Poseidon: An Efficient Communication Interface for Distributed Deep Learning on GPU Clusters

[1,2]Hao Zhang, [3]Zeyu Zheng, [1,2]Wei Dai, [2]Qirong Ho and [1,2]Eric P. Xing

[1]Carnegie Mellon University, [2]Petuum Inc., [3]Peking University

Abstract

Deep learning models can take weeks to train on a single GPU-equipped machine, necessitating scaling out DL training to a GPU-cluster. However, current distributed DL implementations can scale poorly due to substantial parameter synchronization over the network, because the high throughput of GPUs allows more data batches to be processed per unit time than CPUs, leading to more frequent network synchronization. We present Poseidon, an efficient communication interface for distributed DL on GPUs. Poseidon exploits the layered model structures in DL programs to overlap communication and computation, reducing bursty network communication. Moreover, Poseidon uses a hybrid communication scheme that optimizes the number of bytes required to synchronize each layer, according to layer properties and the number of machines. We show that Poseidon is applicable to different DL frameworks, by implementing Poseidon into Caffe and TensorFlow, and showing that, with Poseidon, both Caffe and TensorFlow can achieve 15.5x speed-up on VGG19-22K network for image classification using 16 single-GPU machines, even under limited bandwidth (10GbE). In particular, Poseidon-enabled TensorFlow achieves 31.5x speed-up with 32 single-GPU machines on Inception-V3, a 50% improvement over the original TensorFlow (20x speed-up). The software is available at `http://poseidon-release.readthedocs.io`.

1 Introduction

Deep learning (DL) is a class of machine learning (ML) approaches with deep architectures that has achieved notable success across a wide spectrum of tasks, including speech recognition [9], visual recognition [31] and language understanding [18]. These DL models exhibit a high degree of model complexity, with many parameters in deeply layered structures that usually take days to train on a GPU-equipped machine. The high computational cost of DL programs on large-scale data necessities its multi-node execution on distributed GPUs in order to keep the training time acceptable.

DL software such as TensorFlow [1] and Caffe [12] allow practitioners to easily experiment with DL models on one or more GPUs per machine. However, their distributed implementation can sometimes scale poorly when using multiple machines to train larger deep networks; for example, on the VGG19-22K network (229M parameters) TensorFlow on 32 machines can be slower than a single machine (Section 5.1). Parallel DL on clusters involves substantial parameter synchronization across the network, where the high computational throughput of GPUs allows more data batches to be processed per minute (than CPUs), leading to more frequent network synchronization that only grows with additional distributed machines. Existing communication strategies, such as parameter servers for ML, can become overwhelmed by the high volume of parameters. Moreover, despite the increasing availability of faster network interfaces such as Infiniband or 40GbE Ethernet, GPUs have continued to grow rapidly in computational power, and continue to produce parameter updates faster than can be naively synchronized over the network – for instance, on a 16-machine cluster with 40GbE Ethernet and one GeForce Titan X GPU per machine, updates from the VGG19-22K model will bottleneck the network, so that only an 8x speedup over a single machine is achieved (i.e. 50% of ideal linear scalability, Section 5.1).

These scalability limitations in distributed DL stem from at least two causes: (1) the gradient updates to be communicated are very large matrices, which quickly saturate network bandwidth; (2) the iterative nature of DL algorithms causes the updates to be transmitted in bursts (at the end of an iteration or batch of data), with significant periods of low network usage in between. We propose that a solution to these two problems should exploit the structure of DL algorithms, on two levels: on the one hand, it should identify ways in which the matrix updates can be separated from each other, and then schedule them in a way that avoids bursty network traffic. On the other hand, the solution should also exploit the structure of the matrix updates themselves, and wherever possible, reduce their size and thus the overall load on the network. For such a solution to be relevant to practitioners (who may have strong preferences for particular frameworks), we would prefer not to exploit specific traits of (for example) TensorFlow's or Caffe's design, but should strive to be relevant to as many existing frameworks as possible.

With this motivation, we design Poseidon, an efficient communication interface for data-parallel DL on distributed GPUs. Poseidon exploits the sequential layer-by-layer structure in DL programs, finding independent GPU computation operations and network communication operations in the training algorithm, so that they can be scheduled together to reduce bursty network communication. Moreover, Poseidon implements a hybrid communication scheme that accounts for each DL program layer's mathematical properties as well as the cluster configuration, in order to compute the network cost of different communication methods, and select the cheapest one – currently, Poseidon implements and supports a parameter server scheme [28] that is well-suited to small matrices, and a sufficient factor broadcasting scheme [29] that performs well on large matrices. We focus on synchronous parallel training, because recent research reveals that synchronous execution yields the fastest per-iteration improvement in accuracy for distributed DL (as measured by wall clock time) on GPUs [7, 2], even if other consistency models (e.g. SSP) can alleviate the straggler problem when exists. Unless otherwise specified, our discussion in this paper assumes synchronous replication of model parameters in each training iteration. That said, we note that Poseidon's design is amendable to asynchronous or bounded-asynchronous consistency models, such as ASP and SSP, which may have future potential for DL.

To demonstrate Poseidon's applicability to multiple DL frameworks, we implement it into two different DL frameworks: Caffe and TensorFlow, and show that Poseidon allows them to scale almost-linearly in algorithm throughput with additional machines, while incurring little additional overhead even in the single machine setting. For distributed execution, with 40GbE network bandwidth available, Poseidon consistently delivers near-linear increases in throughput across various models and engines: e.g. 31.5x speedup on training the Inception-V3 network using TensorFlow engine on 32 nodes, which improves 50% upon the original TensorFlow (20x); When training a 229M parameter network (VGG19-22K), Poseidon still achieves near-linear speedup (30x on 32 nodes) using both Caffe and TensorFlow engines, while distributed TensorFlow sometimes experiences negative scaling with additional machines. Our experiments also confirm that Poseidon successfully alleviates network communication bottlenecks, by reducing the required bandwidth for parallelizing large models. For example, when training VGG19-22K under limited bandwidth (10GbE), in contrast to a parameter server based paralleization which only achieves 4x speedup with 16 machines, Poseidon effectively reduces the communication overheads by automatically specializing the best communication method for each layer, and is able to keep linear scaling on throughput.

Compared to other communication reduction methods [4, 32], Poseidon demonstrates either systems advantages (increased algorithm throughput) or statistical advantages (fewer algorithm steps or iterations to reach a fixed termination criteria). Poseidon does not suffer much from imbalanced communication loads, which we found to be the case when using the sufficient factor strategy used in Project Adam [4]. Poseidon also guarantees that the number of algorithm steps to reach termination remains unchanged, unlike the 1-bit quantization strategy used in CNTK [32] which is approximate and can hurt statistical performance in some applications.

2 Large-scale Deep Learning

In this section, we first introduce deep learning by taking a convolutional neural network (CNN) as an example. We then formulate the DL training as an iterative-convergent algorithm, and describe two strategies, parameter server (PS) and sufficient factor broadcasting (SFB), for parallelizing such computation on clusters.

2.1 Distributed Deep Learning

DL programs are distinguished from other ML programs mainly by their use of neural networks (NNs), a family of hierarchical models containing many layers, from as few as 5-10 [14] to as many as 100s [10]. Figure 1 illustrates a neural network with 5 layers. The first layer (green) is an input layer that reads data in application-specific formats, e.g. raw pixels if the neural network is trained to classify images. The input layer is connected to a sequence of intermediate layers (yellow, cyan). Every intermediate layer consists of a few neurons, where each neuron applies a function transformation f on its input, and produces an output. A vector output is obtained by concatenating the output of all neurons from a layer. By stacking multiple intermediate layers, the NN can transform raw input data one layer at a time, first into a series of intermediate representations, and finally into the desired output or prediction (red). DL programmers usually need to specify the computation of a layer by defining two properties of its neurons. The first is the transformation function $f(W,x)$, where x is the input to the neuron, and W is an *optional trainable* parame-

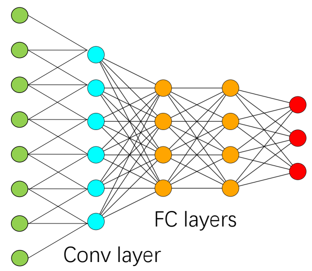

Figure 1: A convolutional neural network with 5 layers.

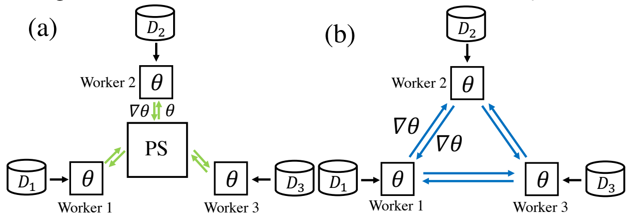

Figure 2: An illustration of the (a) parameter server and (b) sufficient factor broadcasting for distributed ML.

ter. The other is the connectivity that determines how the neuron should be connected to its adjacent layer. For instance, a convolutional neural network has two types of neuron: convolutional (CONV) neuron (cyan) that are only locally connected to a subset of neurons in its previous layer, and fully-connected (FC) neurons (yellow).

Most neural networks need to be trained with data before giving accurate predictions. Stochastic gradient descent (SGD) and backpropagation are commonly employed to train NNs iteratively – each iteration performs a feed forward (FF) pass followed with a backpropagation (BP) pass. In the FF pass, the network takes a training sample as input, forwards from its input layer to output layer to produce a prediction. A loss function is defined to evaluate the prediction error, which is then backpropagated through the network reversely, during which the network parameters are updated by their gradients towards where the prediction error would decrease. After repeating a sufficient number of passes, the network will usually converge to some state where the loss is close to a minima, and the training is then terminated.

In a mathematical form, given data D and a loss function $\mathcal{L}$, fitting the parameters θ of a NN can be modeled as an *iterative-convergent* algorithm that repeatedly executing the update equation

$$\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)}) \tag{1}$$

until θ reaches some stopping criteria, where t denotes the iteration. The update function $\nabla_{\mathcal{L}}$ calculates the gradients of $\mathcal{L}$ over current data $D_i (D_i \in D)$. The gradients are then scaled by a learning rate ε and applied on θ as updates. As the gradients are additive over data samples i, i.e. $\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \sum_i \nabla_{\mathcal{L}}(\theta^{(t-1)}, D_i)$, for efficiency, we usually feed a batch of training samples $D^{(t)} (D^{(t)} \subset D)$ at each training iteration t, as in Eq.1.

In large-scale deep learning, data D are usually too large to process on a single machine in acceptable time. To speedup the training, we usually resort to *data parallelism*, a parallelization strategy that partitions the data D and distributes to a cluster of computational worker machines (indexed by $p = 1, \cdots, P$), as illustrated in Figure 2. At each iteration t, every worker fetches a batch $D_p^{(t)}$ from its data partition and computes the gradients $\nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$. Gradients from all workers are then aggregated and applied to update $\theta^{(t)}$ to $\theta^{(t+1)}$ following

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)}) \tag{2}$$

Data-parallelism allows data to be locally partitioned to each worker, which is advantageous for large datasets, it however requires every worker to have read and write access to the shared model parameters θ, which causes communication among workers; this shared access can be provided by a parameter server architecture [28, 4] (Figure 2a) or a peer-to-peer broadcasting architecture [29] (Figure 2b), both are designed for general-purpose data-parallel ML programs on CPUs.

Parameter Server. A parameter server (PS) is a distributed shared memory system that provides systematic abstraction of iterative-convergent algorithms in data-parallel distributed ML. Typically, PS enables each worker to access the global model parameters θ via network communications following the client-server scheme. It is natural to parallelize DL over distributed workers following the PS scheme, by letting the execution of the update function $\nabla_{\mathcal{L}}$ take place only on each worker (denoted as worker node) over data subsets therein, and the application of the updates to model parameters θ take place on the server (denoted as server nodes), and a consistency scheme coordinate the synchronization among server and workers (Figure 2a).

Sufficient Factor Broadcasting. Most ML models fall into the family of *matrix-parameterized models* (MPMs), which represent their parameters θ as a set of matrices. For some MPMs, including NNs, when being trained using SGD, their gradient $\nabla\theta$ over a training sample is a rank-1 matrix, which can be casted as the outer product of two vectors u and v: $\nabla\theta = uv^\top$, where u and v are called *sufficient factors* (SFs). Sufficient factor broadcasting (SFB) [29] is designed to parallelize these MPMs, by broadcasting SFs u, v among workers, and then reconstructing the gradient matrices $\nabla\theta$ using u, v locally. SFB presents three key differences from PS: (1) SFB employs a peer-to-peer communication strategy that transmits SFs instead of full matrices, which sometimes yields lower communication load; (2) unlike gradients, SFs are not additive over training samples, i.e. the number of SFs needed to be transmitted grows linearly with the number of data samples (not data batches); (3) the overall communication overheads of SFB increase quadratically with the number of workers.

2.2 Parallel DL on Distributed GPUs

Modern DL models are mostly trained using NVIDIA GPUs, because the nature of computations GPUs were designed to accelerate happens to be the same as those encountered in DL — when feeding a data batch into a NN, the computation happening in both FC and CONV layers can be expressed as matrix-matrix multiplications or convolutions, which exactly match the SIMD operation that could be efficiently performed by GPUs.

In practice, DL practitioners often use single-node software frameworks, such as Caffe [12] and Torch [6], which mathematically derive the correct training algorithm and execute it on GPU by calling GPU-based acceleration libraries, such as CUDA and cuDNN. It is thus straightforward to parallelize these programs across distributed GPUs using either PS or SFB, by moving the computation from CPU to GPU, and performing memory copy operations (between RAM and GPUs) or communication (among multiple nodes) whenever needed. However, we argue below, and show empirically in Section 5 that these usually lead to suboptimal performance.

The inefficiency is mainly caused by parameter synchronization via the network. Compared to CPUs, GPUs are order-of-magnitude more efficient in matrix computations; the production of gradients on GPUs are much faster than they can be naively synchronized over the network. As a result, the training computation are usually bottleneck by communications. For example, when training AlexNet [14] (61.5M parameters) on Titan X with a standard batch size 256, 240 million of gradients will be generated per second on each GPU (0.25s/batch). If we parallel the training on 8 nodes using a PS, with every node also holding $1/8$ of parameters as a PS shard; then, every node needs to transfer $240\text{M}\times 7/8 \times 4 = 840\text{M}$ float parameters in one second to make sure the next iteration of computation not being blocked. Apparently, the demanded throughput ($>$26Gbps) exceeds the bandwidth that commodity Ethernet (i.e. 1GbE and 10GbE Ethernet) provides; the GPUs distributed across clusters cannot be fully utilized. Practically, it is usually difficult to partition the parameters completely equally, which will result in more severe bandwidth demands, or bursty communication traffic on several server nodes (as we will show in Section 5.3), which prevents the trivial realization of efficient DL on distributed GPUs [1]. We next describe our strategies and system design to overcome the aforementioned obstacles.

[1]Another overhead of distributed DL on GPUs is caused by the frequent memory copy operations between RAM and GPU memory, which is minor compared to network communication according to our empirical results. However, our strategies in this paper can also minimize this overhead.

3 Poseidon Design

In this section, we first analyze the DL program in both a single-node and distributed environment by decomposing the program into a sequence of operations. Based on it, we introduce two strategies to address the issues.

3.1 The Structure of DL Programs

At the core of the DL program is the backpropagation algorithm that performs forward-backward pass through the network repeatedly. If we define a full pass through the network at iteration t as a **C**ompute step C_t, the DL computation on a single node is thus a sequence of C steps $[C_1, \cdots, C_T]$, with T as the total number of iterations. When executing on distributed GPUs, inter-machine communication is required after each C step to guarantee the synchronized replication of model parameters. We similarly define the **S**ynchronization step S_t as the process that a worker sends out locally generated parameter updates and then receives updated parameters from remote workers at iteration t. Therefore, a naive parallelization of DL training over distributed GPUs, using either PS or SVB, can be expressed as alternating the two steps defined above – we note that DL training is highly sequential; the communication and computation perform sequentially, waiting each other to finish. Only by C_t finishes can S_t start, and C_{t+1} is blocked until updates from other workers are received (S_t finishes).

Decompose C_t and S_t. Fortunately, the sequential nature of the backpropagation algorithm enables us to break down C_t and S_t into a series of smaller operations. Denote a **f**orward pass through the lth layer as f_t^l and a **b**ackward pass through it as b_t^l, C_t thus can be equivalently represented as a sequence of forward and backward operations as $C_t = [f_t^1, \cdots, f_t^L, b_t^L, \cdots, b_t^1]$ with L as the number of layers. Similarly, because every layer of a NN contains an independent set of parameters, a synchronization step S_t can be represented as $S_t = (s_t^1, \cdots, s_t^L)$, by defining each operation s_t^l as the synchronization of parameters in layer l. If we further decompose $s_t^l = [o_t^l, i_t^l]$ as first sending out local parameter updates of layer l (o_t^l) and reads in the updated parameters for layer l remotely (i_t^l). We finally rewrite a training iteration in a finer resolution:

$$[C_t, S_t] = [f_t^1, \cdots, f_t^L, b_t^L, \cdots, b_t^1, s_t^L, \cdots, s_t^1] \qquad (3)$$

Dependencies of Operations. A forward operation f_t^l cannot happen earlier than its previous ones $f_t^i (i < l)$, because it needs the output of f_t^{l-1} as input. Similarly, a backward operation b_t^l depends on $b_t^i (i > l)$. A synchronization operation s_t^l depends on b_t^l as it cannot start until the parameter updates in layer l is generated. Moreover, to respect the BSP constraint, f_t^l cannot happen earlier than s_{t-1}^l, as s_{t-1}^l will modify parameters in layer l.

Taking into consideration these dependencies, we try to exploit the independecies remained among these operations. Our first strategy, *wait-free backpropagation*, overlaps C_t and S_t by partially rescheduling those b_t and s_t that are independent. Our second strategy, *hybrid communication*, utilizes the independency among s_t, and tries to reduce the communication overheads by specializing different communication methods for different s_t.

3.2 Wait-free Backpropagation

The wait-free backpropagation (WFBP) is designed to overlap communication overheads with the computation based on two key independencies in the program: (1) the send-out operation o_t^l is independent of backward operations $b_t^i (i < l)$, so they could be executed concurrently without blocking each other; (2) the read-in operation i_t^l could update the layer parameters as long as b_t^l was finished, without blocking the subsequent backward operations $b_t^i (i < l)$. Therefore, we can enforce each layer l to start its communication once its gradients are generated after b_t^l, so that the time spent on operation s_t^l could be overlapped with those of $b_t^i (i < l)$.

WFBP is most beneficial for training DL models that have their parameters concentrating at upper layers (FC layers) but computation concentrating at lower layers (CONV layers)[2], e.g. VGG [23] and AdamNet [4, 7]), because it overlaps the communication of top layers (90% of communication time) with the computation of bottom layers (90% of computation time). Besides chain-like NNs, WFBP is generally applicable to other non-chain like structures (e.g. tree-like structures), as the parameter optimization for deep neural networks depends on adjacent layers (and not the whole network), there is always an opportunity for parameter optimization (i.e. computation) and communication from different layers to be performed concurrently.

Some DL frameworks, such as TensorFlow, represent the data dependencies of DL programs using graphs, therefore implicitly enable auto-parallelization. However, they fail on exploring the potential opportunities of parallelization between iterations. For example, TensorFlow needs to fetch the updated parameters from the remote storage at the beginning of each iteration, while it is possible to overlap this communication procedure with the computation procedure of the previous iteration. In comparison, WFBP enforces this overlapping by explicitly pipelining compute, send and receive procedures. We describe our implementation of WFBP in Section 4 and empirically show its effectiveness in Section 5.1.

3.3 Hybrid Communication

While WFBP overlaps communication and computation, it does not reduce the communication overhead. In some situations where the network bandwidth is limited (e.g. commodity Ethernet or the Ethernet is shared with other communication-heavy applications), the communication would still be unacceptably slow. To address the issue, we introduce a hybrid communication (HybComm) strategy that combines the best of PS and SFB, by being aware of both the mathematical property of DL models and the structure of computing clusters.

[2]Most classification models will fall into this family if the number of classes to be classified is large.

Method	Server	Worker	Server & Worker
PS	$2P_1MN/P_2$	$2MN$	$2MN(P_1+P_2-2)/P_2$
SFB	**N/A**	$2K(P_1-1)(M+N)$	**N/A**
Adam (max)	$P_1MN+P_1K(M+N)$	$K(M+N)+MN$	$(P_1-1)(MN+KM+KN)$

Table 1: Estimated communication cost of PS, SFB and Adam for synchrnizing the parameters of a $M \times N$ FC layer on a cluster with P_1 workers and P_2 servers, when batchsize is K.

Our idea comes from two observations: first, as presented in Section 3.1, the synchronization operations $\{S_t^l\}_{l=1}^L$ are independent of each other, meaning that we can use different communication methods for different S_t^l by specializing o_t^l and i_t^l according to the two methods described in Figure 2; second, a NN structure is usually predefined and fixed throughout the training – by measuring the number of parameters needed to transferred, we are able to estimate the communication overhead, so that we can always choose the optimal method even before the communication happens.

Consider training a CNN, the overheads of S_t^l could be explicitly estimated as follows (Table 1): assume the batch size is K, the number of workers and server nodes as P_1 and P_2 (assume parameters are equally partitioned over all server nodes), respectively. On one hand, if l is an FC layer (with shape $M \times N$), synchronizing its parameters via PS will need to transfer $2MN$ parameters for a worker node, $2P_1MN/P_2$ for a server node, and $2MN(P_1+P_2-2)/P_2$ for a node that is both a server and a worker, compared to $2K(M+N)(P_1-1)$ for a single node using SFB. On the other hand, if l is a CONV layer, the updates are indecomposable but sparse, so that we can directly resort to PS. Therefore, the synchronization overheads depend not only on the model (type and shape of the layer), but also the size of the clusters. The optimal solution usually changes with M, N, K and P. HybComm takes into account these factors and allows to dynamically adjust the communication method for different parts of a model – it always chooses the best communication method from available ones whenever it results in fewer communication overheads.

Microsoft Adam [4] employs a different communication strategy from those in Figure 2. Instead of broadcasting SFs across workers, they first send SFs to a parameter server shard, then pull back the whole updated parameter matrices. This seems to reduce the total number of parameters needed to be communicated, but usually leads to load imbalancing; the server node that holds the corresponding parameter shard overloads because it has to broadcast the parameter matrices to all workers ($P_1MN + P_1K(M+N)$ messages need to be broadcasted), which easily causes communication bottleneck (Section 5.3). It is noticeable that, reconstructing gradients from SFs may cause extra computation cost, wich are often negligible compared to communication in GPU based distributed DLs. We describe our implementation of HybComm in the next section, and assess its effectiveness in Section 5.

4 Implementation

We implement Poseidon as a light-weight communication library that could be easily plugged into existing DL frameworks, to create an efficient data-parallel version of them. This section first elaborates Poseidon's system architecture and APIs, and then describes a sample implementation how to modify a framework using Poseidon to enable distributed execution.

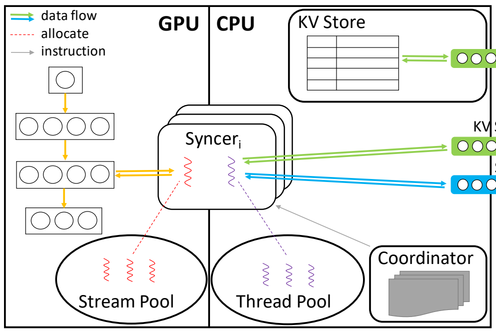

Figure 3: An overview of the architecture of Poseidon.

4.1 System Implementation and APIs

Figure 3 illustrates the architecture of Poseidon: a C++ communication library that manages parameter communication for DL programs running on distributed GPUs. It has three main components: coordinator, that maintains the model and the cluster configuration; KV store, a shared memory key-value store that provides support for parameter server based communication; client library, which is plugged in to DL programs to handle parameter communication. Their APIs are listed in Table 2.

Coordinator. To setup distributed training, the client program (e.g. Caffe) first instantiates Poseidon by creating a coordinator within its process. Coordinators will first collect necessary information, including the cluster information (e.g. the number of workers and server nodes, their TCP-IP addresses) and the model architecture (e.g. the number of layers, layer types, number of neurons and how they are connected, etc.). With the information, the coordinator will initialize the KV stores and the client library with two steps: (1) allocate proper TCP-IP ports for each PS shard and peer worker; (2) determine what parameters should be transmitted via the KV store and what by SFB, and hash the parameters equally to each KV store if necessary, and save the mapping in the information book, which, throughout the whole training, is maintained and synchronized across nodes, and could be accessed elsewhere through coordinator's `Query` API. Besides, the coordinator provides another API `METHOD` that takes in a layer and returns the optimal communication method for it according to the strategy described in Section 3.3 (Algorithm 1).

Algorithm 1 Get the best comm method of layer l

```
1:  function METHOD($l$)
2:      layer_property = Query($l$.name)
3:      $P_1, P_2, K$ = Query('n_worker', 'n_server', 'batchsize')
4:      if layer_property.type == 'FC' then
5:          $M$ = layer_property.width
6:          $N$ = layer_property.height
7:          if $2K(P_1-1)(M+N) \leq \frac{2MN(P_1+P_2-2)}{P_2}$ then
8:              return 'SFB'
9:          end if
10:     end if
11:     return 'PS'
12: end function
```

KV Store. The KV store is implemented based on a bulk synchronous parameter server [28, 7], and instantiated by coordinators on a list of user-specified "server" machines. Each instance of the KV store holds one shard of the globally shared model parameters in the form of a set of KV pairs, of which each KV pair is stored on a chunk of RAM. Poseidon set the size of a KV pair to a fixed small size (e.g. 2MB), so as to partition and distribute model parameters to server nodes as equally as possible, reducing the risk of Ethernet bottleneck. Each KV store instance manages a parameter buffer on RAM, and provides PS-like APIs, such as `Receive` and `Send`, for receiving and applying updates from client libraries, or sending out parameters. It will regularly checkpoint current parameter states for fault tolerance.

Client Library. Poseidon coordinates with DL programs via its client library. Particularly, users plug the client library into their training program, and the client library will create a *syncer* for each NN layer during network assembling (so that each layer one-to-one maps to one syncer), accounting for its parameter synchronization. Each sycner is then initialized, for example, setting up connections to its corresponding PS shards or (remote) peer syncers according to the coordinator's information book, and allocating a small memory buffer for receiving remote parameter matrices or SFs, etc.

The client library manages a CPU thread pool and a GPU stream pool on the worker machine, which can be allocated by the syncer APIs when there is a syncer job created. The syncer has three main APIs, `Send`, `Receive` and `Move`, to be used in client programs.

Method	Owner	Arguments	Description
`Method`	Coordinator	A layer name or index	Get the best communication method of a layer
`Query`	Coordinator	A list of property names	Query information from coordinators' information book
`Send`	Syncer	None	Send out the parameter updates of the corresponding layer
`Receive`	Syncer	None	Receive parameter updates from either parameter server or peer workers
`Move`	Syncer	A GPU stream and a direction indicator	Move contents between GPU and CPU, do transformations and application of updates if needed
`Send`	KV store	updated parameters	Send out the updated parameters
`Receive`	KV store	parameter buffer of KV stores	Receive gradient updates from workers

Table 2: Poseidon APIs for parameter synchronization.

The `Move` API takes care of the memory movement on RAM and GPU memory, and performs necessary computation, e.g. the transformation between SFs and gradients, and the application of updates. It is multi-threaded using the CUDA asynchronous APIs, and will trigger an allocation from the client library's thread/stream pools when a syncer job starts (see L14 of Algorithm 2). The `Send` and `Receive` are communication APIs that synchronize layer parameters across different model replicas. The `Send` API is nonblocking; it sends out parameter updates during backpropagation once they are generated, following the protocol returned by coordinator's `Method` API. The `Receive` API will be called once `Send` is finished. It requests either fresh parameter matrices from the KV stores or SFs from its peer syncers, and will block its current thread until it receives all of what it requested. The received messages are put into the syncer's memory buffer for the `Move` API to fetch.

Managing Consistency. Poseidon implements the bulk synchronous consistency (BSP) model as follows. The client library maintains a binary vector C with length the number of syncers and values reset to zeros at the start of each iteration. A syncer will set its corresponding entry in C as 1 when its job finished, and the client starts next iteration when all entries are 1. While, the KV store maintains a zero-initialized count value for each KV pair at the start of each iteration. Every time when there is an update being applied on a KV pair, its count value is increased by 1. The KV pair will be broadcasted via its `Send` API when its count equals to the number of workers. Poseidon handles stragglers by simply dropping them. Although asynchronous models can alleviate the straggler problem in distributed ML [11], Poseidon focuses on synchronous parallel training, because synchronous execution yields the fastest per-iteration improvement in accuracy for distributed DL (as measured by wall clock time) on GPUs [7, 2] (see Section 5.1).

4.2 Integrate Poseidon with DL Programs

Poseidon could be plugged into most existing DL frameworks to enable efficient distributed execution. Algorithm 2 provides an example. Specifically, one needs to first include Poseidon's client library into the framework, then figure out where the backpropagation proceeds (L6), and insert Poseidon's syner APIs in between gradient generation and application (L7). We demonstrate in Section 5.1 that with slight modifications (150 lines of code for Caffe and 250 lines of code for TensorFlow), both Poseidon-enable Caffe and TensorFlow deliver linear scalings up to 32 GPU machines.

Algorithm 2 Parallelize a DL program using Posiedon

```
1:  function TRAIN(net)
2:      for $iter = 1 \to T$ do
3:          $sync\_count = 0$
4:          net.Forward()
5:          for $l = L \to 1$ do
6:              net.BackwardThrough($l$)
7:              thread_pool.Schedule(sync($l$))
8:          end for
9:          wait_until($sync\_count == net.num\_layers$)
10:     end for
11: end function
12: function SYNC($l$)
13:     stream = stream_pool.Allocate()
14:     syncers[$l$].Move(stream, GPU2CPU)
15:     syncers[$l$].method = coordinator.Method($l$)
16:     syncers[$l$].Send()
17:     syncers[$l$].Receive()
18:     syncers[$l$].Move(stream, CPU2GPU)
19:     sync_count++
20: end function
```

5 Evaluation

In this section, we evaluate Poseidon's performance on scaling up DL with distributed GPUs. We focus on the image classification task where DL is most successfully applied. Our evaluation reveals the following results: (1) Poseidon has little overhead when plugged into existing frameworks; it achieves near-linear speedup across different NNs and frameworks, on up to 32 Titan X-equipped machines; (2) Poseidon's system design effectively improves GPU and bandwidth utilization; (3) Poseidon's communication strategy HybComm effectively alleviates the communication bottleneck, thus achieves better speedups under limited bandwidth; Also, it compares favorably to other communication-reduction methods, such as the SF strategy in Adam [4], and the 1-bit quantization in CNTK [32].

Cluster Configuration. We conduct our experiments on a GPU cluster with each node equipped with a NVIDIA GeForce TITAN X GPU card, an Intel 16-core CPU and 64GB RAM, interconnected via a 40-Gigabit Ethernet switch. All cluster nodes have shared access to a NFS and read data through the Ethernet interface. We run our system on UBUNTU 16.04, with NVIDIA driver version 361.62, CUDA 8.0 and cuDNN v5.

Computation Engines. We deploy Poseidon on two DL frameworks, Caffe [12] and TensorFlow [1]. For Caffe, we use the official version at 2016/06/30 as the single node baseline, and modify it using Poseidon's client library API for distributed execution. For TensorFlow, we use its open source version r0.10, and replace its functionality for distributed execution with Poseidon's client library, and compare it to its original version.

Dataset and Models. Our experiments use three well-known image classification datasets. (1) CIFAR-10 [13], which contains 32×32 colored images of 10 classes, with 50K images for training and 10K for testing; (2) ILSVRC12 [20], a subset of ImageNet22K that has 1.28 million of training images and 50K validation images in 1,000 categories; (3) ImageNet22K [20], the largest public dataset for image classification, including 14,197,087 labeled images from 21,841 categories.

We test Poseidon's scalability across different neural networks: (1) CIFAR-10 quick: a toy CNN from Caffe that converges at 73% accuracy for classifying images in CIFAR-10 dataset; (2) GoogLeNet [24]: a 22-layer CNN with 5M parameters. (3) Inception-V3 [25]: the ImageNet winner, an improved version of GoogLeNet from TensorFlow; (4) VGG19: A popular feature extraction network in the computer vision community [23] that has 16 CONV layers and 3 FC layers, in total 143M parameters; (5) VGG19-22K: we modify the VGG19 network by replacing its 1000-way classifier with a 21841-way classifier, to classify images from the ImageNet22K dataset. The modified network has 229M parameters. (6) ResNet-152: the ImageNet winner network with 152 layers. We list their statistics and configurations in Table 3.

Metrics. In this paper, we mainly focus on metrics that measure the system performance, such as speedups on throughput (number of images scanned per second). Our experiments focus on medium-scale distributed cluster with up to 32 machines, which distributed DL empirically benefits most from. Larger clusters require larger batch sizes, which hurt the convergence rate of each iteration [3, 7]. For completeness, we also report the statistical performance (time/epoch to converge) on ResNet-152. Poseidon uses synchronized replication which enables many models to converge in fewer steps [1, 7, 3, 2].

5.1 Scalability

To demonstrate Poseidon's scalability, we train CNNs using Poseidon with different computational engines, and compare different systems in terms of their speedups on throughput. For Caffe engine, we train GoogLeNet VGG19 and VGG19-22K networks; for TensorFlow engine, we train Inception-V3, VGG-19, VGG19-22K.

Model	# Params	Dataset	Batchsize
CIFAR-10 quick	145.6K	CIFAR10	100
GoogLeNet	5M	ILSVRC12	128
Inception-V3	27M	ILSVRC12	32
VGG19	143M	ILSVRC12	32
VGG19-22K	229M	ImageNet22K	32
ResNet-152	60.2M	ILSVRC12	32

Table 3: Neural networks for evaluation. Single-node batchsize is reported. The batchsize is chose based on the standards reported in literature (usually the maximum batch size that can fill in the GPU memory).

Caffe Engine. Figure 4 shows the throughput vs. number of workers when training the three networks using Caffe engine, given 40GbE Ethernet bandwidth available. We compare the following systems: (1) *Caffe*: unmodified Caffe that executes on a single GPU; (2) *Caffe+PS*: we parallelize Caffe using a vanilla PS, i.e. the parameter synchronization happens sequentially after the backpropagation in each iteration; (3) *Caffe+WFBP*: Parallelized Caffe using Poseidon so the communication and computation are overlapped. However, we disable HybComm so that parameters are synchronized onlyvia PS; (4) *Poseidon*: the full version of Poseidon-Caffe.

Poseidon shows little overheads when combined with Caffe; running on a single node with no communication involved, Poseidon-Caffe can process 257, 35.5 and 34.2 images per second when training GoogLeNet, VGG19 and VGG19-22K, respectively, as compared to the original Caffe, which can process 257, 35.5 and 34.6 images, and Caffe+PS, which can only process 213.3, 21.3 and 18.5 images per second, due to the overheads caused by memory copy operations between RAM and GPU, which have been overlapped by Poseidon with the computation. In distributed environment, the rescheduling of computation and communication significantly improves the throughput: when training GoogLeNet and VGG19, incorporating WFBP achieves almost linear scalings up to 32 machines, and for the larger VGG19-22K network, Caffe+WFBP achieves 21.5x speedup on 32 machines. We conclude that rescheduling and multi-threading the communication and computation are key to the performance of distributed DL on GPUs, even when the bandwidth resource is abundant. Poseidon provides an effective implementation to overlap these operations for DL frameworks, to guarantee better GPU utilization.

When the available bandwidth is sufficient, Poseidon's HybComm strategy shows small improvement on training GoogLeNet and VGG19. However, when training VGG19-22K which has three FC layers that occupy 91% of model parameters, it improves over Caffe-WFBP from 21.5x to 29.5x on 32 nodes. It is noticeable that we also extend Poseidon-Caffe to support single-node multi-GPU environment, which shows linear speedups with up to 4 GPUs when training all three networks, outperform-

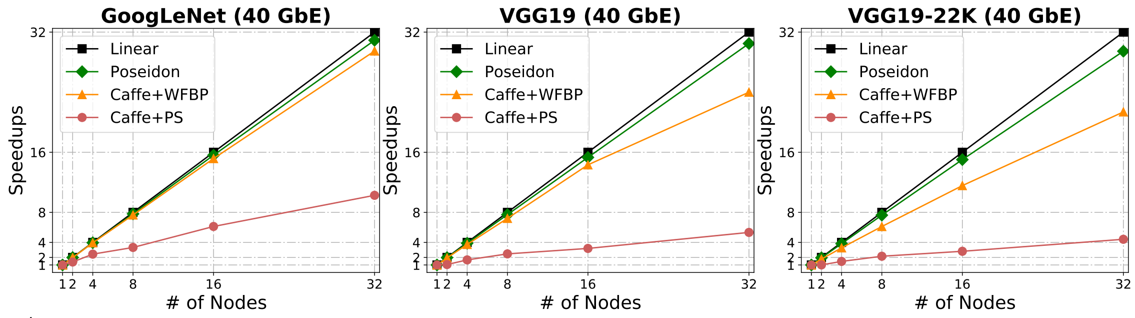

Figure 4: Throughput scaling when training GoogLeNet, VGG19 and VGG19-22K using Poseidon-parallelized Caffe and 40GbE bandwidth. Single-node Caffe is set as baseline (i.e. speedup = 1).

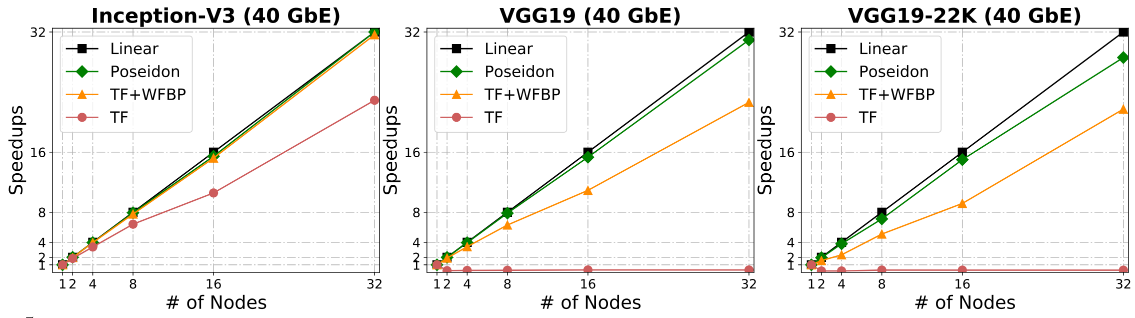

Figure 5: Throughput scaling when training Inception-V3, VGG19 and VGG19-22K using Poseidon-parallelized TensorFlow and 40GbE bandwidth. Single-node TensorFlow is set as baseline (i.e. speedup = 1).

ing Caffe's multi-GPU version, which shows only 3x and 2x speedups when training GoogLeNet and VGG19.

TensorFlow Engine. We also modify TensorFlow using Poseidon, and compare the following systems in terms of speedup on throughput: (1) *TF*: TensorFlow with its original distributed executions; (2) *TF+WFBP*: we modify TensorFlow using Poseidon's client library. Specifically, we change the training operator in TensorFlow, so that instead of being applied, the parameter updates will be synchronized via *Poseidon's PS interface with WFBP*; (3) *Poseidon*: the full version of Poseidon-parallelized TensorFlow with HybComm enabled.

We train Inception-V3, VGG19 and VGG19-22K models and report the results in Figure 5. Running on a single node, Poseidon processes 43.2, 38.2 and 34.5 images per second on training Inception-V3, VGG19 and VGG19-22K, while original TensorFlow processes 43.2, 38.5 and 34.8 images per second on these three models, respectively – little overhead is introduced by our modification. In distributed execution, Poseidon achieves almost linear speedup on up to 32 machines. Distributed TensorFlow, however, demonstrates only 10x speedup on training Inception-V3 and even fails to scale on training the other two networks in our experiments.

To investigate the problem of TensorFlow and explain how Poseidon improves upon it, we illustrates in Figure 6 the (averaged) ratio of busy and stall time of a GPU when training the three networks using different systems on 8 nodes. Observe that Poseidon keeps GPUs busy in most of the time, while TensorFlow wastes much time on waiting for parameter synchronization. The inefficiency of distributed TensorFlow stems from two sources. First, TensorFlow partitions model parameters in a coarse-grained granularity – each tensor (instead of a KV pair) in the model is assigned to a PS shard. A big tensor (such as the parameter matrix in VGG19) is highly likely to create communication bottleneck on its located server node. Poseidon fixes this problem by partitioning parameters among server nodes in a finer-grained granularity using KV pairs, so that every node has evenly distributed communication load; as an evidence, TF-WFBP demonstrates higher computation-to-stall ratio in Figure 6. Second, TensorFlow has no strategy to reduce the communication overheads while Poseidon's HybComm effectively reduces the size of messages. As a result, Poseidon further improves upon TF-WFBP from 22x to 30x on 32 nodes.

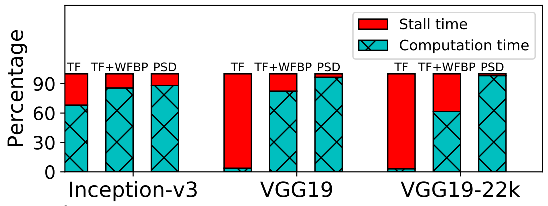

Figure 6: Breakdown of GPU computation and stall time when training the three networks on 8 nodes using different systems.

Statistical Performance. For completeness, we report in Figure 7 the statistical performance for training ResNet-152 using Poseidon. Poseidon achieves near-linear speedups on both system throughput and statistical convergence: Poseidon delivers 31x speedup in terms of throughput, and reaches 0.24 reported error with less than 90 epochs with both 16 and 32 nodes – thus linear scale-ups in terms of time to accuracy, compared to 8 nodes with batchsize = 32×8, which is a standard setting as in [10], echoing recent results that synchronous training on distributed GPUs yields better performance than asynchronous training in terms of time to quality for some neural networks [7, 2].

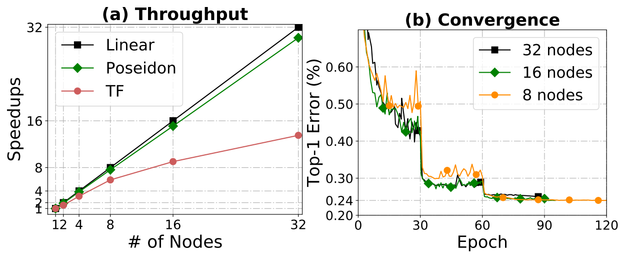

Figure 7: (a) Speedup v.s. number of nodes and (b) Test error v.s. epochs for training ResNet-152 using Poseidon-TensorFlow and the original TensorFlow.

5.2 Bandwidth Experiments

To further assess Poseidon's HybComm strategy, we simulate the environment where network bandwidth is limited. We use Linux traffic control tool `tc` to lower the available bandwidth on each node, and compare the training throughput between with and without HybComm. We focus on Caffe engine in this section because it is lighter and less optimized than TensorFlow.

Figure 8 plots the speedup on throughput vs. number of workers when training GoogLeNet, VGG19 and VGG19-22K with different maximum bandwidth. Clearly, limited bandwidth prevents a standard PS-based system from linearly scaling with number of nodes; for example, given 10GbE bandwidth (which is a commonly-deployed Ethernet configuration in most cloud computing platforms), training VGG19 using PS on 16 nodes can only be accelerated by 8x. This observation confirms our argument that limited bandwidth would result in communication bottleneck when training big models on distributed GPUs. Fortunately, Poseidon significantly alleviates this issue. Under limited bandwidth, it constantly improves the throughput by directly reducing the size of messages needed to be communicated, especially when the batch size is small; when training VGG19 and VGG19-22K, Poseidon achieves near-linear speedup on 16 machines using only 10GbE bandwidth, while an optimized PS would otherwise need 30GbE or even higher to achieve. Note that Poseidon will never underperform a traditional PS scheme because it will reduce to a parameter server whenever it results in less communication overheads; for instance, we observe that Poseidon reduces to PS when training GoogLeNet on 16 nodes, because GoogleNet only has one thin FC layer (1000×1024) and is trained with a large batch size (128).

5.3 Comparisons to Other Methods

In this section, we compare Poseidon against other communication methods, including Adam [4] and CNTK 1-bit quantization [32], and show Poseidon's advantages.

Adam. To save bandwidth, Adam [4] synchronizes the parameters of a FC layer by first pushing SFs generated on all workers to a PS node, and then pulling back the full parameter matrices thereafter. As direct comparisons to Adam [4] are inaccessible, we implement its strategy in Poseidon, and compare it (denoted as *Adam*) to *TF-WFBP* and *Poseidon* by monitoring the network traffic of each machine when training VGG19 on 8 nodes using TensorFlow engine. As shown in Figure 9, the communication workload is highly imbalanced using Adam's strategy. Unlike a traditional PS (TF-WFBP) where the parameters are equally distributed over multiple shards, Adam cannot partition the parameters of FC layers because of their usage of SFs. Although the "push" operation uses SFs to reduce message size, the "pull" requires some server nodes to broadcast big matrices to each worker node, which creates bursty traffic that results in communication bottleneck on them. By contrast, Poseidon either equally partitions parameters over multiple PS shards, or transmit SFs among peer workers, both are communication load-balanced that avoid bursty communication situations. Quantitatively, Adam delivers 5x speedup with 8 nodes when training VGG19.

CNTK 1-bit Quantization. We also compare Poseidon to the 1-bit quantization technique proposed in CNTK [32]. Following CNTK, we quantize the parameters in FC layers, and add the quantization residual to the parameter updates of next iteration (denoted as *Poseidon-1bit*). We then train the CIFAR-10 quick network, and plot the training loss and test error vs. iterations for two systems (both have linear scaling on throughput). As in Figure 10, 1-bit quantization yields worse convergence in terms of accuracy – on 4 GPUs, it achieves 50% accuray after 3K iterations of training, while Poseidon quickly converges to 70% accuracy at iteration 1000. We conjecture that this is caused by the quantization residual, which is equivalent to delayed updates that may hurt the convergence performance when training NNs on images, as confirmed by [7].

6 Related Work

PS-based Distributed DL Systems. Based on the parameter server [28, 17] architecture, a number of CPU-based distributed DL systems have been developed, such as [33, 26, 8, 15] and Adam [4]. They are purely PS-based systems on CPU-only clusters, whereas we address the more challenging case of GPU clusters.

Scaling up DL on distributed GPUs is an active field of research. Coates *et al.* [5] build a GPU-based multi-

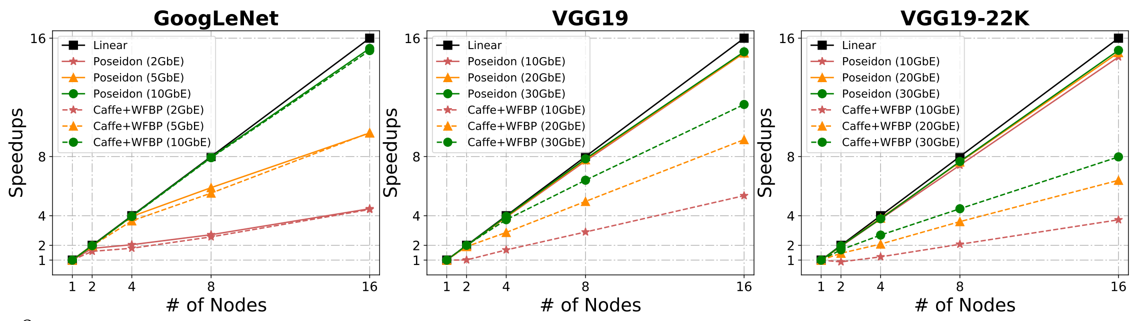

Figure 8: Throughput scaling when training GoogLeNet, VGG19 and VGG19-22K using Poseidon-parallelized Caffe with *varying network bandwidth*. Single-node Caffe is set as baseline (speedup = 1).

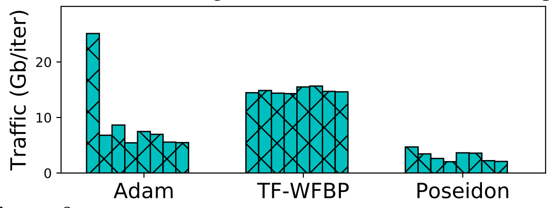

Figure 9: Communication load averaged over multiple iterations when training VGG19 on 8 nodes using *Adam*, *TF-WFBP* and *Poseidon* with TensorFlow engine. Each bar represents the network traffic on a node.

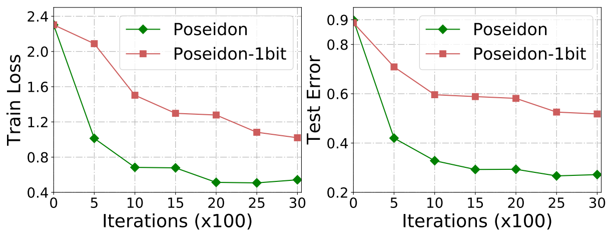

Figure 10: Training loss and test accuracy vs. iteration when training CIFAR-10 quick network using Poseidon and 1-bit quantization on 4GPUs with Caffe engine.

machine system for DL using model parallelism rather than data parallelism, and their implementation is rather specialized for a fixed model structure while demanding specialized hardware, such as InfiBand networking. TensorFlow [1] is Google's distributed ML platform that uses a dataflow graph to represent DL models, and synchronizes model parameters via a parameter server. It therefore cannot dynamically adjust its communication method depending on the layer and cluster information as Poseidon does (Section 5.1). MXNet [3] is another DL system that uses PS for distributed execution, and supports TensorFlow-like graph representations for DL models. By auto-parallelizing independent subgraphs, both MXNet and TensorFlow can implicitly overlap the communication and computation. By contrast, Poseidon has a more explicit way to overlap them via its client library. Thus, Poseidon can be also easily used to parallelize non-graph-based frameworks. Moreover, both MXNet and TensorFlow do not address the bottleneck caused by limited network bandwidth, which undermines their scalability when training large models with dense layers (e.g. VGGNet, big softmax). Besides, Cui *et al.* propose GeePS [7] that manages the limited GPU memory and report speedups on distributed GPUs. Same with TensorFlow and MXNet, GeePS does not address the issue of limited network bandwidth. Therefore, Poseidon's technique could be combined with them to enable better training speedup. Also of note are several efforts to port Caffe onto other distributed platforms, such as SparkNet [19] and YahooCaffe [30], the former reports a 4-5 times speedup with 10 machines (and hence less scalability than our results herein).

Other distributed ML systems. CNTK [32] is Microsoft's distributed DL framework that supports distributed executions and addresses the problem of communication bottleneck via the 1-bit quantization technique. CNTK demonstrates little negative impact on convergence in speech domains [22, 21]. However, in some other domains (Section 5.3), the performance is usually compromised by noisy gradients [1, 7]. By contrast, Poseidon's HybComm reduces the communication while always guaranteeing synchronous training. There are also growing interest in parallelizing ML applications using peer-to-peer communication, such as MALT [16], SFB [29] and Ako [27]. Poseidon draws inspiration from these worksm but goes one step further as it is an adaptive best-of-both-worlds protocol, which will select client-server communication whenever it would result in fewer overheads. Therefore, any advanced technique from these frameworks could be adopted by Poseidon to address the communication bottleneck.

7 Conclusion

We present Poseidon, a scalable and efficient communication library for large-scale DL on distributed GPUs. Poseidon's design focuses on efficiently harnessing multiple, distributed GPUs with limited communication bandwidth, in order to maximally scale up existing single-machine DL frameworks. Poseidon is orthogonal to TensorFlow, MXNet or other DL frameworks – the techniques present in Poseidon could be used to produce a better distributed version of them as we have shown in the paper. We empirically show that Poseidon constantly delivers linear speedups using up to 32 nodes and limited bandwidth, on a variety of neural network models, datasets and computation engines and compares favorably to Adam and Microsoft CNTK.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. *arXiv preprint arXiv:1605.08695* (2016).

[2] Chen, J., Monga, R., Bengio, S., and Jozefowicz, R. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981* (2016).

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

[4] Chilimbi, T., Apacible, Y. S. J., and Kalyanaraman, K. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI* (2014).

[5] Coates, A., Huval, B., Wang, T., Wu, D. J., Ng, A. Y., and Catanzaro, B. Deep Learning with COTS HPC Systems. In *ICML* (2013).

[6] Collobert, R., Kavukcuoglu, K., and Farabet, C. Torch7: A Matlab-like Environment for Machine Learning. In *NIPSW* (2011).

[7] Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., and Xing, E. P. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 4.

[8] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large Scale Distributed Deep Networks. In *NIPS* (2012).

[9] Deng, L., Li, J., Huang, J.-T., Yao, K., Yu, D., Seide, F., Seltzer, M. L., Zweig, G., He, X., Williams, J., Gong, Y., and Acero, A. Recent Advances in Deep Learning for Speech Research at Microsoft. In *ICASSP* (2013).

[10] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385* (2015).

[11] Ho, Q., Cipar, J., Cui, H., Kim, J. K., Lee, S., Gibbons, P. B., Gibson, G. A., Ganger, G. R., and Xing, E. P. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS* (2013).

[12] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM* (2014).

[13] Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images. Master's thesis, University of Toronto, 2009.

[14] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS* (2012).

[15] Le, Q. V., Monga, R., Devin, M., Chen, K., Corrado, G. S., Dean, J., and Ng, A. Y. Building High-level Features Using Large Scale Unsupervised Learning. In *ICML* (2012).

[16] Li, H., Kadav, A., Kruus, E., and Ungureanu, C. Malt: distributed data-parallelism for existing ml applications. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 3.

[17] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 583–598.

[18] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient Estimation of Word Representations in Vector Space. In *ICLRW* (2013).

[19] Moritz, P., Nishihara, R., Stoica, I., and Jordan, M. I. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051* (2015).

[20] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *IJCV* (2015), 1–42.

[21] Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *INTERSPEECH* (2014), pp. 1058–1062.

[22] Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. On parallelizability of stochastic gradient descent for speech dnns. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2014), IEEE, pp. 235–239.

[23] Simonyan, K., and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR* (2015).

[24] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *CVPR* (2015).

[25] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567* (2015).

[26] Wang, W., Chen, G., Dinh, T. T. A., Gao, J., Ooi, B. C., Tan, K.-L., and Wang, S. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *MM* (2015).

[27] Watcharapichat, P., Morales, V. L., Fernandez, R. C., and Pietzuch, P. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016), ACM, pp. 84–97.

[28] Wei, J., Dai, W., Qiao, A., Ho, Q., Cui, H., Ganger, G. R., Gibbons, P. B., Gibson, G. A., and Xing, E. P. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *SoCC* (2015).

[29] Xie, P., Kim, J. K., Zhou, Y., Ho, Q., Kumar, A., Yu, Y., and Xing, E. Distributed Machine Learning via Sufficient Factor Broadcasting. In *arXiv* (2015).

[30] Yahoo. `http://yahoohadoop.tumblr.com/post/129872361846/large-scale-distributed-deep-learning-on-hadoop`.

[31] Yan, Z., Zhang, H., Jagadeesh, V., DeCoste, D., Di, W., and Piramuthu, R. Hd-cnn: Hierarchical deep convolutional neural network for image classification. *ICCV* (2015).

[32] Yu, D., Eversole, A., Seltzer, M., Yao, K., Huang, Z., Guenter, B., Kuchaiev, O., Zhang, Y., Seide, F., Wang, H., et al. An introduction to computational networks and the computational network toolkit. Tech. rep.

[33] Zou, Y., Jin, X., Li, Y., Guo, Z., Wang, E., and Xiao, B. Mariana: Tencent Deep Learning Platform and its Applications. In *VLDB Endowment* (2014).