Herring: Rethinking the Parameter Server at Scale for the Cloud

Indu Thangakrishnan, Derya Cavdar, Can Karakus,
Piyush Ghai, Yauheni Selivonchyk, Cory Pruce
Amazon Web Services
{thangakr, dcavdar, cakarak, ghaipiyu, yauheni, cpruce}@amazon.com

***Abstract*—Training large deep neural networks is time-consuming and may take days or even weeks to complete. Although parameter-server-based approaches were initially popular in distributed training, scalability issues led the field to move towards all-reduce-based approaches. Recent developments in cloud networking technologies, however, such as the Elastic Fabric Adapter (EFA) and Scalable Reliable Datagram (SRD), motivate a re-thinking of the parameter-server approach to address its fundamental inefficiencies. To this end, we introduce a novel communication library, *Herring*, which is designed to alleviate the performance bottlenecks in parameter-server-based training. We show that gradient reduction with Herring is twice as fast as all-reduce-based methods. We further demonstrate that training deep learning models like $\text{BERT}_{\text{large}}$ using Herring outperforms all-reduce-based training, achieving 85% scaling efficiency on large clusters with up to 2048 NVIDIA V100 GPUs without accuracy drop.**

***Index Terms*—Deep Learning, Data Parallel Training, Scalability, Distributed Parallel Computing**

I. Introduction

As machine learning datasets and state-of-the-art deep neural networks grow in size, there is a growing interest in efficient distributed training techniques. Some of the recent state-of-the-art NLP models like BERT [1] and RoBERTa [2] have hundreds of millions of parameters and can take days, even weeks to train, unless large GPU clusters are used. However, the use of such large GPU clusters come with significant systems challenges, mainly centered around designing how the GPUs communicate with each other. An inefficient design can lead to extremely poor scaling efficiency for large models, wasting computational resources.

In the early days of deep learning, parameter-server-based distributed training was the dominant paradigm [3]–[5]. In this approach, the model parameters are partitioned across a number of parameter servers. The worker nodes each sample a batch of data, fetch the latest version of model parameters from the parameter servers, compute gradients on the sampled batch of data, and push the result to the parameter servers. This process can be performed synchronously, where after each step each worker node waits for updates from all worker nodes to reach the servers, or asynchronously, where the worker nodes fetch parameters and push gradients independently from each other. A model whose parameters take M bytes requires the worker nodes to transmit and receive M bytes at every training step, and requires only two hops of communication, regardless of the size of the cluster.

While parameter servers were popular in the early days of deep learning, allreduce based distributed training methods have gained more popularity recently, especially led by Horovod [6]. Allreduce based distributed training does not require separate parameter servers. At the end of each iteration, the workers directly communicate with each other and average the gradients using MPI style allreduce algorithms. While there are a number of different allreduce algorithms, one of the most popular implementation has been the ring algorithm owing to its optimal bandwidth use. In this algorithm, worker i receives gradients from worker $i-1$, adds its gradient and passes the result to worker $i+1$ (with wrap-around). After a single round, the summed gradients traverse the ring once again to distribute the result to all workers[1].

Note that the ring allreduce technique requires a total of $2(n-1)$ communication hops for n workers, and transmitting and receiving $2M$ bytes for a model size M, a substantial increase compared to 2 hops and M bytes in the case of parameter server. There are other variants of allreduce algorithms aiming reduce the number of communication hops [7]–[9]. Nvidia's double tree allreduce algorithm offers reduced latency by using logarithmic number of hops [10]. Although these approaches reduce the latency incurred, most of the deep learning workloads suffer from inefficient use of bandwidth.

Despite this increased communication load, allreduce-based techniques tend to outperform parameter server approaches. This is because the existing parameter server implementations fail to use the available network bandwidth efficiently because of the increased number of parallel data streams between nodes with different streams transferring data at different rate and suboptimal use of network bandwidth on the servers because not all servers are active at all times during the backward pass.

Recently, Amazon Web Services (AWS) announced the availability of the Elastic Fabric Adapter (EFA) [11], which uses OS bypass and the Scalable Reliable Datagram. EFA takes advantage of cloud resources and characteristics like the multi-path backbone to avoid the performance bottlenecks that TCP suffers. There are hundreds of possible paths between two AWS instances. EFA splits the buffer to be sent into multiple ethernet jumbo frames and sprays them across multiple paths to the destination. It uses equal-cost multi-path routing

[1]In practice, these two rounds are implemented as the reduce-scatter and allgather steps.

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

(ECMP) to find the most effective paths for messages and messages can be delieverd out-of-order. Notably, this enables an EFA-enabled compute node to sustain a large number of parallel communication flows, where the bandwidth is equally shared between each flow.

Motivated by EFA, we revisit the parameter server approach to take advantage of its lower communication requirements, by addressing its specific shortcomings. We develop a new library, called *Herring*[2], which combines EFA with a novel parameter sharding technique that makes better use of the available network bandwidth.

We empirically compare the performance of *Herring* with parameter server and allreduce-based approaches over $\text{BERT}_{\text{large}}$ training. Using *Herring*, we train $\text{BERT}_{\text{large}}$ using 2048 NVIDIA V100 GPUs (256 EC2 `p3dn.24xlarge` instances) in 62 minutes. Herring achieves a scaling efficiency of 85% across 2048 GPUs. To our knowledge, this is the highest scaling efficiency achieved at this scale to train $\text{BERT}_{\text{large}}$ in the cloud.

Our main contributions are as follows:

- We demonstrate that parameter servers scale better than allreduce-based methods for distributed training even on very large clusters.
- We propose a novel parameter sharding method that allows more optimal utilization of network bandwidth.
- We develop *Herring*, a highly scalable distributed data-parallel training library that works with TensorFlow, PyTorch and MXNet.
- We demonstrate the scaling behavior of Herring with respect to model and cluster size, and compare to NCCL.
- Combining *Herring* with other graph-level optimizations, we are able to train $\text{BERT}_{\text{large}}$ using the largest cluster in the cloud with the highest recorded scaling efficiency at that scale.

II. Background and Related Work

A. Parameter Servers

Distributed training in the early days on deep learning focused a lot on parameter servers [3]–[5], [12]–[14]. Workers synchronize gradients with parameter servers synchronously or asynchronously. While asynchronous training, as in [3], is faster because of absence of synchronization between workers, synchronous training is used more often because of higher accuracy and easier reproducibility.

Some of the inherent benefits of parameter-server-based approach are:

- **Two-hop communication**: Irrespective of the number of workers, the gradient aggregation can be done in two communication hops, which adds robustness against latency.
- **Asynchronous parameter synchronization**: The parameter server architecture allows for asynchronous training [15], [16] for added straggler resilience, which can be suitable for some workloads.
- **Reduced bandwidth usage**: In a parameter server based distributed training of a model with M parameters, after processing every minibatch, workers send and receive M bytes from the server. This is much less than the almost $2M$ bytes that is sent and received by each node in allreduce-based training.

While parameter servers transfer less data on the network compared to allreduce-based methods, main drawback of parameter-server-base approach is the requirement of additional computation resources dedicated for gradient averaging. Although deep learning workloads require powerful GPU instances, cheap CPU instances can be used for parameter-servers. Hence cost of parameter-servers are comparably low. In addition to increased computational resource requirement, existing parameter-server implementations possess a number of key performance bottlenecks, which limits their scalability:

- **Unequal bandwidth allocation**: As the number of workers grows, a single parameter server needs to simultaneously communicate over an increasing number of TCP flows. Due to TCP's head-of-line blocking, this results in the bandwidth being shared increasingly inequitably between the TCP flows, resulting in straggler effects.
- **Bursty communication**: Parameter servers treat variable as an atomic unit and do not partition them across multiple servers. Since gradients for different layers become available sequentially during backpropagation, at any given instant only a subset of the parameter servers will be communicating with workers. This results in an under-utilization of the aggregate bandwidth available to the parameter servers.
- **Network congestion**: When the number of parameter-servers is small, each parameter-server will communicate with many number of workers. In that case, network connection between parameter-server and workers can become a bottleneck. In order to avoid this type of network bottlenecks we use large number of cheap CPU instances as parameter-servers in our experiments.

More recently, BytePS [17] attempted to address the limitations of the parameter server, with some success. However, as we will demonstrate, relying on TCP on ethernet and suboptimal parameter sharding across the servers limits the scalability of BytePS.

B. Allreduce-based approaches

Motivated by the limitations of the parameter server, Horovod [6] introduced allreduce-based distributed training. Horovod was inspired by the bandwidth-optimal ring-allreduce algorithms proposed in [18], [19]. This led to a surge of interest in allreduce algorithms based on other topologies as well, including hierarchical allreduce, two-tree binary allreduce [10], butterfly allreduce [9] and torus allreduce [20], [21]. Although these variations of allreduce-based algorithms improves performance of latency-bound workloads, most of

[2]A reference to the large computing clusters *Herring* is useful for, in light of the fact that herrings tend to swim in large shoals.

the deep learning workloads limited by bandwidth [22]. Hence designing communication libraries for efficient bandwidth usage is crucial for deep learning workloads. Further intra-node [23], and cross-node [24], [25] optimizations enabled allreduce techniques to reach very large scale for certain problems. Specifically for $\text{BERT}_{\text{large}}$, the work in [26] achieved $\text{BERT}_{\text{large}}$training in 76 minutes.

Despite the successes of allreduce-based training, higher number of network hops and larger data transfer raises the question that parameter server approach might be able to perform faster if its specific inefficiencies are addressed.

C. Gradient compression

A gradient compression literature developed in parallel, to perform efficient distributed training. Some earlier works along this direction [27]–[29] considered quantization of gradients. Another line of work [30]–[32] focused on sparsification of gradients, while [31] proposed an error compensation mechanism to reduce compression error. Infrequent updates, as a communication reduction method has been studied in [33], and the work in [34] combined all three approaches.

One advantage of parameter server approach with respect to allreduce is that it enables greater flexibility in gradient compression options. This is because the repeated summation in every step of allreduce causes the intermediate results the go out of the allowed quantization levels for quantization-based approaches. Similarly, if a sparsification technique is used, with every summation step, the sparsity is gradually lost. The two-hop communication pattern of parameter server is thus better suited to incorporate such compression techniques.

III. Herring

We introduce Herring - a parameter-server based distributed training library that is optimized for large scale distributed training. Herring uses EFA and *balanced fusion buffer* to optimally use the total bandwidth available across all nodes in the cluster. Herring reduces gradients hierarchically, reducing them inside the node first and then reducing across nodes. This enables more efficient use of PCIe bandwidth in the node and helps keep the gradient averaging related burden on GPU low.

Herring outperforms current allreduce-based and parameter-server-based approaches by combining inherent advantages of parameter server approach, in particular, lower bandwidth utilization, constant two-hop communication with careful fusion buffer implementation and efficient utilization of EFA. Herring supports all three major deep learning frameworks - TensorFlow, PyTorch and MxNet with only few lines of code change.

1) Balanced fusion buffers (BFB): In parameter server based training, model parameters are sharded across multiple parameter servers. Parameter servers treat variables as atomic unit and place each variable on one server. Since gradients become available sequentially during the backward pass, at any given instant there is imbalance in the volume of data being sent and received from different servers. Some servers will be receiving and sending more data, some less and some none. This problem becomes worse as the number of parameter servers increase.

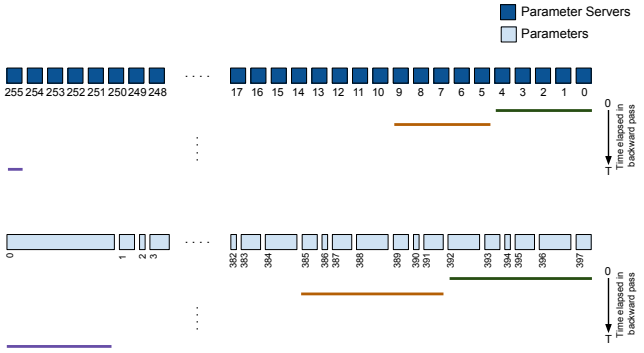


Fig. 1: Parameter servers treat trainable variables as atomic units and place them on one of the parameter servers. At different time periods during the backward pass, gradients corresponding to different variables are produced and workers communicate with the servers those variables reside in. As a result each server is active only during a fraction of the duration of each backward pass. This figure shows three different time periods during the backward pass using different colors. It shows the variables for which gradients are available to be averaged and the servers those variables reside in.

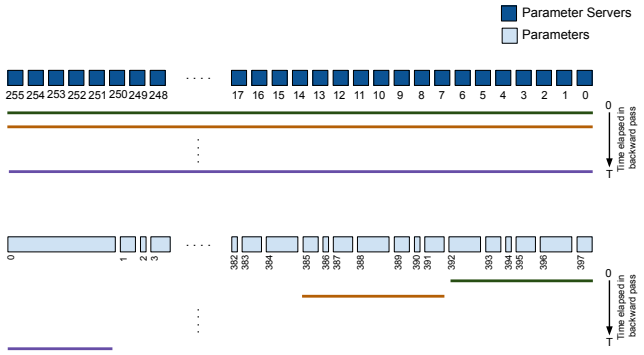


Fig. 2: Instead of sharding variables across parameter servers, Herring shards fusion buffers that are usually a concatenation of a number of gradients across all parameter servers. Since each fusion buffer is distributed equally across all parameter servers, workers are able to use all of the servers to average each fusion buffer.

To illustrate the problem of treating variables as atomic units placed on a single parameter server, consider a large model like $\text{BERT}_{\text{large}}$ being trained on a large cluster with 256 worker nodes and 256 parameter servers. $\text{BERT}_{\text{large}}$ contains 398 trainable variables. If 398 variables were sharded over 256 parameter servers, some servers will own only one parameter. During the backward pass each server will be active only when one or two of the total of 398 gradients are being reduced. The server responsible for the last variable produced during the backward pass for example will be inactive during the entire backward pass and will start to receive gradients only after

the entire backward pass is complete.

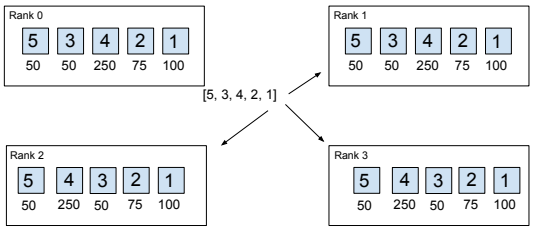


Fig. 3: In the first iteration, each rank observes gradients being produced in slightly different order. After the first iteration the order in which gradients we produced in the first worker is broadcasted to all workers.

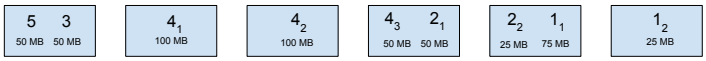


Fig. 4: Variables are arranged in the fusion buffer based on the order in which they were produced during the first iteration in the first worker.

The problem becomes worse if one of the last gradients produced during the backward pass also happens to be a large tensor. For example, in the case of $\text{BERT}_{\text{large}}$, the last gradient produced corresponds to the embedding layer, consists of 31 million parameters and takes 60 MB in fp16 representation. Figure 1 illustrates this problem.

Herring addresses these problems by introducing *balanced fusion buffers*. Balanced fusion buffer is a buffer in the GPU that holds the gradients until the size of the buffer exceeds a threshold. In a setup with N parameter servers, when threshold exceeds, the BFB is copied to CPU memory, BFB is then sharded into N parts and ith part is sent to ith parameter server. Each server receives exactly the same number of bytes from a balanced fusion buffer. ith server receives ith partition of the BFB from all workers, sums them up and sends the results back to all workers. Since all the servers participate equally in averaging each balanced fusion buffer, server bandwidth is efficiently utilized. Figure 2 illustrates how Herring uses Balanced Fusion Buffer.

Each iteration of the training process involves averaging multiple fusion buffers, each buffer containing gradients corresponding to multiple variables. Gradients for a single variable could be split across multiple balanced fusion buffers. Total size of all fusion buffers is equal to the total size of all trainable variables in the model. All BFBs are of exactly the same size except for the last buffer which could be smaller. Figure 4 illustrates this.

In the next section, we describe how we make sure fusion buffers in different GPUs contain the same variables in the same offset so that server can average the same parameters from multiple workers.

2) Balanced Fusion Buffer without Negotiation: Note that for balanced fusion buffers to work, all workers must place gradients corresponding to the same variable in exactly same offset in exactly the same BFB. The workers also need to know the order in which the BFBs need to be sent to the server. Herring does this by assigning an order index to each parameter based on the order in which the corresponding gradient was produced during the first iteration in the first worker. The first worker then broadcasts this order to every other worker as shown in Figure 3.

Each worker then computes the position of each parameter in the fusion buffer depending on the broadcasted order. A variable$i$ is placed before variable$j$ in the backward pass if gradient corresponding to variable$i$ was received before the gradient corresponding to variable$j$ in the first iteration in the first worker. This gradient placement is illustrated in Figure 4.

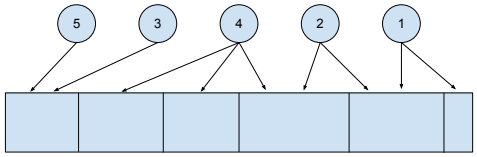


Fig. 5: When parameters in subsequent iterations arrive in the same order as the first iteration in worker 0 gradients are copied sequentially into the balanced fusion buffer.

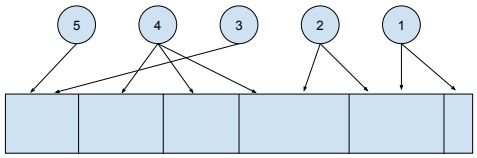


Fig. 6: Sometimes gradients could arrive in slightly different order compared to the first iteration in first worker. In such cases, gradients are still placed in the same BFB offset as they would have been placed if the gradients were produced in the same order.

In every iteration after the first iteration, when a gradient is computed, Herring places it in the precomputed location in the predetermined BFB as shown in Figure 5. When all gradients that belong to a BFB is available, Herring kicks off the process to average that BFB. While computing the backward pass, gradients are not generated in the same order for every iteration on every GPU. So, a gradient sometimes could be placed into a BFB that is not next-in-line for averaging as shown in Figure 6.

We are not aware of a parameter server implementation that fuses gradients into larger buffer but Horovod fuses gradients into a larger buffer to reduce the number of alllreduce operations. Horovod does this by letting the workers negotiate and find out which set of gradients is ready to be allreduced across all workers. Horovod does not assume static communication schedule as a consequence it has a negotiation phase. Herring avoids such negotiation to avoid the cost involved in negotiation as shown in Figure 7. Apart from the direct

delay due to negotiation, there is also an indirect delay because gradients cannot be placed into the fusion buffer when they are ready. Copy to fusion buffer must be delayed until after negotiation is done. In certain conditions when we can not hold on to the original buffer too long without without incurring performance penalty, this will result in double copy.

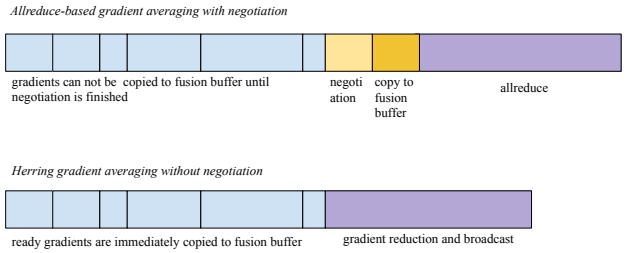


Fig. 7: Gradient averaging without negotiation

3) Proxy Parameter Servers: While P3dn.24xl instances have 100 gbps connectivity, a single endpoint can only use around 70 Gbps of that bandwidth. To utilize all the 100 Gbps, it it required to use more processes. A simple way to do this is to run two server processes per node. But those two processes will end up owning two different set of parameters and each worker will have to communicate with both those processes resulting in 2N flows to each server where N is the number of workers. While EFA is better than default network adapter ENA(Elastic Network Adapter) [35] at handling large number of parallel connections, it is still better to keep the number of parallel flows low especially for large clusters.

Herring increase the number of processes without increasing the number of flows by introducing proxy parameter servers as shown in Figure 8. Each server instance runs 8 proxy parameter servers. Each of these proxy parameter servers communicate with approximately $N/8$ workers where N is the total number of workers. The gradients are read directly into a shared memory shared between all proxy servers and the parameter server. Parameter Server accesses gradients from all workers through shared memory, computes the average and stores results back into the shared memory. Proxy servers then push the gradients back to workers.

A. Herring Workflow

Herring set up consists of same number of parameter server and worker nodes where both workers and servers have same network bandwidth.

First iteration:

As gradients are computed for variables, each variable is assigned an order index in the first worker. Order index starts with 0 and goes up to $M-1$ where M is the number of trainable variables in the model as shown in Figure 3. After the first iteration, a list of variable indices ordered by the order in which their corresponding gradient got computed during the first pass in $node0$ is broadcasted to all workers. Each worker then assigns these variables to a certain offset inside a certain balanced fusion buffer. Variables are placed

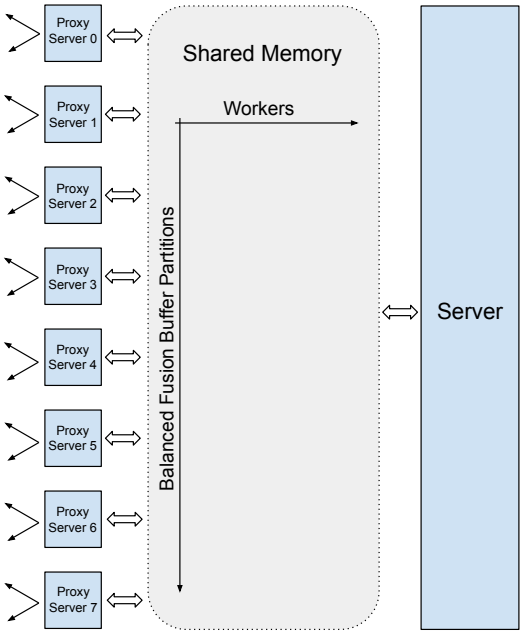


Fig. 8: Herring increases the number of processes in each server node without increasing the number of parallel connections by using proxy parameter servers that only communicate with a subset of workers. This picture shows the processes inside a server node in a cluster with 16 worker nodes and 16 server nodes. Since there are 16 workers, each proxy server process communicates with 2 worker processes in two different worker nodes.

sequentially next to each other in the list of BFBs. ith variable is placed before jth variable if i appears before j in the list broadcasted by the first worker. Each fusion buffer is of the same size except for the last buffer which could be smaller. Size of the balanced fusion buffer is automatically computed based on the cluster size and model size. User can override this using an environment variable.

Subsequent iterations:

Step 1. As gradients become available during the backward pass, they are copied into the balanced fusion buffer (BFB). When the next fusion buffer is ready (all gradients that belong to that BFB are ready), the worker process uses `MPI_Barrier` to wait until the same fusion buffer is ready in other GPUs in the same node. When specific fusion buffer is ready on all GPUs in the same node, `ncclReduceScatter` is used to average the fusion buffer inside the node using the NVLinks. Each GPU gets averaged output for $1/n_{GPU}$ of the fusion buffer where n_{GPU} is the number of GPUs within the node.

Step 2. ReduceScattered fusion buffer is then copied to the CPU memory of the GPU instance using the PCIe link. Since each GPU copies $1/n_{GPU}$ of the buffer, the total bandwidth used for copy is $n_{PCIE} \times bw_{PCIe}$ where n_{PCIE} is equal to

the number of PCIe links and bw_{PCIe} is the bandwidth of each PCIe link (around 12 GBps in EC2 `P3dn.24xlarge`) instances. NCCL uses one PCIe link in the node for device-to-host copy and another link for host-to-device copy which could bottleneck the PCIe link. Using all the PCIe links for device-to-host and host-to-device copies help transfer gradients to CPU memory n_{PCIE} times faster ($n_{PCIE} = 4$ for an EC2 `P3dn.24xlarge` instance).

Step 3. In a server node running $n_{ServerProc}$ server processes, each server process receives msg_{len} bytes at a time from $N/n_ServerProc$ worker process, and places it in a shared memory region. After a part of a BFB is received from all worker into a server node, a different process called NC averages the gradients and signals the server processes. Check Section III-3 for why we use multiple server processes in each server node.

Step 4. Server processes send averaged gradients back to workers. When averaged gradients are received, they are copied to GPUs using all the PCIe links similar to Step 1. At the end of this step, each GPU has averaged gradients for $1/n_{GPU}$th of the fusion buffer.

Step 5. By the time a GPU receives $1/n_{GPU}$ of the updated gradients, other GPUs might still have not received their fraction of the result. Scheduling a `ncclAllGather` immediately can lead to waiting for other GPUs to get their fraction of the results. Further, note that some gradients might still be in the reduce-scatter step, and scheduling an immediate allgather in the CUDA stream would delay the reduce-scatter operation for these gradients, degrading performance. To prevent this, worker processes use `MPI_Barrier` to wait until `ReduceScatter` is done for all BFBs. Once `MPI_Barrier` succeeds, `ncclAllGather` is called and NVLinks are used to share the fractional all-reduce results with all GPUs on the node.

Step 6. Finally, all GPUs have averaged gradients in the fusion buffer. This result is then handed over to the framework (TensorFlow or PyTorch).

IV. Evaluation

In this section, we present empirical performance evaluation of Herring compared to allreduce-based and parameter-server-based distributed training libraries namely Horovod and BytePS.

Horovod is an allreduce-based distributed training library that internally uses NCCL to perform the all-reduce operation. For Horovod experiments we use NCCL double binary tree algorithm [7], which has shown to outperform ring allreduce algorithm on large clusters. We use NCCL built with EFA support.

BytePS is a parameter-server implementation which uses NCCL for intra-node communication and has its own implementation of inter node communication. As of now, BytePS does not support EFA. Although we use the same instance type, BytePS experiments use TCP instead of EFA for inter-node communication.

We first compare Herring and NCCL gradient averaging time for different data and cluster sizes. We then present an analysis of the change in throughput and scaling efficiency with change in cluster size. Finally, we present time to train and scalability analysis of $\text{BERT}_{\text{large}}$ with Horovod and Herring on very large distributed training settings where we achieve 85% scaling efficieny on a cluster with 2048 GPUs.

A. Gradient averaging benchmarks

In this section, we compare time taken to average gradients with Herring and NCCL. For Herring experiments we use one CPU instance for each GPU instance. For example, we use 32 parameter servers for 256 GPU experiments and 64 parameter servers for 512 GPU experiments since each GPU instance contains 8 GPUs.

In Figure 10, we present time spent on averaging gradients for different data length and number of GPUs. Both Herring and NCCL utilize EFA. We experiment with parameter sizes ranging from 100MB to 1.6GB to cover both small and large neural networks. Some examples of model sizes include Resnet50: 25M parameters, $\text{BERT}_{\text{base}}$: 110M, $\text{BERT}_{\text{large}}$ 340M parameters.

Herring is able to average gradients in half the time NCCL takes because each node sends and receives only half as much data compared to NCCL. In MPI style allreduce, Rabenseifner's algorithm is the most commonly used algorithm to all-reduce long messages. The algorithm does all-reduce by doing reduce-scatter followed by all-gather. Time taken to reduce-scatter is $lg(p)\alpha+(p-1)\frac{n}{p}\beta+(p-1)\frac{n}{p}\gamma$. Time taken to all-gather is $lg(p)\alpha+(p-1)\frac{n}{p}\beta$. α is the time taken to transfer the first byte which captures the latency of the network. β is the time taken to transfer subsequent bytes which captures the bandwidth of the network. γ is the time taken to add one byte which captures the compute capacity of the node. Total time taken for all-reduce is $2lg(p)\alpha + 2(p-1)\frac{n}{p}\beta + (p-1)\frac{n}{p}\gamma$

Note that $(p-1)\frac{n}{p}\beta$ is counted twice because it is required once for reduce-scatter and once again for all-gather. Also, $(p-1)\frac{n}{p}$ is $\approx$ n for large clusters. For example when 100 MB is all-reduced across 64 nodes, data transferred is $2*63*100/64 = 196.8$ MB

Herring performs all-reduce by having the workers send gradients to server once, average gradients on the server and receive it back once. A naive implementation would be to perform those operations serially in which case the time taken would be $2\alpha + 2(n-1)\beta + n*p*\gamma$ where $\alpha + (n-1)\beta$ is the time taken to send gradients to server, $n*p*\gamma$ is the time taken to add n bytes of gradients received from p workers and $\alpha+(n-1)\beta$ is the time taken to send averaged gradients back to workers.

Unlike the naive implementation, Herring pipelines the three operations involved in gradient averaging. Gradients are sent to servers in small segments not exceeding the size of an ethernet jumbo frame which is 9000 bytes. As each ethernet jumbo frame is received on the server, it is accumulated with the same segment received from other workers thus far. This accumulation happens asynchronously and does not

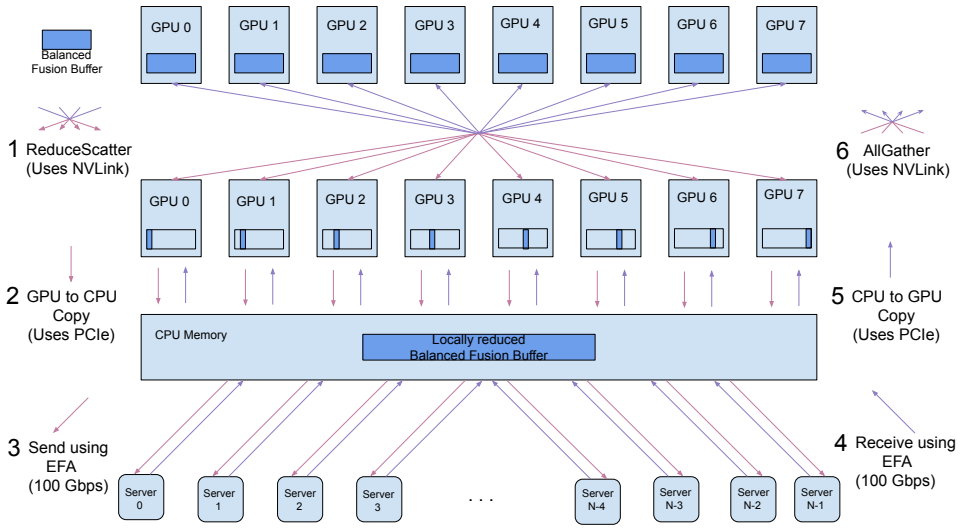
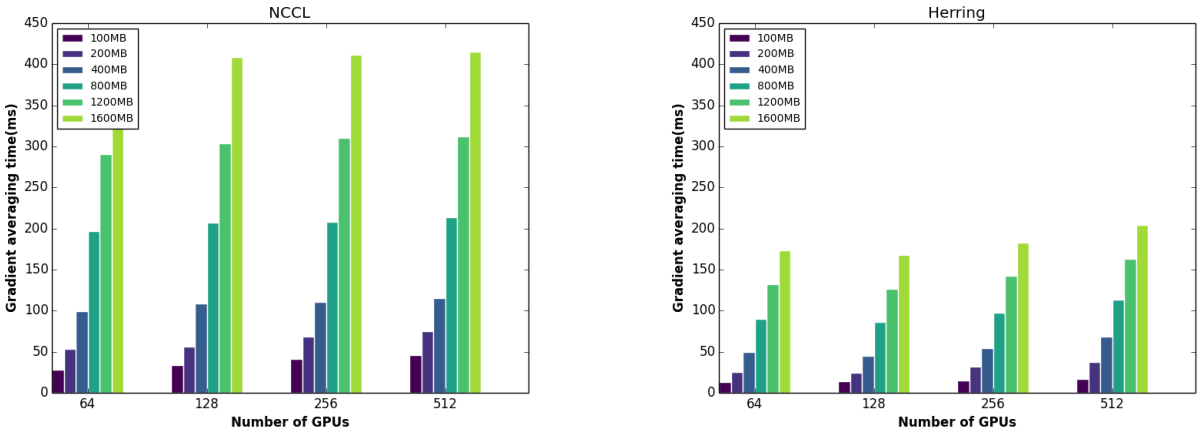


Fig. 9: Herring gradient averaging workflow



(a) Time spent on allreduce with NCCL

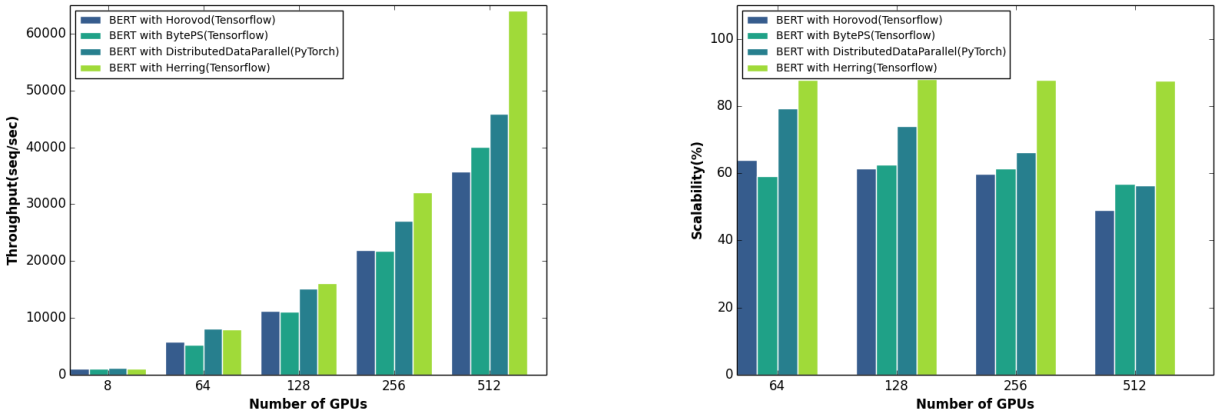
(b) Time spent on gradient averaging with Herring.

Fig. 10: Time spent on gradient averaging with changing number of GPUs and parameter sizes with NCCL and Herring using EFA enabled 100Gbps network.

block receiving or sending further segments. When a segment is received and accumulated from all workers, the result is sent to all workers. The send operation overlaps with receive operation and accumulation operations corresponding to other segments.

Time taken for a herring all-reduce operation is $p\alpha + \gamma + n\beta$. α is the time taken to send one ethernet frame to server. The first segment needs to arrive from all of the p workers before the first response could be sent back to the workers. γ is the time taken to accumulate data received in one ethernet frame onto the same segment received from other workers. γ is counted only once because rest of the accumulation overlaps with sending or receiving gradients. β is the time taken to send one byte of data. β is only counted once (corresponding to gradients being sent back to the workers) because all segments except one corresponding to sending gradients from worker to server overlaps with other operations already counted in the expression.

This speedup over NCCL is observed only with EFA adapter with EFA routing packets through many different paths in the network dynamically switching away from paths that are momentarily slower because of network congestion. We don't see similar speedup in gradient averaging time in TCP network presumably because of imbalanced rate of transfer of packets across multiple TCP streams because of head-of-line blocking in TCP.



(a) Throughput

(b) Scaling efficiency

Fig. 11: Scalability and throughput with Horovod(with EFA) and BytePS(without EFA), DistributedDataParallel(with EFA) and Herring(with EFA) training $\text{BERT}_{\text{large}}$(phase1)

B. Throughput and scalability

For large models with hundreds of millions of parameters, gradient aggregation introduces significant data transfer after each training iteration. Depending on the model size, communication overhead could become a significant portion of the training time. Our experiments shows that communication can take nearly half of the training time with 512 or more GPUs. When communication becomes a bottleneck overall training is slowed down and computing resources become underutilized. Hence it is very important to have efficient communication library for distributed training to efficiently utilize expensive GPU resources.

In Figure 11, we show how throughput and scaling efficiency varies with cluster size when training with Horovod, BytePS and Herring. For throughput experiments we use $\text{BERT}_{\text{large}}$ training with sequence length 128 and batch size of 64 per GPU. We use Tensorflow framework for Horovod, BytePS and Herring experiments, we use PyTorch framework for DistributedDataParallel(apex.parallel.DistributedDataParallel) experiments. For BytePS and Herring experiments we use 32 parameter-servers for 256 GPU experiments and 64 parameter-servers for 256 GPU experiments.

Single node (8 GPU) throughput is very similar across all four libraries compared. On single node, PyTorch DistributedDataParallel throughput is 12% higher because of faster CUDA kernels. Single GPU throughput for Horovod and Herring is 1143 seq/sec, BytePS is 1104 and PyTorch is 1276 seq/sec. Due to faster PyTorch kernels DistributedDataParallel experiments achieve higher throughput for 8 and 64 GPUs compared to Tensorflow. However with larger clusters high communication costs of DistributedDataParallel overpowers speed up from fast kernels. Note that we experiment Horovod, BytePS and DistributedDataParallel with different system level configuration parameters to achieve best performance.

In Figure 11a, firstly we observe that Herring provides substantially higher throughput for all cluster sizes. While Horovod and BytePS provides very similar throughput for 64, 128 and 256 GPUs throughput gap with Herring increases as the number of GPUs increase. DistributedDataParallel provides higher throughput than Horovod and BytePS. When training with 512 GPUs(64 nodes) Herring provides 1.78 times throughput of Horovod, 1.59 times throughput of BytePS and 1.39 times throughput of DistributedDataParallel.

In Figure 11b, we present scaling efficiency of Horovod, BytePS, DistributedDataParallel and Herring. Scaling efficiency is directly proportional to throughput and represents time to train improvements with increasing cluster size. Here we use single node throughput as base and calculate scaling efficiency based on single node throughput. We use weak scaling efficiency, i.e. we keep the batch size per GPU same and measure total throughput in the cluster.

Scalability of Horovod starts from 65% and declines rapidly with increasing number of GPUs. As a parameter-server-based approach BytePS provides better throughput with 512 GPUs compared to Horovod. However BytePS suffers from limited scalability(55%) with 512 GPUs due to not utilizing EFA and not balancing network load on servers. While DistributedDataParallel achieves 80% scalability on 64 GPUs, its scalability with 512 GPUs is around 56% The most striking observation is Herring provides not only higher but also sustained scaling efficiency. Herring consistently provides more than 85% scalability over all cluster sizes.

One of the reasons why Herring is able to achieve high scaling efficiency is because gradient reduction is done on CPUs instead of GPUs. While ncclAllReduce can be called during the backward pass in a different CUDA stream than the one used for backward pass, both backward pass and ncclAllReduce compete for the same GPU resources. As a result, in most cases calling ncclAllReduce in parallel during

the backward pass does not make the iteration time any shorter than calling ncclAllReduce after the backward pass is done. Table I compares BERT iteration time while using different thresholds to all-reduce gradients on a 64 GPU cluster. When a threshold of T is used, ncclAllreduce is called whenever there is more than T bytes to all-reduce. When T is set to the size of the model, all-reduce happens after the entire backward pass with no overlap. Note that overlapping ncclAllreduce with backward pass does not improve throughput for BERT. Herring on the other hand does gradient reduction on CPUs and therefore overlaps most of the gradient computation with the backward pass. Gradients produced toward the end of the backward pass alone need to be reduced after the backward pass.

TABLE I: With NCCL, step time for BERT is shortest when gradient averaging is done separately after the backward pass. This is because both `ncclAllReduce` and backward pass compete for the same GPU resources.

Threshold	Throughput	Step Time
680 MB	1.4	714
200 MB	1.39	720
50 MB	1.3	769
25 MB	1.22	820

Reducing gradients on the CPU is not a new technique and that alone doesn't gurantee high scaling efficiency. To illustrate this, table II shows a list of gradients produced in the last ten milliseconds of the BERT backward pass.

Note that the gradients vary largely in size with the largest being 61 MB and the smallest being 2 KB. It is impossible to balance these variables across a large number of servers while treating the variables as atomic units. Herring does not treat variables as atomic units. Herring fuses them in balanced fusion buffer and shards the BFB across all servers. In the above case, the balanced fusion buffer would be 86684 KB in size. Each server will send and receive $(1/N_s) * N_w$ of 86684 KB where N_s is the number of server nodes and N_w is the number of worker nodes.

Table III compares the amount of data the server hosting $parameter_0$ will send and receive during each iteration for different cluster sizes. Note that in the case of Herring the amount of data sent and received by each node per iteration remains constant irrespective of the cluster size.

Since workers need to receive all reduced gradients before next iteration of training could begin, iteration time will depend on the slowest node in the cluster. Balanced Fusion Buffer ensures all nodes send and receive the same amount of data without creating bottlenecks on any one node.

In summary, Herring consistently provides significantly higher throughput compared to Horovod and BytePS in distributed training. It is worth noting that, in addition to high scaling efficiency, Herring is able provide sustained scaling efficiency with increasing cluster size.

TABLE II: Gradients produced in the last 10 milliseconds of BERT backward pass.

Time(us)	Variable Index	Gradient Length	Gradient Size(KB)
379795	17	4194304	8192
380525	16	4096	8
3822661	15	4194304	8192
3824614	13	1024	2
3825005	14	1024	2
3826271	12	1024	2
3831110	11	1048576	2048
3842720	10	1024	2
3849548	9	1048576	2048
3850360	8	1024	2
3857206	7	1048576	2048
3857998	6	1024	2
3863448	5	1048576	2048
3865702	3	1024	2
3866097	4	1024	2
3869997	2	2048	4
3873000	1	524288	1024
3882258	0	31260672	61056

TABLE III: Amount of data sent / received by servers after each iteration.

Cluster Size	Data sent / received by traditional parameter server	Data sent / received by Herring server
8	477	84.6
16	954	84.6
32	1908	84.6
64	3816	84.6

C. BERT training

In addition to throughput and scalability experiments, we also show impact of Herring on time to train with BERT model which is a commonly used NLP model. We use $\text{BERT}_{\text{large}}$ with 340 million parameters and we use the same dataset in [36] which is a combination of Wikipedia(2.5 billion words) and BookCorpus(800 million words). We use "whole word masking" where masking is done on all tokens corresponding to a word compared to masking a single token at a time. We evaluate the model quality on SQuAD v1.1 fine-tuning task. We use F1 score as the accuracy metric for our experiments.

We follow mixed batch training approach in [26]. In mixed batch training approach, model is trained for 9/10 epochs for first phase and 1/10 epochs for second phase. In the first phase model is trained with a maximum sequence length of 128 where in second stage model learns long term dependencies by training with 512 sequence length. In our experiments, we

use large batch size starting from 16K to 96K for first phase and 2K to 32K for second phase. Using large batch size allows us to scale to multiple instances while effectively utilizing each GPU. However large batch training also introduces convergence challenges [37], [38]. In order to achieve efficient convergence with large batch training, we need to carefully select optimizer, warm up and learning rate schedule. For large batch convergence, we use LAMB optimizer [26]. We use re-warm-up in the second stage i.e., started learning rate from zero. We adopt polynomial learning rate decay with a power of 0.5.

Graph level optimizations: For further improvements on time train, we apply graph level optimizations on training in addition to Herring. We used fp16 computation to speed up training and achieve memory savings. In addition to fp16 computation, we leverage XLA [39] to achieve speed ups with operator fusion. We also use fp16 communication to reduce the bandwidth usage during gradient averaging. With 340 million model parameters, at each step BERT requires 680MB gradient update with fp16 communication. Lastly, we implement LAMB optimizer in CUDA level. We obtain 4% improvement on step time with LAMB CUDA implementation compared to python level implementation. We apply graph level optimizations on both Horovod and Herring end to end training experiments.

D. Speed up

In Table IV we show time to train and accuracy of $\text{BERT}_{\text{large}}$ training with different number of GPUs. When we scale training through different cluster sizes, we adjust the number of steps based on the effective batch size to keep the number of examples processed same for each experiment. Our accuracy target is to achieve 90+ F1 score with Squad v1.1 fine-tuning task.

First, we compare Horovod and Herring end to end $\text{BERT}_{\text{large}}$ training with 256, 512 GPUs. For 256 GPUs, time to train Bert model is 423.5 minutes with Herring compared to 639 minutes with Horovod. Significant difference in time to train is also observed with 512 GPUs as well. While $\text{BERT}_{\text{large}}$ model can be trained in 216 minutes with Herring, training with Horovod takes 390 minutes. When we look at scalability, we are able to achieve almost linear scaling efficiency with Herring. Although Horovod also provides 1.5 speed up with 512 GPUs compared to 256 GPUs, it takes more time train with Horovod than Herring.

Next, we focus on scaling to an extremely large cluster of 2048 GPUs while keeping the same scaling efficiency and accuracy. Although we are able to train $\text{BERT}_{\text{large}}$ in 71.5 minutes by combining *Herring* with other graph-level optimizations described in Section IV-C, we are able to further improve the training time of $\text{BERT}_{\text{large}}$, and bring down the time-to-train to 62 minutes. For 62 minutes training, we increased effective batch size in phase1. With careful hyperparameter tunning we succeed to achieve 90.49 F1 score while cutting training time by 13%. To our knowledge, we achieve the highest scaling efficiency on the largest cluster used to train BERT in the cloud.

E. Cost of training

We empirically compare the performance of *Herring* with parameter server and allreduce-based approaches with $\text{BERT}_{\text{large}}$ training. We show we can train $\text{BERT}_{\text{large}}$ model much faster with Herring compared to other approaches. Herring achieves performance improvements by adopting a parameter-server-based approach and adopting EFA hence better utilizing bandwidth. Although there are many advantages of parameter-server-based communication, there is an additional resource cost due to allocated parameter-servers compared to allreduce-based approaches. Although the use of additional instances as parameter servers does incur a cost, Herring enables faster training which compensates the cost of using additional servers. Herring uses `p3dn.24xlarge instances` instances as workers, and `c5n.18xlarge instance` instances as parameter-server. As of today, hourly cost of `c5n.18xlarge instances`instance is around 1/10th of the cost of a `p3dn.24xlarge instances` instance. As shown in Table IV, Herring provides 33% speed-up with 256 GPUs, and 44% speed-up with 512 GPUs on training time, which dominates the additional cost of the parameter servers.

V. Ablation studies

A. Impact of EFA

Elastic Fabric Adapter (EFA) helps avoid TCP head-of-line blocking by splitting a tensor into many small pieces and routing them through hundreds of different paths over the network. Without EFA, TCP brings down throughput considerably. Table V below shows time taken to average 100 MB of gradients in milliseconds with and without EFA.

B. Impact of proxies

Single process is unable to fully utilize the available bandwidth in a node. To optimally utilize the available bandwidth, Herring uses multiple proxy servers. Each proxy server communicates with a subset of workers. Table VI shows time taken to average 100 MB of gradients in milliseconds with and without proxies.

VI. Conclusion

Distributed training is critical for reducing training time of large deep neural networks. However, communication can easily become a bottleneck without efficient deep learning specific communication libraries. To this end, we revisit parameter-server based communication. In this paper, we propose a new communication library called Herring, to alleviate the bottlenecks of parameter-server based distributed training as well as leveraging EFA. Herring solves unfair bandwidth allocation, unbalanced workload on parameters and bursty communication problems by careful integration of balanced fusion buffers and leveraging EFA. We show that Herring can provide substantially better throughput compared to current allreduce-based and parameter-server-based approaches,

TABLE IV: Time to train and accuracy results with $\text{BERT}_{\text{large}}$ mixed batch training. All experiments process same number of examples, i.e., we adjust number of steps based on batch size. We evaluate the pretrained $\text{BERT}_{\text{large}}$ model on Squad v1.1. We evaluate model accuracy by F1 score of Squad v1.1 fine-tuning. We use p3dn.24xlarge EC2 instances for our experiments.

GPUs	Batch size in Phase1	Batch size in Phase2	Total training time(min)	F1 score acc	Distributed communication library
256	16K	2K	639	91.45	Horovod
256	16K	2K	423.5	91.23	Herring
512	32K	4K	390	91.16	Horovod
512	32K	4K	216.5	91.42	Herring
2048	64K	32K	71.5	90.79	Herring
2048	96K	32K	62	90.48	Herring

TABLE V: Impact of EFA on gradient averaging time(milliseconds) with 100MB gradients

Gradient averaging time(ms)	64 GPUs	128 GPUs	256 GPUs	512 GPUs
Herring with EFA	13.457	14.21	15.237	16.449
Herring w/o EFA	20.53	24.21	27.49	31.32

TABLE VI: Impact of balanced proxies on gradient averaging time(milliseconds) with 100MB gradients

Step time(ms)	64 GPUs	128 GPUs	256 GPUs	512 GPUs
Herring with Proxy	13.457	14.21	15.237	16.449
Herring w/o Proxy	17.83	19.3	20.2	22.2

namely Horovod and BytePS. We also show Herring provides sustained scalability with very large cluster sizes. By using Herring, we are able train $\text{BERT}_{\text{large}}$ in 62 minutes and achieve 85% scaling efficiency without any degradation in accuracy.

REFERENCES

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[2] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231. [Online]. Available: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[4] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 583–598.

[5] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 19–27. [Online]. Available: http://papers.nips.cc/paper/5597-communication-efficient-distributed-machine-learning-with-the-parameter-server.pdf

[6] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[7] NVidia. (2019) Massively scale your deep learning training with nccl 2.4. [Online]. Available: https://devblogs.nvidia.com/massively-scale-deep-learning-training-nccl-2-4/

[8] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, Mar. 2005.

[9] J. F. Canny and H. Zhao, "Butterfly mixing: Accelerating incremental-update algorithms on clusters," in *Proceedings of the 13th SIAM International Conference on Data Mining, May 2-4, 2013. Austin, Texas, USA*. SIAM, 2013, pp. 785–793.

[10] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," *Parallel Computing*, vol. 35, no. 12, pp. 581 – 594, 2009, selected papers from the 14th European PVM/MPI Users Group Meeting. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819109000957

[11] AWS. (2019) Elastic fabric adapter. [Online]. Available: https://aws.amazon.com/hpc/efa/

[12] 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 *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2901318.2901323

[13] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[14] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," *CoRR*, vol. abs/1706.03292, 2017. [Online]. Available: http://arxiv.org/abs/1706.03292

[15] S. Li, T. Ben-Nun, S. D. Girolamo, D. Alistarh, and T. Hoefler, "Taming unbalanced training workloads in deep learning with partial collective operations," *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb 2020.

[16] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," 2017.

[17] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.

[18] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[19] B. Research. (2017) Bringing hpc techniques deep learning. [Online]. Available: http://research.baidu.com/bringing-hpc-techniques-deep-learning/

[20] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, "Image classification at supercomputer scale," *CoRR*, vol. abs/1811.06992, 2018. [Online]. Available: http://arxiv.org/abs/1811.06992

[21] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama *et al.*, "Massively distributed sgd: Imagenet/resnet-50 training in a flash," *arXiv preprint arXiv:1811.05233*, 2018.

[22] M. Wang, C. Meng, G. Long, C. Wu, J. Yang, W. Lin, and Y. Jia, "Characterizing deep learning training workloads on alibaba-pai," 2019.

[23] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica, "Blink: Fast and generic collectives for distributed ml," 2019.

[24] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[25] N. Laanait, J. Romero, J. Yin, M. T. Young, S. Treichler, V. Starchenko, A. Borisevich, A. Sergeev, and M. Matheson, "Exascale deep learning for scientific inverse problems," *arXiv preprint arXiv:1909.11150*, 2019.

[26] Y. You, J. Li, J. Hseu, X. Song, J. Demmel, and C. Hsieh, "Reducing BERT pre-training time from 3 days to 76 minutes," *CoRR*, vol. abs/1904.00962, 2019. [Online]. Available: http://arxiv.org/abs/1904.00962

[27] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," in *Advances in Neural Information Processing Systems*, 2017, pp. 1709–1720.

[28] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *CoRR*, vol. abs/1712.01887, 2017. [Online]. Available: http://arxiv.org/abs/1712.01887

[29] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, "signsgd: Compressed optimisation for non-convex problems," *arXiv preprint arXiv:1802.04434*, 2018.

[30] D. Alistarh, T. Hoefler, M. Johansson, N. Konstantinov, S. Khirirat, and C. Renggli, "The convergence of sparsified gradient methods," in *Advances in Neural Information Processing Systems*, 2018, pp. 5973–5983.

[31] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified sgd with memory," in *Advances in Neural Information Processing Systems*, 2018, pp. 4447–4458.

[32] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization," in *Advances in Neural Information Processing Systems*, 2018, pp. 1299–1309.

[33] S. U. Stich, "Local sgd converges fast and communicates little," *arXiv preprint arXiv:1805.09767*, 2018.

[34] D. Basu, D. Data, C. Karakus, and S. Diggavi, "Qsparse-local-sgd: Distributed sgd with quantization, sparsification and local computations," in *Advances in Neural Information Processing Systems*, 2019, pp. 14 668–14 679.

[35] AWS. (2016) Elastic network adapter. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2016/06/introducing-elastic-network-adapter-ena-the-next-generation-network-interface-for-ec2-instances

[36] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[37] E. Hoffer, I. Hubara, and D. Soudry, "Train longer, generalize better: closing the generalization gap in large batch training of neural networks," 2017.

[38] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," 2016.

[39] TensorFlow. (2017) Xla:optimizing compiler for machine learning. [Online]. Available: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla