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

With the fast development of deep learning (DL), the communication is increasingly a bottleneck for distributed workloads, and a series of optimization works have been done to scale out successfully. Nevertheless, the network behavior has not been investigated much yet. We intend to analyze the network behavior and then carry out some research through network simulation. Under this circumstance, an accurate communication measurement is necessary, as it is an effective way to study the network behavior and 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 DL training. In this article, we first provide detailed analyses about the communication mechanism of MXNet, which is a representative framework for distributed DL. Secondly, we define the DLC trace format to describe and record the communication behaviors. Third, we present the implementation of method for trace capturing. Finally, we make some statistics and analyses about the distributed DL training, including communication pattern, overlap ratio between computation and communication, computation overhead, synchronization overhead, update overhead, and so forth. Both the statistics and analyses are based on the trace files captured in a cluster with six machines. On the one hand, our trace files provide a sketch on the DLC, which contributes to understanding the communication details. On the other hand, the captured trace files can be used for figuring out various overheads, 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. 2019. SketchDLC: A Sketch on Distributed Deep Learning Communication via Trace Capturing. *ACM Trans. Archit. Code Optim.* 16, 2, Article 7 (April 2019), 26 pages.

<https://doi.org/10.1145/3312570>

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 to image

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

This project was partially supported by the National Science and Technology Major Projects on Core Electronic Devices, High-End Generic Chips and Basic Software under grant 2018ZX01028101.

Authors' addresses: Y. Xu, D. Dong (corresponding author), W. Xu, and X. Liao, National University of Defense Technology, 109 Deya Road, Changsha, Hunan, 410073, China; emails: {xuyemaovip, dong, xuweixia, xkliao}@nudt.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/04-ART7

<https://doi.org/10.1145/3312570>

recognition to breast cancer detection [43] to self-driving cars [38]. There are two main driving forces behind the momentum that DL has gained [5]: first is the public availability of versatile training datasets like ImageNet [16] and CIFAR [27], and the notable advances in computing capability turn out to be the second one [13]. The resurgence of DL has also triggered the developments of DL frameworks like MXNet [9], TensorFlow [3], Caffe [23], Microsoft CNTK [14], and Theano [6]. However, research efforts have been mostly concentrated on designing better and deeper neural networks (DNNs) to improve the accuracy of complex tasks. DNNs like GoogleNet [41], AlexNet [28], and ResNet [21] have a large number of parameters to be trained, which call for a 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 DL training.

To implement the distributed training, different DL frameworks utilize different mechanisms to accomplish the distributed communication, such as the ZeroMQ [49] in MXNet, gRPC in TensorFlow [3], NCCL in Caffe-MPI 2.0 [2], Gloo in Caffe2 [19], and some other communication libraries. Although a 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. First, the steadily increasing computational capability of general accelerators and the developments of specialized accelerators [8, 10, 11, 24] 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 are inevitable to scale out successfully.

A series of works have been done to reduce the communication overhead. With regard to the majorization designs in hardware, DGX-1 [17] and DGX-2 [18] by NVIDIA improve the communication efficiency greatly via the use of NVLink and NVSwitch technologies; the *Next Platform* presents the computational networking approach to handle the communication data directly in the network [35]; and the new HDR 200G switch and adapter of Mellanox supports in-network computing and in-network memory to improve the communication efficiency [34], and so forth. Optimizations from a software perspective include two main aspects: overlapping the computation and communication [5, 50] and reducing the network traffic [13, 20, 32, 47]. The overlapping strategy eliminates the unnecessary waiting overhead to make full use of the network and computation resources. However, the network reduction strategy is much more effective, and methods of pruning the redundant connections, weight sharing, and deep gradients compressing [32] can cut down the network traffic to a large extent.

We argue that the distributed communication efficiency can be improved by optimizing the function of the network device, which is different from the works mentioned earlier. Before the implementation in real hardware, we decide to evaluate the performance through network simulation. A comprehensive measurement of the deep learning communication (DLC) is necessary before the simulation. Besides, carrying out the simulation calls for two conditions. First is a model of the system that 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 an 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 [4]. With regard 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 a real system network. Synthetic traffic patterns come up to our mind first with regard to the generation of communication trace. This type of trace suits the system with a simple communication pattern, whose key features can be clearly expressed by mathematical expressions. However, the communication trace of most real applications cannot be easily described by mathematical expressions. The synthetic traces do not consider the interactions generated by real applications, whereas 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. Trivino et al. [42] propose a self-related trace model to capture the trace for a network-on-chip (NoC) simulator, whereas other works [4, 7, 33] present the mechanisms to get communication traces of MPI-parallel applications. Miguel-Alonso et al. [33] make use of the multi-processing environment (MPE) [25], which is a set of libraries and tools used for generating and analyzing traces of parallel applications, to get the trace files. Casanova et al. [7] come up with the time-independent trace replay framework to capture the time-independent traces. Andújar et al. [4] raise 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, and Klenk and Fröning [26] provide 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 a 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 preceding motivations, we take great efforts to capture the communication traces of DL applications based on MXNet, an efficient and flexible framework that we are familiar with. 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 work out various types of overheads. 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. Second, 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, using the MXNet: first, how the parameters and gradients are transmitted in MXNet during the training process; second, how to identify the communication operations that belong to the same iteration; third, how to describe the dependency relationships between various kinds of communication operations; and finally, determining the information that should be contained in the trace format to present comprehensive traces.

To tackle the preceding challenges, we carry out various attempts, and finally we come up with *SketchDLC*, which provides a sketch on DLC via trace capturing; it contains the following contributions. In the first place, we provide some analyses about the communication mechanism of MXNet. Second, we define a trace format to describe and record the communication operations. Third, we implement the method of trace capturing in MXNet and confirm the communication mechanism with the captured trace files. Last, we make some statistics about the communication features and analyses about various types of overheads.

The rest of the article is organized as follows. Section 2 provides an introduction to the large-scale distributed DL. Section 3 gives detailed insights on the communication mechanism of MXNet. Section 4 presents the format of DLC trace and the implementation of the trace capturing method. Section 5 describes the analyses on various types of overheads based on the obtained communication traces. Section 6 talks about the related work, and Section 7 concludes.

2 BACKGROUND

In this section, we provide a brief introduction to the distributed frameworks for large-scale DL and introduce a series of training strategies, such as data-parallel and model-parallel training, and synchronous and asynchronous training.

2.1 Distributed Frameworks for DL

With the fast growth in data volume and model size, distributed training turns out to be a prerequisite for solving large-scale DL problems. To implement the distributed training, two types of distributed training frameworks, the parameter server structure (PS) and the peer-to-peer structure (P2P) (Figure 1), are proposed.

PS and P2P. A parameter server is a distributed shared memory that offers systematic abstraction of iteration-convergent algorithms in data-parallel and model-parallel distributed ML [50]. 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. There is no server node in P2P; every worker node keeps all the parameters, and accomplishes the computing tasks and communication operations. Each worker will receive the gradients from other workers, then update the local parameters. The communication overhead of a naive P2P 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 [31]. Compared to PS, it is unnecessary to regain the parameters from server nodes to start the next iteration.

2.2 Synchronous and Asynchronous Distributed Stochastic Gradient Descent

In training neural networks, one epoch is a single pass through the whole training set. However, the full training set is usually divided into batches, and one iteration means the completion of training for one batch. Therefore, one epoch always contains a few iterations. During the distributed training with stochastic gradient descent (SGD), all the computation nodes conduct their own iterations with different batch training data. The synchronous distributed SGD performs the renewal of parameters with Equation (1) [32], whereas the asynchronous SGD update the parameters with Equation (2), 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 batch size, $B_{k,t}$ is the batch of the k^{th} node at iteration t , $\nabla f(x, w_t)$ is the gradients calculated by the auto-differentiation in the backward step for updating the parameters.

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

$$w_{t+1} = w_t - \eta \frac{1}{b} \sum_{x \in B_{k,t}} \nabla f(x, w_t) \quad (2)$$

The synchronous SGD will not conduct the update operation until the arrivals of all the corresponding gradients, leading to the synchronization overhead in the training process. The

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

Fig. 2. Data parallel and model parallel.

(a)

(b)

(c)

Fig. 3. Communication mechanism of MXNet.

asynchronous SGD is the SGD implemented without any locking, such as Hogwild [39], the server node can carry out the update operation as long as the gradients of one single node arrive. However, the asynchronous SGD is mainly applicable to sparse DL problems.

2.3 Data Parallel and Model Parallel

Figure 2 demonstrates the data parallel and model parallel. As for data parallel, each machine has one model replica, and the machines use different batches to conduct the forward and backward calculation. The server group is responsible for maintaining and updating the parameters. The Downpour SGD [15] is a classical algorithm for data parallel training. For larger models, the user may partition the model across several machines so that the responsibility for computation is assigned to different machines. Different machines have different parameters to train, and communication is needed among the machines (red lines). Even in cases where a node has multiple edges crossing a partition boundary, its state is only sent to the machine on the other side of that boundary once [15]. Nevertheless, there is no communication between the computation machines in data parallel mode, and every computation machine has all the parameters.

3 OUR INSIGHTS INTO THE COMMUNICATION MECHANISM OF MXNET

Different from getting the VEF traces by using the trace files generated by PARAVERT tools [37], 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 understanding of its communication mechanism.

3.1 MXNet Based on the PS

As a distributed ML library, MXNet is implemented on the basis of the PS, 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.

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 $\langle \text{key}, \text{value} \rangle$ 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 first (Figure 3(c)).

3.2 Details of the Communication Mechanism

Three main functions are included in the MXNet library, namely the resource management, and 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, whereas the communications with the scheduler node mainly happen in the beginning. This section provides some fine-grain communication analyses of MXNet.

3.2.1 Point-to-Point Message Passing. Most P2Ps make use of the MPI communication library, which contains abundant communication patterns, to implement their distributed communication, whereas the PS 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 PS-based framework, 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. In stage 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. In stage 2, the worker that ranks 0 sends the initialized parameters to the servers to do the initialization operation. In stage 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. Stages 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 requests for the corresponding worker and server, respectively. We define eight operation requests in the following, and they are introduced in their execution order. The order of requests in Figure 4 also matches their execution order in the worker and server nodes. During the training process, every parameter has its own series of requests in each iteration. Although the trace records of all parameters seem to be chaotic in the trace files, the request records for each parameter strictly keep the following order:

- ***Push_Send_Worker***: The worker node pushes the calculated gradients to the server node.
- ***Push_Recv_Server***: The server node receives the gradients pushed by the worker.
- ***Push_Send_Server***: The server node sends the *Push* confirmation message back to the worker node.
- ***Push_Recv_Worker***: The worker node receives the confirmation message of *Push* operation from the server.

Fig. 4. Operations of the worker and server nodes.

Fig. 5. Dependency relationships in the worker and server node.

- ***Pull_Send_Worker***: The worker node sends the *Pull* request to the server to get the parameters.
- ***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.
- ***Pull_Recv_Worker***: The worker node receives the parameters sent from the server.

3.2.3 Dependency Types. The 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 the own operations of each node, and there are eight types of data dependencies during the training process: four types for the worker node and four types for the server node. The dependency relationships are expressed through the communication requests (see Figure 5):

- ***PRW_on_PSW***: The *Push_Recv_Worker* operation request depends on the *Push_Send_Worker* operation request.
- ***PLSW_on_PRW***: The *Pull_Send_Worker* operation request depends on the *Push_Recv_Worker* operation request.
- ***PLRW_on_PLSW***: The *Pull_Recv_Worker* operation request depends on the *Pull_Send_Worker* operation request.
- ***PSW_on_PLRW***: The *Push_Send_Worker* operation request depends on the *Pull_Recv_Worker* operation request.
- ***PSS_on_PRS***: The *Push_Send_Server* operation request depends on the *Push_Recv_Server* operation request.
- ***PLRS_on_PSS***: The *Pull_Recv_Server* operation request depends on the *Push_Send_Server* operation request.
- ***PLSS_on_PLRS***: The *Pull_Send_Server* operation request depends on the *Pull_Recv_Server* operation request.
- ***PRS_on_PLSS***: The *Push_Recv_Server* operation request depends on the *Pull_Send_Server* operation request.

We further divide the eight types of data dependencies into two classifications: the *one-to-one* dependency and *one-to-many* dependency:

One-to-one: The current communication request is dependent on just one previous request in the same iteration.

One-to-many: The current communication request is dependent on a series of communication requests that belong to the last or the same iteration.

With regard to the worker node, *PRW_on_PSW*, *PLSW_on_PRW*, and *PLRW_on_PLSW* belong to the classification of *one-to-one* dependency, whereas *PSW_on_PLRW* is the *one-to-many* type. Different from the worker node, *PLRS_on_PSS* and *PLSS_on_PLRS* in the server node are the types of *one-to-one* dependency, whereas *PSS_on_PRS* (Synchronous) and *PRS_on_PLSS* 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* communication requests in the previous iteration to pull back all the parameters for computation.

3.2.4 Computation vs Communication. As mentioned earlier, 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 6). *Phase ①* is the computation-only stage, in which the worker node just conducts the computation task. *Phase ②* is the overlap stage, during which the communication process has begun 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 phases 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 phases 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

We put forward to obtain the communication trace with three targets. The first one is presenting a overview about the communication behaviors of each node. The second one is making some general statistics and analysis about the communication features and various overheads. The last one is laying the foundation for network simulation via providing distributed traces, which are obtained from real applications. Under this circumstance, an MXNet trace should contain the information needed to understand and replay the communication operations, as well as 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 reading and replaying the trace. We aim to provide a readable trace, and the trace 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.
- *length* is the size of delivered message measured in bytes. *Message* in each communication record contains the metadata and the data transmitted for neural network training, and thus the length consists of the data size of both, which is calculated in the message passing process and can be used directly.
- *num_pp* is the sequence number used for recording the dispatched order of *Push/Pull* operations, which can be dispatched before they are executed. Every two trace records in the same trace file and share the same *num_pp* correspond to the same operation. 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 results from the non-blocking communication mechanism in MXNet.

Table 1. DLC Trace Format

<i>id</i>	<i>src</i>	<i>dst</i>	<i>length</i>	<i>num_pp</i>	<i>operation</i>	<i>op_id</i>	<i>dep_type</i>	<i>d_time</i>	<i>time_sec</i>	<i>time_usec</i>	<i>id_dep</i>
• <i>operation</i> records the communication operation request of the node. Its format is <i>Push(Pull)_Send(Recv)_node</i>, just as the contents explained in Section 3.2.2. • <i>op_id</i> is the identifier of operation request in each node, which presents detailed information about the operation and is composed of three pieces of contents. The format is <i>key-operation_num-node</i>, where – <i>key</i> is the index of parameters in the neural network. – <i>operation_num</i> here is the communication identifier for the <i>Push/Pull</i> operation requests introduced in Section 3.2.2. Every training iteration contains four successive communication identifiers for the four operation requests in the worker or the server node. – <i>node</i> is the role played by the nodes, attached with the rank id (w1 represents worker 1, and s0 is server 0 in short). • <i>dep_type</i> is used to describe the dependency types introduced in Section 3.2.3, which takes values from 0 to 8. The value of 0 represents the independent records, and the other numbers tag the dependency types introduced in Section 3.2.3. • <i>d_time</i> is the dependency time measured in microseconds (μs). In fact, values of this field is used to record the <i>one-to-one</i> dependency time. With regard to the type of <i>one-to-many</i> dependency, <i>d_time</i> is not accurate because of the synchronization overhead and wait overhead, and that is why we introduce the <i>time_sec</i>, <i>time_usec</i> fields. • <i>time_sec</i>, <i>time_usec</i> are the moments each communication operation is conducted: <i>time_sec</i> is measured in seconds, and <i>time_usec</i> is measured in microseconds (μs). • <i>id_dep</i> is the identifier(s) of the record(s) on which the current record depends.											

When designing the trace format, we consider some properties of the DL application. For example, every parameter calls for a pair of *Push/Pull* operations to complete the update process, which correspond to eight communication requests: four in the worker node and four in the server node. To identify the communication records in the trace files that belong to the same parameter, we design the *op_id* and *operation* fields. The records with the same *key* in the *op_id* field belong to the same parameter, and the *operation* field presents the specific communication requests. Besides, the DL training is an iterative process and identifying each iteration is necessary for analyzing the trace files. Therefore, we add the *operation_num* in the *op_id* field, and it is used for recording the serial numbers of the communication requests. Since the parameters need four communication requests (a pair of *Push/Pull*) to update their value in each iteration, we use four continuous numbers to tag the four communication requests in the worker or server node. For example, if 5, 6, 7, and 8 belong to the same iteration, once the value of *operation_num* in the next record is 9, we know a new iteration begins. In addition, we also take the dependency relationships, various types of overheads, and the dispatch and execution orders of *Push/Pull* operations into consideration as well. As a matter of fact, we design the format from the view of DLC features.

After defining the format, we are able to obtain readable and comprehensive trace files. The trace files contribute to a detailed understanding of the communication mechanism. Besides, we can figure out various types of time overheads, which allows us to analyze the overlap ratio of communication and computation, and provides 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, whereas the communication overhead includes a series of other pivotal

Fig. 6. Analysis of computation and communication overheads.

Fig. 7. Computation and search overhead.

overheads, including *search overhead*, *update overhead*, *synchronization overhead*, and *wait overhead*. The specific meaning of each overhead is shown next:

- *Computation overhead*: The time used for the forward and backward computing process to calculate the gradients. The interval between the final *Pull_Recv_Worker* operation request of the last iteration and the last *Push_Send_Worker* of the current iteration is the computation overhead, as the T_0 in Figure 7, or the sum of overheads in phase ① and phase ② (Figure 6). We measure the computation from the last *Pull_Recv_Worker* because the parameters in the first few layers are finally pulled back. In the training process, computation overhead depends on the neural network, the batch size for each iteration as well as the configuration of the worker node. However, even with the same configuration, the computation overhead differs from node to node.
- *Communication overhead*: The time used for delivering the gradients and parameters. We define the time interval between the first *Push_Send_Worker* and the last *Pull_Recv_Worker* as the communication overhead. Although the update overhead, search overhead, and synchronization overhead are covered in the communication overhead, it is not the sum of them. In fact, the time for delivering the data and these overheads are mainly parallel relationships.
- *Search overhead*: The time for the server node to search the parameter according to the pull request from the worker node, as the T_1 in Figure 7. In fact, the search overhead is the value in the d_time field, which equals the interval between the record of *Pull_Recv_Server* and the *Pull_Send_Server* requests.
- *Synchronization overhead*: This only exists in the synchronous update mode, which results from the difference in computation capability and the network congestion. The time interval between the first and the last *Push_Recv_Server* operation records for the same parameter in one iteration is the synchronization overhead, as the T_2 in Figure 8. The synchronization overhead usually increases as the scale of the cluster increases. On the one hand, a cluster with larger scale may expand the difference in computation capability. On the other hand, a larger cluster will worsen the network congestion. Both of them will lead to a larger synchronization overhead.
- *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 8). After receiving the gradients from the worker node(s), the server node will conduct the update operation, and every parameter has its own update overhead. With regard to the asynchronous update mode, the update overhead is the d_time of *Push_Send_Server* records in the trace file, whereas this does not

Fig. 8. Synchronization and update overhead.

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 (receiver == scheduler)
5  | | keep the record in trace_sch_record.txt
6  | else /* receiver == 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 (receiver == scheduler)
13 | | keep the record in trace_sch_record.txt
14 | else /* receiver == 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

```

---

suit the synchronous mode, whose d_time of *Push_Send_Server* records 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 needs to get all the updated parameters from the server group. The interval between the first *Pull_Recv_Worker* and the last *Pull_Recv_Worker* is the *wait overhead*, and every worker node has its own 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. The preceding overhead analysis is the basis for working out the program for overhead calculation.

4.3 Implementation

We initially planned to develop an independent lightweight library to obtain the communication trace. Some necessary information, which is redundant in the distributed communication process, is not available in the final communication APIs (*SendMsg()* & *RecvMsg()*). To provide a readable and useful communication trace for simulation and analysis, we must modify the source code of MXNet to acquire the extra information. This section first presents the available information for trace capturing and then elaborates on the main modifications for obtaining the additional information.

4.3.1 Available Information. All the communication requests in MXNet are finally completed by invoking the *SendMsg()* and *RecvMsg()* functions, and their arguments belong to the data type *Message* (**message.h**). With the *Message* data, we can obtain the node ids of the sender and receiver, and then the *src* and *dst* can be obtained. The communication operation can also be recognized via the variable of *push*: 0 for *Pull* and 1 for *Push*. Furthermore, 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 deduce 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()* functions, 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 DLC traces, we design two new functions in the **zmq_van.h** file, *Trace_Send_()* and *Trace_Recv_()*, the former one is being embedded in *SendMsg()* and the latter in *RecvMsg()*. Algorithm 1 illustrates the overall procedures of *Trace_Send_()* and

is responsible for all the sending operation records, namely *Push_Send_Worker*, *Pull_Send_Worker*, *Push_Send_Server*, and *Pull_Send_Server*. The algorithm for *Trace_Recv_()* is similar to Algorithm 1, and it is responsible for all the receiving operation records.

Apart from the available information mentioned earlier, keeping a piece of the operation record calls for some extra information. Thus, *num_pp* needs the dispatch number of the *Push/Pull* operations, *op_id* demands the *key* and *operation_num*, and *id_dep* is dependent on *op_id*. The *id*, *time_sec*, and *time_usec* can be acquired directly and have nothing to do with the MXNet library. Therefore, we 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_()*. When the necessary information is ready, a convenient method is needed to figure out the *op_id* and *d_time*, resulting in the use of the *unordered_map* data structure ([zmq_van.h](#)) and the definition of *Trace_* ([message.h](#)). The *d_time* and *op_id* can be easily acquired according to the time and *operation_num* in *Trace_*. Once *op_id* is captured, *id_dep* can also be expressed according to the dependency relationships.

Thereafter, we can collect the information needed in the trace format, which eventually contributes to a readable and comprehensive DLC trace.

5 A SKETCH ON DISTRIBUTED DLC AND OVERHEADS

As is stated earlier, we aim at providing useful and readable trace files for a comprehensive measurement of the DLC. We eventually fulfill our target by introducing the DLC trace format. In this section, we first introduce the neural networks we used and the hardware configuration in our experiments. Then, we present some statistics about the communication features. Finally, we analyze various types of overheads in the training process and the effects from different batches, different cluster scales, and so forth.

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 six machines. Each machine has 40 2.60GHz Intel cores, two K80 (dual GPUs) GPUs, 64G memory, and a 56Gbps IB network card. All the machines are installed with Ubuntu 16.04, CUDA 8.0, and cuDNN 5. 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 (one scheduler node, one server node, and four worker nodes), 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. We used two datasets for our experiments: (1) the MNIST [29] dataset, which contains a training set of 60,000 examples and a test set of 10,000 examples, and (2) the ImageNet [16] ILSFRC2012 dataset, the subset of ImageNet containing 1,000 categories and 1.2 million images.

Neural networks. Corresponding to the datasets, a set of representative neural networks with different model sizes are applied in our experiments (Table 2) [21, 22, 28, 30, 40]. The model size corresponds to the communication volume between workers and parameter servers processes in each iteration, which is the sum of sizes of all the variable tensors and the metadata for them during the communication process. These benchmarks cover both computation- and communication-intensive workloads. For example, the ResNet-152 model is a typical computation-intensive workload, whereas the VGG-16 is mainly a bottleneck in communication.

5.2 General Statistics of Trace Files

This section presents our general observations about the MXNet communication with the captured trace files, such as the message size, data volume, and communication pattern.

Table 2. DL Benchmarks

Benchmark	Model Size (MB)	Variable Tensor#	Million Mult-Adds
LeNet-5	1.72	8	10
Inception-bn	45.2	278	2,500
ResNet-50	102.2	156	2,040
Alexnet	203.4	16	660
ResNet-152	240.8	463	5,760
VGG-16	553.4	32	15,350

Table 3. Data Transmitted in One Iteration

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 4. Data Transmitted in One Iteration

Model	Push_Send (KB)	Push_Recv (KB)	Pull_Send (KB)	Pull_Recv (KB)
LeNet-5	1720	0.15	0.22	1720

Message size. The process of updating one parameter demands four basic communication operation requests in the worker nodes and server nodes, respectively. Take LeNet-5 as an example: Table 3 depicts the message sizes for different parameters during the communication process of worker nodes. Columns 2 through 5 correspond to the operation requests of *Push_Send_Worker*, *Push_Recv_Worker*, *Pull_Send_Worker*, and *Push_Recv_Worker*. Columns 2 and 5 record the message lengths of gradients and parameters transmission, and they vary greatly for different *keys*: the shortest messages is 73 bytes, whereas the longest message is up to 1.6MB. Column 3 records the sizes of confirmation messages for *Push_Send_Worker* requests, whereas Column 4 keeps the message sizes of *Pull_Send_Worker* requests. Values of both columns are constant and small in the whole training process.

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, and acquiring the updated parameters are the operations done in each iteration. The total message size contained in one iteration is constant (Table 4), and thus the data volume can be worked out according to the number of iterations.

Communication pattern. According to the message lengths shown in Table 3, communication traffic during the training process is the combination of long messages and short messages, and this is decided by the features of DL applications. *Push_Send_Worker* and *Pull_Recv_Worker* are responsible for the transfer of gradients and the updated parameters, and these are the combinations of long and short messages. The transmission of confirmation message and request message are conducted through the *Push_Recv_Worker* and *Pull_Send_Worker* operations, and these are short messages that keep constant in the training process. Contrapose to 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 handle all of them, which degenerates the distributed performance. Besides, the order of dispatching *Push* and *Pull* operations keep the ascending order in the beginning, whereas they are out of order in the completion phase, and we can notice the difference from the captured trace files. This results from the fact that the before the gradients or parameters are ready, the *Push* and *Pull* operations are dispatched first.

5.3 Overheads Analysis

We carry out the experiments on a cluster with six machines; however, our method is applicable to large-scale clusters. This section demonstrates that the obtained MXNet trace provides access to detailed analysis of some overheads. The overhead statistics provided here are based on the trace files of the benchmarks mentioned earlier, and we use 300 training iterations for each neural

Fig. 9. Trend of computation overheads with four workers.

Fig. 10. Computation overheads with four workers.

network: 300 is not a fixed number, and it can be replaced by other numbers as long as the results of statistics are convincing.

When considering the distributed training, the batch size per node, cluster scale, model size, and the number of computations are the main factors. Changing the batch size and the number of computations impose influences on the computation overheads, and various cluster scales and models sizes lead to different communication overheads and synchronization overheads. For update overhead and search overhead, which are the overheads in the server node, a larger cluster scale means more operations and leads to greater overheads. Therefore, we make the overheads statistics based on their own factors in this section.

5.3.1 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 the *Push_Send_Worker* operation request depends on the gradients calculated in the backward process. When the final *Push_Send_Worker* is dispatched, this means that the computation process is complete.

Impact of different batch sizes. To analyze the impact of different batch sizes on the various types of overheads, we run the six benchmarks with different batch sizes. When running them with different batch sizes, the number of workers is 4. The GPU memory of K80 is 12GB, and we pick the batch sizes according to the model and available GPU memory in our experiments.

Figure 9 illustrates the trend of the average computation overheads using various batch sizes. The computation overheads of the benchmarks vary greatly, and it is inconvenient to put them together as a single figure. Therefore, we normalize the computation overheads for each benchmark. The computation overheads of all the benchmarks increase as the batch size increases, except LeNet. When the batch size is 128, there is a slight drop in the computation overhead of LeNet. This is because the model size of LeNet is small and the number of computation cycles required for its training is also small. Changing the batch size from 32 to 128 has little influence on the computation overhead and even a slight drop due to the fluctuation in the overhead. Even though the computation overheads of all benchmarks show an increasing trend, we can divide those models into two groups: Inception-bn, ResNet-50, and ResNet-152 in the first group and LeNet, AlexNet, and VGG-16 in the second group. The computation overheads of the first group apparently increase along with the increasing batch size because models in this group are computation intensive, and the batch size has a great influence on the requirement of the computation cycles. The computation overheads of the second group increase slightly in the beginning, proving that models in this group are not computation intensive and that a slight increase in batch size do not clearly affect the computation overhead increase. As the batch size goes up further, the computation overheads begin to increase because a relatively large increase in batch size does increase the requirement of

Fig. 11. Increase rate of forward and backward overheads and their proportions in the computation overhead growth rate under different batch sizes.

computation cycles. Besides the batch size, various neural networks also lead to different number of computations, and the influence on computation overhead is discussed later (Figure 10).

The increase in computation overheads includes the increase of time for forward computing and backward computing. We define the time interval between the final *Pull_Recv_Worker* of the previous iteration and the first *Push_Send_Worker* of the current iteration as the *forward time*, and the interval between the first *Push_Send_Worker* and the last *Push_Send_Worker* of the same iteration as the *backward time*. In fact, the forward time equals phase ① in Figure 6, whereas the backward time is nearly equal to phase ②. With the captured trace files of the worker nodes, we calculate the forward and backward overheads separately and then calculate their increase rates using Equation (3). We also evaluate their contributions to the increase rate of the computation overhead with Equation (4). *FW_Rate* and *BW_Rate* are the forward increase rate and backward increase rate, respectively. *FW_Contr* and *BW_Contr* correspond to the proportions of forward and backward overheads in the computation overhead growth. In addition, *batch(i)* corresponds to the i^{th} batch size in our experiments. If *batch(i)* is 64, then *batch(i-1)* is 32. *FW* and *BW* are the forward and backward computational overheads, respectively, in the same iteration. *CP* represents the computation overhead in the training process.

$$FW\_Rate_{batch(i)} = \frac{FW_{batch(i)} - FW_{batch(i-1)}}{FW_{batch(i-1)}}; BW\_Rate_{batch(i)} = \frac{BW_{batch(i)} - BW_{batch(i-1)}}{BW_{batch(i-1)}} \quad (3)$$

$$FW\_Contr_{batch(i)} = \frac{FW_{batch(i)} - FW_{batch(i-1)}}{CP_{batch(i-1)}}; BW\_Contr_{batch(i)} = \frac{BW_{batch(i)} - BW_{batch(i-1)}}{CP_{batch(i-1)}} \quad (4)$$

Figure 11 illustrates the increase rates of forward and backward overheads of three benchmarks and their increase proportions in the computation overhead growth rate. To get rid of the node, which will crash when the batch size is near its maximum value, we run all the applications with only two workers. Therefore, we can get one more set of experimental data. According to the preceding three subgraphs, it is notable that the batch size imposes similar influences on the forward and backward overheads. A larger batch size usually leads to a greater increase rate, but the gaps are not obvious when the batch size is greater than 64. The three subgraphs in the following illustrate the increase proportions of forward and backward overheads in the computation overhead, and we can notice that the backward overhead contributes more to the increase of computation

Fig. 12. Communication overheads with four

Fig. 13. Communication overheads of different model sizes.

overhead. This results from the fact that in the computation process, the value of backward overhead is much greater than the forward overhead. Our statistics show that the average ratios between backward and forward overheads for Inception-bn, ResNet-50, and ResNet-152 are $2.06x$, $2.37x$, and $2.38x$ separately. Besides, the ratios under different batch sizes just vary slightly.

Impact of different number of computations. Different neural networks have different model sizes and different numbers of computations. The numbers of computations of ResNet-50 and VGG-16 are around $3.1x$ and $23.3x$ those of AlexNet, whereas the calculation amount of VGG-16 is around $7.5x$ that of ResNet-50. Figure 10 shows the computation overheads of three benchmarks at different batch sizes and also illustrates the comparison in computation overheads between every two benchmarks. Both the values of *Res vs Alex* and *VGG vs Alex* increase as the batch size increases and reach their maximum values when the batch size is 256. The value of *VGG vs Res* experiences a slight drop with the growth of batch size, and the average value is 1.66 . Considering the number of their computations, we can conclude that the number of computations does have an impact on the computation overhead, but this is not a linear relationship. Furthermore, the computation overheads are not mainly affected by the number of computations. The calculation amount of VGG-16 is nearly $21x$ that of AlexNet, but its overhead is only $3.12x$ greater when the batch size is 32. There should be some additional overheads in the computing process, which take up a big part of the computation overhead. However, as the batch size increases, time that is purely used for computing increases, whereas the additional overheads do not increase as much as the pure computing time; then, the values of *VGG vs Alex* and *Res vs Alex* begin to increase. The value of *VGG vs Res* drops slightly because the increase rate of ResNet-50 is a slightly bigger than that of VGG-16. In fact, from Figure 11, it is noteworthy that when we double the batch size, the increase rate of the computation overhead does not double as well. This is because the time that is purely used for computing is just a part of the computation overhead.

5.3.2 Communication Overhead. We define the time interval between the first *Push_Send_Worker* and the last *Pull_Recv_Worker* in the same iteration as the communication overhead, which is mainly affected by the model sizes and the scale of the cluster. In fact, the communication overhead covers the synchronization overhead, update overhead, and search overhead. However, these three types of overheads take up only a small part of the communication overhead. Figure 12 presents the trend of communication overheads using different batch sizes, and it is noteworthy that the communication overheads fluctuate slightly alongside the increase in batch size. In this section, we introduce the impacts of different model sizes and cluster sizes.

Impact of different model sizes. The model sizes listed in Table 2 are calculated using the captured trace files, which cover the size of the metadata and keys. This is because the metadata and keys will also be transmitted in the training process along with the parameters. To fully analyze

Fig. 14. Various types of overheads at different cluster sizes. The batch size is 64.

the influences of different model sizes on the communication overhead, we run the benchmarks on clusters of different sizes and compare the corresponding communication overheads (Figure 13). The batch size of the benchmarks is set to 64. We do not use LeNet-5 in this section because its model size and communication overhead are too small. Table 2 illustrates the model size of each benchmark, and the model size of ResNet-50 is 2.26x that of Inception-bn (marked as *Res50 vs Inc*); the values of *Alex vs Res50*, *Res152 vs Alex*, and *VGG vs Res152* are 1.99x, 1.18x, and 2.30, respectively. As shown in Figure 13, we list the overheads according to the model size, from small to large, and we compare the overheads between every two adjacent benchmarks. It is obvious that the model size affects the communication overhead. However, the extent of the impact is related to the model size of the network model. The value of *Res50 vs Inc* is 2.26x, whereas the average communication overhead of ResNet-50 is only 1.35x that of Inception-bn. The value of *Alex vs Res50* is 1.99x, and the average communication overhead ratio is 1.82x. The model size ratio and the communication overhead ratio gradually become closer. Considering the values of *Res152 vs Alex* (1.18x) and *VGG vs Res152* (2.30x), the average communication ratios are 1.26x and 2.38x, respectively, which exceed their ratios of model sizes. Therefore, a larger model size has more of an influence on the communication overheads than a smaller one. A larger model means that more messages need to be transmitted, which will probably deteriorate network congestion worse, bringing about more communication overheads than models of small sizes.

Impact of different cluster sizes. To analyze the effects of different cluster sizes on the communication overhead, we run the benchmarks on clusters of different sizes, from 1 to 4, and the batch size of the applications is set to 64. Figure 14 shows the communication overheads using different cluster sizes, together with other types of overheads. We combine these overheads to make it convenient to conduct a comprehensive comparison between them. Furthermore, it is unnecessary to divide them into different groups according to the type of overhead. From Figure 14, we can see that the cluster size obviously affects the communication overhead; the relationship between the cluster size and the communication overhead are almost linear. In terms of the benchmarks of LeNet-5, AlexNet, and VGG-16, the communication overhead is greater than the computation overhead with only one worker, and as the cluster size increases, the gap between them becomes larger. This is because the computation overhead fluctuates slightly when the cluster size varies, whereas the communication overhead increases sharply. For Inception-bn, ResNet-50, and

Fig. 15. Overlap ratio at different batch sizes.

Fig. 16. Overlap ratios of various cluster sizes.

ResNet-152, the communication overhead is less than the computation overhead when the cluster size is small. However, the communication overhead becomes greater when the cluster size increases. Therefore, as the cluster size gets bigger and bigger, the communication overhead will be a severe bottleneck, which must be handled properly. Just improving the computational capability may not be able to bring about a satisfactory improvement in performance. Furthermore, when the cluster size is sufficient large, all the neural networks will be communication intensive.

5.3.3 Overlap Ratio of Computation and Communication. With the captured trace files, we can easily calculate the overlap ratio between computation and communication. The overhead in phase ① (Figure 6) is the interval between the last *Pull_Recv_Worker* of the last iteration and the first *Push_Send_Worker* in the current iteration, which equals the time for forward computing. The time interval between the first *Push_Send_Worker* and the last *Push_Send_Worker* equals to the overhead in phase ②, which is nearly equal to the time for backward computing. As for the communication overhead in phase ③, this is the interval between the first *Push_Send_Worker* and the last *Pull_Recv_Worker* in the same iteration. Therefore, the overlap ratio can be described as $T_b/(T_f + T_c)$, where T_b means the time for backward computing, T_f represents the time for forward computing, and T_c is the communication overhead.

Overlap ratios under different batch sizes. To analyze the overlap ratios of various batch sizes, we run the benchmarks on a cluster with four worker nodes. Figure 15 illustrates the overlap ratios of all benchmarks using different batch sizes. It is apparent that the overlap ratios of all applications increase with larger batch sizes, just as the computation overheads. From Figure 12, we know the communication overhead (phase ③) fluctuates slightly as the batch size changes. In fact, the overlap time (phase ②) is nearly equal to the time for backward computing in the training process. Figure 11 proves that the backward overhead contributes more to the increase of computation overhead than the forward overhead (phase ①). Therefore, the overlap ratio increases as the batch size grows in accordance with $(a/b) < (a + m)/(b + m)$. However, the benchmarks can be divided into two groups: LeNet-5, AlexNet, and VGG-16 in one group and Inception-bn, ResNet-50, and ResNet-152 in the other one. According to Figure 9, the computation overheads of the first group are much less sensitive to the increase in batch size than the second group. The low increase rate of computation for the first group leads to its low increase rate in overlap ratio. What should be noticed is that a larger overlap ratio here does not mean shorter training time, because the computation overhead goes up accordingly.

Impact of different cluster sizes. To analyze the effects of various cluster sizes on the overlap ratio, we run the benchmarks on clusters of different sizes, and the batch size is set to 64. Figure 16 illustrates the trends in overlap ratios, and the overlap ratios of all benchmarks decrease when the cluster size increases. From Figure 14, it is noteworthy that the computation overheads of the benchmarks change little when the cluster size increases, whereas the communication overheads

increase sharply. As explained earlier, the overlap ratio is described as $T_b/(T_f + T_c)$; thus, its value will go down as the cluster size increases. When the number of workers is 1, the overlap ratios of ResNet-50, ResNet-152, and Inception-bn are all above 0.6, and their overlap time is almost equal to their communication time (Figure 14), which means that the communication process is near to its end when the backward computing completes. We classify these benchmarks as computation intensive. Different from the computation-intensive benchmarks, the overlap ratios of LeNet-5, AlexNet, and VGG-16 are all near to 0.2, and their communication overheads are much greater than the computation overheads. These benchmarks are communication intensive.

From the preceding discussion, we can draw several conclusions. First, a larger batch size has a positive effect on the overlap ratio, whereas a larger cluster size has a negative effect. Second, the overlap ratios of computation-intensive benchmarks fluctuate much greater than the communication-intensive ones when the batch and cluster sizes change. Third, adjusting the cluster size should be combined with a change in the batch size to protect the overlap ratio from varying sharply.

5.3.4 Synchronization Overhead. The synchronization overhead is a part of the communication overhead that only exists in the synchronous training mode. It results from the different computational capabilities of the worker nodes and the influences of network congestion. The trace file of the server node can be used for calculating the synchronization overhead. In the training process, every parameter has its own synchronization overhead. Therefore, we identify all the *Push_Recv_Server* operation records for each parameter in the same iteration and calculate the time interval between the first and last *Push_Recv_Server* records. The calculated value is the synchronization overhead for that parameter. Figure 12 shows that the communication overheads fluctuate slightly with the growth in batch size. Therefore, we do not consider the influences of various batch sizes on the synchronization overheads; rather, we analyze the impacts of different cluster sizes and model sizes in the following content.

Impact of different cluster sizes. Cluster size has an indirect effect on the synchronization overhead because a larger cluster will probably increase the difference in computation capability between nodes and may also adversely affect the network congestion. To analyze its influences, we run the benchmarks on clusters of different sizes with the batch size set to 64. Figure 17 lists the synchronization overheads of five benchmarks. Considering that the synchronization overhead of LeNet-5 is too small, we simply omit its data from this figure. We compare the overheads of each benchmark using different cluster sizes. The 1.69 in the figure means the synchronization overhead of Inception-bn with three workers is 1.69x that with two workers. The 1.66 means that the overhead of four workers is 1.66x that with three workers. For each benchmark, the synchronization overhead has similar growth rates when the number of clusters is increased by 1 (1.69 vs 1.66, 1.94 vs 2.07, 1.94 vs 1.95). Therefore, it may be concluded that the cluster size greatly affects the synchronization overhead. Based on Figure 14, the synchronization overhead does play an obvious role in the increment of communication overhead along with the growth in cluster size.

Impact of different model sizes. In addition to cluster size, model size also plays an important role in affecting synchronization overhead. A larger model size usually leads to a greater synchronization overhead because it usually lead to worse network congestion than a small one. Table 5 presents comparisons of synchronization overheads between two benchmarks of different model sizes. Taking 2.34 as an example, this means that when the number of workers is 2, the synchronization overhead of ResNet-50 is 2.34x that of Inception-bn. The seventh row shows the comparisons of model sizes between two benchmarks; 2.26 means that the model size of ResNet-50 is 2.26x that of Inception-bn. It is apparent that the model size has a great impact on the synchronization overhead. Nevertheless, the influence of the small model size is not as great as the influences of the

Fig. 17. Synchronization overheads at various cluster sizes. The batch size is 64.

Table 5. Comparisons of Synchronization Overheads

No. of Workers	Res50 vs Inc	Alex vs Res50	Res152 vs Alex	VGG vs Res 152
2	2.34	1.42	1.24	3.35
3	1.96	1.94	1.52	2.73
4	2.11	2.25	1.48	2.64
Average	2.14	1.87	1.41	2.91
Model Size	2.26	1.99	1.18	2.30

(a) LeNet-5

(b) AlexNet

(c) VGG-16

(d) Inception-bn

(e) ResNet-50

(f) ResNet-152

Fig. 18. Synchronization, update, and search overheads with four workers. The batch size is 64.

large model size. As the second and third columns illustrate, the ratio of model size is greater than the average ratio of synchronization overheads. In the meanwhile, the fourth and fifth columns show that the average ratio of synchronization overheads is greater than that of model sizes.

Synchronization overheads for different parameters. In the training process, each parameter has its own synchronization overhead. LeNet-5 and AlexNet only have 8 and 16 parameters, respectively, and thus we list the synchronization overheads of all parameters. For the other four benchmarks, we only list the overheads of 20 parameters. These 20 parameters range from the parameter of hundreds of bytes to the parameter of the maximum message length in each benchmark. Figure 18 demonstrates the synchronization overheads of the benchmarks together with the update overheads and search overheads, which will be discussed in the next section. The blue curves represent the synchronization overheads; the keys are the index of parameters, and they are in ascending order. A greater key means that the parameter belongs to a deeper layer of the neural network. Taking AlexNet as an example, 0 corresponds to the parameter of the first layer, whereas 15 is the index of the parameter in the last layer. It is apparent that the synchronization overheads of all benchmarks start to decrease when the key is larger than a specific value: 11 for AlexNet, 26 for VGG-16, 144 for Inception-bn, and so on. When the key takes the maximum value, the synchronization overhead takes the minimum value. This is because the network is idle at the beginning of communication. During our experiments, we do not run any other computation or communication tasks in all the nodes. In the forward computing process, no data are transmitted through the network. Once the backward computing process starts and the gradients of the last layer are available, the gradients are pushed into the network. In the beginning of communication, there is no network congestion, and the gradients from different worker nodes can

Fig. 19. Update overheads of four benchmarks at different cluster sizes. The batch size is 64.

be passed smoothly to the server node; thus, the synchronization overhead is low. As the backward process continues, more and more gradients are pushed into the network, then the network congestion increases and the synchronization overhead starts to increase. When the degree of network congestion is stable, the synchronization overhead is stable as well. After the completion of backward computing, no more gradients are pushed into the network, and the degree of network congestion is alleviated. Therefore, the synchronization overheads drop slightly. Furthermore, we can notice that the minimum synchronization overheads of AlexNet and VGG-16 are higher than those of Inception-bn, ResNet-50, and ResNet-152. This is because the former two benchmarks are communication intensive, whereas the latter three models are computation intensive. The time interval between the gradients pushed into the network is longer for the computation-intensive benchmarks. Their network congestion levels are better than the communication-intensive ones, leading to lower synchronization overheads. We can also conclude that the synchronization overhead of each parameter has little to do with the message length. This is because the keys listed in the figures correspond to the parameters, which vary greatly in message lengths, whereas the synchronization overheads do not fluctuate obviously as the message lengths change.

5.3.5 Update Overhead. As for the asynchronous training mode, this is equal to the d_time of *Push_Send_Server* operation records, and no more calculations are needed. Different from the asynchronous mode, update overhead under the synchronous mode calls for additional disposals. The time interval between the last *Push_Recv_Server* and the first *Push_Send_Server* for the same parameter is the update overhead. Update overhead belongs to the server node, and it is not related to the computation in the worker node or communication in the network. We discuss the effects from different cluster sizes next, because more workers lead to more data handling in the server node.

Impact of different cluster sizes. To update one parameter under the synchronous training mode, the server node needs to wait for the arrivals of the corresponding gradients in all workers. More worker nodes means that there are more data that need to be dealt with in one update operation; thus, a larger cluster size will probably lead to a greater update overhead. Figure 19 demonstrates the update overheads of different parameters using different cluster sizes, and the batch size is set to 64. For benchmarks that have more than 20 parameters, we only list the overheads of 20 parameters, whose message lengths are of different magnitudes. The keys in the figures follow a strict ascending order. From the curves in Figure 19, it is noteworthy that a larger cluster size contributes to larger update overheads. However, the influence on the parameters with small keys are much more apparent. With the increase in keys, the update overheads using different cluster sizes begin to be similar or equal. When the key is greater than a specific value, the update overhead begins to decrease as the value of the key increases. We attribute this finding to the situation of the server node being idle at the start of updating operations. During our experiments, we do not run any other tasks on the server node. When the gradients of the last layer or the last few layers are available, they are transmitted to the server node to conduct the update operations. The idle server node is able to handle these tasks immediately, leading to low update overheads. As more and more gradients are delivered, the resource competition in the server node makes the update

Fig. 20. Search overheads of four benchmarks at different cluster sizes. The batch size is 64.

overhead increase. It is the sequence of the parameters that plays an important role in affecting the update overhead instead of the message length. It is important to note that the fluctuations of update overheads vary from one benchmark to another. We figure out the update overhead according to the time interval between the last *Push_Recv_Server* and the first *Push_Send_Server* for the same parameter in the trace file. Nevertheless, sometimes there are *Push_Recv_Server* or *Push_Send_Server* records of other parameters between them, with the execution of other operation requests prolonging the update overhead. As the number of other operation requests varies, the value of the update overhead varies. The features of each benchmark mainly result from their communication characteristics, which affect the number of other operation requests between the *Push_Recv_Server* and *Push_Send_Server*.

5.3.6 Search Overhead. Search overhead is the time spent by the server node seeking the parameter according to the *Pull_Recv_Server* request, and this 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 the *d_time* of *Pull_Send_Server* operation records. The search overhead is related to the cluster size and message length of the parameter. More worker nodes lead to more *Pull_Recv_Server* requests, and the larger message length contributes to longer times for visiting the memory.

Figure 20 illustrates the search overheads of different parameters using different cluster sizes, and the batch size is 64. We can see that a larger cluster size leads to larger search overheads. Just like the influences on update overheads, the effects on parameters with small keys are more apparent. The search overheads of Inception-bn and ResNet-50 first increase and then decrease in line with the increase in keys. Different from these two benchmarks, the overheads of AlexNet and VGG-16 are relatively stable in the beginning and then start to decrease, except for some parameters with very large message lengths (10, 12 for AlexNet and 26, 28 for VGG-16). The message lengths of 10 and 12 in AlexNet are 104MB and 67.1MB, which are much larger than the message lengths of the other parameters. The same applies to the lengths of 26 and 28 in VGG-16, at 411.0MB and 67.1MB, respectively. Therefore, the message length plays a role in affecting the search overhead when it is sufficiently large. Otherwise, it is the sequence of the parameters that decides the search overhead instead of the message lengths. As for VGG-16, the overheads of 0, 1, and 2 are much greater than most of the following keys. We check the trace file of the server node and notice that for 0, 1, and 2, there are a series of other *Pull_Send_Server* operation records between their *Pull_Recv_Server* and *Pull_Send_Server* records, leading to larger search overheads. We also check the trace files of Inception-bn and ResNet-50. The increase in search overheads is mainly due to the same reason.

Wait overhead. This is 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 dependent on some of the other overheads mentioned earlier. Considering that wait overhead is a combination of other types of overheads, we omit it from our statistics.

Fig. 21. Various types of overheads at different batch sizes. The cluster size is 4.

5.3.7 Comparisons of Various Overheads. Figures 14 and 21 illustrate various overheads at different cluster sizes and batch sizes (the cluster size is 4). From Figure 14, we can notice that the time for transmitting data always makes the bottleneck from the curves of the communication overheads. Although the synchronization overhead, update overhead, and search overhead are parts of the communication overhead, they take only small proportions unless the model size is large, such as with VGG-16 and ResNet-152. Their synchronization overhead increases obviously as the cluster size goes up. For computation-intensive benchmarks, the computation overhead is also a severe bottleneck, although not for the communication-intensive benchmarks. In fact, we can draw similar conclusions from Figure 21. As the batch size grows, the computation overhead increases accordingly (overlap time is part of the computation overhead) and the other types are not obviously affected. For benchmarks LeNet-5, AlexNet, and VGG-16, which are communication-intensive ones, the time for data transmission is the most severe bottleneck as the batch size varies. In the meantime, the synchronization and computation overheads also play apparent impacts. With regard to the computation-intensive benchmarks, the computation overhead and time for data transmission are both the bottlenecks when the batch size is large enough.

6 RELATED WORK

With the rapid development of DL, the complexity of neural networks and the expansion of training data have driven the requirement for computing power and communication bandwidth greatly, thus limiting the efficiency of distributed training to some extent [13]. In response to this problem, a series of research works in hardware and software has been done to scale out successfully, which can be summarized from four different layers, namely the application layer, framework layer, architecture layer, and network layer.

With regard to the application layer, Han et al. [20] propose to prune redundant connections to simplify the neural network; TernGrad [46] uses ternary gradients to aggressively reduce the communication time. With regard to the framework layer, S-Caffe [5], Project Adam [13], Hogwild [39], ISGD [44], Bosen [45], and others are designed to enhance the distributed scalability, and improving the communication mechanism is their shared target. With regard to the architecture layer, DianNao [8], DaDianNao [11], PRIME [12], OLAcel [36], and a series of other works carry out the optimizations by leveraging the characteristics of DL. With regard to the network layer,

the application of RDMA is getting popular. Xue et al. [48] replace the gRPC with RDMA to build a fast distributed DL platform. The HiDL team at Ohio State University [1] is also conducting research to provide RDMA support for TensorFlow. Most of these optimization works focus on overlapping the computation and communication, reducing the communication traffic, and adopting an efficient communication library, and the network behavior has not been investigated much yet. Therefore, we conduct a measurement of the network behavior with SketchDLC. Besides, we intend to conduct some optimizations on the network protocols and devices by way of simulation, and accurate network simulation calls for comprehensive measurements and accurate communication trace as well.

7 CONCLUSION

We mainly introduce three parts of contents in this article: the reasons for obtaining the communication traces of DL applications, the coarse-grain procedures on how to get the final trace files, and some analyses based on the captured trace files.

Providing a comprehensive measurement on the distributed DLC 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 first by 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 DLC by trace capturing. On the one hand, we can learn about the communication details and overheads. On the other hand, the trace files for network simulation are prepared.

The process to get the final trace files consists of 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 a particular description of the communication mechanism in Section 3. The definition of trace format and modifications of framework are introduced in Section 4. In fact, the defined trace format is applicable to DL frameworks that conduct their communication process by way of (key, value). However, the implementations vary from one framework to another to acquire the information needed in the trace format.

The analyses of overheads is based on the trace files obtained from the benchmarks, 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 model size, cluster size, batch size, message lengths, and sequence of the parameters impose more or less influences on various types of overheads and the overlap ratio; the details are introduced in Section 5. The overheads counted earlier 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 experiments to generate the corresponding processing time.

In the future, 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. Besides, our work is open source on GitHub, along with the trace sample and some implementation details (<https://github.com/CynthiaProtector/SketchDLC>).

ACKNOWLEDGMENTS

We sincerely thank all the reviewers for their constructive feedback and detailed advice. We also genuinely appreciate Prof. Hongyi Lu for providing us with the GPU cluster and thank Erci Xu, Haowen Deng, and Yang Zhao for their useful suggestions.

REFERENCES

- [1] High-Performance Deep Learning (HiDL). 2018. RDMA-TensorFlow 0.9.1. Retrieved March 21, 2019 from <http://hidl.cse.ohio-state.edu>.
- [2] GitHub. 2018. Caffe-MPI 2.0. Retrieved March 21, 2019 from <https://github.com/Caffe-MPI/>.
- [3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv:1603.04467.
- [4] Francisco 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 *Proceedings of the 2015 IEEE International Conference on Cluster Computing (CLUSTER'15)*. IEEE, Los Alamitos, CA, 841–848.
- [5] 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, New York, NY, 193–205.
- [6] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, et al. 2012. Theano: New features and speed improvements. arXiv:1211.5590.
- [7] 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.
- [8] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGPLAN Notices* 49, 4 (2014), 269–284.
- [9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, et al. 2015. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv:1512.01274.
- [10] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao family: Energy-efficient hardware accelerators for machine learning. *Communications of the ACM* 59, 11 (2016), 105–112.
- [11] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, et al. 2014. DaDianNao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 609–622.
- [12] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, et al. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. *ACM SIGARCH Computer Architecture News* 44, 3, 27–39.
- [13] Trishul M. Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 571–582.
- [14] Microsoft Azure. 2018. The Microsoft Cognitive Toolkit. Retrieved March 21, 2019 from <https://www.cntk.ai/>.
- [15] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, et al. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*. 1223–1231.
- [16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*. IEEE, Los Alamitos, CA, 248–255.
- [17] NVIDIA. 2018. NVIDIA DGX-1. Retrieved March 21, 2019 from <https://www.nvidia.com/en-us/data-center/dgx-1/>.
- [18] NVIDIA. 2018. NVIDIA DGX-2. Retrieved March 21, 2019 from <https://www.nvidia.com/en-us/data-center/dgx-2/>.
- [19] Caffe2. 2018. Gloo. Retrieved March 21, 2019 from <https://caffe2.ai/docs/distributed-training.html>.
- [20] 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.
- [21] 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.
- [22] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv:1502.03167.
- [23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, et al. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*. ACM, New York, NY, 675–678.
- [24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, New York, NY, 1–12.
- [25] Edward Karrels and Ewing Lusk. 1994. Performance analysis of MPI programs. In *Environments and Tools for Parallel Scientific Computing*, J. J. Dongarra and B. Tourancheau (Eds.). Advances in Parallel Computing. North-Holland, 195–200.
- [26] Benjamin Klenk and Holger Fröning. 2017. An overview of MPI characteristics of exascale proxy applications. In *Proceedings of the International Supercomputing Conference*. 217–236.

- [27] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. Vol. 1. No. 4. Technical report, University of Toronto.
- [28] 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.
- [29] Yann LeCun. 1998. The MNIST database of handwritten digits. Retrieved March 21, 2019 from <http://yann.lecun.com/exdb/mnist/>.
- [30] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86, 11 (1998), 2278–2324.
- [31] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, et al. 2014. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 583–598.
- [32] Yujun Lin, Song Han, Huihui Mao, Yu Wang, and William J. Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. arXiv:1712.01887.
- [33] J. Miguel-Alonso, J. Navaridas, and F. J. Ridruejo. 2009. Interconnection network simulation using traces of MPI applications. *International Journal of Parallel Programming* 37, 2 (2009), 153–174.
- [34] HPC Wire. 2017. In-Network Computing and Next Generation HDR 200G InfiniBand. Retrieved March 21, 2019 from <https://www.hpcwire.com/2017/10/23/network-computing-next-generation-hdr-200g-infiniband/>.
- [35] Next Platform. 2018. Programmable Networks Train Neural Nets Faster. Retrieved March 21, 2019 from <https://www.nextplatform.com/2018/02/14/programmable-networks-train-neural-nets-faster/>.
- [36] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, Los Alamitos, CA, 688–698.
- [37] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. 1995. PARAVÉR: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and Occam Developments*, Vol. 44. IOS Press, Amsterdam, Netherlands, 17–31.
- [38] NVIDIA. 2018. Scalable AI Platform for Autonomous Driving. Retrieved March 21, 2019 from <https://www.nvidia.com/en-us/self-driving-cars/drive-platform/>.
- [39] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*. 693–701.
- [40] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556.
- [41] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, et al. 2015. Going deeper with convolutions. arXiv:1409.4842.
- [42] 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 *Proceedings of the 2011 International Conference on High Performance Computing and Simulation (HPCS'11)*. IEEE, Los Alamitos, CA, 819–824.
- [43] Dayong Wang, Aditya Khosla, Rishab Gargya, Humayun Irshad, and Andrew H. Beck. 2016. Deep learning for identifying metastatic breast cancer. arXiv:1606.05718.
- [44] Linnan Wang, Yi Yang, Renqiang Min, and Srimat Chakradhar. 2017. Accelerating deep neural network training with inconsistent stochastic gradient descent. *Neural Networks* 93 (2017), 219–229.
- [45] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, et al. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the 6th ACM Symposium on Cloud Computing*. ACM, New York, NY, 381–394.
- [46] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. TernGrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in Neural Information Processing Systems*. 1509–1519.
- [47] 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:1511.08486.
- [48] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. 2018. RPC considered harmful: Fast distributed deep learning on RDMA. arXiv:1805.08430.
- [49] Wikipedia. 2018. ZeroMQ. Available at <https://en.wikipedia.org/>.
- [50] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, et al. 2017. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 181–193.

Received May 2018; revised November 2018; accepted February 2019