Network Traffic Characteristics of Machine Learning Frameworks Under the Microscope

Johannes Zerwas*, Kaan Aykurt*, Stefan Schmid‡§, Andreas Blenk*†
*Chair of Communication Networks, Technical University of Munich, Germany
†Faculty of Computer Science, University of Vienna, Austria
‡TU Berlin, Germany
§Fraunhofer Institute for Secure Information Technology (SIT), Germany

***Abstract*—High computational demands of complex deep learning models led to workload distribution across multiple machines. Many frameworks for distributed machine learning (DML) have been developed and are employed in practice for orchestrating workload distribution.**

In this paper, we analyze and compare network behaviors of three widely used state-of-the-art DML frameworks. The study reveals that traffic can largely vary across the frameworks. While some frameworks exhibit well predictable patterns, others are less structured. We further explore whether and how it is possible to relate the network traffic to the DML jobs' attributes, and present a multiple linear regression model accordingly. Our results can inform the networking community about traffic characteristics and contribute toward the generation of realistic DML traffic for simulation studies.

***Index Terms*—distributed machine learning, network traffic measurement**

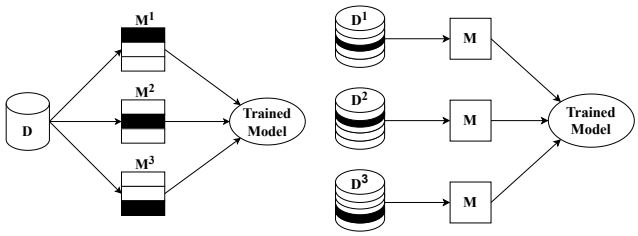
I. Introduction

Machine learning (ML) is becoming an integral part of today's applications from business over health to entertainment [1]–[3]. The increased demand for high performance models has led to a continuous growth of model complexities and dataset sizes [4], [5]. This growth has outpaced the technological advancements in the hardware field and, hence, workload is distributed across several workers. In particular, data parallelism that partitions and distributes the dataset across multiple machines in a cluster training the model, has received much attention both in academia and industry.

As the workload is distributed to several workers, computational restraints shift to communication bottlenecks [6]: Improvements in hardware accelerators for computing, such as GPUs or TPUs, increased the frequency of gradient updates which demands for higher performance from the network. It has been shown that neglecting this can result in reduced benefits of distributed training [6], [7]. Hence, aside from scaling up bare network capacity, e.g., from 10G or 40G to 100G, a variety of frameworks for distributed training of ML models (DML) has been proposed which take communication into account [8]–[13]. On one hand these frameworks aim to reduce the training time by reducing the communication time with computation methods, e.g., compressing gradients [14]. On the other hand, they also address the communication between workers, e.g., by training in synchronous or asynchronous manner [15] or by using different communication structures such as server/client [10], rings [8], trees or full-meshed patterns [9]. Recent proposals, such as `KungFu`, can even adapt their communication structure during run-time to account for individual performance of workers or to mitigate network issues [9].

Motivated by these challenges, the networking community has started making great efforts to support DML workloads. Proposed solutions range from tailored flow scheduling [16], [17], via in-network aggregation of gradients leveraging programmable data-planes [18]–[20], to specifically crafted high-bandwidth topologies [21]–[23]. While the variety of (dynamic) communication patterns suggests heterogenity of the resulting network traffic, the network-level improvements assume one particular pattern, e.g., parameter-server [23] or ring-reduce [21]. This might especially be problematic when targeting public cloud environments [24] with many tenants that might use different frameworks. More specifically, the traces and distributions used for evaluating networking algorithms or topology designs cover mainly web-based and related backend services but neglect the nature of DML network traffic, e.g., [25], [26]. Available traces from the DML regime cover only the DML job requirements but do not account for network traffic [27]. To the best of our knowledge, traffic traces of DML workloads are not available. Consequently, networks for DML are often evaluated using testbed implementations, e.g., [16], [18] – a costly undergoing. Simulative evaluations considering DML traffic rely on simple approaches and are limited to single variants of distributed training procedures [18], [23], [28]. Available theoretical models also lack in-depth involvement of batch size, model, and framework. An analysis of the network traffic, which grasps the heterogeneity of available frameworks and configurations, and the impact of network conditions, is missing.

To illustrate heterogeneity of DML traffic, this paper overviews and characterizes the communication behavior of three state-of-the-art industry standard DML frameworks. It analyzes traffic traces from running DML frameworks in more than 75 scenarios varying trained deep learning models, framework configurations, and packet loss. Further, it provides insights into the flow patterns and the throughput of the network. By relating networking resource consumption to ML training metrics such as accuracy, another cost aspect of DML training is presented. To this end, the study derives an integral



(a) Model Parallelism. (b) Data Parallelism.

Fig. 1. Two approaches to parallelization of distributed training: Model and Data Parallelism. Figure based on [15, Fig. 2].

model to estimate the data consumption of a given training job. Overall, we believe that our results provide interesting insights to be reused, e.g., in generative traffic models for simulations, and future network algorithm and topology design studies. As a contribution to the research community, we make our code and traces publicly available.[1]

The remainder is structured as follows: Sec. II gives a brief overview of DML and the available frameworks. Sec. III lists related traces or generation methodologies for DML traffic. Sec. IV describes the testbed and measurement procedure. Sec. V provides the flow-level analysis of the communication behavior and outlines potential influence factors of bandwidth consumption. Finally, Sec. VI introduces a regression model to uncover the drivers of network utilization. We conclude and discuss future work in Sec. VII.

II. Background

This section gives an introduction to DML in general, introduces available communication flavors and frameworks.

A. Distributed Training

There are two types of distributed training utilized by common frameworks:

- **Model Parallelism**: distributing parts of a single model to multiple nodes using the same training data (Fig. 1(a)).
- **Data Parallelism**: training multiple instances of the same model on different subsets of the training data (Fig. 1(b)).

Model parallelism is beneficial if the model does not fit into the memory of a single node. However, since most commonly used models still fit onto single nodes, it is less often applied in practice. Therefore, our analysis focuses on data parallelism.

Data parallel methods divide the training dataset into non-overlapping partitions (Fig. 1). All of the workers apply the same model on different subsets of the training data using the parallelized variant of the Stochastic Gradient Descent (SGD) algorithm. The exchanged gradients are then aggregated and sent back to each worker to update their local version of the model. This process can be done either in synchronous or asynchronous manner.

In the synchronous variant, each worker waits until all the other workers complete their calculations and only then exchanges parameters. The asynchronous variant enables each worker to work at their own pace, without being affected by the slow workers. This approach increases the fault tolerance, as the system will continue to function even if one of the nodes fail, at the cost of introducing stale gradients, in which the slow workers train and update an old version of the model.

B. Communication Topologies

There are several popular structures such as trees, rings and parameter server (PS) architectures, which are being used in practice for DML clusters [15]. The synchronous all-reduce paradigm [29] is used for mesh, tree and ring structures, whereas asynchronous training is generally implemented on PS architectures.

The implementation of all-reduce consists of each worker updating all other workers. In the ring structure, updates are only sent to the next worker and thus forming a ring-like structure. For tree structures, communication and exchange is directed via parent nodes. PS uses a more centralized approach: All the variables are stored in parameter server(s) and workers communicate with the parameter server(s) for pushing and pulling updated gradients. A *chief* is responsible for coordinating the process.

Various communication topologies have different needs and hence multiple communication libraries exist in the literature [30].

C. Communication Libraries

The two main communication primitives are point-to-point communication and collective communication. Point-to-point communication serves the purpose for transmitting a message between a pair of processes, whereas collective communication is used for transmission of messages among groups of processes. Frequently used libraries for the implementation of communication strategies include Google Remote Procedure Call (gRPC) [31], Message Passing Interface (MPI) [32], Facebook's Gloo [33] and NVIDIA Collective Communication Library (NCCL) [34].

gRPC relies on point-to-point communication, in the spirit of client/server communication where a client can directly call a method on a server. The library has no collective communication support.

MPI is the most often used, de facto standard for high-performance computing and supports collective communication [30]. It has many implementations ranging from widely used open source OpenMPI library to commercial ones.

Gloo is a collective communications library developed by Facebook specially for ML applications.

NCCL is developed to achieve high bandwidth over PCIe and NVLink between GPUs in a single node, or over Ethernet and InfiniBand for connections across machines. It is optimized for NVIDIA GPUs, implementing multi-node and multi-GPU communication standards supporting collective communication as well as point-to-point communication. It is often used for accelerating collective communication in DML [9].

Communication libraries and topologies are made available to frameworks, which in turn orchestrate the distributed training process.

[1] https://github.com/tum-lkn/dml-network-traffic-analysis

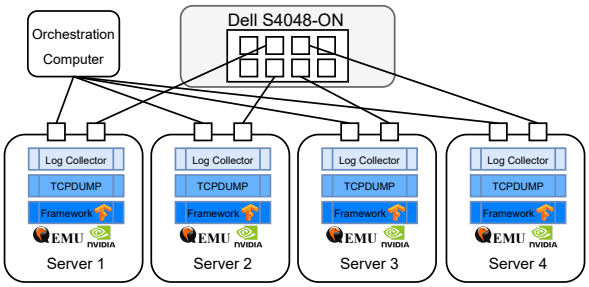


Fig. 2. Overview of the testbed.

D. DML Frameworks

The DML survey presented in [15] outlines the existing libraries and frameworks for distributing the training workload. Our evaluation focuses on three frameworks which provide *Data Parallelism*, and represent state-of-the-art in research and/or are adopted by industry: `TensorFlow` Distributed [10], `Horovod` [8] and `KungFu` [9].

TensorFlow has its own distributed training implementation that allows to distribute tasks across multiple GPUs, TPUs, or machines employing data parallel methods. It uses gRPC as communication library [30] and supports synchronous and asynchronous communication strategies through ring all-reduce algorithms on top of gRPC, NCCL's all-reduce algorithms, and parameter server strategies.

Horovod is an industry standard distributed deep learning training framework supporting `TensorFlow`, Keras [35], PyTorch [12], and Apache MXNet [36]. It distributes the training process with minimal code addition to the single node training scripts and supports only synchronous communication strategies through ring all-reduce algorithms. `Horovod` can use Gloo or MPI for collective communication. It also supports NCCL for tensor operations, however, primitive administrative operations such as gathering the number of workers and the ranks of the workers require MPI or Gloo.

KungFu is a framework for adaptively distributing the ML workload. It provides synchronous and asynchronous training, and includes an online monitoring component which adapts the communication architecture of the workers according to the network state during training. Its distributed synchronous optimizer (referred as Synchronous SGD optimizer) is equivalent to the one in `Horovod`. The asynchronous optimizer (referred to as Pair Averaging optimizer) is the implementation of Asynchronous Decentralized Parallel SGD [37]. `KungFu` includes its own collective communication API which can be accelerated via NCCL. Moreover, `KungFu` provides many topologies for the connectivity patterns of the workers, e.g., Tree, Star, Clique and Binary Tree Star (BTS).

III. Related Work

We are not aware of any thorough comparison of network traffic from different DML frameworks. However, there are several relevant works, which shed light on the topic from different perspectives. From a computing resources perspective, insights about requirements and analysis on GPU usage from production DML clusters have been provided in the literature [27]. But they neglect the communication aspect and do not focus on network analysis. Since several studies have shown that communication is a bottleneck to distributed training [6], many research papers focus on improving the communication efficiency such that increasing the number of GPUs scales-out linearly. Solutions include compressing the gradients [14], micro-managing communication patterns of nodes by coordinating distributed gradient computations [38], pruning model parameters [4] or scheduling communication [39]. They use additional (local) computation to reduce communication and focus on improving the total training time but they leave aside implications on the network.

In order to achieve linear scale-out, literature focuses on modifying the all-reduce architecture to hierarchical all-reduce [40], and developing network-based systems such as in-network aggregation to overcome communication bottlenecks [18], [19], flow schedulers [17] or tailored topologies [22], [23]. These works revolve around network traffic of DML but focus on point solutions for specific frameworks or communication patterns. For instance, [23] assumes only PS-based DML while [18] considers only ring-reduce DML traffic in the evaluation. Our study provides a comparison of traffic patterns as observed from publicly available frameworks.

In addition to proposing new designs, some works examine different approaches' throughput in terms of image per second trained and scalability [41], [42]. Other works aim to measure network performance of distributed training approaches as in [6], [30]. All these researches focus on identifying the communication bottleneck to the total training time but leave patterns on the network aside. There exists papers in the literature which simulate performance of DML on various topologies [18], [28]. However, these works include simplified assumptions about flow distributions and the used DML frameworks. That is, they consider given flow sizes and only homogeneous communication structures. Finally, there are papers outlining link utilization metrics of DML training [43]. However, these papers also lack either distinction between frameworks, model sizes, or do not specify temporal patterns.

IV. Measurement Setup

Our goal is to explore and compare the network behavior of frameworks and models on a small representative setup. This section describes the testbed and the measurement procedure for tracing the network traffic.

A. Testbed

Fig. 2 shows the testbed. It consists of four servers running Ubuntu 18.04 (5.15.0-47-generic kernel) with 128 GB of RAM and Intel Xeon Silver 4114 @ 2.2 GHz (20 cores). Each of them contains one NVIDIA Tesla T4 GPU and is connected via a 10 G Ethernet port to a Dell S4048-ON switch in L2 forwarding mode. The servers run kernel-based virtual machines (VM) with 48 GB of RAM and 8 pinned CPU cores. Each of the VMs is running the latest version of "generic/ubuntu1804" Vagrant Box. The GPUs and NICs are handed over to the VMs

using PCI pass-through capabilities. Communication libraries and frameworks are installed using the default instructions from the respective webpages or repositories. Specifically, the testbed uses `TensorFlow` 2.3.0, `Horovod` v0.22, `KungFu` 0.2. Each VM is configured to collect packets originating from itself via *tcpdump*. Moreover, application-level logs are collected with respect to each training step (training accuracy, loss, and step duration) for further analysis.

B. Settings

In order to analyze and understand the communication pattern structure, four models of different sizes are trained: MobileNetv2 (14 MB) [44], DenseNet201 (80 MB) [45], ResNet50 (97 MB) [46] and ResNet101 (171 MB) [46]. The models are selected from the most popular deep learning libraries and are also used as benchmarks in similar studies [9], [30]. We use the CIFAR10 [47] dataset to train the models. It consists of 60 000 32x32 color images with 50 000 training and 10 000 test samples. If not stated otherwise, we set the batch size per worker to 64. All networks are trained with random weight initialization (cold start). We use the Adam optimizer [48] and train for 20 epochs where an epoch considers all samples in the training dataset.

The compared training framework settings are:

- `TensorFlow` Distributed Framework: Ring-Reduce Synchronous SGD (S-SGD) on GPU;
- `TensorFlow` Distributed Framework: Parameter Server Training on CPU (PS);
- `Horovod` Framework: Ring Reduce and Hierarchical Reduce S-SGD on GPU;
- `KungFu` Framework: S-SGD on GPU;
- `KungFu` Framework: Asynchronous Decentralized Parallel SGD on GPU (PairAvg).

For further variation, `Horovod` uses NCCL in addition to MPI and Gloo, and `TensorFlow` is also run with NCCL support in addition to gRPC. For `KungFu`, BTS, Tree, Star and Clique connectivity patterns are run. We exclude additional optimization strategies described in Sec. III since they are not an inherent component of the frameworks. All the measurements include four nodes, except PS training, which includes six nodes. Each of the framework settings and models are measured once – more than 75 scenarios in total.

V. Evaluation

In this section, we analyze the packet traces collected during the distributed training process in order to compare the network traffic of different frameworks, models, and communication backends.

A. Total Data Transferred and Communication Patterns

Data Transferred. Fig. 3 shows the total transmitted volume per model and framework together with the model sizes. The volumes for 20 epochs range between $\approx$ 140 GB and $\approx$ 4 000 GB. The trained model is the main influencing factor on the total transmitted volume. `TensorFlow`, `Horovod` and `KungFu` S-SGD distribute the SGD via the same approaches.

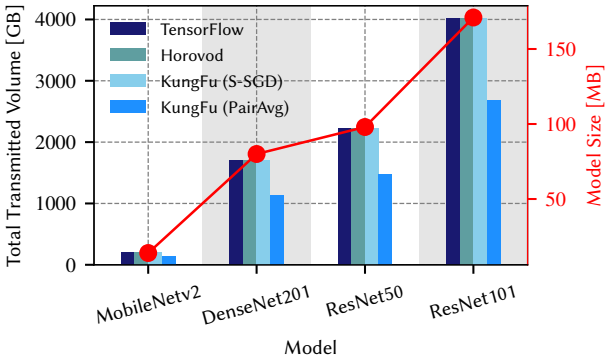
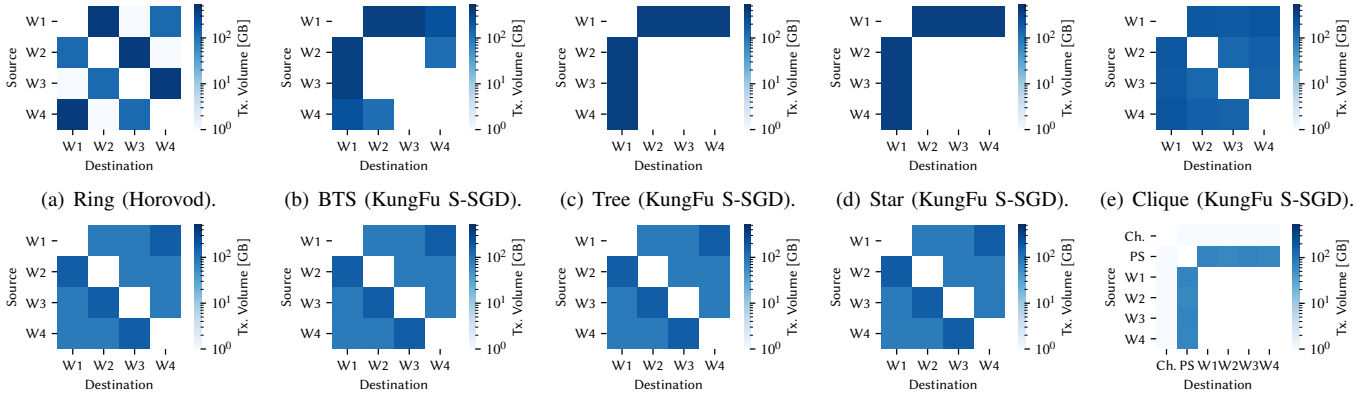


Fig. 3. Comparison of total traffic volume for 20 training epochs across four models and four framework configurations. Traffic volumes vary with the size of the model being trained (right axis).

Hence, they transmit the same total amount of data. Compared to these, `KungFu` PairAvg is more communication efficient; it transmits $\approx$ 30% less data for all of the models. Other influence factors are not observed. For instance, measurements that use RMSProp instead of Adam or NCCL instead of gRPC or MPI, do not differ in the transmitted volumes.

The right axis in Fig. 3 presents the model sizes. Model size is seen to be positively correlated with total amount of data transferred during a run. This is in line with similar assessments, e.g., as done in [18]. Larger models have more variables, accordingly the gradients exchanged in-between steps are expected to be larger and therefore, more data is transmitted.

Communication Patterns. Fig. 4 shows heatmaps of the total transmitted volume between nodes to illustrate the communication pattern of various architectures. For the ring-reduce pattern (Fig. 4(a)), the dark squares indicate the neighboring nodes in the ring. In this case, the structure of the ring is $W1 - W2 - W3 - W4 - W1$. In all frameworks, the user can specify the order of the ring. On average, the intense communication pairs in the ring accumulate 566 GB, whereas the opposite direction in the ring amounts to less than 1 GB. These opposite flows are constituted from acknowledgements. `KungFu` S-SGD with BTS (Fig. 4(b)) illustrates a pattern where W1 is the root of a tree and relays all the communication. W4 is a child of W2 and relays $1/3$ of the traffic through it. The reason for the direct communication between W1 and W4 could not be clarified. Tree and Star topologies use only W1 as relaying node (Fig. 4(c) & 4(d)). The Clique structure (Fig. 4(e)) shows the most dense pattern for S-SGD with data transmissions for all available communication pairs. `KungFu` PairAvg (Fig. 4(f)-4(i)) consistently employs a full-mesh pattern. A more intense structure similar to the ring pattern is evident. Finally, PS training pattern (Fig. 4(j)) shows that workers are communicating only with the parameter server. No cross-communication amongst the workers is observed. The chief distributes the tasks to all the workers and the parameter server. Ring, tree and PS structures exploit predictable patterns for generalization to larger scales whereas asynchronous training requires further analysis with larger



(a) Ring (Horovod). (b) BTS (KungFu S-SGD). (c) Tree (KungFu S-SGD). (d) Star (KungFu S-SGD). (e) Clique (KungFu S-SGD).

(f) BTS (KungFu PairAvg). (g) Tree (KungFu PairAvg). (h) Star (KungFu PairAvg). (i) Clique (KungFu PairAvg). (j) PS (TensorFlow).

Fig. 4. Communication Patterns: heatmap of total transmitted volume per framework configuration, averaged over all models. Connectivity patterns depend on the distributed optimizer and training strategy in use. "Ch." stands for Chief in Fig. 4(j).

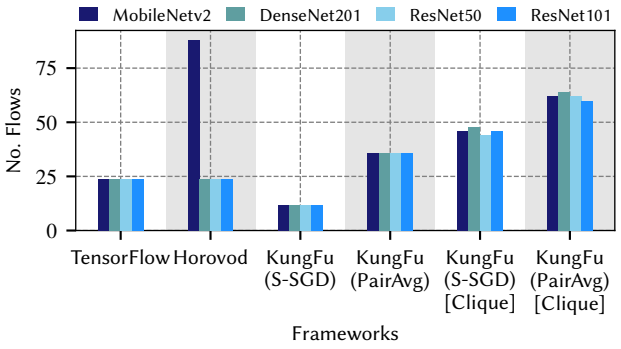


Fig. 5. Number of flows per framework and model. `TensorFlow` and `Horovod` represent ring topology. For `KungFu`, BTS, Tree and Star topologies have same number of flows whereas Clique topology has more flows.

settings.

B. Flow Analysis

Having analyzed the total traffic, we are now interested in how many network flows are used and how traffic is distributed among flows. We define a flow as the five tuple of source and destination IP addresses, transport protocol, and source and destination ports.

1) Flow Structure: Fig. 5 reports the total number of flows per framework and model. The number of flows in a framework is constant across models except for `Horovod` and `KungFu` with Clique topology. Moreover, the number of flows in a distributed training scenario depends mainly on the number of workers and the communication strategies. Repeated measurements with the same settings indicate that the number of flows do not change. The number of large flows for serving gradient exchanges varies. For synchronous training, there are four big flows in the ring topology, six big flows in BTS and Tree, and eight big flows for the case of the asynchronous PairAvg optimizer. These big flows make up to 99% of the total transmitted volume on average.

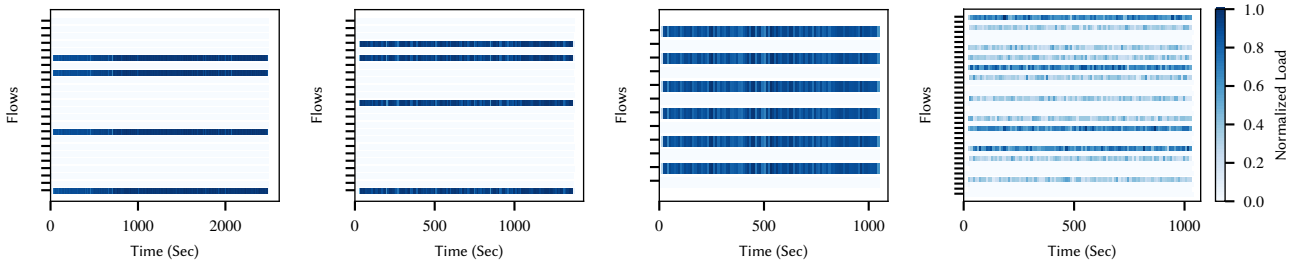
`TensorFlow` using ring all-reduce strategy has 24 flows. For 4 workers in ring structure, 4 main flows exchange gradients and other 4 serve for acknowledgements in the reverse direction. However, `TensorFlow` opens side connections between the non-neighboring nodes in the ring. The traces contain per run 16 of such flows which each amounts only to 1 MB traffic on average with 912 KB variance.

`Horovod` has 24 flows except for a single MobileNetv2 measurement with 88 flows. When the VM and `Horovod` is set up for the first time, `Horovod` runs initialization checks between the servers and this causes the number of flows to grow to 88. This behavior is repeated every hour due to caching mechanisms. Although the number of flows in `TensorFlow` and `Horovod` is the same, 8 flows are used by SSH connections in `Horovod`.

`KungFu` S-SGD, using BTS, Tree and Star communication topologies, has 12 flows in total, consisting of 2 flows in both directions for each edge of the tree. The Clique topology serves 46 flows on average. In addition to the 12 flows for serving gradient exchanges, it has 22 small flows consisting of two sizes with 54 Bytes and 74 Bytes. The remaining flows have an average size of 63 MB. `KungFu` PairAvg's pattern has 36 flows in total for BTS, Tree and Star communication patterns and 64 for Clique topology. W1 has 4 flows with the other workers in both directions making up 24 flows. The remaining 12 flows are between the rest of the workers in both directions. As W1 is the root node of the tree, it has double the number of connections. Similar to S-SGD case, small flows amounting to 54 Bytes and 74 Bytes make up the difference between flow numbers between Clique topology and other topologies. Overall, we conclude that the chosen communication strategy impacts the number of flows independently of the framework.

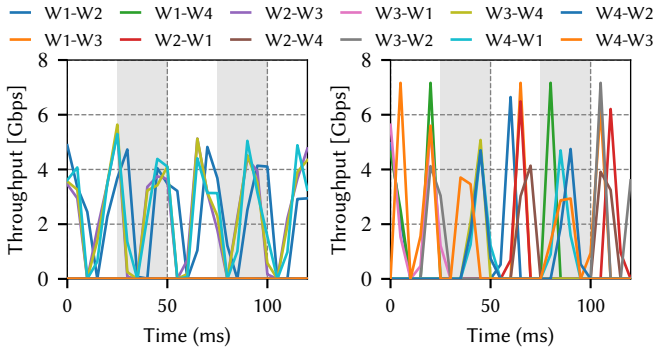
2) Temporal Behavior of Flows: Fig. 6 presents the temporal behavior of the flows.[2] For each flow, it illustrates the transmitted data per 10 s time window. The more solid the bar a window, the more data is transmitted. Values are normalized to the maximum throughput per scenario. As seen from the number of opaque bars, large flows only form a small fraction and serve uniformly throughout the whole training process. They exist mainly due to serving the gradient exchanges which

[2]We omit the PS scenario in the following since it was trained on CPU and hence the timings are not comparable.



(a) TensorFlow. (b) Horovod. (c) KungFu (S-SGD; BTS). (d) KungFu (PairAvg; BTS).

Fig. 6. Throughput per flow over time. Horizontal bars show individual flows. The opacity of the bar indicates the throughput per 10 s time window normalized to the maximum throughput in a scenario. Trained model is ResNet50. `KungFu` PairAvg distributes traffic over more flows than the other scenarios.



(a) Synchronous. (b) Asynchronous.

Fig. 7. Throughput per communication pair over time. Synchronous communication shows clearly periodic behavior (a). Less structure is evident in the asynchronous case (b). The ring structure of the synchronous case has only 4 flows forming the ring; the other flows are 0.

TABLE I
THROUGHPUT VALUES [GBPS]: TOTAL THROUGHPUT IN ALL OF THE LINKS, WITHOUT NCCL ACCELERATION. 95% CONFIDENCE INTERVALS ARE REPORTED IN SQUARE BRACKETS.

	TensorFlow	**Horovod**	**KungFu (S-SGD)**	**KungFu (PairAvg)**
MobileNetv2	4.91 [4.82, 5.00]	**8.19** [7.80, 8.58]	6.96 [6.14, 7.78]	4.55 [4.06, 5.04]
DenseNet201	6.42 [6.29, 6.55]	11.59 [11.39, 11.80]	**12.33** [12.00, 12.65]	10.09 [9.88, 10.29]
ResNet50	6.81 [6.71, 6.90]	12.80 [12.12, 13.48]	**13.17** [12.68, 13.67]	11.72 [11.46, 11.98]
ResNet101	6.33 [6.19, 6.47]	**13.68** [13.37, 14.00]	13.56 [13.18, 13.95]	12.08 [11.76, 12.41]

contribute towards learning process. The smaller flows are also present during the whole run. However, the transmitted volume is low. For the ring-reduce patterns employed by `Horovod` and `TensorFlow`, all the big flows start at the very beginning of the training. This is different for `KungFu` S-SGD optimizer. A closer look with one second precision reveals that two sets of flows start at the very beginning and the other two starts slightly after. This difference is caused by the BTS pattern. All the traffic is relayed via the root node of the tree. There is a slight delay between receiving and sending at the root and hence the arrival times of the flows are slightly different. In the asynchronous case implemented by `KungFu` PairAvg, flows arrive independently of each other. Besides, the data transmission shows less of a continuous pattern. This is not very much clear since all workers have the same specifications and do not deviate in terms of computation power.

The flow-level analysis reveals different patterns across the frameworks. Besides the number of flows, also the intra-flow behavior differs. To assess this more in detail, the data transmission rates of the communication pairs are evaluated in the following subsection.

C. Intra-flow Patterns

As a first analysis of the intra-flow behavior, Table I presents the average accumulated throughput values for the four 10 G links in the testbed. To obtain a larger number of samples, one run is split into multiple windows (batch mean) and in addition 10 runs per framework are collected. Values are reported from 30 samples in total. None of the frameworks makes efficient usage of the available bandwidth without NCCL acceleration. We note that `Horovod` and `KungFu` S-SGD consistently obtain higher throughput values, followed by `KungFu` PairAvg and `TensorFlow`. Within a framework, the trained model affects the observed throughput. Smaller models, e.g. MobileNetv2 and DenseNet201, result in lower average throughput values than the larger models such as ResNet101.

Larger models lead to larger gradients being transmitted at the end of a step; hence, larger models lead to more data transmission. However, contrary to the expectation, framework causes a bottleneck and bandwidth utilization is closely linked to framework and communication backend in use rather than the model size. `TensorFlow` bottlenecks the communication at total throughput around 4.9 Gbps for MobileNetv2 and 6.5 Gbps for the other models. The lower utilization of `TensorFlow` is caused by the lack of collective communication support of gRPC library. OpenMPI as used by `Horovod` outperforms gRPC and `KungFu`'s dedicated API further improves upon OpenMPI. Since `KungFu` PairAvg optimizer consumes less data overall in comparison to the other frameworks, it translates into lower bandwidth utilization.

In order to understand how the synchronous and asynchronous case behave on a smaller time scale, Fig. 7 zooms into 250 ms of the captured data. It compares the throughput for each communication pair across these cases for training MobileNetv2 model. Synchronous (Fig. 7(a)) uses `Horovod`

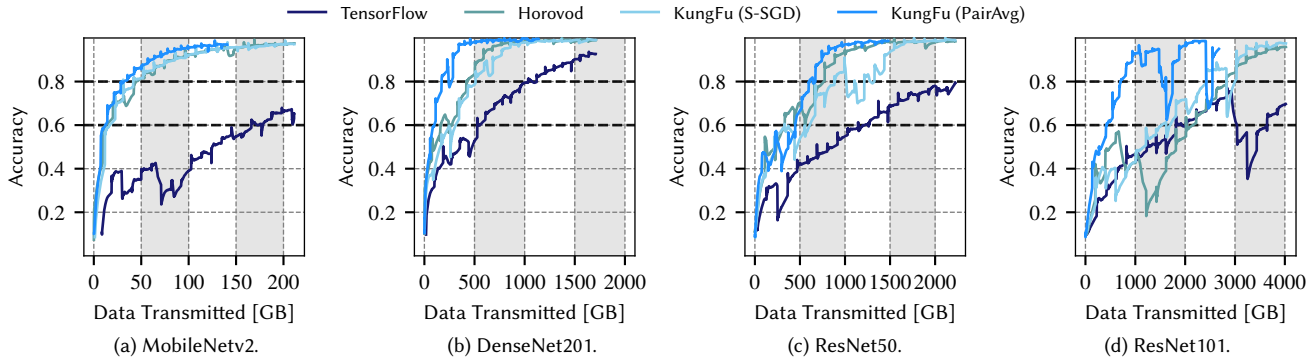


Fig. 8. Accuracy vs. transmitted volume. Dashed lines indicate 60% and 80% accuracy levels respectively. Each framework exhibits diminishing gains in accuracy with respect to transmitted data.

TABLE II
THROUGHPUT VALUES [GBPS]: THROUGHPUT OF A FLOW WHICH CARRIES GRADIENT EXCHANGE. SQUARE BRACKETS REPORT 95% CONFIDENCE INTERVALS.

	Average Throughput	Peak Throughput
`TensorFlow - gRPC`	1.65 [1.61, 1.68]	9.60
`TensorFlow - gRPC - NCCL`	**6.38** [6.25, 6.51]	9.78
`Horovod - MPI`	3.29 [3.20, 3.39]	8.07
`Horovod - MPI - NCCL`	**6.61** [6.55, 6.67]	9.74
`Horovod - Gloo`	4.70 [4.56, 4.83]	9.61
`Horovod - Gloo - NCCL`	**6.23** [6.11, 6.35]	9.76
KungFu (S-SGD)	2.76 [2.65, 2.87]	9.56
KungFu (PairAvg)	1.42 [1.28, 1.56]	9.62

with the ring-reduce pattern. It uses all connections forming the ring in a burst and stop fashion. The peaks correspond to gradient exchanges and the valleys correspond to computation intervals. Since the training is synchronized across the workers, they exchange gradients at the same time. The asynchronous implementation uses `KungFu` PairAvg (Fig. 7(b)). It shows peaks and bottoms at independent times without a particular structure as all the workers are pushing updates once the calculation is complete. Moreover, it can be seen that less than 50% of the bandwidth is utilized at peaks for `Horovod` whereas `KungFu` PairAvg utilizes over 60% bandwidth.

Table II compares eight combinations of frameworks and communication backends for a single communication pair which is used for gradient exchange while training a ResNet50 model. Throughput significantly depends on the communication backend. gRPC utilizes less than 20% of the available bandwidth. MPI and Gloo backends make better use of the bandwidth with collective communication support (NCCL). They obtain average throughputs of 3.29 Gbps and 4.70 Gbps respectively. `KungFu` seems to under-perform in comparison to other backends for a single flow. However, as it uses six or eight communication pairs depending on training strategy, it makes up for it on the total throughput. Finally, accelerating gRPC, MPI, or Gloo with NCCL most efficiently uses of the available bandwidth by providing fast collectives. Using NCCL utilizes around 65% of the whole bandwidth on average. For all cases, the peak throughput is close to the link capacity.

From an intra-flow perspective, there are clear phases of computation and communication for all frameworks independently of the distribution approach. All frameworks show the periodic nature of synchronous all-reduce. However, depending on the framework, there are specific nuances in flow timings. The asynchronous case exhibits less structure. A deeper analysis is left for future work.

D. The Cost of Accuracy

Fig. 8 shows the accuracy on training dataset vs accumulated data transmitted for the considered models and frameworks. Overall, we observe the expected increase in accuracy with transmitting more data and diminishing gains in accuracy with more data transmitted. However, depending on the used framework and model, there are differences. Starting with MobileNetV2 (Fig. 8(a)), `TensorFlow` achieves much lower accuracy of ≈ 0.65 compared to the other frameworks. As the synchronous distribution approach for `KungFu` and `Horovod` is implemented in a similar fashion which wraps `TensorFlow`'s optimizer, it can be seen that 80% accuracy can be reached with < 50 GB data for MobileNetV2 by both frameworks with the specified setting; for DenseNet201 500 GB (Fig. 8(b)), for ResNet50 around 750 GB (Fig. 8(c)), and for ResNet101 2.6 TB are required (Fig. 8(d)). `KungFu` PairAvg outperforms all the other frameworks as it reaches same accuracy with around 30% less data consumption.

The used frameworks and desired training performance need to be considered when estimating the cost of training in the network. A direct relation between the amount of transmitted data and the achieved accuracy is hard to infer. The main relation is introduced by the number of training steps as also elaborated by other works. Hence, we limit the modeling to the per step transmitted volume in the following.

E. Impact of Batch Size

Batch size refers to the number of data samples used by each worker in a single training step. The DML frameworks exchange data between the workers at the end of each step. Therefore, batch size and communication/computation ratio are closely linked to one another. Fig. 9 shows the throughput

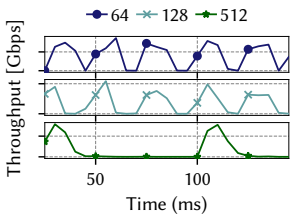


Fig. 9. Throughput of one communication pair over time for three batch sizes. Batch size affects the periodicity of communication. Trained model is MobileNetv2.

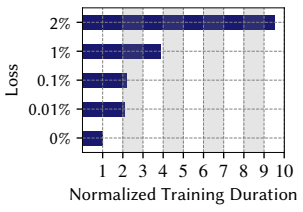


Fig. 10. Training duration normalized with respect to the measurement without loss. Introducing loss impacts the training time drastically.

on a single link for batch sizes 64, 128 and 512. Bigger batch sizes, increase computation to communication ratio: the duration of the valleys with no communication increases. Although the batch size effects the communication ratio, transferred volume per step remains the same and the communication bursts utilize similar bandwidth, on average, to exchange the gradients. This behavior is consistent across all frameworks and models (figures are omitted for brevity).

F. Impact of Packet Loss

In order to simulate a complex topology with network contention, we introduce additional packet loss to the measurements. Fig. 10 shows the total training duration for MobileNetv2 trained with `TensorFlow` normalized by the case with 0% loss. Increased packet loss results in drastically larger training times. With 0.01% loss, the time is already doubled; 2% loss lead to training duration 8 times larger than for 0% loss. An explanation is given by the performance decrease of TCP in lossy networks. Introducing loss reduces the exponential growth of the congestion window, which results in lower throughput and, hence, increased training time.

VI. Prediction Model

Given an ML model and training parameters, our measurements show that data transmitted per step is constant throughout the training process (cf. Fig. 6). Since gradient exchange is performed at the end of every step and it only depends on the model, the size of the exchange remains constant. In addition, the previous analysis shows that data transmitted for the whole training process depends mainly on the framework and communication topology in use and the model being trained. In order to find out the main drivers of bandwidth consumption, we build a model to predict transmitted volume in each step ($Data_{Step}$) given the framework, ML model, and training parameters. We regress all the available parameters on $Data_{Step}$ and report the significance of each parameter using Ordinary Least Squares Multiple Linear Regression. The model can be summarized as

$$\begin{aligned} Data_{Step} =& \beta_0 + \beta_1 \cdot Model\_Size \\ &+ \beta_2 \cdot Framework\_Backend \\ &+ \beta_3 \cdot Batch\_Size + \beta_4 \cdot Comm\_Topology. \end{aligned} \tag{1}$$

$Model\_Size$ is a continuous variable while rest of the variables are binary indicators. Framework and communication backend are stacked into a single variable ($Framework\_Backend$) in order to eliminate correlation between regressors and to get an unbiased estimate.

The regression yields an R^2 of 0.98. This indicates a very strong correlation between the regressor and the variables. Moreover, the regression results show that 95% confidence intervals for $Batch\_Size$ and $Comm\_Topology$ include 0 and therefore, they are statistically insignificant at $\alpha = 0.05$.

$Framework\_Backend$ is only significant for `KungFu` PairAvg, whereas it is insignificant for the other configurations. The coefficient is negative. This confirms that `KungFu` PairAvg consumes significantly less data. Moreover, the coefficient of the $Model\_Size$ is also significant: an increase of $1\,\mathrm{MB}$ in model size corresponds to $5.7\,\mathrm{MB}$ increase in data transmitted per step on average with 95% confidence.

Finally, correlating model size and data transmitted per step yields a correlation value of 0.97. This indicates that model size explains most of the variance in transmitted volume per step.

VII. Conclusion

This paper investigated the network traffic of three state-of-the-art DML frameworks with varying training parameters. We report key network metrics such as throughput, flow, and intra-flow patterns and relate them to training accuracy. The findings indicate that the traffic in terms of volume and communication patterns varies depending on the framework and the specific configuration. Our model for predicting the transmitted volume per step indicates that the model size is the main determinant of transmitted volume.

We believe that our measurement campaign opens interesting future research directions and may even help improve traffic engineering for machine learning applications. The establishment of the network behavior further provides guidance for realistic traffic modeling and generation for simulation studies and users to tune their data center architecture according to the communication needs of the training system.

Acknowledgments

This work received funding by the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project ”5G Testbed Bayern mit Schwerpunktanwendung eHealth”, and was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 438892507.

References

[1] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, “Openml: Networked science in machine learning,” *ACM SIGKDD Explor. Newsl.*, vol. 15, no. 2, p. 49–60, Jun. 2014.

[2] R. S. Olson, W. La Cava, P. Orzechowski, R. J. Urbanowicz, and J. H. Moore, “Pmlb: a large benchmark suite for machine learning evaluation and comparison,” *BioData mining*, vol. 10, no. 1, pp. 1–13, 2017.

[3] L. Columbus. What’s New In Gartner’s Hype Cycle For AI, 2020. Forbes. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2020/10/04/whats-new-in-gartners-hype-cycle-for-ai-2020/

[4] S. Lee, H. Kim, J. Park, J. Jang, C. Jeong, and S. Yoon, "Tensorlightning: A traffic-efficient distributed deep learning on commodity spark clusters," *IEEE Access*, vol. 6, pp. 27 671–27 680, 2018.

[5] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in NIPS*, vol. 32, pp. 103–112, 2019.

[6] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin, "Is network the bottleneck of distributed training?" in *Proc. ACM NetAI*, 2020, p. 8–13.

[7] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *Proc. USENIX ATC '17*, 2017, pp. 181–193.

[8] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[9] L. Mai, G. Li, M. Wagenländer, K. Fertakis, A.-O. Brabete, and P. Pietzuch, "Kungfu: Making training in distributed machine learning adaptive," in *Proc. USENIX OSDI 20*, 2020, pp. 937–954.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[11] "Apache singa." [Online]. Available: https://singa.apache.org/

[12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in NIPS 32*, 2019, pp. 8024–8035.

[13] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," 2020.

[14] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[15] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–33, 2020.

[16] S. Wang, D. Li, and J. Geng, "Geryon: Accelerating distributed cnn training by network-level flow scheduling," in *Proc. IEEE INFOCOM 2020*, 2020, pp. 1678–1687.

[17] W. Li, S. Chen, K. Li, H. Qi, R. Xu, and S. Zhang, "Efficient online scheduling for coflow-aware machine learning clusters," *IEEE TCC*, pp. 1–1, 2020.

[18] N. Gebara, M. Ghobadi, and C. Paolo, "In-network aggregation for shared machine learning clusters," *Proc. MLSys 2021)*, 2021.

[19] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "ATP: In-network aggregation for multi-tenant learning," in *Proc. USENIX NSDI*, 2021, pp. 741–761.

[20] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," *arXiv preprint arXiv:1903.06701*, 2019.

[21] M. Khani, M. Ghobadi, M. Alizadeh, Z. Zhu, M. Glick, K. Bergman, A. Vahdat, B. Klenk, and E. Ebrahimi, "Terarack: A tbps rack for machine learning training," 2020.

[22] Y. Lu, H. Gu, X. Yu, and P. Li, "X-nest: A scalable, flexible, and high-performance network architecture for distributed machine learning," *IEEE Journal of Lightwave Technology*, pp. 1–1, 2021.

[23] L. Liu, Q. Jin, D. Wang, H. Yu, G. Sun, and S. Luo, "Psnet: Reconfigurable network topology design for accelerating parameter server architecture based distributed machine learning," *Future Generation Computer Systems*, vol. 106, pp. 320–332, 2020.

[24] T. N. B. Duong and N. Q. Sang, "Distributed machine learning on iaas clouds," in *Proc. 5th IEEE Intl. Conf. on Cloud Computing and Intelligence Systems (CCIS)*, 2018, pp. 58–62.

[25] H. Ballani, P. Costa, R. Behrendt, D. Cletheroe, I. Haller, K. Jozwik, F. Karinou, S. Lange, K. Shi, B. Thomsen, and H. Williams, "Sirius: A flat datacenter network with nanosecond optical switching," in *Proc. ACM SIGCOMM*, 2020, p. 782–797.

[26] W. M. Mellette, R. Das, Y. Guo, R. McGuinness, A. C. Snoeren, and G. Porter, "Expanding across time to deliver bandwidth efficiency and low latency," in *Proc. USENIX NSDI*, 2020, pp. 1–18.

[27] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *Proc. USENIX ATC*, 2019, pp. 947–960.

[28] S. Wang, D. Li, J. Geng, Y. Gu, and Y. Cheng, "Impact of network topology on the performance of dml: Theoretical analysis and practical factors," in *Proc. IEEE INFOCOM*, 2019, pp. 1729–1737.

[29] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford, "A reliable effective terascale linear learning system," *Journal of Machine Learning Research*, vol. 15, no. 32, pp. 1111–1133, 2014.

[30] A. A. Awan, J. Bédorf, C. Chu, H. Subramoni, and D. K. Panda, "Scalable distributed dnn training using tensorflow and cuda-aware mpi: Characterization, designs, and performance evaluation," in *Proc. IEEE/ACM CCGRID*, 2019, pp. 498–507.

[31] "Google's remote procedure call library." [Online]. Available: http://www.grpc.io/

[32] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104.

[33] "Gloo." [Online]. Available: https://github.com/facebookincubator/gloo

[34] "Nvidia - nccl." [Online]. Available: https://github.com/NVIDIA/nccl

[35] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[36] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[37] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," in *Proc. ICML*, 2018, pp. 3043–3052.

[38] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro, "Deep learning with cots hpc systems," in *Proc. ICML*, 2013, p. III–1337–III–1345.

[39] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," *arXiv preprint arXiv:1803.03288*, 2018.

[40] M. Cho, U. Finkler, M. Serrano, D. Kung, and H. Hunter, "Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," *IBM Journal of Research and Development*, pp. 1–1, October 2019.

[41] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters," in *Proc. USENIX OSDI*, 2020, pp. 463–479.

[42] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019.

[43] S. Li, Y. Qin, Z. Jiang, and W. Yang, "Efficient communication scheduling for parameter synchronization of dml in data center networks," *IEEE TNSE*, pp. 1–1, 2021.

[44] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[45] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE CVPR*, 2017, pp. 4700–4708.

[46] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE CVPR*, 2016, pp. 770–778.

[47] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[48] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICML*, 2014.