Prophet: Speeding up Distributed DNN Training with Predictable Communication Scheduling

Zhenwei Zhang¹, Qiang Qi², Ruitao Shang³, Li Chen†, Fei Xu*

Email: ¹10165102154@stu.ecnu.edu.cn ²fxu@cs.ecnu.edu.cn

¹,²,³,*East China Normal University

†University of Louisiana at Lafayette
Background

Data Transmission in Distributed Deep Learning

➢ In each training iteration, amounts of data (e.g., gradients/parameters, intermediate data) should be transferred across devices.

➢ Parameter Server, a centralized communication architecture is typically used for data parallelism.
Background
Bottleneck and Related Works

➢ The communication can block the computation of Workers.

➢ Several related works design priority-based communication scheduling strategies.
  ➢ e.g., P3 is dedicated to overlapping the backward propagation with the gradient push process.
Motivation
Existing Problems of Related Works

- The GPU utilization is still low in some situations.
- The GPU utilization can dramatically decrease to zero (i.e., totally idle) during the pull operation of model parameters.

The slow network transmission makes the GPUs fail to timely acquire the model parameters and thus delays the computation (i.e., forward propagation).
Motivation
Existing Problems of Related Works

➢ Non-negligible performance **overhead** of state-of-the-art priority-based communication scheduling strategies.

![Graph showing training rate vs. partition size](image)

**Training rate (samples/sec)**
- **Partition size (KB)**
  - 0
  - 100
  - 200
  - 300
  - 400

![Graph showing training rate and credit size vs. iteration](image)

**Training rate (samples/sec)**
- **Iteration**
  - 0
  - 200
  - 400
  - 600
  - 800
  - 1000

**Credit size (MB)**
- 3.00
- 6.25
- 9.50
- 12.75

The smaller size of partitions dramatically decreases the DDNN training rate.
The training rate of the ResNet50 model fluctuates rapidly.
Motivation
Stepwise Pattern

➢ Communication characteristic of gradient transfer follows a stepwise pattern over time.

➢ The gradient data requires aggregation before transmission, which can be considered as the main cause of stepwise pattern.

➢ Such a pattern is independent of the DDNN training frameworks, DNN models, datasets, and hardware architectures.
Motivation

Prophet

➢ 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 the network bandwidth resources without blocking the higher-priority gradients, so that the critical gradients (e.g., gradient 0) can be transferred as fast as possible, and with negligible runtime overhead.
Motivation
An Illustrative Example

- Default *MXNet* transmits gradients with the default *FIFO* order;
- *P3* slices the gradients into small partitions to ensure a timely *preemption*, but with non-negligible overhead;
- *ByteScheduler* configures the credit size to avoid the partition overhead, while keeping a relatively high preemption rate;
- *Prophet* introduces the concept of *gradient blocks* by identifying the stepwise pattern for DDNN training, greedily transferring the gradient data through a lightweight job profiling.

*The notation indicates the start time of forward propagation*

*The notation denotes the assembled gradients that are transferred at a time*
Modeling Illustration of DDNN Training Time

\[ T_{all} = \sum_{i \in \mathcal{X}} T_{bp}^{(i)} + \sum_{i \in \mathcal{X}} T_{fp}^{(i)} + T_{communication} - T_{overlap} + T_{wait}, \]

- the total idle time of GPU resources during the entire training process
- the time cost for backward propagation
- the time cost for forward propagation
Modeling Illustration of DDNN Training Time

\[ T_{\text{wait}} = \sum_{i \in X, i \neq 0} \left( u(i) - P(i-1) \right) + (u(0) - c(0)) \]

- completion time of parameter update
- completion time of forward propagation
- part of GPU idle time

*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 \( u(i) - p(i-1) \).*

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

- GPUs are in the working state during the backward propagation until gradient 0 is generated at \( c(0) \)
- GPUs are back to the working state when gradient 0 updates its parameters at \( u(0) \)
- 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
Modeling
Problem Formulation

➢ How to schedule the gradient transfer time to minimize $T_{\text{wait}}$ (and thus to minimize $T_{\text{all}}$) to maximize the GPU resource utilization.

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

s.t.

$t(i) \geq c(i)$, 

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

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

can only be pushed after it is generated

avoids the concurrent gradient transfer

during the forward propagation, un-transmitted gradients should be transmitted in the order of priority
The gradients should be transferred before any higher-priority gradient is generated in the backward propagation.

\[ t^{(i)} + E^{(i)} \leq c^{(k)}, \quad \forall k \in X < i, \quad \text{when} \quad t^{(i)} \leq c^{(0)}. \]

This implies that we can take advantage of the stepwise pattern (e.g., block time interval) to find the optimal gradient transfer.

* Gradients are transferred in the order of priorities in the forward propagation.
Algorithm Design Principles

➢ 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.

➢ **Take advantage of our observed stepwise pattern.**
Algorithm Design Pseudocode

determine the start time of gradient transfer during both the backward and forward propagation

greedily assemble each gradient if it can be transmitted before any higher-priority gradient is generated

assemble the gradient into a gradient block

gradients are transferred in the order of priority in forward propagation

ensure critical gradients are transferred as fast as possible

---

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 \( i \in X \).
Output: Start time \( t(i) \) of gradient transfer (i.e., the start time to push each gradient \( i \in X \).

1: Initialize: Estimated gradient transmission time \( E(i) \leftarrow s(i) / B \) (by Eq. (5) and Eq. (10)), and the expected transfer time interval \( A(i) \leftarrow \min \{E(i) - T_{used}\} \), \( t(i) \in X \).
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 \( \emptyset \) then
5: if the current time is in the backward propagation then
6: \( T_{used} \leftarrow T_{used} + c(p) \);
7: Set \( q \) as the currently highest-priority gradient that is ready to be transferred;
8: end while;
9: \( T_{used} \leftarrow T_{used} + c(p) \);
10: End if;
11: \( T_{used} \leftarrow T_{used} + E(q) \);
12: Determine the start time of gradient transfer during both the backward and forward propagation.
13: while the gradient \( q \) can be greedily transferred within the expected time interval \( A(q) - T_{used} \) do
14: Assemble the gradient \( q \) into a gradient block and update \( t(q) \leftarrow T_{used} + c(p) \);
15: Set \( t(p) \) as the earliest available scheduling time \( t_{next} \);
16: Update the next available time \( t_{next} \leftarrow t_{next} + E(p) \);
17: end while;
18: Set \( t(0) \) as the generation time \( c(0) \) of gradient \( 0 \);
19: Update the next available time \( t_{next} \leftarrow t(0) + E(0) \);
20: end while
21: return the start time \( t(i) \) of gradient transfer;
```
Algorithm Design
Implementation of *Prophet*

- *Prophet* is implemented based on the abstraction layer of *BytePS*.
- designed to support multiple frameworks

Pre-trains the DNN model to obtain the gradient information

Periodically acquires the available network bandwidth

Leverages the gradient and network bandwidth information to find the optimal start time of gradient transfer
Evaluation
Experimental Setup & Metrics

➢ **Eight g3.8xlarge** instances (i.e., 1 PS and 7 worker nodes) in Amazon EC2
  ➢ 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
  ➢ varying network bandwidth from 1 Gbps to 10 Gbps.

➢ **Four** representative DNN models
  ➢ ResNet18
  ➢ ResNet50
  ➢ ResNet152
  ➢ Inception-v3

➢ **Three** evaluation metrics
  ➢ The model **training rate**
  ➢ The **GPU utilization and network uplink/downlink throughput**
  ➢ The **wait time** of each gradient data and **start time** of the forward propagations
Evaluation
Effectiveness of Prophet

- Prophet can significantly improve the training rate by 10% – 40% compared with ByteScheduler, for different DNN models and batch sizes.
Evaluation Effectiveness of *Prophet*

- *Prophet* can significantly reduce both the wait time of gradient transfer and the transfer time of gradients.
Evaluation
Robustness of Prophet

➢ Under different batch sizes:

<table>
<thead>
<tr>
<th>Model and batch size</th>
<th>Rate of Prophet (samples/sec)</th>
<th>Rate of ByteScheduler (samples/sec)</th>
<th>Performance improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet18 (16)</td>
<td>32.46</td>
<td>29.06</td>
<td>11.6%</td>
</tr>
<tr>
<td>ResNet18 (64)</td>
<td>15</td>
<td>115</td>
<td>33%</td>
</tr>
<tr>
<td>ResNet50 (16)</td>
<td>14.44</td>
<td>14.22</td>
<td>1.5%</td>
</tr>
<tr>
<td>ResNet50 (32)</td>
<td>34.8</td>
<td>28.5</td>
<td>22%</td>
</tr>
<tr>
<td>ResNet50 (64)</td>
<td>60</td>
<td>44</td>
<td>36%</td>
</tr>
</tbody>
</table>

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.

➢ Under different bandwidth conditions:

Prophet achieves relatively higher DDNN training performance by 11.7% - 39.1% compared with default MXNet and P3.

➢ In heterogeneous environments:

Both Prophet and ByteScheduler outperform the default MXNet in heterogeneous environments.

Prophet slightly improves the training performance by 2.3% compared with ByteScheduler in heterogeneous environments.
Summary

➢ 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.

➢ *Prophet* leverages our observed stepwise pattern of gradient transfer start time to make the forward propagation start as early as possible to greedily reduce the waiting (idle) time of GPU resources during the DDNN training process.

➢ *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.
Thank You