DOES COMPRESSING ACTIVATIONS HELP MODEL PARALLEL TRAINING?

Song Bian [*1] **Dacheng Li** [*2] **Hongyi Wang** [2] **Eric P. Xing** [2 3 4] **Shivaram Venkataraman** [1]

ABSTRACT

Foundation models have superior performance across a wide array of machine learning tasks. The training of these models typically involves model parallelism (MP) to navigate the constraints of GPU memory capacity. However, MP strategies involve transmitting model activations between GPUs, which can hinder training speed in large clusters. Previous research has examined gradient compression in data-parallel contexts, but its applicability in MP settings remains largely unexplored.

In this paper, we investigate the unique characteristics of compression in MP and study why strategies from gradient compression might not be directly applicable to MP scenarios. Subsequently, to systematically understand the capabilities and limitations of Model Parallelism Compression, we present a benchmarking framework **MCBench**. MCBench not only includes four major categories of compression algorithms but also includes several widely used models spanning language and vision tasks on a well-established distributed training framework, Megatron-LM. We initiate the first comprehensive empirical study by using MCBench. Our empirical study encompasses both the fine-tuning and pre-training of FMs. We probe over 200 unique training configurations and present results using 10 widely used datasets. To comprehend the scalability of compression advantages with the expansion of model size and cluster size, we propose a novel cost model designed specifically for training with MP compression. The insights derived from our findings can help direct the future development of new MP compression algorithms for distributed training. Our code is available at https://github.com/uw-mad-dash/MCBench

1 INTRODUCTION

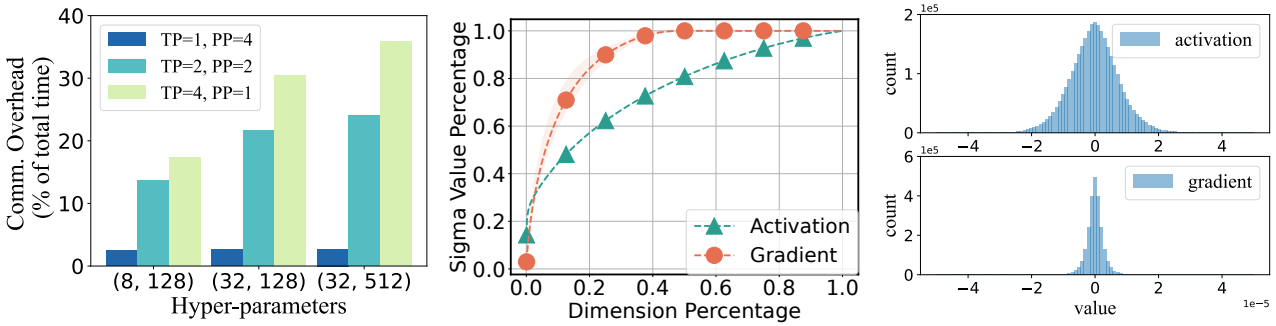
Foundation models (FMs) have taken center stage for a multitude of machine learning tasks given their exceptional performance across various benchmarks (Zhang et al., 2022; Liang et al., 2022; Touvron et al., 2023). Nevertheless, these state-of-the-art FMs have a staggering volume of parameters, making it a challenge for a single GPU to hold the entire model (Brown et al., 2020; Radford et al., 2021; Bommasani et al., 2021). For instance, models such as GPT-3 (Brown et al., 2020) and OPT (Zhang et al., 2022) have up to 175 billion parameters requiring over 700GB of GPU memory. Consequently, pre-training of FMs often involves distributing model parameters across multiple GPUs, a process known as *model parallelism* (MP) (Jia et al., 2019; Zhuang et al., 2022; Zheng et al., 2022). Pipeline model parallelism, a recognized MP technique, allocates the layers of an FM across multiple GPUs (Narayanan et al., 2019; Huang et al., 2019), while tensor model parallelism, another popular MP strategy, partitions neurons or elements within each FM layer (Shoeybi et al., 2019). These MP strategies, both tensor and pipeline model parallelism, have proved effective in minimizing the memory footprint of FM training (Shoeybi et al., 2019). However, the combination of these MP strategies often leads to considerable communication overhead, as illustrated in Figure 1(a). For instance, a frequently used tensor model parallelism strategy requires two all-reduce communications with significant message sizes for every Transformer layer in each iteration. This communication overhead can severely slow down the FM pre-training and fine-tuning (Shoeybi et al., 2019).

To address the high communication overhead in MP, one solution is to *compress* the messages exchanged among GPUs, such as activation values. Numerous prior works in the data-parallel setting have explored gradient compression to reduce the communication costs of training (Bernstein et al., 2018; Lin et al., 2017; Wang et al., 2018b; Vogels et al., 2019; Agarwal et al., 2021; Song et al., 2023).

However, **compression in MP is fundamentally different from that in data parallelism**. Firstly, gradients exhibit a low-rank nature, while activations do not (Figure 1(b)). Consequently, low-rank gradient compression methods, which have been shown to deliver end-to-end speedup in data-parallel training, may not be directly applicable to MP (Vogels et al., 2019). Second, gradients and activations have

[*]Equal contribution [1]Department of Computer Science, University of Wisconsin-Madison [2]Machine Learning Department, Carnegie Mellon University [3]MBZUAI [4]Petuum Inc.. Correspondence to: Song Bian <songbian@cs.wisc.edu>.

Proceedings of the 7^{th} MLSys Conference, Santa Clara, CA, USA, 2024.



(a) Communication Analysis (b) Low-Rank Analysis (c) Distribution

Figure 1. Figure (a) illustrates the communication overhead of model parallelism on $\text{BERT}_{\text{LAREG}}$ across 4 GPUs, with varying batch sizes and sequence lengths. The x-axis represents the combination of batch size and sequence length. In Figure (b), curves are plotted based on the ordered singular values from the SVD decomposition, revealing that while the gradient is low-rank, the activation is not. The activation corresponds to the output of the 12^{th} transformer layer in the $\text{BERT}_{\text{LARGE}}$ model. Figure (c) examines the element distribution of activation and gradient for the $\text{BERT}_{\text{LARGE}}$ model.

considerably different distributions as shown in Figure 1(c). Thus, sparsification-based methods may also not be suitable for MP settings (Han et al., 2015).

While recent research (Wang et al., 2022) has shown the promise of quantization-based compression for pipeline parallel training on wide-area networks, it remains unclear which compression algorithms would work well for other parallelism strategies (e.g., tensor parallelism) or in environments with higher bandwidth. Furthermore, as model sizes (OpenAI, 2023) and cluster sizes (Zhang et al., 2022) rapidly increase, practitioners need tools that can answer questions like: "Will using quantization lead to better performance compared to Top-K?" or "What will be the throughput benefits if we use a cluster twice as large with NVLink?"

To address the above challenge, we aim to create a representative benchmarking tool that can fairly evaluates MP compression algorithms and meet the following goals: (1) Allow ML researchers to easily implement new compression algorithms and test it with a wide range of datasets and models (2) Allow developers to easily compare compression methods in different MP settings ; (3) Handle both fine-tuning and pre-training tasks (4) Enable ML practitioners to ask what-if questions to evaluate if MP compression will be effective at scale.

We propose **MCBench**, a benchmarking tool that builds up on Megtraon-LM (Shoeybi et al., 2019) and develops APIs for implementing and evaluating MP compression. As a part of MCBench, we implement quantization-based compression method (Alistarh et al., 2017; Bernstein et al., 2018), low-rank and sparsification-based gradient compression techniques such as PowerSGD (Vogels et al., 2019), Top-K (Alistarh et al., 2018; Aji & Heafield, 2017), and Random-K (Xu et al., 2020; Stich et al., 2018), along with a learning-based compression approach, *i.e.,* auto-encoders (AEs) (Hinton & Zemel, 1993). We then show examples on how these algorithms can be integrated with several popular datasets such as CIFAR-10, and CIFAR-100 and models such as $\text{BERT}_{\text{BASE}}$, ViT-Base, OPT, and XLM-RoBERTa-XL from HuggingFace (Wolf et al., 2020). In addition, this benchmark can also be extended to integrate recent models such as LLaMA (Touvron et al., 2023) and Code LLaMA (Rozière et al., 2023).

Using MCBench, we present the first comprehensive study of achieving communication efficiency using compression methods in MP settings. Our experiments cover both pre-training and fine-tuning tasks, assessing the impact of various compression methods on throughput and accuracy. In total, we evaluate compression methods across over 200 unique configurations, employing various compression algorithms, training stages, models, and more than 10 datasets (Krizhevsky et al., 2009; Wang et al., 2018a). We also develop the first cost model for MP with activation compression, validate it with experimental data (Section 4.4), and use it to derive insights into MP compression at scale. Our cost model and experiments reveal the following **key takeaways**:

1. Learning-based compression methods are most suitable for MP. In the pre-training stage, only learning-based methods (AEs) offer speedup (up to **16%**) and preserve the model's accuracy. Sparsification-based methods (Top-K) improve training time but compromise accuracy. Quantization methods maintain the model's accuracy but slow down the training time. Low-rank methods (PowerSGD) slow down training time and degrade accuracy. In the fine-tuning stage, all evaluated compression methods fail to provide end-to-end speedup, because their encoding and decoding overhead outweighs the reduced communication time. We provide further analysis in § 4.2 and § 4.3.

2. MP compression can yield around 25% throughput improvements for large models when cluster size is pro-

portionally scaled. Using our cost model for MP with activation compression, we find that when we perform weak scaling (Narayanan et al., 2021b) (*i.e.,* we increase cluster size proportional to model size), AE-based compression can yield around 25% throughput improvements for models with 100s of billions of parameters (Brown et al., 2020).

3. MP compression should be avoided for challenging ML tasks. Our findings reveal that employing MP compression can significantly compromise model accuracy, especially in the context of challenging ML tasks, *e.g.,* CIFAR-100 is more challenging than CIFAR-10 (Section § 4.2 has an in-depth discussion). Moreover, FMs like ViT-Base, when exposed to MP compression, can deliver performance worse than smaller models such as ResNet-50. As a result, we recommend that for challenging ML tasks, FMs should be trained without applying MP compression.

In conclusion, we have four key contributions: ❶ We establish a comprehensive benchmark MCBench that covers diverse tasks, models, and compression methods to meet design principles. ❷ By using MCBench, we carry out the first empirical study on MP compression for FMs, taking into account various compression methods, training stages, downstream tasks, and models. We extensively evaluate four major categories of compression algorithms across over 200 different settings and 10 popular datasets. ❸ We develop the first general cost model for MP with activation compression, demonstrating its correctness using experimental results and analyzing speedup as model size and cluster size scale; Our findings offer valuable insights for future MP compression research. ❹ We make MCBench available as an open-source project.

2 Background

In this section we first provide the necessary background about data parallelism and model parallelism first. Then, we discuss the need for developing a new benchmark for model parallelism compression.

Data Parallelism (DP). DP divides the training examples among multiple workers (Li et al., 2014; Ho et al., 2013) and replicates the model at each worker. During each iteration, each worker calculates the model gradient based on its assigned examples and then synchronizes the gradient with the other workers (Sergeev & Del Balso, 2018). However, DP requires each worker to compute and synchronize gradients for the entire model, which can become challenging as the model size increases. One issue is that the large gradients can create a communication bottleneck, and several previous studies have proposed gradient compression methods (Seide et al., 2014; Bernstein et al., 2018; Dettmers, 2015; Lin et al., 2017; Wang et al., 2018b) to address this. However, gradient compression methods have moderate performance improvement or even lead to an overall slowdown in DP due to the non-negligible compression overheads (Agarwal et al., 2022; Wang et al.). Furthermore, the worker may not have enough memory to train with the entire model, in which case model parallelism may be necessary.

Model Parallelism (MP). Model parallelism (MP) divides the model among multiple workers, allowing large models to be trained by only requiring each worker to maintain a portion of the entire model in memory. There are two main paradigms for MP: inter-layer pipeline model parallelism (PP) and intra-layer tensor model parallelism (TP). PP divides the layers among workers, with each worker executing the forward and backward computations in a pipelined fashion across different training examples (Narayanan et al., 2019; 2021a). For example, a mini-batch of training examples can be partitioned into smaller micro-batches (Huang et al., 2019), with the forward computation of the first micro-batch taking place on one worker while the forward computation of the second micro-batch happens on another worker in parallel. PP involves a communication overhead due to the point-to-point communication between workers. TP (Lu et al., 2017; Shazeer et al., 2018; Kim et al., 2016) divides the tensor computations among workers. In particular, we consider a specialized strategy developed for Transformer models that divides the two GEMM layers in the attention module column-wise and then row-wise, with the same partitioning applied to the MLP module (Shoeybi et al., 2019; Narayanan et al., 2021b). However, TP still involves a communication overhead due to the need for two all-to-all collective operations in each layer, motivating the use of compression to reduce the communication overhead of MP (Shoeybi et al., 2019).

Benchmarking Model Parallelism Compression. Compared with data parallelism compression (Agarwal et al., 2022), our community has a limited understanding of model parallelism compression. This is because we don't have tools that can answer questions about which compression algorithms are more effective and at what scale should one use compression. Therefore, we aim to fill this gap by developing a model parallelism compression benchmark (called **MCBench**) based on Megatron-LM.

3 MCBench Design and Implementation

In this section, we first describe the interface and implementation of compression algorithms in MCBench. Then, we introduce how we build a connection between MCBench and Hugging Face to enable loading models and datasets. Finally, in order to answer what-if questions in terms of scale, we describe an analytical cost model that we develop as a part of MCBench.

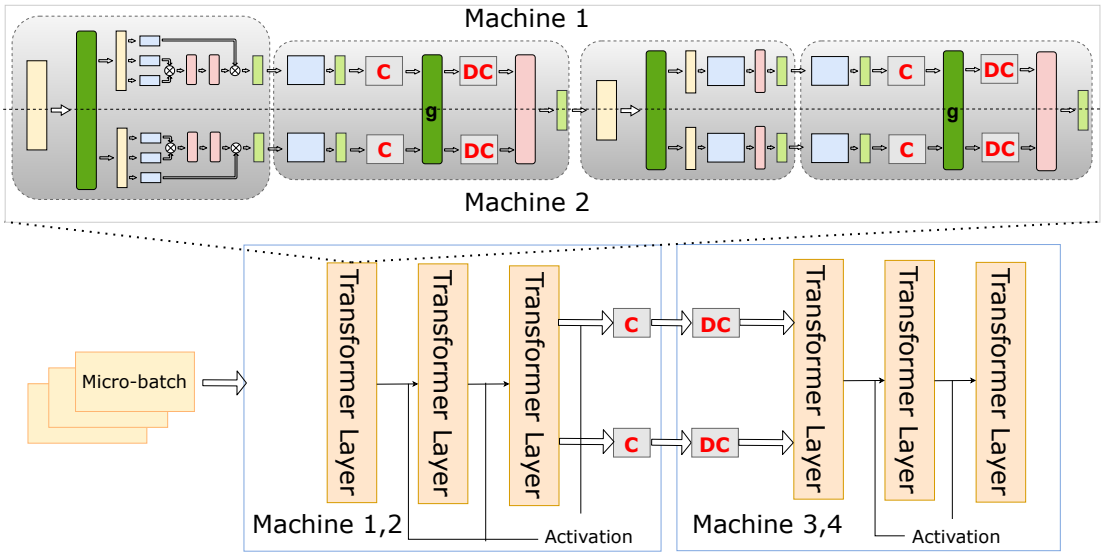


Figure 2. Illustration of compression on a 6-Layer Transformer model with 4 machines. Machine 1 and Machine 2 maintain the first three layers according to the TP strategy (pipeline stage 1). g stands for an all-reduce operation in the forward pass. A compression method C is used to reduce the message size for the all-reduce operation to reduce TP communication time. Correspondingly, a de-compression method DC is used after the communication.

3.1 Model Parallelism Compression Algorithms

In order to enable ML researchers to design and evaluate new compression algorithms, we develop a new API that integrates with Megatron-LM. Our API design ensures that the algorithms can be used in both tensor parallel and pipeline parallel settings. To ascertain the generality of our API, we implement a range of representative compression methods including low-rank-based approaches, sparsification-based approaches, learning-based approaches, and quantization-based approaches (as illustrated in Figure 2).

Our implementation includes:

- For learning-based approach (AEs), which compress messages using a pair of small neural networks (Hinton & Zemel, 1993), we multiply the activation using a learnable matrix $W_e \in \mathbb{R}^{h \times h_c}$ (the encoder) before the all-reduce step, where $h_c < h$ is the compressed size. After the all-reduce step, another learnable matrix of dimension $W_d \in \mathbb{R}^{h_c \times h}$ (the decoder) is used to decompress the compressed activation.

- For the sparsification-based approaches (Top-K and Random-K), we use `torch.topk` function to select the k largest absolute values of the activation and `random.sample` function to randomly select k values from the activation respectively (Stich et al., 2018).

- For the low-rank-based approach (PowerSGD), we follow the algorithm mentioned in (Vogels et al., 2019) and use `torch.linalg.qr` to do orthogonalize operation.

- Our implementation of the quantization-based approaches is based on the code released by (Wang et al., 2022):

$$A^Q = \lceil \frac{A}{\Delta} \rfloor, \quad \Delta = \frac{\max(|A|)}{2^{N_b - 1} - 1}$$

where A is the activation, A^Q is the quantized activation, Δ is the quantization step size, $\lceil \cdot \rfloor$ is the rounding function, and N_b is the number of bits. We omit some implementation details here due to the limited space. Please refer to Appendix B for details.

3.2 Adding Datasets and Models

Existing code in Megatron-LM (@898a89) only provides three transformer-based models (BERT$_{\text{LARGE}}$, GPT, and T5). and a few language datasets for fine-tuning tasks. To meet our benckmark's design goal, we build a connection between Megatron-LM and Hugging Face (Wolf et al., 2020) to enable the following features: (1) Load datasets from Hugging Face; (2) Implement the Hugging Face models by using functions provided by Megatron-LM; (3) Convert Hugging Face checkpoints into Megatron-LM format; (4) Split the checkpoints according to MP settings.

We consider image classification with the ViT-Base model as an example to explain the above steps in detail. First, we use the `load_dataset` function from Hugging Face to load CIFAR-10 and CIFAR-100 datasets. Second, we implement the following three parts of transformer-based models: (1) preprocessing; (2) transformer; (3) post-processing. For the preprocessing step, we transform the input into an embedding format that the model can utilize. Subsequently,

we employ the `ParallelTransformer` function from Megatron-LM to implement the transformer layers of the model. The `ParallelTransformer` function allows us to train the models under various MP settings. Finally, the post-processing part of ViT-Base is a simple linear layer.

To handle different checkpoint formats, and given that a model's checkpoint can be saved as a dictionary, we transform the Hugging Face checkpoint format to the Megatron-LM format by reordering the keys and values in the dictionary. Then, we split the checkpoints into several files based on MP settings such that they can be loaded into Megatron-LM for fine-tuning tasks.

3.3 Analytical Cost Model

While our implementation of MP compression algorithms can be used to measure accuracy and performance with empirical experiments, in MCBench, we also aim to help developers understand how the effects of compression will change as we scale model size and cluster size. In order to minimize the amount of time and resources required to answer scaling questions, we develop a cost model that captures the speedup from the above-named compression methods under various settings.

To develop our cost model, we consider the model parallelism scaling strategy developed in (Narayanan et al., 2021b). Concretely, we use tensor model parallelism in the same node, and pipeline model parallelism across nodes. We build a performance model for MP compression for real-world settings similar to (Narayanan et al., 2019) in two steps. First, we develop our cost model on a single-node, and analyze how costs change as we scale up the model size on a single node. Second, we increase the cluster size and, according to the model-parallelism strategy we choose, we assign additional GPUs to pipeline parallelism, and use off-the-shelf pipeline parallelism cost models to predict the performance (Li et al., 2022b; Zheng et al., 2022).

Denote the vocabulary size as V, hidden size as h, sequence length as s, and batch size as B. From (Narayanan et al., 2021b), we know that the number of floating points operations (FLOPs) and all-reduce message size in a Transformer layer is $96Bsh^2+16Bs^2h+6BshV$, and Bsh respectively.

Cost Model on Single Node. Without compression, the total time of a Transformer layer can be modeled as a sum of the all-reduce communication step and the computation time step. These two steps can not overlap because the all-reduce communication depends on the computational results:

$$T = T_{comp}(96Bsh^2 + 16Bs^2h + 6BshV) + T_{comm}(Bsh) \quad (1)$$

Let X be the compression method and the all-reduce message size after compression be M_c. The total time of a single Transformer layer with model parallelism compression can be written as:

$$T_X = T_{comp}(96Bsh^2 + 16Bs^2h + 6BshV) + T_{comm}(M_c) + T_{overhead} \quad (2)$$

where $T_{overhead}$ is the computation time of the compression algorithm. The speedup with L transformer layers on the single node is $\frac{L\times T}{L\times T_X} = T/T_X$.

Scaling Up the Cluster Size. Next, we analyze the speedup when scaling up the cluster size by combining the pipeline parallelism cost model developed in (Li et al., 2022b; Zheng et al., 2022). Formally, the running time is modeled as a sum of per-micro-batch pipeline communication time, per-micro-batch of non-straggler pipeline execution time, and the *per-mini-batch* straggler pipeline execution time. To use the cost model, we denote the number of micro-batches as m, the number of nodes (the cluster size) n, the number of layers L, the pipeline communication time p or p_C.

We use the default pipeline layer assignment strategy in (Shoeybi et al., 2019), which balances the number of transformer layers. Thus, every pipeline stage requires the same amount of time: $\frac{L}{n}T$ or $\frac{L}{n}T_X$. We use the pipeline communication model in (Jia et al., 2019; Li et al., 2022b), $p = \frac{Bsh}{w}$, $p_X = \frac{M_c}{w}$, where w is the bandwidth. Thus the overall speedup can be written as:

$$\frac{(\frac{m-1}{n}+1)\times L\times T + (n-1)\times\frac{Bsh}{w}}{(\frac{m-1}{n}+1)\times L\times T_X + (n-1)\times\frac{M_c}{w}} \quad (3)$$

In Section 4, we validate the cost model and analyze the implications.

4 Performance Analysis With MCBench

In this section, we present the first comprehensive study of model parallelism compression by using MCBench to answer the following questions:

- What is the impact of activation compression on system throughput and which compression method achieves the best throughput?
- What is the impact of compression on model accuracy? How do different downstream tasks affect the performance of compression methods?
- What happens when we scale up the model size and the cluster size?

We answer these questions in the context of two commonly used scenarios: fine-tuning on the GLUE benchmark (Wang

et al., 2018a), CIFAR-10, CIFAR-100 (Krizhevsky et al., 2009), and pre-training on the Wikipedia (Devlin et al., 2018) and the BooksCorpus (Zhu et al., 2015) datasets.

4.1 Experimental Setup

We next describe the system configuration used, models tested, and other experiment settings.

System Configuration. In default, we use AWS p3.8xlarge instances where each instance is equipped with 4 V100 GPUs, 10 Gbps interconnect bandwidth, and NVLink in each instance.

Models. We use the $\text{BERT}_{\text{LARGE}}$ model provided by Megatron-LM (Shoeybi et al., 2019) which has 345M parameters. We configure the model to have 24 layers with each layer having a hidden size of 1024 and 16 attention heads. Moreover, we also integrate $\text{BERT}_{\text{BASE}}$, ViT-Base (Dosovitskiy et al., 2020), XLM-RoBERTa-XL (Duplyakin et al., 2019), and OPT-3B (Zhang et al., 2022) into Megatron-LM (Shoeybi et al., 2019). Both $\text{BERT}_{\text{BASE}}$ and ViT-Base have 12 layers with each layer having a hidden size of 768 and 12 attention heads. Additionally, XLM-RoBERTa-XL has 36 layers, each with a hidden size of 2560 and 32 attention heads. Similarly, OPT-3B has 32 layers, each also having a hidden size of 2560 and 32 attention heads. We use fp16 training in all experiments.

Experimental Settings. For fine-tuning, we follow similar settings in previous studies (Devlin et al., 2018; Liu et al., 2019; Dosovitskiy et al., 2020; Li et al., 2022a). We use a micro-batch size of 32 and a sequence length of 512 for $\text{BERT}_{\text{LARGE}}$, a micro-batch size of 32 and a sequence length of 128 for $\text{BERT}_{\text{BASE}}$, and a micro-batch size of 512 with a patch size of 32 for ViT-Base. Unless otherwise specified, the model evaluated is $\text{BERT}_{\text{LARGE}}$. We use 4 GPUs (in one machine) for fine-tuning. We vary the tensor model-parallel size and the pipeline model-parallel size across the following three parallelism degrees: $\{(1,4),(2,2),(4,1)\}$, where the first number of the tuple represents the tensor model-parallel degree and the second number of the tuple stands for the pipeline model-parallel degree. For pre-training, we use 4 machines with 16 GPUs in total. We use the recipe from (Izsak et al., 2021) which uses large batch size with shorter sequence length. We set micro-batch size 128, global batch size 1024, and sequence length 128. To study the impact of the distributed settings, we use the following three different parallelism degrees: $\{(2,8),(4,4),(8,2)\}$.

Hyperparameters. We also evaluate compression algorithms with different hyper-parameters. For AE, we vary compression dimension between $\{50, 100\}$. For Top-K and Random-K algorithms, we keep the same compression *ratio* as AE (*i.e.,* we compress the activation around 10 and 20 times). We evaluate quantization with $\{2, 4\}$ bits. Due to the instability of PowerSGD under FP16 training, we used FP64 to execute the PowerSGD algorithm.

Notation	Description
A1	AE with encoder output dimension 50
A2	AE with encoder output dimension 100
T1	Top-K: same comp. ratio as A1
T2	Top-K: same comp. ratio as A2
R1	Rand-K: same comp. ratio as A1
R2	Rand-K: same comp. ratio as A2
Q1	Quantization: reduce the precision to 2 bits
Q2	Quantization: reduce the precision to 4 bits
P1	PowerSGD: same comp. ratio as A1
P2	PowerSGD: same comp. ratio as A2
TP	Tensor model-parallelism degree
PP	Pipeline model-parallelism degree

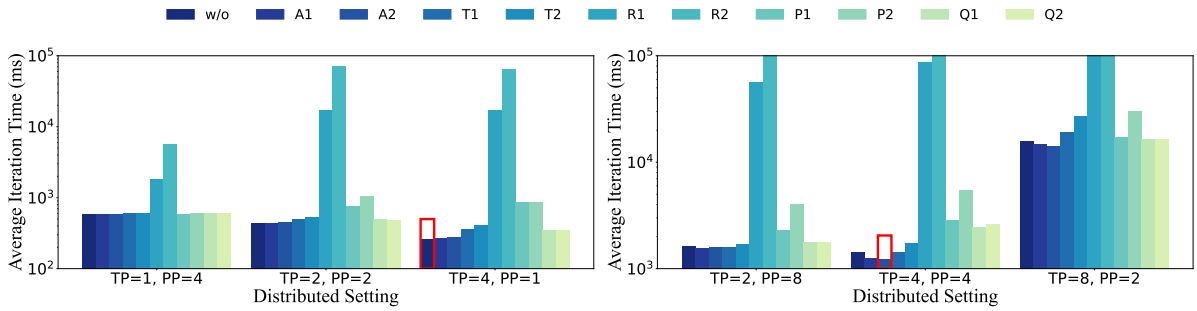
Table 1. Notation Table. TP/PP stands for the degree of tensor/pipeline model parallelism. 'comm' and 'comp' are short for 'communication' and 'compression'.

By default, we perform experiments on $\text{BERT}_{\text{LARGE}}$ model with 24 layers and compress the activation for the last 12 layers. For instance, when the pipeline model-parallel degree is 2 and the tensor model-parallel degree is 2, we compress the activation between two pipeline stages and the communication cost over tensor parallelism in the last 12 layers (we evaluate the impact of varying the number of compression layers in Appendix A.4). Similarly, we compress the activation for the last 6 layers when we perform experiments on $\text{BERT}_{\text{BASE}}$ and ViT-Base. Furthermore, we compress the activations of the final 18 layers of XLM-RoBERTa-XL and the final 16 layers of OPT-3B. Due to the limited space, we only present results from $\text{BERT}_{\text{LARGE}}$, ViT-Base, and XLM-RoBERTa-XL in this section and include other results in the Appendix A.

4.2 Fine-tuning on Single Node

Takeaway 1 *Among all evaluated compression methods, none of the techniques can be used to improve system throughput (by more than 1%) by compressing activations when doing fine-tuning tasks.*

When running fine-tuning experiments on a p3.8xlarge instance on Amazon EC2, we observe that we cannot improve system throughput by using any compression algorithms from Figure 3(a). We also find that the best configuration for fine-tuning is TP=4, PP=1 without using any evaluated compression methods. This is primarily due to the high bandwidth of NVLink, which means that communication is not a significant bottleneck, and the overhead of compres-



(a) Fine-tuning (b) Pre-training

Figure 3. Average iteration time (ms) for fine-tuning (left) and pre-training (right) with various compression techniques and distributed setting. For each setting, we repeat experiments for 5 times. Red rectangular boxes highlight the best method. 'w/o' is short for 'with of compression'.

Compression Algorithm	MNLI-(m/mm)	QQP	SST-2	MRPC	CoLA	QNLI	RTE	STS-B	Avg.
w/o	88.07/88.70	92.02	95.07	88.46	62.22	93.39	82.67	89.16	86.64
A1	85.42/85.43	91.07	92.09	86.14	54.18	91.31	70.04	87.61	82.59
A2	85.53/85.65	91.24	93.23	85.86	55.93	91.01	65.34	87.76	82.40
T1	32.05/32.18	74.31	83.60	70.78	0.00	58.37	51.99	0.00	44.81
T2	44.12/45.67	39.68	90.83	78.09	0.00	84.42	49.82	62.70	55.04
P1	83.03/83.43	91.08	90.60	79.07	0.00	89.73	59.21	74.66	72.31
P2	83.46/83.77	91.10	91.86	81.62	51.32	89.71	50.54	85.44	78.76
Q1	87.25/87.81	91.71	93.46	87.01	55.99	61.38	67.51	88.02	80.02
Q2	87.85/88.47	91.93	93.23	87.42	57.67	93.01	78.34	87.43	85.04

Table 2. Fine-tuning results over GLUE dataset with tensor model-parallel size 2 and pipeline model-parallel size 2. F1 scores are reported for QQP and MRPC, Matthews correlation coefficients are reported for CoLA, and Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks.

Compression Algorithm	CIFAR-10	CIFAR-100
w/o	98.81	91.83
A1	97.19	82.76
A2	97.14	83.51
T1	61.77	6.27
T2	65.54	10.74
P1	28.50	47.95
P2	21.32	5.02
Q1	92.96	84.09
Q2	98.85	91.94

Table 3. Fine-tuning results over CIFAR 10 and CIFAR 100 on ViT-Base with tensor model-parallel size 2 and pipeline model-parallel size 2.

sion algorithms, which in turn leads to additional time spent in encoding/decoding.

Moreover, we see similar results as with other models and our takeaway still holds for the XLM-RoBERTa-XL model. Table 4 shows the results from evaluating XLM-RoBERTa-XL over Cloudlab (Duplyakin et al., 2019) d8545 instances where each instance is equipped with 4 A100 GPUs and NVLink. Note that the hidden dimension for XLM-RoBERTa-XL is 2560, we set the output dimensions of A1 and A2 to 128 and 256, respectively, to maintain the same compression ratio used in prior experiments. Other compression methods maintain the same compression ratio as the autoencoder. In addition, Q1 and Q2 still reduce the precision to 2 bits and 4 bits respectively.

Takeaway 2 *For fine-tuning tasks, model parallelism compression can considerably degrade the model's accuracy, especially in the face of complex tasks. Additionally, Transformer-based models employing model parallelism compression can potentially underperform compared to smaller models.*

Given the higher complexity of CIFAR-100 over CIFAR-10 and RTE's status as one of the most challenging tasks in the GLUE benchmarks, an examination of Table 2 and Table 3 reveals that AE struggles to uphold the fine-tuning accuracy across the RTE and CIFAR-100 tasks. Furthermore, in contrast to the accuracy detailed in (Dosovitskiy et al., 2020), ViT-Base featuring AE compression exhibits inferior performance to Resnet-50 on CIFAR-10 and CIFAR-100 during fine-tuning. Hence, we suggest training smaller models for complex tasks to speed up.

4.3 Pre-training on Multiple Nodes

Takeaway 3 *Among all evaluated methods, AE is the best compression methods over pre-training. AE not only pre-*

Distributed Setting	w/o	A1	A2	T1	T2	P1
TP=2, PP=2	549.81	**545.12**	553.34	593.20	613.24	1,077.11
Distributed Setting	w/o	P2	R1	R2	Q1	Q2
TP=2, PP=2	549.81	1,609.01	16,525.53	35,240.28	602.86	609.45

Table 4. The average iteration time (ms) for fine-tuning XLM-RoBERTa-XL with various compression techniques by setting TP=2, PP=2. The results are collected from the Cloudlab d8545 machine **with NVLink** by using batch size 16, and sequence length 512. The best setting is **bolded** in the table. And the settings which see benefits compared with the baseline, are underlined.

Compression Algorithm	MNLI-(m/mm)	QQP	SST-2	MRPC	CoLA	QNLI	RTE	STS-B	Avg.
w/o	84.87/84.79	91.25	92.43	86.84	56.36	92.26	70.40	86.83	82.89
A2	83.77/84.32	91.14	91.63	86.55	58.61	91.96	71.48	87.16	82.96
T2	61.06/60.93	80.74	80.16	63.83	10.01	59.55	47.29	0.37	51.55
Q2	84.47/85.32	91.36	93.23	85.10	58.84	91.69	71.84	86.39	83.14

Table 5. Fine-tuning results over GLUE dataset by using the checkpoint obtained by pre-training. F1 scores are reported for QQP and MRPC, Matthews correlation coefficient is reported for CoLA, and Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks.

Compression Algorithm	Forward	Backward	Optimizer	Waiting & Pipeline Comm.	Total Time	Tensor Enc.	Tensor Dec.	Tensor Comm.
w/o	467.73	419.26	7.42	527.99	1,422.40	\	\	91.08
A1	546.95	455.26	7.29	233.47	1,242.97	8.64	16.20	32.76
A2	459.26	467.51	9.64	286.78	1,223.20	12.96	20.52	43.56
T1	813.03	433.42	7.35	156.67	1,410.47	108.00	268.92	115.92
T2	1,068.38	444.26	6.75	202.48	1,721.87	153.36	427.68	151.56
R1	78,906.91	444.88	6.08	3,707.37	83,065.23	73,847.16	279.72	649.44
R2	\	\	\	\	>100,000	\	\	\
P1	2,032.54	605.72	4.01	202.60	2,844.88	169.96	7.53	32.18
P2	4,316.74	712.26	6.81	359.70	5,395.51	405.12	10.20	46.82
Q1	803.63	417.33	8.61	1,205.46	2,435.03	90.72	304.56	193.68
Q2	805.33	417.74	7.55	1,364.32	2,594.94	85.32	271.08	111.60

Table 6. We breakdown the average iteration time (ms) for pre-training with various compression techniques when using tensor model-parallel size 4, pipeline model-parallel size 4, micro batch size 128, global batch size 1024, and sequence length 128. The results are collected from 4 AWS p3.8xlarge machines **with NVLink**. The total time (ms) is divided into following parts: forward step, backward step, optimizer, and waiting & pipeline communication. The last three columns further breakdown the tensor encoder/decoder and communication times which are considered part of the forward step.

serves the model's accuracy but also achieves the highest pre-training throughput.

For pre-training tasks, from Figure 3(b), we observe that, by using A2 to compress the activation over the last 12 layers, we can improve throughput for pre-training by 16%. From Table 5, compared with the baseline (without compression), we can observe that using AE is able to keep the accuracy when compared to the uncompressed model. In addition, **we observe that we can use the AE at the pre-training phase and remove it during the fine-tuning phase.** In other words, we only need to load the parameter of the $\text{BERT}_{\text{Large}}$ model to do fine-tuning, and the parameters of the AE can be ignored. Furthermore, Table 5 shows that pre-trained models suffer significant accuracy loss when using Top-K for compression. Finally, we find that quantization can preserve the model's accuracy, but we cannot achieve end-to-end speedup since the overhead of the quantization

method is too large as shown in Table 6. In conclusion, it is not a good choice to compress the activation by using quantization or Top-K.

Pipeline Stages	Comm. (w/o)	Comm. (A2)
$0 \leftrightarrow 1$	77.82	76.13
$1 \leftrightarrow 2$	88.69	13.19
$2 \leftrightarrow 3$	97.67	14.09

Table 7. The average communication time (ms) per iteration between two pipeline stages. The first column indicates the pipeline stage. And the second column shows the communication time per iteration without compression. Moreover, the third column presents the communication time with A2. We only compress the activation in the last 12 layers and thus the time for the first pipeline stage is unchanged.

Takeaway 4 *Compressing activation for models can improve throughput for multi-node pre-training by 16%. The reason behind this is that we can reduce the communication cost between the two pipeline stages. In addition, the main factor affecting the performance of the compression algorithm is the overhead of the compression methods.*

From Table 6, we notice that using AE and Top-K can reduce the waiting time and pipeline communication time of pre-training. This is because the inter-node bandwidth (10Gbps) is smaller than the intra-node bandwidth (40GB/s with NVLink), so compression is effective at reducing the communication time between two pipeline stages. From Table 7, we can observe that, by using A2 to compress the activation over the last 12 layers, we can reduce the communication cost between the two pipeline stages effectively. Additionally, as seen in Table 6, the performance impact of MP compression is primarily due to the overhead of the compression techniques.

4.4 Analysis of Scaling Up

We next study how the benefits from compression will change with increase in model sizes / cluster size. Previous studies (Narayanan et al., 2021b) have focused solely on modeling the computational costs of Transformer-based models, overlooking the communication costs and compression overheads associated with model parallelism. To overcome these limitations, we develop the first cost model specifically tailored for hybrid-parallel distributed training that incorporates compression techniques. Our model accounts for both compression overhead and communication costs across nodes, making it more generally applicable for distributed training.

Given the promising results from AE in the previous section, we choose AE as the compression method with our cost model from § 3.3. We first validate the correctness of our cost model by comparing its prediction to ground truth. The ground truth is real experimental results collected from the AWS platform by using MCBench.

Modeling T_{comp}. We model T_{comp}, the computation time of each Transformer layer, as a linear function of FLOPs with the coefficient α that corresponds to the peak performance of the GPU. In particular, we estimate α using ground truth wall clock time of the largest hidden size we can fit, where the GPU is likely to be of the peak utilization (Williams et al., 2009). During experiments, we found that fitting α using time of smaller hidden sizes can result in a *30x* higher prediction time for larger hidden sizes because of low GPU utilization. Our prediction versus the ground truth time is plotted in Figure 4(a).

Modeling T_{comm}. we model T_{comm}, the communication time of each Transformer layer, as a piece-wise function of the message size (Agarwal et al., 2022). Formally,

$$T_{comm}(Bsh) = \begin{cases} C & \text{if } Bsh < d \\ \beta Bsh & \text{if } Bsh \geq d \end{cases}$$

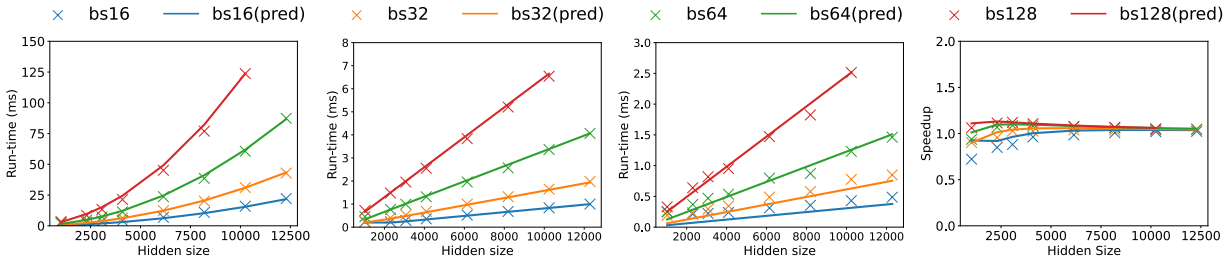
If the message size is smaller than a threshold d, then $T_{comm}(Bsh)$ is a constant C because the worker needs to launch at least one communication round (Li et al., 2020). Otherwise, the number of communication rounds is proportional to the message size. The fitting results are shown in Figure 4(b).

Modeling $T_{overhead}$. In AE, $T_{overhead}$ is the encoder and decoder computation time. It is a batched matrix multiplication with input dimension $B \times s \times h$ and $h \times h_c$ ($h_c < h$). $T_{overhead} = \gamma Bsh$ since $h \times h_c$ is negligible compared to $B \times s \times h$. The fitting results are shown in Figure 4(c).

Using the above cost model, we now compute the speedup as we vary the FM size by computing $\frac{T}{T_{AE}}$. Using a fixed encoder dimension h_c for AE (we set h_c to 100), the communication time in Eq. (2) can be modeled as $T_{comm}(Bsh_c)$. Compared to the setting without compression, the computation time remains unchanged. In addition, $T_{comm}(Bsh_c)$ is roughly equal to C because Bsh_c is usually smaller than the threshold d.

Since each Transformer layer has identical configurations in popular Transformer models (Devlin et al., 2018; Radford et al., 2018), the overall speedup ratio from using compression will remain the same as we vary the number of layers. Thus, we can estimate the speedup of different hidden sizes of any number of Transformer layers using $\frac{T}{T_{AE}}$. We provide the fitting result for this fraction in Figure 4(d).

To sum up, the fitting results shown in Figure 4 indicate that our cost model can predict the performance of MP compression correctly with various batch size and hidden size. Therefore, we use the cost model to estimate speedup for various FM sizes and cluster sizes next.



(a) T_{comp} (b) T_{comm} (c) $T_{overhead}$ (d) Speedup

Figure 4. Cost model validation with different batch size and hidden sizes. From left to right, we show computation time, communication time, overhead by using AE compression, and end-to-end speedup. We use a fixed tensor model-parallel degree 4. 'bs' is short for 'batch size', and 'pred' means the line is predicted by our developed cost model.

Understanding the Trend The asymptotic behavior of large hidden size h based on Eq. (1) and (2):

$$\frac{T}{T_{AE}} \approx \frac{\alpha \times \text{FLOPs} + \beta Bsh}{\alpha \times \text{FLOPs} + \gamma Bsh + C}$$

where $\text{FLOPs} = 96Bsh^2 + 16Bs^2h + 6BshV$.

For a fixed cluster, as hidden size increases, the benefits from model parallelism compression diminish and has less than 5% benefit when hidden size ≥ 12288. [1]

Scaling Up the Cluster Size. The overall speedup can be obtained by using $M_c = Bsh_c$ and $T_X = T_{AE}$ based on Eq. (3). Under the pre-training setup, the cost model predicts an acceleration of 15%, which is in agreement with our experimental results. From Table 8, the setting with a hidden size of 6144 is very similar to GPT-2 (Radford et al., 2019), and we see that we can achieve $\sim$1.09x speedup. When the hidden size is scaled up to 25,600, AE compression can achieve $\sim$1.26x speedup. This shows that if we increase the number of nodes when we increase in the number of layers, AE compression benefits can increase. Moreover, from Table 14 in the Appendix, model compression with AE attains $\sim 25\%$ per-iteration speedup at scale.

In summary, model parallelism compression has diminishing returns if we only scale up the model on a fixed cluster. To gain benefits, one needs to also **properly manage other parameters in the cost model, *e.g.*, scaling up the number of nodes and using pipeline parallelism**.

4.5 Limitations

Our experimental study, due to resource constraints, does not encompass some of the recent advances in FMs pre-training, such as LLaMA (Touvron et al., 2023). Nevertheless, our cost model can offer accurate guidance on performance at these scales. Additionally, our study does not account for the *error feedback* (EF) schemes, which are frequently incorporated in data-parallel compression (Seide et al., 2014; Stich et al., 2018). While EF might help improve accuracy, we note that using EF will increase the overhead of compression methods.

[1] In our hardware setup, $\alpha \approx 6.33 \times 10^{-13}$, $\beta \approx 3.37 \times 10^{-8}$, $\gamma \approx 1.5 \times 10^{-10}$, $C \approx 0.2$, and we pick $d = 204,800$.

hidden size	# layers	# nodes	batch size	speedup
6144	40	1	1024	1.09×
8192	48	2	1536	1.08×
10240	60	4	1792	1.09×
12288	80	8	2304	1.10×
16384	96	16	2048	1.15×
20480	105	35	2520	1.22×
25600	128	64	3072	1.26×

Table 8. Weak-scaling speedup for the Transformer models. The degree of tensor model parallelism is 4, and the micro-batch size is $\min\{128, \text{batch size}/\#\text{ nodes}\}$. We follow the other hyper-parameters as in Table 1 of (Narayanan et al., 2021b).

5 Related Work

In this section, we first outline the evolution of large foundation models, followed by an examination of strategies for training them at scale. Lastly, we discuss previous research that accelerates distributed ML model training by employing compression techniques.

Foundation models, first introduced by Vaswani et al. (2017) in the context of machine translation, have demonstrated effectiveness in various language understanding tasks, such as text generation, text classification, and question answering (Devlin et al., 2018; Radford et al., 2018; Wang et al., 2018a; Rajpurkar et al., 2016). More recent research has expanded foundation models to encompass images (Dosovitskiy et al., 2020; Touvron et al., 2021), audio (Gong et al., 2021), and video (Sharir et al., 2021). Due to their extensive applications, multiple optimizations have been proposed to expedite foundation model training, including optimized

I/O management (Dao et al., 2022) and simplifying the attention module (Wang et al., 2020). In our research, we focus on accelerating the training of foundation models in a *distributed* setting by reducing communication cost.

Various parallelism strategies have been proposed for training foundation models in a distributed manner. Megatron (Shoeybi et al., 2019) presents tensor model parallelism, which enables parallel computation in attention layers and subsequent matrix multiplications. DeepSpeed (Rasley et al., 2020) adopts a specialized pipeline parallelism form (Huang et al., 2019; Narayanan et al., 2019) that treats a transformer layer as the smallest unit in pipeline stages. It further integrates tensor model parallelism from Megatron and data parallelism to train foundation models with trillions of parameters. Li et al. (Li et al., 2022b) investigate a more intricate model parallelism strategy space for foundation models and use a cost model to automatically find the optimal strategy.

However, distributed ML model training requires frequent and significant synchronization between workers. This synchronization leads to substantial communication costs. To mitigate the communication bottleneck, several methods have been proposed to compress message size. One approach focuses on data parallelism settings, where workers exchange model gradients (Wang et al., 2021; Agarwal et al., 2022) during backward propagation. Techniques for reducing gradient communication include low-rank updates (Wang et al., 2018b), sparsification (Lin et al., 2017), and quantization (Seide et al., 2014; Bernstein et al., 2018; Dettmers, 2015). Another recent direction acknowledges that large neural network activations during forward propagation can benefit from compression (Wang et al., 2022). This approach employs quantization to compress the activation volume between pipeline parallelism workers, specifically targeting the geo-distributed setting with low network bandwidth. Moreover, this approach also needs additional storage of activations which makes it challenging to use with large datasets that are typically used for pre-training. In this paper, we conduct a comprehensive evaluation of various popular compression techniques, assessing their impact on tensor and pipeline parallelism within a typical cloud computing environment.

6 Conclusion

In this work, we studied the impact of compressing activations for models trained using model parallelism. We first developed a general performance model for model parallelism compression. Next, we implemented and integrated several popular compression algorithms into an existing distributed training framework (Megatron-LM) and evaluated their performance in terms of throughput and accuracy under various settings. Our results show that learning-based compression algorithms are the most effective approach for compressing activations in model parallelism. Based on the experimental results, we evaluate the correctness of our performance model and analyze the speedup when scaling up the model. Our experiments provide valuable insights for the development of compression algorithms in the future.

Acknowledgments

Shivaram Venkataraman is supported by the Office of the Vice Chancellor for Research and Graduate Education at UW-Madison with funding from the Wisconsin Alumni Research Foundation and by NSF award OAC2311766. Eric Xing is supported by NGA HM04762010002, NSF IIS1955532, NSF CNS2008248, NIGMS R01GM140467, NSF IIS2123952, NSF BCS2040381, NSF IIS2311990, a Semiconductor Research Corporation (SRC) AIHW award, and DARPA ECOLE HR00112390063.

References

Agarwal, S., Wang, H., Lee, K., Venkataraman, S., and Papailiopoulos, D. Adaptive gradient communication via critical learning regime identification. *Proceedings of Machine Learning and Systems*, 3:55–80, 2021.

Agarwal, S., Wang, H., Venkataraman, S., and Papailiopoulos, D. On the utility of gradient compression in distributed training systems. *Proceedings of Machine Learning and Systems*, 4:652–672, 2022.

Aji, A. F. and Heafield, K. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. Qsgd: Communication-efficient sgd via gradient quantization and encoding. *Advances in neural information processing systems*, 30, 2017.

Alistarh, D., Hoefler, T., Johansson, M., Konstantinov, N., Khirirat, S., and Renggli, C. The convergence of sparsified gradient methods. *Advances in Neural Information Processing Systems*, 31, 2018.

Bernstein, J., Wang, Y.-X., Azizzadenesheli, K., and Anandkumar, A. signsgd: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*, pp. 560–569. PMLR, 2018.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *arXiv preprint arXiv:2205.14135*, 2022.

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

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.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pp. 1–14, 2019.

Gong, Y., Chung, Y.-A., and Glass, J. Ast: Audio spectrogram transformer. *arXiv preprint arXiv:2104.01778*, 2021.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Hinton, G. E. and Zemel, R. Autoencoders, minimum description length and helmholtz free energy. *Advances in neural information processing systems*, 6, 1993.

Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J. K., Gibbons, P. B., Gibson, G. A., Ganger, G., and Xing, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pp. 1223–1231, 2013.

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, 2019.

Izsak, P., Berchansky, M., and Levy, O. How to train bert with an academic budget. *arXiv preprint arXiv:2104.07705*, 2021.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

Kim, J. K., Ho, Q., Lee, S., Zheng, X., Dai, W., Gibson, G. A., and Xing, E. P. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, pp. 1–16, 2016.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

Li, D., Shao, R., Wang, H., Guo, H., Xing, E. P., and Zhang, H. Mpcformer: fast, performant and private transformer inference with mpc. *arXiv preprint arXiv:2211.01452*, 2022a.

Li, D., Wang, H., Xing, E., and Zhang, H. Amp: Automatically finding model parallel strategies with heterogeneity awareness. *arXiv preprint arXiv:2210.07297*, 2022b.

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

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.

Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., Yasunaga, M., Zhang, Y., Narayanan, D., Wu, Y., Kumar, A., et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.

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.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Lu, W., Yan, G., Li, J., Gong, S., Han, Y., and Li, X. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 553–564. IEEE, 2017.

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.

Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pp. 7937–7947. PMLR, 2021a.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021b.

OpenAI, R. Gpt-4 technical report. *arXiv*, pp. 2303–08774, 2023.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. Improving language understanding by generative pre-training. 2018.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pp. 8748–8763. PMLR, 2021.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

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.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

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*. Citeseer, 2014.

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

Sharir, G., Noy, A., and Zelnik-Manor, L. An image is worth 16x16 words, what is a video worth? *arXiv preprint arXiv:2103.13915*, 2021.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

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.

Song, J., Yim, J., Jung, J., Jang, H., Kim, H.-J., Kim, Y., and Lee, J. Optimus-cc: Efficient large nlp model training with 3d parallelism aware communication compression. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 560–573, 2023.

Stich, S. U., Cordonnier, J.-B., and Jaggi, M. Sparsified sgd with memory. *Advances in Neural Information Processing Systems*, 31, 2018.

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., and Jégou, H. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, pp. 10347–10357. PMLR, 2021.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Vogels, T., Karimireddy, S. P., and Jaggi, M. Powersgd: Practical low-rank gradient compression for distributed optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018a.

Wang, H., Sievert, S., Liu, S., Charles, Z., Papailiopoulos, D., and Wright, S. Atomo: Communication-efficient

learning via atomic sparsification. *Advances in Neural Information Processing Systems*, 31, 2018b.

Wang, H., Agarwal, S., and Papailiopoulos, D. Pufferfish: communication-efficient models at no extra cost. *Proceedings of Machine Learning and Systems*, 3:365–386, 2021.

Wang, J., Yuan, B., Rimanic, L., He, Y., Dao, T., Chen, B., Re, C., and Zhang, C. Fine-tuning language models over slow networks using activation compression with guarantees. *arXiv preprint arXiv:2206.01299*, 2022.

Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Wang, Z., Wu, X. C., Xu, Z., and Ng, T. E. Cupcake: Acompression optimizer for scalable communication-efficient distributed training.

Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.

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, 2020.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Gonzalez, J. E., et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pp. 19–27, 2015.

Zhuang, Y., Zhao, H., Zheng, L., Li, Z., Xing, E. P., Ho, Q., Gonzalez, J. E., Stoica, I., and Zhang, H. On optimizing the communication of model parallelism. *arXiv preprint arXiv:2211.05322*, 2022.