ON THE UTILITY OF GRADIENT COMPRESSION IN DISTRIBUTED TRAINING SYSTEMS

Saurabh Agarwal[1] Hongyi Wang[2] Shivaram Venkataraman[1] Dimitris Papailiopoulos[3]

ABSTRACT

A rich body of prior work has highlighted the existence of communication bottlenecks in synchronous data-parallel training. To alleviate these bottlenecks, a long line of recent research proposes gradient and model compression methods. In this work, we evaluate the efficacy of gradient compression methods and compare their scalability with optimized implementations of synchronous data-parallel SGD across more than 200 realistic distributed setups. Surprisingly, we observe that only in 6 cases out of more than 200, gradient compression methods provide speedup over optimized synchronous data-parallel training in the typical data-center setting. We conduct an extensive investigation to identify the root causes of this phenomenon, and offer a performance model that can be used to identify the benefits of gradient compression for a variety of system setups. Based on our analysis, we propose a list of desirable properties that gradient compression methods should satisfy, in order for them to provide meaningful utility. Our code is available at `https://github.com/uw-mad-dash/GradCompressionUtility`.

1 INTRODUCTION

Synchronous data parallel stochastic gradient descent (SGD) is one of the most widely adopted approaches for distributed learning (Li et al., 2020; Narayanan et al., 2019; Dean et al., 2012a). One iteration of distributed data parallel SGD comprises two main phases: gradient computation and gradient aggregation. During the computation phase, the gradient of the model is typically computed using backpropagation. This is followed by an aggregation phase, where gradients are synchronously averaged among all participating nodes (Iandola et al., 2016; Goyal et al., 2017). During this second phase, for state-of-the-art neural network models, millions to billions of parameters are communicated among nodes (Brown et al., 2020), which has been shown to lead to communication bottlenecks (Dean et al., 2012b; Seide et al., 2014; Qi et al., 2017; Grubic et al., 2018; Alistarh et al., 2017).

Alleviating communication bottlenecks in distributed training has been an active area of research in recent years. A long line of work has focused on lossy gradient compression methods to mitigate communication costs. Lossy gradient compression methods typically use techniques such as low-precision training (Seide et al., 2014; Alistarh et al., 2017; Bernstein et al., 2018a; Wen et al., 2017), sparsification (Aji & Heafield, 2017; Lin et al., 2017), or low-rank updates (Wang et al., 2018; Vogels et al., 2019), with the common goal of reduced communication. Although these methods require significant effort to integrate into deep learning frameworks and often introduce extra hyper-parameters, they promise significant reductions in communication, *e.g.*, POWERSGD (Vogels et al., 2019) provides a greater than $100\times$ reduction in communication with minimal effect on accuracy on certain tasks.

Concurrent to the work on gradient compression, a number of system-level optimizations have been proposed to speed up distributed data-parallel synchronous SGD (sync-SGD). Techniques like ring-reduce (Thakur et al., 2005) and tree-reduce (Sanders et al., 2009) have been implemented in several high performance communication libraries (*e.g.*, NCCL and Gloo) which in turn are tightly integrated into popular deep learning libraries like PyTorch (Paszke et al., 2019; Li et al., 2020) and Tensorflow (Abadi et al., 2016). Both ring-reduce and tree-reduce, are bandwidth efficient and have a constant, and logarithmic dependence on the number of nodes, respectively, *i.e.*, the total number of bytes communicated remains sublinear in the number of machines used for training. To further reduce the observed overhead of communication, recent systems *overlap* the gradient computation and communication phases (Li et al., 2020; Sergeev & Del Balso, 2018). Figure 1 illustrates how overlapping the backward pass and the communication phases are implemented. Figure 2 shows the extent to which overlapping

[1]Department of Computer Science, University of Wisconsin-Madison [2]Machine Learning Department, Carnegie Mellon University [3]Department of Electrical and Computer Engineering, University of Wisconsin-Madison . Correspondence to: Saurabh Agarwal <agarwal@cs.wisc.edu>.

Proceedings of the 5^{th} *MLSys Conference*, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).

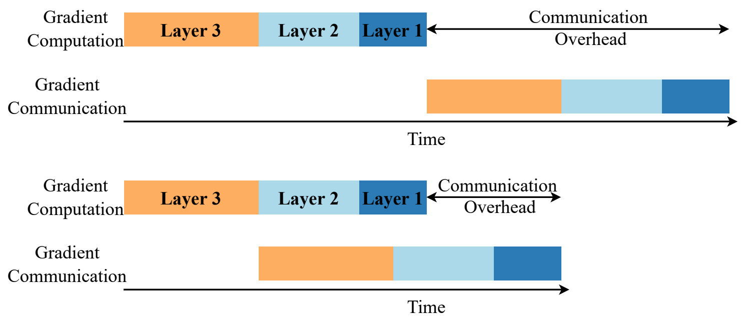


Figure 1. **Illustration of how overlapping can reduce the total iteration time.** (Above) Gradient computation and communication done serially. (Below) Gradient computation and communication being overlapped, *i.e.*, when the gradient of a layer is computed, it is communicated right after the gradient of the previous layer.

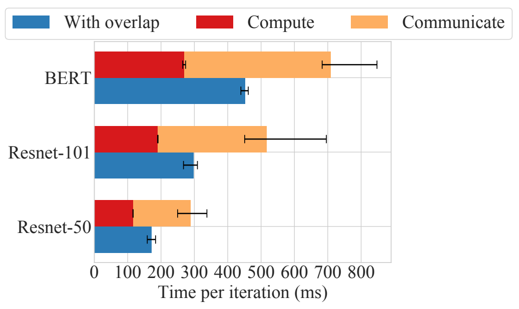


Figure 2. **Effect of Overlap:** We plot the iteration time for computation and gradient synchronization for 64 GPUs, both with and without overlap. In case of Resnet-50 we observe that overlapping reduces iteration time by upto 46%.

helps improves the scalability of distributed training. For PyTorch DDP (Li et al., 2020) with ResNet-50, we observe an almost 46% reduction in time per iteration when overlapping communication with the backward pass. These system-level optimizations are transparent to users, *i.e.*, there is no requirement for additional hyper-parameters and the user need not worry about accuracy degradation.

Given the above two trends, our objective in this work is to measure the utility of gradient compression in distributed training. We empirically compare an off-the-shelf implementation of syncSGD with three popular gradient compression methods, implemented using new functionality (PytorchCommHooks, 2021) provided in PyTorch v1.8 for efficiently integrating gradient compression. The compression methods we compare against are, SIGNSGD (Bernstein et al., 2018a;b), MSTOP-K (Shi et al., 2021), and POWERSGD (Vogels et al., 2019). We test these methods on three popular models, *i.e.*, ResNet-50, ResNet-101 and BERT$_{\text{BASE}}$ (Devlin et al., 2018), and conduct large scale evaluation with up to 96 GPUs. Overall, we test across more than 200 experimental settings accounting for different models, compression algorithms, compression ratios, batch sizes, network bandwidths, and etc.

Our Contributions. We observe that due to the aforementioned systems optimizations to speed up syncSGD, at typical data-center bandwidths, there is *limited opportunity for gradient compression* to provide significant performance improvements. Even for communication heavy models like BERT$_{\text{BASE}}$ (Devlin et al., 2018), the difference between linear scaling and observed per iteration time for off-the-shelf (PyTorch DDP) implementations of syncSGD is approximately 200 milliseconds when using 96 GPUs. For gradient compression methods to provide speedups they need to perform encode-decode and communication within this limited time-frame. However, we find that existing gradient compression methods have high encode-decode times (upwards of 50 milliseconds as shown in Table 2) and impose additional restrictions on communication protocols, which significantly limit the speedups from gradient compression.

Further, we observe that gradient compression methods cannot fully utilize system optimizations like overlapping compression and backward pass. This is because both gradient compression and backward pass are compute intensive and compete for GPU resources leading to an overall slowdown. We also find that when overlapping gradient communication with the backward pass, large batch sizes further reduce the communication overhead per iteration. The reason is that large batch sizes increase the time spent in computation providing more opportunity to "hide" communication overheads.

Finally, we also observe that, as reported by previous works (Vogels et al., 2019; Cho et al.), *all-reduce* compatible gradient compression methods scale better. For instance, SIGNSGD, which is not compatible with *all-reduce*, takes 1042ms for a single iteration of ResNet-101 on 96 GPUs, while POWERSGD Rank-16, which is compatible with *all-reduce*, takes only 470ms, while syncSGD which is also compatible with *all-reduce* takes only 262ms.

To understand the regimes in which gradient compression can be helpful, we develop an analytical performance model and verify its accuracy. Using the performance model we investigate how various factors like network bandwidth and compute availability affect the scalability of distributed training and discuss scenarios where gradient compression can be effective. For instance, in Figure 3 with the aid of our performance model we show that at lower bandwidths gradient compression can provide significant benefits. The markers in Figure 3 are measurements on actual hardware, showing how close our performance model tracks the observed values in actual experiments. Our performance model also suggests that algorithm designers should focus on reducing the overhead of compression rather than trying to achieve high compression ratios. This is because at typical data-

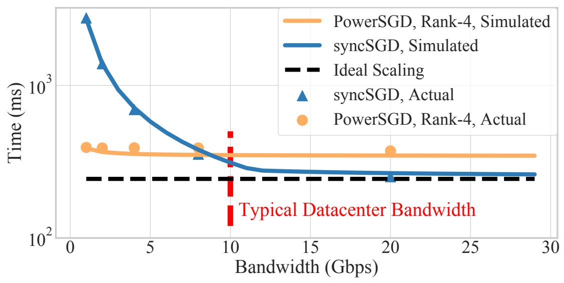


Figure 3. **Evaluating effect of network bandwidth (simulated):** Above curve is for Resnet-101, batch size 64 on 64 GPUs. We observe that at bandwidth lower than 8.2 Gbps, PowerSGD Rank-4 can provide speedups but above that syncSGD performs better.

center bandwidths ($>$ 10Gbps) we only need a compression ratio of $\approx 4\times$ even for large models like ResNet-101 and BERT$_{\text{BASE}}$ to achieve almost linear scalability. Finally, in Section 5, we also discuss several scenarios where existing gradient compression schemes can be effectively used to improve per iteration times.

We would like to point out that our results are derived from analyzing per-iteration times and do not account for any loss in accuracy incurred by gradient compression. In that sense our analysis is *generous* to gradient compression methods, as many lead to some small accuracy loss. This loss typically requires a larger number of iterations to overcome, or mitigation techniques with additional computation or memory footprint (*e.g.*, the error feedback scheme (Seide et al., 2014; Karimireddy et al., 2019; Stich et al., 2018)).

In summary, our analysis establishes that for synchronous data-parallel distributed training in the ubiquitous datacenter setting($\approx$ 10Gbps bandwidth), for popular DNNs, on commonly used hardware (NVIDIA-V100 GPUs), gradient compression methods do not provide any of the promised speedups, once we account for system level optimizations in syncSGD. However, we also show that there do exist setups apart from the common data center setups where gradient compression can provide significant benefits. To identify regimes where gradient compression can provide benefits, we develop a performance model that can be used by both practitioners and researchers to predict performance at a large scale without the need of performing any real experiments. Based on our empirical analysis and performance model we also provide guidelines for building future gradient compression algorithms.

2 Background and Related Work

We first provide a brief background of several different threads of prior work that aim at enabling faster distributed machine learning.

Table 1. **Comparing aggregation schemes:** We show how latency and bandwidth term scale for different aggregation strategies. α is the latency, β is the inverse of bandwidth, and n is the size of vector communicated. p is the number of machines

Algorithm	Latency	Bandwidth
Ring Reduce	$2(p-1)\alpha$	$2\beta\frac{(p-1)}{p}n$
Tree Reduce	$2\alpha\log p$	$2\beta(\log p)n$
Parameter Server	2α	$2\beta(p-1)n$

2.1 Gradient Compression

Several lossy gradient compression methods based on quantization (Alistarh et al., 2017; Bernstein et al., 2018a; Karimireddy et al., 2019; Dettmers, 2015; Seide et al., 2014; Wen et al., 2017; Bernstein et al., 2018b; Yu et al., 2019; Li et al., 2018; Horvath et al., 2019; Tang et al., 2019; Dryden et al., 2016; Strom, 2015; Gandikota et al., 2021; Zheng et al., 2019; Zhang et al., 2017; Wu et al., 2018; Tang et al., 2018a), sparsification (Stich et al., 2018; Lin et al., 2018; Aji & Heafield, 2017; Alistarh et al., 2018; Lin et al., 2018; Shi et al., 2019a;b; Fang et al., 2019; M Abdelmoniem et al., 2021; Shi et al., 2021; Wangni et al., 2018; Tang et al., 2018a; Sattler et al., 2019a;b), low rank decomposition (Wang et al., 2018; Vogels et al., 2019; Wang et al., 2021), and other approaches (Acharya et al., 2019; Suresh et al., 2017; Ivkin et al., 2019) have been proposed in literature. Recent surveys (Xu et al., 2020a; Tang et al., 2020) describe these methods in detail.

In this work, we benchmark several popular gradient compression schemes (Table 2), and we then pick three gradient compression schemes which have the least compression overheads and high compression ratios for detailed analysis. We chose, quantization based SIGNSGD (Bernstein et al., 2018a;b), low-rank decomposition based POWERSGD (Vogels et al., 2019) and sparsification based MSTOP-K (Shi et al., 2021). We compare and evaluate these schemes to see if they provide any benefit over off-the-shelf implementation of syncSGD, *i.e.*, PyTorch DDP (Li et al., 2020).

2.2 System Advances

Next, we provide a brief overview of several system advances which have been applied to syncSGD to improve the performance of distributed training.

All-reduce. In recent years, systems have shifted from using a parameter server based topology to an all-reduce topology for gradient synchronization. For example, we observe that all submissions to DawnBench (Coleman et al., 2019) use all-reduce for performing distributed training.

Communication costs can be typically modeled using a cost model (Sarvotham et al., 2001) where cost of sending/receiving a vector of size n is computed as the sum of latency and bandwidth requirements. There are several optimizations (Rabenseifner, 2004; Thakur et al., 2005; Hoefler et al., 2011; Sanders et al., 2009) for all-reduce based collectives like ring-reduce (Barnett et al., 1994), tree-reduce (Sanders et al., 2009), recursive doubling (Ueno & Yokota, 2019), 2D-Torus (Mikami et al., 2018; Jouppi et al., 2017), and etc. These optimizations explore the trade-off between the latency and bandwidth terms. We list latency and bandwidth terms for a few aggregation strategies in Table 1 for synchronizing a vector of size n among p machines. In Table 1, α represents the latency term (typically between 0.5 to 1ms in public clouds) and β represents bandwidth term. We would like to point out that the bandwidth requirement for ring reduce stays almost constant even with increase in number of machines p. High performance implementations like NVIDIA-NCCL (ncc) dynamically chooses between tree and ring reduce based on several factors like number of machines, bandwidth, interconnect, communication size to list a few. In this work for simplicity, we analyze our results with the communication model of ring-reduce.

Communication and Computation Overlap. Gradients for DNNs are calculated layerwise, therefore, gradients of later layers are available before initial layers. Instead of waiting for the availability of all the gradients, popular deep learning frameworks (Li et al., 2020; Paszke et al., 2019; Abadi et al., 2016) start gradient communication when some of the gradients are available. This leads to overlapping gradient computation with communication, hiding the time spent in communication. Figure 1 illustrates how overlap can provide speedups. In Figure 2, we observe that overlapping can provide speedups of almost 46% for ResNet-50.

Bucketing Gradients. Calling the all-reduce collective per layer can often lead to large overheads. To amortize the overhead of calling all-reduce, optimized implementation of syncSGD (Li et al., 2020; Sergeev & Del Balso, 2018) create fixed size buckets. Once the gradients for a bucket are calculated then *all-reduce* is called on the entire bucket. Bucket sizes are typically large (25 MB by default in PyTorch).

In this paper, we benchmark the runtime of the systems with the aforementioned optimizations to compare against gradient compression methods on real-world computer vision and natural language processing tasks.

2.3 Other Related Work

Several works have looked at improving communication efficiency by use of Gossip based protocols (Lian et al., 2017; Tang et al., 2018b; Koloskova et al., 2019b;a). Other methods have looked into improving efficiency of distributed training by enabling use of large batch sizes (You et al., 2019; 2017; Smith et al., 2017; Devarakonda et al., 2017) or lower precision (Micikevicius et al., 2017) without accuracy loss. Other works have also looked at different forms of parallelism (Jia et al., 2018b;a; Huang et al., 2019; Shoeybi et al., 2019; Narayanan et al., 2019; Rasley et al., 2020) for speeding up distributed training. MLPerf (Mattson et al., 2019) and DawnBench (Coleman et al., 2019) are two well known industry supported efforts to perform periodic benchmarking on training and inference speed at scale. Our findings about scalability of all-reduce based compression scheme has also been reported by prior works (Vogels et al., 2019; Cho et al.). A recent survey (Xu et al., 2020a) quantitatively compares several gradient compression methods. However unlike our work it does not account for systems optimization like overlap of communication and computation. Zhang et al. (2020) study whether network is the bottleneck in distributed training. Unlike (Zhang et al., 2020) and other listed works, our study focuses on the utility of gradient compression methods in several different settings and analyzes others aspects beyond network bandwidth like compute availability, batch size, model size, system advances etc. Further, our performance model allows to reason about performance of distributed training and to predict the performance gains without running large scale experiments.

Table 2. Encode-Decode of gradient compression methods for ResNet-50 on V100 GPUs.

Type	Method	$T_{encode\_decode}(ms)$	All-Reduce
Sparsification	MS-TopK - 1%	103	✗
	DGC - 1%	221	✗
	TopK - 1%	273	✗
	RandomK - 1%	163	✓
Quantization	SignSGD	16	✗
	QSGD-2bit	39	✗
	TernGrad	94	✗
Low Rank	PowerSGD-Rank 4	45	✓
	ATOMO-Rank 4	1586	✗

3 Evaluating Gradient Compression

In this section, we perform a detailed experimental evaluation comparing the scalability of gradient compression methods with an optimized syncSGD implementation. We start by analyzing the effects of overlapping gradient compression with gradient computation. Next we run large scale experiments to study how gradient compression methods scale across a range of models.

Methodology. We begin by comparing the overhead of compression methods which have been reported to scale well. Table 2 shows the time for compression and decompression for **nine gradient compression methods** using

ResNet-50 on 64 V100 GPUs. We observe that most gradient compression methods take around 100ms for compressing and decompressing gradients of ResNet-50 on 64 GPUs. However, there are some methods which are considerably faster, *e.g.*, SIGNSGD takes only 16ms for encoding-decoding. Among low-rank methods we find that POWERSGD is around $45\times$ faster than ATOMO (another low rank method) (Wang et al., 2018). Based on this comparison, we choose the most scalable method in each category. Among quantization based methods we choose SIGNSGD (Bernstein et al., 2018a;b) which achieves $32\times$ compression ratio by only communicating the sign of the gradient. Among sparsification based methods we choose MSTOP-K (Shi et al., 2021), a scalable TOP-K method and among low rank methods we choose POWERSGD, a low overhead method with compression ratios of around $100\times$. For syncSGD we use PyTorch-DDP module (Li et al., 2020).

We would like to point out that we use optimistic compression ratios, *e.g.*, for POWERSGD we use Rank-4, 8, and 16. Such high compression ratios have been shown to work (Vogels et al., 2019) for small datasets like CIFAR-10 and WIKITEXT-2 but can lead to accuracy loss for large datasets (Vogels et al., 2019; Ramesh et al., 2021). While for MSTOP-K we are again being optimistic and consider dropping 99.9% gradients and assuming that it will have no loss in accuracy. We chose these since we wanted to consider a best case scenario for gradient compression methods.

We use ResNet-50 (97MB), ResNet-101 (170MB) and $\text{BERT}_{\text{BASE}}$ (418MB) as the models to study given their disparate communication and computation requirements. Similar models were used by prior works (Vogels et al., 2019; Xu et al., 2020b) in gradient compression to compare the performance of gradient compression schemes and our code can be easily used to benchmark other models as well. For timing measurements on vision models we use the ImageNet dataset (Deng et al., 2009) and we fine-tune the $\text{BERT}_{\text{BASE}}$ model on Sogou News dataset (Sun et al., 2019). For the timing measurements, we run 60 iterations for each setup and discard the first 10. We plot the mean of the remaining 50. The error bars in the figure correspond to minimum and maximum values.

Our experiments are conducted over *p3.8xlarge* instances on Amazon EC2. Each instance is equipped with 4 V100 GPUs and provides around 10Gpbs of bandwidth. We scale our experiments up to 96 GPUs (24 *p3.8xlarge* instances) and consider weak scaling, *i.e.*, the number of inputs per worker is kept constant as the number of workers increase. This is a commonly used scenario for evaluating the scalability of deep learning training (Coleman et al., 2019; Narayanan et al., 2019). Thus, when we refer to a particular batch size, it is the batch size at each worker.

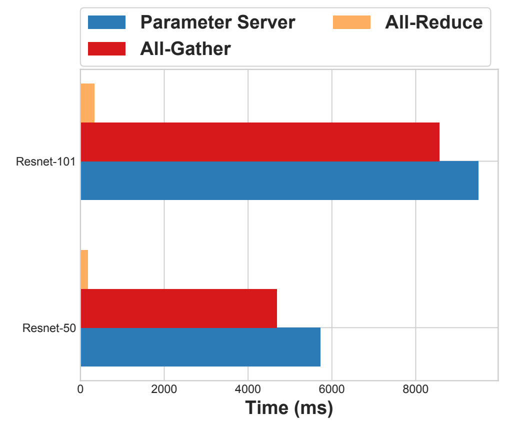


Figure 4. **Comparison between communication collectives:** We observe that among communication schemes *all-reduce* performs significantly better than gather based schemes. Among gather based schemes, *all-gather* performs better than Parameter Server.

Using Per Iteration Time as A Metric Instead of Accuracy. We consistently use time per iteration as the metric for evaluation. It is well known from prior works that gradient compression methods can lead to some final model accuracy loss (Xu et al., 2020b) when used for training. Our main goal in this paper is to study the scalability of distributed training and compare per iteration time of syncSGD against state-of-the-art gradient compression methods. Though important, the final model accuracy that the gradient compression methods achieve is not the main focus of this paper. The per-iteration speedup is a more critical question as if there is limited speedup from using gradient compression then there is no incentive to deploy such methods irrespective of the accuracy. Another reason for not performing an accuracy based study is that gradient compression methods often introduce new hyper-parameters while also requiring modifications to existing hyper-parameters like the learning rate schedule. It is often non-trivial to find optimal hyper-parameters which balance compression and accuracy loss and we plan to study this in future work.

Implementation Details. For performing gradient compression we have two sets of implementations. In the first set of implementation we overlap compression with the gradient computation (backward pass). In the second set, we perform gradient compression after gradient computation. In the first implementation we overlap the compression and communication with the backward pass using DDP communication hooks (PytorchCommHooks, 2021) which were introduced recently in PyTorch v1.8. For the non-overlap version, we observed that compressing gradients incurred least overhead when gradients are collected into a single large matrix and compressed, as it reduces the number of CUDA calls and allows full use of the compute available on the GPU. Therefore in our non-overlapped implementation, instead of compressing gradients layer by layer we collect all the gradients in a single matrix and then perform compression.

For implementing compression algorithms we have used the author provided open source implementations. For POWERSGD without overlap, we used the author provided code which is JIT-optimized (pytorch jit, 2021). For POWERSGD with overlap we used the one supported in PyTorch natively (pyt, 2020). For SIGNSGD we used the author provided C++ library which packs signs into bitmaps, an operation that is not natively supported by PyTorch. MSTOP-K is implemented using vector instructions thus avoiding expensive for loops.

For communication we used highly optimized NVIDIA-NCCL library. For POWERSGD we used *all-reduce* collective since POWERSGD is compatible with *all-reduce*, while for SIGNSGD and MSTOP-K we used *all-gather* collective operation. We used NCCL all-gather instead of traditional parameter server architecture, since in our experiments we observed that all-gather outperforms parameter server in the data center setting. Figure 4 compares performance of all-reduce, all-gather and parameter server for 64 GPUs. We will open-source the code and data post review process.

3.1 Overlapping Compression and Computation

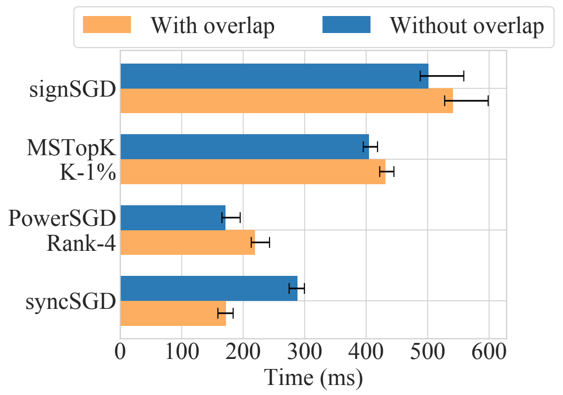


Figure 5. **Overlapping Gradient Compression with Computation:** Overlapping compression leads to requiring more time per iteration than performing it sequentially, due to resource contention for compute resources. The results are for 64 GPUs.

We observe that when gradient compression is performed in parallel with the backward computation it is slower than performing gradient compression after completing backward pass. Figure 5 depicts this phenomenon on ResNet-50 using POWERSGD Rank-4, MSTOP-K-1%, and SIGNSGD. Since both gradient compression and gradient computation are compute-heavy steps, when performed in parallel they end up competing for compute resources on the GPU leading to an overall slow down. On the other hand, syncSGD only performs *all-reduce* operation which is communication heavy with very little compute, thus efficiently utilizing the communication resources on the GPU without affecting the backward pass. Since we consistently observe that compression schemes perform better when not overlapped, for the next set of experiments we use *non-overlapped versions of compression*. For more detailed analysis of the compression overlapped, we refer the reader to Appendix A. Using a GPU profiler, we also verified that gradient compression does indeed block the progress of gradient computation. Figure 13 in Appendix shows a snapshot of activity on the GPU collected using Nsys (Nvidia-Nsight, 2021).

Table 3. **Encode & Decode times for ResNet-50:** Even for a small network like ResNet-50, where time for backward pass is ≈ 122ms, gradient compression methods have high overhead

Compression Method	Compression Parameter	Compression Ratio	$T_{encode-decode}$(ms)
POWERSGD	Rank-4	72×	45
	Rank-8	37×	64
	Rank-16	19×	130
MSTOP-K	1%	100×	103
	.1%	1000×	104
SIGNSGD		32×	16.34

In summary we find:

Takeaway 1 *Gradient Compression methods are not good candidates for overlap with gradient computations on popular GPUs like V100s since both gradient compression and computation are compute heavy processes leading to an overall slowdown.*

3.2 Comparing Gradient Compression with Optimized syncSGD

We next analyse the performance of gradient compression methods against syncSGD.

PowerSGD. We first study the scalability of PowerSGD when compared to syncSGD for ResNet-50, ResNet-101 , and BERT$_{\text{BASE}}$. We use Rank-4, 8 and 16 as discussed previously. As shown in Figure 6 we can see that PowerSGD with Rank 4, 8, and, 16 is *slower* than syncSGD for ResNet-50 and ResNet-101 with batch size 64 (We investigate varying batch sizes in Section 3.3). This is primarily because syncSGD does not incur any overheads from compression and is able to overlap communication with computation. On the other hand, for BERT$_{\text{BASE}}$, which is a much larger model (490MB), we see that for 96 GPUs, Rank-4 and Rank-8 are faster than syncSGD by around 18.8% and 11.3% respectively, while Rank-16 still takes longer than syncSGD.

MSTOP-K. Since the MSTOP-K (Shi et al., 2021) operator is incompatible with *all-reduce* we use *all-gather* for communication. As shown in Figure 7, only in 2 out of 15 different setups we observe a minuscule speedup (around 1.3%) when compared against syncSGD. These speedups are achieved when using MSTOP-K-0.1%, i.e., when 99.9% of the entries in the gradient are dropped. Also, due to high

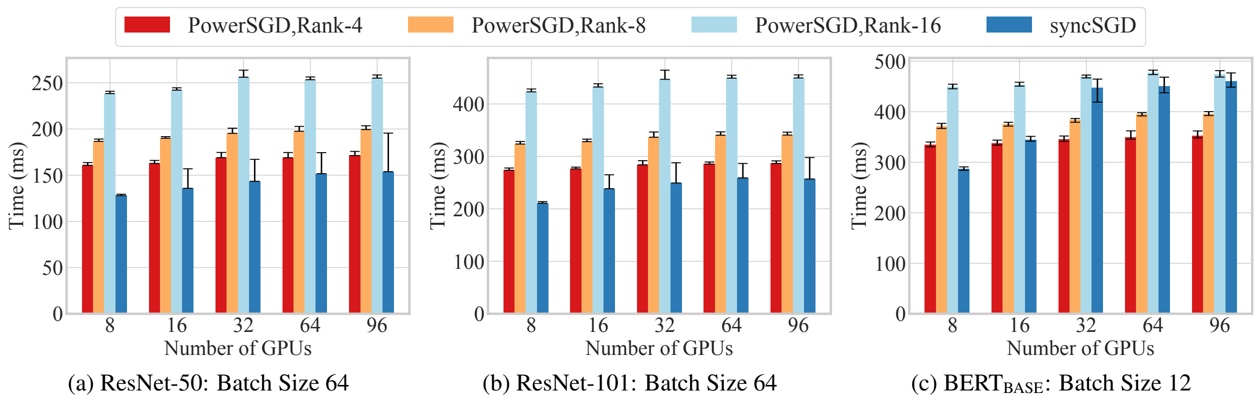


Figure 6. **Scalability of PowerSGD:** When compared against an optimized implementation of syncSGD, PowerSGD provides speedups only in case of $\text{BERT}_{\text{BASE}}$ when using Rank-4 and Rank-8 above 32 GPUs. In other cases it has a high per iteration time.

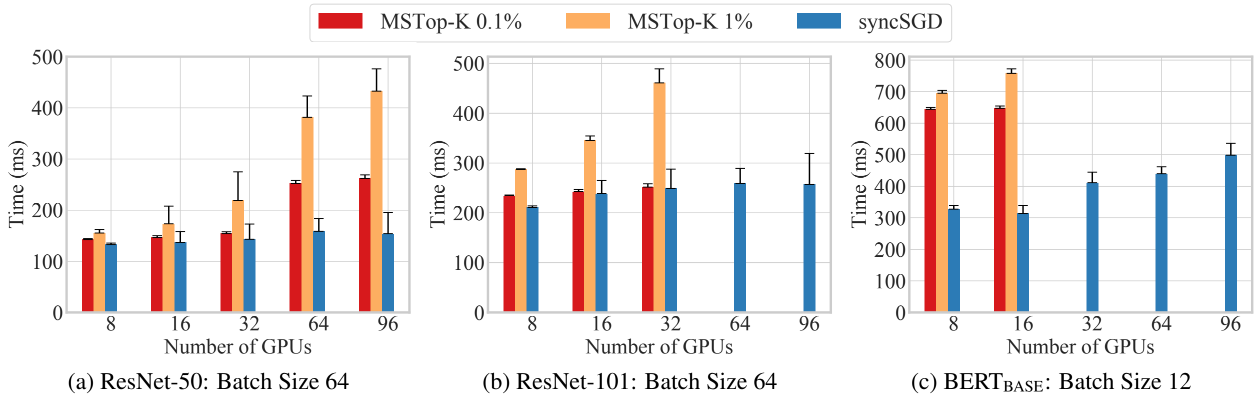


Figure 7. **Scalability of MSTop-K:** Comparing MSTop-K against syncSGD we observe due to lack of compatibility with *all-reduce* MSTop-K performs slower than or comparable to syncSGD . For ResNet-101 and BERT we could not scale Top-K beyond 16 and 32 GPUs respectively, due to running out of memory as memory requirement increasing linearly with number of machines.

memory requirements for creating buffers for the all-gather primitive MSTop-K does not scale beyond 32 GPUs for ResNet-101 and 16 GPUs for BERT on a V100 GPU.

SIGNSGD. We study SIGNSGD with majority vote, where 1 bit is sent for each float (32 bit) leading to 32× compression. Majority vote operation is not associative thus requiring use of all-gather. Figure 8, shows that despite SIGNSGD being extremely quick to encode and decode, due to lack of compatibility with *all reduce*, communication time scales linearly. Further, due to overheads in creating buffers for the all-gather primitive we can not scale SIGNSGD on $\text{BERT}_{\text{BASE}}$ beyond 32 GPUs.

Why Does Gradient Compression Not Lead to Significant Speedups? We identify three reasons for lack of speedups. First as stated in Section 2.2, compression methods are poor candidates for overlapping with gradient computation. Meanwhile, syncSGD as shown in Figure 5 is able to benefit from overlapping communication and backward pass, which provides it a significant advantage.

The second reason, as depicted in Table 3, is high overhead of compression. Since syncSGD, because of system advances has improved scaling, we observe in Figure 10 that even for large models like $\text{BERT}_{\text{BASE}}$, for 96 GPUs the difference from ideal speedup is around 200ms. This indicates, that for compression algorithms to be a viable alternative, they need to be extremely fast and perform compression and communication in less than 200ms even for very large models. However, in Table 2 we observe that several gradient compression take more than 100ms to perform compression on ResNet-50 (97MB), a much smaller network than $\text{BERT}_{\text{BASE}}$(418 MB), leading to slowdowns.

Third reason for slowdown, as pointed by prior works (Vogels et al., 2019; Cho et al.) and experiments in previous section, is lack of compatibility with *all-reduce*. Compression methods compatible with *all-reduce* like PowerSGD are able to scale better. For an operation to be compatible with *all-reduce* it must be associative, *i.e.*, the order of operations should not matter. However, Table 2 shows that several gradient compression methods are incompatible with *all-reduce*. In these cases, to perform gradient aggregation, the workers need to perform an all-gather operation, leading to poor scalability as we increase the number of processors.

Takeaway 2 *Existing gradient compression methods pro-*

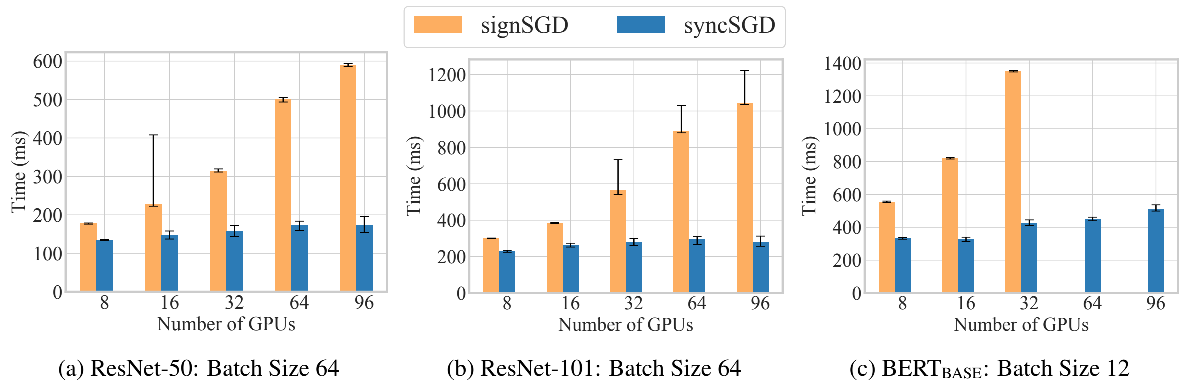


Figure 8. **Scalability of SIGNSGD:** Due to lack of support for *all-reduce* and linearly increasing decode time, across all three models, SIGNSGD performs considerably slower than syncSGD. For BERT$_{\text{BASE}}$ we were not able to scale signSGD beyond 32 GPUs because we ran out of memory on a V100 GPU. This is due to the memory requirement increasing linearly with number of machines.

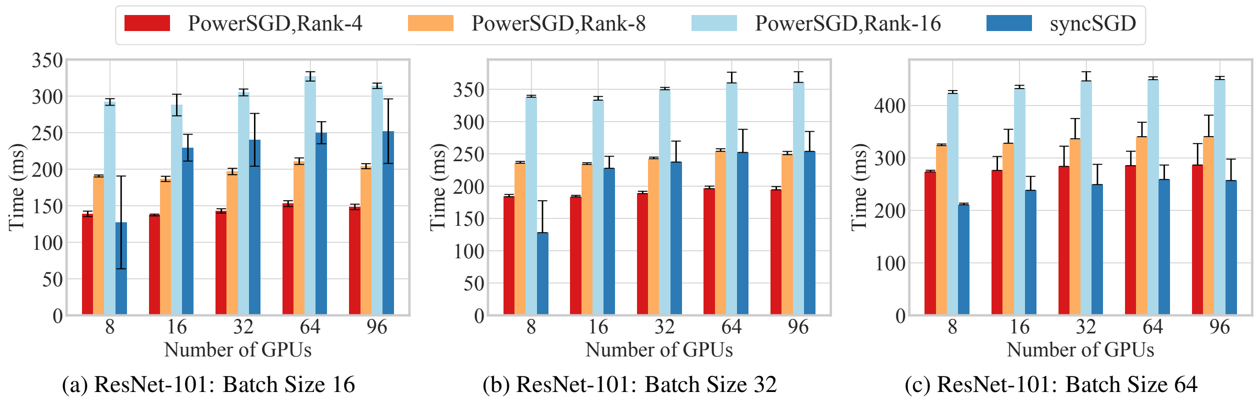


Figure 9. **Effect of varying batch size:** Here we compare POWERSGD against ResNet-101 on different batch sizes. We observe that large batch sizes provide more opportunity to syncSGD to hide the communication time, meanwhile at small batch sizes due to reduced computation time this overlap is not possible. Therefore gradient compression methods become more useful at small batch sizes.

vide limited benefits either due to encoding overheads or due to lack of compatibility with all-reduce across a range of models.

3.3 Effect of Batch Size on Scalability

For analysing the effect of varying batch sizes, we compare PowerSGD against syncSGD since it is the most scalable method we encounter. In Figure 9, for ResNet-101, we find that the benefits of using PowerSGD with Rank-4 drops as the batch size increases. For instance, when using 96 GPUs, PowerSGD Rank-4 provides almost 42.5% speedup when training using batch size 16. This speedup drops to 25.7% for batch size 32 and with batch size 64, we observe that PowerSGD Rank-4 is around 6.3% slower than synSGD. In general, increasing batch size leads to an increase in the compute time which in turn provides more opportunity for syncSGD to overlap computation and communication.

Takeaway 3 *Using large batch sizes often provides enough opportunity for syncSGD to overlap communication with communication thus reducing the extent of benefits achieved from using gradient compression.*

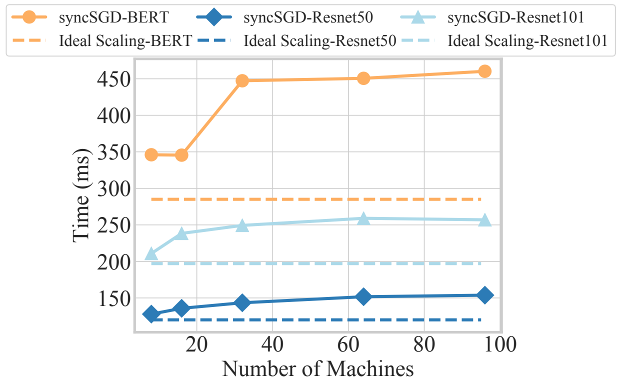


Figure 10. **Difference between linear scaling and observed performance:** We observe that the difference between linear scaling and syncSGD is less than 200 ms at 10Gpbs. This leaves little opportunity for gradient compression methods to provide speedups.

4 IDENTIFYING REGIMES OF HIGH GRADIENT COMPRESSION UTILITY

In the previous section we looked at the performance of distributed training and gradient compression of popular mod-

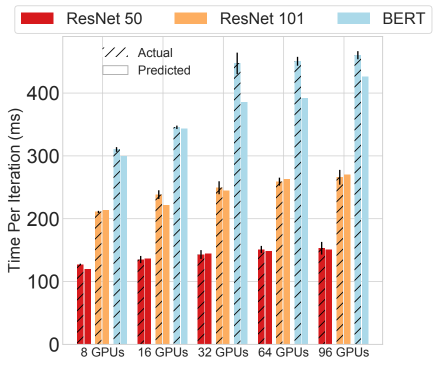


Figure 11. **Verifying performance model for syncSGD:** Our performance model matches the actual performance for all three models across wide range of GPUs. The median difference between predictions and actual runtime is 1.8%.

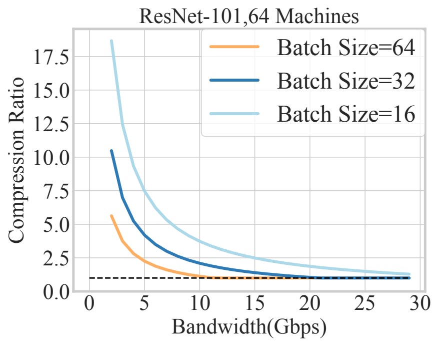


Figure 12. **Required gradient compression for near linear speedups (simulated):** Above figure is for ResNet-101 simulated for 64 machines. We observe that the required gradient compression for near linear scaling at 10 Gbps even for quite small batch sizes is around $4\times$.

els on existing hardware. Next we try to identify regimes, in terms of hardware or model characteristics, where gradient compression can provide significant gains *i.e.*, how will our above results change if we had 100Gbps bandwidth or an $8\times$ faster GPU. To answer such questions, we develop a performance model that can be used to reason about expected performance under different setups.

4.1 Performance Model for Distributed Data Parallel.

Based on optimizations listed for syncSGD in (Li et al., 2020) we build an analytical performance model. We assume the model can be partitioned into k buckets, where the first $k-1$ buckets are of size b and the last bucket is of size $\hat{b}$, where $\hat{b} \leq b$. The time observed for backward pass and gradient synchronization for synSGD becomes:

$$T_{obs} \approx max(\gamma T_{comp}, (k-1) \times T_{comm}(b, p, BW)) + T_{comm}(\hat{b}, p, BW)$$

where T_{obs} is the total time observed for backward pass and synchronization, T_{comp} is the compute time for the backward pass on single machine, $(k-1) \times T_{comm}(b, p, BW)$ is the time required to communicate $k-1$ gradient buckets of size b across p GPUs at BW bandwidth, and $T_{comm}(\hat{b}, p, BW)$ is the time to communicate the last bucket of size $\hat{b}$, which can not be overlapped with computation. Finally, γ represents the factor of slowdown in backward pass due to overlap with communication. We observe γ to between 1.04 to 1.1. In case of syncSGD when using ring-reduce, $T_{comm}(b, p, BW)$ becomes

$$T_{comm}(b, p, BW) = 2\alpha \times (p-1) + 2 \times b \times \frac{(p-1)}{p \times BW} \quad (1)$$

where α is the latency coefficient, b is the bucket size, p is the number of GPUs and BW is the bandwidth available. The performance model for gradient compression methods is in Appendix B.

Verifying Performance Model. We empirically verify our performance model using the same experimental setup as mention in Section 3. As shown in Figure 11 we observe that our model very closely tracks the actual performance in all cases. The median difference between our prediction and actual runtime is 1.8% and the maximum is 13.7%. More details on verification and how we measure the values to input into the performance model can be found in Appendix C.

Utility of the Performance Model. The performance model requires calculation of latency term(α), correction term(γ), time for backward pass and available bandwidth. The goal of the performance model is to allow analysis in the case where hardware is not available, *e.g.*, one can ask a question: at what bandwidth will a particular gradient compression scheme outperform syncSGD for ResNet-50? This is not possible to run since our choice of hardware is constrained by existing bandwidth options (10Gbps or 25Gbps on Amazon EC2 etc.). Another benefit of the performance model is that it characterizes which factors affect training time and how those factors interact, helping researchers and developers to reason about performance.

Limitations. Currently, our performance model only supports the data-parallel setting and is not applicable on other forms of distributed training like model or pipeline parallelism, *i.e.*, we do not consider cases where the model can not fit in single GPU memory. Further, we do not account for asynchronous methods (Dean et al., 2012b; Grishchenko et al., 2018; Mota et al., 2013), *i.e.*, we assume that gradient synchronization is required after every iteration.

4.2 Insights from the Performance Model

How Much Should We Compress? Using the performance model we investigate how much compression is required for linear scalability. Figure 12 shows that even at small batch-sizes for ResNet-101 we need around $4\times$ compression for linear scalability, which is significantly smaller than what most compression methods offer. Our analysis shows that for linear scaling we do not need extremely high compression ratios. In Appendix D, we show that reducing encode-decode time even at the expense of decreased compression ratio helps. As an extreme case, prior work shows by-passing gradient encoding and decoding steps by changing the structure of DNN (Wang et al., 2021).

Effect of Network Bandwidth on Gradient Compression. Figure 3 shows comparison between speedups for ResNet-101 when using syncSGD and PowerSGD Rank-4 at different network bandwidths. In addition to estimating time taken with our performance model, we also use the TC command (tc, 2020) to limit bandwidth on a real cluster, thereby verifying our performance model (the markers represent measurements on hardware). The figure shows that gradient compression is very useful in low bandwidth settings (≤ 8 Gbps). Although low bandwidths are uncommon in data centers (10 Gbps is minimum with a V100 GPU on Amazon EC2), this shows that in certain cases like wide-area learning (Bonawitz et al., 2019) gradient compression methods can be extremely useful. Several other insights and analysis from our performance model is in Appendix D.

5 Discussion

In this section we discuss the insights from our study as well as limitations and applicability of our findings.

Limitations Due to Hardware Changes. We have performed all our experiments in the standard data-center setting on AWS using *p3.8xlarge* instances which provide bandwidth of around 10Gbps. For training, we used NVIDIA V100 GPUs which are extensively used for DNN training. We believe that such a setup is very close to the typical setup used for distributed DNN training. All our insights are specific to this ubiquitous setup. However, if training moves to hardware with very different compute capabilities then our findings will also change.

Additionally, if gradient computation methods are designed such that they can be offloaded to auxiliary hardware like network interface cards or switches then potentially gradient compression can be overlapped with backward pass leading to better scalability than currently observed results. Similarly, if custom hardware is designed to perform gradient compression, that will also negate our findings on the utility of gradient compression. However, we believe that our analysis can guide hardware designers to understand additional functionalities which will be needed if they want to add support for gradient compression algorithms.

Gradient Compression for Different Parallelisation Strategies. In our work we have only analysed the extremely popular data parallel setting.However, recently PowerSGD was used by (Ramesh et al., 2021) to train DALL-E a 12 billion parameter version of GPT-3 for generating images from text descriptions. Ramesh et al. (2021) used PowerSGD to reduce communications among nodes for scalability. But due to huge size of the model, Ramesh et al. (2021) used a hybrid-parallel scheme, which combined data-parallelism and model-parallelism. In this approach the model was split in model-parallel mode within a machine and was using data-parallel mode to sync with other machines. This hybrid parallelism allowed Ramesh et al. (2021) to efficiently overlap encode/decode and communication operations as all other GPUs except one will be idle while doing backward pass on a given machine. However as we show in Section 3.1 this type of overlap is not possible in data-parallel setting. Analysing such hybrid-schemes for extremely large models is an avenue for future work.

Effect of Low Level Optimizations. For compression algorithms, we have used the best publicly available implementation of all the compression methods. For communication, we use NVIDIA-NCCL library which is specifically designed for NVIDIA GPUs and has been shown to achieve the best performance among communication libraries. However, it is possible that even better implementations of these compression algorithms can be created which may yield better results and may change the findings. We believe this is unlikely because existing algorithms like signSGD and PowerSGD already use heavily optimized CUDA primitives.

Enhanced Utility of Gradient Compression for Low Compute Density Workloads. Highly scalable syncSGD implementations (Li et al., 2020; Sergeev & Del Balso, 2018) rely on the overlap between communication and backward pass to provide high speedup. But if the compute density decreases then the amount of overlap possible will reduce, making it impossible to "hide" communication. An example of reduced compute density is small batch-size and we find gradient compression does indeed provide speedups for small batches(Section 3.3). However, recent work has focused on increasing the batch size (memory permitting) (Devarakonda et al., 2017; Yao et al., 2018) and designing algorithms to improve accuracy when using large batches.

6 Conclusion

In this work, we study several gradient compression methods used to accelerate distributed ML training. We discover that existing gradient compression methods provide marginal speedups in a datacenter setup due to the overheads in compression. We develop a performance model that can help algorithm designers build scalable gradient compression algorithms. Our performance model also allows users to conduct what-if analyses and determine how much compression they need given a hardware setup. We believe this analysis provides the community clarity on the desirable properties for gradient compression and will lead to methods that can provide improved scalability in the future.

Acknowledgement

We will like to thank the anonymous reviewers for their insightful comments. This work is supported by an award from the American Family Insurance Data Science Institute. Shivaram Venkataraman is also supported by the National Science Foundation grant CNS-1838733 and by the Office of the Vice Chancellor for Research and Graduate Education at University of Wisconsin-Madison with funding from the Wisconsin Alumni Research Foundation. Dimitris Papailipoulos acknowledges the support of NSF CAREER Award #1844951, Sony Faculty Innovation Award, AFOSR & AFRL Center of Excellence Award FA9550-18-1-0166, NSF TRIPODS Award #1740707,and ONR YIA Grant N00014-21-1-2806

References

Iperf-the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/iperf-doc.php. Accessed: December 10, 2020.

Massively scale your deep learning training with nccl 2.4. https://bit.ly/341nGfs. Accessed: December 10, 2020.

Bit2byte extension. https://github.com/jiaweizzhao/signSGD-with-Majority-Vote/tree/master/main/bit2byte-extension, 2020. Accessed: February 12, 2022.

Powersgd hook in pytorch. https://pytorch.org/docs/stable/ddp_comm_hooks.html#powersgd-communication-hook, 2020. Accessed: May 20, 2021.

tc - show / manipulate traffic control settings. https://man7.org/linux/man-pages/man8/tc.8.html, 2020. Accessed: May 25, 2021.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Acharya, J., De Sa, C., Foster, D., and Sridharan, K. Distributed learning with sublinear communication. In *International Conference on Machine Learning*, pp. 40–50. PMLR, 2019.

Aji, A. F. and Heafield, K. Sparse communication for distributed gradient descent. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 440–445, 2017.

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

Alistarh, D., Hoefler, T., Johansson, M., Konstantinov, N., Khirirat, S., and Renggli, C. The convergence of sparsified gradient methods. In *NeurIPS*, 2018.

Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pp. 63–74, 2010.

Barnett, M., Shuler, L., van De Geijn, R., Gupta, S., Payne, D. G., and Watts, J. Interprocessor collective communication library (intercom). In *Proceedings of IEEE Scalable High Performance Computing Conference*, pp. 357–364. IEEE, 1994.

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

Bernstein, J., Zhao, J., Azizzadenesheli, K., and Anandkumar, A. signsgd with majority vote is communication efficient and fault tolerant. In *International Conference on Learning Representations*, 2018b.

Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konečnỳ, J., Mazzocchi, S., McMahan, H. B., et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901, 2020.

Chen, Y., Griffith, R., Liu, J., Katz, R. H., and Joseph, A. D. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pp. 73–82, 2009.

Cho, M., Muthusamy, V., Nemanich, B., and Puri, R. Gradzip: Gradient compression using alternating matrix factorization for large-scale deep learning.

Coleman, C., Kang, D., Narayanan, D., Nardi, L., Zhao, T., Zhang, J., Bailis, P., Olukotun, K., Ré, C., and Zaharia, M. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *ACM SIGOPS Operating Systems Review*, 53(1):14–25, 2019.

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

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pp. 1223–1231, 2012b.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Dettmers, T. 8-bit approximations for parallelism in deep learning. *arXiv preprint arXiv:1511.04561*, 2015.

Devarakonda, A., Naumov, M., and Garland, M. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017.

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

Dryden, N., Moon, T., Jacobs, S. A., and Van Essen, B. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pp. 1–8. IEEE, 2016.

Fang, J., Fu, H., Yang, G., and Hsieh, C.-J. Redsync: reducing synchronization bandwidth for distributed deep learning training system. *Journal of Parallel and Distributed Computing*, 133:30–39, 2019.

Gandikota, V., Kane, D., Maity, R. K., and Mazumdar, A. vqsgd: Vector quantized stochastic gradient descent. In *International Conference on Artificial Intelligence and Statistics*, pp. 2197–2205. PMLR, 2021.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Grishchenko, D., Iutzeler, F., Malick, J., and Amini, M.-R. Asynchronous distributed learning with sparse communications and identification. 2018.

Grubic, D., Tam, L., Alistarh, D., and Zhang, C. Synchronous multi-GPU deep learning with low-precision communication: An experimental study. 2018.

Hoefler, T., Gropp, W., Kramer, W., and Snir, M. Performance modeling for systematic performance tuning. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. IEEE, 2011.

Horvath, S., Ho, C.-Y., Horvath, L., Sahu, A. N., Canini, M., and Richtarik, P. Natural compression for distributed deep learning. *arXiv preprint arXiv:1905.10988*, 2019.

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

Iandola, F. N., Moskewicz, M. W., Ashraf, K., and Keutzer, K. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2592–2600, 2016.

Ivkin, N., Rothchild, D., Ullah, E., Stoica, I., Arora, R., et al. Communication-efficient distributed sgd with sketching. In *NeurIPS*, 2019.

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924*, 2018a.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018b.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

Karimireddy, S. P., Rebjock, Q., Stich, S., and Jaggi, M. Error feedback fixes signsgd and other gradient compression schemes. In *International Conference on Machine Learning*, pp. 3252–3261, 2019.

Koloskova, A., Lin, T., Stich, S. U., and Jaggi, M. Decentralized deep learning with arbitrary communication compression. *arXiv preprint arXiv:1907.09356*, 2019a.

Koloskova, A., Stich, S., and Jaggi, M. Decentralized stochastic optimization and gossip algorithms with compressed communication. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 3478–3487. PMLR, 09–15 Jun 2019b. URL http://proceedings.mlr.press/v97/koloskova19a.html.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Li, Y., Park, J., Alian, M., Yuan, Y., Qu, Z., Pan, P., Wang, R., Schwing, A., Esmaeilzadeh, H., and Kim, N. S. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 175–188. IEEE, 2018.

Lian, X., Zhang, C., Zhang, H., Hsieh, C.-J., Zhang, W., and Liu, J. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. *arXiv preprint arXiv:1705.09056*, 2017.

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

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, B. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

M Abdelmoniem, A., Elzanaty, A., Alouini, M.-S., and Canini, M. An efficient statistical-based gradient compression technique for distributed training systems. *Proceedings of Machine Learning and Systems*, 3, 2021.

Mattson, P., Cheng, C., Coleman, C., Diamos, G., Micikevicius, P., Patterson, D., Tang, H., Wei, G.-Y., Bailis, P., Bittorf, V., et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

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

Mota, J. F., Xavier, J. M., Aguiar, P. M., and Püschel, M. D-admm: A communication-efficient distributed algorithm for separable optimization. *IEEE Transactions on Signal Processing*, 61(10):2718–2723, 2013.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

Nvidia-Nsight. Nsight systems. https://developer.nvidia.com/nsight-systems, 2021. Accessed: May 12, 2021.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pp. 8026–8037, 2019.

pytorch jit. Jit optimized. https://pytorch.org/docs/stable/jit.html, 2021. Accessed: May 12, 2021.

PytorchCommHooks. Ddp communication hooks. https://pytorch.org/docs/1.8.0/ddp_comm_hooks.html, 2021. Accessed: May 12, 2021.

Qi, H., Sparks, E. R., and Talwalkar, A. Paleo: A performance model for deep neural networks. In *Proceedings of the International Conference on Learning Representations*, 2017.

Rabenseifner, R. Optimization of collective reduction operations. In *International Conference on Computational Science*, pp. 1–9. Springer, 2004.

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., and Sutskever, I. Zero-shot text-to-image generation, 2021.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, 2020.

Sanders, P., Speck, J., and Träff, J. L. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

Sarvotham, S., Riedi, R., and Baraniuk, R. Connection-level analysis and modeling of network traffic. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pp. 99–103, 2001.

Sattler, F., Wiedemann, S., Müller, K.-R., and Samek, W. Robust and communication-efficient federated learning from non-iid data. *IEEE transactions on neural networks and learning systems*, 31(9):3400–3413, 2019a.

Sattler, F., Wiedemann, S., Müller, K.-R., and Samek, W. Sparse binary compression: Towards distributed deep learning with minimal communication. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2019b.

Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

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

Shi, S., Chu, X., Cheung, K. C., and See, S. Understanding top-k sparsification in distributed deep learning. *arXiv preprint arXiv:1911.08772*, 2019a.

Shi, S., Wang, Q., Zhao, K., Tang, Z., Wang, Y., Huang, X., and Chu, X. A distributed synchronous sgd algorithm with global top-k sparsification for low bandwidth networks. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2238–2247. IEEE, 2019b.

Shi, S., Zhou, X., Song, S., Wang, X., Zhu, Z., Huang, X., Jiang, X., Zhou, F., Guo, Z., Xie, L., et al. Towards scalable distributed training of deep learning on public cloud clusters. *Proceedings of Machine Learning and Systems*, 3, 2021.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Smith, S. L., Kindermans, P.-J., Ying, C., and Le, Q. V. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

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

Strom, N. Scalable distributed DNN training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

Sun, C., Qiu, X., Xu, Y., and Huang, X. How to fine-tune bert for text classification? In *China National Conference on Chinese Computational Linguistics*, pp. 194–206. Springer, 2019.

Suresh, A. T., Felix, X. Y., Kumar, S., and McMahan, H. B. Distributed mean estimation with limited communication. In *International Conference on Machine Learning*, pp. 3329–3337. PMLR, 2017.

Tang, H., Gan, S., Zhang, C., Zhang, T., and Liu, J. Communication compression for decentralized training. In *NeurIPS*, 2018a.

Tang, H., Lian, X., Yan, M., Zhang, C., and Liu, J. d^2: Decentralized training over decentralized data. In *International Conference on Machine Learning*, pp. 4848–4856. PMLR, 2018b.

Tang, H., Yu, C., Lian, X., Zhang, T., and Liu, J. Doublesqueeze: Parallel stochastic gradient descent with double-pass error-compensated compression. In *International Conference on Machine Learning*, pp. 6155–6165. PMLR, 2019.

Tang, Z., Shi, S., Chu, X., Wang, W., and Li, B. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

Ueno, Y. and Yokota, R. Exhaustive study of hierarchical allreduce patterns for large messages between gpus. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 430–439, 2019. doi: 10.1109/CCGRID.2019.00057.

Vogels, T., Karimireddy, S. P., and Jaggi, M. Powersgd: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems*, pp. 14236–14245, 2019.

Wang, H., Sievert, S., Liu, S., Charles, Z., Papailiopoulos, D., and Wright, S. Atomo: Communication-efficient learning via atomic sparsification. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 9850–9861. Curran Associates, Inc., 2018.

Wang, H., Agarwal, S., and Papailiopoulos, D. Pufferfish: Communication-efficient models at no extra cost. *Proceedings of Machine Learning and Systems*, 3, 2021.

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

Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pp. 1509–1519, 2017.

Wu, J., Huang, W., Huang, J., and Zhang, T. Error compensated quantized sgd and its applications to large-scale distributed optimization. In *International Conference on Machine Learning*, pp. 5325–5333. PMLR, 2018.

Xu, H., Ho, C.-Y., Abdelmoniem, A. M., Dutta, A., Bergou, E. H., Karatsenidis, K., Canini, M., and Kalnis, P. Compressed communication for distributed deep learning: Survey and quantitative evaluation, 2020a. URL http://hdl.handle.net/10754/662495.

Xu, H., Ho, C.-Y., Abdelmoniem, A. M., Dutta, A., Bergou, E. H., Karatsenidis, K., Canini, M., and Kalnis, P. Compressed communication for distributed deep learning: Survey and quantitative evaluation. Technical report, 2020b.

Yao, Z., Gholami, A., Arfeen, D., Liaw, R., Gonzalez, J., Keutzer, K., and Mahoney, M. Large batch size training of neural networks with adversarial training and second-order information. *arXiv preprint arXiv:1810.01021*, 2018.

You, Y., Gitman, I., and Ginsburg, B. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888*, 6, 2017.

You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., and Hsieh, C.-J. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

Yu, Y., Wu, J., and Huang, J. Exploring fast and communication-efficient algorithms in large-scale distributed networks. *arXiv preprint arXiv:1901.08924*, 2019.

Zhang, H., Li, J., Kara, K., Alistarh, D., Liu, J., and Zhang, C. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *International Conference on Machine Learning*, pp. 4035–4043, 2017.

Zhang, Z., Chang, C., Lin, H., Wang, Y., Arora, R., and Jin, X. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, pp. 8–13, 2020.

Zheng, S., Huang, Z., and Kwok, J. Communication-efficient distributed blockwise momentum sgd with error-feedback. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32, pp. 11450–11460. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/80c0e8c4457441901351e4abbcf8c75c-Paper.pdf.

A OVERLAP GRADIENT COMPRESSION WITH COMPUTATION

In this section, we include additional results in which we consider overlapping gradient compression with gradient computation. POWERSGD was recently implemented with overlap in PyTorch v1.8 (pyt, 2020). For integrating SIGNSGD and MSTOP-K we used the recently introduced DDP Communication hook (PytorchCommHooks, 2021) interface. The DDP Communication hook interface was recently added in PyTorch v1.8. Comparing Figure 6 with Figure 14, Figure 7 with Figure 15 and Figure 8 with Figure 16 we observe that overlapping gradient compression with gradient computation is slower compared to performing gradient compression post gradient computation. Therefore to consider the best case for gradient compression, in the main paper we only consider gradient compression being performed post backward pass. As discussed in the main paper this phenomenon can be primarily attributed to both compression and backward pass being compute intensive and thus competing for the same resources on the GPU, leading to an overall slowdown. Using Nsys (Nvidia-Nsight, 2021) we also verify that indeed our implementation is trying to overlap gradient compression with computation by running backward pass and gradient compression in separate CUDA streams. Figure 13 shows a snapshot of this. In Figure 13 we also observe that gradient compression stream blocks gradient computation stream due to compute intensive nature of gradient compression which is able to completely utilize the GPU.

B PERFORMANCE MODEL FOR GRADIENT COMPRESSION

In Section 4.1 we described our performance model for syncSGD with system optimizations. Here we describe our performance model for gradient compression.

From the perspective of performance, the scalability of a compression method depends on two main factors i) can the aggregation be performed using *all-reduce* ii) the encode decode time for compression. Table 2 classifies a number of gradient compression methods based on compatibility

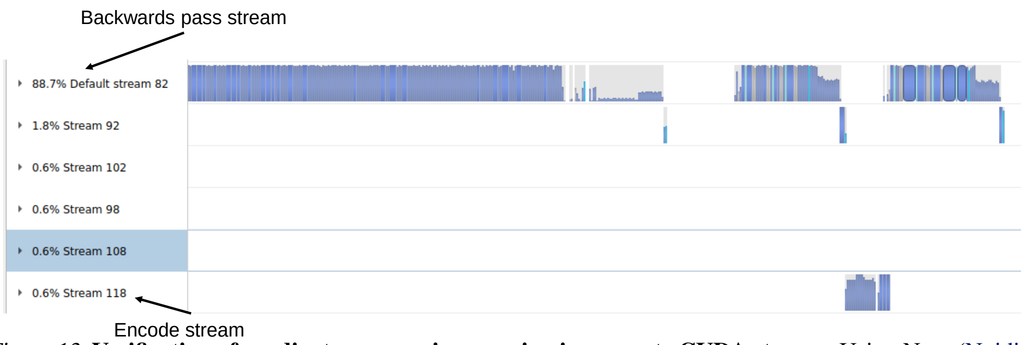
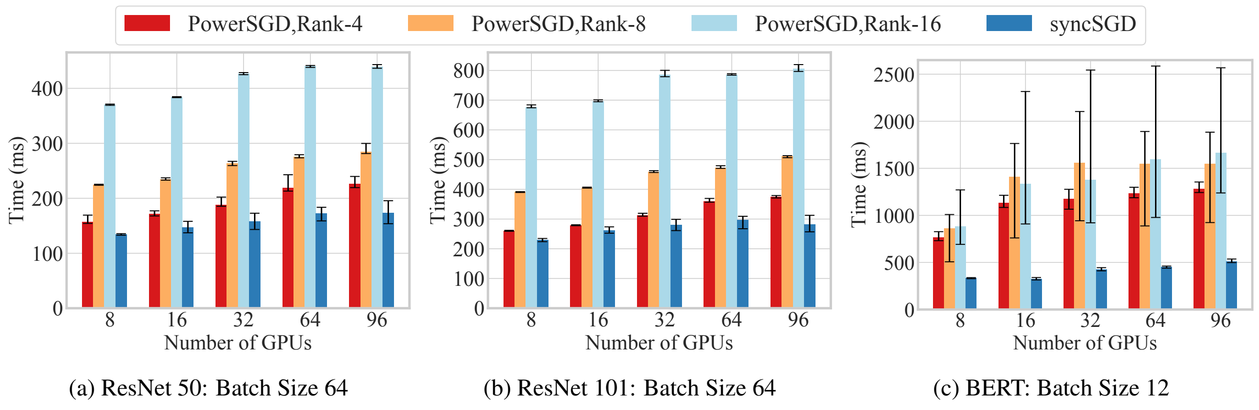


Figure 13. **Verification of gradient compression running in separate CUDA stream:** Using Nsys (Nvidia-Nsight, 2021) we verified that indeed our implementation which overlaps gradient computation and compression, tries to performs compression in separate CUDA stream. However, we observe that when encoding is being performed it completely blocks the backward pass stream. This due to the compute intensive nature of gradient compression which uses all the available compute units on the GPU.



(a) ResNet 50: Batch Size 64 (b) ResNet 101: Batch Size 64 (c) BERT: Batch Size 12

Figure 14. **Scalability of PowerSGD with overlap:** When POWERSGD is overlapped with backward we observe that it does not provide speedups in any of our experiments when compared against an optimized implementation of syncSGD.

with *all-reduce*. Ideally for high scalability we would like the method to be both *all-reduce* compatible and have low encode-decode time.

In Section 3.1 and Appendix A we have shown that the best case from the perspective of runtime will be performing gradient compression post backward pass. Based on this finding, a generic performance model will be

$$T_{obs} \approx T_{comp} + T_{encode-decode} + T_{comm}(\hat{b}, p, BW)$$

where T_{comp} is the time required for gradient computation, $T_{encode-decode}$ is the overhead of compressing and decompressing the gradients. Since after compression gradients are extremely small they are then sent in a single bucket, $T_{comm}(\hat{b}, p, BW)$ is the time required to communicate compressed gradients of size $\hat{b}$, across p GPUs at BW bandwidth. We now derive specific performance models for studying gradient compression schemes from the generic model stated above.

PowerSGD. POWERSGD requires sending two low rank matrices, P and Q. But $T_{encode-decode}$ as stated in Table 3 has high overhead. The performance model becomes-

$$\begin{aligned} T_{obs} \approx & T_{comp} + T_{encode-decode} + \\ & T_{comm}(P, p, BW) + T_{comm}(Q, p, BW) \end{aligned}$$

Where p is the number of GPUs, and T_{comm} is calculated using Equation 1.

MSTOP-K. For MSTOP-K the output of compression is the TOP-K% gradient values ($\hat{g}$) and their corresponding indices ($\hat{i}$). Further, TOP-K operator is not compatible with *all-reduce*, therefore we need to use *all-gather* collective, thus T_{comm} will be calculated from

$$T_{comm}(\hat{g}, p, BW) = 2\alpha + \frac{\hat{g} \times (p-1)}{BW}$$

where $\hat{g}$ is the gradient size, p is the number of GPUs. A similar calculation applies to $\hat{i}$ the indices. Overall the

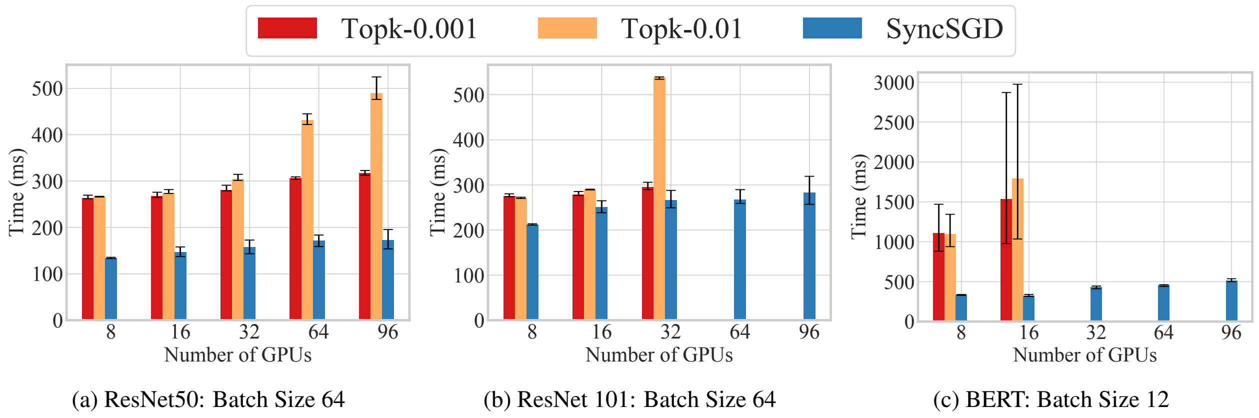


Figure 15. **Scalability of MSTop-K with overlap:** Comparing the time taken for gradient computation and aggregation for MSTop-K (with overlap) with syncSGD. For BERT and ResNet-101 we could not scale MSTop-K beyond 16 and 32 GPUs respectively, due to memory requirement of MSTop-K increasing linearly with number of machines and running out of available memory.

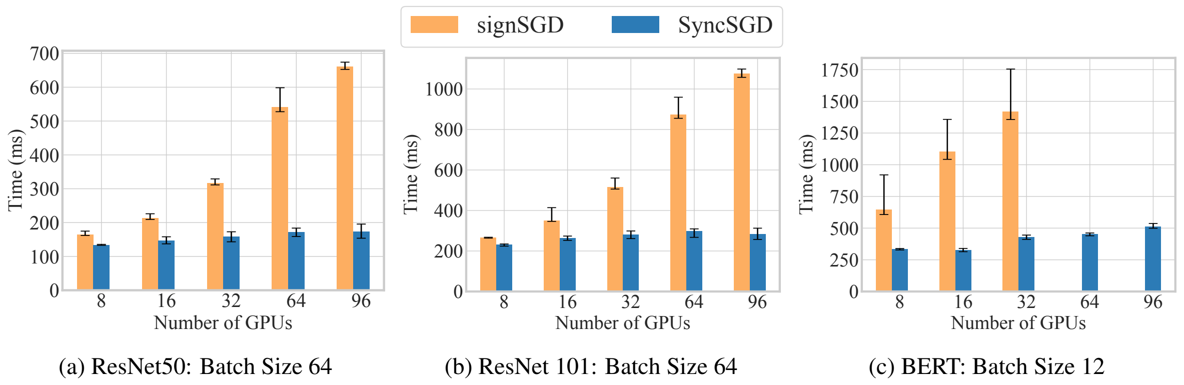


Figure 16. **Scalability of signSGD with overlap:** We compare the time taken for gradient computation and aggregation for signSGD with syncSGD. For BERT we could not scale signSGD beyond 32 GPUs, because the memory requirement of signSGD increase linearly with number of machines and for BERT we ran out available memory.

performance model becomes.

$$T_{obs} \approx T_{comp} + T_{encode-decode} + T_{comm}(\hat{g}, p, BW) + T_{comm}(\hat{i}, p, BW)$$

SIGNSGD. SignSGD, only sends 1bit for each 32bit leading to around 32× gradient compression. However SignSGD is not compatible with all-reduce leading to a performance model as follows:

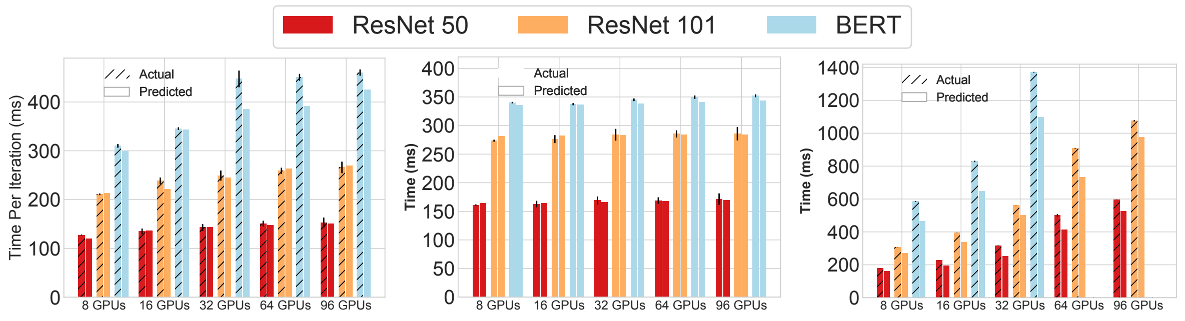
$$T_{obs} \approx T_{comp} + T_{encode-decode} + T_{comm}(\hat{g}, p, BW)$$

where $T_{comm}(\hat{g}, p, BW) = \frac{\hat{g}\times(p-1)}{BW}$ and $\hat{g} = \frac{g}{32}$. For SIGNSGD we only consider *all-gather* collective, *i.e.*, each node receives the encoded gradients from all other nodes.

C VERIFICATION PERFORMANCE MODEL

In this section we describe how we verify our performance model and calculate the values required for using our analytical performance model.

In case of syncSGD the backward pass and gradient synchronization are overlapped, therefore it is not easy to segregate the time spent in communication and time spent in computation. First we calculate just the time taken for backward pass on a single machine this forms T_{comp} in the performance model. To calculate γ, we run distributed training but with Nsight Systems profiling switched on. From Nsight systems we track kernels launched during backward pass and find how long does it takes for the compute phase of backward pass. The ratio between the two allows us to calculate γ. For all our experiments we disable NCCL auto tuning and forced it to use ring algorithm by setting the NCCL_TREE_THRESHOLD=0. To calculate $T_{encode-decode}$ we calculate the time required for compression and decompression for each iteration and plug it in the model. Before each run we calculate available bandwidth between each pair of instances using iperf3 (ipe) and take the minimum of these values as BW. For calculating α



(a) syncSGD (b) POWERSGD (c) SIGNSGD

Figure 17. **Evaluating our performance model on actual hardware:** We evaluate our performance model on AWS on p3.8xlarge instance. We observe that our performance model quite closely tracks the actual performance of both syncSGD implementation of PyTorch as well as performance of gradient compression methods. Before all experiments we calculated the available pairwise bandwidth using iperf3(ipe), and calculate the latency term by performing all reduce based on the vector of size equivalent to number of machines. For BERT we could not scale signSGD beyond 32 GPUs, because signSGD's memory requirement increase linearly with number of machines and for BERT we ran out available memory.

we perform ring-reduce on a small tensor and divide the obtained value by $(p-1)$ where p is the number of GPUs.

Figure 17 shows that our model closely tracks the experiments performed on real hardware. In case of syncSGD and POWERSGD (schemes using all-reduce) we observe the maximum deviation from actual experiments to be around 9.1%. In case of SIGNSGD the maximum deviation observed is 19.1%, the reason for high difference for SIGNSGD is that *all-gather* collective has an all to all pattern which causes degraded network performance due to widely reported issues of incast (Chen et al., 2009; Alizadeh et al., 2010). In future a utility which can simulate the traffic pattern of *all-gather* collective and provide us more accurate measurements of the effective bandwidth available during all-to-all communications can be helpful in providing better estimates of per iteration time.

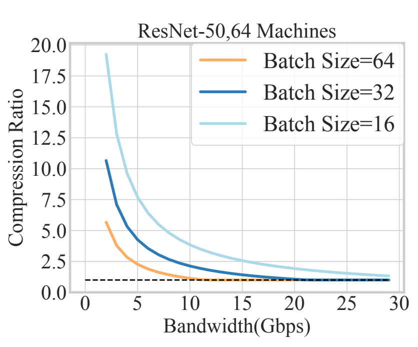
Using the Performance Model. To use the performance model, similar to verification we calculate T_{comp}, the time for backward pass on a single machine for a given batch size and model. It depends on hardware, computation requirements of the model and the batch size used for training. For gradient compression methods we also calculate $T_{encode-decode}$ for SIGNSGD, TOP-K and POWERSGD. We only include the computation time and disregard the time for extracting gradients, or copying back the decompressed gradients to the model. As these timings can be improved with tighter integration with the training frameworks. For this calculation we run each experiment 60 times and discard the first 10, we assign the mean of remaining 50 as $T_{encode-decode}$. Table 3 shows the times for T_{comp} and $T_{encode-decode}$ for ResNet-50 when using V100 GPU on AWS. Thus without running large scale experiments, practitioners and researchers can utilize our performance model to predict speedups when performing distributed training with and without using gradient compression.

D WHAT-IF ANALYSIS

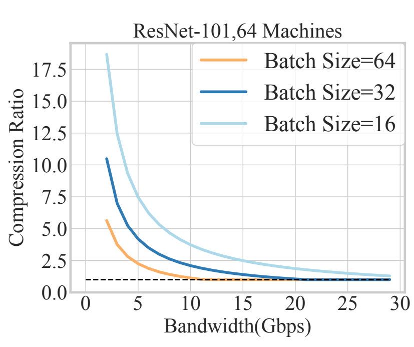
Our performance model also allows us to consider several what-if scenarios. To understand how and where gradient compression methods will be useful, we can vary several factors like compute availability, encode-decode time, network bandwidth etc. Based on our results in Section 3.2 which show that POWERSGD Rank-4 is the most scalable compression scheme, we use PowerSGD with Rank-4 as the baseline for these what-if analyses.

Required Compression for linear scaling. Existing gradient compression methods provide massive amount of compression which often leads to poor accuracy. Using our performance model we study the amount of gradient compression required for linear scaling. Figure 18 shows that in most common models at 10 Gbps we do not need compression greater than $4\times$. This shows that focus of gradient compression should be to reduce the overheads of compression rather than providing very high compression rates.

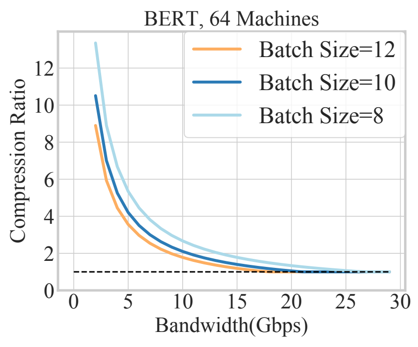
Effect of Network Bandwidth In Figure 19 we vary network bandwidth available from 1Gbps to 30Gbps and see how this changes the speedup offered by PowerSGD. We see that, for example, in the case of Resnet-50, PowerSGD offers considerable speedup at low network bandwidths (1-7 Gbps) but becomes slower than synchronous SGD when bandwidth available becomes $> 9Gbps$. This is due to the fact that syncSGD benefits more from availability of higher bandwidth since it communicates significantly more while PowerSGD is still limited by extra time spent in the encode-decode step. For BERT which is a communication heavy network, PowerSGD becomes slower than syncSGD at around 15Gbps. In Figure 19 the markers represent values



(a) ResNet50: 64 GPUs

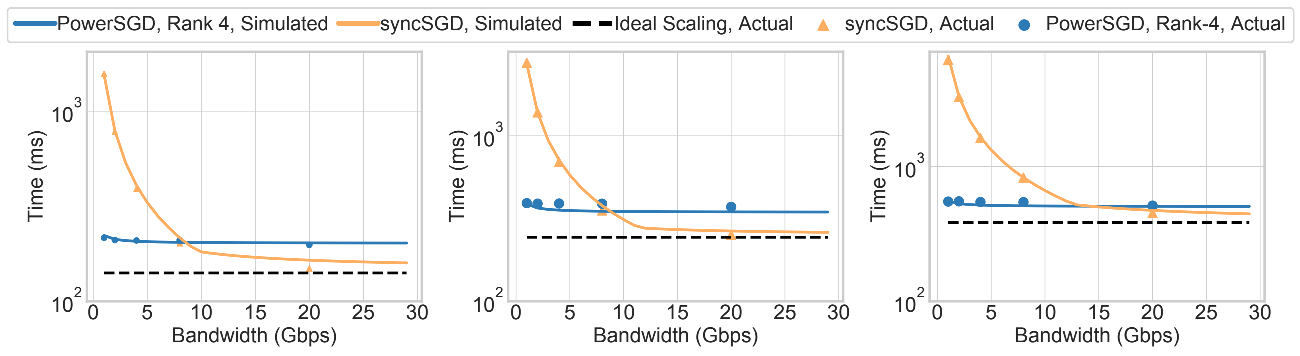


(b) ResNet101: 64 GPUs



(c) BERT: 64 GPUs

Figure 18. **Required gradient compression for near optimal speedups (simulated):** We observe that the required gradient compression for near optimal scaling is quite small. At 10 Gbps even for quite small batch sizes we need less than $4\times$ gradient compression, which is quite small compared to what popular gradient compression methods.



(a) ResNet50: Batch Size 64

(b) ResNet101: Batch Size 64

(c) BERT: Batch Size 12

Figure 19. **Evaluating effect of network bandwidth on training (simulated):** We vary bandwidth availability and analyse the performance of synchronous SGD vs PowerSGD Rank 4. We observe that as bandwidth increase significantly it helps synchronous SGD since it has a larger communication overhead. Moreover we observe the PowerSGD provides massive gains at extremely low bandwidth (1Gbps) but as bandwidth scales we see PowerSGD gets bounded by compute availability. The markers are values from actual experiments, this also shows how close our performance model is to actual measurement.

from actual experiments. To perform these experiments we used the *tc* command in linux to modify the available bandwidth. For experiments with bandwidth less than 10Gbps we used *p3.8xlarge* instances which provide a maximum of 10Gbps bandwidth. And for 20 Gbps experiment we used *p3.16xlarge* instance which provides 25 Gbps bandwidth. The markers are extremely close to the values from our analytical performance model thus verifying that our performance model can indeed be useful in several settings.

Effect of faster compute. Next we analyze how the effect of gradient compression changes when newer hardware with higher compute capabilities arrive in future.

In Figure 20, we plot the effect of compute capabilities improving by up to $4\times$, while network bandwidth remains constant at 10 Gbps. We can see that for Resnet-50, PowerSGD with Rank-4 can provide 1.75x speedup if the compute becomes around 3.5x faster.

There are two reasons for this, (i) As compute gets faster, the encode-decode time also reduces by the same factor, (ii) with a faster backward pass, there is less opportunity for synchronous SGD to overlap computation with communication, making it communication bound.

Tradeoff between encode-decode time and compression ratio. Finally, we explore the tradeoff between the effect of reducing encode-decode time, while simultaneously decreasing the compression ratios by similar proportions. For this we consider a hypothetical gradient compression scheme in which if we decrease encode-decode time by a factor k the size of gradients communicated increases by lk. For example, if say $k = 2$ and $l = 2$ then a 2x decrease in encode-decode time would be accompanied by a 4x increase in size of gradients. This setup is to study what would happen if we had compression schemes that offered a variety of trade-off points. We vary k from 1 to 4 in increments of 1 and try 1,2 and 3 as values of l. Using PowerSGD with

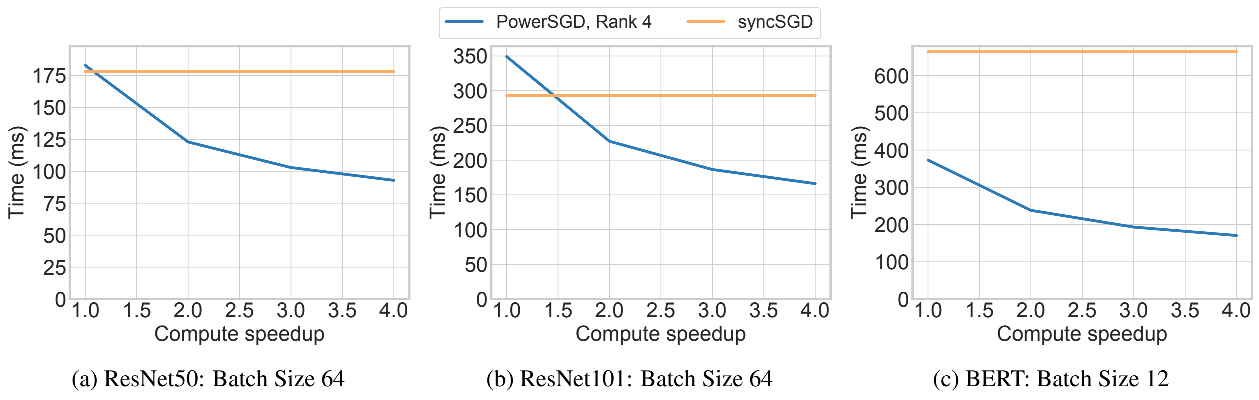


Figure 20. **Evaluating effect of compute speedup on training time (simulated):**Assuming network capacity remains at 10Gigabit but compute capabilities go up, we observe in that case PowerSGD will end up providing significant benefit, meanwhile synchronous SGD will end up being communication bound and will not be able to utilize increased compute. Showing that if compute capabilities increase drastically but network bandwidth remains stagnant, gradient compression methods will become useful.

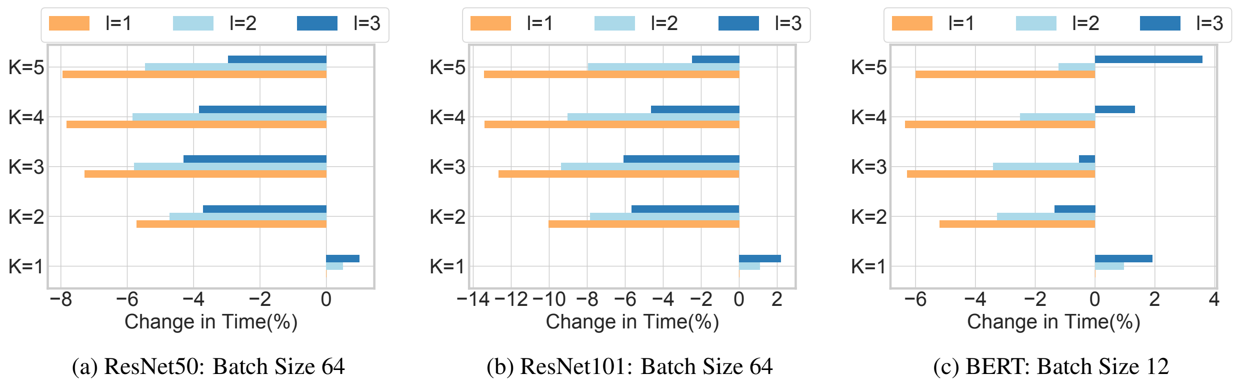


Figure 21. **Varying encoding-decoding time and compression (simulated) :** We observe that reducing encode-decode time even if it leads to reduced gradient compression is very useful and can make methods like PowerSGD more viable.

Rank-4 as the baseline, we see in Figure 21 that any reduction in encode-decode time even at the expense of increased communication helps.

E Artifact Appendix

E.1 Abstract:

We provide the artifacts to reproduce all the results in the paper. For all our experiments we use PyTorch v1.8.1+cu111, the code uses Python 3.8 . We have used AWS *p3.8xlarge* instances to run experiments. For easy reproducibility we are also providing a public AMI - *ami-0eea3ad7fabaa0125*

E.2 Artifact check-list (meta-information)

- **Algorithm:** All the gradient compression algorithms are implemented in *gradient_reducers.py*
- **Compilation:** The code is in Python, therefore requiring no compilation. However for replicating signSGD one will need to install the bit2byte extension (bit, 2020). The provided AMI has the extension pre-installed.
- **Data set:** We use two datasets for evaluation Imagenet and Sogou News datasets.
- **Run-time environment:** For all our experiments we ran Ubuntu Linux, with Cuda 11.1 .
- **Hardware:** We used p3.8xlarge instances on AWS for all our experiments. We run scaling from 2 p3.8xlarge to upto 24 p3.8xlarge.
- **Metrics:** For all the experiments we collect per-iteration time for training.
- **Output:** Our existing code writes all files locally on disk as well as to AWS S3
- **How much disk space required (approximately)?:** Around 1 Terabyte of disk space.
- **How much time is needed to prepare workflow (approximately)?:** To get data and setup all the experiments one would ideally require 2-3 hrs. If the evaluators have access to AWS, the authors have also provided a public AMI - *ami-0eea3ad7fabaa0125* which has the datasets an dependencies pre-installed.
- **How much time is needed to complete experiments (approximately)?:** Ideally it should take less than an hour to run and get all the numbers.
- **Publicly available?:** Yes. Apart from Github we are have also archived our code on Figshare - `https://figshare.com/s/f0c3e00293b0690f76c1`
- **Data licenses (if publicly available)?:** We use Imagenet and Sogou News datasets which come with their own licenses.

E.3 Description

We provide the code to reproduce all the experiments in this paper. The authors are providing a github repository with all the documentation to run the experiments.

E.3.1 How delivered

All our code is present on the github repository: `https://github.com/uw-mad-dash/GradCompressionUtility`. For easy setup on AWS we also provide a public ami - *ami-0eea3ad7fabaa0125*, which can be used to launch large scale experiments.

E.3.2 Hardware dependencies

For all our experiments we used *p3.8xlarge* instances on AWS. We run scaling experiments from 2 p3.8xlarge instances to upto 24 p3.8xlarge. Atleast 2 p3.8xlarge instances are needed to replicate a portion of our results.

E.3.3 Software dependencies

We need standard Pytorch 1.8.1+cu111 installed with dependencies like Numpy. For easy installation and setup we have also provided a public ami - *ami-0eea3ad7fabaa0125*.

E.3.4 Data sets

For all our experiments we used Imagenet and Sogou News dataset. The instructions to download those are available in the Readme. All the datasets are already available in the public AMI.

E.4 Experiment workflow

We have provided a detailed Readme which provides bash scripts to automatically run the experiments.

E.5 Experiment customization

The experiment can be customized by trying on different hardware setups. One example for this will be to run these experiments on slower GPUs but with faster interconnects. Another option would be trying a smaller batch size, which will increase the number of synchronizations.