CCF Transactions on High Performance Computing (2021) 3:171–185
https://doi.org/10.1007/s42514-021-00064-x

REGULAR PAPER

Rationing bandwidth resources for mitigating network resource contention in distributed DNN training clusters

Qiang Qi[1] · Fei Xu[1] · Li Chen[2] · Zhi Zhou[3]

Received: 31 August 2020 / Accepted: 6 February 2021 / Published online: 2 March 2021

Abstract

Distributed deep neural network (DDNN) training becomes increasingly compelling as the DNN model gets complex and the dataset grows large. Through an in-depth analysis of the latest Microsoft GPU cluster trace, we show that the *co-located* Parameter Server (PS) configuration is not uncommon in production DDNN training clusters, which inevitably causes intense network resource contention among the co-located PS and worker tasks. Our motivation experiments on Amazon EC2 further show that such network resource contention brings severe performance variation to DDNN training jobs. While existing works largely mitigate the *inter-job* network resource contention, the *intra-job* (i.e., task-level) network resource contention among the co-located PS and worker tasks has received comparably little attention. To tackle such performance issues, in this paper, we design and implement `Nebula`, a **Ne**twork **b**andwidth reso**u**rce al**l**oc**a**tion strategy for DDNN training tasks, in order to mitigate the network resource contention and alleviate the performance variation of DDNN training jobs. `Nebula` monitors the *weights* of co-located PS and workers and rations the network bandwidth resources for the two tasks by comparing the corresponding task weights. We implement a prototype of `Nebula` and conduct extensive prototype experiments with representative DNN models trained on Amazon EC2. Our experiment results demonstrate that `Nebula` can reduce the iteration time of a DDNN training job by up to 25% and improve the cluster resource utilization by up to 30% in comparison to MXNet, yet with practically acceptable runtime overhead.

Keywords Distributed DNN training · Bandwidth allocation · Network resource contention

1 Introduction

Distributed deep neural network (DDNN) training has received widespread attention recently, as it is able to train the DNN models in parallel with different approaches such as data parallelism (Mayer and Jacobsen 2020), model parallelism (Mirhoseini et al. 2017), and pipeline parallelism (Narayanan et al. 2019). Among these parallelism methods above, data parallelism with the Parameter Server (PS) architecture has been widely adopted in production DDNN training clusters (Gu et al. 2019) of big companies like Google and Microsoft. To reduce the intermediate data movement and achieve good training performance and scalability, the *co-located* PS configuration is set as the *default* (i.e., not uncommon) in popular DDNN training frameworks (e.g., MXNet) (Luo et al. 2018). Our analysis of the latest Microsoft GPU cluster (Jeon et al. 2019) in Sect. 2.1 further shows that such a co-located PS configuration is deployed on around 77% of machines. By co-locating the PS and worker tasks on the same machine, however, the widely-adopted

✉ Fei Xu
fxu@cs.ecnu.edu.cn

Qiang Qi
51184506067@stu.ecnu.edu.cn

Li Chen
li.chen@louisiana.edu

Zhi Zhou
zhouzhi9@mail.sysu.edu.cn

[1] Shanghai Key Laboratory of Multidimensional Information Processing, School of Computer Science and Technology, East China Normal University, Shanghai, China

[2] School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, USA

[3] Guangdong Key Laboratory of Big Data Analysis and Processing, School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China

co-located PS configuration can cause severe *network resource contention* among PS and worker tasks, thereby degrading the DDNN training performance.

To alleviate the network resource contention and speed up DDNN training performance, there have been many research works dedicated to reducing the network traffic, such as model quantization (Guptaand et al. 2015) and gradient sparsification (Lin et al. 2017). Nevertheless, reducing the network traffic cannot *fundamentally* solve the network resource contention problem, simply because the compression of communication data (i.e., model parameters, gradient data) can bring extra computation overhead and the compressed data is still likely to be large. Meanwhile, there have also been recent works to mitigate the *inter-job* network resource contention, which is mainly caused by the contention of network resources among *multiple jobs* co-located on the same machine. Such an issue can largely be solved by scheduling or placing the DDNN training jobs to the appropriate machines in GPU clusters (Wang et al. 2020b), while considering several other key factors such as fairness (Kshiteej et al. 2020) and interference (Ukidave et al. 2016). However, there has been scant research attention paid to the *intra-job* (i.e., task-level) network resource contention among the co-located PS and worker tasks of one DDNN training job.

The *intra-job* network resource contention in the co-located PS configuration can cause severe *performance variation* to DDNN training jobs. As evidenced by our motivation experiments in Sect. 2.2, the iteration time for the ResNet152 model trained in the co-located PS configuration can vary five times larger than that in the non-colocated PS configuration, leading to a comparatively low network resource utilization and thus prolonging the DDNN training time as discussed in Sect. 2.3. By analyzing the network communication mechanism of MXNet (Chen et al. 2015), we further confirm that such intense *intra-job* network resource contention can be caused by the *uneven distribution of model parameters among different PS*, which inevitably makes the communication operations of co-located PS and worker tasks compete for network uplink or downlink bandwidth resources on the same machine.

To deal with such performance issues above, in this paper, we design `Nebula`, *a simple yet effective* network bandwidth allocation strategy to mitigate the network resource contention and alleviate the performance variation of DDNN training jobs. Specifically, we first conduct a theoretical analysis of the performance variation caused by the *intra-job* network resource contention among the co-located PS and worker tasks. Based on such an in-depth analysis, we further design our `Nebula` strategy to adequately allocate the network bandwidth online for the co-located PS and worker tasks by monitoring the corresponding task weights. We implement a prototype of `Nebula` consisting of a `Nebula` monitor and a `Nebula` controller. With the aim of minimizing the training iteration time, the `Nebula` controller adequately rations the network bandwidth resources for PS and workers tasks by comparing the task weights calculated by the `Nebula` monitor. To the best of our knowledge, `Nebula` is the first attempt to analyze and solve the *intra-job* network resource contention among the co-located PS and worker tasks during the execution of a DDNN training job.

We evaluate the effectiveness and runtime overhead of `Nebula` with extensive prototype experiments on a cluster of g3.8xlarge instances in Amazon EC2. Our experimental results with four representative DNN models (i.e., AlexNet, ResNet101, ResNet50, and VGG16) show that `Nebula` can reduce the iteration time of DDNN training jobs by 15.2–25.0%, and improve the average utilization of CPU and network cluster resources by up to 30%, as compared with MXNet. In addition, `Nebula` incurs acceptable runtime overhead in practice.

The rest of the paper is organized as follows. Section 2 illustrates the severity of DDNN training performance variation caused by network resource contention in the *co-located* PS configuration. Through analyzing such a performance variation problem in Sects. 3, 4 further designs and implements `Nebula` to adequately ration network bandwidth resources for DDNN training tasks, so as to speed up the performance of DDNN training jobs. Section 5 extensively evaluates the performance gains and runtime overhead of `Nebula`. We discuss our contribution in the context of related work in Sect. 6. Finally, we conclude this paper in Sect. 7.

2 Background and motivation

In this section, we first analyze the DDNN training performance variation caused by the network resource contention in the *co-located* PS configuration. We then present an illustrative example to show how to speed up DDNN training performance simply by rationing adequate network bandwidth resources for PS and worker tasks.

2.1 Co-located PS configuration in DDNN training clusters

Training DNN models in distributed manner is becoming increasingly compelling, as the model gets complex and the training dataset becomes large. In general, each DDNN training job is comprised of two types of training tasks (i.e., PS tasks and worker tasks) running in the GPU cluster. For each training *iteration*, the PS tasks execute `collection` and `broadcast` operations, which collect the gradient data from worker tasks and send the updated model parameters

Table 1 DDNN training performance with different co-located PS configurations in a 16-node GPU cluster

Co-located	DDNN training rate (samples/s)		
PS number	ResNet50	ResNet152	VGG16
8	32.2	16.6	7.0
12	37.1	12.7	9.2
16	49.9	24.1	11.8

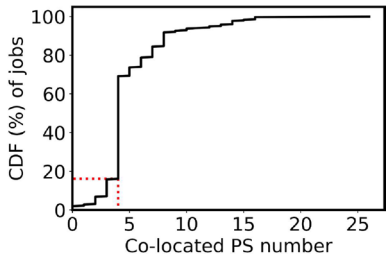

Fig. 1 CDF of the co-located PS number for DDNN training jobs in a Microsoft GPU cluster trace

to worker tasks, respectively. Correspondingly, the worker tasks execute `push` and `pull` operations, which send the gradient data to PS tasks and receive the updated model parameters from PS tasks, respectively. To reduce the data movement and achieve good DDNN training performance and scalability, the *co-located* PS configuration (i.e., co-locating PS and worker tasks on the same machine) is the default in many DDNN training frameworks (e.g., MXNet). In more detail, each machine hosts a PS task and multiple worker tasks, and thus the PS task shares network and computation resources with the worker tasks in the co-located PS configuration.

To understand the DDNN training performance benefits with the co-located PS configuration, we conduct a real-world experiment by training three traditional image-classification DNN models in a 16-node GPU cluster. As shown in Table 1, the DDNN training rate is steadily increased as we add more co-located PS tasks to DDNN training jobs, by varying the co-located PS number from 8 to 16. The rationale is that, the co-located PS task can make full use of CPU and intra-machine bandwidth resources to reduce the data movement over the network. Accordingly, each GPU device hosts a worker task and the CPU processors host the PS task on one GPU machine.

To validate the prevalence of the co-located PS configuration, we further *estimate* the number of co-located PS by analyzing a DDNN training job trace from a Microsoft GPU cluster (Jeon et al. 2019). As shown in Fig. 1, we observe that over 80% of jobs are configured with four or more co-located PS (with workers). Furthermore, we infer that the *co-location* of PS and worker tasks occurs when the CPU

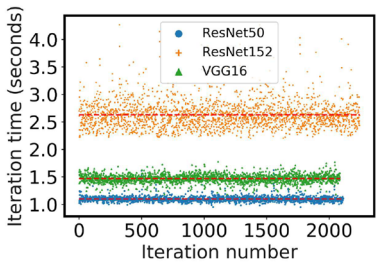

Fig. 2 Distribution of the iteration time for training representative DNN models

utilization of the machine exceeds 50% (Jeon et al. 2019). We find that the co-located PS configuration is deployed on around 77% of the machines in the Microsoft GPU cluster. Our analysis above on the Microsoft cluster trace demonstrates that the *co-located PS configuration is not uncommon* in production DDNN training clusters.

2.2 Understanding performance variation caused by network resource contention

Though the co-located PS configuration has been widely adopted in production DDNN training clusters, it inevitably brings severe *performance variation* to DDNN training jobs. To illustrate that, we conduct another motivation experiment by training ResNet50, ResNet152, and VGG16 on a 2-node cluster with the co-located PS configuration (i.e., each machine hosting one PS task and one worker task). As shown in Fig. 2, we observe that the iteration time[1] for DDNN training *fluctuates wildly*, and the coefficient of variation (CV) of DDNN training iteration time for the three models is around 0.1, which is 3–5 times larger than the CV obtained in the non-colocated PS configuration. Accordingly, we conjecture that such a severe performance variation is mainly caused by the network resource contention between the co-located PS and worker tasks.

To understand the performance variation of DDNN training, we further analyze the network communication mechanism of MXNet to show how the network resource contention of PS and worker tasks occurs. In particular, we neglect the intra-machine data transfer operations and only focus on the operations of PS and worker tasks consuming network resources. As depicted in Fig. 3, we observe that the PS task completes the `collection` operation for its gradient data *earlier* than the `push` operation of the worker task (pushing gradient data to PS tasks on other machines), which inevitably leads to the `broadcast` operation and the

[1] We consider the iteration time as the difference of end time of `pull` operations for two adjacent iterations (Zhang et al. 2017).

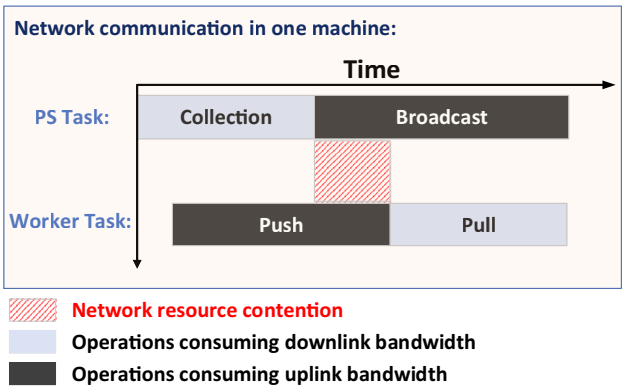

Fig. 3 Network resource contention of PS and worker tasks in one machine with the co-located PS configuration

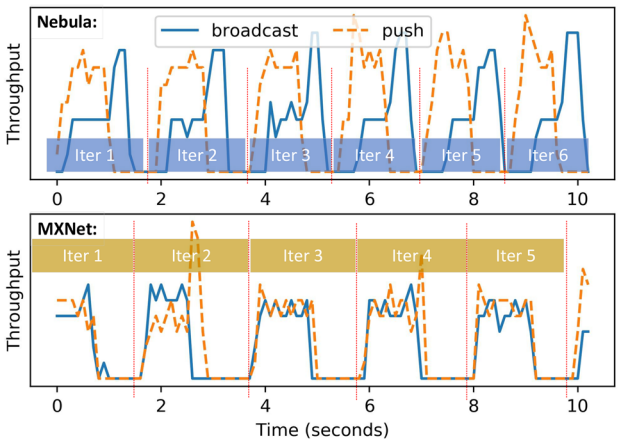

Fig. 4 Comparison of network throughput achieved by `Nebula` and MXNet during training ResNet50

`push` operation contending for the uplink bandwidth of the machine for a certain period of time.

We then summarize two conditions that can cause network resource contention of co-located PS and worker tasks as follows.

(a) `Broadcast` starts during the process of `push`.
(b) `Pull` begins during the execution of `collection`.

In general, the conditions (a) and (b) occur within one training iteration, and Fig. 3 belongs to the condition (a) above. As the iteration time is mainly determined by the `push` and `pull` operations of the worker task, such network resource contention inevitably *prolongs* the DDNN training time and brings severe variation to the iteration time of worker tasks *within one iteration* and *across iterations*. In addition, the overall network resource utilization and GPU utilization are correspondingly decreased. We will formally analyze such network resource contention in Sect. 3.1.

As a result, we focus on mitigating the performance variation of DDNN training jobs caused by network resource contention, through adequately rationing network bandwidth resources for co-located PS and worker tasks. In particular, we do not consider the frameworks with a global barrier (e.g., Tensorflow (Abadi et al. 2016)) due to its strict limitation of parameter synchronization.

2.3 An illustrative example

With the aim of mitigating the network resource contention as discussed above, we propose a *simple yet effective* network bandwidth allocation strategy named `Nebula`, in order to ration network bandwidth resources for co-located PS and worker tasks. `Nebula` is able to deal with the network resource contention and regulate the network throughput to enhance the efficiency of network bandwidth resources, thereby speeding up DDNN training.

To illustrate the performance benefits obtained by `Nebula`, we conduct a motivation experiment by training ResNet50 on a 2-node GPU cluster, and each node is a g3.4xlarge EC2 instance. As observed in Fig. 4, MXNet can only train 5 iterations over 10 s, while `Nebula` can complete 6 iterations within 10 s. In more detail, `Nebula` can achieve a faster DDNN training rate of ResNet50 (with 19.6 samples/s) by 18.8% as compared with that of MXNet (with 16.5 samples/s). This is because `Nebula` *adequately* allocates network bandwidth resources for the `broadcast` operation (in PS tasks) and the `push` operation (in worker tasks), while MXNet allows the two operations contending for network bandwidth resources arbitrarily. Accordingly, `Nebula` achieves higher overall network resource utilization as compared with the MXNet, as depicted in Fig. 4, so that the DDNN training performance can be significantly improved by `Nebula`.

Summary The network resource contention of *co-located* PS and worker tasks inevitably brings severe performance variation to DDNN training jobs. Judiciously *rationing network bandwidth* resources for PS and worker tasks can alleviate such network bandwidth contention and significantly speed up the performance of DDNN training jobs.

3 Problem analysis and formulation

In this section, we first analyze the root cause of DDNN training performance variation. Then, we build an analytical model to formulate the completion time of each communication operation and the iteration time in a DDNN training job. The key notations of our DDNN training performance model are summarized in Table 2.

Table 2 Key notations of DDNN training performance model

Notation	Definition
$\mathcal{M}$	Set of machines in the GPU cluster
s	Size of the model parameters
B	Available uplink bandwidth in one machine
$\overline{B_p}, \overline{B_w}$	Average uplink bandwidth for the PS task during the transmission of `broadcast` and `collection`, and that for the worker task during the transmission of `push` and `pull`
$S^{op}_{m,i}$	Start time of one operation *op* in the i-th iteration on machine m
$E^{op}_{m,i}$	End time of one operation *op* in the i-th iteration on machine m
c^{op}_{m}	Communication data size of one operation *op* in machine m
$T_{m,i}$	The i-th iteration time on machine m
T_i	The i-th iteration time of a DDNN training job

3.1 Analyzing DDNN training performance variation

We consider a DDNN training job executed in a GPU cluster (denoted by $\mathcal{M}$) with the *co-located* PS configuration, and each machine contains one PS task and multiple worker tasks. One of the worker tasks serves as the *master* worker task (Luo et al. 2020), which takes charge of inter-machine communication over the network.

In fact, the root cause of network resource contention in the co-located PS configuration is that the model parameters are *unevenly* distributed among different PS. The rationale is that, such an *uneven* parameter distribution will definitely lead to different completion time of communication operations of PS and worker tasks, thereby resulting in the two conditions for network resource contention enumerated in Sect. 2.2. Such resource contention can *get severer and converge to a worst contention case* during the execution of a DDNN training job, which will be analyzed as follows.

We assume a case [i.e., condition (a) in Sect. 2.2] that the `pull` operation is longer than the `broadcast` operation by ϵ, and also the communication operations in each iteration i spend the same execution time, which can be formulated as

$$T = E^{op}_{m,i} - S^{op}_{m,i}, \quad op \in \{\texttt{push}, \texttt{pull}\}$$
$$T + \epsilon = E^{op}_{m,i} - S^{op}_{m,i}, \quad op \in \{\texttt{clt}, \texttt{bct}\}$$

where $S^{op}_{m,i}$ and $E^{op}_{m,i}$ denote the start and end time of four communication operations including `push` and `pull` for worker tasks, and `collection` and `broadcast` for PS tasks. As shown in Fig. 5, the `pull` and `collection` operations first contend for the network downlink bandwidth. Suppose PS and worker tasks equally share the bandwidth resource, such downlink bandwidth contention between `pull` and `collection` operations can last for 2ϵ. In the next iteration, the `broadcast` operation starts during the process of `push` operation (i.e., condition (a) in Sect. 2.2) and thus contends for the network uplink bandwidth. Such resource contention will last for 4ϵ. In such a way, each communication operation in the following iterations will spend more time to transfer the same amount of data.

The *worst case* of such resource contention discussed above is that the `push` operation fully overlaps with the `broadcast` operation. The rationale is that, the Bulk synchronous parallel (BSP) mechanism mandates the PS task to collect all the gradient data from `push` operations of worker tasks, before starting the *broadcast* operation. In such a worst case, the network resource contention will last for almost the total iteration time (i.e., $4T$), while the original iteration time without any network resource contention is $2T + 2\epsilon$. As a result, the network resource contention in co-located PS configuration results in the severe performance variation *within one iteration and across iterations* as illustrated in Sect. 2.2.

3.2 Modeling iteration time of DDNN training jobs

Before modeling the iteration time, we first analyze the start time and end time for four communication operations, which are `push` and `pull` for worker tasks, and `collection` and `broadcast` for PS tasks. In particular, we neglect the intra-machine communication time and the aggregation time

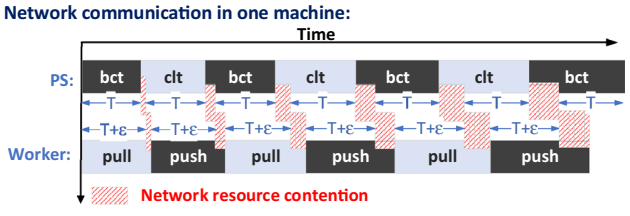

Fig. 5 Severity analysis of network resource contention in the co-located PS configuration. `bct` and `clt` denote the abbreviations for the `broadcast` and `collection` operations, respectively

of gradient data. As the *master* worker task *equally* coordinates network resources on a machine, the worker tasks have the same iteration time on the same machine.

As one iteration i starts from the ending of `pull` operation for iteration $i-1$, we formulate the *start time* of `push` and `broadcast` operations using the end time of iteration $i-1$. Specifically, the `push` operation cannot start until the worker has completed the backward propagation. The `broadcast` operation begins once the PS task has finished the `collection` operation, when the workers have `push`ed the gradient data to the PS task. Accordingly, the start time of `push` operation $S_{m,i}^{push}$ and `broadcast` operation $S_{m,i}^{bct}$ is given by

$$\begin{aligned} S_{m,i}^{push} &= E_{m,i-1}^{pull} + \tau_{m,i}, \\ S_{m,i}^{bct} &= \max_{n\in\mathcal{M}, n\neq m} \left(S_{n,i}^{push} + \frac{c_n^{push}}{\overline{B_w}} \right), \end{aligned} \tag{1}$$

where $E_{m,i-1}^{pull}$ denotes the end time of `pull` operation in iteration $i-1$ on machine m, and $\tau_{m,i}$ denotes the calculation time of forward propagation and backward propagation. c_n^{push} denotes the communication data size of `push` operation on machine n, and $\overline{B_w} \in (0, B]$ denotes the average network uplink bandwidth allocated to the worker task on one machine during the process of `push` operation. Accordingly, $\frac{c_n^{push}}{\overline{B_w}}$ denotes the communication time of `push` operation on machine n.

To obtain the *end time* $E_{m,i}^{pull}$ of `pull` operation in iteration i on machine m, we proceed to formulate the end time of `broadcast` operations, as $E_{m,i}^{pull}$ is determined by the slowest `broadcast` operations on the other machines n (i.e., $n \neq m$), which is given by

$$\begin{aligned} E_{m,i}^{pull} &= \max_{n\in\mathcal{M}, n\neq m} E_{n,i}^{bct}, \\ E_{m,i}^{bct} &= S_{m,i}^{bct} + \frac{c_m^{bct}}{\overline{B_p}}, \end{aligned} \tag{2}$$

where $E_{n,i}^{bct}$ denotes the end time of `broadcast` operation in iteration i on machine n. c_m^{bct} denotes the communication data size of `broadcast` operation on machine m, and $\overline{B_p} \in (0, B]$ denotes the network uplink bandwidth allocated to the PS task on one machine during the process of `broadcast` operation. Accordingly, $\frac{c_m^{bct}}{\overline{B_p}}$ denotes the communication time of `broadcast` operation on machine m.

By substituting Eqs. (1) into (2), we can obtain the end time $E_{m,i}^{pull}$ of `pull` operation in iteration i in terms of the end time of `pull` operation in the last iteration $i-1$ and uplink bandwidth for PS and worker tasks, which is given by

$$\begin{aligned} E_{m,i}^{pull} &= \max_{n,q} \left(\frac{c_n^{bct}}{\overline{B_p}} + \frac{c_q^{push}}{\overline{B_w}} + E_{q,i-1}^{pull} + \tau_{q,i} \right), \\ &\forall n, q \in \mathcal{M}, n \neq m, q \neq n \end{aligned} \tag{3}$$

where the first term of Eq. (3) denotes the `broadcast` time on machine n, and the second term denotes the communication time of `push` operation on machine q. The third term and the fourth term denote the end time of `pull` operation in iteration $i-1$ and the calculation time in iteration i on machine q, respectively.

To speed up DDNN training process and mitigate its performance variation, our objective turns out to be minimizing each iteration time for DDNN training jobs (i.e., the training time of the i-th iteration given the end time of the last iteration $i-1$). In particular, the DDNN training time T_i for iteration i is determined by the machine that has the longest execution time (i.e., the largest $T_{m,i}, \forall m \in \mathcal{M}$). Accordingly, our optimization problem can be formulated in Eq. (4) as below.

$$\begin{aligned} \min_{\overline{B_p}, \overline{B_w}} T_i &= \min_{\overline{B_p}, \overline{B_w}} \left(\max_{m\in\mathcal{M}} T_{m,i} \right) \\ &= \min_{\overline{B_p}, \overline{B_w}} \left(\max_{m\in\mathcal{M}} \left(E_{m,i}^{pull} - E_{m,i-1}^{pull} \right) \right) \\ &= \min_{\overline{B_p}, \overline{B_w}} \left(\max_{m,n,q\in\mathcal{M}} \left(\frac{c_n^{bct}}{\overline{B_p}} + \frac{c_q^{push}}{\overline{B_w}} + A \right) \right) \end{aligned} \tag{4}$$

$$\text{s.t. } \overline{B_p}, \overline{B_w} \in (0, B], \tag{5}$$

$$n \neq m, q \neq n, \quad \forall m, n, q \in \mathcal{M}, \tag{6}$$

where $A = \tau_{q,i} + E_{q,i-1}^{pull} - E_{m,i-1}^{pull}$ denotes a value that depends on the end time of the iteration $i-1$ and the calculation time of iteration i. The first two terms denote the data communication time on the slowest machines, which can be significantly influenced by the network resource contention during DDNN training. Constraint (5) indicates that the average uplink bandwidth which one task can consume should be a positive value and less than the available uplink bandwidth B. Both $\overline{B_p}$ and $\overline{B_w}$ can be equal to B if there is *no network resource contention* during the data communication of DDNN training. Constraint (6) implies that the cluster has two machines at least and m can be equal to q under the circumstance.

Though our optimization problem can be formulated in a closed-form expression, Eq. (4) is still in *the form of a min-max optimization problem* (Russell and Norvig 2020). Our optimization problem is not continuous and indifferentiable. Accordingly, our bandwidth allocation problem cannot be solved by a traditional gradient optimization approach, and thus we turn to designing a bandwidth allocation heuristic in Sect. 4 to solve such a performance optimization problem.

4 Rationing network bandwidth resources for DDNN training tasks

Based on our problem analysis and formulation above, we proceed to design Nebula, a network bandwidth allocation strategy for co-located PS and worker tasks in order to mitigate network resource contention.

4.1 Design of network bandwidth allocation strategy

We design Nebula strategy in Algorithm 1 by following a quite *simple and intuitive* heuristic: We leverage the *task weight* to ration adequate network bandwidth resources to the co-located PS and worker tasks once the network resource contention is severe. Nebula aims to answer *when and how* to allocate the network bandwidth for the co-located PS and worker tasks. In particular, Nebula mitigates the resource contention by focusing on the network *uplink* bandwidth, and accordingly the resource contention on network *downlink* bandwidth will also be alleviated.

Algorithm 1 Nebula: Network bandwidth resource allocation strategy for co-located PS and worker tasks.

Input: Available bandwidth of one machine B, the network bandwidth allocation coefficient θ, and the weight threshold w_0.

1: **Initialize:** A map storing the classification of communication data (including model parameters and gradient data): $\texttt{mp} \leftarrow \langle \texttt{null}, \texttt{null} \rangle$; Weights of PS and worker tasks for network uplink bandwidth: $w_w \leftarrow 0$, $w_p \leftarrow 0$; Flag of network bandwidth allocation for PS tasks and worker tasks: $flag \leftarrow 0$;
2: Obtain the communication data information through profiling of DDNN training jobs;
3: Classify the communication data according to the start time of data transfer, and store the *layer index of communication data* and its *class* into mp;
4: **while** one communication data with the layer index i starts or finishes its network transfer **do**
5: **if** the data i belongs to the worker task **then**
6: $w_w \leftarrow w_w \pm e^{-\texttt{mp}[i]}$;
7: **else**
8: $w_p \leftarrow w_p \pm e^{-\texttt{mp}[i]}$;
9: **end if**
10: **if** $w_p - w_w \geq w_0$ && $flag \neq 1$ **then**
11: Cancel the bandwidth limit of broadcast operation and limit push operation bandwidth by $B \cdot \theta$;
12: $flag \leftarrow 1$;
13: **else if** $w_w - w_p \geq w_0$ && $flag \neq 2$ **then**
14: Cancel the bandwidth limit of push operation, and limit broadcast operation by $B \cdot \theta$;
15: $flag \leftarrow 2$;
16: **end if**
17: **end while**

Specifically, Nebula has three input parameters including the available network bandwidth B of one machine, the bandwidth allocation coefficient $\theta \in (0, 1]$, and a weight threshold w_0 for deciding *when* to allocate network bandwidth for PS and worker tasks. In particular, $B \cdot \theta$ indicates the value to limit (i.e., *how* to allocate) network bandwidth for the PS or worker task during DDNN training. Before the execution of a DDNN training job, we first initialize a map data structure $\texttt{mp} = \langle \texttt{data}, \texttt{class} \rangle$ to store the class information of each communication data (i.e., model parameters and gradient data), where data denotes the layer index of the communication data. We also initialize the weights for PS and worker tasks (i.e., the *task weight* is the sum of weights of communication data packets carried by the task), as well as a flag that indicates whether the network bandwidth is appropriately allocated on the PS or worker tasks (line 1). We then leverage job profiling (e.g., within 10 iterations) to obtain the communication data information including the layer index of communication data and the start time of data transfer. According to the transfer start time of communication data, we classify the communication data into several classes (e.g., 10 for ResNet50) by a simple clustering algorithm (lines 2–3). After that, we proceed to monitor the status of PS and worker tasks, and compare the two task weights (i.e., w_w, w_p) to adequately ration network bandwidth resources for the co-located PS and worker tasks in each iteration.

In more detail, once a communication data with the layer index i starts or finishes its network transfer, we have to add or delete the weight of the communication data from the corresponding task weight. In particular, we empirically use an exponential function to calculate the weight of each communication data, by the intuition that the communication data with a small layer index has a higher transfer priority compared with the data with a larger layer index (lines 4–9). The rationale is that, the communication data generated by the front layer (e.g., the layer 0) is eagerly required by the pull operation of the next training iteration (Jayarajan et al. 2019). After task weights have been updated, Nebula further decides *whether* to enforce bandwidth allocation for tasks. If w_p is larger than w_w by exceeding an empirical threshold (i.e., $w_0 = 1$ by default), we limit the bandwidth of the push operation by $B \cdot \theta$ and cancel the bandwidth limit of broadcast. We also set the bandwidth allocation flag as 1, which indicates that the bandwidth of push operation has been limited (lines 10–12). Similarly, if w_w is larger than w_p by exceeding w_0, we limit the bandwidth of the broadcast operation by $B \cdot \theta$ and cancel the bandwidth limit of push, and set the flag as 2 which indicates that the bandwidth of broadcast operation has been limited (lines 13–16).

Remark Nebula adopts a bandwidth allocation coefficient $\theta \in (0, 1]$ to control the network bandwidth for PS and worker tasks. The value of θ will definitely affect the performance of DDNN training jobs. For simplicity, we only

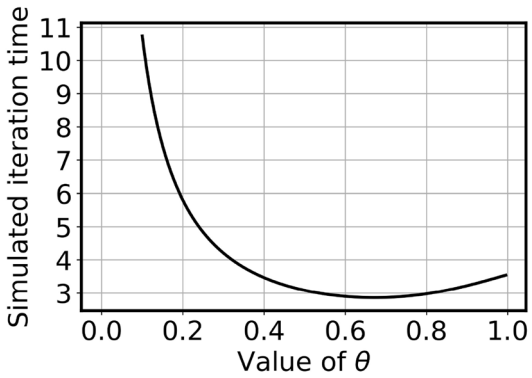

Fig. 6 Simulated relationship between the bandwidth allocation coefficient $\theta \in (0, 1]$ and DDNN training iteration time

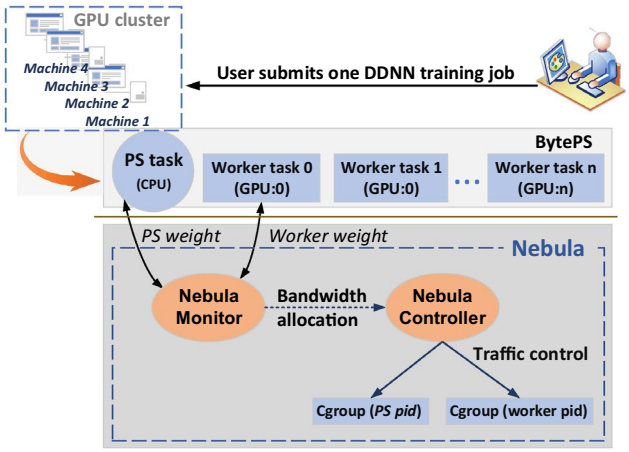

Fig. 7 `Nebula` implementation based on BytePS

consider the factors related to the network bandwidth $\overline{B_p}, \overline{B_w}$ in Eq. (4). By assuming $c_q^{push} = c_m^{bct} = c$, the optimization problem in Eq. (4) can be reduced to $\min(\frac{c}{\overline{B_w}} + \frac{c}{\overline{B_p}}) = c \cdot \min(\frac{1}{\overline{B_w}} + \frac{1}{\overline{B_p}})$, where $\overline{B_p}$ and $\overline{B_w}$ can be equal to B if no network contention exists. Therefore, the sum of $\overline{B_w}$ and $\overline{B_p}$ can represent the degree of network resource contention between PS and worker tasks. The larger value their sum is, the severer this contention is. As the network resource contention exists in most cases as evidenced in Sect. 2.2, we simply assume $(\overline{B_w} + \overline{B_p}) = X \sim \mathcal{N}(\mu, \delta^2)$ which distributes in the interval $[B, 2B]$. By setting $\overline{B_w}$ and $\overline{B_p}$ as $B \cdot \theta$ when the network bandwidth is appropriately allocated, the optimization problem can be simplified as minimizing the expectation of the function below.

$$E\left(\frac{1}{\theta} + \frac{1}{\frac{X}{B} - \theta}\right), \tag{7}$$

where $X \sim \mathcal{N}(1.5B, 0.25B)$. As depicted in Fig. 6, we observe that the general trend of the iteration time is to first decrease and then increase as the value of θ increases. Accordingly, we empirically set the value of θ as 0.4 by default, as it can achieve good DDNN training performance in most cases. We will validate our analysis of θ above in Sect. 5.2.

4.2 Implementation of Nebula

We implement a prototype of our `Nebula` based on BytePS[2], as BytePS is a generic communication scheduler for the mainstream DDNN training frameworks (e.g., MXNet, TensorFlow). Specifically, our prototype of `Nebula` is implemented upon BytePS v0.2.4 with over 200 lines of C++ and Linux Shell codes. As shown in Fig. 7, `Nebula` is executed on each machine, which includes two modules elaborated as follows.

`Nebula` *monitor* `Nebula` monitor is actually the implementation of Algorithm 1. We process the *json* file generated by job profiling and obtain the layer index and start time of communication data. In a DDNN training job, data communication occurs frequently and can be classified into several types. To adapt to network resource contention, `Nebula` embeds the monitor inside the DDNN training framework, and invokes a trigger once one communication data starts or finishes its network transfer. Such a trigger further accesses the weights of PS tasks and worker tasks, and then `Nebula` leverages the communication data information through profiling to update the task weights. With the updated task weights, `Nebula` is able to decide *when and how* to allocate network bandwidth for PS and worker tasks.

`Nebula` *controller* `Nebula` controller leverages the `tc` tool and `cgroup` to limit the network bandwidth of one specific process by automatically captured its `pid`. The corresponding `cgroup` and `tc` files are created as long as `pid` is obtained with the launch of PS and worker tasks. Once the task weights are changed and the network bandwidth limits are required to be enforced, a Shell script with two bandwidth values will be generated to modify the `cgroup` and `tc` files created before, so that the network bandwidth of PS and worker tasks can be limited accordingly.

`Nebula` offers two main advantages over the existing mainstream DDNN training frameworks as follows. (1) `Nebula` is the first to deal with the *performance variation* problem caused by network resource contention, through exploiting the characteristics of data communication (i.e., task weights, communication data classification) in DDNN training. (2) `Nebula` is a *framework-independent* tool to mitigate network resource contention by the adequate network bandwidth allocation for co-located PS and worker tasks.

[2] https://github.com/bytedance/byteps.

`Nebula` *discussion* We discuss a practical issue related to the implementation of `Nebula`, which is *how to control the network communication with low overhead*. Many recent works (e.g., P3 (Jayarajan et al. 2019), Geryon (Wang et al. 2020c)) focus on network congestion and resource contention during DDNN training. They generally involve several modifications in the transport layer and network layer. Compared with these works, `Nebula` exploits the `tc` tool to configure the traffic control in the Linux kernel with low overhead, which will be validated in Sect. 5.3. Nevertheless, as the model becomes more complex, the communication conditions will change more rapidly during DDNN training. To respond to such rapid network changes, frequently rationing network bandwidth will inevitably bring critical network fluctuations and impact DDNN training performance. To ease such network fluctuations in the environment of rapid network changes, we plan to leverage the Software Defined Network (SDN) techniques to implement `Nebula` by allocating network bandwidth for co-located PS and worker tasks of DDNN training jobs.

5 Performance evaluation

In this section, we evaluate the effectiveness and runtime overhead of `Nebula` by carrying out a set of real-world experiments trained with four representative DNN models on Amazon EC2.

5.1 Experimental setup

Cluster configurations We set up a DDNN training cluster on Amazon EC2 using up to 8 g3.8xlarge instances. Each EC2 instance is equipped with 2 Tesla M60 GPUs, 32 vCPUs, 244 GB SSD, and the network bandwidth ranging from 1 to 5 Gbps. Specifically, the network bandwidth of instances is set as 1 Gbps when training AlexNet, ResNet50, and ResNet101. The network bandwidth of instances is set as 4 Gbps when training VGG16 due to its large model parameter size. We launch two worker tasks on two GPUs and one PS task on the CPUs on each EC2 instance.

Workloads and datasets We choose four representative DNN models which are AlexNet, ResNet50, ResNet101, and VGG16. We select AlexNet because it is considered as one of the most influential DNN models in Computer Vision. VGG is a classical DNN model due to its large model parameter size, and ResNet is adopted as the de-facto standard for image classification. We use ImageNet 2012 (i.e., ILSVRC2012_img_val) as our training dataset.

Baseline We compare `Nebula` with BytePS (Jiang et al. 2020) on MXNet (Chen et al. 2015) as `Nebula` is implemented based on BytePS (discussed in Sect. 4.2). For simplicity, we denote the baseline (i.e., BytePS on MXNet) as MXNet henceforth, and we configure the parameters in BytePS as default values and restrict the size of each communication data packet less than 4 MB.

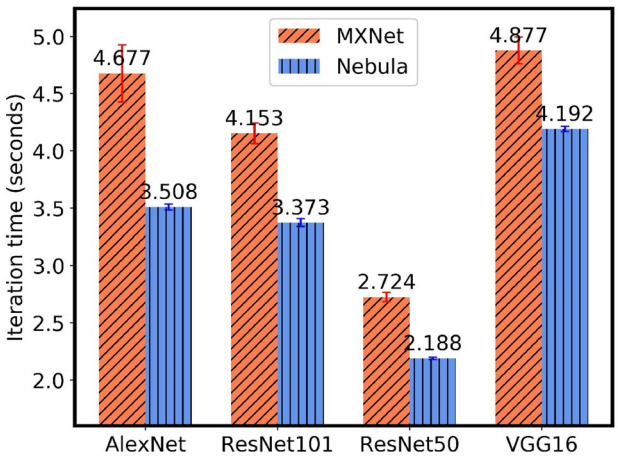

Fig. 8 Average iteration execution time of four representative models trained with MXNet and `Nebula`

Metrics We illustrate the effectiveness of `Nebula` using three important metrics: the iteration execution time, the variance of iteration completion time, and the GPU and network resource utilization of EC2 instances. In particular, we record the iteration completion time by monitoring the training progress of worker tasks on each instance and calculating the variance of the completion time among all instances *within one iteration*. The variance of iteration completion time indicates the performance variation of the DDNN training workload.

5.2 Effectiveness of Nebula

Iteration execution time Figure 8 compares the execution time when training four representative DNN models with `Nebula` and MXNet. Each bar denotes the average iteration time of the first 100 iterations of all worker tasks. Moreover, we repeat the experiments for each DNN model for *five* times and illustrate the average iteration time with error bars of standard deviations. Specifically, we observe that `Nebula` shortens the average iteration completion time by 15.2–25.0% as compared to MXNet. The performance improvement with `Nebula` is significant in training AlexNet (i.e., 25.0%). As a classical DNN model, AlexNet training can be accelerated by parallel computing significantly (Krizhevsky et al. 2012) and will not be blocked by several special modules in DNN models (e.g., shortcuts). Accordingly, short computation time and over 60 MB model parameters lead to highly intense network resource

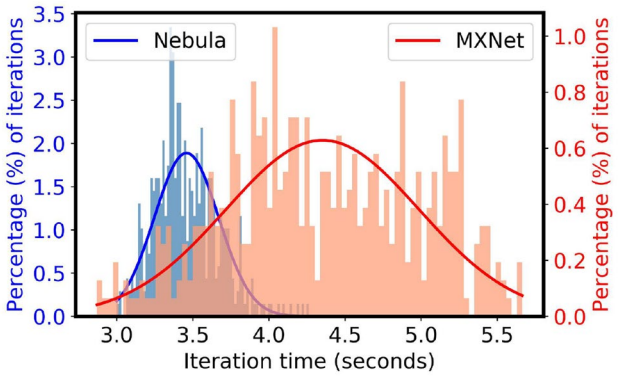

Fig. 9 Distribution of iteration execution time *across iterations* when training ResNet101 model with MXNet and Nebula

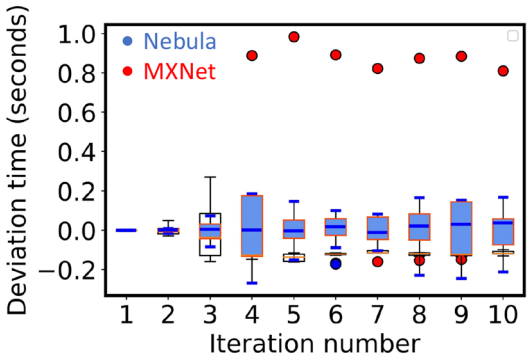

Fig. 10 Iteration completion time of all workers *within one iteration* when training AlexNet model with MXNet and Nebula

contention when training AlexNet, which can be effectively alleviated by our Nebula bandwidth allocation strategy.

Furthermore, we observe that the performance variation of iteration execution time (*across iterations*) with MXNet are greater than that with Nebula. The rationale is that Nebula alleviates the network contention and allocates appropriate network bandwidth to the co-located PS and worker tasks. Figure 9 compares the distribution of the iteration execution time during training ResNet101 with Nebula and MXNet. We observe that Nebula keeps the iteration execution time in an ideal time interval and thus accelerates the DNN training process. The average iteration time with Nebula is 3.45 s, while the average iteration time with MXNet is 4.35 s. Moreover, the standard deviation with Nebula is only 0.21 s, which is much lower than 0.63 s with MXNet. In sum, Nebula is able to reduce the performance variation of DDNN training across iterations as compared with MXNet.

Variations of iteration completion time We proceed to look into the DDNN training process on each EC2 instance by taking AlexNet as an example. We record the iteration completion time of each instance *within one iteration* in order to seek the root cause of the performance improvement with Nebula. Specifically, we train AlexNet with 8 g3.8xlarge EC2 instances, and we calculate the average completion time in each iteration and the time deviation across instances. Fig. 10 shows the deviations of the 8 instances from the average iteration time for the first 10 iterations. We observe that the time deviation with Nebula is stable within a small time interval (i.e., 0.2 s), while the outliers are scattered over seconds with MXNet. Accordingly, Nebula achieves the stable iteration execution time across EC2 instances within one iteration, as compared with MXNet.

Moreover, we examine the iteration completion time of all the workers (i.e., EC2 instances) when training different DNN models with 4 g3.8xlarge instances. Fig. 11 depicts the CDF of time variance values of all workers *within one iteration* for each DNN model. The variance of AlexNet and ResNet101 can be mitigated by Nebula significantly and there is relatively small performance improvement for VGG16, which is consistent with our experiment results in Fig. 8. Specifically, Nebula maintains the time variance less than 0.4 s for over 95% of iterations when training ResNet101, but the maximum time variance is still large. This is because many small parameters exist in ResNet101 which can cause the network fluctuations. As for the VGG16 model, the time variance is much greater as the communication time is over 1 second with MXNet. Nebula can reduce such a time variance down to 0.5 s for VGG16. In sum, Nebula can reduce the time variance by up to 65.2% (e.g., from 0.58 to 0.20 s in ResNet101), as compared with MXNet.

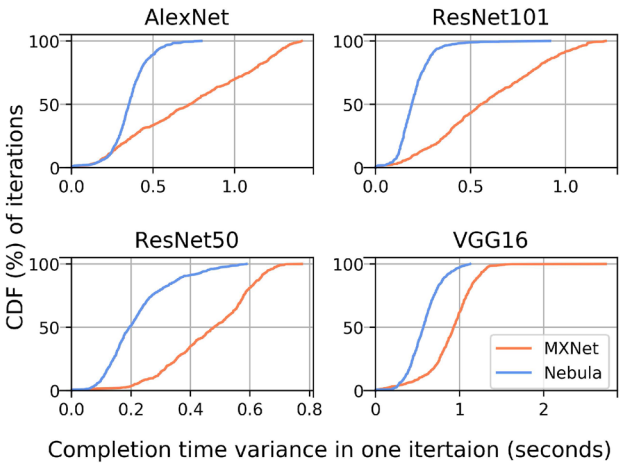

Fig. 11 CDF of the iteration completion time variance of all workers *within one iteration* when training different DNN models with MXNet and Nebula

GPU and network resource utilization To examine whether Nebula can improve the resource utilization, we

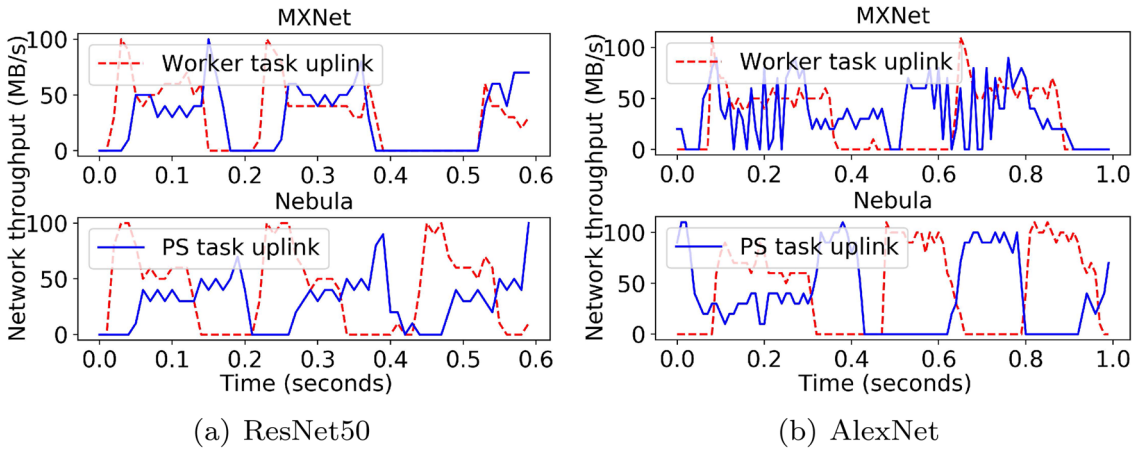

Fig. 12 Comparison of network throughput over time of when training **a** ResNet50 and **b** AlexNet models with MXNet and `Nebula`

further take a closer look at the network throughput during the training of AlexNet and ResNet50. Fig. 12 shows the network uplink throughput of the PS and worker tasks co-located in one instance. Specifically, the network resource contention is intense when training ResNet50 with MXNet as shown in Fig. 12a. The worker task occupies more network bandwidth in the first iteration, and the next iteration starts without too much delay. The PS task occupies more network uplink bandwidth in the second iteration, and we can observe there is an apparent delay that lasts over 100 ms before the third iteration starts. This is because such network resource contention leads to a long completion time of `push` and thus postpones the `pull` operation further. With `Nebula`, such intense network contention is alleviated because of our reasonable rationing strategy. As shown in Fig. 12b, the contention for the uplink bandwidth between the co-located PS and worker tasks for AlexNet are severer than that for ResNet50 in each iteration. Our `Nebula` bandwidth allocation strategy alleviates the network resource contention among PS and worker tasks. We observe that `Nebula` completes almost three iterations while MXNet only completes two iterations within one second.

In addition, we calculate the average value and the standard deviation of the network throughput and GPU utilization of a worker node during the execution of the four DNN models, as summarized in Table 3. `Nebula` achieves both higher and more stable GPU utilization and network throughput in comparison to MXNet. FOr instance, `Nebula` improves the average network throughput by up to 30% (e.g., from 63.6 to 82.5 MB/s for AlexNet). Similarly,`Nebula` also increases the average GPU utilization by 15.4–26.7% (e.g., from 17.7 to 21.5% for ResNet50). As a result, the small deviation of resource utilization further validates the low variance of iteration execution time achieved by our `Nebula` bandwidth allocation strategy.

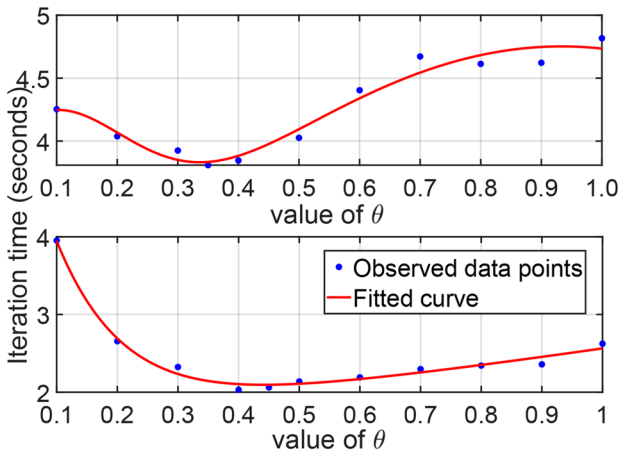

Fig. 13 Observed and fitted iteration execution time of AlexNet (the upper) and ResNet50 (the lower) model training by varying different values of θ from 0.1 to 1

Table 3 The #average (#standard deviation) of network throughput and GPU utilization of a machine when training different DNN models with MXNet and `Nebula`

	Throughput (MB/s)		GPU (%)	
	MXNet	`Nebula`	MXNet	`Nebula`
AlexNet	63.6 (3.5)	82.5 (0.6)	2.2 (0.10)	2.9 (0.03)
ResNet101	61.2 (1.0)	74.3 (0.9)	18.1 (0.52)	21.8 (0.30)
ResNet50	53.1 (0.8)	65.1 (0.5)	17.7 (0.20)	21.5 (0.15)
VGG16	164.2 (3.9)	189.2 (1.6)	15.6 (0.37)	18.0 (0.13)

Sensitivity of θ: In our experiments above, we simply set θ as a constant value 0.4 by default. The value of θ will impact the training performance according to our analysis in Sect. 4.1. To examine the DDNN training performance

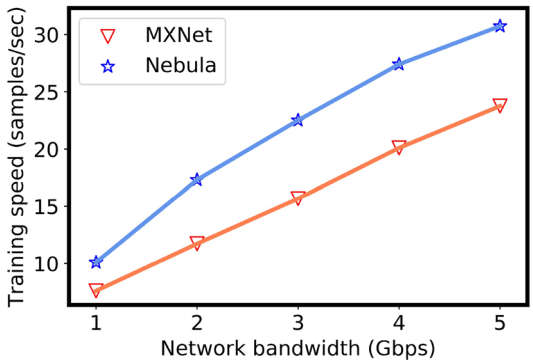

Fig. 14 Comparison of DDNN training performance of AlexNet with various available network bandwidth ranging from 1 Gbps to 5 Gbps for MXNet and `Nebula`

with different values of θ, we alter θ as 11 sample values (i.e., 0.1, 0.2, 0.3, 0.35, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1) when training ResNet50 and AlexNet. We then fit these 11 sample points with a Gaussian function, as the expected iteration time is associated with a Gaussian function in the mathematical expression. As shown in Fig. 13, the fitted curve on the model ResNet50 (the lower one) is accordant with our simulation curve of θ in Fig. 6 (Sect. 4.1), while the fitted curve on AlexNet is basically consistent with our simulated curve. We observe that the best value of θ is located in the interval from 0.3 to 0.6. The sensitivity of θ obtained on other DNN models shows the similar characteristics. 0.4 is an appropriate value of θ because it can effectively alleviate the network resource contention in most cases. Even 0.4 does not perform very well on several DNN models, we can manually tune it online without incurring much performance overhead.

Different available network bandwidth To obtain complementary insights and verify the effectiveness of `Nebula`, we conduct another experiment to evaluate the DDNN training performance under different available network bandwidth for training machines. Specifically, we train AlexNet with different available bandwidth ranging from 1 Gbps to 5 Gbps. As shown in Fig. 14, the DDNN training speed with `Nebula` consistently outperforms that with MXNet. The performance improvement (i.e., 29.6–47.6%) varies as the network bandwidth increases. In more detail, the training speedup can reach up to 47.6% when the available network bandwidth is 2 Gbps, while the training speed is only improved by around 30% with the bandwidth setting as 1 Gbps and 5 Gbps. Accordingly, our analysis above indicates that `Nebula` can achieve better performance gain when the available network bandwidth resource is stringent (i.e., $B = 1$ – 5 Gbps as the network contention exists), while the DDNN training performance with `Nebula` gradually converges to that with MXNet when the available network bandwidth resource is sufficient (i.e., $B = 5$–10 Gbps as the network contention is mitigated) in our experimental setup.

5.3 Runtime overhead of Nebula

We *first* estimate the computation overhead of `Nebula` during DDNN training. According to Algorithm 1, the computation time of `Nebula` can be determined by the number of communication data packets which is denoted as num_p, and it is mainly associated with the DNN model size and the cluster scale. Even for a large DNN model like VGG16, the computation complexity of `Nebula` is within $\mathcal{O}(num_p)$, where $num_p = 10^5$. *Second,* `Nebula` controller allocates the network bandwidth for PS and worker tasks, which impacts the network performance during DDNN training. Through analyzing the log file of `Nebula`, we find that the network bandwidth of PS and worker tasks is enforced for less than four times within one iteration, and the network bandwidth of PS and worker tasks is limited twice in most cases. Also, we can adjust the weight threshold w_0 (i.e., 1 by default) to control the number of bandwidth limitations enforced for PS and worker tasks. *Finally,* we examine the profiling overhead of DDNN training workloads in `Nebula`. The job profiling time equals to the DDNN training time for the first 5 iterations, which are 17.5, 16.6, 11.0, and 20.5 s for AlexNet, ResNet101, ResNet50, and VGG16, respectively, in our experiment. In general, the DDNN training workloads are periodically executed and contain thousands of training iterations. As a result, the overall runtime overhead of `Nebula` above is acceptable in practice.

6 Related work

6.1 Mitigating network resource contention in DDNN training

There have been several works on mitigating network contention in DDNN training. The contention among the co-located tasks is actually *the competition of network resources on one GPU machine*. It commonly occurs in the multi-job (e.g., multi-tenant) environments and the resource contention will prolong the whole training process when sharing network resources. To address such an issue, MLNet (Mai et al. 2015) has proposed the network prioritization to arbitrate the access to network resources. A more recent work (Wang et al. 2020b) proposes a communication contention-aware scheduling algorithm (Ada-SRSF) based on the execution of DDL jobs in the form of Directed Acyclic Graphs (DAGs). For the cloud-based environment, PLink (Luo et al. 2020) can efficiently generate aggregation plans for DDNN training jobs to adapt to the changing network conditions. While these works above require modifications to the DNN

training framework, Nebula is actually a supplementary tool of network bandwidth allocator, which does not require any changes to the training framework. Similarly, Tensorlights (Huang et al. 2019) is an end-host traffic scheduler to mitigate the inter-job network contention. Compared with its time slice rotation strategy to allocate network resources among jobs, Nebula applies a quota-based allocation strategy and adjusts it with the change of network conditions and mitigates the intra-job network resource contention.

Moreover, many previous works are dedicated to dealing with the network bottleneck of PS, which is caused by *the competition of network resources on PS among worker tasks*. For instance, Poseidon (Zhang et al. 2017) employs a decentralized architecture to transfer large layers to alleviate the PS network bottleneck. R2SP (Chen et al. 2019) focuses on how to share the PS bandwidth resources by time division multiplexing and thus to fully utilize the network resources of PS. While such PS network bottleneck can be mitigated in the co-located PS configuration, Nebula allocates the network bandwidth dynamically for PS and worker tasks during the training process and aims to fully utilize the network resources of GPU machines. In addition, Optimus (Peng et al. 2018) adequately adjusts the number and placement of workers and PS online, which can mitigate the network resource contention from the aspect of PS and worker configurations. A more recent work (Berral et al. 2020) employs a combination of two machine learning algorithms to explore behaviors/patterns of containers and allocate CPU/memory resources dynamically, but the network resource is ignored. These prior works above are mainly deployed with the non-co-located PS configuration. In contrast, Nebula focuses on the common DDNN training environment (i.e., in the widely-used co-located PS configuration) and solving the *intra-job* network resource contention.

6.2 Communication scheduling of DDNN training

To optimize the DDNN training performance through network communication, a number of works are devoted to overlapping the computation and communication by designing advanced *communication scheduling* strategies. These scheduling strategies are mainly to answer *what granularity the communication data packets belong to* and *when to transfer the communication data packets*.

To answer the first question, i.e., *what granularity the communication data packets belong to*, Poseidon (Zhang et al. 2017) sets the gradient data of one layer as the unit of communication data, while P3 (Jayarajan et al. 2019) partitions a large layer into several small parts to further increase the performance benefits of overlapping. Similarly, our implementation of Nebula is based on the layer slicing operations to obtain such performance benefits of overlapping. For the All-Reduce training architecture, MG-WFBP (Shi et al. 2019) merges the layer data to decrease the communication overhead, and a more recent work (Shi et al. 2020) focuses on solving a trade-off optimization problem to answer whether to merge gradient data or not in sparsification.

As for the second question, i.e., *when to transfer the communication data packets*, Poseidon (Zhang et al. 2017) specifies the dependency among different layers within one iteration to maximize the performance benefits of overlapping. With the particular focus on overlapping between two adjacent iterations, P3 (Jayarajan et al. 2019) attempts to overlap the communication in the current iteration and the forwarding propagation in the next iteration. It schedules the transmission of data packets by a priority-based strategy. ByteScheduler (Peng et al. 2019) introduces a data transfer window to fully utilize the network bandwidth, which in turn weakens the preemption ability of the data packets with high priority. Different from the prior works above, Nebula presents how to ration network resources for co-located PS and worker tasks to mitigate the network contention, which mainly reduces the communication time from the aspect of the network links. In addition, we plan to extend Nebula to support several recent orthogonal works on network-level flow scheduling (e.g., (Wang et al. 2020c)).

6.3 Bandwidth allocation for VMs and applications

There have been many previous works on the bandwidth allocation for virtual machines (VMs) and applications. For example, Falloc (Guo et al. 2015) models the datacenter bandwidth allocation as a cooperative game to achieve the VM-based fairness in datacenter networks. AppBag (Shen et al. 2020) is an application-aware bandwidth guarantee framework. Unlike the complicated methods above including the asymmetric Nash bargaining solution in Falloc, JCAB (Wang et al. 2020a) based on Lyapunov optimization and Markov approximation, and a recent work (Panayiotou et al. 2019) based on reinforcement learning, Nebula proposes a simple heuristic bandwidth rationing algorithm to adjust the network bandwidth for co-located PS and worker tasks, thereby achieving good DDNN training performance. Moreover, these prior works above usually require amounts of historical data, and AppBag (Shen et al. 2020) allocates the accurate bandwidth to VMs with one-step-ahead traffic information. In contrast, Nebula only requires the real-time available network bandwidth data, which can be generated during the training process and be captured on each GPU machine with a low cost. In addition, different from the implementation on the emulated network (Xu et al. 2017) and the SDN environment, Nebula and Falloc are lightweight and implemented based on the *tc* controller.

7 Conclusion and future work

To mitigate the network resource contention and alleviate the performance variations of DDNN training jobs, this paper presents the design and implementation of Nebula, a simple yet effective network bandwidth resource allocation strategy by monitoring the *weights* of co-located PS and worker tasks. Specifically, Nebula first acquires the communication data information including the layer index and its class based on job profiling. Nebula then calculates the weights of two co-located tasks according to the class information of communication data transferred by PS and worker tasks. By comparing the two task weights, Nebula adequately rations the network bandwidth resource online for co-located PS and worker tasks, so as to speed up the performance of DDNN training jobs. We implement a prototype of Nebula based on BytePS, which is an open-source general distributed training framework. We conduct extensive prototype experiments by training representative DNN models on Amazon EC2. Our experiment results demonstrate that Nebula is able to reduce the iteration time of a DDNN training job by up to 25%, and improve the cluster resource utilization by up to 30% in comparison to MXNet.

As our future work, we plan to extend Nebula in two directions: (1) incorporating the factor of power consumption into our bandwidth allocation strategy, and (2) allocating the network bandwidth for PS and worker tasks using the SDN techniques.

Acknowledgements This work was supported in part by the NSFC under grant No.61972158, in part by the Science and Technology Commission of Shanghai Municipality under grant No.20511102802 and No.18DZ2270800, and in part by the Tencent Corporation. Li Chen's work was supported by a grant from BoRSF-RCS under the contract LEQSF(2019-22)-RD-A-21. Zhi Zhou's work was supported in part by the NSFC under grant No.61802449.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: a system for large-scale machine learning. Proc. USENIX OSDI **2016**, 265–283 (2016)

Berral JL, Wang C, Youssef A (2020) AI4DL: mining behaviors of deep learning workloads for resource management. In: Proceedings of USENIX HotCloud (2020)

Chen, C., Wang, W., Li, B.: Round-Robin synchronization: mitigating communication Bottlenecks in parameter servers. Proc IEEE INFOCOM **2019**, 532–540 (2019)

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., Zhang, Z.: Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems (2015). arXiv preprint arXiv:151201274

Gu, J., Chowdhury, M., Shin, K.G., Zhu, Y., Jeon, M., Qian, J., Liu, H., Guo, C.: Tiresias: a GPU cluster manager for distributed deep learning. Proc. USENIX NSDI **2019**, 485–500 (2019)

Guo, J., Liu, F., Lui, J.C.S., Jin, H.: Fair network bandwidth allocation in iaas datacenters via a cooperative game approach. IEEE/ACM Trans. Netw. **24**, 873–886 (2015)

Guptaand, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. Proc. ICML **2015**, 1737–1746 (2015)

Huang, X.S., Chen, A., Ng, T.: Green, yellow, yield: end-host traffic scheduling for distributed deep learning with tensorlights. Proc. IEEE IPDPSW **2019**, 430–437 (2019)

Jayarajan, A., Wei, J., Gibson, G., Fedorova, A., Pekhimenko, G.: Priority-based parameter propagation for distributed DNN training. In: Talwalkar, A., Smith, V., Zaharia, M. (eds.) Proceedings of Machine Learning and Systems 2019, vol. 3, pp. 132–145 (2019)

Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., Yang, F.: Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. Proc. USENIX ATC **2019**, 947–960 (2019)

Jiang, Y., Zhu, Y., Lan, C., Yi, B., Cui, Y., Guo, C.: A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In: Proceedings of USENIX OSDI, pp 463–479 (2020)

Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Proc. NIPS **2012**, 1097–1105 (2012)

Kshiteej, M., Arjun, B., Arjun, S., Shivaram, V., Aditya, A., Amar, P., Shuchi, C.: Themis: fair and efficient GPU cluster scheduling. Proc. USENIX NSDI **2020**, 289–304 (2020)

Lin, Y., Han, S., Mao, H., Wang, Y., Dally, W.J.: Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. (2017) arXiv preprint arXiv:171201887

Luo, L., Nelson, J., Ceze, L., Phanishayee, A., Krishnamurthy, A.: Parameter hub: a rack-scale parameter server for distributed deep neural network training. Proc. ACM SOCC **2018**, 41–54 (2018)

Luo L, West P, Krishnamurthy A, Ceze L, Nelson J (2020) PLink: Discovering and Exploiting Datacenter Network Locality for Efficient Cloud-based Distributed Training. Proc. of MLSys 2020

Mai, L., Hong, C., Costa, P. (2015) Optimizing network performance in distributed machine learning. In: Proceedings of USENIX HotCloud 2015

Mayer, R., Jacobsen, H.A.: Scalable deep learning on distributed infrastructures: challenges, techniques, and tools. ACM Comput Surv (CSUR) **53**, 1–37 (2020)

Mirhoseini, A., Pham, H., Le, Q.V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., Dean, J.: Device placement optimization with reinforcement learning. Proc. ICML **2017**, 2430–2439 (2017)

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N.R., Ganger, G.R., Gibbons, P.B., Zaharia, M.: PipeDream: generalized pipeline parallelism for DNN training. Proc. ACM SOSP **2019**, 1–15 (2019)

Panayiotou, T., Manousakis, K., Chatzis, S.P., Ellinas, G.: A data-driven bandwidth allocation framework With QoS considerations for EONs. J Lightwave Technol **37**, 1853–1864 (2019)

Peng, Y., Bao, Y., Chen, Y., Wu, C., Guo, C.: Optimus: an efficient dynamic resource scheduler for deep learning clusters. Proc. EuroSys **2018**, 1–14 (2018)

Peng, Y., Zhu, Y., Chen, Y., Bao, Y., Yi, B., Lan, C., Wu, C., Guo, C.: A generic communication scheduler for distributed DNN training acceleration. Proc. ACM SOSP **2019**, 16–29 (2019)

Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall, New York (2020)

Shen, D., Luo, J., Dong, F., Jin, J., Zhang, J., Shen, J.: Facilitating Application-Aware Bandwidth Allocation in the Cloud with

One-Step-Ahead Traffic Information. IEEE Trans. Serv. Comput. **13**, 381–394 (2020)
Shi, S., Chu, X., Li, B.: MG-WFBP: efficient data communication for distributed synchronous SGD algorithms. Proc. IEEE INFOCOM **2019**, 172–180 (2019)
Shi, S., Wang, Q., Chu, X., Li, B., Qin, Y., Liu, R., Zhao, X.: Communication-efficient distributed deep learning with merged gradient sparsification on gpus. In: Proceedings of IEEE INFOCOM 2020 (2020)
Ukidave, Y., Li, X., Kaeli, D.: Mystic: predictive scheduling for Gpu based cloud servers using machine learning. Proc. IEEE IPDPS **2016**, 353–362 (2016)
Wang, C., Zhang, S., Chen, Y., Qian, Z., Wu, J., Xiao, M.: Joint configuration adaptation and bandwidth allocation for edge-based real-time video analytics. Proc. IEEE INFOCOM **2020**, 1–10 (2020)
Wang, Q., Shi, S., Wang, C., Chu, X. Communication Contention Aware Scheduling of Multiple Deep Learning Training Jobs. (2020b). arXiv preprint arXiv:200210105
Wang, S., Li, D., Geng, J.: Geryon: accelerating distributed CNN training by network-level flow scheduling. Proc. IEEE INFOCOM **2020**, 1678–1687 (2020)
Xu, F., Ye, W., Liu, Y., Zhang, W.: Ufalloc: towards utility max-min fairness of bandwidth allocation for applications in datacenter networks. Mobile Netw. Appl. **22**, 161–173 (2017)
Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., Hu, Z., Wei, J., Xie, P., Xing, E.P.: Poseidon: an efficient communication architecture for distributed deep learning on GPU clusters. Proc. USENIX ATC **2017**, 181–193 (2017)

Qiang Qi received his BS degree in Computer Science from East China Normal University (ECNU) in 2018. He is currently pursuing his MS degree in Computer Science in the School of Computer Science and Technology at ECNU. His current research interests focus on cloud datacenter networks and distributed machine learning systems.

Fei Xu received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2014. He received Outstanding Doctoral Dissertation Award in Hubei province, China, and ACM Wuhan & Hubei Computer Society Doctoral Dissertation Award in 2015. He is currently an associate professor with the School of Computer Science and Technology, East China Normal University, Shanghai, China. His research interests include cloud computing and datacenter, virtualization technology, and distributed systems.

Li Chen received the BEngr degree from the Department of Computer Science and Technology, Huazhong University of Science and Technology, China, in 2012 and the MASc degree from the Department of Electrical and Computer Engineering, University of Toronto, in 2014 and the PhD degree in computer science and engineering from the Department of Electrical and Computer Engineering, University of Toronto, in 2018. She is currently an assistant professor with the Department of Computer Science, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, USA. Her research interests include big data analytics systems, cloud computing, datacenter networking, and resource allocation.

Zhi Zhou received the B.S., M.E., and Ph.D. degrees in 2012, 2014, and 2017, respectively, all from the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan, China. He is currently an associate professor in the School of Computer Science and Engineering at Sun Yat-sen University, Guangzhou, China. In 2016, he was a visiting scholar at University of Göttingen. He was nominated for the 2019 CCF Outstanding Doctoral Dissertation Award, the sole recipient of the 2018 ACM Wuhan & Hubei Computer Society Doctoral Dissertation Award, and a recipient of the Best Paper Award of IEEE UIC 2018. His research interests include edge computing, cloud computing, and distributed systems.