# A Sum-of-Ratios Multi-Dimensional-Knapsack Decomposition for DNN Resource Scheduling

Menglu Yu[1] Chuan Wu[2] Bo Ji[3] Jia Liu[4]
[1]Department of Computer Science, Iowa State University
[2]Department of Computer Science, The University of Hong Kong
[3]Department of Computer Science, Virginia Tech
[4]Department of Electrical and Computer Engineering, The Ohio State University

***Abstract*—In recent years, to sustain the resource-intensive computational needs for training deep neural networks (DNNs), it is widely accepted that exploiting the parallelism in large-scale computing clusters is critical for the efficient deployments of DNN training jobs. However, existing resource schedulers for traditional computing clusters are not well suited for DNN training, which results in unsatisfactory job completion time performance. The limitations of these resource scheduling schemes motivate us to propose a new computing cluster resource scheduling framework that is able to leverage the special layered structure of DNN jobs and significantly improve their job completion times. Our contributions in this paper are three-fold: i) We develop a new resource scheduling analytical model by considering DNN's layered structure, which enables us to analytically formulate the resource scheduling optimization problem for DNN training in computing clusters; ii) Based on the proposed performance analytical model, we then develop an efficient resource scheduling algorithm based on the widely adopted parameter-server architecture using a sum-of-ratios multi-dimensional-knapsack decomposition (SMD) method to offer strong performance guarantee; iii) We conduct extensive numerical experiments to demonstrate the effectiveness of the proposed schedule algorithm and its superior performance over the state of the art.**

## I. Introduction

In recent years, deep-learning-based applications are quickly finding their ways into our everyday life, including healthcare, automobile, retail, smart homes, just to name a few. However, these applications also generate and inject a large volume of resource-intensive computing jobs for training deep neural networks (DNNs), which are used in various systems for computer vision, natural language processing, online recommendation, etc. In order to sustain such a rapidly growing need for DNN training in recent years, it is widely accepted that a viable solution is to exploit the vast *parallelism* in distributed computing architectures to schedule deep learning jobs. To date, however, most traditional resource scheduling schemes for computing clusters are not designed for DNN training (e.g., Google's Borg System [1] employs static resource allocation specified by the users upon job submissions). Also, most of the recently proposed scheduling schemes designed for DNN jobs (e.g., Gandiva [2] and Tiresias [3]) are heuristic approaches, which provide no performance guarantee. In light of the increasing importance of DNN-based applications, there is a pressing need for developing provably efficient resource schedulers tailored for DNN training in computing clusters.

However, developing such resource scheduling algorithms for DNN training clusters is highly non-trivial. In a computing cluster, DNN training jobs are submitted over time with various competing resource requirements (numbers of CPUs and GPUs, size of memory, etc.), and the training process is both resource-intensive and time-consuming. For example, researchers showed that it could take 115 minutes to train a model with ResNet50 dataset [4] on a DGX-1 machine with 8 V100 GPUs [5], and even 3–5 days to train the DeepSpeech2 model [6] on the LibriSpeech dataset [7] using 16 GPUs [6]. Also, to date, there is a lack of a tractable and accurate analytical model that takes different mechanisms of communication-computation overlapping into consideration based on the layered structure of DNN.[1] Furthermore, similar to the design of most scheduling algorithms for large-scale distributed computing clusters, the computing resource limitation for DNN computing jobs naturally leads to integer bin-packing-like constraints in the scheduling problem, which makes the problem NP-Hard. Also, the objective function of the DNN resource scheduling problem has a sum-of-ratios structure due to the computational speed characterization in DNN training. As a result, the scheduling problem is *non-convex* even with continuous relaxation, which introduces yet another layer of difficulty to the already-challenging problem.

In this paper, we overcome the above challenges and develop a suite of scheduling algorithmic techniques for efficient DNN training in computing clusters Our main results and technical contributions are summarized as follows:

- We develop a new performance analysis framework for scheduling DNN training jobs. Specifically, we first propose a *unified* analytical model to characterize the DNN training, which captures a variety of ways to overlap communication and computation by exploiting the layered structure of DNNs.We then formulate the job admission and resource

This work has been supported in part by NSF grants CAREER CNS-1943226, CCF-1758736, ONR grant N00014-17-1-2417, a Google Faculty Research Award, NSF CNS-1651947, and Hong Kong RGC grants HKU 17204619, 17208920.

[1]To speed up the training process, the idea of exploiting the special layered structure of DNNs to overlap communication and computation has been explored recently in [8]–[10], which showed that the training throughput of the MXNet framework could be improved by 25%–70%.

scheduling problem for DNN training by associating the above unified analytical model with both synchronous and asynchronous stochastic gradient descent (SGD) algorithms. Each job is associated with a utility function, which is non-increasing with respect to its job completion time. The objective of the analytical model is to maximize the overall utility (i.e., minimize the overall training completion time).

- Based on the above analytical framework, we formulate the resource scheduling problem as a mixed-integer nonlinear programming problem (MINLP), and prove its NP-hardness. To overcome this fundamental hardness, we propose a divide-and-conquer approach called SMD (sum-of-ratio multi-dimensional-knapsack decomposition). Specifically, based on a keen observation of the physical reality of resource requests in most distributed cloud computing systems in practice (e.g., Amazon's EC2), we show that our resource scheduling problem has a *decomposition* structure. Under this decomposition, the inner subproblem is a mixed-integer sum-of-ratios problem with packing constraints. Thanks to the lower dimensionality of the inner subproblem, we are able to develop an efficient $\epsilon$-approximation algorithm based on grid-searching coupled with randomized rounding for solving this subproblem.
- Upon solving the inner subproblem, we show that the outer subproblem reduces to a multi-dimensional knapsack problem (MKP), which also admits an efficient $\epsilon$-approximation algorithm. By combining both steps, we establish the overall approximation ratio of the proposed SMD approach. To verify the efficacy of our proposed algorithm, we conduct numerical experiments based on Google cluster traces [11]. Our results show that the proposed SMD approach significantly outperforms the equal server-worker allocation scheme (widely used in practice) and a state-of-the-art approach called Optimus [12].

The remainder of this paper is organized as follows. In Section II, we review the literature to put our work in a comparative perspective. In Section III, we introduce the system model and problem formulation. Section IV presents our algorithms and their performance analysis. Section V shows numerical results, and Section VI concludes this paper.

## II. Related Work

Due to the rise of machine learning (ML) applications and their high computational workload, optimizing resource scheduling to facilitate distributed ML frameworks have attracted a great amount of interest in recent years (see, e.g., [13]–[16] and the references therein). DNN training jobs have unique characteristics (e.g., iterativeness and layered structure), which could be leveraged to overlap computation and communication time between iterations to reduce the training time. However, these existing works were designed for resource allocation to support *general* ML jobs in computing clusters, which may not be tailored for DNN training jobs. As a result, when being applied in DNN training, their performance is suboptimal in general since they do not leverage the aforementioned characteristics of DNN training.

To date, research on computing cluster scheduling optimization tailored for DNN training remains relatively new with limited results. Most of the early attempts in this area (e.g., [17] and references therein) only considered static allocation of workers and parameter servers (PS). To our knowledge, Yan *et al.* [18] was the first to investigate the performance of distributed ML frameworks, and developed a DNN performance model at a layer-level granularity (e.g., the model considers the computation time of each operator on a specific CPU and the NN structures). Subsequently, Gandiva [2] exploited intra-job predictability to time-slice GPUs efficiently across multiple jobs to better-fit GPUs. Tiresias [3] aimed to reduce the job completion times when the jobs' execution times are unpredictable due to non-smooth loss curves during a trial-and-error exploration. The most recent work [19] focused on making machine learning workloads complete in a finish-time fair manner. However, these schedulers are based on heuristic approaches. Also, the training completion time of a job is significantly affected by resource allocation. Studies in [20] showed that increasing resources did not contribute to a linear increase of the training speed and could even slow down the training process. Since static numbers of workers and PSs specified by users are suboptimal in general, researchers have also started to consider dynamic scheduling algorithms to determine optimal numbers of workers and PSs to optimize the training speed. To our knowledge, the first dynamic scheduling algorithm with performance guarantee was reported by Bao *et al.* [21], where they designed an online scheduling algorithm for deep learning jobs. However, their studies relied on strong assumptions and simplified modeling of deep learning jobs.

The most related work to ours is Optimus [20], where Peng *et al.* developed a heuristic resource allocation algorithm for the distributed deep learning jobs. Our work differs from [20] in the following key aspects: 1) In [20], the authors built the performance model without taking the DNN layered structure into consideration. As will be shown later, this yields suboptimal scheduling decisions in general. By contrast, we develop an analytical model that considers the layered characteristics of DNN training, which captures communication-computation overlapping in state-of-the-art DNN training systems [9]; and 2) Optimus proposed a dynamic resource scheduler, which is only a heuristic with no performance guarantee based on their online-fitted resource-performance models. In comparison, we propose a resource scheduler that leverages an analytical model to offer strong performance guarantees.

## III. System Model and Problem Formulation

In this section, we first present our general problem formulation for DNN training in distributed computing clusters in Section III-A. We then specialize the problem formulation under both synchronous and asynchronous SGD with the basic sequential training model [22] in Section III-B. Lastly, in Section III-C, we further generalize and refine our analytical model to include two more advanced DNN training models.

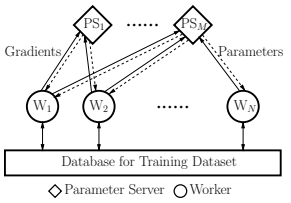

Fig. 1: PS-based architecture.

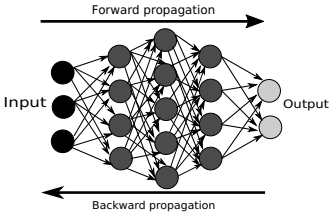

Fig. 2: DNN training.

### A. General Problem Formulation

In this paper, we focus on data parallelism in data centers with the PS architecture [23], [24][2]. As shown in Fig. 1, in each iteration, each worker fetches a mini-batch of samples from its local dataset and computes a stochastic gradient. Upon finishing computation, each worker sends the gradient to the PSs to update the model parameters following an optimization algorithm, e.g., the stochastic gradient descent (SGD) method. The workers then will pull the updated parameters from the PSs, fetch the next mini-batch of samples and proceed to the next iteration. This process repeats until some convergence criterion of the optimization algorithm is met.

The gradient computation at each worker is based on the specific DNN as illustrated in Fig. 2. In each iteration, one mini-batch of samples is used by a worker to compute the loss from the initial layer to the last layer, which is referred to as *forward propagation* (FP). After FP, a stochastic gradient of the DNN model parameters will be computed in the reverse order of layers in a *backward propagation* (BP) fashion. The computed gradients are used for updates at the PSs.

We consider the setting where DNN training jobs are submitted by users to the computing cluster to be processed periodically. These submitted jobs are trained using workers and PSs implemented via virtual machines or containers, which share resources in the underlying servers. Upon submission, the user will specify the resource needed for the job, based on which our algorithm will determine the numbers of workers and PSs if the job is admitted in the current scheduling interval. The jobs that are submitted during the scheduling interval are called *active jobs*. We assume that the scheduling interval is sufficiently large (e.g., hours) such that all the scheduled active jobs can be completed within a scheduling interval.

We use $E[i]$ to denote the total number of iterations to train for job $i$, which is specified by the user.[3] We use $f(p[i], w[i])$ to denote the current training speed of job $i$ (i.e., the number of iterations completed per unit time), which is a function of the number of PSs ($p[i]$) and the number of workers ($w[i]$). Then, the completion time of job $i$ can be estimated as $\frac{E[i]}{f(p[i],w[i])}$. Let $\mathcal{I}$ denote the set of active jobs submitted in the scheduling interval with $|\mathcal{I}| = I$, and $\mathcal{R}$ denote the set of computing resources (e.g., CPU, GPU and memory). Let $O^r[i]$ and $G^r[i]$ be the amount of type-$r$ resources demanded by a worker and a PS for job $i$, respectively. Let $v^r[i]$ be the resource limit specified by the user for job $i$, i.e., the maximum amount of resource $r$ the user may need. Let $C^r$ denote the capacity of type-$r$ resource in the computing cluster. Due to resource limits in the cluster, the system may not be able to process all jobs in $\mathcal{I}$ in the current scheduling interval. Thus, we use a binary variable $x_i$ to indicate whether job $i$ is admitted in this interval ($x_i = 1$) or not ($x_i = 0$). Let $\mu_i(\cdot) \geq 0$ be the utility function associated with job $i$, which is *non-increasing* with respect to the completion time $\frac{E[i]}{f(p[i],w[i])}$. In this paper, our goal is to optimize the admission decision $\mathbf{x} \triangleq \{x_i, i \in \mathcal{I}\}$ and resource allocation decisions $\mathbf{p} \triangleq \{p[i], i \in \mathcal{I}\}$ and $\mathbf{w} \triangleq \{w[i], i \in \mathcal{I}\}$ to maximize the overall utility for the submitted jobs in each scheduling interval. This optimization problem can be formulated as:

$$\underset{\mathbf{w},\mathbf{p},\mathbf{x}}{\text{Maximize}} \quad \sum_{i\in\mathcal{I}} \mu_i\left(\frac{E[i]}{f(p[i],w[i])}\right)x_i \tag{1}$$

$$\text{subject to} \quad \sum_{i\in\mathcal{I}} v^r[i]x_i \leq C^r, \quad \forall r \in \mathcal{R}, \tag{2}$$

$$(O^r[i]w[i]+G^r[i]p[i])x_i \leq v^r[i], \forall i \in \mathcal{I}, r \in \mathcal{R}, \tag{3}$$

$$p[i] \in \mathbb{Z}^{++}, w[i] \in \mathbb{Z}^{++}, x_i \in \{0,1\}, \quad \forall i \in \mathcal{I}.$$

Constraint (2) ensures that the sum of maximum resource demands from admitted active jobs does not exceed the cluster's resource capacity. Constraint (3) ensures that the allocated resources to run workers and PSs in each job $i$ do not exceed the resource limits specified by the job owner.

### B. Training Speed Modeling

In the general problem formulation above, the training speed function $f(p[i], w[i])$ remains to be defined. Next, we will first derive the training speed $f(p[i], w[i])$ per iteration per worker based on the basic sequential computation-communication model [22], which will serve as a baseline for two more advanced computation-communication models in Sec. III-C.

**Notation:** We let $N[i]$ denote the number of layers in the DNN model of job $i$. We use $r_j[i]$ to denote the time spent for sending gradients to or receiving parameters from the PSs for layer $j$ of job $i$ (assuming equal time for pushing gradients and pulling updated parameters). Let $b_j[i]$ and $f_j[i]$ be the BP and FP computation times for layer $j$ of job $i$, respectively. We use $\kappa_j[i]$ to denote the start time of sending gradients for layer $j$ of job $i$, and we let $s_j[i]$ be the start time of receiving parameters for layer $j$ of job $i$. Let $\tau_j[i]$ be the start time of FP for layer $j$ of job $i$. For lighter notation, we will omit the job index "$[i]$" in the subsequent training speed modeling if there is only one training job involved in the context (e.g., $r_j := r_j[i]$. We will revive "$[i]$" if confusion may arise). Key notation is summarized in Table I for ease of reference.

The sequential model [22] is illustrated in Fig. 3, where push/pull of gradients/parameters start after BP of all layers are done, the push/pull of layers are done sequentially from the highest layer to the first layer, and FP of the next iteration starts after all push/pull are done. For an $N$-layer DNN model, it is easy to see that the per-sample training time $t$ can be

[2]Due to the homogeneous processing speed in data centers, we assume that the straggling issue is minor in this paper.

[3]In practice, to prevent spending excessively long time waiting for the training process of a DNN job to converge, a maximum number of training iterations is usually set by the user.

TABLE I: Notation.

| | |
|---|---|
| $\mathcal{I}$ | The set of active jobs in the scheduling interval |
| $\mathcal{P}[i]/\mathcal{R}$ | The set of PSs of job $i$ / The set of resource types |
| $t[i]$ | Time of training one sample on a worker of job $i$ |
| $t_m[i]$ | Time of one training step on a worker of job $i$ |
| $r_j[i]$ | # of unit time for sending/receiving gradients to and parameters from the PSs at layer $j$ of job $i$ |
| $q_{ij}$ | # of unit time for parameter update at layer $j$ of job $i$ |
| $b_j[i]$ | # of unit time for BP at layer $j$ of job $i$ |
| $f_j[i]$ | # of unit time for FP at layer $j$ of job $j$ |
| $\kappa_j[i]$ | Start time of sending gradients of job $i$ at layer $j$ |
| $e_j[i]$ | End time of communication of job $i$ at layer $j$ in the priority based model |
| $s_j[i]$ | Start time of receiving parameters of job $i$ at layer $j$ |
| $\tau_j[i]$ | Start time of FP of job $i$ at layer $j$ |
| $N[i]$ | # of layers in DNN model of job $i$ |
| $E[i]/g[i]$ | # of training iterations for job $i$ / Model size of job $i$ |
| $K[i]/m[i]$ | Global batch size of job $i$ / One mini-batch size of job $i$ |
| $C^r$ | Capacity of type-$r$ resource in the DL cluster |
| $O^r[i]$ | Type-$r$ resource required by a worker in job $i$ |
| $G^r[i]$ | Type-$r$ resource required by a PS in job $i$ |
| $v^r[i]$ | The resource limit specified by the user for job $i$ |
| $w[i]/p[i]$ | # of workers of job $i$ / # of PSs of job $i$ |
| $B[i]$ | Bandwidth capacity of each PS of job $i$ |
| $t_f[i]$ | Average time of processing a sample in the FP of job $i$ |
| $t_b[i]$ | Average time of processing a sample in the BP of job $i$ |
| $t_r[i]$ | Average time of processing a sample in the communication time of job $i$ |

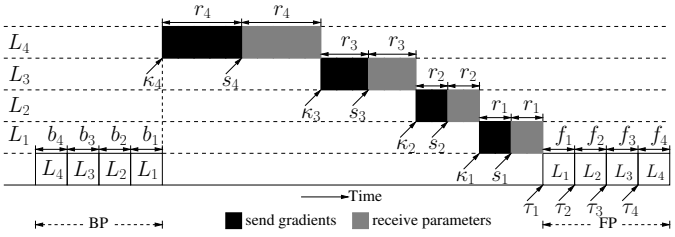


Fig. 3: Sequential-based model.

computed as $\sum_{j=1}^{N} b_j + 2\sum_{j=1}^{N} r_j + \sum_{j=1}^{N} f_j$. We let $t_b \triangleq \sum_{j=1}^{N} b_j$, $t_r \triangleq 2\sum_{j=1}^{N} r_j$, and $t_f \triangleq \sum_{j=1}^{N} f_j$ denote the BP time, the communication time, and the FP time for processing a sample of job $i$, respectively. Note that the key feature of the sequential model is that computation phases (i.e., BP and FP) and communication phases are conducted in a *sequential* fashion, which underutilizes the channel.

Consider a job with $p$ PSs and $w$ workers. We use $B$ to denote the bandwidth between each pair of PS and worker. Let $g$ be the model size (i.e., the number of elements in its gradient vector). Let $m$ be the mini-batch size. Then the FP time for processing a mini-batch can be calculated as $m \cdot t_f$. The BP time for processing a mini-batch does not depend on $m$ and can be computed as $t_b$. We use $w'_\rho$ to denote the average number of workers that send the computed gradients simultaneously to a PS $\rho \in \mathcal{P}$, where $\mathcal{P}$ denotes the set of PSs. Then, the bandwidth occupied by each worker is $\frac{B}{w'_\rho}$.[4] We use $g_j$ to denote the gradient size of layer $j$ and let $p_j$ be the corresponding number of PSs that layer $j$ will send its gradients to and receive its parameters from. For DNN models in practice, the number of neurons of a layer is usually much larger than the number of PSs, so it holds that $g_j \gg p_j$. We also assume that $g_j$ is evenly divided into all PSs (implying $p_j = p$).[5] Then, the time for sending gradients/receiving parameters under the symmetric assumption (equal communication speed for uplink and downlink) for each layer $j$ can be computed as: $\frac{g_j/p_j}{B/w'_\rho} = \frac{g_j/p}{B/w'_\rho}$. It then follows that the data communication time can be computed as $2\sum_{j=1}^{N} \frac{g_j/p}{B/w'_\rho} = 2\frac{\sum_{j=1}^{N} g_j/p}{B/w'_\rho} = 2\frac{g/p}{B/w'_\rho}$.

In addition to the communication and computation time, there exists extra communication overhead (e.g., establishing TCP connections) that increases linearly with the number of workers and PSs, which can be computed as $\beta_1 w + \beta_2 p$, where $\beta_1$ and $\beta_2$ are constants that depend on the underlying system [20]. Thus, the per-iteration-per-mini-batch training time can be computed as $t_m = \max_{\rho\in\mathcal{P}} \left[ mt_f + t_b + 2\frac{g/p}{B/w'_\rho} + \beta_1 w + \beta_2 p \right]$. Next, we derive the training speed function (i.e., how many iterations can be completed per unit time). We consider both synchronous and asynchronous trainings.

*1) Synchronous Training:* In synchronous training, PSs perform an update only after receiving gradients from all workers in each iteration. We let $K$ denote the global batch size, which is fixed throughout all iterations in synchronous training. Keeping a fixed global batch size is conformal to the standard SGD implementation, which is important for ensuring the same model result in the training process [27], [28]. We assume that $K$ is equally divided among all workers, which implies that the local batch size is $m = \frac{K}{w'_\rho} = \frac{K}{w}$. As a result, the training speed is equal to $\frac{1}{t_m}$ and can be modeled as [20]:

$$f(p,w) = \left( \frac{K}{w} t_f + t_b + 2\frac{g/p}{B/w} + \beta_1 w + \beta_2 p \right)^{-1}.$$

*2) Asynchronous Training:* In asynchronous training, once the PSs receive gradients from a worker, they immediately update their parameters. The expected number of steps completed by each worker in one time unit is $\frac{1}{t_m}$. Thus, we can estimate the total number of steps performed by all workers in one time unit as $\frac{w}{t_m}$. Since $w'_\rho$ is proportional to the number of workers $w$ (i.e., the number of concurrent workers that send gradients to the same PS also increases as $w$ increases), we have $w'_\rho = \alpha w$ for some $\alpha \in (0,1)$. Hence, the training speed function can be modeled as [20]:

$$f(p,w) = w\left( mt_f + t_b + 2\alpha\frac{g/p}{B/w} + \beta_1 w + \beta_2 p \right)^{-1}.$$

### C. Generalization to Advanced Training Models

In this subsection, we introduce two more advanced training models, namely, i) the wait-free model [22] and ii) the priority-based model [9], which overlap communication and

[4]In order to guarantee data transfer performance of each instance, it is common to reserve bandwidth for a VM/container for the accelerated computing in the cluster. For example, the reserved bandwidth of Amazon EC2 GPU instance P2 on AWS is 10Gbps or 25Gbps [25]. Thus we can safely assume that the workers have the same bandwidth.

[5]For lower implementation complexity and managed overhead, most distributed ML frameworks (e.g., Tensorflow [26]) adopt roughly equal parameter allocation by default.

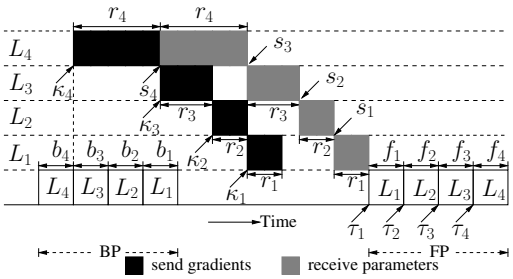


Fig. 4: Wait-free based model.

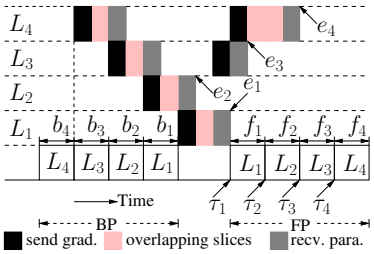


Fig. 5: Priority based model.

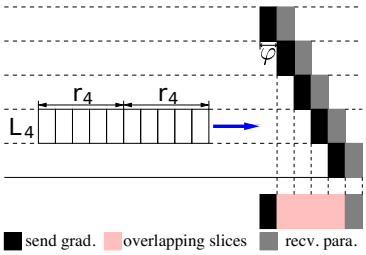


Fig. 6: Zoom-in view of slicing.

computation to further reduce per-iteration delay. We remark that, although both models were not proposed by us, we are the *first* to quantitatively characterize the training speed for both models. Interestingly, it turns out that our training speed modeling for the sequential model can be generalized to these two more advanced communication-computation models (proofs of Lemmas 1 and 2 are omitted due to space limitation).[6]

*1) Wait-free model [22]:* As shown in Fig. 4, starting from the final layer, each layer takes turn to send gradients to the PSs immediately after finishing its own BP and the completion of gradient pushing of its subsequent layer. Compared to the sequential model, a key feature in this model is that part of the communication and computation can be conducted *simultaneously*. We derive per-sample training time $t$ as follows:

**Lemma 1** (Wait-free model)**.** *For the wait-free model, the start time of gradient-sending* $\kappa_j = \max\{\sum_{k=j}^{N} b_k, \kappa_{j+1} + r_{j+1}\}$*, for* $j = 1, \ldots, N-1$*; otherwise it is* $b_N$*. The start time of parameter-receiving* $s_j = \max\{\kappa_j + r_j, s_{j+1} + r_{j+1}\}$*, for* $j = 1, \ldots, N-1$*; otherwise it is* $b_N + r_N$*. The FP start time* $\tau_j = \tau_{j-1} + f_{j-1}$*, for* $j = 2, \ldots, N$*; otherwise it is* $s_1 + r_1$*. It thus follows that the per-sample training time is* $t = \tau_N + f_N$*.*

Some important remarks for Lemma 1 are in order: 1) If the parameter size is skewed in the wait-free model, layers of larger sizes can introduce larger delay to layers of smaller sizes due to non-preemption. 2) In the wait-free model, only after receiving the updated parameters for the first layer, the FP of the next iteration can get started (see $\tau_1$ in Fig. 4); thus the gradient sending of the subsequent layers causes delays to the gradient sending of the initial layer, which in turn induces delay for the next iteration. Also, since the BP progresses in the reverse order of layers (i.e., from the last (output) layer to the first (input) layer), the gradients are also generated and sent in that order. As a result, no overlap between computation and communication is possible during FP (see Fig. 4).

*2) Priority-based model [9]:* An insight from the discussions above is that the closer a layer is to the input layer, the higher priority its gradient/parameter communication should have. The reason is that once the transmission of this layer is completed, the corresponding FP of this layer can start without waiting for all communications to be finished, thus significantly reducing delay. As shown in Fig. 5, the layer with a smaller index (i.e., closer to the input layer) always has a higher priority and its communication can preempt that of layers with larger indices. This induces further communication-computation overlapping and helps the next iteration get started as soon as possible.

Also, to better align communication and computation, the idea of "parameter slicing" can be used to mitigate the impact of skewed layer sizes. As shown in Fig. 6, we split each layer into smaller slices of size $\varphi$, whose communications can be further overlapped. Slices of the same layer have the same priority and the communication order of each slice within the same layer could be arbitrary.

We let $e_j[i]$ be the end time of communication for layer $j$ of job $i$. We derive the per-sample training time $t$ as follows:

**Lemma 2** (Priority-based model)**.** *For the priority-based model, the communication end time* $e_j = \sum_{k=2}^{j} r_k - \sum_{k=1}^{j-1} b_k + \max_{1 \le k \le j-1} e_k$*, if* $\sum_{k=2}^{j} r_k > \sum_{k=1}^{j-1} b_k$*; otherwise* $e_j = 0$ *for* $j = 2, \ldots, N$ *and* $e_1 = \sum_{k=1}^{N} b_k + r_1 + \varphi$*. The FP start time* $\tau_j = \max\{\tau_{j-1} + f_{j-1}, e_j\}$ *for* $j = 2, \ldots, N$ *and* $\tau_1 = e_1$*. It thus follows that the per-sample training time is* $t = \tau_N + f_N$*.*

Note that $t$ has a recursive property. Depending on whether the system is computation or communication dominant in each layer, Lemma 2 could lead to different expressions.

*3) Unified expression for per-iteration training time:* Note that, in both wait-free and priority-based model, the per-sample training time $t$ is DNN-dependent and determined by each layer's size. Nonetheless, as long as the DNN of a training job is given, we can compute $t$ by using Lemmas 1 and 2. Also, we note that the communication-computation overlapping in wait-free and priority-based models effectively reduces the BP time, communication time, and FP time to certain fractions of those in the sequential model. Thus, the per-iteration training time of both models can be expressed as $t_m = \max_{\rho} \left[\eta_1 m t_f + \eta_2 t_b + 2\eta_3 \frac{g/p}{B/w'_\rho} + \beta_1 w + \beta_2 p\right]$, where the coefficients $\eta_1, \eta_2, \eta_3 \in (0, 1]$ are DNN-model-dependent.

We let $H_f$, $H_b$ and $H_r$ be the FP time, BP time and communication time for processing a sample, respectively. Then these coefficients are defined as $\eta_1 \triangleq \frac{H_f}{\sum_{k=1}^{N} f_k}$, $\eta_2 \triangleq \frac{H_b}{\sum_{k=1}^{N} b_k}$, and $\eta_3 \triangleq \frac{H_r}{2\sum_{k=1}^{N} r_k}$. For instance, for the wait-free instance in Fig. 4, we can compute the coefficients as $\eta_1 = \frac{\sum_{k=1}^{4} f_k}{\sum_{k=1}^{4} f_k} = 1$, $\eta_2 = \frac{b_4}{\sum_{k=1}^{4} b_k}$, and $\eta_3 = \frac{2r_4 + r_3 + r_2 + r_1}{2\sum_{k=1}^{4} r_k}$. With this approach,

[6]All missing proofs can be found in our Tech Report [29].

the training speed function of a job under both synchronous training and asynchronous trainings can be generalized as:

$$f(p,w)=1/(\eta_1\frac{K}{w}t_f+\eta_2 t_b+2\eta_3\frac{g/p}{B/w}+\beta_1 w+\beta_2 p), \quad (4)$$

$$f(p,w)=w/(\eta_1 m t_f+\eta_2 t_b+2\eta_3\alpha\frac{g/p}{B/w}+\beta_1 w+\beta_2 p). \quad (5)$$

Note that the sequential model is a special case with $\eta_1=1$, $\eta_2=1$ and $\eta_3=1$. In the next section, we will see that these unified expressions in (4) and (5) enable us to design a suite of approximation algorithms to solve Problem (1).

## IV. Solution Approach

Due to the fundamental hardness of Problem (1) (to be shown soon), in this section, we will propose an approximation algorithmic approach, which we term sum-of-ratio multi-dimensional-knapsack decomposition (SMD), to solve this problem. In what follows, we will organize and present our approximation algorithmic approach in three main steps:

*Step 1) Sum-of-Ratios Multi-Dimensional-Knapsack Decomposition (SMD):* First, we note that in Problem (1), the resource scheduling decision variables $(w[i],p[i])$ are independent across jobs due to the fact that each summand in (1) only depends on each job $i$. Therefore, we can decompose the problem into an inner subproblem and an outer subproblem as follows. First, by setting $x_i=1,\forall i$, the inner resource allocation sub-problem for job $i$ can be written as:

$$\underset{w,p}{\text{Maximize}} \quad \mu\Big(\frac{E}{f(p,w)}\Big) \quad (6)$$

$$\text{subject to} \quad (O^r w+G^r p)\le v^r, \quad \forall r\in\mathcal{R}, \quad (7)$$

$$p\in\mathbb{Z}^{++}, w\in\mathbb{Z}^{++}. \quad (8)$$

Recall that the training speed function $f(p,w)$ has different forms for synchronous and asynchronous trainings. Hence, Problem (6) can be further specialized as follows:

*a) Synchronous training:* In this case, we have:

$$\underset{w,p}{\text{Maximize}} \ \mu\bigg(\theta^1 w+\theta^2 p+\theta^3+\frac{\theta^4 w}{p}+\frac{\theta^5}{w}\bigg) \quad (9)$$
$$\text{subject to Constraints } (7)-(8),$$

where $\theta^1=E\beta_1$, $\theta^2=E\beta_2$, $\theta^3=E\eta_2 t_b$, $\theta^4=2E\eta_3 g/B$, and $\theta^5=\eta_1 EKt_f$.

*b) Asynchronous training:* In this case, we have:

$$\underset{w,p}{\text{Maximize}} \ \mu\bigg(\theta'^1+\frac{\theta'^2 p}{w}+\frac{\theta'^3}{w}+\frac{\theta'^4}{p}\bigg) \quad (10)$$
$$\text{subject to Constraints } (7)-(8),$$

where $\theta'^1=E\beta_1$, $\theta'^2=E\beta_2$, $\theta'^3=E(\eta_1 m t_f+\eta_2 t_b)$, and $\theta'^4=2E\alpha\eta_3 g/B$.

It is clear that Problem (6) is a mixed-integer nonlinear programming (MINLP) problem, which is NP-Hard in general [30]. In addition, even with continuous relaxation, it remains in the class of sum-of-ratios optimization problems, which is well-known to be NP-complete [31]. But thanks to the low dimensionality of the inner subproblem (a consequence of SMD), we will propose an $\epsilon$-approximation algorithm for solving the inner subproblem. Upon solving (6), the outer subproblem reduces to selecting active jobs to be run in the current scheduling interval:

$$\underset{\mathbf{x}}{\text{Maximize}} \ \sum_{i\in\mathcal{I}}\mu_i\bigg(\frac{E[i]}{f(p[i],w[i])}\bigg)x_i \quad (11)$$
$$\text{subject to Constraint } (2), x_i\in\{0,1\},\forall i\in\mathcal{I}.$$

Since $\mu_i(\frac{E[i]}{f(p[i],w[i])})$ is known after solving the inner subproblem, it is clear that the outer subproblem is a multi-dimensional knapsack problem (MKP) in essence. Therefore, in what follows, we will consider solving the inner sum-of-ratios subproblem and the outer MKP problem separately.

*Step 2) Solving the inner sum-of-ratios subproblem:* We first consider the inner subproblems (9) and (10) after relaxing the integrality constraint (8). Recall that the utility function $\mu_i(\cdot)$ is non-increasing. Thus, both problems can be equivalently reformulated as a sum-of-ratios problem with affine constraints, which can be written in the following general form:

$$\underset{\mathbf{x}}{\text{Minimize}} \ \zeta(\mathbf{x})=\sum_{j\in\mathcal{J}}\frac{\mathbf{a}_j^\top\mathbf{x}+q_j}{\mathbf{c}_j^\top\mathbf{x}+d_j}=\sum_{j\in\mathcal{J}}\zeta_j(\mathbf{x}) \quad (12)$$
$$\text{subject to } \mathbf{A}\mathbf{x}\le\mathbf{C},\mathbf{x}\ge\mathbf{0},$$

where $\mathbf{a}_j,\mathbf{c}_j\in\mathbb{R}^n, q_j,d_j\in\mathbb{R}$, $\forall j\in\mathcal{J}$, and $\mathbf{A}\in\mathbb{R}^{m\times n}, C\in\mathbb{R}^m$. We also note that in both problems $\mathbf{a}_j^\top\mathbf{x}+q_j>0, \mathbf{c}_j^\top\mathbf{x}+d_j>0,\forall j\in\mathcal{J},\forall\mathbf{x}\in\Omega$, where $\Omega=\{\mathbf{x}\in\mathbb{R}^n|\mathbf{A}\mathbf{x}\le\mathbf{C},\mathbf{x}\ge\mathbf{0}\}$ denote the feasible domain of the problem. The sum-of-ratios problem with affine constraints is known to be NP-complete [31] but $\epsilon$-approximation approach [32] exists.

Unlike the problem in [32] with covering constraints (may lead to unbounded search space), we exploit the special *packing-like* constraint structure in (7) to first obtain a tight upper bound for each ratio term to significantly reduce the search space. Specifically, we let $\zeta_j(\mathbf{x})\triangleq\frac{\mathbf{a}_j^\top\mathbf{x}+q_j}{\mathbf{c}_j^\top\mathbf{x}+d_j}$, which is a linear fractional programming with non-empty and bounded feasible region $\Omega$. We choose the lower and upper bounds as $l_j=\min_{\mathbf{x}\in\Omega}\zeta_j(\mathbf{x})$ and $\phi_j=\max_{\mathbf{x}\in\Omega}\zeta_j(\mathbf{x})$. Then transform the problem into a linear program using the Charnes-Cooper transformation [33], which then can be solved efficiently. Without loss of generality, we assume that the last summand $J$ has the largest ratio between the upper and lower bounds, i.e., $J=\arg\max_{j\in\mathcal{J}}\left\{\frac{\phi_j}{l_j}\right\}$. Then, the feasible domain for the $J-1$ summands is a polytope characterized as $\mathcal{H}=[l_1,\phi_1]\times[l_2,\phi_2]\times\ldots\times[l_{J-1},\phi_{J-1}]$. We let $\chi\in\mathbb{R}^{J-1}$ and $z\in\mathbb{R}$. Then, Problem (12) can be transformed into the following equivalent formulation:

$$\underset{\mathbf{x}\in\Omega,z}{\text{Minimize}} \ \sum_{j=1}^{J-1}\chi_j+z \quad (13)$$
$$\text{subject to } \zeta_j(\mathbf{x})\le\chi_j, j=1,..,J-1,$$
$$\zeta_J(\mathbf{x})=z,$$
$$\chi=(\chi_1,..,\chi_{J-1})\in\mathcal{H}.$$

We can see from the reformulated Problem (13) that, if a point $\chi \in \mathcal{H}$ is given, there is only one variable $z$ associated with the summand $\zeta_J(\mathbf{x})$ to be solved. Thus, the complexity of the problem is reduced significantly.

*Step 2.1): Determining the Set of Grid Points:* After finding the range for each summand in the objective function, we divide the polytope $\mathcal{H}$ into smaller polytopes to perform a grid search, where the granularity is controlled by a precision parameter $\epsilon$. We first find the largest integer number that does not exceed the upper bound $\phi_j$ of each summand when searching from the lower bound $l_j$, i.e., $\lambda_j = \arg\max\{n \in \mathbb{N} | l_j(1+\epsilon)^n \leq \phi_j\}, j = 1, ..., J-1$. Then, we have the grid points set for each summand as $\mathcal{Q}_j^\epsilon = \{l_j, l_j(1+\epsilon), \ldots, l_j(1+\epsilon)^{\lambda_j}\}, j = 1, ..., J-1$. Next, by searching all the $J-1$ summands, we can obtain the search grid set as $\mathcal{T}^\epsilon = \{(\nu_1, \nu_2, \ldots, \nu_{J-1}) | \nu_j \in \mathcal{Q}_j^\epsilon, j = 1, ..., J-1\}$. It is clear that for any $(\chi_1, \chi_2, \ldots, \chi_{J-1}) \in \mathcal{H}$, we can always find a point $(\nu_1, \nu_2, .., \nu_{J-1}) \in \mathcal{T}^\epsilon$, such that $\chi_j \in [\nu_j, (1+\epsilon)\nu_j], j = 1, ..., J-1$, thus $\mathcal{H}$ can be approximated by the set $\mathcal{T}^\epsilon$.

Hence, Problem (13) can be solved by iterating over $\nu \in \mathcal{T}^\epsilon$. For a given $\nu$, the subproblem needs to be solved is as follows:

$$\begin{aligned} \underset{\mathbf{x}\in\Omega, z}{\text{Minimize}}\ & \Psi(\nu) = \sum_{j=1}^{J-1} \nu_j + z \\ \text{subject to}\ & \zeta_j(\mathbf{x}) \leq \nu_j, j = 1, ..., J-1, \\ & \zeta_J(\mathbf{x}) = z. \end{aligned} \tag{14}$$

Notice that for a given $\nu \in \mathcal{T}^\epsilon$, the term $\sum_{j=1}^{J-1} \nu_j$ becomes a constant, then the equivalent formulation to Problem (14) is:

$$\begin{aligned} \underset{\mathbf{x}\in\Omega}{\text{Minimize}}\ & \zeta_J(\mathbf{x}) = \frac{\mathbf{a}_J^\top \mathbf{x} + q_j}{\mathbf{c}_J^\top \mathbf{x} + d_J} \\ \text{subject to}\ & \zeta_j(\mathbf{x}) \leq \nu_j, j = 1, .., J-1, \end{aligned} \tag{15}$$

which again can be transformed into a linear program using the Charnes-Cooper transformation [33] and solved efficiently.

The basic idea of the algorithm is first to perform the dimensionality reduction to reduce $J$ summands to $J-1$ terms, and then divide the feasible domain into smaller nonuniform grids. The feasible polytope domain obtained by finding the lower and upper bound of each summand is used to confine the space of grid points. Then, the original sum-of-ratios problem can be transformed and decomposed into a set of linear programming (LP) subproblems, each of which is associated with a grid point. As a result, the computational cost boils down to solving LP subproblems related to points in $\mathcal{T}^\epsilon$.

Note, however, that the total number of grid points still increases exponentially as the number of summands in (12) increases, which is intractable as the problem size gets large. Fortunately, we note that both inner problems (9) and (10) have just a few summands (four and three, respectively) thanks to SMD. Thus, it remains affordable to adopt a grid-search-based approach. By leveraging the special feature of cloud systems that each job has its reserved resources, we can reduce the high dimensionality of the problem and decompose it as in Problem (6). Thus, the problem is reduced to solving $I$ times of sum-of-linear-ratios with a small number of terms, which can be solved efficiently (Algorithm 1).

**Algorithm 1:** $\epsilon$-Approximation for the Continuous Relaxation of the Inner Subproblems (9) and (10).

1 **Initialization:** Set $\tilde{L} = +\infty$. Let $\epsilon \in (0,1)$, $\tilde{\mathbf{x}} = \emptyset$;
2 Obtain the set $\mathcal{T}^\epsilon$ as described in Step 2.1);
3 **for** $\nu \in \mathcal{T}^\epsilon$ **do**
4 Solve Problem (14) to obtain the solution $\mathbf{x}^\nu$ with the objective value $\Psi(\nu) = \sum_{j=1}^{J-1} \nu_j + \zeta_J(\mathbf{x}^\nu)$;
5 **if** $\Psi(\nu) < \tilde{L}$ **then**
6 $\tilde{L} = \Psi(\nu)$, $\tilde{\mathbf{x}} = \mathbf{x}^\nu$;
7 **return** $\tilde{\mathbf{x}}$;

**Algorithm 2:** Randomized Rounding Scheme.

1 Pick some $\delta \in (0,1]$. Pick some integer $F \geq 1$. Let $cnt \leftarrow 0$. Let $\mathbf{x}' = M_\delta \bar{\mathbf{x}}$ for some $0 < M_\delta \leq 1$ ($M_\delta$ signifies its dependence on $\delta$ and is to be specified);
2 Randomly round $\mathbf{x}'$ to $\hat{\mathbf{x}} \in \mathbb{Z}_+^n$ as: $\hat{x}_j = \lceil x'_j \rceil$ w.p. $x'_j - \lfloor x'_j \rfloor$ and $\hat{x}_j = \lfloor x'_j \rfloor$ w.p. $\lceil \bar{x}_j \rceil - \bar{x}_j$, otherwise;
3 If $\hat{\mathbf{x}}$ is infeasible or $cnt < F$, then $cnt \leftarrow cnt+1$, go to 2.

In Algorithm 1, we first obtain the grid points over the feasible domain in Line 2. We then iterate each point and update the objective $\tilde{L}$ and the fractional solution $\tilde{\mathbf{x}}$ in Line 6 if the current objective value is smaller. Finally, we return the solution with the smallest objective value. Following similar analysis as in [32] (hence omitted for brevity), we can show that Algorithm 1 is an $\epsilon$-approximation.

Upon solving the continuous relaxation of inner sum-of-ratios subproblems (9) and (10), it remains to obtain an integer solution to calculate the utility. This is still an NP-Hard integer programming problem with generalized packing-type constraints in (7). We propose the following solution approach: First, we solve the continuous relaxation of $\min\left\{ \sum_{l=1}^{L} \frac{\mathbf{a}_l^\top \mathbf{x}}{\mathbf{d}_l^\top \mathbf{x}} + \mathbf{c}^\top \mathbf{x} : \mathbf{B}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \in \mathbb{Z}_+^n \right\}$, where $\mathbf{B} \in \mathbb{R}_+^{r\times n}$, $\mathbf{b} \in \mathbb{R}_+^r$, and $\mathbf{a}_l, \mathbf{d}_l, \mathbf{c} \in \mathbb{R}_+^n, \forall l$. Let $\bar{\mathbf{x}}$ be the obtained fractional optimal solution. Then, we propose a randomized rounding scheme as shown in Alg. 2 to round $\bar{\mathbf{x}}$ and arrive at an integer solution.

*Step 3) Solving the outer MKP subproblem:* The general formulation of the outer MKP subproblem can be written as:

$$\begin{aligned} \underset{\mathbf{x}}{\text{Maximize}}\ & \sum_{i\in\mathcal{I}} u_i x_i \\ \text{subject to Constraints (2)}, & x_i \in \{0,1\}, \forall i \in \mathcal{I}, \end{aligned} \tag{16}$$

where $u_i \triangleq \mu_i(\frac{E[i]}{f(p[i],w[i])})$ is the utility value of each job $i$. Our goal is to select jobs that maximize the total utility among all the submitted jobs $\mathcal{I}$. The MKP problem is still a well-known NP-Hard problem [34] but admits an $\epsilon$-approximation solution[7] that runs in polynomial time if $\epsilon$ and $I$ are fixed. Here, we adopt the $\epsilon$-approximation scheme [35] to our

[7]This $\epsilon$-value is different and should not be confused with the $\epsilon$ in Step 2.

problem setting. We let $T(\mathcal{S}) = \{t \in \mathcal{I} \backslash S : u_t > \min(u_i : i \in \mathcal{S})\}$, for $\mathcal{S} \subset \mathcal{I}$. Let $LP(\mathcal{S})$ be the linear program obtained from Problem (16) by setting $x_i = 1$, if $i \in \mathcal{S}$; and set it to 0, if $i \in T(\mathcal{S})$. Let $\mathbf{x}^B(\mathcal{S})$ be an optimal basic feasible solution to $LP(\mathcal{S})$. The main idea of this approach is to solve $LP(\mathcal{S})$ for all $\mathcal{S} \subset \mathcal{I}$. Then, we round down the solution $\mathbf{x}^B(\mathcal{S})$. Finally, we return the best solution with the largest objective value. The difference between $\mathbf{x}^B(\mathcal{S})$ and $\lfloor \mathbf{x}^B(\mathcal{S}) \rfloor$ is small since $R$ is a fixed small number in our case.

### A. Performance Analysis

We now examine the overall approximation ratio of our proposed algorithms. Note that the key component in our algorithm is the proposed randomized rounding scheme in Algorithm 2 for Problem (6). Thus, we first prove the following result for the randomized rounding algorithm (proofs are omitted due to space limitation).

**Lemma 3** (Rounding)**.** *Let $W_b \triangleq \min\{b_i/[\mathbf{B}]_{ij} : [\mathbf{B}]_{ij} > 0\}$. Let L be the number of sum-of-ratios terms. Pick some constant $\delta \in (0, 1]$, and define $M_\delta$ as:*

$$M_\delta \triangleq 1 + \frac{3\ln(2r/\delta)}{2W_b} - \sqrt{\left(\frac{3\ln(2r/\delta)}{2W_b}\right)^2 + \frac{3\ln(2r/\delta)}{W_b}}.$$

*With probability greater than $1-\delta$, $\hat{\mathbf{x}}$ achieves a cost at most $\frac{8L/M_\delta+4}{\delta}$ times the cost of $\bar{\mathbf{x}}$, and $\Pr\{(\mathbf{B}\hat{\mathbf{x}})_i > b_i, \exists i\} \leq \frac{\delta}{2r}$.*

Note that $\delta$ is used for characterizing the randomized rounding algorithm's performance. Lemma 3 indicates that with probability $1-\delta$, one achieves an approximation ratio at most $\frac{8L/M_\delta+4}{\delta}$ with the stated probabilistic feasibility guarantee. From the statement, we can see that the approximation ratio is ultimately determined by $\delta$, since $M_\delta$ increases as $\delta$ increases. Thus, if one desires a better approximation ratio, then a larger $\delta$ should be picked. That is, there exists a trade-off between the approximation ratio value and its achieving probability, both of which are quantified by $\delta$.

The approximation ratio of our algorithm is the worst-case upper bound of the ratio between the overall utility of admitted jobs obtained by the optimal solution of Problem (1) and the total utility achieved by Algorithm SMD in the overall time horizon. By specializing Lemma 3 with parameters in Problem (6), we have the following result for Algorithm 2:

**Theorem 4** (Approximation Ratio of Rounding in Alg. 2)**.** *Let $\delta$ be selected as in Lemma 3, and let $M_\delta$ be defined as in Lemma 3. With probability greater than $1-\delta$, Algorithm 2 obtains a schedule $\{w[i], p[i], \forall i\}$ that has an approximation ratio at most $\frac{24/M_\delta+4}{\delta}$ with $\Pr\{LHS(7) > v^r[i]\} \leq \frac{\delta}{8}$.*

Let $\epsilon_1$ and $\epsilon_2$ be the performance ratios of solving the sum-of-ratios problem and MKP, respectively, where $\epsilon_1, \epsilon_2 \in (0, 1)$. Let $\delta$ and $M_\delta$ be defined as in Lemma 3. Let $\tau_i$ be the completion time of job $i$ and $\tau_i^*$ be the optimal completion time of job $i$. Let $F$ be chosen as in Alg. 2. We let $\mu^* \triangleq \max_i\{\mu_i(\tau_i^*)\}$, and let $\mu' \triangleq \min_i\{\mu_i\big(\tau_i^*(1+\epsilon_1)(24/M_\delta+4)/\delta\big)\}$. Following Theorem 4, we can establish the overall approximation ratio and running time complexity of the SMD approach as follows:

**Theorem 5** (Overall Approximation Ratio of SMD)**.** *With probability greater than $(1-(\delta/8)^F)^I$, the proposed SMD-based method returns a feasible solution with $\frac{\mu'(1-\epsilon_2)}{\mu^*}$-approximation performance guarantee.*

**Theorem 6** (Polynomial Running Time)**.** *We let $T_i^s$ and $T_i^a$ be the time complexity for solving the sum-of-ratios problem under synchronous and asynchronous training of job $i$, respectively, which can be solved in polynomial time [32]. Let $T_i = \max\{T_i^s, T_i^a\}$. Let $T_2$ be the time complexity for MKP problem, which is polynomial [35]. Then, the overall time complexity of Algorithm SMD is $O(\sum_{i\in\mathcal{I}}(T_i + F) + T_2)$.*

## V. Numerical Evaluation

We conduct simulation studies to evaluate the efficacy of our proposed algorithms. In our simulation, the computing cluster follows the real-world system in [36] with job parameters generated uniformly at random from the following intervals: $E[i] \in [50, 200]$, $g[i] \in [30, 575]$ MB, $m[i] \in [10, 100]$, $K[i] \in [1, 100] \cdot m[i]$, $N[i] \in [10, 100]$. We consider four types of resources: GPU, CPU, memory and storage. For fair comparisons, we use similar settings as in [24] [17] [16] and set resource demands of each worker as: 0–4 GPUs, 1–10 vCPUs, 2–32 GB memory, and 5–10GB storage. We set resource configuration of each PS as: 1–10 vCPUs, 2–32GB memory and 5-10GB storage. We set bandwidth capacity of each PS to $B[i] \in [5, 20]$ Gbps. Capacities of virtual instances to run workers/PSs are set according to resource configuration of Amazon EC2 C4 instances. We set the resource limit of each job to $\vartheta$ times of the resource limit of each instance with $\vartheta \in [1, 20]$ (according to the reality that each user is limited to a maximum of 20 instances per region in Amazon EC2). We set $b_j[i] \in [1, 300]$ ms, $f_j[i] \in [1, 500]$ ms and $r_j[i] \in [80, 500]$ ms following the traces collected from the experiments in Optimus [20] based on Google cluster trace [11], which include jobs' training losses, training speeds with various resource configuration, each server's resource capacities, job configuration such as requirements of workers/PSs, as well as DL model specifications like parameter size. Then, we set $t_b[i] = \sum_{j=1}^{N[i]} b_j[i]$ and $t_f[i] = \sum_{j=1}^{N[i]} f_j[i]$. We set $\beta_1[i] \in [3, 4]$, $\beta_2[i] \in [0, 0.01]$ and $\alpha[i] \in [0, 1]$ following the tested values from Optimus [20]. We use a Sigmoid utility function [20], [37], $\mu_i(\pi_i) = \frac{\gamma_1}{1+e^{\gamma_2(\pi_i-\gamma_3)}}$ with $\gamma_1 \in [1, 100]$, $\gamma_2 \in [4, 6]$ and $\gamma_3 \in [1, 15]$. Note that this range of $\gamma_2$ corresponds to time-critical jobs [21].

We first compare our SMD algorithm with two baseline resource allocation policies: (1) ESW (setting the ratio of number of workers to number of PSs to 1:1 [38] for each job); and (2) Optimus [20] (compare the utility gain by adding one more worker and one more PS and choose the one with larger utility gain). Since Optimus estimates the training speed function based on an online learning approach by monitoring the convergence rate, we use our own speed function to estimate the utility for each job. We set $\epsilon_1 = \epsilon_2 = 0.01$, $M_\delta = 1$, and $I = 50$. To study how the total utility changes as the computing cluster resource capacity increases,

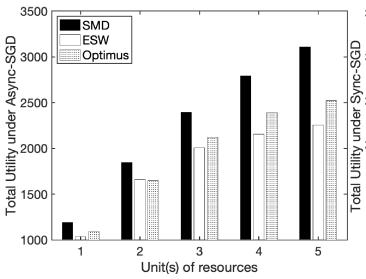


Fig. 7: Total utility vs. cluster resources (Async-SGD).

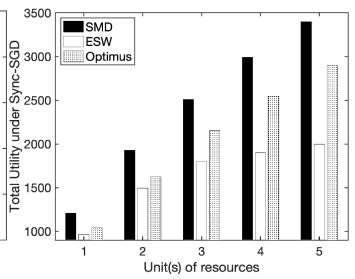


Fig. 8: Total utility vs. cluster resources (Sync-SGD).

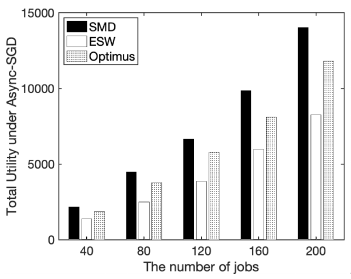


Fig. 9: Total utility vs. number of jobs (Async-SGD).

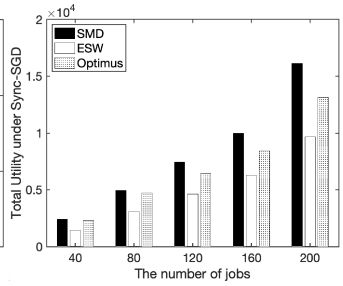


Fig. 10: Total utility vs. number of jobs (Sync-SGD).

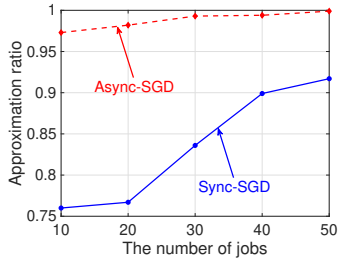


Fig. 11: The comparison of approximation ratios.

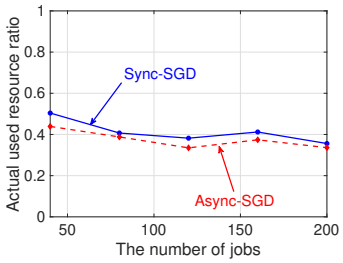


Fig. 12: The actual used resources ratios.

we set one unit of resources as follows: (vCPU $=$ 3400, GPU $=$ 600, Memory $=$ 1400 GB, Storage $=$ 1200 GB), and vary the resource capacity using 1-5 times of the unit resource. The comparison results under both Sync-SGD and Async-SGD are shown in Figs. 7–10. We can see that SMD significantly outperforms other policies and the gains in total utility becomes more pronounced as the number of jobs and resources in the computing cluster increases.

Next, we examine the approximation ratio of SMD. We evaluate the performance in terms of the ratio between the total utility obtained by our algorithm and the optimal total utility. The optimal utility is computed by enumerating all the possible combinations of numbers of workers and PSs for each job, and the combination with the largest utility will be returned. We vary the number of jobs per scheduling interval from 10 to 50. We also set the cluster resource capacity as 1000 times of that of a virtual instance. The results are shown in Fig. 11. We can see that the ratio is much better than the theoretical bound and becomes larger as the number of jobs increases, which implies that our algorithm is scalable. Further, Sync-SGD has a worse approximation ratio since it is more sensitive to the changes of numbers of workers and PSs based on Eqn. (9) due to the linear term $\theta^1 w + \theta^2 p$. In other words, the error introduced from the "grid search" and randomized rounding when solving the inner sum-of-ratios-subproblem could lead to more utility loss compared to asynchronous training. Recall that the randomized rounding scheme is the key of our proposed Algorithm SMD. The packing constraints (7) are easier to satisfy with a smaller $M_\delta$. Theorem 4 suggests that there is a trade-off: if we set $M_\delta$ to be close to one to pursue a better total utility result, the rounding time could be large to obtain a feasible solution. As $M_\delta$ becomes larger, the probability of violating the packing constraints increases, meaning that we need to have more rounding attempts to obtain an integer feasible solution. However, according to our numerical experiences, if the machine's resource capacity is relatively large compared to the jobs' resource demands per worker/PS, the number of rounding attempts is small and not sensitive to $M_\delta$.

Lastly, we examine the actual used resources in our algorithm. A key feature of sum-of-ratios problems is that optimality is not necessarily obtained when the resource capacity constraints are binding. In other words, compared to the number of workers and parameter servers, the ratio between the number of workers and parameter servers plays a more critical role to minimize the training completion time. If such a better ratio can be found, it is possible that the system can save resources while having the same or even better performance in terms of the average training completion time. Hence, compared to other resource allocation policies that use as much resources specified by the user as possible, our SMD method may use much less resources while achieving the optimal performance. We let $\epsilon = 0.01$ and vary the number of jobs per scheduling interval from 40 to 200. We can see from Fig. 12 that the actual resources used is 30%-50% of that specified by the users. From the system's perspective, the unused resources can be released and allocated to other jobs.

## VI. Conclusions

In this paper, we studied resource scheduling for DNN jobs in computing clusters. We demonstrated that the problem can be formulated as a non-convex integer non-linear program with bin-packing constraints, which is NP-Hard. We proposed an approximation scheduling algorithm based on a sum-of-ratios multi-dimensional knapsack (SMD) approach. Specifically, we developed a performance model that considers the special layered structure of DNN under different parameter synchronization mechanisms. Through careful investigation of the structure of the non-convex problem, we decomposed the problem based on SMD and proposed a suite of approximation techniques to solve the packing-type integer program with performance guarantees. Evaluation under realistic settings confirmed superior performances of SMD over existing works. DNN resource scheduling remains an under-explored area. Future extensions of this work may include, e.g., datacenter topology and communication contention among jobs.

## REFERENCES


[1] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proc. of the 10th ACM European Conference on Computer Systems (Eurosys)*, 2015.

[2] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. of USENIX OSDI*, 2018.

[3] J. Gu, K. G. Chowdhury, M. abd Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A gpu cluster manager for distributed deep learning," in *NSDI*, 2019.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[5] "Mlperf training results." [Online]. Available: https://mlperf.org/training-results-0-6/

[6] E. B. Dario Amodei, Rishita Anubhai, C. Case, J. Casper, B. Catanzaro, J. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *Proc. of the 33th International Conference on Machine Learning (ICML)*, 2016.

[7] "Librispeech asr corpus." [Online]. Available: http://www.openslr.org/12

[8] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," in *Proceedings of Systems and Machine Learning (SysML)*, 2019.

[9] J. W. Anand Jayarajan, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," in *Proceedings of Systems and Machine Learning (SysML)*, 2019.

[10] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[11] "Google cluster workload traces," 2015. [Online]. Available: https://github.com/google/cluster-data

[12] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proc. of ACM EuroSys*, 2018.

[13] 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 the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[14] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy *et al.*, "Morpheus: Towards automated slos for enterprise clusters," in *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[15] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. HarcholBalter, and G. R. Ganger, "Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proc. of the 11th ACM European Conference on Computer Systems (Eurosys)*, 2016.

[16] 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.

[17] 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.

[18] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance modeling and scalability optimization of distributed deep learning systems," in *Proc. of the 21th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2015.

[19] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, and A. Akella, "Themis: Fair and efficient gpu cluster scheduling," in *NSDI*, 2020.

[20] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proc. of the 13th ACM European Conference on Computer Systems (Eurosys)*, 2018.

[21] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online job scheduling in distributed machine learning clusters," in *Proc. of IEEE INFOCOM*, 2018.

[22] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing2, "Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters," in *In 2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 181–193.

[23] J. Dean, G. S. Corrado, R. Monga, K. Chen, and others., "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1223–1231.

[24] M. Li, D. G. Andersen *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. of USENIX OSDI*, 2014.

[25] "Amazon ec2 instances," https://aws.amazon.com/ec2/instance-types/.

[26] M. Abadi, P. Barham *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. of USENIX OSDI*, 2016.

[27] P. Goyal, P. Dolla, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[28] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty, *Nonlinear Programming: Theory and Algorithms*, 3rd ed. New York, NY: John Wiley & Sons Inc., 2006.

[29] M. Yu, C. Wu, B. Ji, and J. Liu, "A sum-of-ratios multi-dimensional-knapsack decomposition for dnn resource scheduling," Ohio State University, Tech. Rep., 2021. [Online]. Available: https://kevinliu-osu-ece.github.io/publications/ML_Networking_SMD_PERP'20.pdf

[30] R. W. Freund and F. Jarre, "Solving the sum-of-ratios problem by an interior-point method," in *Journal of Global Optimization*, 2001.

[31] S. Schaible, "A note on the sum of a linear and linear-fractional function," in *Naval Research Logistics Quarterly*, 1977.

[32] P. Shen, B. Huang, and L. Wang, "Range division and linearization algorithm for a class of linear ratios optimization problems," in *J. Comput. Appl. Math*, vol. 350, 2019, pp. 324–342.

[33] A. Chames and W. Cooper, "Programming with linear fractional functionals,," in *Naval Research Logistics Quarterly*, 1962.

[34] P. Chu and J. Beasley, "A genetic algorithm for the multidimensional knapsack problem," in *Journal of Heuristics*, vol. 4, 1998, pp. 63–86.

[35] A. FRIEZE and M. CLARKE, "Approximation algorithms for the m-dimensional 0-1 knapsack problem: Worst-case and probabilistic analysis," in *European Journal of Operational Research*, vol. 15, 1984, pp. 100–109.

[36] 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.

[37] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. Tsang, "Need for speed: Cora scheduler for optimizing completiontimes in the cloud," in *Proc. of IEEE INFOCOM*, 2015.

[38] "Run deep learning with paddlepaddle on kubernetes," 2017. [Online]. Available: https://kubernetes.io/blog/2017/02/run-deep-learning-with-paddlepaddle-on-kubernetes/