Online Job Scheduling in Distributed Machine Learning Clusters

Yixin Bao*, Yanghua Peng*, Chuan Wu*, Zongpeng Li†
*Department of Computer Science, The University of Hong Kong, Email: {yxbao,yhpeng,cwu}@cs.hku.hk
*Department of Computer Science, University of Calgary, Email: zongpeng@ucalgary.ca

***Abstract*—Nowadays large-scale distributed machine learning systems have been deployed to support various analytics and intelligence services in IT firms. To train a large dataset and derive the prediction/inference model, e.g., a deep neural network, multiple workers are run in parallel to train partitions of the input dataset, and update shared model parameters. In a shared cluster handling multiple training jobs, a fundamental issue is how to efficiently schedule jobs and set the number of concurrent workers to run for each job, such that server resources are maximally utilized and model training can be completed in time. Targeting a distributed machine learning system using the parameter server framework, we design an online algorithm for scheduling the arriving jobs and deciding the adjusted numbers of concurrent workers and parameter servers for each job over its course, to maximize overall utility of all jobs, contingent on their completion times. Our online algorithm design utilizes a primal-dual framework coupled with efficient dual subroutines, achieving good long-term performance guarantees with polynomial time complexity. Practical effectiveness of the online algorithm is evaluated using trace-driven simulation and testbed experiments, which demonstrate its outperformance as compared to commonly adopted scheduling algorithms in today's cloud systems.**

I. Introduction

Most leading IT companies have deployed distributed machine learning (ML) systems, which train various machine learning models over large datasets for providing AI-driven services. For example, Google uses its scalable ML framework, TensorFlow, to power products such as Google Photos and Google Cloud Speech [1]. Microsoft employs its distributed cognitive toolkit, CNTK, for speech recognition and image related learning tasks [2]. Baidu developed a PArallel Distributed Deep LEarning (PaddlePaddle) system and extensively uses large-scale ML for advertising, group shopping, etc. [3]. In these scenarios, large ML clusters with hundreds or thousands of (GPU) servers are deployed, where many internal/external training jobs are run to derive various prediction/inference models, *e.g.*, Deep Neural Networks (DNNs), Logistic Regression (LR), and Latent Dirichlet Allocation.

Training machine learning models is typically resource intensive and time consuming. For example, it takes 23.4 hours to train a GoogLeNet model using the ImageNet dataset on a Titan supercomputer server with 32 NVIDIA K20 GPUs [4][5]. A fundamental challenge faced by an ML cluster operator is how to efficiently schedule submitted training jobs to maximally exploit available server resources (especially the expensive GPU cards), and to complete training in an expedited fashion. In representative distributed ML systems [1][2][3][6], training is done in parallel by multiple concurrent *workers*. There are two parallelism models: *data parallelism*, where the input dataset is partitioned among the workers, and each worker has a local copy of the entire ML model, computes model parameter changes using allocated data chunks, and exchanges computation results with other workers to come up with the right global parameter updates [7][6]; *model parallelism*, where the ML model is partitioned among workers and each worker updates part of the parameters using the entire dataset [8]. Data parallelism has been more widely adopted than model parallelism, given that most ML models can be entirely stored in the memory of modern GPUs, eliminating the need for model partition. For example, latest NVIDIA GPU models (TITAN X and Tesla) have a memory of 16GB or 24GB, sufficient for most state-of-the-art models (*e.g.*, [9][10]). We focus on data parallelism in this work.

A typical approach to exchange parameter changes among workers is through a parameter server framework [7][8]: There are one or multiple *parameter servers* (typically implemented as virtualized instances using virtual machines or containers), and model parameters are evenly divided and maintained by the parameter servers. In each training iteration, a worker sends its computed parameter changes to the parameter servers; the parameter servers update their maintained parameters respectively, and send updated parameters back to the worker. The number of concurrent workers, as well as the number of parameter servers to support parameter exchange, decide the training speed and completion time of a job.

How are training jobs scheduled in the existing ML systems? Google uses Borg [11] as the ML cluster scheduler. Microsoft, Tencent, and Baidu use customized versions of YARN-like schedulers [12] for managing ML jobs, based on our exchanges with their employees. The default scheduling policies of these schedulers are typically FIFO (as in Spark [13]), Dominant Resource Fairness Scheduling [14] (as in YARN [12] and Mesos [15]), or priority-based greedy approaches (as in Borg [11]). To our knowledge, none of these systems allow a varying number of concurrent workers in a training job, which is specified by the job owner and remains fixed throughout the training course. Such static resource allocation to jobs may not fully utilize the (often expensive)

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17204715, 17225516, C7036-15G (CRF), grants NSFC 61628209 and NSFC 61571335, HKU URC Matching Funding, and Hubei Science Foundation 2016CFA030, 2017AAA125.

ML cluster resources, preventing the best training speeds.

We propose an online job scheduling algorithm, tailored for operating a shared ML cluster running multiple training jobs. The algorithm, referred to as *OASiS*, computes the best job execution schedule upon the arrival of each job, based on projected resource availability in the future course and potential job utility to achieve (contingent on its completion time). Judging whether the potential job utility outweighs resource consumption, the algorithm decides admitting the job or not, and runs the job according to the best schedule if admitted. With the schedule, the numbers of workers and parameter servers and their deployment on servers are dynamically adjusted during the course of the job, for expedited training adapting to resource availability at different times. Over the long run, we seek overall job utility maximization.

Our online algorithm design utilizes an online primal-dual framework coupled with dual subroutines, to efficiently tackle the combinatorial online optimization problem. Based on the primal-dual framework, we maintain meticulously computed (dual) resource prices according to time-varying resource consumption levels (less resources when new jobs are admitted and more when jobs are completed), and decide job admission and resource allocation accordingly. Given the resource prices, the dual subroutines include efficient, optimal algorithms to compute the best schedule of worker and parameter server deployment for each job, exploiting a dynamic programming structure of the underlying multi-timeslot multi-dimensional resource packing problem.

We rigorously prove polynomial running time of our online algorithm, and its long-term performance guarantee in terms of a good competitive ratio in total job utility. We evaluate practical effectiveness of *OASiS* using trace-driven simulation and testbed experiments, by implementing it as a new scheduler module in Kubernetes [16] for MXNet – a popular distributed machine learning platform [6]. The results show that *OASiS* outperforms commonly adopted scheduling policies.

II. Related Work

A. Distributed Machine Learning Systems

A number of distributed ML frameworks have been designed and deployed, *e.g.*, TensorFlow [1], CNTK [2], PaddlePaddle [3], MXNet [6]. The parameter server framework, mainly due to Li *et al.* [7], has been incorporated in some of them (*e.g.*, [6][8]). In these systems, a static set of workers are employed; new workers are deployed only upon failure of existing ones. Most adopt Borg or YARN-like schedulers for ML cluster management [11][12].

Recently in the literature, Dorm [17] advocates partitioning an ML cluster, runs one ML application per partition, and dynamically resizes the partitions for resource efficiency and fairness, by solving a mixed integer linear program (MILP) using a standard solver. In comparison, we design an online algorithm to guide resource allocation over time with proven performance. Dolphin [18] solves a cost-minimizing problem to find an optimal number of nodes to use for an ML job, and reconfigures the system dynamically. It focuses on runtime optimization of one ML job, instead of optimal resource allocation among multiple concurrent jobs. Similarly, Yan *et al.* [19] develop performance models to quantify the impact of model and data partitioning and system provisioning on training performance of a DNN, where online job scheduling and resource sharing are not considered.

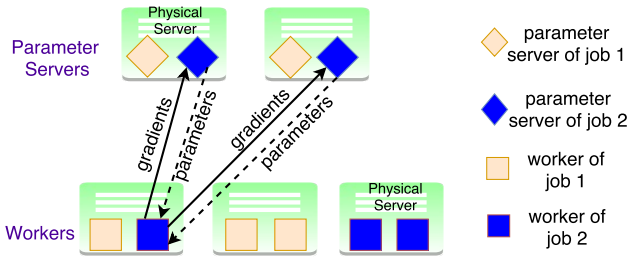

Fig. 1: The distributed machine learning system

B. Job Scheduling and Resource Allocation in Cloud Systems

In the offline setting, Huang *et al.* [20] and Chen *et al.* [21] study cloud job scheduling problems, targeting max-min fairness among jobs. For online scheduling, Azar *et al.* [22] propose an online preemptive job scheduling algorithm achieving a constant competitive ratio, for jobs running on a single machine with constant job utility. Lucier *et al.* [23] propose an efficient heuristic for online job scheduling with preemption, aiming to maximize total value of all jobs. The resources allocated to each job are fixed over time and the job value is not influenced by completion time. Zhou *et al.* [24] and Zhang *et al.* [25] design mechanisms for online cloud resource allocation and pricing, where no adjustment of allocated resources in a job is considered.

Xiao *et al.* [26] design a scheduler for automatic scaling of Internet applications in a cloud, targeting high demand satisfaction ratio and short request-response time. TetriSched [27] enables resource scaling by periodically solving a schedule optimization problem among all pending jobs to compute their amounts of resources in need. These work do not provide theoretical guarantee for long-term performance.

III. Problem Model

A. Distributed Machine Learning System

Fig. 1 illustrates an ML cluster, where a set of I training jobs are submitted in an online fashion during timespan $1, 2, \ldots, T$. The training jobs come with large input datasets, and derive potentially different ML models using *data parallel* training and the parameter server framework [7]. A job i arrives at time $a_i \in [T]$,[1] using a number of workers and parameter servers for model training.

Workers and parameter servers are implemented on virtual machines (VMs) or containers in the physical servers. The ML cluster hosts H physical servers for worker deployment. Each machine $h \in [H]$ has a capacity c_h^r of type-r resource. There are K other physical servers for running parameter servers, and each server $k \in [K]$ has a capacity c_k^r of type-r resource. Let R

[1]We define $[X] = \{1, 2, \ldots, X\}$ throughout the paper, where X can be different quantities.

be the total number of resource types, including GPU, CPU, memory, disk storage and bandwidth capacity of the server NIC. We practically assume two types of physical machines for running workers and parameter servers separately, given that parameter servers are typically placed on machines with high bandwidth but without GPU, while workers run on GPU servers. Such a separation between workers and parameter servers has been witnessed in existing ML systems [7][8].

Workers and parameter servers are customized for each job, and not shared among different jobs. Each worker (parameter server) of job i occupies a w_i^r (s_i^r) amount of type-r resource, $\forall r \in [R]$. An amount of bandwidth b_i (B_i) is reserved for each worker (parameter server) of job i, *i.e.*, $b_i = w_i^{\text{bandwidth}}$ ($B_i = s_i^{\text{bandwidth}}$). We do not distinguish upload and download bandwidth, but assume they are symmetric. Bandwidth reservation for a VM or container is common for accelerated computing in cloud platforms, to guarantee data transfer performance of each instance, *e.g.*, the reserved bandwidth of EC2 GPU instance P2 on AWS is 10Gbps or 20Gbps [28].

B. Asynchronous Training Workflow

The input dataset to a training job is stored in a distributed storage system (*e.g.*, HDFS [29]). The dataset is divided into equal-sized *data chunks* trained by different workers. Each data chunk is further divided into equal-sized *mini-batches*.

Upon start, a worker fetches a data chunk.[2] Then the worker processes the first mini-batch in the data chunk, *i.e.*, computes what changes to be made to the parameters (to approach their optimal values) in the ML model, using data in the mini-batch. Parameter changes are typically expressed as *gradients* (directions of changes), and a distributed stochastic gradient descent method is typically used by workers to jointly improve the parameters [7]. For example, when training an LR model for ad click-through-rate prediction, parameters are the weights of features (*e.g.*, text, image used in an ad) in the prediction model, and gradients are the changes of weights [30].

After processing a mini-batch, the worker sends gradients to the parameter servers for parameter updates. The parameter servers in a job are usually responsible for an evenly divided share of the parameters. In the above example, if there are two parameter servers, each will be responsible for half of the weights, and gradients computed by a worker are divided and sent to parameter servers maintaining respective weights. Upon receiving updated parameters from all parameter servers, the worker continues computing gradients using the next mini-batch, and so on. After an entire data chunk is processed, the worker continues training the next data chunk assigned to it.

Fig. 2 illustrates the *asynchronous training* workflow in our system, *i.e.*, the training progress at different workers in a job is not synchronized and each parameter server updates its parameters each time upon receiving gradients from a worker. In the above example, a parameter server updates its weights using a formula like $new\ weight = old\ weight - stepsize \times gradient\ computed\ by\ the\ worker$,

[2]The ML framework, *e.g.*, PaddlePaddle, assigns data chunks to workers.

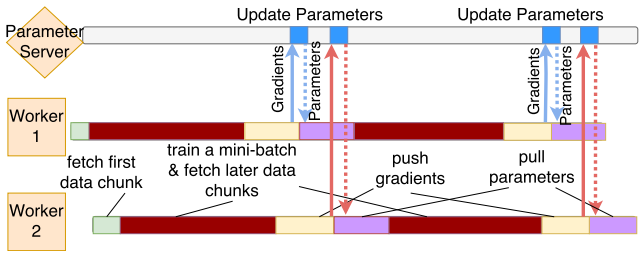

Fig. 2: Workflow in a training job

and then sends updated weights back to the worker. Another representative training mode in today's ML systems is synchronous training, where training progress at all workers is synchronized and each parameter server updates its parameters after it has collected gradients from all workers in each training iteration (*i.e.*, training of one mini-batch). Asynchronous training achieves better bandwidth utilization, as gradients and updated parameters are sent from/to workers at different times, and hence potentially faster convergence. Further, model accuracy achieved with asynchronous training is not affected by changes of worker population through the course [7][8] (as what we advocate), while it varies with synchronous training if different numbers of concurrent workers are used [5][19].

Let N_i be the number of input data chunks in job i, each divided into M_i mini-batches. Let τ_i denote the training time (gradient computation) for each mini-batch in job i, which is assumed to be equal for all mini-batches on all workers in the same job, given the same resource allocation per worker. Let e_i be the size of gradients produced by each worker of job i after processing a mini-batch, which is the same as the size of updated parameters that the worker will receive from all parameter servers, since the total numbers of gradients and parameters are always the same and both use the same float point representation [5]. The time for sending gradients to or receiving updated parameters from all parameter servers can be computed as $\frac{e_i}{b_i}$ (bandwidth at a parameter server is typically large enough to receive gradients /send parameters from/to multiple workers). When training ResNet-152 model on ImageNet dataset [9][4], each data chunk is 128MB in size, a mini-batch is about 6MB in size, and training one mini-batch takes about one second, while training a data chunk takes less than one minute; the size of gradients/parameters exchanged between a worker and parameter servers is about 241MB.

We ignore worker/parameter server setup time, since the image containing the training program can be pre-stored in a physical machine or fetched in a very short time (*e.g.*, a container image of hundreds of MBs can be fetched within seconds in a 10Gbps network). We also ignore the time for a worker to fetch data chunks from distributed storage, since a worker only needs to explicitly retrieve the first chunk, and fetching time of later chunks can be hidden behind training through pipelining. Fetching one data chunk takes much shorter time than training, *e.g.*, less than 1s in a 10Gbps network for a 128MB chunk. With asynchronous training, the computation time at a parameter server for updating

parameters using gradients from only one worker is very small (around tens of milliseconds in ResNet-152) and hence negligible too.

In an ML job, input data chunks can be repeatedly trained for multiple rounds. An *epoch* [8] is the duration when all data chunks are trained once. A training job i stops after E_i epochs in our system.

C. Offline Optimization Problem

Upon arrival of an ML job i at a_i, the following decisions are made: (i) Whether the job should be admitted, denoted by a binary variable x_i: $x_i = 1$ if job i is admitted, and $x_i = 0$, otherwise. Admission control is common in cloud management systems [11][12], and jobs that are not admitted can be queued or resubmitted at a later time beyond T. (ii) The number of workers of job i to deploy on physical server $h \in [H]$ in each time slot at and after a_i, indicated by integer variable $y_{ih}(t)$. (iii) The number of parameter servers of job i to deploy on physical server $k \in [K]$ in each time slot at and after a_i, denoted by integer variable $z_{ik}(t)$.

Given that it is not practical to adjust worker and parameter server deployment frequently, the length of each time slot is potentially much larger than the duration of an epoch. For example, one time slot can be 1 hour or longer.

Let $\hat{t}_i$ be the completion time slot of job i. Each job i has a non-negative utility $f_i(\hat{t}_i - a_i)$, non-increasing with $\hat{t}_i - a_i$, specifying the job's value in terms of different completion times [20]. The offline optimization problem to maximize overall utility is formulated as follows. Important notation is summarized in Table I.

$$\max \sum_{i\in[I]} x_i f_i(\hat{t}_i - a_i) \tag{1}$$

subject to:

$$\sum_{t\in[T]}\sum_{h\in[H]} y_{ih}(t) \geq E_i N_i M_i(\tau_i + 2e_i/b_i)x_i, \forall i \in [I] \tag{2}$$

$$\sum_{h\in[H]} y_{ih}(t) \leq N_i x_i, \forall i \in [I], t \in [T] : t \geq a_i \tag{3}$$

$$\sum_{i\in[I]} w_i^r y_{ih}(t) \leq c_h^r, \forall t \in [T], r \in [R], h \in [H] \tag{4}$$

$$\sum_{i\in[I]} s_i^r z_{ik}(t) \leq c_k^r, \forall t \in [T], r \in [R], k \in [K] \tag{5}$$

$$\sum_{h\in[H]} y_{ih}(t) b_i \leq \sum_{k\in[K]} z_{ik}(t) B_i, \forall i \in [I], t \in [T] \tag{6}$$

$$\sum_{k\in[K]} z_{ik}(t) \leq \sum_{h\in[H]} y_{ih}(t), \forall i \in [I], t \in [T] \tag{7}$$

$$\hat{t}_i = \arg\max_{t\in[T]}\{\sum_{h\in[H]} y_{ih}(t) > 0\}, \forall i \in [I] \tag{8}$$

$$y_{ih}(t) = 0, \forall i \in [I], h \in [H], t < a_i \tag{9}$$

$$z_{ik}(t) = 0, \forall i \in [I], k \in [K], t < a_i \tag{10}$$

$$x_i \in \{0, 1\}, \forall i \in [I] \tag{11}$$

$$y_{ih}(t) \in \{0, 1, \ldots\}, \forall i \in [I], t \in [T], h \in [H] \tag{12}$$

$$z_{ik}(t) \in \{0, 1, \ldots\}, \forall i \in [I], t \in [T], k \in [K] \tag{13}$$

Constraint (2) ensures that for each admitted job i, a sufficient number of workers are deployed to accomplish training of its dataset for E_i epochs. Here, $\tau_i + 2e_i/b_i$ is the time for training a mini-batch, sending gradients to parameter servers, and receiving updated parameters from parameter servers. $E_i N_i M_i$ is the total count of mini-batches trained in the job. $\sum_{t\in[T]}\sum_{h\in[H]} y_{ih}(t)$ indicates the total amount of work time that all deployed workers in job i provide. (3) specifies the concurrent number of workers of job i should be no more than the number of data chunks N_i, to ensure that one data chunk is processed by at most one worker in each time slot (such that data chunks are trained evenly over time). (4) and (5) are resource capacity constraints on physical machines for worker and parameter server deployment, respectively. (6) guarantees that the total bandwidth of parameter servers is no smaller than total bandwidth of all workers in each job, *i.e.*, parameter servers will not be bottlenecks during gradient/parameter exchange. (7) upper bounds the number of parameter servers by the number of workers at any time in each job, which is common in practical ML systems [7][8]. (8) gives the completion time slot of job i. (9) and (10) set worker and parameter server numbers to 0 before a job's arrival.

The optimization problem involves integer variables and non-conventional constraints in (8). We design an efficient online algorithm to solve it in an online fashion, without assuming knowledge of any future job arrivals.

TABLE I: Notation

I	# of jobs	T	system timespan
$\hat{t}_i$	completion time of job i	a_i	arrival time of job i
R	# of resource types	N_i	# of data chunks in i
x_i	accept job i or not	$f_i(\cdot)$	job i's utility

E_i	# of training epochs for job i
M_i	# of mini-batches in a data chunk of job i
$H(K)$	# of servers to deploy workers (parameter servers)
$c_h^r(c_k^r)$	capacity of type-r resource on server h (k) to deploy workers (parameter servers)
$w_i^r(s_i^r)$	type-r resource of a worker (parameter server) in i
$y_{ih}(t)$	# of workers of job i deployed on server h in t
$z_{ik}(t)$	# of parameter servers of i deployed on server k in t
$b_i(B_i)$	bandwidth of a worker (parameter server) of job i
τ_i	time to train a mini-batch in job i
e_i	size of gradients/parameters exchanged between a worker and parameter servers in job i

IV. Online Algorithm

A. Problem Reformulation

To circumvent the non-conventional constraint (8), we reformulate problem (1) into the following integer linear program (ILP). Here $\mathcal{L}_i$ is the set of feasible schedules for jobs i, each corresponding to the set of decisions $(y_{ih}(t), z_{ik}(t), \forall h \in [H], k \in [K], t \in [T])$ satisfying constraints (2)(3)(6)(7)(9)-(13). There is potentially an exponential number of feasible schedules for each job, due to combinatorial nature of those constraints. Decision variables in the ILP are binary variables x_{il}, indicating whether job i is admitted and scheduled according to schedule $l \in \mathcal{L}_i$ or not, $\forall i \in [I], l \in \mathcal{L}_i$. Job i's

completion time according to schedule l is t_{il}. $y^l_{ih}(t)$ ($z^l_{ik}(t)$) is the given number of workers (parameter servers) on server h (k) in t in job i's schedule l (not decision variables in (14)).

$$\max_{\mathbf{x}} \sum_{i\in[I]} \sum_{l\in\mathcal{L}_i} x_{il} f_i(t_{il} - a_i) \tag{14}$$

s.t.

$$\sum_{i\in[I]} \sum_{l:t\in l, h\in(t,l)} w_i^r y^l_{ih}(t) x_{il} \le c_h^r, \forall t \in [T], r \in [R], h \in [H] \tag{15}$$

$$\sum_{i\in[I]} \sum_{l:t\in l, k\in(t,l)} s_i^r z^l_{ik}(t) x_{il} \le c_k^r, \forall t \in [T], r \in [R], k \in [K] \tag{16}$$

$$\sum_{l\in\mathcal{L}_i} x_{il} \le 1, \forall i \in [I] \tag{17}$$

$$x_{il} \in \{0,1\}, \forall i \in [I], l \in \mathcal{L}_i \tag{18}$$

We use $t \in l, h \in (t,l), k \in (t,l)$ to indicate that schedule l uses server h to deploy worker(s) and server k to deploy parameter server(s) for job i in t. (14), (15) and (16) are equivalent to (1), (4) and (5), respectively. (17) and (18) correspond to (2)(3)(6)-(13). Problems (1) and (14) are equivalent since a feasible solution to (1) has a corresponding feasible solution to (14), and vice versa, with the same objective values. Though the number of variables in (14), x_{il}'s, is potentially exponential, we will design an efficient online algorithm to solve (14) in polynomial time, exploiting the primal-dual framework [31]. We formulate the dual of (14) by relaxing integrality constraints (18) and associating dual variables $p_h^r(t)$, $q_k^r(t)$ and μ_i with (15), (16) and (17), respectively.

$$\min \sum_{i\in[I]} \mu_i + \sum_{t\in[T]} \sum_{h\in[H]} \sum_{r\in[R]} p_h^r(t) c_h^r + \sum_{t\in[T]} \sum_{k\in[K]} \sum_{r\in[R]} q_k^r(t) c_k^r \tag{19}$$

$$s.t.\ \mu_i \ge f_i(t_{il} - a_i) - \sum_{t\in l} \sum_{h\in(t,l)} \sum_{r\in[R]} p_h^r(t) w_i^r y^l_{ih}(t) - \sum_{t\in l} \sum_{k\in(t,l)} \sum_{r\in[R]} q_k^r(t) s_i^r z^l_{ik}(t), \forall i \in [I], l \in \mathcal{L}_i \tag{20}$$

$$p_h^r(t) \ge 0, q_k^r(t) \ge 0, \forall t \in [T], h \in [H], k \in [K], r \in [R]$$
$$\mu_i \ge 0, \forall i \in [I]$$

The dual variable $p_h^r(t)$ ($q_k^r(t)$), associated with the primal capacity constraint on server h (k), can be interpreted as the unit cost for type-r resource on the server in t. Then $\sum_{t\in l} \sum_{h\in(t,l)} \sum_{r\in[R]} p_h^r(t) w_i^r y^l_{ih}(t)$ ($\sum_{t\in l} \sum_{k\in(t,l)} \sum_{r\in[R]} q_k^r(t) s_i^r z^l_{ik}(t)$) is the total resource cost of all workers (parameter servers) of job i with schedule l. The RHS of (20) is job utility minus overall resource cost for job i with schedule l. The following should hold to minimize the dual objective: $\mu_i = \max\{0, \max_{l\in\mathcal{L}_i} \text{RHS of (20)}\}$. Hence, μ_i can be nicely interpreted as the payoff of admitting job i according to the best schedule l^*:

$$l^* = \arg\max_{l\in\mathcal{L}_i} \quad \text{RHS of (20)} \tag{21}$$

B. Online Algorithm

These observations inspire the design of an online algorithm: Upon arrival of job i, we compute the best schedule l^* of job i (assuming job admitted). Then we check if the RHS of (20) achieved by l^* is positive. If so ($\mu_i > 0$, positive payoff), we accept job i and run it according to l^* ($x_{il^*} = 1$); otherwise (zero payoff), job i is rejected ($x_{il} = 0, \forall l \in \mathcal{L}_i$). The rationale is that, as resources are limited, we wish to accept jobs with larger utility and lower resource consumption, to maximize (14). A positive payoff indicates that the job utility is high enough to justify resource consumption, and we schedule the job in a way that maximizes its payoff.

Algorithm 1 *OASiS*: Online Algorithm for Scheduling ML Jobs

Input: $T, c_h^r, c_k^r, \forall h \in [H], k \in [K], r \in [R]$
Output: $x_i, y_{ih}(t), z_{ik}(t), \forall i \in [I], t \in [T], h \in [H], k \in [K]$
1: Initialize $y_{ih}(t) = 0, z_{ik}(t) = 0, g_h^r(t) = 0, v_k^r(t) = 0, p_h^r(t) = P_h^r(0), q_k^r(t) = Q_k^r(0), \forall i \in [I], t \in [T], h \in [H], k \in [K], r \in [R]$
2: **Upon** arrival of job i **do**
3: Compute the best schedule l^* and payoff μ_i for job i using Alg. 2
4: **if** $\mu_i > 0$ **then**
5: Set $x_i = 1$
6: Set $y_{ih}(t), z_{ik}(t)$ according to schedule l^*, $\forall t \in l^*, h \in (t,l^*), k \in (t,l^*)$
7: Update $g_h^r(t) = g_h^r(t) + w_i^r y_{ih}(t), \forall t \in l^*, h \in (t,l^*), r \in [R]$
8: Update $p_h^r(t) = P_h^r(g_h^r(t)), \forall t \in l^*, h \in (t,l^*), r \in [R]$
9: Update $v_k^r(t) = v_k^r(t) + s_i^r z_{ik}(t), \forall t \in l^*, k \in (t,l^*), r \in [R]$
10: Update $q_k^r(t) = Q_k^r(v_k^r(t)), \forall t \in l^*, k \in (t,l^*), r \in [R]$
11: Accept and launch job i according to schedule l^*
12: **else**
13: Set $x_i = 0$ and reject job i
14: **end if**
15: **end upon**

To implement this idea, we need to resolve the following: (i) Solve (21) to find the best schedule l^* for job i. Simply enumerating all feasible schedules is not practical, given the exponential size of set $\mathcal{L}_i$. We will design an efficient subroutine to produce l^* in polynomial time in Sec. IV-C. (ii) Compute dual resource prices $p_h^r(t)$'s and $q_k^r(t)$'s, to ensure a positive payoff for job schedules achieving high utilities (if there are enough resources to accommodate them), and non-positive payoff for job schedules resulting in low utilities or without available resources.

The sketch of our online algorithm, *OASiS*, is in Alg. 1. In line 3, Alg. 2 is the subroutine to compute l^*. In line 7 (9), $g_h^r(t)$ ($v_k^r(t)$) records the amount of allocated type-r resource on server h (k) for (future) time slot t. In lines 8 and 10, we update dual resource prices using carefully designed price functions $P_h^r(\cdot)$ and $Q_k^r(\cdot)$, respectively:

$$P_h^r(g_h^r(t)) = L_1 \Big(\frac{U_1^r}{L_1}\Big)^{\frac{g_h^r(t)}{c_h^r}}, \quad Q_k^r(v_k^r(t)) = L_2 \Big(\frac{U_2^r}{L_2}\Big)^{\frac{v_k^r(t)}{c_k^r}} \tag{22}$$

$$\text{where} \quad U_1^r = \max_{i\in[I]} \frac{f_i(\lceil E_i M_i(\tau_i + 2e_i/b_i)\rceil - a_i)}{w_i^r}, \forall r \in [R] \tag{23}$$

$$U_2^r = \max_{i\in[I]} \frac{f_i(\lceil E_i M_i(\tau_i + 2e_i/b_i)\rceil - a_i)}{s_i^r}, \forall r \in [R] \tag{24}$$

$$L_1 = \frac{1}{4\eta_1} \min_{i\in[I]} \frac{f_i(T - a_i)}{\sum_{r\in[R]} \lceil E_i N_i M_i(\tau_i + 2e_i/b_i)\rceil w_i^r} \tag{25}$$

$$L_2 = \frac{1}{4\eta_2} \min_{i\in[I]} \frac{f_i(\lceil T - a_i)}{\sum_{r\in[R]} \lceil E_i N_i M_i(\tau_i + 2e_i/b_i)\rceil s_i^r} \tag{26}$$

U_1^r (U_2^r) is the maximum per-unit-resource job utility for type-r resource on physical servers to deploy workers (parameter servers), among all jobs. Here, $f_i(\lceil E_i M_i(\tau_i + 2e_i/b_i)\rceil - a_i)$ is the largest utility that job i can achieve, by using the maximum number of workers (*i.e.*, N_i) at all times in E_i training epochs to achieve the shortest job completion time $\lceil \frac{E_i N_i M_i(\tau_i+2e_i/b_i)}{N_i} \rceil = \lceil E_i M_i(\tau_i + 2e_i/b_i)\rceil$. L_1 (L_2) represents the minimum unit-time-unit-resource job utility on physical servers to deploy workers (parameter servers), among all jobs. Here, $f_i(T - a_i)$ is the smallest utility that job i may achieve, when it ends at T. η_1 and η_2 are scaling factors satisfying $\frac{1}{\eta_1} \leq \frac{\lceil E_i N_i M_i(\tau_i+2e_i/b_i)\rceil \sum_{r\in[R]} w_i^r}{T\sum_{h\in[H]}\sum_{r\in[R]} c_h^r}, \forall i \in [I]$, and $\frac{1}{\eta_2} \leq \frac{\lceil E_i N_i M_i(\tau_i+2e_i/b_i)\rceil \sum_{r\in[R]} s_i^r}{T\sum_{k\in[K]}\sum_{r\in[R]} c_k^r}, \forall i \in [I]$, to ensure the initial value of dual objective is bounded.

The rationales behind our price functions are as follows. (i) The prices should be low enough at the beginning to accept many incoming jobs. When $g_h^r(t) = 0$, $v_k^r(t) = 0$, we have $p_h^r(t) = L_1, q_k^r(t) = L_2, \forall h \in [H], k \in [K], r \in [R]$, and then any job can be admitted at this point since L_1 and L_2 represent the lowest unit job utility (a formal proof is given in our technical report [32]). (ii) The prices increase exponentially when the allocated amounts of resources increase, to filter out jobs with low utilities which arrive early, and to reserve resources for jobs with higher utilities that may arrive later. (iii) The respective price should be high enough when a resource on a server is exhausted, such that no more jobs requiring this resource are admitted. When $g_h^r(t) = c_h^r$ or $v_k^r(t) = c_k^r$, we have $p_h^r(t) = U_1^r$ or $q_k^r(t) = U_2^r$, and no more jobs requiring these resources would be admitted since U_1^r and U_2^r are the largest unit job utilities (proof in [32]). The price functions are important to guarantee a good competitive ratio for our online algorithm.

U_1^r, U_2^r, L_1 and L_2 are required to compute price functions in Alg. 1, whose exact values are not known before all jobs have arrived. Instead, we adopt their estimated values (based on past experience) in our online algorithm, and will evaluate impact of inaccurate estimates in Sec. V.

C. Subroutine for Finding Best Job Schedule

The optimization problem in (21) to compute the best schedule l^* for job i is equivalent to the following:

$$\max_{\hat{t}_i,\mathbf{y},\mathbf{z}} f_i(\hat{t}_i - a_i) - \sum_{t\in[T]}\sum_{h\in[H]}\sum_{r\in[R]} p_h^r(t) w_i^r y_{ih}(t) - \sum_{t\in[T]}\sum_{k\in[K]}\sum_{r\in[R]} q_k^r(t) s_i^r z_{ik}(t) \quad (27)$$

$$s.t.\ g_h^r(t) + w_i^r y_{ih}(t) \leq c_h^r, \forall t \in [T], r \in [R], h \in [H]$$
$$v_k^r(t) + s_i^r z_{ik}(t) \leq c_k^r, \forall t \in [T], r \in [R], k \in [K]$$
Constraints (2)(3)(6)-(10)(12)(13), where $x_i = 1$

We next show that (27) can be efficiently and optimally solved using dynamic programming and a greedy algorithm. When we fix $\hat{t}_i$, (27) is simplified to the following ILP, where $\mathcal{T}_i = \hat{t}_i, \mathcal{D}_i = E_i N_i$:

$$\min_{\mathbf{y},\mathbf{z}} \quad \mathbf{cost}(\mathcal{T}_\mathbf{i}, \mathcal{D}_\mathbf{i}) = \sum_{t\in[a_i,\mathcal{T}_i]}\sum_{h\in[H]}\sum_{r\in[R]} p_h^r(t) w_i^r y_{ih}(t) + \sum_{t\in[a_i,\mathcal{T}_i]}\sum_{k\in[K]}\sum_{r\in[R]} q_k^r(t) s_i^r z_{ik}(t) \quad (28)$$

$$s.t. \sum_{t\in[a_i,\mathcal{T}_i]}\sum_{h\in[H]} y_{ih}(t) \geq \mathcal{D}_i M_i(\tau_i + 2e_i/b_i) \quad (29)$$

$$y_{ih}(t) \leq \min_{r\in[R]} \lfloor \frac{c_h^r - g_h^r(t)}{w_i^r} \rfloor, \forall h \in [H], t \in [a_i, \mathcal{T}_i] \quad (30)$$

$$z_{ik}(t) \leq \min_{r\in[R]} \lfloor \frac{c_k^r - v_k^r(t)}{s_i^r} \rfloor, \forall k \in [K], t \in [a_i, \mathcal{T}_i] \quad (31)$$

(3)(6)(7)(12)(13), where $t \in [a_i, \mathcal{T}_i]$

In problem (28), deployment decisions in different time slots are coupled only in constraint (29), which requires sufficient workers and parameter servers to be deployed such that all N_i data chunks are trained for E_i epochs during $[a_i, \mathcal{T}_i]$. We refer to $\mathcal{D}_i$ in the RHS of (29) as *training workload*, indicating the total count of data chunks trained (a data chunk is counted E_i times if trained for E_i times). Since the time for training a data chunk is much smaller than the duration of a time slot, we may safely assume a worker trains an integer number of data chunks in each time slot. The training workload is distributed over different time slots in $[a_i, \mathcal{T}_i]$. If we know how much training workload (denoted by $\mathcal{D}_i(t)$) is to be fulfilled in a time slot t, we are left with a further simplified problem:

$$\min \mathbf{cost\_t}(t, \mathcal{D}_i(t)) = \sum_{h\in[H]}\sum_{r\in[R]} p_h^r(t) w_i^r y_{ih}(t) + \sum_{k\in[K]}\sum_{r\in[R]} q_k^r(t) s_i^r z_{ik}(t) \quad (32)$$

$$s.t. \sum_{h\in[H]} y_{ih}(t) \geq \mathcal{D}_i(t) M_i(\tau_i + 2e_i/b_i)$$
(30)(31)(3)(6)(7)(12)(13), for the specific t

Though (32) is an ILP, it can be optimally solved using a greedy algorithm (to be discussed in Alg. 2 and analyzed in Theorem 1). Therefore, we come up with the following algorithm to find the best schedule for job i: enumerate end times $\hat{t}_i$ from a_i to T; given $\hat{t}_i$, design a dynamic programming approach to compute how to best distribute the training workload over time slots in $[a_i, \hat{t}_i]$; then use the greedy algorithm to decide deployment of workers and parameter servers in each time slot. Our algorithm is given in Alg. 2.

In Alg. 2, we enumerate job completion time slot $\hat{t}_i$ (line 2) and find the optimal schedule with each $\hat{t}_i$ by calling function **DP_COST** (line 3). Then we compare the payoffs achieved by schedules at different completion times and decide the best schedule achieving the highest payoff (lines 4-7).

Lines 10-20 implement a dynamic programming function: $cost(\hat{t}_i, E_i N_i) = \min_{d\in[0, E_i N_i]} cost\_t(\hat{t}_i, d) + cost(\hat{t}_i - 1, E_i N_i - d)$ We enumerate training workload d to be finished in time slot $\hat{t}_i$ from 0 to $E_i N_i$ (lines 12-13), and let the rest workload $E_i N_i - d$ be carried out in $[a_i, \hat{t}_i - 1]$ (line 14). We compare the resulting costs (value of objective function (28)) and identify the schedule achieving the smallest cost (lines 15-17). Finding the best schedule for workload $E_i N_i - d$ in $[a_i, \hat{t}_i - 1]$ is

Algorithm 2 Subroutine for Deriving Best Schedule of Job i

Input: T, $p_h^r(t), g_h^r(t), q_k^r(t), v_k^r(t), c_h^r, c_k^r, \forall h \in [H], k \in [K], r \in [R], t \in [T]$
Output: best schedule l^* and payoff μ_i for job i

1: Initialize $\mu_i = 0$, $l^* = \emptyset$, $y_{ih}(t) = 0, z_{ik}(t) = 0, \forall t \in [T], h \in [H], k \in [K]$
2: **for** $\hat{t}_i = a_i$ to T **do**
3: $(cost, l) = DP\_COST(\hat{t}_i, E_i N_i)$
4: $\mu_{il} = f_i(\hat{t}_i - a_i) - cost$
5: **if** $\mu_{il} > \mu_i$ **then**
6: $l^* \Leftarrow l$, $\mu_i = \mu_{il}$
7: **end if**
8: **end for**
9: return l^*, μ_i
10: **function** $\mathbf{DP\_COST}(\mathcal{T}_i, \mathcal{D}_i)$
11: $min\_cost = \infty$, $l = \emptyset$
12: **for** $d = 0$ to $\mathcal{D}_i$ **do**
13: $(cost\_t, \mathbf{y}(\mathcal{T}_i), \mathbf{z}(\mathcal{T}_i)) = COST\_t(\mathcal{T}_i, d)$
14: $(cost, l') = DP\_COST(\mathcal{T}_i - 1, \mathcal{D}_i - d)$
15: **if** $min\_cost > cost\_t + cost$ **then**
16: $min\_cost = cost\_t + cost$, $l \Leftarrow l' \cup \{\mathbf{y}(\mathcal{T}_i), \mathbf{z}(\mathcal{T}_i)\}$
17: **end if**
18: **end for**
19: Return min_cost, l
20: **end function**
21: **function** $\mathbf{COST\_t}(t, d)$
22: Initialize $y_{ih}(t) = 0, z_{ik}(t) = 0, \forall h \in [H], k \in [K]$
23: Sort servers in $[H]$ according to $\sum_{r\in[R]} p_h^r(t) w_i^r$ in non-decreasing order into $h_1, h_2, \ldots, h_H$
24: $D = \lceil dM_i(\tau_i + 2e_i/b_i) \rceil$;
25: **for** $j = 1, \ldots, H$ **do**/*deploy workers*/
26: $y_{ih_j}(t) = \min\Big\{ \min_{r\in[R]} \lfloor \frac{c_h^r - g_h^r(t)}{w_i^r} \rfloor,$
27: $N_i - \sum_{j'=1}^{j-1} y_{ih_{j'}}(t), D \Big\}$
28: $D = D - y_{ih_j}(t)$
29: **end for**
30: **if** $D > 0$ **then**/*not all workload can be handled*/
31: Return $cost\_t = +\infty, \mathbf{y}, \mathbf{z}$
32: **end if**
33: Sort servers in $[K]$ according to $\sum_{r\in[R]} q_k^r(t) s_i^r$ in non-decreasing order into $k_1, k_2, \ldots, k_K$
34: **for** $j = 1, \ldots, K$ **do**/*deploy parameter servers*/
35: $z_{ik_j}(t) = \min\Big\{ \min_{r\in[R]} \lfloor \frac{c_k^r - v_k^r(t)}{s_i^r} \rfloor,$
36: $\lceil \sum_{h\in[H]} y_{ih}(t) \frac{b_i}{B_i} \rceil - \sum_{j'=1}^{j-1} z_{ik_{j'}}(t),$
37: $\sum_{h\in[H]} y_{ih}(t) - \sum_{j'=1}^{j-1} z_{ik_{j'}}(t) \Big\}$
38: **end for**
39: **if** $\sum_{k\in[K]} z_{ik}(t) < \frac{b_i}{B_i} \sum_{h\in[H]} y_{ih}(t)$ **then**/*not enough parameter servers can be deployed*/
40: Return $cost\_t = +\infty, \mathbf{y}, \mathbf{z}$
41: **end if**
42: $cost\_t = \sum_{h\in[H]} \sum_{r\in[R]} p_h^r(t) w_i^r y_{ih}(t) + \sum_{k\in[K]} \sum_{r\in[R]} q_k^r(t) s_i^r z_{ik}(t)$
43: Return $cost\_t, \mathbf{y}(t), \mathbf{z}(t)$
44: **end function**

the same as finding the best schedule to carry out workload $E_i N_i$ in $[a_i, \hat{t}_i]$ except for at a smaller scale, and hence the function calls itself in line 14 (a.k.a. dynamic programming). Note that we always store the results of $COST\_t(t, d)$ and $DP\_COST(\mathcal{T}_i, \mathcal{D}_i)$ computed at different $\hat{t}_i$'s, to avoid re-computing the same subproblem in later iterations.

COST_t in lines 21-44 computes the optimal worker and parameter server deployment to fulfil workload d in time slot t. We sort servers for worker deployment in non-decreasing order of overall resource price $\sum_{r\in[R]} p_h^r(t) w_i^r$ (line 23), and maximally deploy workers starting from the cheapest server, respecting capacity constraint (30) and upper bound N_i on the number of workers in (3), to fulfil workload d (lines 24-29). Parameter servers are deployed in a similar greedy fashion. The total number of parameter servers guarantees sufficient bandwidth to serve workers (constraint (6)) but not over-provisioning (constraint (7)), subject to capacity constraint (31) (lines 34-38). If not enough workers or parameter servers can be deployed, fulfilling workload d in t is infeasible (lines 30-32, 39-41); otherwise, we return total deployment cost in t (value of objective function (32)) and the schedule.

D. Theoretical Analysis

We next analyze our online algorithm in terms of correctness, time complexity, and competitive ratio. All missing proofs can be found in our technical report [32].

Theorem 1 (Optimality of Subroutine)**.** *Alg. 2 produces an optimal solution of problem (27), in which* **COST_t** *computes an optimal solution of problem (32).*

Theorem 2 (Correctness)**.** OASiS *in Alg. 1 (together with Alg. 2) computes a feasible solution to problems (1) (14) (19).*

Though our online algorithm involves a dynamic programming approach, we prove its polynomial time complexity as follows.

Theorem 3 (Polynomial Running Time)**.** OASiS *in Alg. 1 (together with Alg. 2) runs in polynomial time to decide job admission and schedule upon arrival of each job i, with time complexity $O(TN_iE_i(H + K) + TN_i^2E_i^2)$.*

The competitive ratio of our online algorithm is the worst-case upper bound of the ratio of the overall utility of admitted jobs derived by the offline optimal solution of (1) to the total utility of admitted jobs achieved by Alg. 1 in the overall system span.

Theorem 4 (Competitive Ratio)**.** OASiS *in Alg. 1 is 2α-competitive, where $\alpha = \max_{r\in[R]}(1, \ln \frac{U_1^r}{L_1}, \ln \frac{U_2^r}{L_2})$ and U_1^r, U_2^r, L_1 and L_2 are defined in (23)-(26).*

Theorem 4 tells that the larger the ratio of the largest utility to the lowest utility that the jobs can achieve is, the worse the ratio is. In this case, if *OASiS* makes a wrong decision, the gap from the offline optimum is larger. If the timespan T or the total amount of resources is larger, the ratio is also worse, as there is more room for the offline algorithm to improve.

V. Performance Evaluation

We next evaluate *OASiS* by simulation studies and testbed experiments based on a prototype system implementation.

A. Simulation Studies

Settings. We simulate an ML system running for $T = 100$-300 time slots, with $H = 50$ servers to host workers (server resource capacities set according to Amazon EC2 C4 instances) and $K = 50$ servers to deploy parameter servers (resource

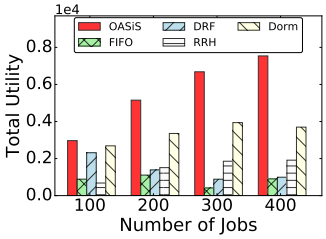

Fig. 3: Total job utility

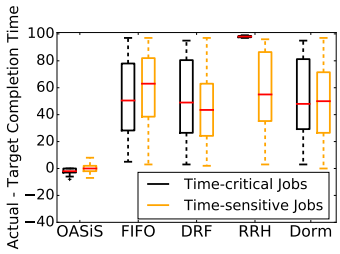

Fig. 4: Completion timeliness

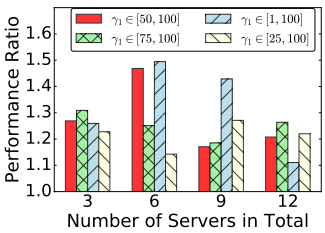

Fig. 5: Performance ratio

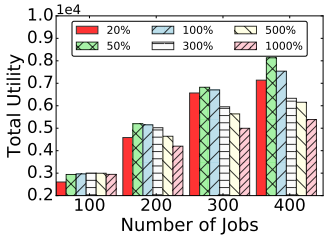

Fig. 6: Total job utility under inaccurate $\frac{U_1^r}{L_1}$, $\frac{U_2^r}{L_2}$

capacities following EC2 GPU instances P2 and G3 randomly [28]). Server bandwidth is set within $[20, 50]$Gbps. Following similar settings in [17][7][8], we set resource demands of each worker as follows: 0 to 4 GPUs, 1 to 10 vCPUs, 2 to 32GB memory, 5 to 10GB storage, and bandwidth of 100Mbps to 5Gbps (b_i). Resource demands of each parameter server are: 1 to 10 vCPUs, 2 to 32GB memory, 5 to 10GB storage, and bandwidth of 5Gbps to 20Gbps (B_i). We set job arrival pattern according to the Google cluster data [33], but may vary job arrival rates. For different jobs, E_i is set within $[50, 200]$, N_i is in $[5, 100]$, M_i is in $[10, 100]$, τ_i is in $[0.001, 0.1]$ time slots, and e_i is within $[30, 575]$MB [5]. We use a sigmoid utility function [20], $f_i(t - a_i) = \frac{\gamma_1}{1+e^{\gamma_2(t-a_i-\gamma_3)}}$, where γ_1 is priority of job i in $[1, 100]$, γ_2 is a decay factor, and γ_3 is the target completion time of job i set in $[1, 15]$. We set $\gamma_2 = 0$ for time-insensitive jobs (constant utility), γ_2 in $[0.01, 1]$ to represent time-sensitive jobs and γ_2 in $[4, 6]$ for time-critical jobs. By default, 10%, 55% and 35% jobs are time-insensitive, -sensitive, and -critical, respectively, in our experiments.

Schemes for comparison. We compare *OASiS* with four representative job scheduling policies in existing cloud platforms. (i) FIFO: default scheduler in Hadoop and Spark [13]; jobs are admitted and run in order of their arrivals, with fixed numbers of workers/parameter servers. (ii) Dominant Resource Fairness Scheduling (DRF): default scheduler in YARN [12] and Mesos [15]; jobs are all admitted and numbers of workers/parameter servers are computed to achieve max-min fairness in dominant resources upon job arrival and job completion [14]. (iii) Risk-Reward Heuristic (RRH) [34]: a job is admitted if its utility minus a delay cost incurred by its admission is larger than a threshold; upon job arrival or completion, unfinished jobs either continue running (always with same worker/parameter server numbers once running) or pause, decided by job's future utility gain minus cost. (iv) Dorm [17]: Jobs are admitted; upon job arrival or completion, numbers and placement of workers/parameter servers of unfinished jobs are recomputed by an MILP resource utilization maximization problem, subject to fairness and adjustment overhead constraints. In (i)-(iii), we place workers and parameter servers on available servers in a round-robin fashion. For FIFO and RRH, the number of workers (parameter servers) is fixed to a number within $[1, 30]$.

Results. Fig. 3 presents the total utility achieved by different schemes, where $T = 300$. *OASiS* performs the best, especially when the number of jobs in the fixed timespan is larger (resources are more scarce).

Fig. 4 shows how well the target completion time is met when 100 time-sensitive and 100 time-critical jobs are run in $T = 100$. The actual completion time minus target completion time (γ_3 in the sigmoid utility function) achieved with *OASiS* is the closest to zero for both types of jobs, with the smallest variance. Among the other schemes, only RRH is job utility (completion time) aware, but its resource utilization is not as efficient so does not perform well either.

Fig. 5 shows the performance ratio of *OASiS*, computed by dividing the total job utility of the offline optimal solution by the total job utility achieved by *OASiS*. Due to the time complexity of solving (1) exactly for the offline optimum, the number of jobs is limited to 10.[3] We set $T = 10$, vary the number of servers (proportionally divided to host workers and parameter servers), and also vary the range of job priorities (γ_1 in the sigmoid function), such that $\max_{r\in[R]}(\frac{U_1^r}{L_1}, \frac{U_2^r}{L_2})$ increases from left to right at each fixed number of servers in the figure. We observe a ratio around 1.1 to 1.5, showing the good performance of our online algorithm. There is no clear trend of increase or decrease of the ratio with more resources and larger $\max_{r\in[R]}(\frac{U_1^r}{L_1}, \frac{U_2^r}{L_2})$ – the factors influencing the worst-case competitive ratio in Theorem 4 (note our simulation scenario may not be the worst case).

In Fig. 6, we use estimated values of $\frac{U_1^r}{L_1}$ and $\frac{U_2^r}{L_2}$ as input to *OASiS*, at different percentages of their actual values ($T = 300$). We observe that an underestimation leads to higher total utility than overestimation when resources are scarce, as it prevents abrupt price rise which may filter out jobs that should be accepted. These results directly reflect impact of using inaccurate estimations of $\frac{U_1^r}{L_1}$ and $\frac{U_2^r}{L_2}$ on performance ratio of *OASiS*.

B. Testbed Experiments

Prototype implementation. We implement a distributed ML system based on MXNet [6] with Kubernetes 1.6 [16]. MXNet is modified to support dynamic adjustment of worker/parameter server numbers. *OASiS* and other 4 scheduling schemes for comparison are implemented as custom schedulers to replace the default one in Kubernetes, respectively. The scheduler constantly queries ongoing jobs and available system resources, and posts scheduling decisions via the

[3] It takes 2 days to compute the optimal offline solution with 10 jobs, while *OASiS* runs for less than 1 second to produce the best schedule for each job in the case of 100 time slots and 80 worker/parameter servers.

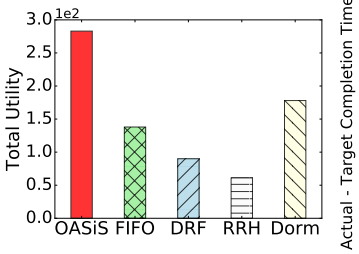

Fig. 7: Total job utility

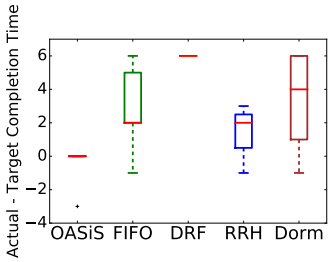

Fig. 8: Completion timeliness

Kubernetes API server. Each worker or parameter server is implemented on a Docker container with 0 to 1 GPU, 1 to 5 CPU cores, 2 to 10GB memory, and 1 to 3Gbps bandwidth. We deploy our system on 9 servers: 2 with 8 CPU cores, 32GB RAM, 1.5TB storage each host parameter servers, and 7 with 32 CPU cores, 80GB RAM, 600GB storage each host workers (there are 4 GPUs deployed on 4 servers). All servers are equipped with a dual-port 1GbE NIC and a dual-port 10GbE NIC. All data are stored in HDFS [29], with chunk size 2MB.

Experimental setup. We run 6 kinds of model training jobs, *i.e.*, AlexNet [35], ResNet-50,101,152 [9], VGG-11 [10], and Inception-BN [36], on ImageNet ILSVRC2012 [4] dataset (we use 200 images (20.3MB)). Each experiment runs for 10 time slots and each time slot is 20 minutes long. 12 jobs arrive in the first 9 time slots and each job runs for 40 minutes to 2 hours. Each data chunk contains 20 or 30 images, each mini-batch contains 10 images, and the number of epochs is in $[4, 30]$. Job utilities are similar to simulation.

Experimental results. We plot the total utility in Fig. 7 and the actual completion time minus target completion time of all jobs in Fig. 8. Compared to Fig. 3 and Fig. 4, the comparison results are similar. With the small number of jobs that we can run on our small testbed, the difference between *OASiS* and others may not be as apparent as that in a large system (as shown by our larger scale simulations). We are confident that the advantage of our algorithm will be more obvious when experimenting on a large testbed.

VI. Conclusion

This paper proposes *OASiS*, an online algorithm for admission and scheduling of asynchronous training jobs in an ML cluster. *OASiS* computes the best schedule to run each job, using a varying number of workers and parameter servers over time for best resource utilization and training expedition, while admitting jobs judiciously based on carefully set resource prices, for long-term utility maximization. Our theoretical analysis shows polynomial running time and a good competitive ratio of *OASiS*. Simulation and experiments on a prototype system show that *OASiS* outperforms common schedulers in real-world cloud systems.

References

[1] M. Abadi, P. Barham *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *Proc. of USENIX OSDI*, 2016.

[2] "Microsoft Cognitive Toolkit," https://www.microsoft.com/en-us/cognitive-toolkit/.

[3] "PaddlePaddle," https://github.com/PaddlePaddle/Paddle.

[4] J. Deng, W. Dong, R. Socher, L. Li, K. Li *et al.*, "ImageNet: A Large-Scale Hierarchical Image Database," in *Proc. of IEEE CVPR*, 2009.

[5] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "FireCaffe: Near-Linear Acceleration of Deep Neural Network Training on Compute Clusters," in *Proc. of IEEE CVPR*, 2016.

[6] T. Chen, M. Li *et al.*, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," in *NIPS Workshop on Machine Learning Systems (LearningSys)*, 2016.

[7] M. Li, D. G. Andersen *et al.*, "Scaling Distributed Machine Learning with the Parameter Server," in *Proc. of USENIX OSDI*, 2014.

[8] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an Efficient and Scalable Deep Learning Training System," in *Proc. of USENIX OSDI*, 2014.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. of IEEE CVPR*, 2016.

[10] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *Proc. of ICLR*, 2015.

[11] A. Verma, L. Pedrosa, M. Korupolu *et al.*, "Large-Scale Cluster Management at Google with Borg," in *Proc. of ACM EuroSys*, 2015.

[12] V. K. Vavilapalli, A. C. Murthy *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proc. of ACM SoCC*, 2013.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin *et al.*, "Spark: Cluster Computing with Working Sets," in *Proc. of USENIX HotCloud*, 2010.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *Proc. of USENIX NSDI*, 2011.

[15] B. Hindman, A. Konwinski *et al.*, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Proc. of USENIX NSDI*, 2011.

[16] "Kubernetes," https://kubernetes.io/.

[17] P. Sun, Y. Wen, N. B. D. Ta, and S. Yan, "Towards Distributed Machine Learning in Shared Clusters: A Dynamically-Partitioned Approach," in *Proc. of IEEE Smart Computing*, 2017.

[18] Y. S. L. Lee *et al.*, "Dolphin: Runtime Optimization for Distributed Machine Learning," in *Proc. of ICML ML Systems Workshop*, 2016.

[19] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems," in *Proc. of ACM SIGKDD*, 2015.

[20] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. Tsang, "Need for Speed: CORA Scheduler for Optimizing Completion-Times in the Cloud," in *Proc. of IEEE INFOCOM*, 2015.

[21] L. Chen, S. Liu *et al.*, "Scheduling Jobs across Geo-Distributed Datacenters with Max-Min Fairness," in *Proc. of IEEE INFOCOM*, 2017.

[22] Y. Azar, I. Kalp-Shaltiel, B. Lucier, I. Menache *et al.*, "Truthful Online Scheduling with Commitments," in *Proc. of ACM EC*, 2015.

[23] B. Lucier, I. Menache, J. S. Naor, and J. Yaniv, "Efficient Online Scheduling for Deadline-Sensitive Jobs," in *Proc. of ACM SPAA*, 2013.

[24] R. Zhou, Z. Li, C. Wu, and Z. Huang, "An Efficient Cloud Market Mechanism for Computing Jobs With Soft Deadlines," *IEEE/ACM Transactions on Networking*, 2017.

[25] X. Zhang, Z. Huang, C. Wu, Z. Li, and F. C. Lau, "Online Auctions in IaaS Clouds: Welfare and Profit Maximization with Server Costs," in *Proc. of ACM SIGMETRICS*, 2015.

[26] Z. Xiao *et al.*, "Automatic Scaling of Internet Applications for Cloud Computing Services," *IEEE Transactions on Computers*, 2014.

[27] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch *et al.*, "TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters," in *Proc. of ACM EuroSys*, 2016.

[28] "Amazon EC2 Instances," https://aws.amazon.com/ec2/instance-types/.

[29] "Apache Hadoop," http://hadoop.apache.org/.

[30] H. B. McMahan, G. Holt, D. Sculley, M. Young *et al.*, "Ad Click Prediction: a View from the Trenches," in *Proc. of ACM SIGKDD*, 2013.

[31] N. Buchbinder, J. S. Naor *et al.*, "The Design of Competitive Online Algorithms via a Primal–Dual Approach," *Foundations and Trends® in Theoretical Computer Science*, 2009.

[32] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online Job Scheduling in Distributed Machine Learning Clusters," *arXiv preprint arXiv:1801.00936*, 2017.

[33] C. Reiss, A. Tumanov *et al.*, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proc. of ACM SoCC*, 2012.

[34] D. E. Irwin, L. E. Grit, and J. S. Chase, "Balancing Risk and Reward in a Market-based Task Service," in *Proc. of IEEE HPDC*, 2004.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *NIPS*, 2012.

[36] S. Ioffe *et al.*, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proc. of ICML*, 2015.