a new one TAGS (Task Assignment based on Guessing Size). The TAGS algorithm is counterintuitive in many respects, including load unbalancing, non-work conserving, and fairness. However, the work [5] assumes tasks are independent. Dean and Barroso [41] proposed a scheduling approach using hedged requests where the client sends each request to two worker machines (workers) and cancels remaining outstanding requests when the first result is received. They also described tied requests, where clients send each request to two servers, but the servers communicate directly about the status of the request: when one server begins executing the request, it cancels the counterpart. However, each task must be scheduled independently to target an environment, thus tasks in a job cannot share the information. Scheduling highly parallel jobs that complete in hundreds of milliseconds poses a big challenge for task schedulers. To address this challenge, Ousterhout *et al.* [2] demonstrated that a decentralized, randomized sampling approach provides near-optimal performance while avoiding the throughput and availability limitations of a centralized design. However, it neglects the dependency between tasks and thus cannot reduce the latency for response time to the minimum by running tasks that are independent of each other in parallel. In contrast to the above works, our proposed DRS considers task dependency and assigns tasks that do not depend on each other to different workers (or different cores of a worker) so that the tasks can run in parallel, and thus further reduces the response time and increases the throughput.

B. Resource Utilization

There is a large body of work on resource allocation. However, many existing works ignore the resource constraints, and thus these works are not realistic. The work [7] demonstrates that over 50% of the jobs at Google have constraints about the machines that they can run on. Fair schedulers such as Quincy [42] and the Hadoop Fair Scheduler [37] take into account data locality, but these schedulers treat locality as a preference rather than a resource constraint (i.e., hard constraint), and can assign tasks non-locally if a suitable machine is not available. Thus these schedulers neglect jobs' resource constraints existing in practical scenarios. Max-min fairness has been widely studied in various areas such as networks, operating systems, and queuing systems [43]–[46]. However, previous works have a common unrealistic assumption, that is, the resources required by tasks are identical. Although Dominant Resource Fairness (DRF) [47] extends max-min fairness to multiple resource types (i.e., CPU, memory), it also assumes that the resource of a given type such as CPU are identical. Unlike previous works, our proposed DRS takes into account different resource constraints of tasks and assigns tasks to workers with the consideration of satisfying tasks' constraints on resources and optimizing resource allocation. Thus DRS can achieve high resource utilization while not violating tasks' constraints on resources.

In our proposed method, DRS splits jobs into tasks by taking into account the constraints of tasks, and assigns the tasks that do not depend on each other to machines (or different cores of a machine) so that these tasks can run in parallel and the response time can thus be further reduced. DRS also takes into account the constraints of jobs (tasks) on resources and schedules jobs (tasks) with the consideration of resource utilization.

VI. Conclusion

This paper presents DRS, a scheduler that provides low resource cost and high throughput with less overhead for parallel short jobs. DRS takes into account the dependency constraints and reduces the response time of jobs by running tasks that are independent of each other in parallel. Also, DRS uses the mutual reinforcement learning to estimate the tasks' waiting time in the queue of workers, and assigns tasks to workers with the consideration of tasks' waiting time in the queue of workers so that the response time can be reduced. Moreover, DRS assigns tasks to workers with the consideration of resource utilization, which helps DRS achieve high resource utilization. We compare our method with the existing methods in various scenarios using a large real cluster and Amazon EC2 cloud service, and demonstrate that DRS outperforms the exiting methods under both the real cluster and Amazon EC2 cloud service. In the future, we will consider the preemption in our scheduler so that DRS can satisfy jobs (tasks) with different priorities. Also, we will consider the tolerance of failures [48] so that our scheduler can be more robust.

Acknowledgements

This research was supported in part by U.S. NSF grants NSF-1404981, IIS-1354123, CNS-1254006, IBM Faculty Award 5501145 and Microsoft Research Faculty Fellowship 8300751.

References

[1] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Batch sampling: Low overhead scheduling for sub-second prallel jobs. *University of California, Berkeley*, 2012.

[2] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proc. of ACM SOSP*, 2013.

[3] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of USENIX ATC*, 2015.

[4] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[5] M. Harchol-Balter. Task assignment with unknown duration. *Journal of the ACM*, 49(2):260–288, March 2002.

[6] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[7] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proc. of SoCC*, 2011.

[8] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Eurosys*, 2013.

[9] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proc. of EuroSys*, 2016.

[10] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proc. of ACM EuroSys*, London, 2016.

[11] J. Liu, H. Shen, and L. Chen. CORP: Cooperative opportunistic resource provisioning for short-lived jobs in cloud systems. In *Proc. of IEEE CLUSTER*, 2016.

[12] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. Hacigms. Activesla: a profit-oriented admission control framework for database-as-a-service providers. In *Proc. of SoCC*, 2011.

[13] C. Qiu, H. Shen, and L. Chen. Probabilistic demand allocation for cloud service brokerage. In *Proc. of INFOCOM*, 2016.

[14] E. Boutin, J. Ekanayake, W. Lin, B. Shi, and J. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, 2014.

[15] I. Pietri, G. Juve, E. Deelman, and R. Sakellariou. A performance model to estimate execution time of scientific workflows on the cloud. In *Proc. of SC*, 2014.

[16] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI*, 2004.

[17] R. Buyya, M. Murshed, D. Abramson, and S. Venugopal. Scheduling parameter sweep applications on global grids: A deadline and budget constrained cost-time optimisation algorithm. *Software: Practice and Experience (SPE)*, 35(5):491–512, April 2005.

[18] P. Xiong, Y. Chi, S. Zhu, H. Moon, C. Pu, and H. Hacigms. Intelligent management of virtualized resources for database systems in cloud environment. In *Proc. of ICDE*, 2011.

[19] J. Liu, H. Shen, and X. Zhang. A survey of mobile crowdsensing techniques: A critical component for the internet of things. In *Proc. of ICCCN Workshop on ContextQoS*, 2016.

[20] P. Xiong, Z. Wang, S. Malkowski, Q. Wang, D. Jayasinghe, and C. Pu. Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach. In *Proc. of ICDCS*, 2011.

[21] J. Liu, H. Shen, and H. Hu. Load-aware and congestion-free state management in network function virtualization. In *Proc. of ICNC*, 2017.

[22] Cplex linear program solver. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/ [accessed in Jul. 2016].

[23] H. Shen, J. Liu, K. Chen, J. Liu, and S. Moyer. SCPS: A social-aware distributed cyber-physical human-centric search engine. *IEEE Transactions on Computers (TC)*, 64:518–532, 2015.

[24] C. Qiu, H. Shen, and L. Yu. Energy-efficient cooperative broadcast in fading wireless networks. In *Proc. of INFOCOM*, 2014.

[25] D. Shah. *Gossip Algorithms*. Foundations and Trends in Networking, 2009.

[26] J. Liu, L. Yu, H. Shen, Y. He, and J. Hallstrom. Characterizing data deliverability of greedy routing in wireless sensor networks. In *Proc. of SECON*, Seattle, June 2015.

[27] K. Jagannathan, M. Markakis, E. Modiano, and J. N. Tsitsiklis. Queue-length asymptotics for generalized max-weight scheduling in the presence of heavy-tailed traffic. *TON*, 20(4):1096–1111, 2012.

[28] J. Liu, H. Shen, and L. Yu. Question quality analysis and prediction in community question answering services with coupled mutual reinforcement. *TSC*, PP(99):1–14, 2015.

[29] Palmetto cluster. http://citi.clemson.edu/palmetto/ [accessed in Jul. 2016].

[30] Amazon EC2. http://aws.amazon.com/ec2 [accessed in Jul. 2016].

[31] Google trace. https://code.google.com/p/googleclusterdata/ [accessed in Jul. 2016].

[32] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proc. of SoCC*, 2012.

[33] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Proc. of SoCC*, 2013.

[34] A. Balasubramanian, A. Sussman, and N. Sadeh. Decentralized preemptive scheduling across heterogeneous multi-core grid resources. In *Proc. of JSSPP*, 2013.

[35] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Trans. on Computer Systems*, 21(2):207–233, 2003.

[36] Z. Li, H. Shen, W. Ligon, and J. Denton. An exploration of designing a hybrid scale-up/out hadoop architecture based on performance measurements. *TPDS*, PP(99):1–1, 2016.

[37] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of EuroSys*, 2010.

[38] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework. In *Proc. of SC*, 2007.

[39] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang. A throughput optimal algorithm for map task scheduling in mapreduce with data locality. *SIGMETRICS Performance Evaluation Review*, 40(4):33–42, 2013.

[40] A. Bar-Noy, S. Guha, Y. Katz, J. Naor, B. Schieber, and H. Shachnai. Throughput maximization of real-time scheduling with batching. *ACM Trans. Algorithms*, 5(2):1–17, 2009.

[41] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[42] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proc. of SOSP*, 2009.

[43] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, second edition edition, 1992.

[44] D. Raz, H. Levy, and B. Avi-Itzhak. A resource-allocation queueing fairness measure. In *Proc. of SIGMETRICS*, pages 130–141, June 2004.

[45] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *Proc. of OSDI*, 1994.

[46] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an m/gi/1. In *Proc. of SIGMETRICS*, 2003.

[47] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, I. Stoica, and S. Shenker. Dominant resource fairness: Fair allocation of multiple resource types. In *Proc. of NSDI*, 2011.

[48] J. Liu and H. Shen. A low-cost multi-failure resilient replication scheme for high data availability in cloud storage. In *Proc. of HiPC*, 2016.