SketchDLC: A Sketch on Distributed Deep Learning Communication via Trace Capturing*

YEMAO XU, DEZUN DONG, WEIXIA XU, and XIANGKE LIAO, National University of Defense Technology, China;

With the fast development of deep learning, the communication is increasingly a bottleneck for distributed workloads, and a series of optimization works have been done to scale out successfully. We intend to optimize the functionalities of network devices to improve the communication efficiency. These optimizations can be implemented directly in practical hardware, or evaluated through network simulation, and simulation is our choice. Before the conduction of simulation, an accurate communication measurement is necessary, as it is the basis for accurate simulation. Therefore, we propose to capture the deep learning communication (DLC) trace to achieve the measurement.

To the best of our knowledge, we make the first attempt to capture the communication trace for deep learning training. In this paper, we firstly provide detailed analyses about the communication mechanism of MXNet, which is a representative framework for distributed deep learning. Secondly, we define the DLC trace format to describe and record the communication behaviors. Thirdly, we present the implementation of method for trace capturing. Fourthly, we verify the communication mechanism by providing a glimpse of the trace files. Finally, we make some statistics and analyses about the distributed deep learning communication, including communication pattern, overlap ratio between computation and communication, synchronization overhead, update overhead etc. Both the statistics and analyses are based on the trace files captured in a cluster with 4 machines. On the one hand, our trace files provide a sketch on the deep learning communication, which contributes to understanding the communication details. On the other hand, the captured trace files can be used for conducting our simulation experiments, as they record the communication behaviors of each node.

CCS Concepts: • **Networks → Network measurement**; Network simulations;

Additional Key Words and Phrases: distributed system, deep learning, parameter server, communication trace.

ACM Reference Format:
Yemao Xu, Dezun Dong, Weixia Xu, and Xiangke Liao. 2010. SketchDLC: A Sketch on Distributed Deep Learning Communication via Trace Capturing. *ACM Trans. Arch. Code Optim.* *, *, Article * (March 2010), 22 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

As a class of machine learning (ML) approaches, deep learning (DL) has achieved impressive success over a wide spectrum of tasks in the past few years, ranging from speech processing, image recognition, breast cancer detection [42] to self driving cars [39]. There are two main driving forces behind the momentum that DL has gained [4], first is the public availability of versatile training data sets like ImageNet [11] and CIFAR [26], while the notable advances in computing capability turns out to be the second one [9]. The resurgence of DL has also triggered the developments of DL frameworks like MXNet [7], TensorFlow [2], Caffe [20], Microsoft CNTK [10] and Theano [5]. However, Research efforts have been mostly concentrated on designing better and deeper neural

*This is a new article, not an extension of a conference paper.

Authors' address: Yemao Xu; Dezun Dong; Weixia Xu; Xiangke Liao, National University of Defense Technology, 109 Deya Rd, Changsha, Hunan, 410073, China; email:{xuyemaovip,dong,xkliao}@nudt.edu.cn.

XXXX-XXXX/2010/3-ART* $15.00
https://doi.org/0000001.0000001

networks (DNN) to improve the accuracy of complex tasks. DNNs like GoogleNet [40], AlexNet [27] and ResNet [19] have a large number of parameters to be trained, which call for significant amount of training time on a single GPU equipped machine, and such large models also need quantities of data to ensure good generalization performance on unseen test data. The increase of model size and training data volume drive the demand for distributed deep learning training.

To implement the distributed training, different DL frameworks utilize different mechanisms to accomplish the distributed communication, such as the ZeroMQ [48] in MXNet, gRPC in TensorFlow [2], NCCL in Caffe-MPI 2.0 [1], Gloo in Caffe2 [14] and some other communication libraries. Though distributed training strategy overcomes the limitations of computing capability and memory space in a single node, the communication of large amounts of parameters and gradients brings about great communication overheads. Firstly, the steadily increasing computational capability of general accelerators and the developments of specialized accelerators lead to the updates of parameters more frequently, making the large matrices of parameter and gradients saturate the network bandwidth more quickly. Besides, the communication pattern of DL algorithms also leads to the gradient and parameter matrices to be delivered in bursts. The combination of frequent updates and bursts contribute to network congestion. The network congestion will degenerate the distributed training performance by generating more communication overheads, so the optimizations on communication overhead is inevitable to scale out successfully.

A series of works have been done to reduce the communication overhead. As to the majorization designs in hardware, DGX-1 [12] and DGX-2 [13] by NVIDIA improve the communication efficiency greatly via the use of NVLink and NVSwitch technologies; Tensor processing unit (TPU) by Google [22] reduces the network traffic via cutting down the accuracy of gradients; Song et al. designed an efficient engine [17] to support the inference of compressed neural network, which optimize the communication indirectly, etc. Optimizations from a software perspective include three main aspects, namely overlapping the computation and communication [4, 49], reducing the communication times [15, 46, 47] and reducing the network traffic [9, 18, 33, 43]. The overlapping strategy eliminates the unnecessary waiting overhead to make full use of the network and computation resources. Researchers increase the batch size greatly to reduce the iteration numbers of each epoch, thus reducing the communication times. The most effective method is the network reduction, methods of pruning the redundant connections, weight sharing and deep gradients compressing can cut down the network traffic to a large extend.

We argue that the distributed communication efficiency can be improved by optimizing the function of network device, which is different from the works mentioned above. Before the implementation in real hardware, we decide to evaluate the performance through network simulation. A comprehensive measurement of the deep learning communication is necessary before the simulation. Besides, carrying out the simulation calls for two conditions: first is a model of the system which contains the main characteristics of the system to be evaluated, or just the simulator. The simulator will implement the operations of the evaluated system, and we develop a in-house one. The second condition goes to a description of the communication behaviors in the system, which is provided to the simulator to perform the simulation [3]. As to network simulation, the description turns out to be the network traffic, also known as communication trace or workload, representing the message exchanging operations by the simulated processes using the network. The purpose of communication trace is to make the simulation as similar as possible to the communication pattern in real system network. Synthetic traffic patterns come up to our mind firstly when regarding to the generation of communication trace. This type of trace suits the system with simple communication pattern, whose key features can be clear expressed by mathematical expressions. However, the majority of real applications' communication trace cannot be easily described by mathematical expressions. The synthetic traces do not consider the interactions generated by real applications, while the communications of DL programs do have dependency relationships among the distributed nodes. Therefore, the communication traces used for network simulation of DL programs should be based on the traffic patterns produced by real applications.

Existing works on trace capturing involve both single node and distributed cluster. Paper [41] proposes a self-related trace model to capture the trace for NoC (Network-on-chip) simulator while papers [3, 6, 36] present the mechanisms to get communication traces of MPI parallel applications. [36] makes use of MPE (Multi-Processing Environment) [23], which is a set of libraries and tools used for generating and analyzing traces of parallel applications, to get the trace files. Paper [6] comes up with the time-independent trace replay framework to capture the time-independent traces. In paper [3], it raises the VEF trace framework to tackle the problem of timestamps in recording traces. In addition to the purpose for simulation, trace files can also be used for analyzing the communication characteristics, paper [24] provides a deep analysis about the MPI communication patterns to find out which parts call for more optimizations based on trace files. In fact, MPI is widely used for distributed communication in peer-to-peer structure, but the already established trace formats share the basic principle that the event records are sorted by time, making it unsuitable for reflecting the dependency based on casual relationships. We aim at providing a readable communication trace, which can present the casual relationships of different communication operations. Besides, with such a trace, we can figure out the overlap ratio of communication and computation, as well as analyze the communication pattern and various types of overheads. To satisfy these requirements, a new trace format is needed.

With the motivations above, we take great efforts to capture the communication traces of DL applications based on MXNet, an efficient and flexible library for DL with the parameter server (PS) structure. On the one hand, the captured trace is a measurement of the network behaviors, which contributes to a further understanding of the distributed communication. On the other hand, we can use the communication traces to launch network simulation experiments. The communication operations in MXNet are implemented through the invoking of communication APIs in ZeroMQ, allowing us to capture the trace from the framework level and the communication library level. We chose the former one for two reasons. First, we can grasp concise dependency relationships. Secondly, the memory occupied by the trace files is much smaller.

We highlight the challenges that need to be settled for the sake of providing a comprehensive and detailed communication trace of PS structure, using the MXNet. Firstly, how the parameters and gradients are transmitted during the training. Secondly, how to identify the communication operations that belong to the same iteration. Thirdly, how to describe the dependency relationships between various kinds of communication operations. Finally, what is the information that should be contained in the trace format to present comprehensive traces.

In order to tackle the challenges above, we carry out various attempts, and finally we come up with *SketchDLC*, which is a sketch on deep learning communication via trace capturing, it contains the following contributions. In the first place, we provide some analyses about the communication mechanism of MXNet. Secondly, we define a trace format to describe and record the communication operations. Thirdly, we implement the method of trace capturing in MXNet and confirm the communication mechanism with the captured trace files. At last, we make some statistics about the communication features and analyses about various types of overheads.

The rest of this paper is organized as follows. Section 2 provides some introductions about the distributed DL and the reason for trace capturing. Section 3 gives detailed insights on the communication mechanism of MXNet. Section 4 presents the format of DDL trace and the implementation of trace capturing method. Section 5 describes the analysis on obtained communication traces. Section 6 talks about the related work and section 7 concludes.

2 BACKGROUND & MOTIVATION

In this section, we first provide brief introductions about the distributed frameworks for large-scale DL and synchronous distributed SGD. Then we describe the in-network computing technique, which promotes the need for capturing DLC trace.

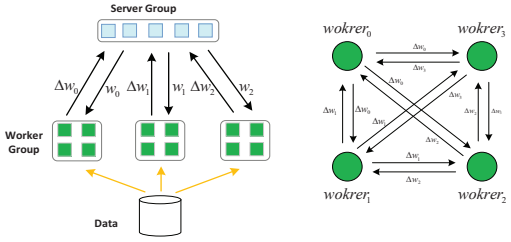

Fig. 1. Parameter server and peer-to-peer structures

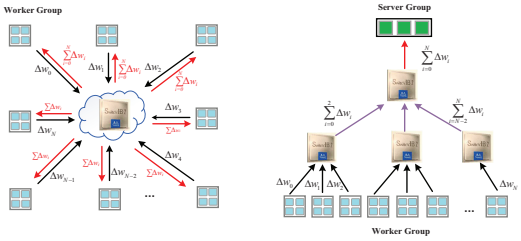

Fig. 2. In-network computing for peer-to-peer and PS

2.1 Distributed framework for DL

With the fast growth in data volume and model size, distributed optimizations turn out to be a prerequisite for solving large scale DL problems. A single machine can no longer solve these problems efficiently due to the limited computing capability and memory space [32]. To implement the distributed training, two types of distributed training frameworks, the parameter server structure and the peer-to-peer structure (Figure 1), are proposed.

Parameter Server Structure (PS). A parameter server is a distributed shared memory that offers systematic-abstraction of iteration-convergent algorithms in data-parallel and model-parallel distributed ML [49]. There are three types of nodes in the PS structure, the scheduler node, server node and worker node, these nodes take on specialized but different roles. The scheduler is responsible for setting up connections and tasks assignment, while the servers maintain the parameters and workers carry on the computing tasks. All the server nodes make up the server group and all the worker nodes come into being different worker groups, each worker group runs only one application at a time. A server node keeps a segment of the global parameters, and the worker nodes access the server group through network communication under the client-server scheme. The training of DL applications includes 4 main steps: (1) Each worker calculates the gradients with their own training data segmentation; (2) Workers transfer their gradients to remote servers; (3) Each server updates their own parameter partition using the aggregated gradients; (4) Workers get the updated parameters from the remote servers and launch the next iteration.

Peer-to-Peer Structure (P2P). There is no sever node in this kind of structure, every worker node keeps all the parameters, accomplishes the computing tasks and communication operations. Each worker will receive the gradients from other workers, then updates the local parameters. The communication overhead of a naive peer-to-peer topology is $O(P^2)$ (where P is the number of workers), which inhibits its scalability to large scale clusters when P goes up to several thousand [32]. Compared with parameter server framework, it is unnecessary to regain the parameters from server nodes to start the next iteration.

2.2 Synchronous Distributed SGD

In the vanilla SGD, the entire training data, known as the "epoch", is divided into quantities of mini-batches, and each iteration involves one mini-batch for one training node. During the distributed training, all the computation nodes conduct their own iterations with different mini-batch training data and the synchronous distributed SGD performs the renewal of parameters with equation(1) [33].

$$w_{t+1} = w_t - \eta \frac{1}{Nb} \sum_{k=0}^{N} \sum_{x \in B_{k,t}} \nabla f(x, w_t) \tag{1}$$

where w are the weights of a neural network, η is the learning rate, N is the number of computation nodes in total, b is the mini-batch size, $B_{k,t}$ $(0 \leq k < N)$ is a sequence of mini-batches sampled from the entire training data at iteration t, $\nabla f(x, w_t)$ is the gradients calculated by the auto-differentiation in the backward step for updating the parameters. We should notice that the number multiplied by the learning rate is the average value of gradients gathered from all other nodes, and this feature can be used for optimizing the communication overhead by offloading the data reduction operation $\sum_{k=0}^{N} \sum_{x \in B_{k,t}} \nabla f(x, w_t)$ to programmable network device in the network, which contributes to reducing the communication traffic greatly.

2.3 In-network computing technique for DL

When it comes to distributed deep learning training, scalability always makes the criterion to evaluate the computing performance. To scale out successfully, a large quantity of efforts has been spared to overlap the computing and communication [4, 49], optimize the overheads of network communication by reducing the communication traffic [9, 18, 33, 43, 49] and improve the computing capability. As is the case with other kinds of high performance applications, computation and network communication are two critical factors during deep learning training process. Concerning the fast growth in computing power, network turns out to be the bottleneck in achieving good performance, especially when the deep learning algorithm runs on a cluster [37]. Deep learning training can be divided into two categories, the computation intensive and network intensive, the former type is less dependent on network than the latter one. As to the network intensive applications, vast amounts of data is pushed into the network and transmitted to the server nodes (Parameter Server) or other compute elements (Peer-to-Peer topology) to train the model. Then they need to aggregate the gradients to update the network weights and resynchronize them with other worker nodes before the next iteration. The increased computing power leads to a more intense requirement of network optimization by producing more data traffic per time unit. The key point to remit the performance bottleneck is reducing the network traffic.

With the appearance of programmable network infrastructures, in-network computing technique has achieved enough attentions [21, 30, 31, 34], which provides chances for greatly cutting down the network traffic. With the support of in-network computing technique, the data reduction operation can be conducted directly in the network, instead of passing all the gradients to the servers or other compute elements, then the communication traffic can be reduced to a large extend. Figure 2 shows the in-network computing technique used for Peer-to-Peer and PS structures. Instead of implementing the aggregation operations directly on programmable network devices, we choose simulation as the way for performance evaluation. To carry out the simulation, a comprehensive measurement of the communication details is needed. Beside, it is necessary to obtain precise communication traces of DL applications to achieve accurate simulation. These two requirements motivate us to propose the idea of *SketchDLC*, as both requirements can be satisfied at the same time. The idea means we provide a sketch on deep learning communication via trace capturing.

3 OUR INSIGHTS INTO THE COMMUNICATION MECHANISM OF MXNET

Different from getting the VEF traces by using the trace files generated by PARAVER tools [38], we obtain the DLC traces by directly extracting information from the MXNet framework during training, and each node get its own trace file after the process. To capture the trace files of MXNet, we need to have a detailed understand of its communication mechanism.

3.1 MXNet Based on Parameter Server

As a distributed ML library, MXNet is implemented on the basis of parameter server structure, and the network communications are completed through the Push/Pull operations (Figure 3(a)), ***Push*** is used for gradients passing and ***Pull*** is used for fetching the updated parameters from the server nodes.

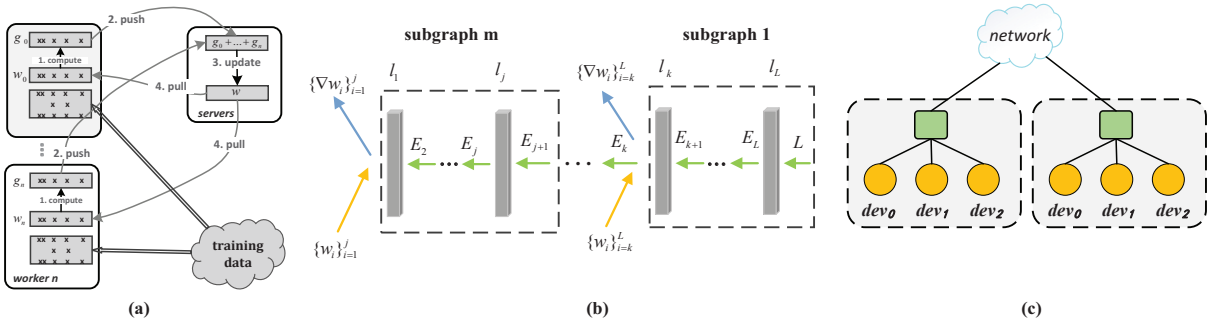

Fig. 3. Communication mechanism of MXNet

As described in section 2.2, synchronous distributed SGD needs to globally aggregate gradients from all other worker nodes, and the *KVStore API* in MXNet provides the distributed aggregation by introducing the <*key, value*> pairs. The *key* is a scalar and the *value* means a tensor or matrix, and every parameter in the neural network corresponds to one unique *key* , starting from zero, then Push/Pull operations are performed on it using the same *key*. The computed gradients are transmitted layer by layer after the computation process, and MXNet overlaps the computation and communication implicitly by auto-parallelizing independent subgraphs (Figure 3(b)). Before the aggregation of gradients from different worker nodes, each worker node will aggregate the gradients in its devices firstly (Figure 3(c)).

3.2 Details of The Communication Mechanism

Three main functions are included in the MXNet library, namely the resource management, computation and parameters maintenance, corresponding to the scheduler node, worker node and server node. Most communication operations occur between the worker node and the server node, while the communications with the scheduler node mainly happen in the beginning. This section provides some fine-grained communication analysis of MXNet.

3.2.1 Point to Point Message Passing. Most peer-to-peer structures make use of the MPI communication library, which contains abundant communication patterns, to implement their distributed communication, while parameter server structure adopts various communication libraries. What the different libraries have in common is that their communications are all point to point message passing operations.

MXNet is a parameter server based library, and all communication behaviors are completed through point to point message passing operations. Take the server group as an example, it needs to aggregate the gradients to accomplish the parameter renewal during the training process, and instead of one collective communication operation, this process is composed of a series of point to point communications. However, the simplicity of MXNet's communication mode makes it easier for us to record and analyze the communication trace.

3.2.2 Operations of Nodes. We divide the whole training process in MXNet into three main stages: (1) Nodes in the cluster set up connection with each other. Both the workers and servers will send their node information to the scheduler node during this stage; (2) Worker that ranks 0 sends the initialized parameters to the servers to do the initialization operation; (3) Workers push the computed gradients to the servers and pull back the updated parameters to conduct the training iterations by invoking the Push/Pull operations. Stage 1 and 2 act as the role of making preparations for the start of training (stage 3). One *Push* or *Pull* operation will trigger two actions for the corresponding worker node and server node, respectively. The worker sends out the *Push* or *Pull* request and

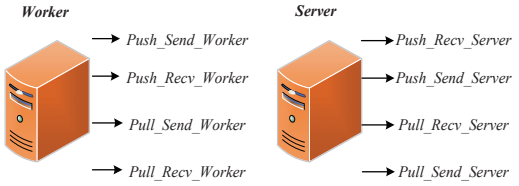

Fig. 4. Operations of worker and server node

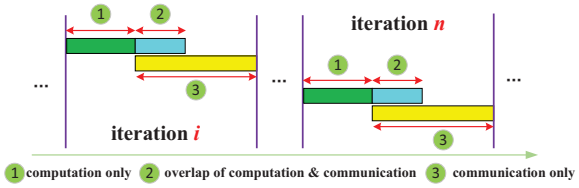

Fig. 5. Analysis of computation and communication overheads

wait for the response, while the server receives the *Push* or *Pull* request and send the response back to worker. We define the operations conducted in each node during the training process (Figure 4).

Push_Send_Worker: the worker node pushes the calculated gradients to the server node.
Push_Recv_Worker: the worker node receives the conformation message of *Push* operation from the server.
Pull_Send_Worker: the worker node sends the *Pull* request to the server to get the parameters.
Pull_Recv_Worker: the worker node receives the parameters sent from the server.
Push_Recv_Server: the server node receives the gradients pushed by the worker.
Push_Send_Server: the server node sends the *Push* conformation message back to the worker node.
Pull_Recv_Server: the server node receives the *Pull* request from the worker node.
Pull_Send_Server: the server node sends the requested parameters back to the worker node.

3.2.3 Dependecny Types. Dependency relationship seems to be the most important content in the point to point communication records. This allows the communication traces to indicate the condition that must be satisfied before processing the next record in the workflow of each node. We describe the dependency relationships in terms of each node's own operations, and there are four types of data dependencies during the training process, which are expressed by Push/Pull operations.

Push_on_Push: the *Push_Recv_Worker* operation depends on the *Push_Send_Worker* operation, and the *Push_Send_Server* operation relies on the *Push_Recv_Server* operation.
Pull_on_Push: the *Pull_Send_Worker* operation depends on the *Push_Recv_Worker* operation, and the *Pull_Recv_Server* operation counts on the *Push_Send_Server* operation.
Pull_on_Pull: the *Pull_Recv_Worker* operation depends on the *Pull_Send_Worker* operation, and the *Pull_Send_Server* operation counts on the *Pull_Recv_Server* operation.
Push_on_Pull: the *Push_Send_Worker* operation depends on the *Pull_Recv_Worker* operation, and the *Push_Recv_Server* operation relies on the *Pull_Send_Server* operation.

We further divide the four types of data dependencies into two classifications, or the ***one-to-one*** dependency and ***one-to-many*** dependency.

one-to-one: the current communication operation is dependent on just one previous operation in the same iteration;
one-to-many: the current communication operation is dependent on a series of communication operations that belong to the last or the same iteration.

As to the worker node, ***Push_on_Push, Pull_on_Push*** and ***Pull_on_Pull*** belong to the classification of ***one-to-one*** dependency, while the ***Push_on_Pull*** is the ***one-to-many*** type. Different from the worker node, ***Pull_on_Push*** and ***Pull_on_Pull*** in the server node are the types of ***one-to-one*** dependency, while the ***Push_on_Push*** and ***Push_on_Pull*** pertain to the ***one-to-many*** type. Take the worker node as a ***one-to-many*** example, the execution of ***Push_Send_Worker*** must wait for the accomplishment of all the ***Pull_Recv_Worker*** operations in the previous iteration to pull back the entire parameters for computation.

Table 1. DLC Trace Format

id	*src*	*dst*	*length*	*num_pp*	*operation*	*op_id*	*dep_type*	*d_time*	*time_sec*	*time_usec*	*id_dep*

3.2.4 Computation vs Communication. As is mentioned above, MXNet overlaps the computation and communication implicitly to improve the communication efficiency. We can divide each iteration into three phases to figure out the overlap ratio of computation and communication (Figure 5). ***Phase*** ① is the computation only stage, the worker node just conducts the computation task. ***Phase*** ② is the overlap stage, during which the communication process has began and the computation task has not finished yet. ***Phase*** ③ is the whole communication process. The practical computation overhead is the sum of overheads in ***Phase*** ① and ②. The time costs in ***Phase*** ③ equals to the communication overhead. The total overhead for one iteration is the sum of overheads in ***Phase*** ① and ③. After calculating the overheads in these ***Phases***, we are able to get the overlap ratio of computation and communication, which is the ratio between the overhead in ***Phase*** ② and the total overhead.

4 CAPTURING THE COMMUNICATION TRACE

As is mentioned above, we put forward to obtain the communication trace with two targets, the former one is making some general statistics and analysis about the communication features, while the latter one is providing a trace produced by real application for network simulation. Under this circumstance, a MXNet trace should contain the information needed to replay the communication operations, and the information for working out various types of overheads.

4.1 Definition of DLC Trace Format

Every node will get its own trace file after the MXNet trace obtaining process. The trace format is a list of information required for replaying and reading the trace, we aim at providing a readable trace and the final format is shown in Table 1.

- *id* is the communication record identifier for each message passing operation, it follows the ascending order and is increased by 1 every time.
- *src* is the source node of the communication record.
- *dst* is the destination node of the communication record.
  During the distributed training, information of *src* and *dst* can be obtained from the meta data, but what we get are node ids instead of server/worker ranks, it is necessary to convert them back into rank ids to provide intuitionistic tags for the nodes. Each worker has a unique rank within [0, numWorkers), so are the servers. As to the scheduler node, both its node id and rank id are 1. To keep the workers, servers and scheduler from sharing the same rank id, we renumber their rank ids with the following equations (2, 3, 4), starting from the worker nodes, then server nodes and end up with the scheduler node. If the cluster has 3 worker nodes, 2 server nodes and 1 scheduler node, then their rank ID will be 0, 1, 2 for the worker group, 3, 4 for the server group and 5 for the scheduler node.

$$worker\_ranker = [0, numWorkers) \tag{2}$$

$$server\_ranker = numWorkers + [0, numWorkers) \tag{3}$$

$$scheduler\_ranker = scheduler.id + numWorkers + numServers - 1 \tag{4}$$

- *length* is the size of delivered message measured in bytes. Message in each communication record contains the meta data and the data transmitted for neural network training, thus the length consists of the data size of both, which is calculated in the message passing process and can be used in the trace files directly.
- *num_pp* is the sequence number of communication operations starts from 0, and each pair with the same value corresponds to the same *Push/Pull* operation. Before the iterations for training, *num_pp* records the operations of setting up connections with the server group and getting the initialized parameters. During the training process, *num_pp* records the dispatched order of Push/Pull operations. In effect, every Push operation is followed by a Pull operation, and they are dispatched in the form of pair, making the odd (even) *num_pp* represents the Push operation while the even (odd) *num_pp* means Pull operation. However, there is a special value (-15) assigned to it when worker node with rank id 0 is conducting the initialization work. What should be noticed is that although the sequence numbers of Push/Pull operations follow the ascending order, they are out of order in the trace files, and this is resulted from the non-blocking communication mechanism in MXNet.
- *operation* is the network action of a node, which records both the communication type and the performer. Its format is *Push(Pull)_Send(Recv)_node*, just as the contents explained in section 3.2.2.
- *op_id* is the identifier of network action in each node, which presents detailed information about the operation and is composed of three pieces of contents. The format is ***key-operation_num-node***, where:
  key is the index of parameters in each layer. Take the convolutional layer as an example, there are two parameters, or the weights and bias, then two ***keys*** are needed to mark the parameters.
  operation_num here is the communication identifier for the *Push/Pull* operations. Every training iteration contains four successive communication identifiers, corresponding to the *Push_Send_Worker* (*Push_Recv_Server*), *Push_Recv_Worker* (*Push_Send_Server*), *Pull_Send_Worker* (*Pull_Recv_Server*) and *Pull_Recv_Worker* (*Pull_Send_Server*) operations separately. It starts from −2 in the trace files of server nodes and 0 in the trace files of worker nodes. Due to the initialization process, worker_0 includes two more communication records than other workers, making 0 and 1 in its trace file signify the initialization operations. The ***operation_num*** in the servers starts from −2 to keep the communication records with the workers in the same iteration with the same ***op_id***. In the trace files of server nodes, −2 and −1 represent the initialization process, 0 and 1 mean the worker group pulls back the parameters before the beginning of formal iteration. 2 and numbers after it belong to the training iterations (2, 3, 4, 5 belong to the first training iteration, and the rest can be done in the same manner). In worker_0, 0 and 1 keep the initialization records, 2 and 3 keep the records of getting the initialized parameters, 4, 5, 6 and 7 is the first training iteration. As to other worker nodes, 0 and 1 are used for parameters obtaining, numbers greater than 1 are for the training iterations.
  node is the role played by the nodes, attached with the rank id (w1 represents the worker 1, and s0 is the server 0 in short).
  Take *4-7-s0* in trace file of worker_0 as an example (Figure 8), this piece of record shows us that the worker node receives the pulled parameters from server 0 in the first iteration, and the key of the received parameter is 4.
- *dep_type* is used to describe the dependency types introduced in section 3.2.3, which takes values from 0 to 4, with the following meaning:
  0: Independent records 1: Push_on_Push records 2: Pull_on_Push records 3: Pull_on_Pull records
  4: Push_on_Pull records
- *d_time* is the dependency time measured in microseconds (μs). In fact, values of this field is used to record the ***one-to-one*** dependency time. As to the type of ***one-to-many*** dependency, *d_time* is not accurate because of the synchronization overhead and wait overhead, and that is why we introduce the *time_sec*, *time_usec* fields. The basis of calculating the dependency time lies in that the accomplishment of message

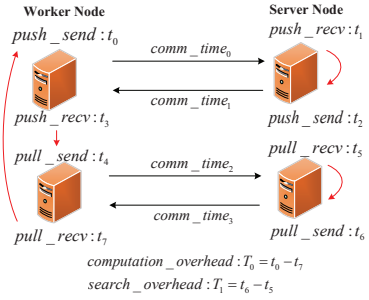

Fig. 6. Computation and search overhead

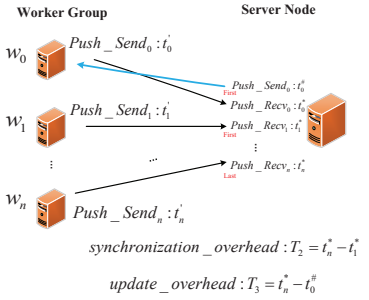

Fig. 7. Synchronization and update overhead

passing for a parameter follow the order of *Push_Send_Worker* (*Push_Recv_Server*), *Push_Recv_Worker* (*Push_Send_Server*), *Pull_Send_Worker* (*Pull_Recv_Server*) and *Pull_Recv_Worker* (*Pull_Send_Server*) strictly in one iteration, and the former operation counts on the latter operation, as shown in Fig. 12. The intervals between the operations are the ***one-to-one*** dependency time.

- *time_sec*, *time_usec* are the moments each communication operation is conducted, *time_sec* is measured in seconds and *time_usec* is measured in microseconds (μs). They are introduced to figure out the ***one-to-many*** dependency time, or the *synchronization overhead* under the synchronous training mode and the *wait overhead* in the worker node for pulling back all the parameters. Besides, we are able to figure out the *computation overhead*, *update overhead* and *search overhead* (4.2) with the *time_sec* and *time_usec*, conducing to a better understanding about the overheads during the training process.
- *id_dep* is the identifier of the record on which it depends. When dealing with the ***one-to-one*** dependency, *id_dep* represents just another piece of record and its format is in accordance with the *op_id* field. Nevertheless, *id_dep* contains a series of records in the ***one-to-many*** dependency situation, as to the server node, contents are all the related communication records with the worker nodes, and to the worker node, it is the ***operation_num*** to represent all the *Pull_Recv_Worker* operations in the last iteration. −1 in this field means the current operation does not rely on any other communication operations, corresponding to the 0 in *dep_type* field.

We are able to obtain readable and comprehensive trace files with this trace format. The trace files contribute to a detailed understanding of the communication mechanism. Besides, we can figure out various types of time overheads, which allow us to analyze the overlap ratio of communication and computation, and provide us with further insights into the time cost during the training process.

4.2 Overheads involved in Training Process

The training process consists of computation overhead and communication overhead. The computation overhead is simple, while the communication overhead includes a series of other pivotal overheads, including *search overhead*, *update overhead*, *synchronization overhead* and *wait overhead*. The specific meaning of each overhead is shown below.

computation overhead: the time used for the forward and backward computing process to calculate the gradients. The interval between the last *Pull_Recv_Worker* operation and the last *Push_Send_Worker* of the next iteration is the computation overhead, as the T_0 in Figure 6, or the sum of overheads in Phase ① and Phase ② (Figure 5).

search overhead: time for the server node to search the parameter according to the pull request from the worker node, as the T_1 in Figure 6. In fact, the *search overhead* is the ***d_time*** between the *Pull_Recv_Server* and the *Pull_Send_Server* operations.

Table 2. Definition of *Message*, *Meta* and *KVPairs*

```
struct Message {
        ......
/* the meta info of this message */
        Meta data;
/* the large chunk of data of this message */
        std::vector<Sarray<char> > data;
        ......
};
```

```
struct Meta {
        ......
/* the node id of the sender */
        int sender;
/* the node id of the receiver */
        int recver;
/* whether or not a push message */
        bool push;
        ......
};
```

```
struct KVPairs {
/* the list of keys */
        SArray<key> keys;
/* the according values */
        Sarray<Val> vals;
/* the according value length
could be empty */
        Sarray<int> lens;
};
```

Table 3. Modification of *Meta*, *KVMeta* and *meta.proto*

```
struct Meta {
      ......
/* whether or not a request message */
      bool request;
/* record the push_pull_operation */
      int push_pull_operation;
/* record the key */
      int recv_key;
      ......
};
```

```
struct KVMeta {
        ......
/* the associated timestamp */
        int timestamp;
/* record the push_pull_operation */
        int push_pull_operation;
/* record the key */
        int recv_key;
};
```

```
message PBMeta {
      ......
/* the  push_pull_operation of the message*/
      optional int32 push_pull_operation = 10;
/* the key of push or pull operation */
      optional int32 recv_key = 11;
      ......
};
```

synchronization overhead: this exists in the synchronous update mode only, which results from the difference in computation capability and the network congestion. The time interval between the first and the last *Push_Recv_Server* operations in one iteration is the *synchronization overhead*, as the T_2 in Figure 7.

update overhead: the time cost for the renewal of parameters in the server node, marked as the interval between the last *Push_Recv_Server* and the first *Push_Send_Server* for the same parameter during one iteration (T_3 in Figure 7). After receiving the gradients from the worker node(s), the sever node will conduct the update operation. As to the asynchronous update mode, the *update overhead* is the ***d_time*** of *Push_Send_Server* records in the trace file, while this does not suit the synchronous mode, whose ***d_time*** of *Push_Send_Server* operation is related to both the *synchronization overhead* and *update overhead*.

wait overhead: this exists only in the worker node. Before the start of the next iteration, the worker node need to get all the updated parameters from the server group, and the interval between the first and the last parameter to be acquired is the *wait overhead*.

These overheads contribute to a better understanding about the communication behaviors, and getting these overheads calls for the precise moment each operation is done. With the DLC trace format, the moments are kept in the field of ***time_sec*** and ***time_usec***, providing us the access to work out those overheads manually.

4.3 Implementation

We planned to develop an independent light-weight library to obtain the communication trace in the first place, while some necessary information, which is redundant in the distributed communication process, is not available in the final communication APIs (*SendMsg( ) & RecvMsg( )*). In order to provide readable and useful communication trace for simulation and analysis, we are obliged to modify the source code of MXNet to acquire the extra information. This section first presents the available information for trace capturing and then elaborates the main modifications for the redundant information.

4.3.1 Available Information. All the point-to-point communication operations are finally completed by invoking the *SendMsg()* and *RecvMsg()* functions, where we get the operation information to generate the communication

Algorithm 1: procedures of *Trace_Send_()*

```
Input: const Message& msg, int send_bytes
Input: int key, struct timeval end
.....
1  if (is_scheduler())
2      keep the record in trace_sch_record.txt
3  if (is_worker())
4      if (recver == scheduler)
5          keep the record in trace_sch_record.txt
6      else  /* recver == server */
7          if  (msg.meta.push)  /* Push_Send_Worker */
8              keep the Push_Send_Worker record in trace_record.txt
9          else /*Pull_Send_Worker */
10             keep the Pull_Send_Worker record in trace_record.txt
11 if (is_server())
12     if (recver == scheduler)
13         keep the record in trace_sch_record.txt
14     else /* recver == worker */
15         if (msg.meta.push) /* Push_Send_Server */
16             keep the Push_Send_Server record in trace_record.txt
17         else /* Pull_Send_Server */
18             keep the Pull_Send_Server record in trace_record.txt
```

Table 4. Usage of *unorder_map* and Definition of *Trace_*

```
1  /* used for the records of worker node */
2  std::unorder_map<int, Trace_> Trace_Push_Send[num_worker]
3  std::unorder_map<int, Trace_> Trace_Push_Recv[num_worker]
4  std::unorder_map<int, Trace_> Trace_Pull_Send[num_worker]
5  std::unorder_map<int, Trace_> Trace_Pull_Send[num_worker]
6  std::unorder_map<int, int> Init_Flag_Wor[num_worker]
7  /* used for the records of server node */
8  std::unorder_map<int, Trace_> Ser_Push_Recv[num_server]
9  std::unorder_map<int, Trace_> Ser_Push_Send[num_server]
10 std::unorder_map<int, Trace_> Ser_Pull_Recv[num_server]
11 std::unorder_map<int, Trace_> Ser_Pull_Send[num_server]
12 std::unorder_map<int, int> Init_Flag_Ser[num_server]
```

```
1  struct Trace_{
2  /* the operation_num used in the field of op_id */
3      int operation_num;
4  /* used for d_time */
5      struct timeval time;
6  };
```

trace, and the arguments of both are the data type of Message. Table 2 shows partial definition of *Message* and *Meta* (message.h), as well as the full definition of *KVPairs* (kv_app.h). *data* in line 4 stores the data of Meta type while data in line 6 storages the contents in *KVPairs*. With the *Meta* data, we are able to get node ids of the sender and receiver, then the ***src*** and ***dst*** can be obtained. The communication operation can also be recognized via the *push* in *Meta*, 0 for *Pull* and 1 for *Push*. Besides, we can identify the role of the current node by using the *is_worker()*, *is_server()* and *is_scheduler()* functions (postoffice.h). Then the ***operation*** can be determined according to the current operation conducted in *SendMsg()* or *RecvMsg()*. Once the ***operation*** is known, we can figure out the ***dep_type*** as the dependency relationships among the operations have been analyzed. The length of each message is calculated in the *SendMsg()* and *RecvMsg()*, which can be used directly. In conclusion, the ***src***, ***dst***, ***length***, ***operation*** and ***dep_type*** in the trace format are available based on the original MXNet library.

4.3.2 Main Modifications. To obtain the DDL traces, we design two new functions in the zmq_van.h file, *Trace_Send_()* and *Trace_Recv_()*, the former one is embedded in *SendMsg()* and the latter one is in *RecvMsg()*. Algorithm 1 illustrates the overall procedures of *Trace_Send_()*, and it is responsible for all the sending opration records, namely the *Push_Send_Worker*, *Pull_Send_Worke*, *Push_Send_Server* and *Pull_Send_Server*. The algorithm of *Trace_Recv_()* is similar to Algoritm 1, and it is in charge of all the receiving operation records

Apart from the available information mentioned above, keeping a piece of operation record calls for some extra information. The ***num_pp*** needs the dispatch number of Push/Pull operations, ***op_id*** demands the *key* and *operation_num*, ***id_dep*** is dependent on ***op_id***. As to ***id***, ***time_sec*** and ***time_usec***, they can be acquired directly and have nothing to do with the MXNet library. Therefore, we should modify some existing data structures to deliver the *key* and *push_pull_operation* (the dispatch number of Push/Pull operations) to *Trace_Send_()* and *Trace_Recv_()*. Actually, *key* can be gained by visiting the *data* in *Message* when conducting most of the communication operations, it is not available only when the server node sends back the confirmation of the *Push* operation, because *key* is not included in *KVMeta* (kv_app.h), which contains the meta information of the response message from server node. *push_pull_operation* is a newly introduced parameter, and to make it more convenient to get the *key*, we add both the *push_pull_operation* and *key* into the *Meta* and *KVMeta* (Table 4), and also alter the meta.proto to deliver the arguments. There are also many other adjustments that should be made to different functions, we will not go into the details for the limited space of article.

When all the necessary information is ready, we demand a convenient method to figure out the ***op_id*** and ***d_time***, leading to the use of *unorder_map* data structure (zmq_van.h) and the definition of *Trace_* (message.h), shown in Table 4. As described above, the basis for obtaining the trace is that the communication for each key in one iteration is independent, and strictly follows the stationary sequence (Worker: *Push_Send_Worker*, *Push_Recv_Worker*, *Pull_Send_Worker* and *Pull_recv_Worker*; Server: *Push_Recv_Server*, *Push_Send_Server*, *Pull_Recv_Server* and *Pull_Send_Server*). Therefore, ***d_time*** can be easily acquired with the time in *Trace_*, and ***op_id*** is also available with the *operation_num* in *Trace_*. We update the *operation_num* for the i^{th} parameter in worker through equations (5) to (8) and only one equation is used for one piece of communication record. Once ***op_id*** is captured, ***id_dep*** can be expressed according to the dependency relationships, too.

$$Trace\_Push\_Recv[w][i].operation\_num = Trace\_Push\_Send[w][i].operation\_num + 1 \quad (5)$$

$$Trace\_Pull\_Send[w][i].operation\_num = Trace\_Push\_Recv[w][i].operation\_num + 1 \quad (6)$$

$$Trace\_Pull\_Recv[w][i].operation\_num = Trace\_Pull\_Send[w][i].operation\_num + 1 \quad (7)$$

$$Trace\_Push\_Send[w][i].operation\_num = Trace\_Pull\_Recv[w][i].operation\_num + 1 \quad (8)$$

After the main modifications mentioned above, we are capable of collecting the information needed in the trace format, and we obtain a readable and comprehensive communication trace eventually.

5 DLC TRACE: AN SKETCH ON DEEP LEARNING COMMUNICATION

As is stated above, we aim at providing useful and readable trace files for a comprehensive measurement of the deep learning communication. We eventually fulfill our target by introducing the DLC trace format. In this section, we confirm the communication mechanism by providing a fraction of the DLC trace, make some statistics about the communication features, and analyze various types of overheads in the training process.

5.1 Experiments

Our method implemented in the MXNet framework for trace capturing directs at cluster with large scale. However, due to the limited experiment resources, we are available to only 4 machines. Each machine has 40 2.60 GHz Intel cores, 2 K80 (dual GPUs) GPUs, 64G memory, and a 56 Gbps network interface. Even though one machine can act different roles (*scheduler*, *server*, *worker*) at the same time, we set each node for just one role during the experiments, which makes it much more convenient to analyze the trace records. The version of MXNet is 0.12.1, and every machine will get its own trace file after the trace obtaining process.

Datasets and Neural Networks. We used two datasets for our experiments: (1) MNIST [28] dataset, which contains a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST and all the digits have been size-normalized and centered in a fix-size image. (2) ImageNet [11] ILSFRC2012 dataset, the subset of ImageNet containing 1000 categories and 1.2 million images. Corresponding to the datasets, two neural networks are applied in our experiments: (1) LeNet-5 [29], which is the latest convolutional network designed for handwritten and machine-printed character recognition. It consists of 8 layers, and there are 8 parameters need to be transmitted in every training iteration, Table 6 shows the message length of every operation in the worker node. (2) Resnet-50 [19], which is a branch of the residual neural network. There are 157 parameters altogether that need to be delivered in every iteration and we just list the message with different lengths (Table 7).

5.2 A Glimpse of DLC Trace

This subsection throws lights on how to read the trace files via providing a trace sample. After detaching the communication records with the scheduler node, we can distinguish three types of contents in the trace files of server nodes and the worker_0 node: the trace file header, the initialization records and the training records

Table 5. Data Transmitted in One Iteration

	Push_Send (MB)	Pull_Send (MB)	Pull_Recv (MB)	Push_Recv (MB)
Resnet-50	102.21	0.00331	0.00471	102.21
Lenet-5	1.72	0.00015	0.00022	1.72

Table 6. Message Length of Lenet-5

Key	Push_Send (B)	Push_Recv (B)	Pull_Send (B)	Pull_Recv (B)
7	73	19	28	76
1	113	19	28	116
3	233	19	28	236
0	2033	19	28	2036
5	2033	19	28	2036
6	20033	19	28	20036
2	100033	19	28	100036
4	1600033	19	28	1600036

Table 7. Message Length of Resnet-50

Key	Push_Send (B)	Push_Recv (B)	Pull_Send (B)	Pull_Recv (B)
0	47	21	30	50
3	291	21	30	294
36	547	21	30	550
15	1059	21	30	1062
43	2083	21	30	2086
156	4036	21	30	4039
135	8228	21	30	8231
7	16419	21	30	16422
2	37667	21	30	37670
13	65571	21	30	65574
35	131107	21	30	131110
10	147491	21	30	147494
41	262179	21	30	262182
42	524323	21	30	524326
38	589859	21	30	589862
78	1048611	21	30	1048614
79	2097187	21	30	2097190
75	2359331	21	30	2359334
133	4194340	21	30	4194343
155	8192036	21	30	8192039
134	8388644	21	30	8388647
130	9437220	21	30	9437223

for parameters renewal. As to the trace files of other worker nodes, they contain only half of the initialization records because they are not responsible for pushing initialized values to the server nodes.

Trace Header. This comes first in the trace file because it contains basic information and some instructions of the trace. As the blue records in Figure 8, there are 2 worker nodes and 1 server node in the experiment, the workers are tagged with 0, 1 while the server is tagged with 2, the hostname of current node is GPU-1-9. As to the contents in ***num_pp*** field, the odd (even) numbers represent ***Push*** operations and the even (odd) numbers means ***Pull*** operations, the special value −15 is the mark for initialization records and 0 tags the process of setting up connections between worker group and server group. Arrows in the push and pull describe the dependency relationships in worker nodes and server nodes. For instance, "push_send_worker —-> push_recv_worer" shows us that the push_recv_worker operation in the worker node relies on the push_send_worker operation.

Initialization records. Trace fragment in Figure 8 is based on Lenet-5, in which only 8 parameters need to be transmitted. The green part keeps the communication records between worker_0 and the server node (we set only one server node) in the initialization phase. We can browse the communication records according to the ***num_pp*** list, the two records with value 0 signify the process of setting up connection with the server, after which the worker node begin to deliver values to the server node to initialize the parameters. Once the parameter is initialized and the worker receive the signal, it sends out the *Pull* request to get back the parameter. For example, 0-0-s0, 0-1-s0, 0-2-s0 and 0-3-s0 correspond to the *initialize* and *Pull* operations of the parameter with key 0, and the latter record relies on the former one, which we can tell from the ***id_dep*** field.

Training records. The basis of trace capturing is the strictly followed execution order of *Push_Send_Worker*, *Push_Recv_Worker*, *Pull_Send_Worker* and *Pull_Recv_Worker* for each *(key, value)* pair in a worker node, and *Push_Recv_Server*, *Push_Send_Server*, *Pull_Recv_Server* and *Pull_Send_Server* in a server node. In the communication phase, a series of Push/Pull operations will be sent out, each ***Push*** operation is followed by a ***Pull*** operation, and one *(key, value)* pair matches one pair of Push/Pull operation. The yellow part in Figure 8 presents the training records of one iteration with Lenet-5. Before the transmission of gradients, the worker node sets up connection with the server group just as it did in the initialization stage. After which 8 ***Push*** operations are conducted, the corresponding values in ***num_pp*** are 21, 23, 19, 15, 13, 17, 9 and 11. However, the ***Push*** operations follow the ascending order when they were sent out, while their execution order depends on when the gradients are ready, leading to the out-of-order records. The worker node carries out the ***Pull*** operation as soon as possible once the confirmation of ***Push*** operation is received from the server. Take the key 6 as an example, whose ***num_pp*** values are 21 and 22. id_36 records the *Push_Send_Worker* operation, whose ***op_id*** is 6-4-s0, and its

== num_workers: = 2 num_servers: = 1 worker_hostname:= GPU-1-12 Worker_ID: = 0,1, Server_ID: =2,
== num_pp: -15: Initialize Server 0: setup messages odd (even): push_operation even (odd): pull_operation
== push operation: push_send_worker → push_recv_worker push_recv_server → push_send_server
== Pull operation: pull_send_worker → pull_recv_worker pull_recv_server → pull_send_server
== op_id: (key-operation_num-role) key: the index of parameters role: (worker or server)
operation_num: the num of push or pull, every four num means an iteration (push_send, push_recv, pull_send, pull_recv)

id	src	dst	length	num_pp	operation	op_id	dep_type	d_time	time_sec	time_usec	id_dep
0	0	2	25	0	OP:= SendCom_TO_Servers						
1	2	0	28	0	OP:= SendCom_To_Servers						
2	0	2	2042	-15	OP:= Push_Send_Worker	0-0-s0	0	0	1516622729	481409	-1
3	2	0	28	-15	OP:= Push_Recv_Worker	0-1-s0	1	13820	1516622729	495229	0-0-s0
4	0	2	28	1	OP:= Pull_Send_Worker	0-2-s0	2	215080	1516622729	710309	0-1-s0
5	2	0	2036	1	OP:= Pull_Recv_Worker	0-3-s0	3	969	1516622729	711278	0-2-s0
6	0	2	122	-15	OP:= Push_Send_Worker	1-0-s0	0	0	1516622729	712717	-1
7	2	0	28	-15	OP:= Push_Recv_Worker	1-1-s0	1	787	1516622729	713504	1-0-s0
8	0	2	28	2	OP:= Pull_Send_Worker	1-2-s0	2	1109	1516622729	714613	1-1-s0
9	2	0	116	2	OP:= Pull_Recv_Worker	1-3-s0	3	730	1516622729	715343	1-2-s0
10	0	2	100042	-15	OP:= Push_Send_Worker	2-0-s0	0	0	1516622729	716931	-1
11	2	0	28	-15	OP:= Push_Recv_Worker	2-1-s0	1	1186	1516622729	718117	2-0-s0
12	0	2	28	3	OP:= Pull_Send_Worker	2-2-s0	2	1283	1516622729	719400	2-1-s0
13	0	2	242	-15	OP:= Push_Send_Worker	3-0-s0	0	0	1516622729	719654	-1
14	2	0	100036	3	OP:= Pull_Recv_Worker	2-3-s0	3	1303	1516622729	720703	2-2-s0
15	2	0	28	-15	OP:= Push_Recv_Worker	3-1-s0	1	1167	1516622729	720821	3-0-s0
16	2	0	236	4	OP:= Pull_Recv_Worker	3-3-s0	3	767	1516622729	722803	3-2-s0
16	0	2	28	4	OP:= Pull_Send_Worker	3-2-s0	2	1215	1516622729	722036	3-1-s0
18	0	2	1600042	-15	OP:= Push_Send_Worker	4-0-s0	0	0	1516622729	724666	-1
19	2	0	28	-15	OP:= Push_Recv_Worker	4-1-s0	1	4226	1516622729	728892	4-0-s0
20	0	2	28	5	OP:= Pull_Send_Worker	4-2-s0	2	1210	1516622729	730102	4-1-s0
21	0	2	2042	-15	OP:= Push_Send_Worker	5-0-s0	0	0	1516622729	730337	-1
22	2	0	1600036	5	OP:= Pull_Recv_Worker	4-3-s0	3	6419	1516622729	736521	4-2-s0
23	2	0	28	-15	OP:= Push_Recv_Worker	5-1-s0	1	6297	1516622729	736634	5-0-s0
24	2	0	2036	6	OP:= Pull_Recv_Worker	5-3-s0	3	746	1516622729	739274	5-2-s0
24	0	2	28	6	OP:= Pull_Send_Worker	5-2-s0	2	1894	1516622729	738528	5-1-s0
26	0	2	20042	-15	OP:= Push_Send_Worker	6-0-s0	0	0	1516622729	740631	-1
27	2	0	28	-15	OP:= Push_Recv_Worker	6-1-s0	1	812	1516622729	741443	6-0-s0
28	0	2	28	7	OP:= Pull_Send_Worker	6-2-s0	2	1002	1516622729	742445	6-1-s0
29	0	2	82	-15	OP:= Push_Send_Worker	7-0-s0	0	0	1516622729	742606	-1
30	2	0	20036	7	OP:= Pull_Recv_Worker	6-3-s0	3	860	1516622729	743305	6-2-s0
31	2	0	28	-15	OP:= Push_Recv_Worker	7-1-s0	1	820	1516622729	743426	7-0-s0
32	0	2	28	8	OP:= Pull_Send_Worker	7-2-s0	2	1189	1516622729	744615	7-1-s0
33	2	0	76	8	OP:= Pull_Recv_Worker	7-3-s0	3	770	1516622729	745385	7-2-s0
34	0	2	25	0	OP:= SendCom_To_Servers						
35	2	0	28	0	OP:= SendCom_To_Servers						
36	0	2	20033	21	OP:= Push_Send_Worker	6-4-s0	4	70213	1516622729	812819	(3-s0,)
37	0	2	73	23	OP:= Push_Send_Worker	7-4-s0	4	69380	1516622729	813995	(3-s0,)
38	0	2	2033	19	OP:= Push_Send_Worker	5-4-s0	4	75957	1516622729	814485	(3-s0,)
39	0	2	233	15	OP:= Push_Send_Worker	3-4-s0	4	92763	1516622729	814799	(3-s0,)
40	0	2	100033	13	OP:= Push_Send_Worker	2-4-s0	4	95239	1516622729	814893	(3-s0,)
41	0	2	1600033	17	OP:= Push_Send_Worker	4-4-s0	4	85015	1516622729	815352	(3-s0,)
42	2	0	19	21	OP:= Push_Recv_Worker	6-5-s0	1	4216	1516622729	817035	6-4-s0
43	0	2	28	22	OP:= Pull_Send_Worker	6-6-s0	2	306	1516622729	817341	6-5-s0
44	0	2	2033	9	OP:= Push_Send_Worker	0-4-s0	4	109072	1516622729	819381	(3-s0,)
45	0	2	113	11	OP:= Push_Send_Worker	1-4-s0	4	104862	1516622729	819475	(3-s0,)
46	2	0	19	23	OP:= Push_Recv_Worker	7-5-s0	1	6052	1516622729	820047	7-4-s0
47	0	2	28	24	OP:= Pull_Send_Worker	7-6-s0	2	274	1516622729	820321	7-5-s0
48	2	0	19	19	OP:= Push_Recv_Worker	5-5-s0	1	6813	1516622729	821298	5-4-s0
49	0	2	28	20	OP:= Pull_Send_Worker	5-6-s0	2	291	1516622729	821589	5-5-s0
50	2	0	19	15	OP:= Push_Recv_Worker	3-5-s0	1	7340	1516622729	822139	3-4-s0
51	0	2	28	16	OP:= Pull_Send_Worker	3-6-s0	2	277	1516622729	822416	3-5-s0
52	2	0	19	13	OP:= Push_Recv_Worker	2-5-s0	1	8553	1516622729	823446	2-4-s0
53	0	2	28	14	OP:= Pull_Send_Worker	2-6-s0	2	300	1516622729	823746	2-5-s0
54	2	0	20036	22	OP:= Pull_Recv_Worker	6-7-s0	3	6817	1516622729	824158	6-6-s0
55	2	0	76	24	OP:= Pull_Recv_Worker	7-7-s0	3	4113	1516622729	824434	7-6-s0
56	2	0	2036	20	OP:= Pull_Recv_Worker	5-7-s0	3	3022	1516622729	824611	5-6-s0
57	2	0	236	16	OP:= Pull_Recv_Worker	3-7-s0	3	2339	1516622729	824755	3-6-s0
58	2	0	19	17	OP:= Push_Recv_Worker	4-5-s0	1	10298	1516622729	825650	4-4-s0
59	0	2	28	18	OP:= Pull_Send_Worker	4-6-s0	2	282	1516622729	825932	4-5-s0
60	2	0	19	9	OP:= Push_Recv_Worker	0-5-s0	1	10822	1516622729	830203	0-4-s0
61	0	2	28	10	OP:= Pull_Send_Worker	0-6-s0	2	308	1516622729	830511	0-5-s0
62	2	0	19	11	OP:= Push_Recv_Worker	1-5-s0	1	11670	1516622729	831145	1-4-s0
63	0	2	28	12	OP:= Pull_Send_Worker	1-6-s0	2	274	1516622729	831419	1-5-s0
64	2	0	100036	14	OP:= Pull_Recv_Worker	2-7-s0	3	8742	1516622729	832488	2-6-s0
65	2	0	1600036	18	OP:= Pull_Recv_Worker	4-7-s0	3	10713	1516622729	836645	4-6-s0
66	2	0	2036	10	OP:= Pull_Recv_Worker	0-7-s0	3	6323	1516622729	836834	0-6-s0
67	2	0	116	12	OP:= Pull_Recv_Worker	1-7-s0	3	5487	1516622729	836906	1-6-s0

Fig. 8. DLC trace fragment from the trace file of worker_0, captured with Lenet-5

execution depends on all the *Pull_Recv_Worker* operations in the last iteration, tagged as 3-s0 in the record. id_42 is the *Push_Recv_Worker*, which counts on the *Push_Send_Worker*, its completion confirms the accomplishment of Push operation, and ***id_dep*** of this record is the ***op_id*** of *Push_Send_Worker*. Then the worker node starts the ***Pull*** operation, whose ***num_pp*** is 22. id_43 is the record of *Pull_Send_Worker* operation, which is dependent on 6-5-s0, the *Push_Recv_Worker* operation. id_54 represents the *Pull_Recv_Worker* operation, and it depends upon the *Pull_Send_Worker*, with the ***op_id*** 6-6-s0. id_36, id_42, id_43 and id_54 record the transmission of gradients and the acquirements of updated parameter with the key 6. The rest records can be analyzed in the same way, and the next iteration cannot start until the completion of all *Pull_Recv_Worker* operations. The yellow part in Figure 8 provides the communication records of one iteration. Lenet-5 has 8 parameters, and each corresponds to 4 operation records, there are 32 communication records in all during one iteration. As to the subsequent communication records in the trace file, they follow the similar pattern to the yellow part, and every iteration contain 32 communication records, except that the operation records of setting up connections no longer show up.

5.3 Analysis of Trace Files

5.3.1 General Statistics. This subsection presents our findings regarding general MXNet communication characteristics, such as message size, data volume, and communication patterns.

Message size: The process of updating one parameter demands 4 basic communication operations in the worker nodes and server nodes, respectively. Table 6 and 7 depict the message sizes for different parameters during the communication process of worker nodes. The third row record the sizes of confirmation messages for ***Push*** operations, and the fourth row keeps the message sizes of ***Pull*** requests, both of them are constant in the whole training process. Raw 2 and 5 record the message lengths for gradients and parameters transmission, and they vary greatly for different *keys*, the shortest messages in Lenet-5 and Resnet-50 are 73 *bytes* and 47 *bytes*, whereas the longest messages are up to 1.6 *MB* and 9.44 *MB*.

Data volume: the cluster starts the training tasks after the initialization phase, and the training tasks consist of nothing but a series of iterations. Reading the same amount of training data, carrying out the forward and backward computation, conducting the communication actions, acquiring the updated parameters are the operations done in each iteration. The total message size contained in one iteration is constant (Table 5) , thus the data volume can be worked out according to the to the number of iterations.

Communication pattern: According to the message lengths shown in Table 6 and 7, communication traffic during the training process is the combination of long messages and short messages, and this is decided by the characters of deep learning applications. *Push_Send_Worker*, *Pull_Recv_Worker*, *Push_Recv_Server* and *Pull_Send_Server* are responsible for the transfer of long messages, the transmission of short messages are conducted by *Push_Recv_Worker*, *Pull_Send_Worker*, *Push_Send_Server* and *Pull_Recv_Server* operations. Contrapose this communication pattern, optimizations can be made in the communication process. We can adopt different measures to handle the delivery of long messages and short messages, instead of the same strategy to deal with all the communication operations, which degenerates the distributed performance.

Computation vs Communication: With the captured trace files, we can easily figure out the overlap ratio of computation and communication. The overhead in ***Phase*** ① (Figure 5) is the interval between the last *Pull_Recv_Worker* of last iteration and the first *Push_Send_Worker* in current iteration. The time interval between the first *Push_Send_Worker* and the last *Push_Send_Worker* equals to the overhead in ***Phase*** ②. As to the communication overhead in ***Phase*** ③, it is the interval between the first *Push_Send_Worker* and the last *Pull_Recv_Worker* in the same iteration. Figure 9 demonstrates the overheads of these three phases. We pick up 380 iterations for Lenet-5 and 165 iterations for Resnet-50, more samples are used in Lenet-5 because its overheads is much less than Resnet-50 and may fluctuate more easily. The average computation overhead and communication

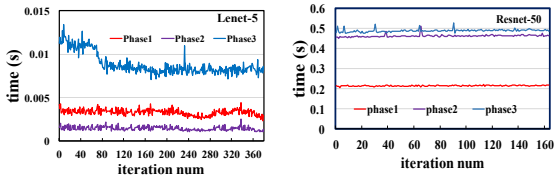

Fig. 9. Time overheads of the three phases

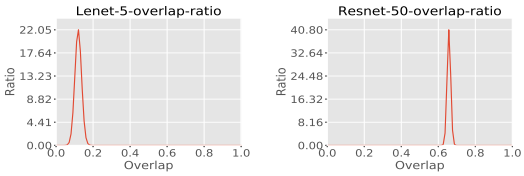

Fig. 10. Distribution curve of overlap ratio

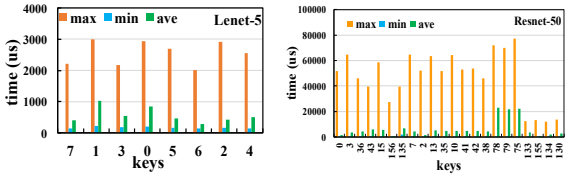

Fig. 11. Update overheads

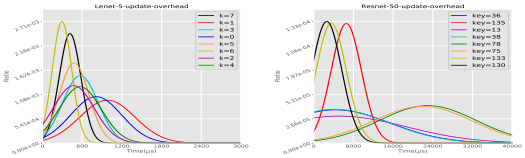

Fig. 12. Distribution curve of update overheads

overhead for Lenet-5 are 0.00474s and 0.00882s, while 0.6761s and 0.487s for Resnet-50. The communication overhead of Resnet-50 is about 56x greater than Lenet-5, and the computation overhead is 143x higher. Figure 10 illustrates the probability distribution of overlap ratio, the average values of Lenet-5 and Resnet-50 are 11.97% and 65.72% respectively. Considering that the model size of Lenet-5 is about 60x smaller than Resnet-50 (Table 5), we can draw the following conclusions. Firstly, the model size imposes nearly linear influences on communication overheads, while much greater effects on computation overhead. Secondly, larger model size contributes to higher overlap ratio, because the backward computation process takes much more time. Thirdly, the increase in model size calls for optimizations in both computation power and communication efficiency.

5.3.2 Overheads Analysis. We carry out the experiments on a cluster with just 4 machines, however, our method is applicable to cluster of large scale. This subsection aims at proving that the MXNet trace obtained gives access to detailed analysis of some overheads. The overhead statistics provided here is based on the trace files of Lenet-5 and Resnet-50, we consider 380 training iterations of the former neural network and 165 iterations of the latter one.

Update overhead: as to the asynchronous training mode, update overhead equals to the ***d_time*** of the *Push_Send_Server* operation records, and no more calculation is needed. The update overhead discussed here is under the synchronous mode.

Figure 11 demonstrates the update overheads of Lenet-5 and Resnet-50, which list the maximum value, minimum value, average value of each parameter. Resnet-50 has 157 parameters in total, and we only list the parameters with different message lengths. The order of keys in the figure is arranged according to the lengths of their corresponding messages, which keeps the ascending order. We can notice that update overheads vary greatly during the training process, but most update operations take much less time than the maximum overhead. Besides, the update overhead has little to do with the message length, as the average update overheads fluctuate and do not keep the ascending order. Finally, the size of the neural network does have an effect on the update overhead, a bigger model leads to greater update overhead. We figure out the average overhead of one update operation with the average update overhead of each key, 564.49 μs for Lenet-5 and 6261.89 μs for Resnet-50. The global average update overhead of Resnet-50 is nearly 11x times greater than the Lenet-5, while its model size is about 60x bigger. Figure 12 illustrate the probability distribution curve of update overheads. The graph of Resnet-5 contains 8 keys, corresponding to message lengths of different magnitudes. Obviously, the minimum and maximum value

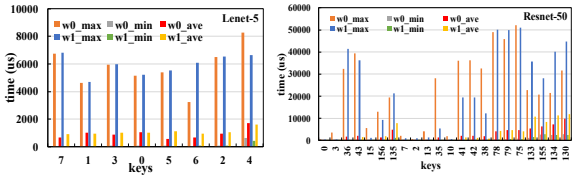

Fig. 13. Search overheads

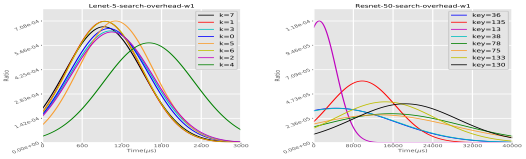

Fig. 14. Distribution curve of search overheads

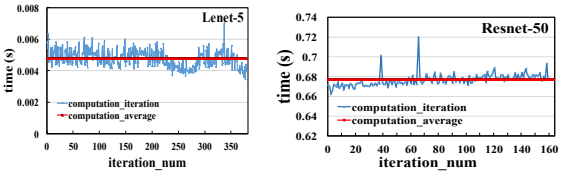

Fig. 15. Computation overheads

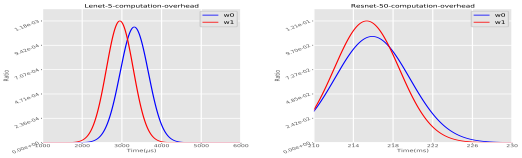

Fig. 16. Distribution curve of computation overheads

may vary greatly, but most of the update overheads fluctuate around the average value. In the network simulation experiments, we can generate the update overheads with accordance to these probability distribution graphs.

Search overhead: Search overhead is the time spent for the server node to seek the parameter according to the *Pull* request, and it has nothing to do with the renewal mechanism. The time interval between the *Pull_Recv_Server* and *Pull_Send_Server* is the search overhead, which equals to the ***d_time*** of the *Pull_Send_Server* operation records. Figure 13 shows the search overheads of Lenet-5 and Resnet-50, based on the pull requests from 2 worker nodes. The maximum values fluctuate greatly in Resnet-50, even though the maximum overhead for each key can be huge, most overheads of search operations are relatively low, making the average overheads much closer to the minimum values. Besides, both the message length and model size have influences on the search overhead. We work out the global average search overhead for one search operation with the average values in the figures. Resnet-50 (3078.46 μs) has a nearly 3 times higher search overhead than Lenet-5 (1007.47 μs). The average search overhead for key 4 in Lenet-5 is obvious greater than the other keys, so are the keys (78, 79, 75, 133, 155, 134, 130) in Resnet-50. When the message length is less than 1 *MB*, the search overheads are close and less than the global average value, and message length starts put influences on the overheads when it is larger than 1 MB. The message length for key 78 and key 130 are 1.05 *MB* and 9.44 *MB* respectively, and we can notice that the search overheads are apparently effected when the message length is increased by the scale of *MB*. Figure 14 shows the probability distribution graphs. It is apparent that the search overhead for key 4 in Lenet-5 is relatively greater, so are the keys (78, 75, 133, 130) in Resnet-50. The curves of Resnet-50 are not as stable as the Lenet-5, because the message lengths for the corresponding keys vary greatly.

Computation overhead: we define the time interval between the final arrived *Pull_Recv_Worker* of the last iteration and the final dispatched *Push_Send_Worker* operation of the current iteration as the computation overhead. The dispatch of *Push_Send_Worker* operations depends on the gradients calculated in the backward process. When the final *Push_Send_Worker* is dispatched, it means the completion of the computation process. Figure 15 illustrates the computation overheads of ***worker_1*** (worker node whose rank id is 1), the red lines in both graphs show the average overhead for one iteration. The overheads are relatively stable which fluctuates around the average value. The computation overhead of Resnet-50 (0.6764s) is nearly 143x higher than Lenet-5 (0.00475s), proving that model size plays an important effect on the computation time. Figure 16 illustrates the probability distribution graphs of computation overheads from both worker nodes. Even though the 2 worker nodes share the same configuration, their computation performance is kind of different. We can also notice the

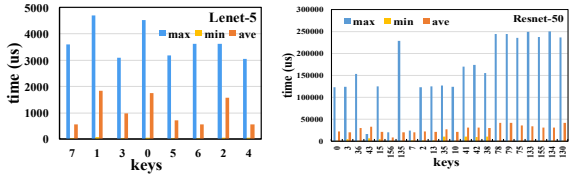

Fig. 17. Synchronization overheads

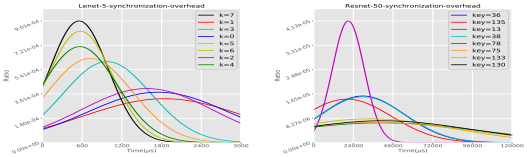

Fig. 18. Distribution curve of synchronization overheads

overheads distribution of Resnet-50 is not as compact as the Lenet-5. This resulted from the situation that bigger neural network calls for more computation cycles, and longer execution time leads to fluctuations more easily.

Synchronization overhead: it results from the different computation performance of the worker nodes, as well as the influences from network communication. Trace file of the server node is used for figuring out the synchronization overhead. We pick out the *Push_Recv_Server* operation records for each key, then calculate the time interval between the first and last *Push_Recv_Server* records that belong to the same iteration. The calculated time interval is the synchronization overhead of the iteration. Figure 17 demonstrates the synchronization overheads of Lenet-5 and Resnet-50. With the average overhead for each key, we get the global average synchronization overhead, 1068.38μs for Lenet-5 and 27975.41μs for Resnet-50. The average overheads for keys fluctuate around the global average overhead and do not follow the ascending order. Therefore, the synchronization overhead have little to do with the message length. However, the global average overhead of Resnet-50 is 26x larger than Lenet-5, proving that the model size imposes great effects on the synchronization overhead. Figure 18 the distribution curves of synchronization overheads, which can be used for generating the synchronization overhead during the network simulation experiments.

Wait overhead: as the time interval between the first received parameter and the last received parameter, the number of parameters transmitted in one iteration will generate decisive effects on it. The wait overhead is easy to understand, it is dependent on some other overheads mentioned above. In addition to search overhead, the update overhead and synchronization overhead are occasionally included. As shown in Figure 8, the *Pull* operations for keys 6, 7, 3 and 5 have finished (id_54, id_55, id_56, id_57) before the start of *Pull* operations for 4, 0 and 1 (id_59, id_61 and id_63), wait overhead in this iteration contains the update overhead and synchronization overhead of keys 4, 0, and 1. Considering the essence that wait overhead is a combination of other kinds of overheads, we leave out it during our statistics.

6 RELATED WORK

With the rapid development of DL, the complexity of neural network models and the expansion of training data have increased the demand for computing power and communication bandwidth greatly, thus limiting the efficiency of distributed training to some extent [9]. In response to this problem, a series of research work in hardware and software has been done to scale out successfully.

Nvidia Corporation releases the DGX-1 [12] to speed up the training process through the utilization of Tesla V100, NVLink and new Tensor Core architecture. DGX-2 [13] is also available now, claiming to be the most powerful AI system for the most complex AI challenges, which is powered by NVIDIA DGX software and a scalable architecture built on NVIDIA NVSwitch. The Intel Knights Landing (KNL) [47] is a general-purpose accelerator, but the distinct features, *Self-hosted Platform*, *Better Memory*, and *Configurable NUMA*, can obviously benefit deep learning applications. The Tensor Processing Unit [22] by Google and the DianNao family [8] by Cambrian Technology can also accelerate the computation, but they are mainly used for the inference phase.

Optimizations from the perspective of software is an active field of research. MXNet and Tensorflow implicitly overlap the communication and computation by auto-parallelizing independent subgraphs. Sufficient Factor

Broadcasting (SFB) [44] is proposed to reduce the network traffic, and applied in Poseidon and Microsoft Adam. Song Han et al raise the methods of pruning redundant connections using a three-step strategy [18] and Deep Gradients Compressing [33] to optimize the communication traffic, and turn out to be extremely successful. Increasing batch size [15, 46, 47] is also effective in improving training efficiency with the warm-up strategy [15], linear scaling rule [25] and LARS algorithm [45].

we draw conclusion from these works that most of the current optimizations are carried out in terms of frameworks or the training algorithms. With the occurrence of programmable network devices, which are successfully applied in data centers [16, 21, 30, 31, 35], we intend to conduct some optimizations on the network protocols and devices in the way of simulation, and accurate network simulation calls for comprehensive measurements and accurate communication trace, leading to the DLC trace introduced in this paper.

7 CONCLUSION

We mainly introduce three parts of contents in this paper, the reasons for proposing to obtain the communication traces of deep learning applications, the coarse-grained procedures on how to get the final trace files, and some analyses on the captured trace files.

Providing a comprehensive measurement on the distributed deep learning communication is the primary goal of our work. We argue that the optimizations on the functions of network devices contribute to the distributed communication efficiency. Instead of implementing the adjustments directly in the network devices, we decide to evaluate the performance firstly through the way of simulation. To conduct the network simulation, a deep understanding of the communication details and a trace file to describe the communication behaviors are needed. Therefore, we propose to carry out a measurement on the deep learning communication in the way of trace capturing. On the one hand, we can learn about the communication details. On the other hand, the trace files for network simulation are prepared.

The process to get the final trace files contain three main procedures, namely a comprehensive understand of the communication mechanism in MXNet, defining the trace format and modifying the framework to implement the trace capturing method. We provide particular description of the communication mechanism in Section 3. The definition of trace format and modifications of framework are introduced in Section4. All the message passing operations in MXNet belong to the point-to-point communication type, and every communication operation corresponds to one record in the trace file of work node or server node. The communications with the scheduler node are peeled from the trace files to make it more convenient to analyze the communication records. Besides, our trace files can be used for figuring out various types of overheads.

The analyses of overheads is based on the trace files obtained from Lenet-5 and Resnet-50, using the datasets of MNIST and ImageNet. The experiment results show that the communication pattern of MXNet is the combination of long messages and short messages. The overlap ratio of computation and communication varies on different deep learning applications. The model size plays an important role in various types of overheads, and message length affects nothing but the search overhead. The overheads (except wait overhead) counted above share the same feature that they are the uncompressed processing time in the worker or server node, and they will be used in the simulation process to generating the corresponding processing time.

A series of works need to be done in the next time. Firstly, we plan to conduct the trace capturing experiment on a large-scale cluster. Secondly, we will make use of more neural networks to conduct the experiments, so we can summarize the communication features of different deep learning applications. Thirdly, we are going to carry out the network simulation experiments based on the captured trace files. Therefore, we are able to evaluate the performance of optimizing the functionalities of network devices. Finally, our work will be open source on github soon (https://github.com/CynthiaProtector/SketchDLC).

REFERENCES

[1] Caffe-MPI 2.0. 2018. https://github.com/Caffe-MPI/.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[3] Franisco J Andújar, Juan A Villar, José L Sánchez, Francisco J Alfaro, and Jesus Escudero-Sahuquillo. 2015. VEF traces: a framework for modelling MPI traffic in interconnection network simulators. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 841–848.

[4] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. 2017. S-Caffe: Co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 193–205.

[5] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. 2012. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590* (2012).

[6] Henri Casanova, Frédéric Desprez, George S Markomanolis, and Frédéric Suter. 2015. Simulation of MPI applications with time-independent traces. *Concurrency and Computation: Practice and Experience* 27, 5 (2015), 1145–1168.

[7] 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).

[8] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao family: energy-efficient hardware accelerators for machine learning. *Commun. ACM* 59, 11 (2016), 105–112.

[9] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *OSDI*, Vol. 14. 571–582.

[10] CNTK. 2018. https://www.cntk.ai/.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 248–255.

[12] DGX-1. 2018. https://www.nvidia.com/en-us/data-center/dgx-1/.

[13] DGX-2. 2018. https://www.nvidia.com/en-us/data-center/dgx-2/.

[14] Gloo. 2018. https://caffe2.ai/docs.

[15] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[16] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. 2016. Scalable hierarchical aggregation protocol (SHArP): a hardware architecture for efficient data reduction. In *Communication Optimizations in HPC (COMHPC), International Workshop on*. IEEE, 1–10.

[17] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 243–254.

[18] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[20] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 121–136.

[22] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 1–12.

[23] Edward Karrels and Ewing Lusk. 1994. Performance analysis of MPI programs. *Environments and Tools for Parallel Scientific Computing* (1994), 195–200.

[24] Benjamin Klenk and Holger Fröning. 2017. An overview of MPI characteristics of exascale proxy applications. In *International Supercomputing Conference*. Springer, 217–236.

[25] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[26] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[28] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).
[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[30] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 137–152.
[31] Jialin Li, Ellis Michael, and Dan RK Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 104–120.
[32] 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 *OSDI*, Vol. 14. 583–598.
[33] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv preprint arXiv:1712.01887* (2017).
[34] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 795–809.
[35] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L Wolf. 2014. Netagg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 249–262.
[36] José Miguel-Alonso, Javier Navaridas, and FJ Ridruejo. 2009. Interconnection network simulation using traces of MPI applications. *International Journal of Parallel Programming* 37, 2 (2009), 153–174.
[37] Programmable networks. 2018. https://www.nextplatform.com/.
[38] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. 1995. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, Vol. 44. IOS Press, 17–31.
[39] Scalable AI platform for Autonomous Driving. 2018. https://www.nvidia.com/en-us/self-driving-cars/drive-platform/.
[40] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. 2015. Going deeper with convolutions. CVPR.
[41] Francisco Trivino, Francisco J Andujar, Francisco J Alfaro, José L Sánchez, and Alberto Ros. 2011. Self-related traces: An alternative to full-system simulation for nocs. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE, 819–824.
[42] Dayong Wang, Aditya Khosla, Rishab Gargeya, Humayun Irshad, and Andrew H Beck. 2016. Deep learning for identifying metastatic breast cancer. *arXiv preprint arXiv:1606.05718* (2016).
[43] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric Xing. 2015. Distributed machine learning via sufficient factor broadcasting. *arXiv preprint arXiv:1511.08486* (2015).
[44] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P Xing. 2016. Lighter-Communication Distributed Machine Learning via Sufficient Factor Broadcasting.. In *UAI*.
[45] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888* (2017).
[46] Yang You, Zhao Zhang, C Hsieh, James Demmel, and Kurt Keutzer. 2017. ImageNet training in minutes. *CoRR, abs/1709.05011* (2017).
[47] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2017. 100-epoch ImageNet training with alexnet in 24 minutes. *ArXiv e-prints* (2017).
[48] ZeroMQ. 2018. https://en.wikipedia.org/.
[49] 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. *arXiv preprint* (2017).

Received February 2007; revised March 2009; accepted June 2009