SWARM PARALLELISM: TRAINING LARGE MODELS CAN BE SURPRISINGLY COMMUNICATION-EFFICIENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Many deep learning applications benefit from using large models with billions of parameters. These models can only be trained with specialized distributed training algorithms that require low-latency and high-bandwidth interconnect. As a result, large models are typically trained in dedicated GPU clusters that can be extremely costly to deploy and operate. In contrast, there are more affordable distributed training setups, such as using cheap "preemptible" instances or pooling together existing resources from multiple regions. However, both these setups come with unique challenges that make it impractical to train large models using conventional model parallelism. In this work, we carefully analyze these challenges and find configurations where *training larger models becomes less communication-intensive*. Based on these observations, we propose SWARM Parallelism[1] — a model-parallel training algorithm designed for swarms of poorly connected, heterogeneous unreliable devices. SWARM creates temporary randomized pipelines between available nodes that are rebalanced in case of failure. To further reduce the network usage of our approach, we develop several compression-aware architecture modifications and evaluate their tradeoffs. Finally, we combine our insights to train a large Transformer language model with 1.1B shared parameters ($\approx$13B before sharing) on a swarm of preemptible T4 GPUs with less than 400Mb/s network throughput.

1 INTRODUCTION

For the past several years, the deep learning community has been growing ever more reliant on large pretrained models. Perhaps the easiest example of this trend is natural language processing, where the parameter count of models grew from hundreds of millions (Vaswani et al., 2017; Radford et al., 2018; Devlin et al., 2019) to billions (Narayanan et al., 2021; Rosset; Raffel et al., 2020; Wang & Komatsuzaki, 2021; Sun et al., 2021) to hundreds of billions (Brown et al., 2020; Lepikhin et al., 2020; Fedus et al., 2021) with consistent gains in quality (Kaplan et al., 2020). Likewise, many models in computer vision are reaching the billion-parameter scale (Henighan et al., 2020; Ramesh et al., 2021; Zhai et al., 2021; Riquelme et al., 2021; Dai et al., 2021; Dhariwal & Nichol, 2021).

At this scale, the model no longer fits into a single accelerator and requires specialized training algorithms that partition model parameters across devices (Krizhevsky et al., 2012; Dean et al., 2012). While these model-parallel algorithms use different partitioning strategies, they all share the need to perform intensive device-to-device communication (Narayanan et al., 2019; 2021). Furthermore, if a single device fails, it will cause the entire training to break down. As a result, model-parallel algorithms are typically deployed in dedicated HPC[2] clusters or supercomputers (Shoeybi et al., 2019; Rajbhandari et al., 2020; Narayanan et al., 2021).

This kind of infrastructure is notoriously expensive to build and operate, available only to few well-funded universities and large corporations (Larrea et al., 2019; Strohmaier et al., 2021; Langston, 2020). Most researchers, especially in developing nations, simply cannot afford to run the necessary experiments to properly evaluate their ideas. This ultimately limits the scientific progress for many important research areas, such as solving NLP problems in "non-mainstream" languages.

[1]Stochastically Wired Adaptively Rebalanced Model Parallelism
[2]HPC stands for high performance computing, typically in supercomputers.

Several recent works propose more cost-efficient distributed training strategies that use fleets of temporary "preemptible" instances that can be dynamically allocated in regions with low demand for hardware and electricity, making them 2–10 times cheaper than their dedicated counterparts (Harlap et al., 2017). Another solution is to train in "collaborations" by pooling together preexisting resources or using the help of volunteers (Diskin et al., 2021; Atre et al., 2021; Ryabinin & Gusev, 2020). However, training in either of those setups requires specialized algorithms that can adapt to the changing number of workers, utilize heterogeneous devices and recover from hardware and network failures. While there are several practical algorithms for unreliable hardware (Kijsipongse et al., 2018; Lin et al., 2020; Ryabinin et al., 2021), they can only train relatively small models that *fit into the memory of the smallest device*. This limits the practical impact of cost-efficient strategies, as most computationally demanding workloads typically train models with billions of parameters.

In this work, we aim to find a practical way of training large models using unreliable heterogeneous devices and slow interconnect. Our contributions can be summarized as such:

- We carefully analyze the existing model-parallel training techniques and formulate the "Square-Cube Law" of distributed training: a counterintuitive observation that, for some methods, *training larger models can actually decrease the network overhead*.
- We develop **SWARM parallelism**: a decentralized model-parallel algorithm[3] that replaces traditional pipelines with randomized temporary routing between swarms of peers, allowing it to operate with heterogeneous devices and recover from node failures. To the best of our knowledge, this is the first general-purpose algorithm capable of billion-scale training on heterogeneous unreliable devices with slow interconnect.
- We study compression-aware architecture variants that utilize bottlenecks and activation sparsity to significantly reduce the network bandwidth requirements. We evaluate the performance and quality tradeoffs of two different methods and find that our best compression technique, that uses the maxout activation function for compression, reduces bandwidth requirements by 4x while degrading performance by only 4–8%.
- Combining SWARM parallelism and compression-aware techniques, we demonstrate that it is possible to train a billion-scale Transformer language model with high training throughput on preemptible low-power T4 GPUs and < 400Mb/s network.

2 Background & Related Work

2.1 Model-parallel training

Deep learning community came up with multiple algorithms for training large models. Most of them work by somehow dividing the model between multiple workers, which is known as model parallelism. The exact way in which these algorithms divide the model significantly affects their training performance and the maximum model size they can support.

Traditional model parallelism. Historically, the first general strategy for training large models was to assign each device to compute a subset of each model layer (e.g. a subset of neurons), then communicate activations between each other (Krizhevsky et al., 2012); see surveys by Ben-Nun & Hoefler (2019) and Tang et al. (2020) for a more comprehensive overview. Since each device stores a fraction of layer parameters, this technique can train extremely wide layers that would not fit into a single GPU. However, applying traditional model parallelism to deep models comes with a significant performance penalty, as it requires all-to-all communication after each layer. As a result, while intra-layer parallelism schemes are still widely used (Shazeer et al., 2018; Rajbhandari et al., 2020), they are usually applied within one physical server in combination with other strategies (Krizhevsky, 2014; Chilimbi et al., 2014; Jia et al., 2019; Narayanan et al., 2021).

Pipeline parallelism circumvents the need for expensive all-to-all communication by assigning each device with one or several full layers (Huang et al., 2019). During the forward pass, each stage applies its subset of layers to the inputs supplied by the previous stage, then sends the outputs of the final layer to the next stage. For backpropagation, this process is reversed, with each pipeline stage passing the gradients to the same device that previously supplied it with input activations.

[3]The code for our experiments can be found at `github.com/iclr2022-submit/swarm`

To better utilize the available devices, the pipeline must process multiple micro-batches per step, allowing each stage to run in parallel on a different batch of inputs. In practice, the number of micro-batches is limited by the available device memory, which results in the reduced device utilization when processing first and last micro-batches, which is known as the "bubble" overhead (Huang et al., 2019). To combat this issue, subsequent studies propose using activation checkpointing, interleaved scheduling, and even asynchronous training (Narayanan et al., 2019; 2021; Huang et al., 2019; Shoeybi et al., 2019; Yang et al., 2019). Another limitation of pipeline parallelism is that it needs to evenly split the compute load across the pipeline stages (Narayanan et al., 2019), which makes it difficult to apply when model layers have uneven computational cost.

Finally, there are two types of algorithms that can be used to train large models: data parallelism with dynamic parameter loading (Rajbhandari et al., 2020; Pudipeddi et al., 2020; Ren et al., 2021) and layer-dependent algorithms such as distributed Mixture-of-Experts (Jacobs et al., 1991; Shazeer et al., 2017; Lepikhin et al., 2020; Fedus et al., 2021). We review these algorithms in Appendix A and compare our approach with pipeline parallelism and offloading in Appendices E and F.

2.2 Distributed Training Outside HPC

The distributed training techniques described in Section 2.1 are designed for clusters of identical devices with rapid and reliable communication, making them a natural fit for the High-Performance Computing (HPC) environment. As we discussed earlier, this infrastructure is not always available or affordable. A more cost-efficient alternative is to use "preemptible" instances (Li et al., 2019; Zhang et al., 2020; Harlap et al., 2017) or volunteer computing (Diskin et al., 2021; Kijsipongse et al., 2018; Atre et al., 2021; Ryabinin & Gusev, 2020). However, these environments are more difficult for distributed training, as each machine can leave training abruptly due to hardware failure or spikes in demand. Furthermore, since there is a limited number of available instances per region, training at scale often requires operating in multiple geographic regions or using multiple instance types.

To summarize, there are five main limitations for distributed training in non-HPC environments: (1) nodes joining and leaving over time, (2) node and network failures, (3) uneven performance from heterogeneous devices, and, finally, (4) low network bandwidth and (5) high latency between regions.

In order to train with a dynamic number of workers, deep learning practitioners came up with elastic training algorithms (TorchElastic; ElasticHorovod). If one peer leaves or fails midway through training, these algorithms dynamically re-balance the load between the remaining devices and continue the training (Harlap et al., 2017; Ryabinin et al., 2021). If new devices join midway through training, they download the latest training parameters from their peers and train alongside them.

Another line of research tackles distributed training on heterogeneous devices with uneven performance. One way to solve this problem is to use asynchronous training, where devices compute gradients at their own pace and aggregate them using either a parameter server (Recht et al., 2011; Kijsipongse et al., 2018) or a decentralized communication network (Lian et al., 2017). This idea allows asynchronous algorithms to fully utilize each individual device, but may reduce the convergence rate due to using "stale" gradients (Recht et al., 2011; Aji & Heafield, 2019). Finally, several studies (Li et al., 2020; Ryabinin et al., 2021; Ren et al., 2021; Diskin et al., 2021) propose hybrid techniques that remove some synchronization bottlenecks while maintaining per-iteration convergence.

However, all of these use cases rely on data-parallel training and thus share a fundamental limitation: each computing node must be able to run the entire model. In contrast, the most compute-intensive tasks require training models with billions of parameters, which is way outside both GPU and host memory of most low-end GPUs. Unlike data-parallel training, the model-parallel algorithms from Section 2.1 are not redundant, which makes them more vulnerable to hardware and network failures. In fact, even a single failed node can cause permanent loss of trained parameters and leave others unable to run forward and backward passes. To the best of our knowledge, there is only one study about training large models with unreliable devices (Ryabinin & Gusev, 2020), but it supports only Mixture-of-Experts and requires at least 1Gb/s interconnect even in toy experiments.

2.3 Communication efficiency & Compression strategies

In this section, we discuss techniques that address training with limited network bandwidth and/or high latency, such as gradient compression or overlapping computation with communication phases.

Efficient parameter gradient communication. Data-parallel training requires the synchronization of gradients after each backpropagation pass, which can be costly if the model has many parameters or network bandwidth is limited. Deep Gradient Compression (Lin et al., 2018) approaches this problem by sparsifying gradients before synchronizing and corrects the momentum to work with sparse updates. PowerSGD (Vogels et al., 2019) compresses gradients via factorization and uses error feedback to counterbalance approximation errors over time. Techniques like 1-bit SGD (Seide et al., 2014), 1-bit Adam (Tang et al., 2021), and 1-bit LAMB (Li et al., 2021) do not synchronize gradients directly, but instead synchronize error-corrected 1-bit local optimizer statistics. Dettmers (2015) uses nonlinear 8-bit quantization to compress gradients before communication. We use this technique in conjunction with compression-aware architectures. Besides direct compression of gradients, another effective technique is to use layer sharing (Lan et al., 2020) which reduces the amount of parameter gradients that need to be synchronized by the factor of how many times a layer is reused.

Overlapping communication and computation. Model, pipeline, and data parallelism all have specific synchronization points at each layer, stage, and parameter update, respectively, and require communication-intensive transfers of gradients or activities. Usually, model training cannot progress until these transfers are completed. One effective way to reduce this cost is to overlap communication with computation and thereby *hiding* the transfer hiding the synchronization latency. This can be achieved by combining multiple parallelization techniques (Krizhevsky, 2014; Rajbhandari et al., 2020), by synchronizing parameter gradients layer-by-layer in lockstep with backpropagation (Paszke et al., 2019), or by using pure pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019). However, pure pipeline parallelism requires many pipeline stages to hide the communication effectively. To overcome this problem, we develop compression-aware architectures that work well even with relatively few pipeline stages. Another approach is to improve the computation/communication ratio instead of overlapping communication. This ratio can be improved by using gradient accumulation (Ott et al., 2018) or large batch sizes (You et al., 2020).

Model Compression methods. A seminal study of compression is work by Han et al. (2016) which uses a pipeline of pruning, quantization, and Huffman coding to compress neural networks for inference. Courbariaux et al. (2014) explores trade-offs between the numeric precision during training and the final test error. Work on 1-bit quantization seeks to trade predictive performance for speedups (Rastegari et al., 2016; Courbariaux et al., 2015). While extreme compression methods have been used for smaller models and convolutional networks successfully (Qin et al., 2020), quantization of large models or Transformers has been more challenging (Ramesh et al., 2021).

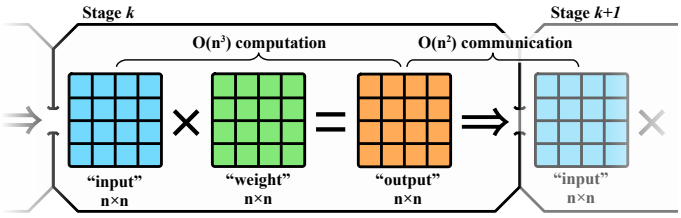
3 COMMUNICATION-EFFICIENT MODEL PARALLELISM

3.1 THE SQUARE-CUBE LAW OF DEEP LEARNING

To better understand the general scaling properties of model parallelism, we must abstract away from the application-specific parameters, such as model architecture, minibatch size, and system design. To that end, we first consider a simplified model of pipeline parallelism. Our "pipeline" consists of k stages, each represented by an $n \times n$ matrix. Intuitively, the first stage represents input data and all subsequent stages are applying "layers" to that data. This model does not account for specifics of any single application, we can use it do capture the intuition that will later apply to real-world cases.

During "training", stages iteratively perform matrix multiplication, then send the output to the subsequent pipeline stage over a throughput-limited network. These two operations have different scaling properties. The compute time for naive matrix multiplication scales as $O(n^3)$. While this can be reduced further in theory (Coppersmith & Winograd, 1990; Alman & Williams, 2021), it is only used for very large matrices (Zhang & Gao, 2015; Fatahalian et al., 2004; Huang et al., 2020), the standard for neural networks is to use $O(n^3)$ algorithms.

In turn, the communication phase requires at most $O(n^2)$ time to transfer a batch of activations (or gradients). Therefore, as we increase model size, the computation time grows faster than communication time, regardless of which matrix multiplication algorithm we use. We refer to this idea as the *square-cube law* after the eponymous principle in physics (Galileo, 1638; Allen, 2013).



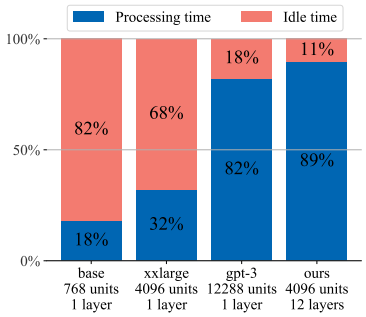


Figure 1: **(Left)** A simplified example of the square-cube law, **(Right)** Relative device utilization for Transformer layers using Tesla V100 and 500Mb/s network bandwidth (see Section 4.1).

This principle applies to many real-world neural networks architectures, albeit with some confounding variables. In convolutional neural networks (Fukushima, 1980), the computation time scales as $O(BHWC^2)$ and the communication is $O(BHWC)$, where B, H, W and C stand for batch size, height, width and the number of channels. Recurrent neural networks (Rumelhart et al., 1986; Hochreiter & Schmidhuber, 1995) need $O(BLH^2)$ compute in terms of batch size, sequence length and hidden size, respectively, and $O(BLH)$ or $O(BH)$ communication, depending on the architecture. With the same notation, transformers (Vaswani et al., 2017) require $O(BL^2H)$ compute for attention layers, $O(BLH^2)$ compute for feedforward layers, but only $O(BLH)$ communication in both cases.

Based on these observations, we conclude that pipeline parallelism naturally grows more communication-efficient with model size. More precisely, increasing the hidden dimension will reduce the communication load per device per unit of time, making it possible to train the model efficiently *with lower network bandwidth* and *higher latency*[4]. While the exact practical ramifications depend on the use case, Section 4.1 demonstrates some of the larger models trained with pipeline parallelism can already train at peak efficiency with only hundreds of Mb/s bandwidth.

Curiously, the square-cube principle also applies to intra-layer parallelism (Krizhevsky et al., 2012; Jacobs et al., 1991), but using this technique at 500Mb/s only becomes feasible for impractically large layer sizes in excess of 65536 units. Data-parallel training with sharding and/or offloading does not scale as well as its communication time scales the size of *model parameters* instead of activations. However, it may be possible to achieve similar scaling with another type of data-parallel training using low-rank gradient approximations (Vogels et al., 2019).

3.2 SWARM PARALLELISM

Traditional pipeline-parallelism can be communication-efficient, but this alone is not enough. Since each swarm participant has different compute and network capabilities, a pipeline formed out of such devices would be bottlenecked by the single weakest link, i.e. the participant with the smallest training throughput. As a result, the more powerful nodes along the pipeline would be underutilized due to either lack of inputs or slow subsequent stages. On top of that, if any single peer fails or leaves training prematurely, it will stall the entire training procedure.

In order to overcome these two challenges, we replace the rigid pipeline structure with randomized temporary "pipelines" formed on the fly during each iteration. Each participant can send their outputs to any peer that serves the next pipeline stage. Thus, if one peer has significantly more compute than others, it can process inputs from multiple predecessors and distribute its outputs across several weaker peers to maximize utilization. Furthermore, if a participant leaves during training, its predecessors can reroute their requests to its neighbors, allowing the training to proceed as long as there is at least one active participant at every pipeline stage (we elaborate on this Appendix C).

The resulting infrastructure consists of multiple consecutive swarms, as depicted in Figure 2. Peers within one swarm serve the same pipeline stage (i.e. the same subset of model layers). During forward pass, peers receive micro-batches of inputs from random predecessors and send activations a random peers in the next stage. For *backward* pass, peers receive gradients w.r.t. outputs and compute gradients w.r.t. layer inputs and accumulates gradients w.r.t. parameters. Each peer has two queues for incoming and outgoing requests to maintain high GPU utilization even with a noticeable communication latency via buffering. Similarly to other pipeline implementations (Huang et al., 2019; Narayanan et al., 2021), SWARM uses activation checkpointing (Griewank & Walther, 2000; Chen et al., 2016) to reduce the memory footprint.

[4]Latency slows the communication down by a constant factor that also grows less important with model size.

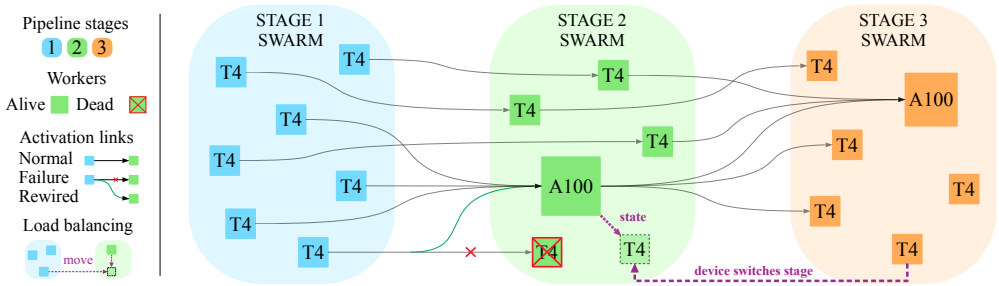


Figure 2: An overview of SWARM parallelism, illustrating both normal operation, device failures and adaptive rebalancing. One of the stage 2 workers leaves and a worker from stage 3 takes its place.

Once enough gradients are accumulated, peers form groups, run All-Reduce to aggregate gradients within their respective pipeline stages and run a global optimizer step. SWARM can also use Delayed Parameter Updates (DPU) to further improve device utilization by performing the optimizer step in parallel to running the next batch. While this is technically asynchronous, DPU is shown to achieve similar per-iteration convergence as fully synchronous training (Ren et al., 2021; Diskin et al., 2021).

Stochastic wiring. To better utilize the capacity of each individual device, we dynamically "wire" each input through each stage and pick devices in proportion to their training throughput. SWARM achieves this by creating several "trainer" processes that route through different pipeline stages using interleaved weighted round-robin algorithm (Katevenis et al., 1991; Tabatabaee et al., 2020) with randomized initial order; these processes do not use GPU and have no trainable parameters. Appendix D contains a detailed description of this component.

Adaptive swarm rebalancing. As we described earlier in Section 2.2, our workers can join and leave training at any time. If any single pipeline stage loses too many peers, the remaining ones will have to deal with an increased processing load, which will inevitably form a bottleneck. SWARM parallelism addresses this problem by allowing peers to dynamically switch between pipeline stages to rebalance the training throughput. Every T=300 seconds, peers measure the utilization rate of each pipeline stage as the queue size. Peers from the most under-utilized pipeline stage will then switch to the most over-utilized one (see Figure 2), download the latest training state from their new neighbors and continue training. Similarly, if a new peer joins midway through training, it automatically finds the most over-utilized pipeline stage by following the same protocol. As a side-effect, this technique allows SWARM to operate with uneven pipeline stages by allocating peers in proportion to the compute requirements of each pipeline stage.

3.3 Compression-aware architectures

Since pipeline parallelism has several distinct points of communication, the network overhead can be reduced considerably by reducing the size of features at these communication points. To exploit this, we develop compression-aware architectures that apply extreme compression at these points. We study two distinct ways of compression: compression through a linear bottleneck layer and compression through a bottleneck induced by the maxout activation function (Goodfellow et al., 2013). We combine our compression-aware architectures with 8-bit quantization and Huffman coding to reduce the communication overhead by another 62%.

Fully Connected Layers – The Baseline: Fully connected layers in transformers consist of a multi-layer perceptron with a single hidden layer and a non-linear activation function. Without biases and with a residual connection (He et al., 2015) from the inputs to the outputs this can be described as: $\text{MLP}(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_2) = \sigma(\mathbf{x}\mathbf{w}_1)\mathbf{w}_2 + \mathbf{x}$,

where $\mathbf{x} \in \mathbb{R}^{b \times s \times m}$, $\mathbf{w}_1 \in \mathbb{R}^{m \times h}$, $\mathbf{w}_2 \in \mathbb{R}^{h \times m}$, and $\sigma(\cdot)$, is a non-linear activation function, such as ReLU (Krizhevsky et al., 2012), and b, s, m, and h are the batch, sequence, model, and hidden dimension of the transformer neural network. To compress the output of the MLP layer, we want to apply a compression layer between two stages. For example, if we have 24 layers and 4 stages, we have 3 compression layers at layer 6, 12, and 18 that bridge these stages.

Bottleneck layers: We experiment with simple bottleneck layers that work by compressing the output features of the MLP by linear projection:

$$\text{Bottleneck}(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_c, \mathbf{w}_d) = \text{LayerNorm}(\text{LayerNorm}(\text{MLP}(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_2))\mathbf{w}_c)\mathbf{w}_\mathbf{d}, \quad (1)$$

where $\mathbf{w}_c \in \mathbb{R}^{m \times c}$, $\mathbf{w}_d \in \mathbb{R}^{c \times m}$ are compression and decompression parameters with compression dimension $c < m$. We find it critical to use a layer norm (Ba et al., 2016) to ensure training without divergence. The parameter tensor $\mathbf{w}_c$ resides in one stage and its outputs are transferred to the next stage which holds the parameter $\mathbf{w}_d$, which requires m/c times less communication compared to the original model. Note that adding a bottleneck only adds two linear layers for the forward pass and decreases the size of MLP activations; thus, its computational overhead is negligible.

Maxout Compression: Compared to bottleneck compression, maxout compression works by using the maxout activation function (Goodfellow et al., 2013) for compression rather than a linear projection. The maxout function of factor k takes inputs with a hidden dimension of d and reduces this dimension by the factor of k by computing the maximum value for each non-overlapping window of k features. We use maxout compression as follows:

$$\text{Maxout}(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_d) = \text{LayerNorm}(\text{maxout}_k(\text{LayerNorm}(\text{MLP}(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_2))))\mathbf{w_d}, \quad (2)$$

where the output is reduced by a factor of k through the maxout function in the previous stage, and these outputs are sent to the next stage which holds the decompression matrix $\mathbf{w}_d \in \mathbb{R}^{m/k \times m}$.

4 Experiments

4.1 Communication efficiency at scale

Before we can meaningfully evaluate SWARM parallelism, we must verify our theoretical observations on communication efficiency. Here we run several controlled experiments that measure the GPU utilization and network usage for different model sizes, using vanilla transformer architecture (Vaswani et al., 2017) with batch size 1 and sequence length 512. To decouple the performance impact from other factors, we run these experiments on homogeneous V100 GPU nodes that serve one pipeline stage over the network with varying latency and bandwidth. The full experimental configuration is deferred to Appendix B.

First, we measure how model size affects the computation to communication ratio at 500 Mb/s network bandwidth in both directions. We consider 4 different model configurations: the base configuration from the BERT paper (Devlin et al., 2019), "xxlarge" ("large" with d_{model}=4096), which is used in several recent works (Lan et al., 2020; Sun et al., 2021; He et al., 2020), the full GPT-3 model with d_{model}=12288 (Brown et al., 2020) and our modified Transformer architecture from Section 4.3 with d_{model}=4096, 3 layers per pipeline stage and 8-bit quantized activations. In the first three configurations, the model consists of 12 Transformer layers placed on 12 servers with a single GPU; in the last one, there are 4 servers, each hosting 3 layers. In addition, we evaluate how the same models perform under different network latency in Table 1.

As depicted in Figure 1 (right), larger models can indeed achieve a better GPU utilization rate, since their communication load grows slower than computation. More importantly, even at a modest 500Mb/s network, the resulting GPU idle time can be pushed into the 10–20% range: either naturally for GPT-3-sized models, or with smaller models through activation compression. Furthermore, these models maintain most of their training efficiency at 100ms latency (Figure 3), which is roughly equivalent to training on different continents (Verizon, 2021).

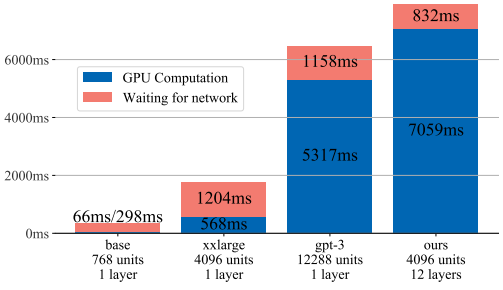


Figure 3: Pipeline computation and idle time per batch at 500Mb/s bandwidth.

Table 1: Relative device utilization at 500Mb/s using simulated network latency.

Latency (RTT)	Relative GPU utilization, (100% - idle time)			
	base	xxlarge	gpt-3	ours
none	18.0%	32.1%	82.1%	89.5%
10ms	11.8%	28.9%	79.3%	87.2%
50ms	4.88%	20.1%	70.3%	79.5%
100ms	2.78%	14.9%	60.2%	71.5%
200ms	1.53%	10.1%	48.5%	59.2%

Table 2: Training of language models on the OpenWebText Corpus (OWT). Our baseline model has 253M parameters and is trained for 8 GPU-days. We apply bottleneck and maxout compression to our baseline in 2 and 4 stages with a compression factor between 2-4x. We can see that maxout outperforms bottleneck compression if many stages or high compression factors are used. WT=WikiText, PTB=Penn Treebank, 1BW=Billion word corpus.

			Validation perplexity					
Model	Stages	Compression	OWT	LAMBADA	WT2	WT103	PTB	1BW
Baseline	–	–	19.7	86.4	56.2	35.4	133.0	80.9
Bottleneck	2	2x	**19.5**	87.7	**56.5**	**35.2**	**129.8**	79.2
Maxout	2	2x	19.6	**85.4**	56.6	**35.2**	**126.8**	**78.8**
Bottleneck	4	2x	21.7	100.0	66.4	40.0	149.6	89.5
Maxout	4	2x	**21.4**	**89.9**	**63.9**	**39.5**	**142.1**	**86.2**
Bottleneck	2	4x	21.6	99.8	64.8	39.6	145.6	88.3
Maxout	2	4x	**20.5**	**89.6**	**60.0**	**37.1**	**141.7**	**83.5**
Bottleneck	4	4x	28.9	141.6	100.2	58.1	235.5	118.3
Maxout	4	4x	**21.3**	**93.5**	**63.6**	**39.2**	**147.7**	**89.1**

4.2 Compression-Aware Architectures

Experimental Setup: As a baseline we train a transformer language model (Vaswani et al., 2017) on the OpenWebText corpus (Gokaslan & Cohen, 2019). We use the following transformer model: sequence size 512, 16 layers with model dimension 1024, and hidden dimension 4096 for a total of 253M parameters. We use BPE encoding (Radford et al., 2019) with vocabulary size of 50264 symbols. We do not use dropout or other regularization since our models underfit. We run our models in fairseq (Ott et al., 2019).

We test bottleneck and maxout compression for a compression factor of 50% and 75% compression compared to the original size over two and four stages. We look at how these compression aware architectures degrade performance compared to the compression that they achieve.

Results: The results of our compression aware architectures are shown in Table 2. We can see that while the bottleneck architecture is competitive with maxout for a compression factor of 2x with two stages, maxout has better perplexities if more stages or a higher compression ratio is used. The out-of-distribution perplexities vary consistently with the in-distribution perplexity, which suggest compression-aware architectures do not degrade out-of-distribution performance more than in-distribution performance. As such, maxout compression is an effective technique to reduce the bandwidth requirements of pipeline parallel training further.

Together with the square-cube law for distributed training, and SWARM parallelism, compression-aware architectures allow for better scaling of large neural networks trained over preemptible low network bandwidth peers. Thus, compression-aware architectures improve accessibility and affordability to train large neural networks outside of HPC environments.

4.3 Real-world convergence and scaling

Finally, to verify the efficiency of SWARM parallelism in a practical scenario, we conduct a series of large-scale distributed experiments using preemptible cloud T4 and A100 GPUs over the non-specialized network. In all experiments below, we train a Transformer language model similar to prior work (Brown et al., 2020; Wang & Komatsuzaki, 2021; Black et al., 2021) with 1.08 billion parameters in total: because of layer sharing, it is equivalent to a 12 times larger network in terms of compute requirements. The detailed setup of all experiments is given in Appendix B.

Specifically, our model consists of 4 stages, each containing a single Transformer decoder block with $d_{model} = 4096$ and 12 layers per pipeline stage, which decreases the size of data that needs to be sent during the aggregation step. We also use 8-bit compression (Dettmers, 2015) for activations and gradients to reduce the communication intensity of training. One additional advantage of using the activation compression (compared to other evaluated approaches) is that it does not require to modify the architecture and evaluate its performance; therefore, we hope to demonstrate that can be a simple yet practical solution for bandwidth-constrained environments.

First, to verify that model parallelism with asynchronous updates does not cause any significant convergence issues, we train the model on the Pile (Gao et al., 2020) dataset with 400 T4 cloud instances, each hosting a single accelerator. As a baseline, we use regular data-parallel training with offloading on 128 A100 GPUs. We report the per-iteration convergence of two setups to amortize the computational performance difference between the setups. Figure 4 shows the results of this experiment: it can be seen that the training dynamics of two approaches are indeed similar, which demonstrates the viability of SWARM parallelism for heterogeneous and poorly-connected devices. We also use the T4 node preemption data of this run to demonstrate the necessity of adaptive rebalancing in a pipeline of unreliable devices; refer to Appendix G for the description.

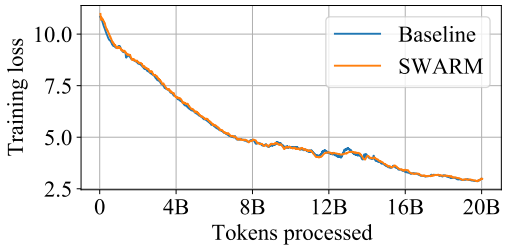


Figure 4: Training convergence comparison.

Table 3: Pipeline throughput comparison.

Hardware setup	Throughput, samples/s		Optimal bandwidth, Mb/s	
	Actual	Best-case	Upload	Download
T4	17.6	19.2	317.8	397.9
A100	16.9	25.5	436.1	545.1
T4 & A100	27.3	—	—	—

In the next experiment, we aim to measure the pipeline throughput in different hardware conditions and to compare it with theoretical best-case performance. For that, we use several setups: first, we use the same 400 preemptible T4 nodes; also, we use 7 instances with 8 A100 GPU each; finally, we combine these fleets to create a heterogeneous setup. We measure the number of samples processed by the pipeline both in our infrastructure and the ideal use case which ignores all network-related operations. As demonstrated in the left two columns of Table 3, asynchronous training of compute-intensive models with compressed activations allows us to achieve high performance without a dedicated networking solution. Furthermore, the load balancing algorithm of SWARM allows us to dynamically and efficiently utilize different hardware without being bottlenecked by slower devices.

Finally, we use the same load testing scenario to estimate the bandwidth required to fully utilize each device type in the above infrastructure. For this, we measure the average incoming and outgoing bandwidth on the nodes that serve the intermediate stage of the pipeline. We summarize our findings in the right two columns of Table 3: it turns out that with layer sharing and 8-bit compression, medium-performance GPUs (such as T4) can be saturated even with moderate network speeds. Although training over the Internet with more efficient hardware might indeed underutilize the accelerator, this issue can be offset by advanced compression strategies such as compression-aware architectures or more layer sharing iterations.

5 Conclusion

In this work, we have have analyzed and evaluated the feasibility of high-throughput training of neural models on unreliable, preemptible peers with low network bandwidth. Through analysis guided by the Square-Cube Law of distributed training, we found that this is feasible by training very large models with pipeline parallelism. Specifically, we propose SWARM parallelism to overcome the challenges of pipeline parallelism for preemptible devices with heterogeneous network bandwidths and computational throughputs. We have shown that SWARM parallelism is highly effective at re-balancing peers and maximizing the aggregate throughput of the entire model-parallel pipeline. We showed that the combination of SWARM parallelism and compression-aware architectures allows to train very large models on cheap preemptible cloud instances or on volunteered resources. As such, our work makes the training of large models accessible and affordable to small research teams that do not have access to dedicated infrastructure such as supercomputers.

REFERENCES

Alham Fikri Aji and Kenneth Heafield. Making asynchronous stochastic gradient descent work for transformers, 2019.

David H. Allen. *How Mechanics Shaped the Modern World*. 2013. ISBN 9783319017013.

Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *SODA*, 2021.

Medha Atre, Birendra Jha, and Ashwini Rao. Distributed deep learning using volunteer computing-like paradigm, 2021.

Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016. URL http://arxiv.org/abs/1607.06450.

Mandeep Baines, Shruti Bhosale, Vittorio Caggiano, Naman Goyal, Siddharth Goyal, Myle Ott, Benjamin Lefaudeux, Vitaliy Liptchinsky, Mike Rabbat, Sam Sheiffer, Anjali Sridhar, and Min Xu. Fairscale: A general purpose modular pytorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale, 2021.

Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4), aug 2019. ISSN 0360-0300. doi: 10.1145/3320060. URL https://doi.org/10.1145/3320060.

Sid Black, Gao Leo, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. URL https://doi.org/10.5281/zenodo.5297715. If you use this software, please cite it using these metadata.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 571–582, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi.

Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In Sanjoy Dasgupta and David McAllester (eds.), *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pp. 1337–1345, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL https://proceedings.mlr.press/v28/coates13.html.

Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990. ISSN 0747-7171. doi: https://doi.org/10.1016/S0747-7171(08)80013-2. URL https://www.sciencedirect.com/science/article/pii/S0747717108800132. Computational algebraic complexity editorial.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 3123–3131, 2015. URL https://proceedings.neurips.cc/paper/2015/hash/3e15cc11f979ed25912dff5b0669f2cd-Abstract.html.

Zihang Dai, Hanxiao Liu, Quoc V. Le, and Mingxing Tan. Coatnet: Marrying convolution and attention for all data sizes. *ArXiv*, abs/2106.04803, 2021.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25:1223–1231, 2012.

Tim Dettmers. 8-bit approximations for parallelism in deep learning. *ICLR*, 2015.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.

Prafulla Dhariwal and Alex Nichol. Diffusion models beat gans on image synthesis. *CoRR*, abs/2105.05233, 2021. URL https://arxiv.org/abs/2105.05233.

Michael Diskin, Alexey Bukhtiyarov, Max Ryabinin, Lucile Saulnier, Quentin Lhoest, Anton Sinitsin, Dmitriy Popov, Dmitry Pyrkin, Maxim Kashirin, Alexander Borzunov, Albert Villanova del Moral, Denis Mazur, Ilia Kobelev, Yacine Jernite, Thomas Wolf, and Gennady Pekhimenko. Distributed deep learning in open collaborations. *CoRR*, abs/2106.10207, 2021. URL https://arxiv.org/abs/2106.10207.

ElasticHorovod. Elastic Horovod. https://horovod.readthedocs.io/en/stable/elastic_include.html. Accessed: 2021-10-04.

Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. pp. 133–137, 01 2004. doi: 10.1145/1058129.1058148.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

Galilei Galileo. *Discorsi e dimostrazioni matematiche intorno a due nuove scienze*. 1638.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2020.

Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. *urlhttp://Skylion007. github. io/OpenWebTextCorpus*, 2019.

Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, and Yoshua Bengio. Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pp. 1319–1327. JMLR.org, 2013. URL http://proceedings.mlr.press/v28/goodfellow13.html.

Andreas Griewank and Andrea Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1510.00149.

Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pp. 589–604, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349383. doi: 10.1145/3064176.3064182. URL https://doi.org/10.1145/3064176.3064182.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2015.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention, 2020.

Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel Ziegler, John Schulman, Dario Amodei, and Sam McCandlish. Scaling laws for autoregressive generative modeling, 10 2020.

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. Technical Report FKI-207-95, Fakultät für Informatik, Technische Universität München, 1995. Revised 1996 (see www.idsia.ch/˜juergen, www7.informatik.tu-muenchen.de/˜hochreit).

Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn. Strassen's algorithm reloaded on gpus. *ACM Trans. Math. Softw.*, 46(1), March 2020. ISSN 0098-3500. doi: 10.1145/3372419. URL https://doi.org/10.1145/3372419.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pp. 103–112, 2019.

Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, March 1991. ISSN 0899-7667. doi: 10.1162/neco.1991.3.1.79. URL https://doi.org/10.1162/neco.1991.3.1.79.

Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia (eds.), *Proceedings of Machine Learning and Systems*, volume 1, pp. 1–13, 2019. URL https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE Journal on Selected Areas in Communications*, 9(8): 1265–1279, 1991. doi: 10.1109/49.105173.

Ekasit Kijsipongse, Apivadee Piyatumrong, and Suriya U-ruekolan. A hybrid gpu cluster and volunteer computing platform for scalable deep learning. *The Journal of Supercomputing*, 04 2018. doi: 10.1007/s11227-018-2375-9.

Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL http://arxiv.org/abs/1404.5997.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

Guillaume Lample, Alexandre Sablayrolles, Marc´ Aurelio Ranzato, Ludovic Denoyer, and Herve Jegou. Large memory layers with product keys. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dÁlché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8546–8557. Curran Associates, Inc., 2019. URL http://papers.nips.cc/paper/9061-large-memory-layers-with-product-keys.pdf.

Zhen-Zhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020.

Jennifer Langston. Microsoft announces new supercomputer, lays out vision for future ai work. https://blogs.microsoft.com/ai/openai-azure-supercomputer/, 2020. Accessed: 2021-10-1.

Verónica Larrea, Wayne Joubert, Michael Brim, Reuben Budiardja, Don Maxwell, Matt Ezell, Christopher Zimmer, Swen Boehm, Wael Elwasif, Sarp Oral, Chris Fuson, Daniel Pelfrey, Oscar Hernandez, Dustin Leverman, Jesse Hanley, Mark Berrill, and Arnold Tharrington. *Scaling the Summit: Deploying the World's Fastest Supercomputer*, pp. 330–351. 12 2019. ISBN 978-3-030-34355-2. doi: 10.1007/978-3-030-34356-9_26.

Dmitry Lepikhin, H. Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Y. Huang, M. Krikun, Noam Shazeer, and Z. Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *ArXiv*, abs/2006.16668, 2020.

Conglong Li, Ammar Ahmad Awan, Hanlin Tang, Samyam Rajbhandari, and Yuxiong He. 1-bit LAMB: communication efficient large-scale large-batch training with lamb's convergence speed. *CoRR*, abs/2104.06069, 2021. URL https://arxiv.org/abs/2104.06069.

Shigang Li, Tal Ben-Nun, Giorgi Nadiradze, Salvatore Digirolamo, Nikoli Dryden, Dan Alistarh, and Torsten Hoefler. Breaking (global) barriers in parallel stochastic optimization with wait-avoiding group averaging. *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2020. ISSN 2161-9883. doi: 10.1109/tpds.2020.3040606. URL http://dx.doi.org/10.1109/TPDS.2020.3040606.

Shijian Li, Robert J Walls, Lijie Xu, and Tian Guo. Speeding up deep learning with transient servers. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 125–135. IEEE, 2019.

Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/f75526659f31040afeb61cb7133e4e6d-Paper.pdf.

Jiahuang Lin, Xin Li, and Gennady Pekhimenko. Multi-node bert-pretraining: Cost-efficient approach, 2020.

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep Gradient Compression: Reducing the communication bandwidth for distributed training. In *The International Conference on Learning Representations*, 2018.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters. *arXiv preprint arXiv:2104.04473*, 2021.

Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pp. 1–9, Brussels, Belgium, October 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-6301. URL https://www.aclweb.org/anthology/W18-6301.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *NAACL-HLT*, 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.

Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. *arXiv preprint arXiv:2002.05645*, 2020.

Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. Binary neural networks: A survey. *CoRR*, abs/2004.03333, 2020. URL https://arxiv.org/abs/2004.03333.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018. URL https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training a trillion parameter models. In *SC*, 2020.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. *arXiv preprint arXiv:2104.07857*, 2021.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (eds.), *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pp. 525–542. Springer, 2016. doi: 10.1007/978-3-319-46493-0_32. URL https://doi.org/10.1007/978-3-319-46493-0_32.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pp. 693–701, 2011.

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training, 2021.

Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. *CoRR*, abs/2106.05974, 2021. URL https://arxiv.org/abs/2106.05974.

Corby Rosset. Turing-nlg: A 17-billion-parameter language model by microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

Max Ryabinin and Anton Gusev. Towards crowdsourced training of large neural networks using decentralized mixture-of-experts. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 3659–3672. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/25ddc0f8c9d3e22e03d3076f98d83cb2-Paper.pdf.

Max Ryabinin, Eduard Gorbunov, Vsevolod Plokhotnyuk, and Gennady Pekhimenko. Moshpit sgd: Communication-efficient decentralized training on heterogeneous unreliable devices, 2021.

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 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.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. *CoRR*, abs/1811.02084, 2018. URL http://arxiv.org/abs/1811.02084.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. Fugaku. https://www.top500.org/system/179807/, 2021. Estimated energy consumption 29,899.23 kW. Accessed: 2021-10-4.

Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2021.

Yu Sun, Shuohuan Wang, Shikun Feng, Siyu Ding, Chao Pang, Junyuan Shang, Jiaxiang Liu, Xuyi Chen, Yanbin Zhao, Yuxiang Lu, Weixin Liu, Zhihua Wu, Weibao Gong, Jianzhong Liang, Zhizhou Shang, Peng Sun, Wei Liu, Xuan Ouyang, Dianhai Yu, Hao Tian, Hua Wu, and Haifeng Wang. ERNIE 3.0: Large-scale knowledge enhanced pre-training for language understanding and generation. *CoRR*, abs/2107.02137, 2021. URL https://arxiv.org/abs/2107.02137.

Seyed Mohammadhossein Tabatabaee, Jean-Yves Le Boudec, and Marc Boyer. Interleaved weighted round-robin: A network calculus analysis. In *2020 32nd International Teletraffic Congress (ITC 32)*, pp. 64–72, 2020. doi: 10.1109/ITC3249928.2020.00016.

Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 1-bit adam: Communication efficient large-scale training with adam's convergence speed. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 10118–10129. PMLR, 2021. URL http://proceedings.mlr.press/v139/tang21a.html.

Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-efficient distributed deep learning: A comprehensive survey, 2020.

TorchElastic. PyTorch Elastic. https://pytorch.org/elastic. Accessed: 2021-10-04.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 30*, pp. 5998–6008. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

Verizon. Monthly ip latency data, 2021. Accessed: 2021-10-05.

Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 14236–14245, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/d9fbed9da256e344c1fa46bb46c34c5f-Abstract.html.

Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

Shang Wang, Yifan Bai, and Gennady Pekhimenko. Bppsa: Scaling back-propagation by parallel scan algorithm. In I. Dhillon, D. Papailiopoulos, and V. Sze (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 451–469, 2020. URL https://proceedings.mlsys.org/paper/2020/file/96da2f590cd7246bbde0051047b0d6f7-Paper.pdf.

Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *ArXiv*, abs/1910.05124, 2019.

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes, 2020.

Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. *CoRR*, abs/2106.04560, 2021. URL https://arxiv.org/abs/2106.04560.

Peng Zhang and Yuxiang Gao. Matrix multiplication on high-density multi-gpu architectures: Theoretical and experimental investigations. volume 9137, pp. 17–30, 06 2015. doi: 10.1007/978-3-319-20119-1_2.

Xiaoxi Zhang, Jianyu Wang, Gauri Joshi, and Carlee Joe-Wong. Machine learning on volatile instances. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pp. 139–148. IEEE, 2020.

SUPPLEMENTARY MATERIAL

A ADDITIONAL RELATED WORK

In this section, we review two additional types of distributed training algorithms that can support training large models.

Dynamic parameter loading. Several recent studies propose alternative execution algorithms that allow training large models with data parallelism. Since neural networks typically use a small fraction of weights at any given moment, the remaining "inactive" weights can be sharded (Rajbhandari et al., 2020) or offloaded to external memory (Pudipeddi et al., 2020; Ren et al., 2021; Rajbhandari et al., 2021). In sharded data-parallelism (Rajbhandari et al., 2020) inactive tensors are distributed across all devices such that each device stores $1/n$th of all parameters. For active layers, the shards are gathered such that each device holds the entire tensor just-in-time for computation. After the computation, the parameters' memory is freed such that only the sharded memory remains (1/Nth per device). This makes it very memory efficient to store model and optimizer states for inactive layers if many devices are available. Similarly to tensor parallelism, these algorithms can support arbitrary models without the need for layer partitioning, and can, in principle, run a large model on a single GPU, which is useful for fine-tuning and inference.

Architecture-specific methods. Finally, some distributed training algorithms take advantage of specific layers, such as locally-connected layers (Dean et al., 2012; Coates et al., 2013), Mixture-of-Experts (Jacobs et al., 1991; Shazeer et al., 2017; Lepikhin et al., 2020), Switch layers (Fedus et al., 2021) or Product Key Memory (Lample et al., 2019). These layers contain many near-independent parts that can be assigned to different devices. They can easily scale to an extremely large number of parameters with a relatively small increase in compute (Shazeer et al., 2017). However, they are also less parameter-efficient (Fedus et al., 2021) and may not apply to all architectures.

B DETAILED EXPERIMENTAL SETUPS

B.1 ADDITIONAL DETAILS FOR SECTION 4.1

We benchmark three popular versions of the Transformer layer:

- Transformer-base: $d_{model} = 768$, $d_{\text{FFN}} = 3072$, 12 heads;
- Transformer-xxlarge: $d_{model} = 4096$, $d_{\text{FFN}} = 16384$, 32 heads;
- "the GPT-3" (Brown et al., 2020): $d_{model} = 12288$, $d_{\text{FFN}} = 49152$, 96 heads.

For simplicity, we set up each experiment with 12 Transformer layers using 12 servers with a single V100-PCIE GPU each. The servers communicate at 500Mbps under 3–6ms latency. Finally, we evaluate GPT-3 with colocated layers — the same architecture as in GPT-3, but this time we place 3 layers per server (4 servers total).

Due to modest communication bandwidth, smaller models spend most of the time waiting for the network. However, that same bandwidth allows for $> 80\%$ GPU utilization when dealing with GPT-3-sized layers and over 90% with GPT-3 layers per device.

The training time reported above is the time required to run forward and backward pass for all layers with a batch of 4×128 tokens, not including the Adam updates. All results are averaged over 1000 consecutive batches, the standard deviations are below 0.1%. All four GPUs are in the same data center, but different servers. Each layer is a TransformerEncoderLayer (torch==1.7.0), wrapped with torch.utils.checkpoint. We use hivemind==0.8.15 (Ryabinin & Gusev, 2020) with a single synchronous trainer based on huggingface BERT training code. However, these results are not specific to hivemind and are likely reproducible in FairScale (Baines et al., 2021) or PyTorch RPC. The only important detail is that the training code should run as much communication as possible in the background, while the GPUs are busy processing batches.

It is important to reuse the same connection for multiple RPC calls, so the TCP buffer does not have to warm up again during each call. Also, our implementation performs quantization asynchronously with communication and other computations.

B.2 ADDITIONAL DETAILS FOR SECTION 4.3

For the language model training experiment, we use the standard Transformer architecture with two modifications. First, we use Rotary Positional Embeddings (Su et al., 2021) to increase the stability of our training runs. Second, we divide the model into four stages: two stages in the middle consist of regular stacked Transformer layers, the first stage also contains the embedding layer and the last stage includes the language modeling head. To reduce the communication intensity of All-Reduce during the data-parallel steps, we leverage weight sharing (similarly to ALBERT from Lan et al., 2020) in each stage.

C ANSWERS TO COMMON QUESTIONS

In this section, we provide answers to several assorted questions about our study and address some of the limitations of SWARM parallelism.

Why not just use data parallelism with offloading? Regular data parallelism requires all-reduce steps where peers exchange gradients. This can be prohibitively expensive for large models. For example, a 1B parameter model with 16-bit gradients requires 2 GB of data to be synchronized between all n devices. We need at least n messages to perform this synchronization. If we have, for example, 100 devices with bidirectional communication, each client would need to send 2 GB of data to finish the synchronization. Thus, with slow interconnects such synchronizations are not practical.

Why not just use fully sharded data parallelism with elasticity? Sharded data parallelism requires all-to-all communication of parameter buffers at each layer. Each of these all-to-all communications can be in parallel and has a size of parameters/n and in total n messages are required. Thus for 1B parameters in 16-bit precision, a total of 2GB of data need to be synchronized for both the

forward and backward pass. If using low-bandwidth devices with 100 Mbit/s speed, this would entail an overhead of 5.5 minutes per forward/backward pass which is difficult to hide under computation. This is exacerbated further because all-to-all communication latency is determined by the lowest peer. Thus sharded data parallelism can be particularly ineffective for environments where peers have heterogeneous network bandwidths.

Zero-offload allows one to train 13B parameters on a single V100, so why do I need SWARM? Training with zero-offload can slow down training due to the slow bandwidth between external memory and the accelerator. Training with SWARM can *accelerate* training while also allowing the training of large models. Also, see Appendix E for a detailed comparison.

Should I use SWARM in a supercomputer? By default, SWARM is worse than traditional parallelism due to extra complexity (see experiments in Section F. However, supercomputers often have heterogeneous devices, in which case SWARM is useful.

When should I avoid using SWARM? SWARM is very efficient at training large models with more than 1B parameters. For smaller models a sharded data-parallel approach can be effective. For HPC environments with homogeneous networking, standard sharded data parallel or pipeline parallel training will be more effective than SWARM because the environment is stable and predictable such that re-balancing is not required. For HPC environments which are so extensive that failure of a nodes is likely, the effectiveness of swarm depends on how many nodes are expected to fail. Elastic sharded data parallelism is better than SWARM if the number of expected failures is relatively low.

How much failure can SWARM handle? As long as there is at least one operational peer at every pipeline stage and at least one trainer SWARM will work without any issues. The same is true for network partitioning which requires at least one trainer and peer in each pipeline stage.

Can I use SWARM without layer sharing or quantization? Yes, SWARM can still be effective in this scenarios. Our bandwidth experiments in the main paper give some idea what the network overhead is. By using no quantization, that means using regular 16-bit activations, the network overhead increases roughly by a factor of two. Without layer sharing the overhead within each pipeline stage to synchronize the gradient is increased by the number of layers not being shared. As such, a rough estimate of the efficiency of SWARM in these scenarios can be estimated by taking our model-size-network-bandwidth-requirements data and multiplying it by the relevant factor.

Do the compression-aware architecture modifications apply only to transformers? Bottleneck and maxout compression are general compression techniques that can be applied to any layer in any architecture. However, their effectiveness may vary depending on where in the model they are applied and what kind of model these are applied to (CNN vs RNN vs Transformer, etc.)

Some configurations in Section 4.1 measure less than 20% GPU idle time, while many HPC systems only achieve $\approx 80\%$ GPU utilization. Does this mean that SWARM is 30% faster? No, because these are different measurement types. Narayanan et al. (2021) measures GPU utilization as a fraction of theoretical peak flop/s of their GPUs. In contrast, we only measure what fraction of time is GPU running the model, regardless of efficiency. Since no deep learning workload can achieve 100% peak flop/s, 20% GPU idle time for SWARM means that it can achieve $\approx 0.8\times$ the training throughput compared to training with a infinitely fast network. As a rule of thumb, one can say that SWARM will run at 20% slower than Narayanan et al. (2021) using the infrastructure that is several times cheaper.

D STOCHASTIC WIRING DETAILS

Our approach uses *stochastic wiring*, a specialized routing algorithm designed around heterogeneous unreliable devices and high network latency. The core idea of stochastic wiring is to route each training microbatch through randomly chosen pipeline stages such that the workload of each peer is proportional to its performance.

From a system design perspective, each device runs a separate *trainer* process that forms microbatches and routes them through pipeline stages (forward and backward pass). As we describe earlier in Section 3.2, trainers run Interleaved Weighted Round Robin (Katevenis et al., 1991; Tabatabaee et al., 2020) scheduling to dynamically assign microbatches to peers based on each peer's training throughput ("samples per second") in a balanced way.

To achieve this, trainers maintain a moving average estimate of each peer's training throughput based on how long does the peer take to process the batch. This moving average estimate is updated each time when the corresponding peer responds to a request. If a peer does not respond or otherwise fails to process the minibatch, trainer will temporarily ban this peer and reset its throughput estimate.

Curiously, different trainers can have different throughput estimates based on the network topology. For instance, if peers are located in two cloud regions, a given peer's trainer will have higher throughput estimate for peers in the same data center. In other words, trainers automatically adjust to the network topology by routing more traffic to peers that are "nearby".

Another important observation is that *stochastic wiring allows SWARM to mitigate network latency*. Unlike existing pipeline algorithms (Huang et al., 2019), SWARM workers do not get blocked if their neighbors take too long to process a minibatch. Instead, each SWARM device maintains a queue of microbatches assigned by trainers. In case of a latency spike, workers keep processing previously queued microbatches, maintaining high device utilization.

E ON THE RELATION BETWEEN SWARM AND ZERO-OFFLOAD

In this section, we argue that depending on the use of DPU, SWARM-parallel training is equivalent to either fully synchronous training or ZeRO-Offload (Ren et al., 2021). That is, SWARM produces exactly the same stepwise updates as conventional distributed training algorithms and will therefore achieve a solution in the same number of steps.

This observation is similar to how many advanced distributed training techniques (Huang et al., 2019; Rajbhandari et al., 2020) are computationally equivalent to regular synchronous training on a single device. For instance, despite using advanced distributed computation strategies, GPipe (Huang et al., 2019) computes exactly the same mathematical expression to obtain gradients and applies those gradients in the same order as any other *synchronous* training algorithm. On the other hand, PipeDream (Narayanan et al., 2019) changes the order in which the updates are applied, introducing the so-called stale gradients (Recht et al., 2011). This allows PipeDream to improve device utilization, but it has been shown to reduce the final model quality in some setups (Wang et al., 2020).

Despite using randomized routing and asynchronous communication between pipeline stages, SWARM still performs optimizer steps synchronously after peers collectively accumulate a global batch size (hyperparameter). While different peers may accumulate a different number of samples, they will all use the same gradient after averaging. Finally, any peer that fails or does not meet this condition for any reason is considered a straggler and must reload its state from neighbors before it can resume training.

This ensures that all surviving peers use non-stale aggregated gradient over the specified batch size when they perform the optimizer step. The only deviation from fully synchronous training is that SWARM uses the same approach for CPU offloading as ZeRO-Offload, and by extension, delayed parameter updates (DPU). While DPU were shown not to affect convergence (Ren et al., 2021), one can disable this functionality and make SWARM fully equivalent to normal training.

Naturally, these guarantees come at the cost of reduced compute utilization as a small portion of devices will need to wait after every step. However, as we show in Section 4.3, SWARM can still train with competitive training throughput due to the fact that large models are trained with increased batch sizes (Brown et al., 2020).

F DETAILED PERFORMANCE COMPARISON

Here, we investigate how SWARM parallelism compares to existing systems for training large models: **GPipe** (Huang et al., 2019) and **ZeRO-Offload** (Ren et al., 2021). The purpose of this section is to compare the training throughput in "ideal" conditions (with homogeneous reliable devices and balanced layers), as deviating from these conditions makes it *infeasible* to train with baseline systems.

We evaluate training performance for sequences of 4 Transformer layers of identical size distributed over 16 workers. The pipeline does not contain embeddings or language modeling heads, as it would result in imbalance between the stages. Similarly to Section 4.1, we use two layer configurations: "xxlarge" (d_{model}=4096, d_{FFN}=16384, 32 heads) and "GPT-3" (d_{model}=12288, d_{FFN}=49152, 96 heads). The microbatch size is 4 for "xxlarge" and 1 for "GPT-3", and the sequence length is 512.

To provide a more detailed view of the training performance, we measure two separate performance statistics: the training throughput and the All-Reduce time. The training throughput measures the rate at which the system can process training sequences, i.e. run forward and backward passes. In turn, the All-Reduce time is the time each system spends to aggregate those accumulated gradients across devices. The total time per step can be computed as `batch_size / throughput + all_reduce_time`. Intuitively, training with small batch sizes is more sensitive to the All-Reduce time since the algorithm needs to run All-Reduce more frequently and vice versa.

Hardware setup: Each worker uses a V100-PCIe GPU with 16 CPU threads (E5 v5-2660v4) and 128 GB RAM. The only exception is for ZeRO-Offload with "GPT-3" layers, where we had to double the RAM size because the system required 190GB at peak. Similarly to Section 4.1, each worker can communicate at a 500Mb/s bandwidth for both upload and download for a total of 1Gb/s. In terms of latency, we consider two setups: with **no latency**, where workers communicate normally within the same rack, and with **latency**, where we inject additional 100 ± 50ms latency in the kernel[5].

GPipe configuration: We use a popular PyTorch-based implementation of GPipe[6]. The model is partitioned into 4 stages repeated over 4 model-parallel groups. To fit into the GPU memory for the "GPT-3" configuration, we offload the optimizer into RAM using ZeRO-Offload. Before averaging, we use PyTorch's built-in All-Reduce to aggregate gradients.

ZeRO-Offload configuration: Each worker runs the entire model individually, then aggregates gradients with peers. For "xxlarge", we use the official implementation from Ren et al. (2021). However, for "GPT-3", we found that offloading the optimizer still does not allow us to fit 4 layers into the GPU memory even with batch size 1. For this reason, we also offload parameters into RAM using `"offload_param"` with device set to `"cpu"`[7].

Table 4: Training performance for "GPT-3".

System	Throughput, samples / s		All-Reduce time seconds / round	
	No latency	Latency	No latency	Latency
SWARM	0.619	**0.558**	441.7	**455.4**
GPipe	**0.633**	0.477	**403**	469.6
Offload	0.382	0.382	1527.9	1635.4

Table 5: Training performance for "xxlarge".

System	Throughput, samples / s		All-Reduce time seconds / round	
	No latency	Latency	No latency	Latency
SWARM	2.358	2.161	45.36	**51.269**
GPipe	2.541	0.957	**44.17**	64.828
Offload	**3.08**	**3.08**	168.71	252.26

We report our measurements in Tables 4 and 5. Table 4 demonstrates that for larger model size, SWARM and GPipe have approximately the same performance without latency. When training with latency, SWARM significantly outperforms GPipe, which is likely caused by the asynchronous queueing (see Appendix D). In turn, when training smaller models (Table 5), ZeRO-Offload outperforms both SWARM and GPipe. This result coincides with our earlier observations in Figure 1, where the same model spent the most of the time waiting for communication between pipeline stages.

We also observe that ZeRO-Offload takes longer to aggregate gradients, likely because each peer must aggregate the entire model, whereas in SWARM and GPipe peers aggregate a single pipeline stage. The variation between All-Reduce time in GPipe and SWARM is due to implementation differences.

[5]More specifically, `tc qdisc add dev <...> root netem delay 100ms 50ms`
[6]The source code is available at `https://github.com/kakaobrain/torchgpipe`
[7]Based on `https://www.deepspeed.ai/docs/config-json/#parameter-offloading`

G ADAPTIVE REBALANCING EVALUATION

In this experiment, we evaluate the efficiency of adaptive peer rebalancing between stages proposed in Section 3.2. We use actual statistics of the number of active T4 nodes from the 32-hour segment of the experiment described in Section 4.3 for a representative sample of training dynamics with unstable participation. We use these data to simulate training dynamics as follow: we use a sequence of events, each consisting of a timestamp and the change in the number of peers (which can be positive or negative). When a worker is removed from the pipeline, we randomly choose the stage it was removed from: that is, removing N peers corresponds to N samples from the uniform distribution over four pipeline stages. To compare our method with the baseline without rebalancing, we run 10 simulations over different random seeds and average the resulting trajectories.

The results of this evaluation are available in Figure 5; for reference, we also provide the performance of a theoretically optimal rebalancing strategy that maintains the highest possible throughput at every moment. It can be seen that even with rebalancing period $T = 300$, adding dynamic rebalancing helps significantly improve the overall throughput of the pipeline during the experiment. When the total number of peers is approximately stable, the rebalanced pipeline also reaches the optimal one in terms of throughput, which shows the efficiency of our strategy even when moving only one node at a time. In addition, we observed that for some brief periods, the performance of the unbalanced pipeline exceeded the throughput of the balanced one due to random choice of disconnecting peers (dropping more from "overrepresented" stages affects the imbalanced pipeline less). However, this held true only for $\approx 4.5\%$ of the entire experiment and was quickly mitigated by adaptive rebalancing.

As expected, decreasing T from 300 to 60 seconds improves both the overall throughput and the speed of convergence to optimal pipeline performance; however, the effect is not as drastic compared to the increase in DHT data transfer. This is also demonstrated by Table 6, which shows the relative throughput of the three configurations compared to the optimal one. Furthermore, the table displays that although initially there is little difference between rebalancing choices, it becomes more pronounced later on as the imbalanced version drifts further from the optimal state.

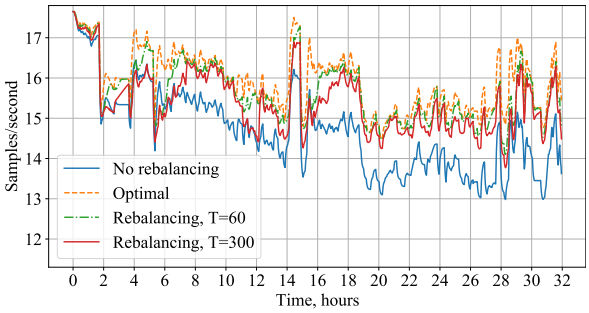


Figure 5: Throughput of rebalancing methods over time.

Table 6: Relative throughput comparison of pipeline rebalancing methods.

Rebalancing	% of optimal		
	Overall	First 1h	Last 1h
None	82.7	99.0	45.4
$T = 300$	95.8	99.4	88.9
$T = 60$	97.6	99.8	91.7