Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training

Youjie Li[†], **Mingchao Yu**[*], **Songze Li**[*], **Salman Avestimehr**[*],
Nam Sung Kim[†], and **Alexander Schwing**[†]

[†]University of Illinois at Urbana-Champaign
[*]University of Southern California

Abstract

Distributed training of deep nets is an important technique to address some of the present day computing challenges like memory consumption and computational demands. Classical distributed approaches, synchronous or asynchronous, are based on the parameter server architecture, *i.e.*, worker nodes compute gradients which are communicated to the parameter server while updated parameters are returned. Recently, distributed training with AllReduce operations gained popularity as well. While many of those operations seem appealing, little is reported about wall-clock training time improvements. In this paper, we carefully analyze the AllReduce based setup, propose timing models which include network latency, bandwidth, cluster size and compute time, and demonstrate that a pipelined training with a width of two combines the best of both synchronous and asynchronous training. Specifically, for a setup consisting of a four-node GPU cluster we show wall-clock time training improvements of up to $5.4\times$ compared to conventional approaches.

1   Introduction

Deep nets [24, 3] are omnipresent across fields from computer vision and natural language processing to computational biology and robotics. Across domains and tasks they have demonstrated impressive results by automatically extracting hierarchical abstractions of representations from many different datasets. The surge in popularity pivoted in the 2010s, with impressive results being demonstrated on the ImageNet dataset [21, 41]. Since then, deep nets have been applied to many more tasks. Prominent examples include recognition of places [52], playing of Atari games [33, 34], and the game of Go [44]. Common to all those methods is the use of large datasets to fuel the many layers of deep nets.

Importantly, in the last few years, the number of layers, or more generally the depth of the computation tree has increased significantly from a few layers for LeNet [25] to several 100s or 1000s [13, 23]. Inherent to the increasing complexity of the computation graph is an increase in training time and often also an increase in the amount of data that is processed. Traditionally, computational performance increases do not keep up with the desired processing needs despite the use of accelerators like GPUs.

Beyond accelerators, parallelization of computation on multiple computers is therefore popular. However, it requires frequent communication to exchange a large amount of data among compute nodes while the bandwidth of network interfaces is limited. This in turn significantly diminishes the benefit of parallelization, as a substantial fraction of training time is spent to communicate data. The fraction of time spent on communication is further increased when applying accelerators [15, 7, 37, 51, 47, 48, 43], as they decrease computation time while leaving communication time untouched.

To take advantage of parallelization across machines, a variety of approaches have been developed starting from the popular MapReduce paradigm [9, 50, 18, 36]. Despite their benefits, communication heavy training of deep nets is often based on custom implementations [8, 6, 35, 19] relying on the parameter server architecture [27, 26, 14], where the centralized server aggregates the gradients from workers and distributes the updated weights, either in a synchronous or asynchronous manner. Recent research proposed to use a decentralized architecture with global synchronization among nodes [12, 32]. However, in common to all the aforementioned techniques, little is reported regarding the timing analysis of distributed deep net training.

In this paper, we analyze the wall-clock time trade-offs between communication and computation. To this end we develop a model to assess the training time based on a set of parameters such as latency, cluster size, network bandwith, model size, *etc*. Based on the results of our model we develop Pipe-SGD, a framework with pipelined training and balanced communication, and show its convergence properties by adjusting proofs of [22, 14]. We also show what types of compression can be efficiently included in an AllReduce based framework. Finally, we assess the speedups of our proposed approach on a GPU cluster of four nodes with 10GbE network, showing wall-clock time training improvements by a factor of $3.2 \sim 5.4\times$ compared to conventional centralized and decentralized approaches without degradation in accuracy.

2 Background

General Training of Deep Nets: Training of deep nets involves finding the parameters w of a predictor $F(x,w)$ given input data x. To this end we minimize a loss function $\ell(F(x,w),y)$ which compares the predictor output $F(x,w)$ for given data x and the current w to the ground-truth annotation y. Given a dataset $\mathscr{D} = \{(x,y)\}$, finding w is formally summarized via:

$$\min_w f_{\mathscr{D}}(w) := \frac{1}{|\mathscr{D}|} \sum_{(x,y)\in\mathscr{D}} \ell(F(x,w),y). \tag{1}$$

Optimization of the objective given in Eq. (1) w.r.t. the parameters w, *e.g.*, via gradient descent using $\frac{\partial f_{\mathscr{D}}}{\partial w}$, can be challenging due to not only the complexity of evaluating the predictor $F(x,w)$ and its derivative, but also the size of the dataset $|\mathscr{D}|$. Consequently, stochastic gradient descent (SGD) emerged as a popular technique. We randomly sample a subset $\mathscr{B}$ of the dataset, often also referred to as a minibatch. Instead of computing the gradient on the entire dataset $\mathscr{D}$, we approximate it using the samples in the minibatch, *i.e.*, we assume $\frac{\partial f_{\mathscr{D}}}{\partial w} \approx \frac{\partial f_{\mathscr{B}}}{\partial w}$. However, for present day datasets and predictors, computation of the gradient $\frac{\partial f_{\mathscr{B}}}{\partial w}$ on a single machine is still challenging. Minibatch sizes $|\mathscr{B}|$ of less than 20 samples are common, *e.g.*, when training for semantic image segmentation [5].

Distributed Training of Deep Nets: To train larger models or to increase the minibatch size, distributed training on multiple compute nodes is used [8, 14, 6, 26, 27, 35, 15]. A popular architecture to facilitate distributed training is the parameter server framework [14, 26, 27]. The parameter server maintains a copy of the current parameters, and communicates with a group of worker nodes, each of which operates on a small minibatch to compute local gradients based on the retrieved parameters w. Upon having completed its task, the worker shares the gradients with the parameter server. Once the parameter server has obtained all or some of the gradients it updates the parameters using the negative gradient direction and afterwards shares the latest values with the workers.

Asynchronous updates where each worker independently pulls w from the server, computes its own local gradient, and pushes results back are available and illustrated in Fig. 1 (a). Due to the asynchrony, minimal synchronization overhead is traded with staleness of gradients. Methods for staleness control exist, which bound the number of delay steps [14]. However, note that stale gradients may slow down training significantly.

Importantly, all those frameworks are based on a centralized compute topology which forms a communication bottleneck, increasing the training time as the cluster size scales. The time taken by pushing gradient, update, and pulling w can be linear in the cluster size due to network congestion.

Therefore, most recently, decentralized training frameworks gained popularity in both the synchronous and asynchronous setting [29, 30]. However, those approaches assume decentralized workers are either completely synchronous (as in Fig. 1 (b)) or completely asynchronous, which requires to either deal with long execution time every iteration or pay for uncontrolled gradient staleness.
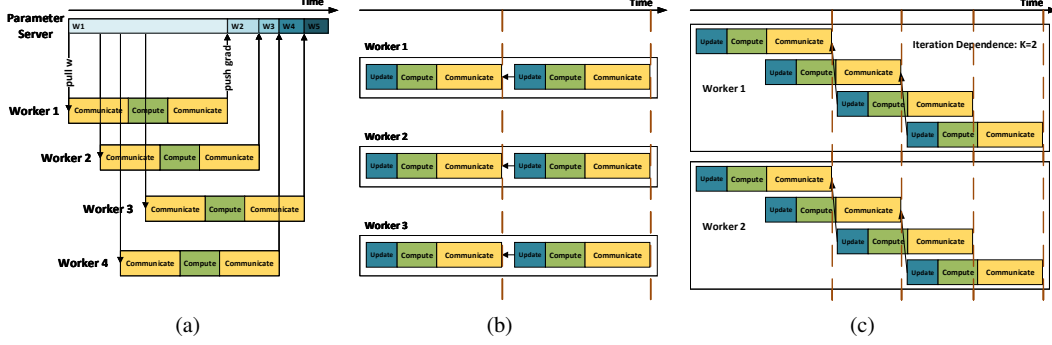
Figure 1: Comparison between different distributed learning frameworks: (a) parameter server with asynchronous training, (b) decentralized synchronous training, and (c) decentralized pipeline training.

Compression in Distributed Training: As the model size increases and cluster size scales, communication overhead in distributed learning system dominates the training time, *e.g.*, up to $80 \sim 90\%$ even in a high-speed network environment [28, 10]. To reduce the communication time, various compression algorithms have been proposed recently [42, 45, 11, 4, 49, 32, 2], some of which focus on reducing the precision of communicated gradients through scalar quantization into 1 bit, while others focus on reducing the quantity of gradients to be transferred. Most compression works, however, only emphasize on achieving high compression ratio or low loss in accuracy without reporting the wall-clock training time.

In practice, compression without knowledge of the communication process is usually counter-productive [28], *i.e.*, the total training time often increases. This is due to the fact that AllReduce is a multi-step algorithm which requires transferred gradients to be compressed and decompressed repeatedly with a worst-case complexity linear in the cluster size, as we discuss below in Sec. 3.2.

3 Decentralized Pipelined Stochastic Gradient Descent

Overview: To address the aforementioned issues (network congestion for a central server, long execution time for synchronous training, and stale gradients in asynchronous training) we propose a new decentralized learning framework, Pipe-SGD, shown in Fig. 1 (c). It balances communication among nodes via AllReduce and pipelines the local training iterations to hide communication time.

We developed Pipe-SGD by analyzing a timing model for wall-clock train time under different resource conditions using various communication approaches. We find that the proposed Pipe-SGD is optimal when gradient updates are delayed by only one iteration and the time taken by each iteration is dominated by local computation on workers. Moreover, we found lossy compression to further reduce communication time without impacting accuracy.

Due to local pipelined training, balanced communication, and compression, the communication time is no longer part of the critical path, *i.e.*, it is completely masked by computation, leading to linear speedup of end-to-end training time as the cluster size scales. Finally, we prove the convergence of Pipe-SGD for convex and strongly convex objectives by adjusting the proof of [22, 14].

3.1 Timing Models and Decentralized Pipe-SGD

Timing Model: We propose timing models based on decentralized synchronous SGD to analyze the wall-clock runtime of training. Each training iteration consists of three major stages: model update, gradient computation, and gradient communication. Classical synchronous SGD (Fig. 1 (b)) runs local iterations on workers sequentially, *i.e.*, each update depends on the gradient from the previous iteration, *i.e.*, the iteration dependency is 1. Therefore the total runtime of synchronous SGD can be formulated easily as:

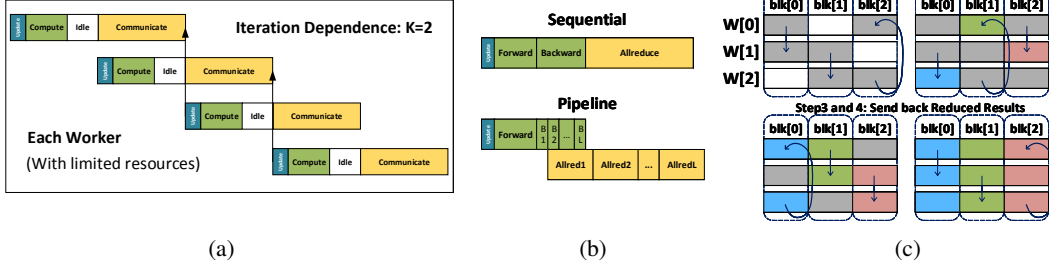$$l_{total\_sync} = T \cdot (l_{up} + l_{comp} + l_{comm}), \tag{2}$$

3

Figure 2: Timing model of Pipe-SGD: (a) each worker with limited resources, (b) sequential v.s. pipelined gradient communication, and (c) an example of gradient communication: Ring-AllReduce.

where T denotes the total number of training iterations and $l_{up}, l_{comp}, l_{comm}$ refer to the time taken by update, compute, and communication, respectively. It is apparent that synchronous SGD depends on the sum of execution time taken by all stages, which leads to long end-to-end training time.

On the contrary, Pipe-SGD relaxes the iteration dependency to K, *i.e.*, each update depends only on the gradients of the K-th last iteration. This enables interleaving between neighboring iterations while maintaining globally synchronized communication, as shown in Fig. 1 (c). If we assume ideal conditions where both computation resources (CPU, GPU, other accelerators) and communication resources (communication links) are unlimited or abundant in counts/bandwidth, then the total runtime of Pipe-SGD is:

$$l_{total\_pipe} = T/K \cdot (l_{up} + l_{comp} + l_{comm}), \tag{3}$$

where K denotes the iteration dependency or the gradient staleness. We observe that the end-to-end training time in Pipe-SGD can be shortened by a factor of K. However, the ideal resource assumption doesn't hold in practice, because both computation and communication resources are strictly limited on each worker node in today's distributed systems. As a result, the timing model for distributed learning is resource bound, either communication or computation bound, as shown in Fig. 2 (a), *i.e.*, the total runtime is:

$$l_{total\_pipe} = T \cdot \max(l_{up} + l_{comp}, l_{comm}), \tag{4}$$

where the total runtime is solely determined by either computation or communication resources, regardless of K (when $K \geq 2$). Also, since gradient updates are always delayed by $(K-1)$ iterations, increasing $K > 2$ only harms, *i.e.*, the optimal value of $K = 2$ for Pipe-SGD with limited resources. Hence, the staleness of gradients is limited to 1 iteration, *i.e.*, the minimal staleness achievable in asynchronous updates. Besides, we generally prefer a computation-bound setting for distributed training system, *i.e.*, $l_{up} + l_{comp} > l_{comm}$. To achieve this we discuss compression techniques in Sec. 3.2.

In addition to pipelined execution of iterations, we also analyze pipelined gradient communication within each iteration to reduce train time. Computation of gradients, *i.e.*, the backward-pass, and communication of gradients are often executed in a strictly sequential manner (see Fig. 2 (b)). However, pipelined gradient communication, *i.e.*, communicating gradients immediately after they are computed, is feasible. Again, we assume limited resources and compare the sequential and pipelined gradient communication in Fig. 2 (b).

To analyze the detailed timing of the those two approaches, we use the timing models for communication [46]. Communication of gradients is an AllReduce operation which aggregates the gradient vector from all workers, performs the sum reduction element-wise, and then sends the result back to all. In practice, the underlying algorithms are much more involved [46]. For example, Ring-AllReduce, one of the fastest AllReduce algorithms, performs gradient aggregation collectively among workers through balanced communication. As shown in Fig. 2 (c), each worker transmits only a block of the entire gradient vector to its neighbor and performs the sum reduction on the received block. This "transmit-and-reduce" runs in parallel on all workers, until the gradient blocks are fully reduced on a worker (different for each block). Afterwards those fully reduced blocks are sent back to the remaining workers along the virtual ring. This approach optimally utilizes the network bandwidth of all nodes.

4

Algorithm 1: Decentralized Pipe-SGD training algorithm for each worker.

On the computation thread of each worker:

1: Initialize by the same model $w[0]$, learning rate γ, iteration dependency K, and number of iterations T.
2: **for** $t = 1,\ldots,T$ **do**
3:     Wait until aggregated gradient g^c_{sum} in compressed format at iteration $[t-K]$ is ready
4:     Decompress $g_{sum}[t-K] \leftarrow \mathcal{D}(g^c_{sum}[t-K])$
5:     Update $w[t] \leftarrow w[t-1] - \gamma \cdot g_{sum}[t-K]$
6:     Load a batch $\mathscr{B}$ of training data
7:     Forward pass to compute current loss $f_{\mathscr{B}}$
8:     Backward pass to compute gradient $g_{local}[t] \leftarrow \frac{\partial f_{\mathscr{B}}}{\partial w[t]}$
9:     Compress $g^c_{local}[t] \leftarrow \mathscr{C}(g_{local}[t])$
10:    Denote local gradient $g^c_{local}[t]$ as ready
11: **end for**

On the communication thread of each worker:

1: Initialize aggregated gradients g^c_{sum} of iteration $[1-K, 1-K+1,\ldots,0]$ as zero and mark them as ready
2: **for** $t = 1,\ldots,(T-K)$ **do**
3:     Wait until local gradient $g^c_{local}[t]$ is ready
4:     AllReduce $g^c_{sum}[t] \leftarrow \sum g^c_{local}[t]$
5:     Denote aggregated gradient $g^c_{sum}[t]$ as ready
6: **end for**

Adopting the Ring-AllReduce model of [46], we obtain the total runtime of Pipe-SGD with sequential gradient communication under the limited resource assumption via:

$$l_{total\_pipe\_s} = T \cdot \max\left(l_{up} + l_{for} + l_{back},\ 2(p-1)\cdot\alpha + 2(\frac{p-1}{p})\cdot n\cdot\beta + (\frac{p-1}{p})\cdot n\cdot\gamma + S\right),\quad (5)$$

where l_{for} and l_{back} denote forward-pass and backward-pass time, p denotes the number of workers, α the network latency, n the model size in bytes, β the byte transfer time, γ the byte sum reduction time, and S the global synchronization time.

Similarly, we obtain the total runtime of Pipe-SGD with pipelined gradient communication via:

$$l_{total\_pipe\_p} = T \cdot \max\left(l_{up} + l_{for} + l_b,\ 2(p-1)L\cdot\alpha + 2(\frac{p-1}{p})\cdot n\cdot\beta + (\frac{p-1}{p})\cdot n\cdot\gamma + L\cdot S\right),\quad (6)$$

where L denotes the number of gradient segments, and l_b denotes the backward-pass time taken by the first segment, which is negligible.

Based on Eq. (5) and Eq. (6) we note: if a pipelined system remains communication bound, then sequential gradient communication is preferred over the pipelined gradient communication (Eq. (5) is smaller than Eq. (6) due to positive L). In practice, distributed training of large models is often communication bound, making sequential exchange the best option.

To sum up, based on our timing models, we find: **Pipe-SGD is optimal for $K = 2$, system is compute bound (after compression), and sequential gradient communication is used**. Note that although our model is derived based on the Ring-AllReduce, this conclusion also applies to other AllReduce algorithms, such as recursive doubling, recursive halving and doubling, pairwise exchange, *etc.* [46].

Decentralized Pipeline SGD: Guided by the timing models, we develop the decentralized Pipe-SGD framework illustrated in Fig. 1 (c) where neighboring training iterations on workers are interleaved with a width of $K = 2$ while the execution within each iteration remains strictly sequential. Decentralized workers perform pipelined training in parallel with synchronization on gradient communication after every iteration. Due to the synchronous nature of our framework, the gradient update is always delayed by $K - 1$ iterations, which enforces a deterministic rather than an uncontrolled staleness. In our optimal setting, the number of iterations for a delayed update is 1, as compared to $O(p)$ where p is the cluster size in the conventional asynchronous parameter server training [14, 30, 1]. Importantly, our framework still enjoys the advantage of an asynchronous approach – interleaving of training iterations to reduce end-to-end runtime. Also, different from the parameter server architecture, we don't congest the head node. Instead, in our case, every worker is only responsible for aggregating part of the gradients in a balanced manner such that communication and aggregate operation time are much more scalable.
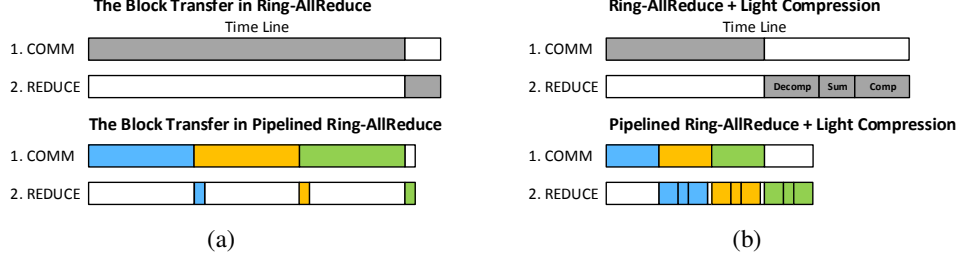
Figure 3: Pipelining within AllReduce: (a) block transfer in native Ring-AllReduce and pipelined Ring-AllReduce, and (b) block transfer with light-weight compression.

More formally, we outline the algorithmic structure of our implementation for each worker in Alg. 1. To be specific, each worker has two threads: one for computation and one for communication, where the former thread consumes the aggregated gradient of the K-th last iteration and generates the local gradient to be communicated, and the latter thread exchanges the local gradient and buffers the aggregated results to be consumed by the former thread.

3.2 Compression in Pipe-SGD

To further reduce the communication time we integrate lossy compression into our decentralized Pipe-SGD framework. Unlike the conventional parameter server or recent decentralized framework transferring parameters over the network [8, 6, 26, 27, 35, 15, 14, 30, 29], our approach communicates only gradients and we justified empirically that gradients are much more tolerant to lossy compression than the model parameters. This seems intuitive since reducing the precision of parameters in every iteration harms the final precision of the trained model directly.

Importantly, as mentioned in Sec. 3.1, compressing the communication overhead contributes to the optimal setting of Pipe-SGD. Once Pipe-SGD is completely computation bound, linear speedups of end-to-end training time can be realized as the cluster size increases. Analytically, we show this observation by deriving the scaling efficiency using the timing model given in Eq. (4). Assume that: 1) the singe-node training takes T_{single} iterations to complete with an execution time of l_{single} taken by each iteration; 2) given a Pipe-SGD cluster with p workers we use the same batch size on each worker as the single-node [12]; 3) the single node and Pipe-SGD train the same epochs on the dataset. From 2) and 3), we find that the total number of iterations required for Pipe-SGD is T_{single}/p, because Pipe-SGD has a p times larger batch size while still training the same number of samples. From this we obtain the scaling efficiency SE of Pipe-SGD via

$$\text{SE} = \frac{\text{Actual Speedup}}{\text{Ideal Speedup}} = \frac{\frac{l_{single} \cdot T_{single}}{l_{total\_pipe}}}{p} = \frac{\frac{l_{single} \cdot T_{single}}{\max(l_{up}+l_{comp}, l_{comm}) \cdot \frac{T_{single}}{p}}}{p} = \frac{l_{up}+l_{comp}}{\max(l_{up}+l_{comp}, l_{comm})}. \quad (7)$$

Thus, we showed that once our system becomes compute bound with compressed communication, Pipe-SGD can achieve linear speedup as the cluster scales, *i.e.*, $SE = 1$.

To maintain applicability of Ring-AllReduce, we choose two simple compression approaches: truncation and scalar quantization. Truncation drops the less significant mantissa bits of floating-point values for each gradient. The scalar quantization discretizes each gradient value into an integer of limited bits, with a quantization range determined by the maximal element of a gradient vector. Due to their simplicity, we easily parallelize those compression approaches to minimize overhead.

Note that compression itself can be compute-heavy and the introduced computation overhead can outweigh the benefit of compressed communication. Particularly when considering that AllReduce based communication performs multiple steps to transfer and reduce the data (see Fig. 2 (c)), requiring repeated invocation of compression and decompression, *i.e.*, for each "transmit-and-reduce" step, with an invocation complexity linear in cluster size. Therefore, many proposed complex compression techniques [42, 45, 11, 4, 49, 32] often fail in the communication-optimal AllReduce setting, resulting in longer wallclock time. For these reasons, compression embedded inside AllReduce must be light, fast and easy to parallelize, such as a floating-point truncation or our element-wise quantization.

Indeed, *pipelining within AllReduce* can help alleviate the heavy overhead of complex compression. However, its benefit might still be limited. Instead of pipelining of training iterations as in Pipe-SGD, *pipelining within AllReduce* interleaves the gradient communication and reduction within each AllReduce process, as illustrated in Fig. 3 (a). Since the communication time is often larger than the reduction time, the latter can be hidden by the former. Once compression is used (as in Fig. 3 (b)), the two stage pipeline becomes (decompression, sum, compression) and (compressed communication) such that light compression overhead can be masked completely. Although complex compression may also benefit from the pipelined Allreduce, the improvement is limited because the time spent by complex compression often outweighs the communication time. For example, we implemented [49] within the pipelined Allreduce and found that the compression overhead is $1.6 \sim 2.3 \times$ the uncompressed communication time and $25.6 \sim 36.8 \times$ the compressed communication time for the benchmarks in Sec. 4, in which case the heavy overhead cannot be masked. Complete masking requires the compression overhead to be smaller than the compressed communication. In the remainder, we only consider light compressions (truncation/quantization) with native AllReduce.

3.3 Convergence

To prove the convergence of Pipe-SGD we adapt the derivation from parameter-server based asynchronous training [14, 22]. We can show that the convergence rate of Pipe-SGD for convex objectives via SGD is $8FL\sqrt{\frac{K}{T}}$, where $K = 2$, F and L are constants for gradient distance and Lipschitz continuity, respectively. We can also show the convergence of Pipe-SGD for strongly convex functions, and find a rate of $O(\frac{\log T}{T})$ for gradient descent. These rates are consistent with [14, 22]. Due to the page limit we defer details to the supplementary material.

4 Experimental Evaluation

In this section, we demonstrate the efficacy of our approach on four benchmarks using three datasets: MNIST [25], CIFAR100 [20] and ImageNet [41]. We briefly review characteristics of those datasets before discussing metrics and setup, and finally presenting experimental results and analysis.

Datasets and Deep Net Architecture

- `MNIST`: The MNIST dataset consists of 60,000 training and 10,000 test images, each showing one of ten possible digits. The images are of size 28×28 pixels with digits located at the center of the images. We use a classical 3-layer perceptron, MNIST-MLP, with both hidden layers being 500-dimensional and with a global batch size of 100.

- `ImageNet`: For our experiments we use 1,281,167 training and 50,000 validation examples from the ImageNet challenge. Each example comprises a color image of 256×256 pixels and belongs to one of 1000 classes. We use the classical AlexNet [21] and ResNet [13], both with a global batch size of 256.

- `CIFAR100`: The CIFAR100 dataset is composed of 50,000 training and 10,000 test examples with 100 classes. The simple AlexNet-style CIFAR100 architecture in [31] is used for benchmarking this datasets. It consists of 3 convolutional layers and 2 fully connected layers followed by a softmax layer. The detailed parameters are available in [31]. Importantly, we adapt this 5 layer CIFAR100-CNN into a convex optimization benchmark, CIFAR100-Convex, to match our proof of convergence. The convexity is achieved by training only the last fully connected layer while fixing the parameters of all previous layers.

Metrics and Setup

We measure the wall-clock time of end-to-end training, *i.e.*, the same number of iterations for different settings. For each benchmark, we evaluate the timing model we proposed using end-to-end train time and detailed timing breakdowns. We plot the test/validation accuracy over training time to evaluate the actual convergence. Also, final top-1 accuracies on the test/validation set are reported. For the setup, we use a cluster of four nodes, each of which consists of a Titan XP GPU [39] and a Xeon CPU E5-2640 [16]. We employ an additional node as the parameter server to support the conventional centralized design. All nodes are connected by 10Gb Ethernet. We implement a distributed training framework in C++ using CUDA 8.0 [38], MKL 2018 [17], and OpenMPI 2.0 [40], which supports the parameter-server and Pipe-SGD approach.
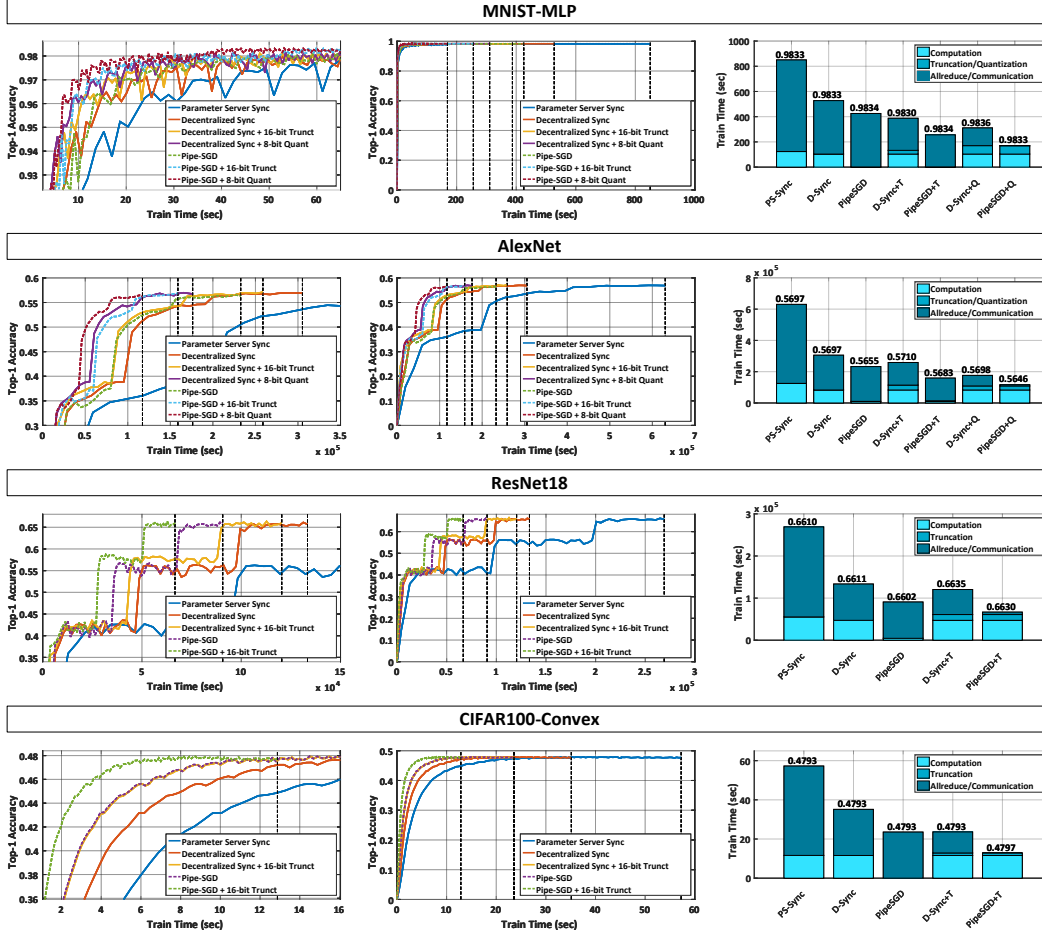
Figure 4: **Experimental results:** Each row shows different benchmarks. The left two columns show convergence via test/validation accuracy *vs.* wallclock training time, where the first column is an inset of the second one. The right most column shows the detailed timing breakdown of end-to-end training. Note that the final top-1 accuracies on test/validation set are labeled on top of the bars.

Results and Analysis

We evaluate the performance of three different frameworks: parameter server with synchronous SGD (PS-Sync), decentralized synchronous SGD (D-Sync), and Pipe-SGD. Our compression schemes, *i.e.*, 16-bit truncation (T) and 8-bit quantization (Q), are also applied to AllReduce communication in D-Sync and Pipe-SGD. Evaluation results are summarized in Fig. 4 where the first two columns show the convergence performances and the third column shows detailed timing breakdowns with final accuracies labeled.

Convergence: From Fig. 4, we observe: decentralized approaches, *i.e.*, D-Sync and Pipe-SGD, converge much faster than the parameter server even without compression, and Pipe-SGD shows the fastest convergence among these frameworks, especially when compression is applied. For example, the convergence curve of the CIFAR100-Convex shows that D-Sync is around 40% faster than PS-Sync and Pipe-SGD is another 37% faster than D-Sync. The advantage of Pipe-SGD is further boosted by compression, *i.e.*, truncation in this case, and demonstrates an additional 46% faster convergence than the D-Sync with the same compression scheme. Therefore Pipe-SGD prevails with a great margin.

Timing Breakdown: From Fig. 4, the comparison between centralized and decentralized designs shows 50% reduction in uncompressed communication time, thus justifying the efficacy of balanced communication. Once compression is applied, further reduction is observed. However, the actual improvement in D-Sync is not ideal considering compression factors of $2\times$ for truncation and $4\times$ for

quantization, because the compression overhead is paid at the critical path of D-Sync. In contrast, our Pipe-SGD can hide this overhead together with computation due to the pipelined nature, as shown in "D-Sync+T" *vs.* "PipeSGD +T" in the MNIST benchmark. As communication is further reduced by quantization, the system becomes compute bound and Pipe-SGD switches to hide the communication instead, thus reaching the optimal setting of Pipe-SGD. This optimum can also be achieved via the simplest truncation for models with less dominant communication time, *e.g.*, ResNet18 and CIFAR100-Convex. As a result, our approach achieves a speedup of $2.0 \sim 3.2\times$ compared to D-Sync and $4.0 \sim 5.4\times$ compared to PS-Sync for these benchmarks. Note that these speedups are based on the comparison between different approaches in the same cluster without scaling the cluster size.

Accuracy: Considering the potential drawback of the 1-iteration staled update and lossy compression in Pipe-SGD, we also evaluate the final test/validation accuracies after end-to-end training, as shown in Fig. 4. Interestingly, in our optimal settings "PipeSGD +T/Q," we find that only AlexNet drops top-1 accuracy by 0.005 compared to baseline D-Sync while all other benchmarks show slightly improved accuracies. To obtain the best accuracies for the two large non-convex models such as AlexNet and ResNet, we employ a similiar warm-up scheme as in [32], *i.e.*, we don't turn on the pipelined training until the 5-th epoch, before which we still stick to D-Sync training to avoid the undesirable gradient change in the initial stage. Since the warm-up period is marginal compared to total number of epochs, the system performance benefits from Pipe-SGD most of the time. Note that for smaller models, especially convex ones (*e.g.*, CIFAR100-Convex), no warm-up is required.

5   Related Work

Li *et al.* [26, 27] proposed a parameter server framework for distributed learning and a few approaches to reduce the cost of communication among compute nodes, such as exchanging only nonzero parameter values, local caching of index list, and random skip of messages to be transmitted. Abadi *et al.* [1] also proposed a centralized framework, TensorFlow, which incorporates model and data parallelism for training deep nets. Both works support the asynchronous setting to improve communication efficiency but without controlling the staleness of the gradient update. Ho *et al.* [14] proposed SSP, another centralized asynchronous framework but with bounded staleness for gradients. The key idea of SSP: 1) each worker has its own iteration index, 2) the slowest and fastest worker must be within S iterations, otherwise, the fastest worker is forced to wait until the slowest worker catches up. However, this bound S applies to the iteration drift among workers instead of directly on the stale updates of the parameter server. As a result, each worker within the bound can still commit their updates to the server asynchronously, making the last gradient update staled heavily. In the worst case, the staleness is linear in the cluster size.

Recently, Lin *et al.* [32] employed Allreduce as the gradient aggregation method in their synchronous framework, but little is reported regarding wallclock time benefits, especially considering that the full synchronous design suffers from the longest execution time among all workers. Besides, Lian *et al.* proposed AD-PSGD [30] which parallelizes the SGD process over decentralized workers in a completely asynchronous fashion. Workers run completely independently, and only communicate with a set of neighboring nodes to exchange trained weights, *i.e.*, neighboring models are averaged to replace each worker's local model in each iteration. However, this approach suffers from uncontrolled staleness, which in practice increases with cluster size and the time taken by each iteration. In addition, such a communication method requires each worker to act as the center node of a local graph, which results in a local communication bottleneck. As a result, each worker suffers from long iteration time which further increases the staleness of weight updates. Although Lian *et al.* [30] compared their framework with the full synchronous design in wall-clock time, the performance turns out to be similar when network speeds are roughly equal.

6   Conclusion

We developed a rigorous timing model for distributed deep net training which takes into account network latency, model size, byte transfer time, *etc*. Based on our timing model and realistic resource assumptions, *e.g.*, limited network bandwidth, we assessed scalability and developed Pipe-SGD, a pipelined training framework which is able to mask the faster of computation or communication time. We showed efficacy of the proposed method on a four-node GPU cluster connected with 10Gb links. Rigorously assessing wall-clock time for Pipe-SGD, we are able to achieve improvements of up to $5.4\times$ compared to conventional approaches.

Acknowledgement

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *NIPS*, 2017.

[3] Y. Bengio, A. Courville, and P. Vincent. Representation Learning: A Review and New Perspectives. *PAMI*, 2013.

[4] C.-Y. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, and K. Gopalakrishnan. AdaComp : Adaptive Residual Gradient Compression for Data-Parallel Distributed Training. In *AAAI*, 2018.

[5] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs. In *ICLR*, 2015.

[6] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[7] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server. In *EuroSys*, 2016.

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

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[10] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. V. Essen. Aluminum: An Asynchronous, GPU-Aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC Systems. In *MLHPC*, 2018.

[11] N. Dryden, T. Moon, S. A. Jacobs, and B. V. Essen. Communication Quantization for Data-Parallel Training of Deep Neural Networks. In *MLHPC*, 2016.

[12] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. In *CVPR*, 2017.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

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

[15] F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer. FireCaffe: Near-linear Acceleration of Deep Neural Network Training on Compute Clusters. In *CVPR*, 2016.

[16] Intel Corporation. Xeon CPU E5, https://www.intel.com/content/www/us/en/products/processors/xeon/e5-processors.html, 2017.

[17] Intel Corporation. Intel Math Kernel Library, https://software.intel.com/en-us/mkl, 2018.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *ACM SIGOP*, 2007.

[19] H. Kim, J. Park, J. Jang, and S. Yoon. Deepspark: A spark-based distributed deep learning framework for commodity clusters. *arXiv:1602.08191 [cs]*, 2016.

[20] A. Krizhevsky. Learning Multiple Layers of Features from Tiny Images, 2009.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

[22] J. Langford, A. J. Smola, and M. Zinkevich. Slow Learners are Fast. In *NIPS*, 2009.

[23] G. Larsson, M. Maire, and G. Shakhnarovich. FractalNet: Ultra-Deep Neural Networks without Residuals. In *https://arxiv.org/abs/1605.07648*, 2016.

[24] Y. LeCun, Y. Bengio, and G. E. Hinton. Deep learning. *Nature*, 2015.

[25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *IEEE*, 1998.

[26] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, 2014.

[27] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. Communication Efficient Distributed Machine Learning with the Parameter Server. In *NIPS*, 2014.

[28] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A.G. Schwing, H. Esmaeilzadeh, and N.S. Kim. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *MICRO*, 2018.

[29] X. Lian, C. Zhang, H. Zhang, C. Hsieh, W. Zhang, and J. Liu. Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent. In *NIPS*, 2017.

[30] X. Lian, W. Zhang, C. Zhang, and J. Liu. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *arXiv:1710.06952v3*, 2018.

[31] R. Liao, A. Schwing, R. Zemel, and R. Urtasun. Learning Deep Parsimonious Representations. In *NIPS*, 2016.

[32] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *ICLR*, 2018.

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*, 2013.

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level Control through Deep Reinforcement Learning. *Nature*, 2015.

[35] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. SparkNet: Training Deep Networks in Spark. In *ICLR*, 2016.

[36] D. G. Murray, R. Isaacs F. McSherry, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *SOSP*, 2013.

[37] Nvidia. GPU-Based Deep Learning Inference: A Performance and Power Analysis. In *Whitepaper*, 2015.

[38] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010.

[39] NVIDIA Corporation. TITAN Xp, https://www.nvidia.com/en-us/design-visualization/products/titan-xp/, 2017.

[40] OpenMPI Community. OpenMPI: A High Performance Message Passing Library, https://www.open-mpi.org/, 2017.

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

[42] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-Bit Stochastic Gradient Descent and Its Application to Data-Parallel Distributed Training of Speech DNNs. In *INTERSPEECH*, 2014.

[43] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Misra, and H. Esmaeilzadeh. From High-Level Deep Neural Models to FPGAs. In *MICRO*, 2016.

[44] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 2016.

[45] N. Strom. Scalable Distributed DNN Training using Commodity GPU Cloud Computing. In *INTERSPEECH*, 2015.

[46] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *IJHPCA*, 2005.

[47] Q. Wang, Y. Li, and P. Li. Liquid State Machine based Pattern Recognition on FPGA with Firing-Activity Dependent Power Gating and Approximate Computing. In *ISCAS*, 2016.

[48] Q. Wang, Y. Li, B. Shao, S. Dey, and Peng Li. Energy Efficient Parallel Neuromorphic Architectures with Approximate Arithmetic on FPGA. *Neurocomputing*, 2017.

[49] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *NIPS*, 2017.

[50] M. Zaharia, M. Chowdhury, Michael J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[51] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*, 2015.

[52] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva. Learning Deep Features for Scene Recognition using Places Database. In *NIPS*, 2014.