Prophet: Speeding up Distributed DNN Training with Predictable Communication Scheduling

Zhenwei Zhang*, Qiang Qi*, Ruitao Shang*, Li Chen†, Fei Xu*
*Shanghai Key Laboratory of Multidimensional Information Processing,
School of Computer Science and Technology, East China Normal University, Shanghai 200062, China.
†School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA 70504, USA.
*fxu@cs.ecnu.edu.cn, †li.chen@louisiana.edu

ABSTRACT

Optimizing performance for Distributed Deep Neural Network (DDNN) training has recently become increasingly compelling, as the DNN model gets complex and the training dataset grows large. While existing works on communication scheduling mostly focus on overlapping the computation and communication to improve DDNN training performance, the GPU and network resources are still *under-utilized* in DDNN training clusters. To tackle this issue, in this paper, we design and implement a *predictable* communication scheduling strategy named *Prophet* to schedule the gradient transfer in an adequate order, with the aim of maximizing the GPU and network resource utilization. Leveraging our observed *stepwise pattern* of gradient transfer start time, *Prophet* first uses the monitored network bandwidth and the profiled time interval among gradients to predict the appropriate number of gradients that can be grouped into *blocks*. Then, these *gradient blocks* can be transferred one by one to guarantee high utilization of GPU and network resources while ensuring the priority of gradient transfer (*i.e.,* low-priority gradients cannot preempt high-priority gradients in the network transfer). *Prophet* can make the forward propagation start as early as possible so as to greedily reduce the waiting (idle) time of GPU resources during the DDNN training process. Prototype experiments with representative DNN models trained on Amazon EC2 demonstrate that *Prophet* can improve the DDNN training performance by up to 40% compared with the state-of-the-art priority-based communication scheduling strategies, yet with negligible runtime performance overhead.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; **Neural networks**; • **Theory of computation** → *Scheduling algorithms.*

KEYWORDS

distributed DNN training, communication scheduling, gradient transfer, resource utilization

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472467

ACM Reference Format:
Zhenwei Zhang*, Qiang Qi*, Ruitao Shang*, Li Chen†, Fei Xu*. 2021. *Prophet:* Speeding up Distributed DNN Training with Predictable Communication Scheduling. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3472467

1 INTRODUCTION

Deep Neural Network (DNN) has widely been used in a variety of applications such as natural language processing, image processing, *etc.* As DNN model training requires a large amount of data and the model structure gets increasingly complex, the available memory space and computational power of a single machine become more stringent [20]. Accordingly, it becomes compelling to train DNN models in a distributed manner [7], as big IT companies have released various Distributed DNN (DDNN) training frameworks such as TensorFlow [1], MXNet [6], PaddlePaddle [19], Horovod [23]. As a result, optimizing DDNN training performance has received much research attention recently.

To execute DDNN training jobs in parallel, there are two parallel schemes including data parallelism [13] and model parallelism [16]. The former one refers to the scheme that all nodes in the training cluster run the whole DNN model, and the training data is split into several chunks for distributed computation. The latter one indicates that cluster nodes run different parts of the DNN model separately, and thus requires careful partitioning of the DNN model. In general, the most common scheme of DDNN training is data parallelism with the Parameter Server (PS) architecture [17]. The PS is responsible for maintaining the up-to-date model parameters, while workers are responsible for computing forward and backward propagation gradients using the latest parameters `pulled` from the PS, and periodically exchanging and updating the model parameters via the network of training clusters [5].

Unfortunately, the network communication of updating model parameters on the PS nodes always becomes a bottleneck, and thus the communication can *block* the computation of workers, which inevitably *under-utilizes* the GPU resources of worker nodes [18]. Though recent high-speed and low-latency network techniques such as Remote Direct Memory Access (RDMA) can alleviate such performance bottleneck, it is reported that RDMA cannot work well with small-sized DNN models [11]. Several research efforts (*e.g.,* Poseidon [29]) are dedicated to overlapping the backward propagation with the gradient `push` process, and integrate such a communication approach into MXNet [6]. Even with enabling such a naive communication scheduling method above, the GPU and network resources can still be *under-utilized.* As evidenced

by our motivation experiment of training ResNet152 with 4 EC2 instances using MXNet (*i.e.,* Sec. 2.2), the GPU utilization of worker nodes can be totally idle over 50% of the iteration time, and the network resources can almost be idle during the computation of workers. Such resource under-utilization mainly originates from delaying the high-priority gradient transfer tasks, and thus the workers cannot start forward propagation until the updated model parameters have been `pulled` from the PS.

To accelerate the DDNN training process, there have been recent works (*e.g.,* P3 [10], TicTac [8]) devoted to explicitly scheduling the transfer of *high-priority* gradients (*e.g.,* `gradient 0`), so as to start the `pull` operations as early as possible. However, such a communication scheduling approach can inevitably bring much performance overhead due to small-sized gradient partitions (as evidenced by Sec. 2.2). Several recent works (*e.g.,* ByteScheduler [21]) focus on improving the network resource utilization by transferring a `credit size` of gradients at a time. Nevertheless, such a credit-based communication scheduling method is likely to sacrifice the transfer priority of gradients, and the auto-tuning process of credit size can significantly degrade the training performance (as evidenced by Sec. 2.2). Accordingly, the existing communication scheduling strategies for DDNN training above are mainly based on gradient prioritization or partitioning, and such *static* configurations of partition size and credit size can hardly adapt to the *dynamic* network environments during the DDNN training [26]. As a result, there has been scant research attention paid to jointly improving the GPU and network resource utilization and ensuring the priority of gradient transfer (*i.e.,* starting the forward propagation as early as possible) to speed up the DDNN training process.

To fill this gap, this paper presents a predictable communication scheduling strategy named *Prophet* to improve the GPU and network resource utilization in DDNN training clusters, while guaranteeing the priority of gradient transfer. Specifically, we make three contributions in *Prophet* as follows.

▷ *First,* we empirically identify and analyze the *stepwise pattern* of gradient transfer start time for DDNN training jobs. We build a DDNN training performance model in terms of gradient transfer order to illustrate that, the root cause of low resource utilization is actually the long wait time of GPU resources, which is essentially caused by the inadequate communication scheduling.

▷ *Second,* we design a *simple yet effective* communication scheduling strategy to schedule the gradient transfer in an adequate order. Specifically, *Prophet* uses the profiled gradient time interval and the periodically monitored network bandwidth to assemble an appropriate number of gradients into *gradient blocks*. It preempts the transfer of high-priority gradients (*i.e.,* ensuring the priority of gradient transfer) while guaranteeing high GPU and network resource utilization.

▷ *Finally,* we implement a prototype of *Prophet* based on an open-source project named BytePS [12] and conduct extensive prototype experiments with representative DNN models on Amazon EC2. Experiment results demonstrate that *Prophet* can improve the DDNN training performance by up to 40% compared with the state-of-the-art priority-based communication scheduling strategies (*e.g.,* ByteScheduler [21], P3 [10]).

The rest of the paper is organized as follows. Sec. 2 analyzes the root cause of the low GPU utilization of worker nodes in DDNN training clusters. Sec. 3 leverages the gradient transfer order and gradient transfer start time to build a performance model of DDNN training workloads. Sec. 4 designs *Prophet* to schedule the gradient transfer in an optimal order, so as to improve the resource utilization of worker nodes and accelerate the DDNN training process. Sec. 5 evaluates the effectiveness and runtime overhead of *Prophet*. Sec. 6 discusses our contribution in the context of related work, and Sec. 7 concludes our paper.

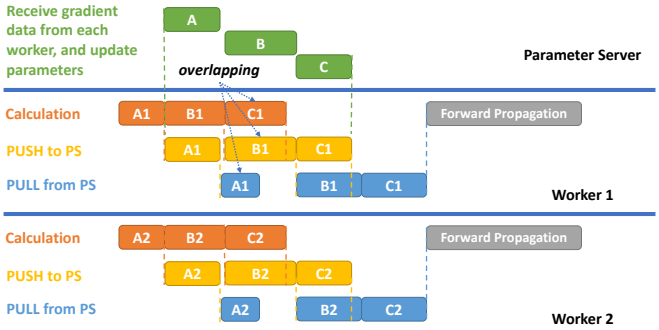


Figure 1: Illustration of scheduling communication data (*i.e.,* gradients or parameters) for one training iteration with the PS architecture. For example, `A`, `B`, `C` are communication data.

2 BACKGROUND AND MOTIVATION

In this section, we first analyze the low GPU utilization of worker nodes and the drawbacks of the state-of-the-art communication scheduling approaches. Then, we present an illustrative example to show how to adequately schedule the communication data to speed up the DDNN training performance.

2.1 Network Communication Scheduling for DDNN Training

Network communication of DDNN training mainly refers to the process of updating model parameters between the *PS* and *worker* nodes through the `pull` and `push` operations. As illustrated in Fig. 1, the workers first `push` the computed gradient data to the PS, and after data aggregation on the PS, the workers then `pull` the updated model parameters from the PS. In general, we refer to model parameters and gradient data as *communication data*, which is transferred in the form of *tensors* via the network. As the conventional DDNN training frameworks adopt the First-in First-Out (FIFO) strategy for data transmission by default, a large communication data is likely to block the transfer of *high-priority* tensors and thus causes low utilization of GPU resources. To solve such an issue, several communication scheduling strategies (*e.g.,* Poseidon [29]) have been proposed to *overlap* the gradient synchronization (*e.g.,* pulling `A1` and pushing `B1`) with backward propagation (*e.g.,* calculating `C1`) as depicted in Fig. 1. Accordingly, the updated model parameters can be `pulled` during the process of backward propagation, which significantly reduces the communication blocking time during the data aggregation process. To improve the GPU resource utilization and training performance, such a scheduling strategy of overlapping computation and communication has now been integrated into the implementation of MXNet [6] by default.

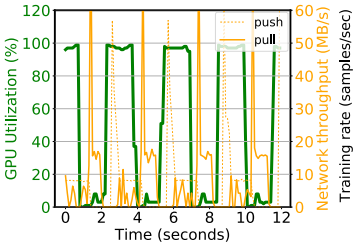
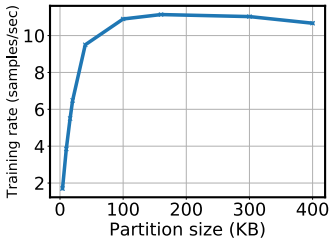
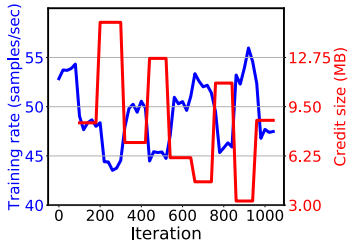


Figure 2: GPU utilization and network throughput over time of a worker node during the training process of ResNet152 model.



(a) P3



(b) ByteScheduler

Figure 3: Non-negligible performance overhead of state-of-the-art priority-based communication scheduling strategies (*i.e.,* P3, ByteScheduler).

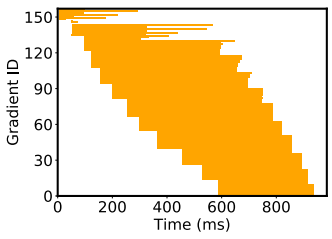


Figure 4: Communication characteristic of gradient transfer, which follows a stepwise pattern over time, by taking ResNet50 as an example.

2.2 Characterizing Resource Utilization of Worker Nodes

Though provided with the communication scheduling strategies in existing DDNN training frameworks, the GPU resources of worker nodes still suffer from low utilization. We illustrate such a problem by conducting a motivation experiment on Amazon EC2 using four `g3.8xlarge` instances (*i.e.,* 1 PS and 3 worker nodes). Each instance is equipped with 32 vCPUs and 2 GPUs and trains ResNet50 and ResNet152 with the ImageNet dataset [14] using MXNet. As shown in Fig. 2, we observe that: Though the `push` operation of gradients overlaps with the computation, the GPU utilization can dramatically decrease to zero (*i.e.,* totally idle) during the `pull` operation of model parameters. The experiment result is mainly caused by the slow network transmission, which makes the GPUs fail to *timely* acquire the model parameters and thus delays the computation (*i.e.,* forward propagation).

To solve such performance issues above, several *priority*-based communication scheduling approaches have recently been proposed. For example, P3 [10] slices the gradients into smaller partitions and transmits them individually from high priority to low priority so as to reduce the communication overhead. However, the smaller size of partitions dramatically decreases the DDNN training rate, as shown in Fig. 3(a). Such severe performance overhead (*i.e.,* TCP connection overhead, TCP slow start, and the synchronization between nodes) is mainly caused by a huge number of partitions and under-utilized sending buffers. Accordingly, ByteScheduler [21] adopts a credit-based communication scheduling strategy to balance the preemption frequency and performance overhead caused by the gradient partitioning. Bayesian optimization is used to explore an appropriate `credit size` for better training performance in ByteScheduler. However, the training rate of the ResNet50 model fluctuates rapidly from 44 – 56 samples/sec by ByteScheduler as shown in Fig. 3(b). Such an experimental result implies that the `credit size` auto-tuning process can bring significant fluctuations to the DDNN training rate, as the `credit size` is dynamically adjusted from around 3 MB to over 13 MB at runtime.

To tackle the drawbacks of P3 [10] and ByteScheduler [21] discussed above, we further analyze the communication scheduling of gradients to unveil the root cause of low GPU utilization of worker nodes. As depicted in Fig. 4, the start time of gradient transfer clearly follows a *stepwise pattern*: Taking MXNet and the ResNet50 model as an example, {`gradient 144 - gradient 156`} are generated within a short time, and after a while, {`gradient 134 - gradient 143`} are generated almost simultaneously. Such a *stepwise pattern* is valid until `gradient 0` is generated. Similarly, we also observe the *stepwise pattern* when training the VGG19 model using TensorFlow, where the gradients can be grouped as four *blocks*: {`gradient 28 - gradient 37`}, {`gradient 14 - gradient 27`}, {`gradient 2 - gradient 13`}, and {`gradient 0 - gradient 1`}. Our experimental results above indicate that the *stepwise pattern* of gradient transfer is independent of the DDNN training frameworks, DNN models, datasets, and hardware architectures.

Root causes of our observed stepwise pattern: Through analyzing the communication mechanisms of DDNN training, we find *the gradient data requires aggregation before transmission,* which can be considered as the main cause of *stepwise pattern*. Specifically, the PS architecture of DDNN training provides a Key-Value interface (*e.g.,* `KVStore` in MXNet, `RendezvousServer` in Horovod, and `Communication Buffer` in TensorFlow) to workers, and stores the gradient data computed by each node in the communication domain. Before the Key-Value storage communicates with the PS to invoke the `push` operation, it has to aggregate the values (*i.e.,* gradient data) with the same key into a data structure through several specific operations (*e.g.,* `GroupKVPairsPush` in MXNet), so as to facilitate the invocation of the relevant communication data. Accordingly, the process above requires aggregating a set of gradient data before each `push` operation, thereby forming such a *stepwise pattern*. Moreover, we find several other factors (*i.e.,* the DNN model optimization, the CPU buffer when performing `copyD2H` operations and the network sending buffer) further contribute to the *stepwise pattern*, as these factors can optimize the gradient aggregation process by allowing the gradient data of one or more layers to conduct update operations together. As a result, we can further leverage such a *stepwise pattern* to optimize the transmission order of gradient data, so as to improve the DDNN training performance.

2.3 An Illustrative Example

To speed up DDNN training performance, we propose a *simple yet effective* communication scheduling strategy named *Prophet* to schedule the gradient transfer in an optimal order, with the aim of improving the GPU and network resource utilization of worker nodes. The **core idea** of *Prophet* is that: It predicts the transferred gradient data size by profiling the time interval between *blocks* and the available network bandwidth during model training. *Prophet* ensures that each gradient can be transferred by greedily utilizing

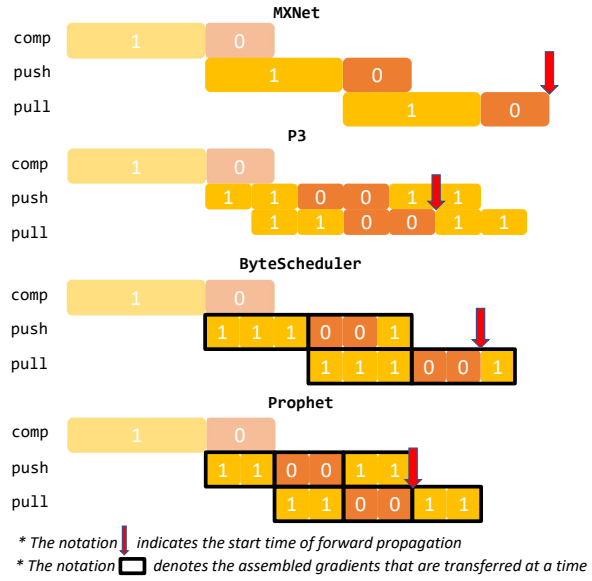


Figure 5: An illustrative example of DDNN training with different communication scheduling strategies (*i.e.,* default MXNet, P3, ByteScheduler, and *Prophet*).

the network bandwidth resources without blocking the higher-priority gradients, so that the critical gradients (*i.e.,* `gradient 0`) can be transferred as fast as possible. Accordingly, the computation of the forward propagation can start early, and thus the GPU and network resource utilization can be improved.

As illustrated in Fig. 5, *Prophet* can achieve the accelerated DDNN training performance, compared with the other scheduling strategies. Specifically, the default MXNet transmits gradients one by one with the default FIFO order. As `gradient 1` takes a long time to be `pushed` to the PS, the worker nodes have to wait even after `gradient 0` is generated. P3 [10] slices the gradients into small partitions to ensure a timely preemption, so that the higher-priority gradients are transmitted earlier. The extra slicing overhead, however, under-utilizes the network bandwidth resources for the small `partition size` as discussed in Sec. 2.2, thereby prolonging the transfer time of a single partition. To avoid such partition overhead, ByteScheduler [21] configures the `credit size` as an empirical value (*i.e.,* 3 times `partition size` in Fig. 5), while keeping a relatively high preemption rate. To reduce the wait time of GPU resources during the gradient transfer, our proposed *Prophet* only assembles the gradients that can greedily utilize the network resources before any higher-priority gradient (*e.g.,* `gradient 0`) is generated. In more detail, when `gradient 1` is generated in Fig. 5, our proposed *Prophet* infers that only two partitions of `gradient 1` can be transmitted before `gradient 0` is generated. It then assembles the two partitions of `gradient 1` and start transmission, so that the high-priority `gradient 0` can be transferred timely and the resource utilization can also be improved. As a result, the *stepwise pattern* provides us an opportunity to optimize the DDNN training performance.

3 MODELING AND PROBLEM FORMULATION

In this section, we first analyze the DDNN training performance in terms of the gradient transfer order in the network transmission queue, the available network bandwidth of workers, and the size of transferred gradients, which has seldom been studied in the literature. We next formulate an optimization problem to minimize the DDNN training time. The key notations in our DDNN training performance model are summarized in Table 1.

Table 1: Key notations in our DDNN training performance model.

Notation	Definition
$\mathcal{X}$	Set of gradient data
i	Gradient index (*i.e.,* priority)
$x^{(i)}$	The i-th gradient
$s^{(i)}$	Size of $x^{(i)}$
$c^{(i)}$	Time when $x^{(i)}$ is generated by backward propagation
$t^{(i)}$	Start time of gradient transfer $x^{(i)}$
$E^{(i)}$	Estimated time cost to `push`/`pull` $x^{(i)}$
$u^{(i)}$	Completion time of parameter update for $x^{(i)}$
$p^{(i)}$	Completion time of forward propagation for $x^{(i)}$
$B^{(i)}$	Actual network bandwidth for transferring $x^{(i)}$
B	Available network bandwidth of workers
$T_{bp}^{(i)}$	Time cost of the backward propagation for $x^{(i)}$
$T_{fp}^{(i)}$	Time cost of the forward propagation for $x^{(i)}$

3.1 Modeling DDNN Training Time with Gradient Transfer Order

As depicted in Fig. 6, DDNN training frameworks compute the gradient data one by one in the backward propagation by default. Though the communication process (*i.e.,* the `push` and `pull` operations) overlaps with the backward propagation, as discussed in Sec. 2.1, GPUs have to *wait* for the high-priority gradients (*e.g.,* `gradient C`) to start forward propagation. Accordingly, we calculate the training time T_{all} of one iteration as below,

$$\begin{aligned} T_{all} &= \sum_{i\in\mathcal{X}} T_{bp}^{(i)} + \sum_{i\in\mathcal{X}} T_{fp}^{(i)} + T_{communication} - T_{overlap} \\ &= \sum_{i\in\mathcal{X}} T_{bp}^{(i)} + \sum_{i\in\mathcal{X}} T_{fp}^{(i)} + T_{wait}, \end{aligned} \tag{1}$$

where $\sum_{i\in\mathcal{X}} T_{bp}^{(i)}$ and $\sum_{i\in\mathcal{X}} T_{fp}^{(i)}$ denote the time cost for backward propagation and forward propagation, respectively, and $T_{wait} = T_{communication} - T_{overlap}$ denotes the total idle time of GPU resources during the entire training process. Accordingly, GPU resource utilization can be maximized as T_{wait} ideally approaches to zero.

In general, GPUs are in the working state during the backward propagation until `gradient 0` is generated at $c^{(0)}$. After the `push` and `pull` operations, GPUs are back to the working state when `gradient 0` updates its parameters at $u^{(0)}$. Accordingly, we obtain an amount of time (*i.e.,* $u^{(0)} - c^{(0)}$) that GPUs are *waiting* for the transfer of `gradient 0`. In the forward propagation, GPUs will be in the *waiting* state when the expected gradient data $x^{(i)}$ has not

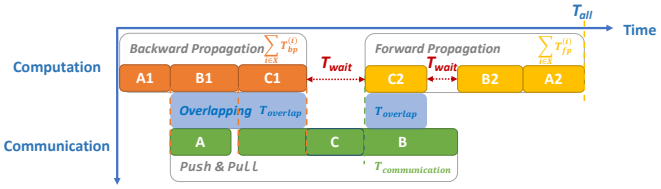


Figure 6: Illustration of DDNN training time T_{all} in one iteration.

been updated while the previous one $x^{(i-1)}$ has finished its forward propagation at $p^{(i-1)}$. Accordingly, we obtain another part of GPU idle time $\left(u^{(i)} - p^{(i-1)}\right)^+$. In particular, the network communication of *timely* transferred gradients actually overlaps with the forward propagation (*i.e.,* $u^{(i)} < p^{(i-1)}$), and thus we only use the *positive part* of $u^{(i)} - p^{(i-1)}$. As a result, we calculate the total GPU wait time T_{wait} by accumulating all the *positive* time intervals as below,

$$T_{wait} = \sum_{i \in \mathcal{X}, i \neq 0} \left(u^{(i)} - p^{(i-1)}\right)^+ + \left(u^{(0)} - c^{(0)}\right). \tag{2}$$

To identify the relationship between T_{wait} and the start time $t^{(i)}$ of gradient transfer (*i.e.,* start time to push $x^{(i)}$), we proceed to formulate $u^{(i)}$ and $p^{(i-1)}$ in terms of $t^{(i)}$. As discussed in Sec. 2.1, the gradient $x^{(i)}$ can start its forward propagation only when the previous gradient $x^{(i-1)}$ finishes the forward propagation and $x^{(i)}$ completes its parameter update process. Accordingly, we calculate the completion time $p^{(i)}$ of forward propagation for gradient $x^{(i)}$ as below,

$$p^{(i)} = \begin{cases} \max\{p^{(i-1)}, u^{(i)}\} + T_{fp}^{(i)} & \text{when} \quad i \neq 0, \\ u^{(0)} + T_{fp}^{(0)} & \text{when} \quad i = 0. \end{cases} \tag{3}$$

According to the parameter update process, we simply *estimate* the completion time $u^{(i)}$ of parameter update for gradient $x^{(i)}$ as below,

$$u^{(i)} = t^{(i)} + 2 \cdot E^{(i)}, \tag{4}$$

where $E^{(i)}$ is the estimated time for executing the `push` or `pull` operation. As it is difficult to obtain the exact transmission time for the gradient $x^{(i)}$, we can *estimate* the gradient transmission time $E^{(i)}$ using the gradient size $s^{(i)}$ and the actual network bandwidth $B^{(i)}$ as below,

$$E^{(i)} = \frac{s^{(i)}}{B^{(i)}}. \tag{5}$$

3.2 Problem Formulation

The key to design a network communication scheduling strategy for DDNN training workloads is: **How to schedule the gradient transfer $t^{(i)}$ to minimize** T_{all} in Eq. (1), so as to maximize the GPU resource utilization. Note that the time cost of backward propagation $T_{bp}^{(i)}$ and forward propagation $T_{fp}^{(i)}$ are not related to the gradient transfer order, and thus they can be considered as constant values. As a result, our optimization problem then can be **reduced to minimizing** T_{wait}, which is formulated as follows,

$$\min_{t^{(i)}} \quad T_{wait} = \sum_{i \in \mathcal{X}, i \neq 0} \left(u^{(i)} - p^{(i-1)}\right)^+ + \left(u^{(0)} - c^{(0)}\right) \tag{6}$$

$$\text{s.t.} \quad t^{(i)} \geqslant c^{(i)}, \tag{7}$$

$$t^{(i)} \notin [t^{(j)}, t^{(j)} + E^{(j)}), \quad \forall j \in \mathcal{X}, j \neq i, \tag{8}$$

$$t^{(i)} > t^{(k)}, \quad \forall k \in \mathcal{X} < i, \quad \text{when} \quad t^{(i)} > c^{(0)}. \tag{9}$$

Constraint (7) indicates that the gradient $x^{(i)}$ can only be pushed after it is generated. To ensure that each gradient is transferred with the full available network bandwidth, Constraint (8) avoids the concurrent gradient transfer. In the forward propagation, Constraint (9) guarantees that the gradients which are not transmitted during the backward propagation should be transmitted according to the order of gradient priority.

To schedule the gradient transfer, we have to leverage the gradient generation time $c^{(i)}$ which follows a *stepwise pattern* and the gradient transmission time $E^{(i)}$ which depends on the actual network bandwidth $B^{(i)}$. Based on the analysis in Sec. 2.2, a smaller partition size $s^{(i)}$ can bring higher network overhead and thus cause lower network throughput $B^{(i)}$, which can be formulated as

$$B^{(i)} = f(s^{(i)}, B). \tag{10}$$

where B denotes the available network bandwidth and the function $f(\cdot)$ in Eq. (10) is nontrivial to model. However, we can infer that $f(s^{(i)}, B)$ approaches to 0 when $s^{(i)}$ is small, and $f(s^{(i)}, B)$ gradually increases to B as $s^{(i)}$ gets large.

By substituting Eq. (3) – Eq. (5) and Eq. (10) into Eq. (6), we find that our optimization problem is hard to solve because the model parameters (*e.g.,* $T_{fp}^{(i)}$, $B^{(i)}$) can only be obtained at runtime. Accordingly, we turn to designing a heuristic algorithm to identify the appropriate start time $t^{(i)}$ of gradient transfer, with the aim of minimizing T_{wait} and maximizing the GPU resource utilization.

Problem analysis: To determine *when and how* gradients are transferred, P3 [10] and ByteScheduler [21] both focus on two principles: (i) Scheduling the gradient transfer according to the priority order, and (ii) during the backward propagation, it is necessary to ensure the individual gradient transfer defined in Constraint (8) while allowing the preemption of gradient transfer. Accordingly, P3 proposes to use an extremely small partition size to satisfy the two principles above. However, P3 fails to achieve good performance under poor network conditions due to the extra overhead of small partitions. To solve such a problem, ByteScheduler uses the `credit size` to achieve a higher network throughput than P3, while guaranteeing the priority of gradient transfer. Nevertheless, the fixed and auto-tuned hyperparameters (*i.e.,* `credit size`) of ByteScheduler are not designed to minimize $\sum_{i \in \mathcal{X}, i \neq 0} \left(u^{(i)} - p^{(i-1)}\right)^+$ (*i.e.,* a critical part in Eq. (6)), thereby introducing the extra performance overhead during the auto-tuning process of `credit size`.

To minimize $\sum_{i \in \mathcal{X}, i \neq 0} \left(u^{(i)} - p^{(i-1)}\right)^+$, we add an *essential* Constraint (11) to our optimization problem in Eq. (6) as below, which indicates that the gradients should be transferred before any high-priority gradient is generated in the backward propagation.

$$t^{(i)} + E^{(i)} \leqslant c^{(k)}, \quad \forall k \in \mathcal{X} < i, \quad \text{when} \quad t^{(i)} \leqslant c^{(0)}. \tag{11}$$

Constraint (11) implies that we can take advantage of the *stepwise pattern* (*i.e.,* block time interval) to find the optimal gradient transfer, and thus optimizing our formulated problem in Eq. (6) can improve the DDNN training performance. Specifically, letting $t^{(0)} = c^{(0)}$ can minimize $u^{(0)} - c^{(0)}$, and then we are able to determine $t^{(1)}$ by minimizing $(u^{(1)} - p^{(0)})^+$ according to Eq. (3) and Eq. (4). Such a process above can be continuously applied to the remaining gradient $i \in \mathcal{X}$, and finally we can obtain a minimum T_{wait} for all gradient transfer in one training iteration.

In particular, we focus on the PS architecture as it has been widely deployed in production DDNN training clusters [15]. As explained in Sec. 2.2, the gradient transmission time $E^{(i)}$ depends on the available network bandwidth and the clear *stepwise pattern* is valid in the PS architecture. We take advantage of the *stepwise pattern* to schedule the gradient transfer in the PS architecture so as to improve the resource utilization.

4 DESIGN AND IMPLEMENTATION OF *PROPHET*

In this section, we present the design and implementation of *Prophet* in Alg. 1, a *simple yet effective* network communication scheduling strategy to judiciously schedule the gradient transfer during DDNN training. *Prophet* aims to activate the forward propagation of next iteration *as early as possible*, so as to improve the GPU and network resource utilization of workers as well as the DDNN training performance.

4.1 Algorithm Design

We design *Prophet* based on the heuristics below: By leveraging the *block* time interval and the monitored network bandwidth, we determine the *start time of gradient transfer* during both the backward propagation and forward propagation, in order to minimize the *idle time* of GPU and network resources. In particular, we determine the gradient transfer start time using the DNN model and the computation capacity as well as the available network bandwidth of workers, which can be obtained by a lightweight job profiling in *Prophet*.

Specifically, we obtain the input parameters of Alg. 1 by job profiling, including the generation time of the i-th gradient $c^{(i)}$, the size of each gradient $s^{(i)}$ and the available network bandwidth of workers B. The output of Alg. 1 is the start time of each gradient transfer. During the DDNN training iteration, we first initialize the estimated transmission time of `gradient` i, denoted by $E^{(i)} \leftarrow \frac{s^{(i)}}{B}$. Using the generation time $c^{(i)}$ of each gradient, the expected time interval to transfer the i-th gradient can be denoted by $A^{(i)} \leftarrow \min |c^{(i)} - c^{(j)}|, j < i$ (`line 1`). It implies that the i-th gradient is expected to complete its transmission before the higher-priority gradients are generated. To schedule the gradient transfer in each training iteration, *Prophet* continuously assign the start time for each gradient transfer until all the gradients are scheduled (`line 2-20`). In particular, we schedule the gradient transfer in the order of gradient priority (`line 3`).

In more detail, we determine the transfer start time $t^{(i)}$ for each gradient i as follows: (i) In the backward propagation (`line 5-11`), we greedily assemble each gradient as long as it can be transmitted before any higher-priority gradients are generated (`line 7-10`). In particular, T_{used} denotes the used transmission time (`line 6`) and we accumulate it once a new gradient is assembled (`line 9`). According to Constraint (11) in our optimization problem, we assemble the `gradient` q into a *gradient block* (`line 8`) and get the currently highest-priority gradient ready to be transferred (`line 10`), only when the residual time $A^{(q)} - T_{used}$ is positive (`line 7`). (ii) In the forward propagation (`line 13-14`), we transfer the remaining gradients in the order of priority (`line 13`), while ensuring gradients are transferred one by one, which is based on Constraint (9) in our optimization problem. We then update the t_{next} accordingly (`line 14`). As the generation of `gradient 0` indicates the completion of backward propagation and the start of forward propagation for the next iteration, we greedily transfer `gradient 0` once it is generated, in order to start the forward propagation as early as possible. Accordingly, we set the transfer start time of `gradient 0` as its generation time (`line 17`) and update the next available time with the estimated transmission time (`line 18`). In particular, Alg. 1 can be executed periodically to schedule the network transfer of gradients in each iteration.

Remark: Alg. 1 runs on each worker, so workers are likely to have different gradient transfer orders especially in a heterogeneous environment. However, the scheduling difference of gradient

Algorithm 1: *Prophet*: Communication scheduling strategy for improving resource utilization of workers and minimizing the DDNN training time.

Input: Current available network bandwidth B of workers, the gradient generation time $c^{(i)}$ and the gradient size $s^{(i)}$ for each gradient $i \in \mathcal{X}$.
Output: Start time $t^{(i)}$ of gradient transfer (*i.e.,* the start time to push each gradient $i \in \mathcal{X}$).

1: **Initialize:** Estimated gradient transmission time $E^{(i)} \leftarrow \frac{s^{(i)}}{B}$ (by Eq. (5) and Eq. (10)), and the expected transfer time interval $A^{(i)} \leftarrow \min |c^{(i)} - c^{(j)}|, j < i$, for each gradient $i \in \mathcal{X}$.
2: **while** exists gradients to be scheduled **do**
3: $p \leftarrow$ the highest priority of gradients ready to be scheduled;
4: **if** $p \neq$ `gradient 0` **then**
5: **if** the current time is in the backward propagation **then**
6: Initialize $T_{used} \leftarrow 0$, $q \leftarrow p$;
7: **while** the `gradient` q can be greedily transferred within the expected time interval $A^{(q)} - T_{used}$ **do**
8: Assemble the gradient q into a *gradient block* and update $t^{(q)} \leftarrow T_{used} + c^{(p)}$;
9: Update the used time $T_{used} \leftarrow T_{used} + E^{(q)}$;
10: Set q as the currently highest-priority gradient that is ready to be transferred;
11: **end while**; `// backward propagation.`
12: **else**
13: Set $t^{(p)}$ as the earliest available scheduling time t_{next};
14: Update the next available time $t_{next} \leftarrow t_{next} + E^{(p)}$;
15: **end if**; `// forward propagation.`
16: **else**
17: Set $t^{(0)}$ as the generation time $c^{(0)}$ of `gradient 0`;
18: Update the next available time $t_{next} \leftarrow t^{(0)} + E^{(0)}$;
19: **end if**
20: **end while**
21: **return** the start time $t^{(i)}$ of gradient transfer;

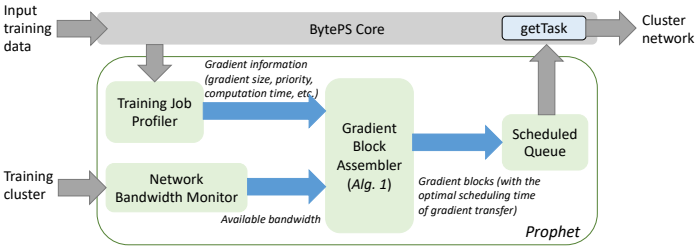


Figure 7: Overview of Prophet prototype based on BytePS.

transfer among workers can be limited and slightly affects the performance of *Prophet*, which will be validated by our prototype experiments in Sec. 5.3. In particular, we explicitly take advantage of the block time interval $A^{(i)}$ (*i.e.,* the stepwise pattern of gradient transfer) to decide an appropriate scheduling plan of gradient transfer (`lines 3-11` in Alg. 1).

4.2 Implementation of *Prophet* Prototype

We implement a prototype of *Prophet* with over 500 C++ and Linux Shell codes, which are publicly available on GitHub[1]. To be compatible with the mainstream DDNN training frameworks, our *Prophet* is implemented based on the abstraction layer of BytePS [12], which is originally designed to support multiple frameworks (*e.g.,* TensorFlow, MXNet, Pytorch). We incorporate our network communication scheduling strategy in Alg. 1 to the BytePS Core as shown in Fig. 7, while retaining the abstraction layer plugins provided by BytePS to interact with the APIs of different frameworks.

Specifically, our prototype implementation of *Prophet* first requires a model *pre-training* process. The training input data is sent to `Training Job Profiler` which *pre-trains* the DNN model for a certain number of iterations (*e.g.,* 50), to obtain the gradient information (*e.g.,* the set of gradient data, the computation time and size of each gradient) required by Alg. 1. To adapt to the dynamic network environments, `Network Bandwidth Monitor` periodically (*e.g.,* every 5 seconds) acquires the available network bandwidth B of workers in the training cluster. Then, `Gradient Block Assembler` leverages the gradient and network bandwidth information obtained above to find the optimal start time of gradient transfer by Alg. 1. It *greedily* assembles an appropriate number of gradients into *blocks*, as long as they can be transferred within the block time interval. In particular, these gradient blocks are pushed to the `Scheduled Queue` while maintaining the priority order of gradients. Finally, the `getTask` interface in the BytePS Core wraps up gradients in the form of the network data, which is scheduled to be transferred over the cluster network. In more detail, the DDNN training framework continuously polls the `Scheduled Queue` and obtains the gradients to transfer the data accordingly to the gradient transfer start time. When the transmission of a gradient block is completed, `Scheduled Queue` maintains the necessary information (*e.g.,* gradient transfer logs, the block size) using the `reportFinish` interface in the BytePS Core for scheduling gradient transfer in the next iteration.

[1]https://github.com/icloud-ecnu/prophet

5 PERFORMANCE EVALUATION

In this section, we carry out a set of prototype experiments on Amazon EC2 to evaluate the effectiveness (*i.e.,* model training rate, GPU and network resource utilization) and runtime overhead of *Prophet*, in comparison to the default MXNet and the state-of-the-art communication scheduling strategies (*i.e.,* ByteScheduler [21] and P3 [10]) using representative DDNN training workloads.

5.1 Experimental Setup

Configurations of DDNN training cluster: We build a distributed GPU cluster with up to 8 `g3.8xlarge` instances (*i.e.,* 1 PS and 7 worker nodes) in Amazon EC2. Each instance is equipped with 32 vCPUs (2.7 GHz Intel Xeon E5-2686 v4 Broadwell), 2 GPUs (NVIDIA Tesla M60 GPU, each is equipped with 2048 parallel processing cores and 8 GB GPU memory), 244 GB memory, and varying network bandwidth from 1 Gbps to 10 Gbps.

DDNN training workloads and datasets: We choose four representative DNN models including ResNet18, ResNet50, ResNet152, and Inception-v3 trained with the ImageNet dataset [14]. The `batch size` is set as 16, 32, or 64. In particular, *Prophet* requires a lightweight job profiling process to collect the DDNN training performance data (*i.e.,* the size, generation time, and priority information of gradients as discussed in Sec. 4.1) in the first 50 iterations.

Baselines and metrics: We compare the performance of *Prophet* with that of the default MXNet and ByteScheduler [21] as well as P3 [10]. The metrics include: (1) the model training rate (*i.e.,* the number of training samples per second), (2) the GPU utilization and network uplink/downlink throughput, and (3) the wait time of each gradient data and start time of forward propagation. As the auto-tuning process of ByteScheduler can degrade the DDNN training performance for a number (*e.g.,* 1,000) of iterations (as discussed in Sec. 2.2), we simply implement ByteScheduler using BytePS [12] with a default `credit size`. In addition, we set the partition size of P3 as 4 MB.

5.2 Effectiveness of *Prophet*

Can *Prophet* improve DDNN training performance? We first examine the effectiveness of *Prophet* by comparing the DDNN training performance achieved by *Prophet* and the state-of-the-art strategy (*i.e.,* ByteScheduler [21]). As shown in Fig. 8, *Prophet* can significantly improve the training rate by 10% – 40% compared with ByteScheduler, for different DNN models and batch sizes. This is because *Prophet* explicitly leverages the *stepwise pattern* to reschedule gradient transfer by accurately predicting the block time interval and obtain the network bandwidth of workers. Accordingly, *Prophet* can effectively transfer an appropriate number of gradients within the desired time interval, and thus guarantee a high GPU and network resource utilization while retaining a timely gradient preemption.

Why *Prophet* can improve the DDNN training performance? We further look into the start time and end time of each gradient transfer with different network communication scheduling strategies. As depicted in Fig. 11, *Prophet* can significantly reduce both the wait time of gradient transfer and the transfer time of gradients, as compared with the default MXNet and ByteScheduler. Our experiment logs reveal that each gradient data takes long time (*i.e.,* 446

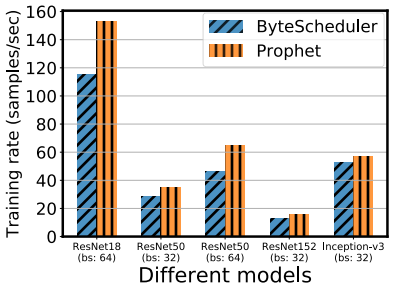


Figure 8: Comparison of training rate of representative DNN models with *Prophet* and ByteScheduler.

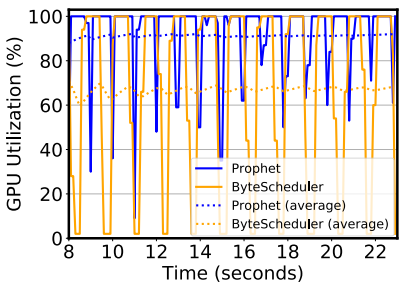


Figure 9: GPU utilization of a worker node over time during the training process of ResNet50.

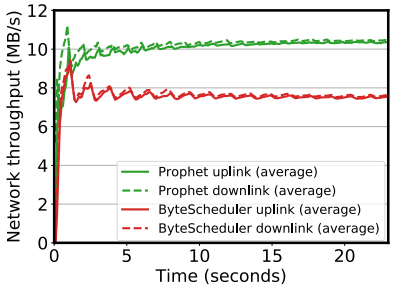


Figure 10: Network throughput of a worker node over time during the training process of ResNet50.

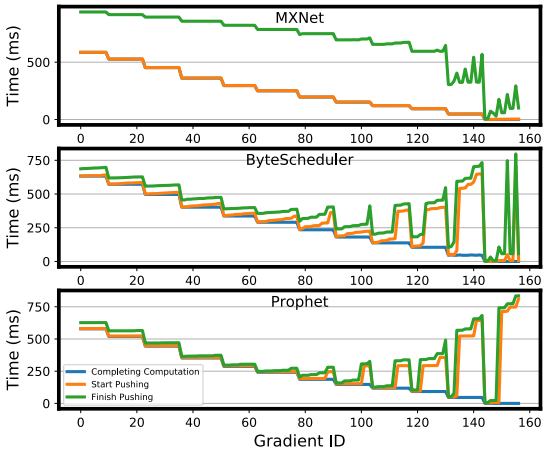


Figure 11: Start time and end time of gradient transfer during the training process of ResNet50 model with MXNet, ByteScheduler, and *Prophet*.

ms in average) to complete network transmission in MXNet, while ByteScheduler and *Prophet* can shorten the gradient transmission time to 135 ms and 125 ms, respectively, by re-scheduling the transfer order of gradients. Taking the network transfer of `gradient 30` as an example, MXNet waits 0.787 ms before transmission and spends 440.065 ms for transmission. ByteScheduler waits 10.359 ms and spends 56.357 ms for transmission. *Prophet* only waits 3.207 ms and spends 22.710 ms for transmission. The average wait time of gradient transfer with *Prophet* is only 26 ms, compared with 67 ms achieved with ByteScheduler, especially for the high-priority gradients (*i.e.,* `gradient 0` - `gradient 80` as shown in Fig. 11).

In the following, we proceed to analyze our experiment results above using three metrics: GPU utilization, network throughput of workers, and start time of forward propagation for each gradient.

GPU utilization: As *Prophet* allows the forward propagation to begin as early as possible, the GPU idle time during the DDNN training can be significantly reduced. As shown in Fig. 9, we observe that *Prophet* can improve the average GPU utilization from 67.85% achieved by ByteScheduler to 91.15%. Meanwhile, we observe that there is a periodical and sharp decrease in GPU utilization over time for both strategies. This indicates the GPU computational wait time cannot be reduced to zero even with *Prophet*, but our *Prophet* still significantly improves the GPU resource utilization compared with ByteScheduler over iterations.

Network uplink/downlink throughput: Prophet utilizes the *stepwise pattern* to assemble them into *gradient blocks*, so that gradients

Table 2: Comparison of training performance of ResNet50 model with *Prophet* and ByteScheduler under different network bandwidth conditions.

Worker bandwidth limit (Mbps)	Rate of *Prophet* (samples/sec)	Rate of ByteScheduler (samples/sec)	Rate of P3 (samples/sec)
1, 000	27.7	25.9	25.16
2, 000	47.9	39.09	37.69
3, 000	60	44	51.22
4, 000	67.06	50.5	64.34
4, 500	69.29	54.14	67.83
6, 000	69.5	70	68.93
10, 000	70.6	71.1	72.83

can be transferred to utilize more network bandwidth resources. As both *Prophet* and ByteScheduler can timely transfer high-priority gradients and thus achieve good overlapping, the network utilization with *Prophet* and ByteScheduler is higher than that with default MXNet. As shown in Fig. 10, *Prophet* achieves higher average network throughput (*i.e.,* 10.3 MB/s) by 37.3% compared with ByteScheduler (*i.e.,* 7.5 MB/s). This is because *Prophet* use *stepwise pattern* to utilize the time interval, and guarantee the priority of gradient transfer. The network throughput of workers fluctuates over time because the block time interval and the size of scheduled gradients are too small to maintain a high network transfer rate.

Start time of forward propagation: Prophet ensures that the higher-priority gradients complete their transmission as early as possible and accordingly the forward propagation can be activated earlier. Accordingly to our experiment logs, we set the timing (*i.e.,* 0 ms) as the start time of the 60-th iteration when training ResNet50. We observe that *Prophet* can take a shorter time to finish one iteration compared with ByteScheduler: *Prophet* starts its 61-th iteration at 856.796 ms, while ByteScheduler starts its 61-th iteration at 1, 416 ms. In the first 15 seconds in our experiment, *Prophet* can complete iterations $60 - 74$, while ByteScheduler only completes iterations $60 - 71$. Such a performance improvement can be aggregated as the number of iterations increases.

5.3 Robustness of *Prophet*

Can *Prophet* effectively work under different bandwidth conditions? We train ResNet18 (batch size 64) with ImageNet under a varying network bandwidth environment. Under 10 Gbps network bandwidth, the default MXNet, P3 [10] and *Prophet* all achieve approximately 220 samples/sec, because the optimization space of

Table 3: Comparison of training performance of ResNet18 and ResNet50 model with *Prophet* and ByteScheduler using different batch sizes.

Model and batch size	Rate of *Prophet* (samples/sec)	Rate of ByteScheduler (samples/sec)	Performance improvement
ResNet18 (16)	32.46	29.06	11.6%
ResNet18 (64)	153	115	33%
ResNet50 (16)	14.44	14.22	1.5%
ResNet50 (32)	34.8	28.5	22%
ResNet50 (64)	60	44	36%

scheduling gradient transfer is marginal as high network bandwidth ensures all tasks can be completed in a short time. When the network bandwidth is limited to 3 Gbps, the default MXNet (*i.e.,* the FIFO strategy) only gets 110 samples/sec, and P3 (*i.e.,* the priority-based strategy) achieves 137 samples/sec, while *Prophet* achieves relatively higher DDNN training performance (*i.e.,* 153 samples/sec) by 11.7% – 39.1% compared with default MXNet and P3. Similarly, we compare the DDNN training rate achieved by *Prophet*, ByteScheduler and P3 using ResNet50 (batch size 64), as listed in Table 2. We observe that the training performance of P3 gets sharply worse while that of ByteScheduler and *Prophet* is gradually lowered down as the network bandwidth gets stringent. This is because the `partition size` of P3 fails to arbitrate the trade-off between gradient preemption and extra partition overhead, especially under the poor network bandwidth condition (1 – 3 Gbps). As compared to ByteScheduler, *Prophet* can improve the training performance by 6.9% – 36.4% in poor network conditions and achieve comparable performance in good network conditions. The rationale is that, *Prophet* leverages the *stepwise pattern* to adjust the scheduling of gradient transfer and thus activates the forward propagation earlier, thereby achieving a higher degree of overlapping between computation and communication than P3 and ByteScheduler in dynamic network conditions (*i.e.,* varying from 1 to 6 Gbps).

How batch size affects the effectiveness of *Prophet*? As listed in Table 3, we observe that *Prophet* outperforms ByteScheduler by 1.5-36% with various DNN models and batch sizes ranging from 16 to 64. This is because *Prophet* can leverage our observed characteristics of *stepwise pattern* to optimize the communication scheduling for different frameworks and models, as discussed in Sec. 2.2. Our experiment result also indicates that a larger batch size leads to a more significant performance improvement achieved by *Prophet*. This is because a larger mini-batch takes a longer time to compute the gradients, which inevitably makes the *stepwise pattern* more obvious and thus prolongs the time interval among blocks. As a result, *Prophet* can leverage such a prolonged block time interval to optimize the communication scheduling, and fully utilize the GPU and network resources during DDNN training.

Can *Prophet* work in heterogeneous environments? We limit the network bandwidth of one worker node to 500 Mbps, and compare the training rate of the default MXNet, ByteScheduler, and *Prophet*. We observe that *Prophet* and ByteScheduler achieve the DDNN training rate of 26.4 samples/sec and 25.8 samples/sec, respectively, compared with the default MXNet (15.09 samples/sec). The results above indicate: (i) Both *Prophet* and ByteScheduler outperform the default MXNet in heterogeneous environments because assembling gradients into *blocks* and starting the forward

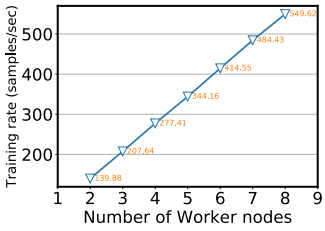


Figure 12: Training performance of ResNet50 with *Prophet* by varying the number of workers from 2 to 8.

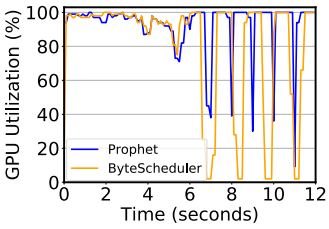


Figure 13: Negligible performance overhead of *Prophet* on GPU utilization.

propagation earlier can still work well in heterogeneous environments. (ii) *Prophet* slightly improves the training performance by 2.3% compared with ByteScheduler in heterogeneous environments. The rationale is that, the network transfer rate in a heterogeneous environment is inevitably limited by the node with the lowest bandwidth (*i.e.,* 500 Mbps), which significantly reduces the optimization space of communication scheduling. In particular, *Prophet* can work with the existing approach [30] which designs a rational model computation functions to assign more computation and transmission tasks to workers with good performance, so as to improve the training performance in heterogeneous environments.

5.4 Runtime Overhead of *Prophet*

The runtime overhead of *Prophet* mainly lies in the job profiling (*e.g.,* obtaining the gradient generation time in first 50 iterations) and monitoring the network bandwidth of workers during the DDNN training process, as *Prophet* does not require tuning hyperparameters (*i.e.,* `credit size`). Specifically, the job profiling overhead for Inception-v3 with `batch size` 32, ResNet50 with `batch size` 64, and ResNet152 with `batch size` 32 are 7 seconds, 9.5 seconds, and 24.7 seconds, respectively. Such job profiling overhead (*i.e.,* within one minute) is negligible compared with over thousands of iterations required by a typical DNN training job. As shown in Fig. 13, the GPU utilization of *Prophet* in the early stage (*i.e.,* from 2 – 6 seconds) of DNN training is slightly lower than that of ByteScheduler. As the DDNN training continues, however, *Prophet* takes shorter time to complete each iteration without auto-tuning overhead of `credit size`, which in turn improves the DDNN training performance. Moreover, Fig. 12 shows that the DDNN training performance of ResNet50 with *Prophet* is roughly linear to the number of workers, as the average training rate per worker slightly decreases from 69.94 samples/sec to 68.83 samples/sec when the number of workers increases from 2 to 8. Such an experimental result further identifies the negligible computation overhead of Alg. 1 in *Prophet*. Therefore, the runtime performance overhead of *Prophet* is practically acceptable for real-world DDNN training jobs.

6 RELATED WORK

6.1 Priority-based Communication Scheduling

To optimize the performance of DDNN training workloads, several recent works design priority-based communication scheduling strategies, such as P3 [10] and TicTac [8]. Specifically, P3 attempts to overlap the communication and forward propagation by slicing

the gradient data to seek for more potentials of overlapping. Similarly, TicTac [8] proposes a communication strategy to schedule the neural network *operations* according to the observation of high network variance during the process of parameter update. These two prior works rely on the blocking call of TCP protocol, and thus they can cause performance degradation due to the under-utilization of network bandwidth resources. In contrast, *Prophet* introduces the concept of *gradient blocks* by identifying the *stepwise pattern* for DDNN training, and we greedily transfer the gradient data in the form of small *blocks* through a lightweight job profiling, so as to mitigate the under-utilization of GPU and network resources.

Moreover, there lacks a theoretical study on communication scheduling in the literature, and we mathematically analyze how the scheduling of gradient transfer affects the DDNN training performance. A more recent work [27] categorizes the network data into two priority levels, and it brings remarkable training performance gains by implementing the network-level scheduling strategy on the network switches. To speed up the DDNN training on the All-Reduce architecture, PACE [2] preemptively schedules the all-reduce tensors based on the DAG information of a DNN model. The priority-based communication scheduling mechanism is still effective in a multi-tenant environment [25]. *Prophet* can works together with these strategies above to improve DDNN training performance.

6.2 Improving Resource Utilization of Network Bandwidth

The network resource utilization can be degraded by two factors: *long wait time for computation* and *high communication overhead*. *First,* many recent works propose to overlap computation and communication to reduce the computation wait time. For example, Poseidon [29] designs a wait-free backward propagation (WFBP) to maximize the overlapping between computation and communication in one iteration, by clearly defining the dependency among layers. *Second,* several recent works identify that the high communication overhead is commonly determined by the gradient transfer granularity. Accordingly, MG-WFBP [24] merges an appropriate number of gradient transfer tasks into single one and creates more chances to overlap computation and communication. ByteScheduler [21] introduces a `credit size` to maintain network resource utilization, which is considered as a typical trade-off between the preemption of gradient transfer and network resource utilization. Different from the static and auto-tuned `credit size` for DDNN training in ByteScheduler, *Prophet* leverages a *dynamic gradient block size for each iteration* to speed up DDNN training. A more recent work [28] batches the model parameters to reduce the start time of forward propagation during the `pull` process. Different from the prior works above, *Prophet* seeks to identify an optimal number of gradient transfer by utilizing the time interval between gradient blocks, so as to greedily reduce the waiting time of GPU resources and start the forward propagation as early as possible.

There have also been a number of works focus on optimizing the parameter synchronization process to improve the network resource utilization. Specifically, relaxing the strict synchronization [9] and moving gradient aggregation inside the network can alleviate the communication overhead and thus increase the network throughput. For example, R2SP [3] mitigates the uneven available network bandwidth of workers over time. DSSP [30] solves the dynamic tuning of the staleness threshold of SSP. LBBSP [4] deals with the worker stragglers in heterogeneous (*i.e.,* non-dedicated) environments. *Prophet* mainly works in the PS architecture using BSP (*i.e.,* Bulk Synchronous Parallel). While these works above mainly optimize the *application-level* computation, *in-network aggregation* [22] can be deployed to alleviate the communication bottleneck using programming switches. For instance, ATP [15] implements the co-design of switch logic (for gradient aggregation) and the end-host networking stack to improve DDNN training performance. *Prophet* can work with these prior techniques above on optimizing either computation or communication, so as to improve the resource utilization of DDNN training clusters.

7 CONCLUSION AND FUTURE WORK

To improve the GPU and network resource utilization of DDNN training clusters, this paper presents the design and implementation of *Prophet*, a *simple yet effective* predictable communication scheduling strategy that decides the transfer order and transfer start time of gradients for DDNN training jobs. Specifically, *Prophet* first obtains the *stepwise pattern* of gradient transfer start time according to a lightweight job profiling. Based on such a pattern, *Prophet* leverages the monitored network bandwidth and the profiled time interval among gradients to assemble the *gradient blocks* (*i.e.,* to decide which gradients should be transferred within the *block* time interval). Through the construction of *gradient blocks*, *Prophet* greedily ensures that `gradient 0` can start its network transfer once the computation of backward propagation is completed, so that the GPU and network resource utilization can be improved during the process of `pull` operations. We implement a prototype system of *Prophet* based on BytePS and conduct extensive prototype experiments by training representative DNN models on Amazon EC2. Our experiment results show that *Prophet* can improve the DDNN training performance by up to 40% compared with the state-of-the-art priority-based communication scheduling strategies, yet with negligible runtime performance overhead.

As our future work, we plan to extend *Prophet* in the following directions: (1) validating the *stepwise pattern* of gradient transfer with the ASP (*i.e.,* Asynchronous Parallel) model, and (2) examining the effectiveness of *Prophet* on more types of cloud instances and GPU hardwares (*e.g.,* `p3` and `p4` EC2 instances).

ACKNOWLEDGMENTS

The corresponding author is Fei Xu (fxu@cs.ecnu.edu.cn). 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, in part by the Natural Science Foundation of Shanghai under grant NO.21ZR1419900, and in part by the Tencent Corporation. Li Chen's work was supported in part by the Louisiana Board of Regents under Contract Numbers LEQSF(2019-22)-RD-A-21 and LEQSF(2021-22)-RD-D-07, and in part by National Science Foundation under Award Number OIA-2019511.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. of USENIX OSDI*. 265–283.

[2] Yixin Bao, Yanghua Peng, Yangrui Chne, and Chuan Wu. 2020. Preemptive All-reduce Scheduling for Expediting Distributed DNN Training. In *Proc. of IEEE INFOCOM*. 626–635.

[3] Chen Chen, Wei Wang, and Bo Li. 2019. Round-Robin Synchronization: Mitigating Communication Bottlenecks in Parameter Servers. In *Proc. of IEEE INFOCOM*. 532–540.

[4] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li. 2020. Semi-Dynamic Load Balancing: Efficient Distributed Learning in Non-Dedicated Environments. In *Proc. of ACM SoCC*. 431–446.

[5] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting Distributed Synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274* (2015).

[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large Scale Distributed Deep Networks. In *Proc. of NIPS*. 1223–1231.

[8] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. 2019. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *Proc. of MLSys*.

[9] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Proc. of NIPS*. 1223–1231.

[10] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based Parameter Propagation for Distributed DNN Training. In *Proc. of MLSys*.

[11] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proc. of USENIX ATC*. 947–960.

[12] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *Proc. of IEEE OSDI*. 463–479.

[13] Janis Keuper and Franz-Josef Preundt. 2016. Distributed Training of Deep Neural Networks: Theoretical and Practical Limits of Parallel Scalability. In *Proc. of IEEE MLHPC*. 19–26.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proc. of NIPS*. 1097–1105.

[15] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-Tenant Learning. In *Proc. of USENIX NSDI*. 741–761.

[16] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. 2014. On Model Parallelization and Scheduling Strategies for Distributed Machine Learning. In *Proc. of NIPS*. 2834–2842.

[17] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proc. of USENIX OSDI*. 583–598.

[18] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2018. Parameter Hub: a Rack-Scale Parameter Server for Distributed Deep Neural Network Training. In *Proc. of ACM SoCC*. 41–54.

[19] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. 2019. PaddlePaddle: An Open-Source Deep Learning Platform from Industrial Practice. *Frontiers of Data and Computing* 1, 1 (2019), 105–115.

[20] Peter Mattson, Paulius Micikevicius, Vijay Janapa Reddi, David Patterson, Christine Cheng, Guenther Schmuelling, Cody Coleman, Hanlin Tang, Greg Diamos, Gu-Yeon Wei, David Kanter, and Carole-Jean Wu. 2020. MLPerf Training Benchmark. *IEEE Micro* 40, 2 (2020), 8–16.

[21] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proc. of ACM SOSP*. 16–29.

[22] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *Proc. of USENIX NSDI*. 785–808.

[23] Alexander Sergeev and Mike Del Balso. 2018. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[24] Shaohuai Shi, Xiaowen Chu, and Bo Li. 2019. MG-WFBP: Efficient Data Communication for Distributed Synchronous SGD Algorithms. In *Proc. of IEEE INFOCOM*. 172–180.

[25] Raajay Viswanathan, Arjun Balasubramanian, and Aditya Akella. 2020. Network-Accelerated Distributed Machine Learning for Multi-Tenant Settings. In *Proc. of ACM SoCC*. 447–461.

[26] Qiang Wang, Shaohuai Shi, Canhui Wang, and Xiaowen Chu. 2020. Communication Contention Aware Scheduling of Multiple Deep Learning Training Jobs. *arXiv preprint arXiv:2002.10105* (2020).

[27] S. Wang, D. Li, and J. Geng. 2020. Geryon: Accelerating Distributed CNN Training by Network-Level Flow Scheduling. In *Proc. of IEEE INFOCOM*. 1678–1687.

[28] Shaoqi Wang, Aidi Pi, and Xiaobo Zhou. 2019. Scalable Distributed DL Training: Batching Communication and Computation. In *Proc. of AAAI*, Vol. 33. 5289–5296.

[29] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *Proc. of USENIX ATC*. 181–193.

[30] Xing Zhao, Aijun An, Junfeng Liu, and Bao Xin Chen. 2019. Dynamic Stale Synchronous Parallel Distributed Training for Deep Learning. In *Proc. of IEEE ICDCS*. 1507–1517.