SWARM Parallelism: Training Large Models Can Be Surprisingly Communication-Efficient

Max Ryabinin [*] [1] [2] Tim Dettmers [*] [3] Michael Diskin [2] [1] Alexander Borzunov [1] [2]

Abstract

Many deep learning applications benefit from using large models with billions of parameters. Training these models is notoriously expensive due to the need for specialized HPC clusters. In this work, we consider alternative setups for training large models: using cheap "preemptible" instances or pooling existing resources from multiple regions. We analyze the performance of existing model-parallel algorithms in these conditions and find configurations where *training larger models becomes less communication-intensive*. Based on these findings, we propose SWARM parallelism[1], a model-parallel training algorithm designed for poorly connected, heterogeneous and unreliable devices. SWARM creates temporary randomized pipelines between nodes that are rebalanced in case of failure. We empirically validate our findings and compare SWARM parallelism with existing large-scale training approaches. Finally, we combine our insights with compression strategies to train a large Transformer language model with 1B shared parameters (≈13B before sharing) on preemptible T4 GPUs with less than 200Mb/s network.

1. Introduction

For the past several years, the deep learning community has been growing more reliant on large pretrained neural networks. The most evident example of this trend is natural language processing, where the parameter count of models has grown from hundreds of millions (Vaswani et al., 2017; Radford et al., 2018; Devlin et al., 2019) to billions (Narayanan et al., 2021; Raffel et al., 2020; Wang & Komatsuzaki, 2021; Sun et al., 2021) to hundreds of billions (Brown et al., 2020; Fedus et al., 2021; Chowdhery et al., 2022; Rae et al., 2021) with consistent gains in quality (Kaplan et al., 2020). Likewise, many models in computer vision are reaching the billion-parameter scale (Ramesh et al., 2021; Zhai et al., 2021; Dai et al., 2021; Dhariwal & Nichol, 2021).

At this scale, the models no longer fit into a single accelerator and require specialized training algorithms that partition the 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). Also, if a single device fails, it will cause the entire training process to break down. As a result, model-parallel algorithms are typically deployed in dedicated high-performance computing (HPC) 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, which makes it available only to a few well-resourced organizations (Larrea et al., 2019; Strohmaier et al., 2021; Langston, 2020). Most researchers cannot afford the experiments necessary for a proper evaluation of their ideas. This ultimately limits the scientific progress for many important research areas, such as solving NLP problems in "non-mainstream" languages.

Several recent works propose more cost-efficient distributed training strategies that leverage 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; Yuan et al., 2022).

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.,

[*] Equal contribution [1] HSE University [2] Yandex [3] University of Washington. Correspondence to: Max Ryabinin <mryabinin0@gmail.com>.

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

[1] SWARM parallelism is a backronym for Stochastically Wired Adaptively Rebalanced Model Parallelism.

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, because today's large-scale experiments often involve models with billions of parameters.

In this work, we aim to find a practical way of training large neural networks using **unreliable heterogeneous devices with slow interconnect**. We begin by studying the impact of model size on the balance between communication and computation costs of pipeline-parallel training. Specifically, increasing the size leads computation costs to grow faster than the network footprint, thus making **household-grade connection speeds** more practical than one might think. This idea inspires the creation of **SWARM parallelism**, a pipeline-parallel approach designed to handle peer failures by prioritizing stable peers with lower latency. In addition, this approach periodically rebalances the pipeline stages, which allows handling devices with different hardware and network speeds.

In summary, we make the following contributions:

- We 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[2]that leverages randomized fault-tolerant pipelines and dynamically rebalances nodes between pipeline stages. To the best of our knowledge, this is the first decentralized algorithm capable of billion-scale training on heterogeneous unreliable devices with slow interconnect.
- Combining insights from the square-cube law, SWARM parallelism, and 8-bit compression, we show that it is possible to train a billion-scale Transformer language model on preemptible servers with low-power GPUs and the network bandwidth of less than 200Mb/s while achieving high training throughput.

2. Background & Related Work

2.1. Model-Parallel Training

Over the past decade, the deep learning community has developed several algorithms for training large neural networks. Most of them work by dividing the model between multiple workers, which is known as model parallelism. The exact way in which these algorithms divide the model determines 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 layer (e.g., a subset of neurons), then communicate the results between each other (Krizhevsky et al., 2012; Ben-Nun & Hoefler, 2019; Tang et al., 2020). Since each device stores a fraction of layer parameters, this technique can train models with extremely wide layers that would not fit into a single GPU. However, applying traditional model parallelism to deep neural networks comes at a significant performance penalty, as it requires all-to-all communication after each layer. As a result, while intra-layer parallelism is still widely used (Shazeer et al., 2018; Rajbhandari et al., 2020), it is 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 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 last layer to the next stage. For the backward pass, this process is reversed, with each pipeline stage passing the gradients to the device that supplied it with input activations.

To better utilize the available devices, the pipeline must process multiple microbatches per step, allowing each stage to run in parallel on a different batch of inputs. In practice, the number of microbatches is limited by the device memory: this results in reduced device utilization when processing the first and the last microbatches, 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).

Aside from model parallelism, there two more strategies for training large models: data parallelism with dynamic parameter loading (Rajbhandari et al., 2020) and model-specific algorithms such as Mixture-of-Experts (Shazeer et al., 2017). We discuss these algorithms in Appendix B and compare the performance of offloading with SWARM in Section 4.2 and Appendix E.

2.2. Distributed Training Outside HPC

The 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 HPC setup. As we discussed earlier, such infrastructure is not always available, and 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 (Kijsipongse et al., 2018; Ryabinin & Gusev, 2020; Atre et al., 2021; Diskin et al., 2021). However, these environments are more difficult for

[2] The code for our experiments can be found at `github.com/yandex-research/swarm`.

distributed training: each machine can disconnect abruptly due to a failure or preemption. Besides, since there is a limited number of available instances per region, training at scale often requires operating across multiple locations or using different instance types.

To handle unstable peers and heterogeneous devices, the research community has proposed elastic and asynchronous training methods, correspondingly. Moreover, training large models over heterogeneous devices can be optimized with global scheduling (Yuan et al., 2022). We describe these methods in more detail in Appendix B; importantly, neither of them are unable to satisfy all the constraints of our setup.

By contrast, the largest models have billions of parameters, which exceeds the memory limits of most low-end computers. However, model-parallel algorithms are not redundant, which makes them more vulnerable to hardware and network failures. There exist two methods that allow training large models with unreliable devices (Ryabinin & Gusev, 2020; Thorpe et al., 2022): however, the first one supports only specific architectures and requires at least 1Gb/s bandwidth, whereas the second one has no publicly available implementations, relies on redundant computations for fault tolerance and considers only the homogeneous setup.

2.3. Communication Efficiency and Compression

In this section, we discuss techniques that address training with limited network bandwidth or high latency, such as gradient compression or overlapping computation with communication phases. These techniques are often necessary for distributed training without high-speed connectivity, because otherwise the performance of the system becomes severely bottlenecked by communication.

Efficient gradient communication. Data-parallel training requires synchronization of gradients after each backward pass, which can be costly if the model has many parameters or the network bandwidth is limited. There exist several methods that approach this problem: for example, Deep Gradient Compression (Lin et al., 2018) sparsifies the gradients and corrects the momentum after synchronization, while PowerSGD (Vogels et al., 2019) factorizes the gradients and uses error feedback to reduce the approximation error. Recently, Wang et al. (2022) proposed to compress the changes of model activations, achieving high-speed communication for finetuning models of up to 1.5B parameters. Alternatively, Dettmers (2016) uses 8-bit quantization to compress gradients before communication. We evaluate it along with compression-aware architectures, leaving the exploration of more advanced approaches to future work.

Besides gradient compression, another effective technique is to use layer sharing (Lan et al., 2020), which reduces the number of aggregated gradients by a factor of how many times each layer is reused.

Overlapping communication and computation. Model, pipeline, and data parallelism all have synchronization points and require transfer of gradients or activations. One way to reduce the transfer cost is to overlap communication with computation, *hiding* the synchronization latency. This overlap can be achieved by combining parallelism techniques (Krizhevsky, 2014; Rajbhandari et al., 2020), by synchronizing 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 stages to effectively hide the latency. To overcome this problem, we study inter-layer compression techniques that work well even with relatively few pipeline stages.

3. Communication-Efficient Model Parallelism

In this section, we outline our approach for training large models with heterogeneous unreliable poorly-connected devices. To that end, the section is organized as follows:

- Section 3.1 analyzes how existing model-parallel algorithms scale with model size and shows conditions where training increasingly larger models leads to less intense network usage;
- Section 3.2 describes SWARM parallelism — a decentralized algorithm for training large models under the conditions outlined in Section 2.2.

3.1. The Square-Cube Law of Distributed Training

To better understand the general scaling properties of model parallelism, we need to abstract away from the application-specific parameters, such as model architecture, batch 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 $n{\times}n$ matrices. Intuitively, the first matrix represents the input data and all subsequent matrices are linear "layers" applied to that data. This model abstracts away from application-specific details, allowing us to capture general relationships that hold for many models.

During "training", stages iteratively perform matrix multiplication and 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 naïve 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). Therefore, deep learning on GPUs typically relies on $O(n^3)$ algorithms.

In turn, the communication phase requires at most $O(n^2)$ time to transfer a batch of $n{\times}n$ activations or gradients. Therefore, as we increase the model size, the computation

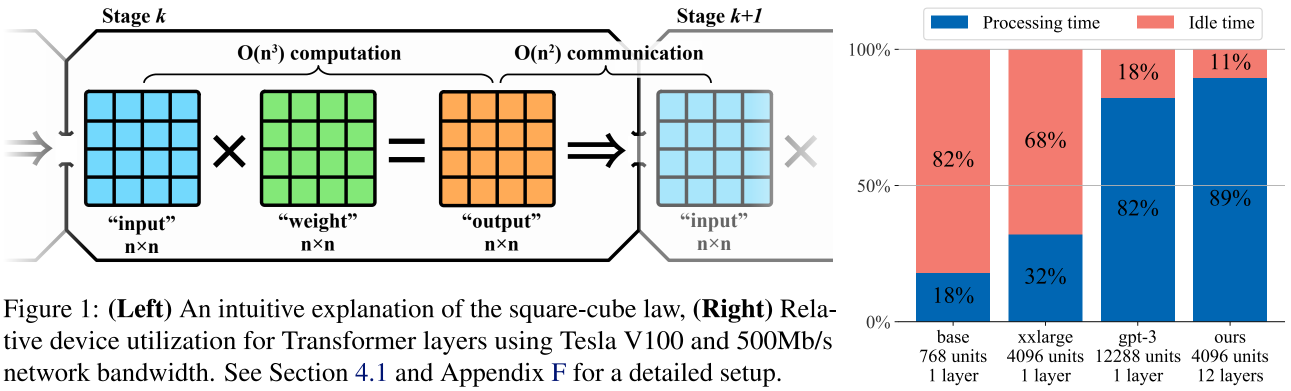


Figure 1: **(Left)** An intuitive explanation of the square-cube law, **(Right)** Relative device utilization for Transformer layers using Tesla V100 and 500Mb/s network bandwidth. See Section 4.1 and Appendix F for a detailed setup.

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).

This principle applies to many real-world neural network 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.

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*[3]. While the exact practical ramifications depend on the use case, Section 4.1 demonstrates that some of the larger models trained with pipeline parallelism can already train at peak efficiency with only hundreds of Mb/s bandwidth.

In theory, the square-cube principle also applies to intralayer parallelism, but using this technique at 500 Mb/s would become practical only for layer sizes of more than 2^{16} units. Data-parallel training with sharding or offloading (Ren et al., 2021) does not scale as well, as its communication time scales with the size of *model parameters* instead of activations. However, it may be possible to achieve similar scaling with gradient compression algorithms.

3.2. SWARM Parallelism

Traditional pipeline parallelism can be communication-efficient, but this alone is not enough for our setups. Since training devices can have 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 node fails or leaves training prematurely, it will stall the entire training procedure.

To overcome these two challenges, we replace the rigid pipeline structure with temporary "pipelines" that are built stochastically 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 is faster than others, it can process inputs from multiple predecessors and distribute its outputs across several weaker peers to maximize utilization. Also, if any participant disconnects, its predecessors can reroute their requests to its neighbors. New peers can download up-to-date parameters and optimizer statistics from remaining workers at the chosen stage. This allows the training to proceed as long as there is at least one active participant per stage: we elaborate on the fault tolerance of SWARM parallelism in Appendix A.

The resulting system consists of several consecutive swarms, as depicted in Figure 2. Peers within one swarm serve the same pipeline stage (i.e., the **same subset of layers** with **the same parameters**). We assume that the model consists of similar "blocks" and thus partition it into evenly sized stages, leaving the study of better strategies (Huang et al., 2019; Narayanan et al., 2019) as future work. During the *forward* pass, peers receive inputs from predecessors (determined on each iteration) and send activations to peers in the next stage. For the *backward* pass, peers receive gradients for outputs, compute gradients for layer inputs and accumulate gradients for parameters. Once enough gradients are accumulated, peers form groups, run All-Reduce to average gradients within their pipeline stages and perform the optimizer step.

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

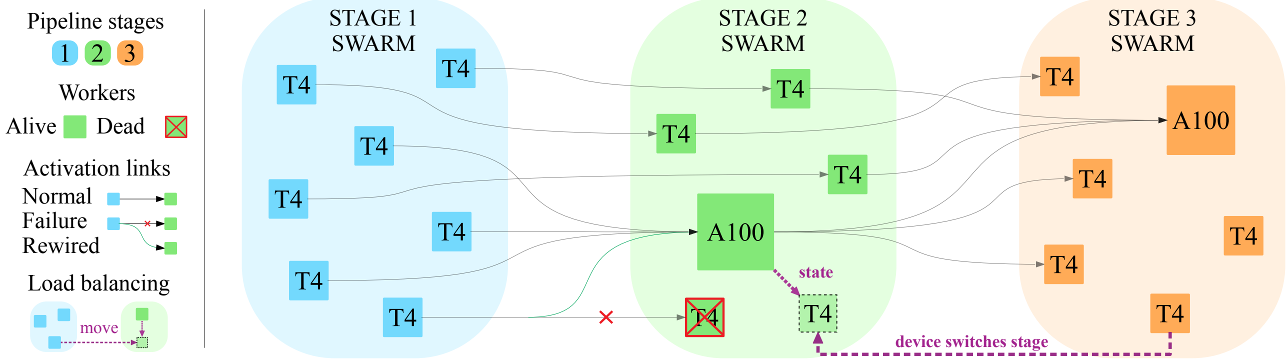


Figure 2: An overview of SWARM parallelism, illustrating both normal operation, device failures and adaptive rebalancing. One of the workers at stage 2 leaves; another peer from stage 3 takes its place by downloading the latest stage 2 parameters and statistics from peers.

SWARM parallelism can also use Delayed Parameter Updates (DPU) (Ren et al., 2021) to further improve hardware utilization by performing the optimizer step in parallel with processing the next batch. While it is technically asynchronous, DPU was shown to achieve similar per-iteration convergence as fully synchronous training, both theoretically (Stich & Karimireddy, 2020; Arjevani et al., 2020) and empirically (Ren et al., 2021; Diskin et al., 2021).

Each peer has queues for incoming and outgoing requests to maintain high GPU utilization under latency and to compensate for varying network speeds. Similarly to other pipeline implementations (Huang et al., 2019; Narayanan et al., 2021), SWARM parallelism uses activation checkpointing (Griewank & Walther, 2000; Chen et al., 2016) to reduce the memory footprint.

Stochastic wiring. To better utilize heterogeneous devices and recover from faults, we dynamically "wire" each input through each stage and pick devices in proportion to their training throughput. To achieve this, SWARM peers run "trainer" processes that route training data through the "stages" of SWARM, balancing the load between peers.

For each pipeline stage, trainers discover which peers currently serve this stage via a Distributed Hash Table (DHT, Maymounkov & Mazieres, 2002). Trainers then assign a microbatch to one of those peers based on their performance. If that peer fails, it is temporarily banned and the microbatch is sent to another peer within the same stage. Note that trainers themselves do not use GPUs and have no trainable parameters, which makes it possible to run multiple trainers per peer.

Each trainer assigns data independently using the Interleaved Weighted Round-Robin (Katevenis et al., 1991; Tabatabaee et al., 2020) scheduler. Our specific implementation of IWRR uses a priority queue: each peer is associated with *the total processing time over all previous requests*. A training minibatch is then routed to the node that has the smallest total processing time. Thus, for instance, if device A takes half as long to process a sample as device B, the routing algorithm will choose A twice as often as B. Finally, if a peer does not respond or fails to process the batch, trainer will "ban" this peer until it reannounces itself in the DHT, which is done every few minutes. For a more detailed description of stochastic wiring, please refer to Appendix C.

Curiously, different trainers can have different throughput estimates for the same device because of the network topology. For instance, if training nodes are split between two cloud regions, a given peer's trainer will have a 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".

Adaptive swarm rebalancing. While stochastic wiring allows for automatic rebalancing within a stage, additional cross-stage rebalancing may be required to maximize throughput, especially when devices are very unreliable. As we described 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 face 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 maximize the training throughput. Every T seconds, peers measure the utilization rate of each pipeline stage as the queue size. Peers from the most underutilized pipeline stage will then switch to the most overutilized one (see Figure 2 for an overview and Appendix D for a formal description and complexity analysis), download the latest training state from their new neighbors and continue training. Similarly, if a new peer joins midway through training, it is assigned to the optimal pipeline stage by following the same protocol. As a side effect, if one pipeline stage requires more compute than others, SWARM will allocate more peers to that stage. In Section 4.4, we evaluate our approach to dynamic rebalancing in realistic conditions.

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 the Transformer architecture (Vaswani et al., 2017) that has been widely adopted in various fields (Lin et al., 2022). 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. We use a batch size of 1 and sequences of 512 tokens; the complete configuration is deferred to Appendix F.

First, we measure how the model size affects the computation to communication ratio at 500 Mb/s network bandwidth in both directions. We consider 4 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., 2021), and a GPT-3-scale model with d_{model}=12288 (Brown et al., 2020). We also evaluate a modified Transformer architecture (“Ours”) as defined in Section 4.3 with d_{model}=4096, 3 layers per pipeline stage and 8-bit quantized activations. As we demonstrate in Appendix I, this compression strategy can significantly reduce network usage with little effect on convergence. 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. Appendix F contains FLOP and parameter counts of each configuration.

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

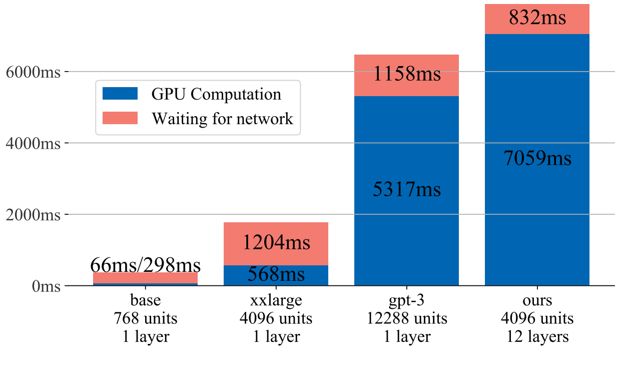


Figure 3: Pipeline computation and idle time per batch at 500 Mb/s bandwidth.

Table 1: Relative device utilization at 500 Mb/s bandwidth and varying 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%

4.2. 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. Still, even in such conditions the performance of different systems can vary across model architectures, and hence we want to identify the cases in which using SWARM is preferable to other approaches. We benchmark individual SWARM components in preemptible setups in Section 4.4 and Appendix H.

We evaluate training performance for sequences of 4 Transformer layers of identical size distributed over 16 workers. Similarly to Section 4.1, we use three layer configurations: “xxlarge” (d_{model}=4096, d_{FFN}=16384, 32 heads), “GPT-3” (d_{model}=12288, d_{FFN}=49152, 96 heads), and “Ours” (d_{model}=4096, d_{FFN}=16384, 32 heads, 16 shared layers per block, last stage holds only the vocabulary projection layer). The microbatch size is 4 for “xxlarge” and 1 for “GPT-3” and “Ours”, 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. More specifically, we measure the time required to process 6250 sequences of 512 tokens, which corresponds to the largest batch size used in Brown et al. (2020). In turn, the All-Reduce time is the time each system spends to aggregate accumulated gradients across devices. 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.

Table 2: Training performance for different model sizes.

System	Throughput, min/batch		All-Reduce time, min	
	No latency	Latency	No latency	Latency
"GPT-3" (4 layers)				
SWARM	168.3	**186.7**	7.4	**7.6**
GPipe	164.5	218.4	**6.7**	7.8
1F1B	**163.3**	216.1		
Offload	272.7	272.7	25.5	27.3
"xxlarge" (4 layers)				
SWARM	44.2	48.2	0.8	**0.9**
GPipe	40.1	108.8	**0.7**	1.1
1F1B	40.8	105.5		
Offload	**33.8**	**33.8**	2.8	4.2
Full "Ours" model (48 shared layers + embeddings)				
SWARM	432.2	452.9	0.8	**1.0**
GPipe	420.0	602.1	**0.7**	1.1
1F1B	408.5	569.2		
Offload	**372.0**	**372.0**	3.2	4.8

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 190 gigabytes at peak. Similarly to Section 4.1, each worker can communicate at a 500 Mb/s bandwidth for both upload and download for a total of 1 Gb/s. In terms of network latency, we consider two setups: with **no latency**, where workers communicate normally within the same rack, and with **latency**, where we introduce additional 100±50ms latency directly in the kernel[4].

GPipe configuration: We use a popular PyTorch-based implementation of GPipe[5]. 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. We evaluate both the standard GPipe schedule and the 1F1B schedule (Narayanan et al., 2019).

ZeRO-Offload configuration: Each worker runs the entire model individually, then exchanges gradients with peers. For "xxlarge", we use the official implementation from (Ren et al., 2021). However, for "GPT-3", we found that optimizer offloading still does not allow us to fit 4 layers into the GPU. For this reason, we also offload the model parameters using the `offload_param` option.

In turn, when training smaller models, ZeRO-Offload outperforms both SWARM and GPipe. This result aligns with our earlier observations in Figure 1, where the same model spent most of the time waiting for the 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. Overall, SWARM is competitive to HPC baselines even in an idealized homogeneous environment.

4.3. Large-Scale Distributed Training

To verify the efficiency of SWARM parallelism in a practical scenario, we conduct a series of large-scale distributed experiments using preemptible (unreliable) cloud T4 and A100 GPUs over a public cloud network.

We train a Transformer language model with the architecture similar to prior work (Brown et al., 2020; Wang & Komatsuzaki, 2021; Black et al., 2021) and 1.01 billion parameters in total. Our model consists of 3 stages, each containing a single Transformer decoder block with $d_{model} = 4096$ and 16 layers per pipeline stage. All workers within a stage serve the same group of layers, and all layers within each group use the same set of parameters, similarly to ALBERT (Lan et al., 2020). On top of this, the first stage also contains the embedding layer, and the last stage includes the language modeling head. Because of layer sharing, this model is equivalent to a 13B model from Brown et al. (2020) in terms of compute costs.

We use 8-bit compression (Dettmers et al., 2022) for activations and gradients to reduce the communication intensity. Additional training setup details are covered in Appendix G. SWARM nodes run rebalancing every $T = 300$ seconds, and trainers measure peer performance using a moving average with $\alpha = 0.1$. However, as we show in Section 4.4, the throughput of SWARM is not very sensitive to the choice of these hyperparameters.

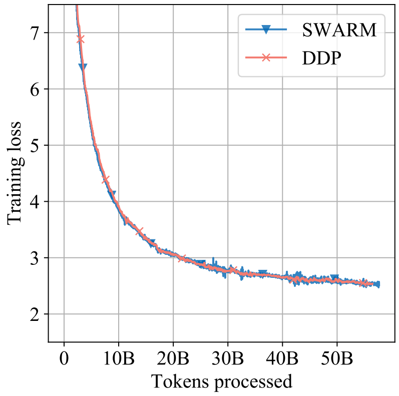


Figure 4: Training convergence comparison.

[4] More specifically, `tc qdisc add dev <...> root netem delay 100ms 50ms`

[5] The source code is available at `https://github.com/kakaobrain/torchgpipe`

Table 3: Pipeline throughput, layer sharing.

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	—	—	—

Table 4: Pipeline throughput, default Transformer.

Hardware setup	Throughput, samples/s	
	Actual	Best-case
T4	8.8	19.3
A100	8.0	25.1
T4 & A100	13.4	—

First, to verify that model parallelism with asynchronous updates does not have significant convergence issues, we train the model on the Pile (Gao et al., 2020) dataset with 400 preemptible T4 instances, each hosting one accelerator. As a baseline, we use regular data-parallel training with offloading on 128 A100 GPUs. We run both experiments for approximately 4 weeks and compare the learning curves.

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.

In the next experiment, we aim to measure the pipeline throughput in different hardware conditions and to compare it with an estimate of best-case pipeline performance. We consider several setups: first, we use the same 400 preemptible T4 nodes; in another setup, we use 7 instances with 8 A100 GPU each; finally, we combine these fleets to create a heterogeneous setup. We examine the performance of the pipeline both with weight sharing and with standard, more common, Transformer blocks.

We measure the number of randomly generated samples processed by the pipeline both in our infrastructure and the ideal case that ignores all network-related operations (i.e., has infinite bandwidth and zero latency). The ideal case is emulated by executing a single pipeline stage 3 times locally on a single server and multiplying the single-node estimates by the number of nodes.

As demonstrated in the left two columns of Table 3 and Table 4, asynchronous training of compute-intensive models with 8-bit compressed activations regardless of the architecture specifics 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.

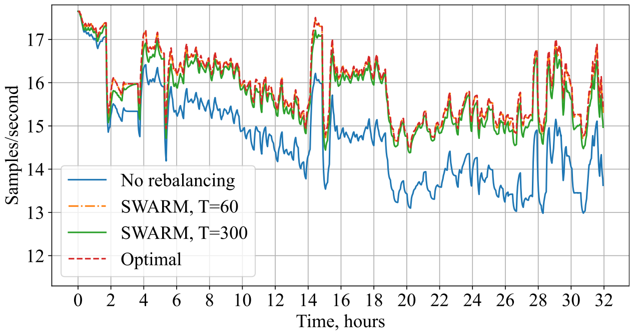


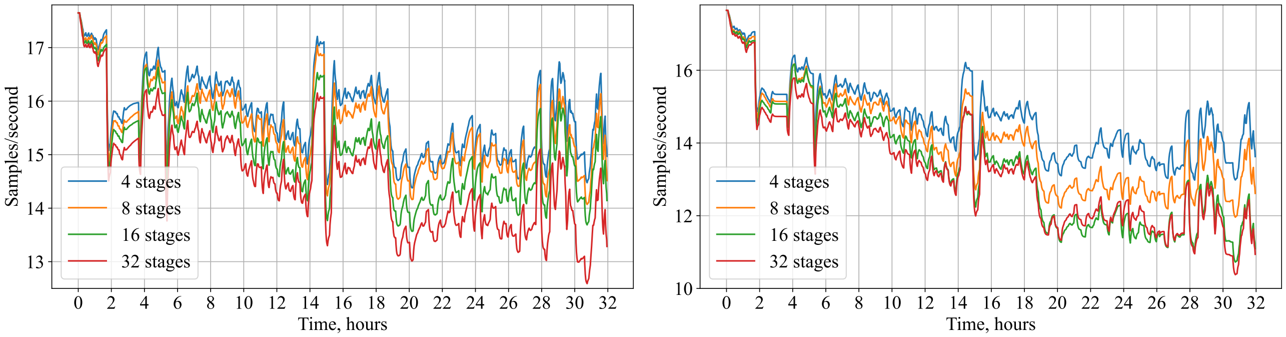
Figure 5: Throughput of rebalancing methods over time.

Next, 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. Based on our main experiment, the optimal total bandwidth is roughly 100Mb/s higher than the values reported in Table 3 due to gradient averaging, loading state from peers, maintaining the DHT and streaming the training data. 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 layer sharing, as shown in Table 3.

4.4. Adaptive Rebalancing Evaluation

In this experiment, we evaluate the efficiency of adaptive peer rebalancing between stages proposed in Section 3.2. We use statistics of the number of active T4 nodes from the 32-hour segment of the experiment described in Section 4.3. We use this data to simulate training dynamics by viewing it as sequence of events, each consisting of a timestamp and a 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. We run 10 simulations with different random seeds and average the resulting trajectories. We compare our strategy with two different values of T to the baseline that has no rebalancing.

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



(a) Adaptive rebalancing of SWARM parallelism.

(b) No rebalancing.

Figure 6: Scaling of pipeline-parallel strategies with respect to the number of stages.

possible throughput at every moment. It can be seen that even with the rebalancing period $T = 300$, our approach significantly improves the overall throughput of the pipeline. When the number of peers is relatively stable, the rebalanced pipeline also approaches the optimal one in terms of throughput, which shows the efficiency of rebalancing 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 the "overrepresented" stages affects the imbalanced pipeline less). However, this held true only for $\approx 4.5\%$ of the 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 volume. This is also demonstrated by Table 5, which shows the relative throughput of the three configurations compared to the optimal one. Furthermore, the table displays that while initially there is little difference between rebalancing choices, it becomes more pronounced later on as the imbalanced version "drifts further" from the optimal state.

Finally, we analyze the scaling properties of rebalancing with respect to the number of stages. To do this, we conduct experiments in the same setup as above ($T = 300$) while changing the number of pipeline stages from 4 to $\{4,\ 8,\ 16,\ 32\}$. To ensure the consistency of throughput across all experiments, we increase the starting number of peers accordingly while keeping the preemption rate constant. As a baseline, we also evaluate the throughput of the pipeline that has no rebalancing.

Table 5: Relative throughput comparison of pipeline rebalancing methods.

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

Figure 6 shows the outcome of this experiment. As displayed in the plots, both strategies drop in performance with the increase in the stage count: while all stages should drop in performance equally in expectation, in practice, the variances are too large while the number of peers is relatively too small for the asymptotic properties to take place. This effect results in more outliers (large drops in the number of peers) in the preemption distribution for more stages. Still, rebalancing allows to partially mitigate the issue: while we observe a more consistent downward trend for the baseline strategy, the rebalanced pipeline regains its performance over time and achieves a higher overall throughput.

5. Conclusion

In this work, we evaluate the feasibility of high-throughput training of billion-scale neural networks on unreliable peers with low network bandwidth. We find that training in this setup can be possible with very large models and pipeline parallelism. To this end, we propose SWARM parallelism to overcome the challenges of pipeline parallelism for preemptible devices with heterogeneous network bandwidths and computational throughputs. We show that our method is highly effective at rebalancing peers and maximizing the aggregate training throughput even in presence of unstable nodes. We also show that training **large models** with **SWARM parallelism** and **compression**-aware architectures enables high utilization of cheap preemptible instances with slow interconnect. As such, our work makes training of large models accessible to researchers that do not have access to dedicated compute infrastructure.

References

Aji, A. F. and Heafield, K. Making asynchronous stochastic gradient descent work for transformers. In *Proceedings of the 3rd Workshop on Neural Generation and Translation*, pp. 80–89, Hong Kong, 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-5608. URL https://aclanthology.org/D19-5608.

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

Alman, J. and Williams, V. V. A refined laser method and faster matrix multiplication. In Marx, D. (ed.), *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pp. 522–539. SIAM, 2021. doi: 10.1137/1.9781611976465.32. URL https://doi.org/10.1137/1.9781611976465.32.

Arjevani, Y., Shamir, O., and Srebro, N. A tight convergence analysis for stochastic gradient descent with delayed updates. In Kontorovich, A. and Neu, G. (eds.), *Proceedings of the 31st International Conference on Algorithmic Learning Theory*, volume 117 of *Proceedings of Machine Learning Research*, pp. 111–132. PMLR, 2020. URL https://proceedings.mlr.press/v117/arjevani20a.html.

Atre, M., Jha, B., and Rao, A. Distributed deep learning using volunteer computing-like paradigm. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021*, pp. 933–942. IEEE, 2021. doi: 10.1109/IPDPSW52791.2021.00144. URL https://doi.org/10.1109/IPDPSW52791.2021.00144.

Ba, L. J., Kiros, J. R., and Hinton, G. E. Layer normalization. *ArXiv preprint*, abs/1607.06450, 2016. URL https://arxiv.org/abs/1607.06450.

Baevski, A. and Auli, M. Adaptive input representations for neural language modeling. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=ByxZX20qFQ.

Baines, M., Bhosale, S., Caggiano, V., Goyal, N., Goyal, S., Ott, M., Lefaudeux, B., Liptchinsky, V., Rabbat, M., Sheiffer, S., Sridhar, A., and Xu, M. Fairscale: A general purpose modular pytorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale, 2021.

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

Black, S., Leo, G., Wang, P., Leahy, C., and Biderman, S. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, 2021. URL https://doi.org/10.5281/zenodo.5297715.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *ArXiv preprint*, abs/1604.06174, 2016. URL https://arxiv.org/abs/1604.06174.

Chilimbi, T., Suzue, Y., Apacible, J., and Kalyanaraman, K. 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, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. PaLM: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022. doi: 10.48550/arXiv.2204.02311. URL https://doi.org/10.48550/arXiv.2204.02311.

Coates, A., Huval, B., Wang, T., Wu, D. J., Catanzaro, B., and Ng, A. Y. Deep learning with COTS HPC systems. 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. 1337–1345. JMLR.org, 2013. URL http://proceedings.mlr.press/v28/coates13.html.

Coppersmith, D. and Winograd, S. 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.

Dai, Z., Liu, H., Le, Q. V., and Tan, M. Coatnet: Marrying convolution and attention for all data sizes. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 3965–3977, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/20568692db622456cc42a2e853ca21f8-Abstract.html.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A. W., Tucker, P. A., Yang, K., and Ng, A. Y. Large scale distributed deep networks. In Bartlett, P. L., Pereira, F. C. N., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pp. 1232–1240, 2012. URL https://proceedings.neurips.cc/paper/2012/hash/6aca97005c68f1206823815f66102863-Abstract.html.

Dettmers, T. 8-bit approximations for parallelism in deep learning. In Bengio, Y. and LeCun, Y. (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/1511.04561.

Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L. 8-bit optimizers via block-wise quantization. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=shpkpVXzo3h.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Dhariwal, P. and Nichol, A. Q. Diffusion models beat gans on image synthesis. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 8780–8794, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/49ad23d1ec9fa4bd8d77d02681df5cfa-Abstract.html.

Diskin, M., Bukhtiyarov, A., Ryabinin, M., Saulnier, L., Lhoest, Q., Sinitsin, A., Popov, D., Pyrkin, D. V., Kashirin, M., Borzunov, A., del Moral, A. V., Mazur, D., Kobelev, I., Jernite, Y., Wolf, T., and Pekhimenko, G. Distributed deep learning in open collaborations. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 7879–7897, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/41a60377ba920919939d83326ebee5a1-Abstract.html.

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

Fatahalian, K., Sugerman, J., and Hanrahan, P. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. pp. 133–137, 2004. doi: 10.1145/1058129.1058148.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *ArXiv preprint*, abs/2101.03961, 2021. URL https://arxiv.org/abs/2101.03961.

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

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

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., Presser, S., and Leahy, C. The pile: An 800gb dataset of diverse text for language modeling, 2020.

Gokaslan, A. and Cohen, V. Openwebtext corpus, 2019. URL http://Skylion007.github.io/OpenWebTextCorpus.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A. C., and Bengio, Y. 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.

Griewank, A. and Walther, A. 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.

Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., and Gibbons, P. B. 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.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pp. 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

He, P., Liu, X., Gao, J., and Chen, W. Deberta: decoding-enhanced bert with disentangled attention. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=XPZIaotutsD.

Hochreiter, S. and Schmidhuber, J. 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).

Huang, J., Yu, C. D., and Geijn, R. A. v. d. Strassen's algorithm reloaded on gpus. *ACM Trans. Math. Softw.*, 46(1), 2020. ISSN 0098-3500. doi: 10.1145/3372419. URL https://doi.org/10.1145/3372419.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M. X., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (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. 103–112, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html.

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

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. In Talwalkar, A., Smith, V., and Zaharia, M. (eds.), *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019. URL https://proceedings.mlsys.org/book/265.pdf.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models, 2020.

Katevenis, M., Sidiropoulos, S., and Courcoubetis, C. 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.

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

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

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In Bartlett, P. L., Pereira, F. C. N., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pp. 1106–1114, 2012. URL https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

Lample, G., Sablayrolles, A., Ranzato, M., Denoyer, L., and Jégou, H. Large memory layers with product keys. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (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. 8546–8557, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/9d8df73a3cfbf3c5b47bc9b50f214aff-Abstract.html.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. ALBERT: A lite BERT for self-supervised learning of language representations. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=H1eA7AEtvS.

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

Larrea, V. G. V., Joubert, W., Brim, M. J., Budiardja, R. D., Maxwell, D., Ezell, M., Zimmer, C., Boehm, S., Elwasif, W. R., Oral, S., Fuson, C., Pelfrey, D., Hernandez, O. R., Leverman, D., Hanley, J., Berrill, M. A., and Tharrington, A. N. Scaling the summit: Deploying the world's fastest supercomputer. In Weiland, M., Juckeland, G., Alam, S. R., and Jagode, H. (eds.), *High Performance Computing - ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers*, volume 11887 of *Lecture Notes in Computer Science*, pp. 330–351. Springer, 2019. doi: 10.1007/978-3-030-34356-9_26. URL https://doi.org/10.1007/978-3-030-34356-9_26.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=qrwe7XHTmYb.

Li, C., Zhang, M., and He, Y. Curriculum learning: A regularization method for efficient and stable billion-scale GPT model pre-training. *ArXiv preprint*, abs/2108.06084, 2021. URL https://arxiv.org/abs/2108.06084.

Li, S., Walls, R. J., Xu, L., and Guo, T. Speeding up deep learning with transient servers. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 125–135. IEEE, 2019.

Li, S., Ben-Nun, T., Nadiradze, G., Digirolamo, S., Dryden, N., Alistarh, D., and Hoefler, T. 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.

Lian, X., Zhang, C., Zhang, H., Hsieh, C., Zhang, W., and Liu, J. Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5330–5340, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/f75526659f31040afeb61cb7133e4e6d-Abstract.html.

Lin, J., Li, X., and Pekhimenko, G. Multi-node bert-pretraining: Cost-efficient approach, 2020.

Lin, T., Wang, Y., Liu, X., and Qiu, X. A survey of transformers. *AI Open*, 3:111–132, 2022. doi: 10.1016/j.aiopen.2022.10.001. URL https://doi.org/10.1016/j.aiopen.2022.10.001.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, B. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=SkhQHMW0W.

Maymounkov, P. and Mazieres, D. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pp. 53–65. Springer, 2002.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=Byj72udxe.

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*, 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.

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. *ArXiv preprint*, abs/2104.04473, 2021. URL https://arxiv.org/abs/2104.04473.

Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pp. 48–53, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-4009. URL https://aclanthology.org/N19-4009.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (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. 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Pudipeddi, B., Mesmakhosroshahi, M., Xi, J., and Bharadwaj, S. Training large neural networks with constant memory using a new execution algorithm. *ArXiv preprint*, abs/2002.05645, 2020. URL https://arxiv.org/abs/2002.05645.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training. 2018. URL https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., Rutherford, E., Hennigan, T., Menick, J., Cassirer, A., Powell, R., van den Driessche, G., Hendricks, L. A., Rauh, M., Huang, P.-S., Glaese, A., Welbl, J., Dathathri, S., Huang, S., Uesato, J., Mellor, J., Higgins, I., Creswell, A., McAleese, N., Wu, A., Elsen, E., Jayakumar, S., Buchatskaya, E., Budden, D., Sutherland, E., Simonyan, K., Paganini, M., Sifre, L., Martens, L., Li, X. L., Kuncoro, A., Nematzadeh, A., Gribovskaya, E., Donato, D., Lazaridou, A., Mensch, A., Lespiau, J.-B., Tsimpoukelli, M., Grigorev, N., Fritz, D., Sottiaux, T., Pajarskas, M., Pohlen, T., Gong, Z., Toyama, D., de Masson d'Autume, C., Li, Y., Terzi, T., Mikulik, V., Babuschkin, I., Clark, A., de Las Casas, D., Guy, A., Jones, C., Bradbury, J., Johnson, M., Hechtman, B., Weidinger, L., Gabriel, I., Isaac, W., Lockhart, E., Osindero, S., Rimell, L., Dyer, C., Vinyals, O., Ayoub, K., Stanway, J., Bennett, L., Hassabis, D., Kavukcuoglu, K., and Irving, G. Scaling language models: Methods, analysis & insights from training gopher, 2021.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimization towards training a trillion parameter models. In *SC*, 2020.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. *ArXiv preprint*, abs/2104.07857, 2021. URL https://arxiv.org/abs/2104.07857.

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., and Sutskever, I. Zero-shot text-to-image generation. In Meila, M. and Zhang, T. (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. 8821–8831. PMLR, 2021. URL http://proceedings.mlr.press/v139/ramesh21a.html.

Recht, B., Ré, C., Wright, S. J., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F. C. N., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, pp. 693–701, 2011. URL https://proceedings.neurips.cc/paper/2011/hash/218a0aefd1d1a4be65601cc6ddc1520e-Abstract.html.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. Zero-offload: Democratizing billion-scale model training, 2021.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

Ryabinin, M. and Gusev, A. Towards crowdsourced training of large neural networks using decentralized mixture-of-experts. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/25ddc0f8c9d3e22e03d3076f98d83cb2-Abstract.html.

Ryabinin, M., Gorbunov, E., Plokhotnyuk, V., and Pekhimenko, G. Moshpit SGD: communication-efficient decentralized training on heterogeneous unreliable devices. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 18195–18211, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/97275a23ca44226c9964043c8462be96-Abstract.html.

Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1715–1725, Berlin, Germany, 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162.

Shazeer, N. GLU variants improve transformer. *ArXiv preprint*, abs/2002.05202, 2020. URL https://arxiv.org/abs/2002.05202.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=B1ckMDqlg.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., and Hechtman, B. A. Mesh-tensorflow: Deep learning for supercomputers. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pp. 10435–10444, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/3a37abdeefe1dab1b30f7c5c7e581b93-Abstract.html.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *ArXiv preprint*, abs/1909.08053, 2019. URL https://arxiv.org/abs/1909.08053.

Stich, S. U. and Karimireddy, S. P. The error-feedback framework: Sgd with delayed gradients. *Journal of Machine Learning Research*, 21(237):1–36, 2020. URL http://jmlr.org/papers/v21/19-748.html.

Strohmaier, E., Dongarra, J., Simon, H., and Meuer, M. Fugaku. https://www.top500.org/system/179807/, 2021. Estimated energy consumption 29,899.23 kW. Accessed: 2021-10-4.

Su, J., Lu, Y., Pan, S., Wen, B., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding, 2021.

Sun, Y., Wang, S., Feng, S., Ding, S., Pang, C., Shang, J., Liu, J., Chen, X., Zhao, Y., Lu, Y., Liu, W., Wu, Z., Gong, W., Liang, J., Shang, Z., Sun, P., Liu, W., Ouyang, X., Yu, D., Tian, H., Wu, H., and Wang, H. ERNIE 3.0: Large-scale knowledge enhanced pre-training for language understanding and generation. *ArXiv preprint*, abs/2107.02137, 2021. URL https://arxiv.org/abs/2107.02137.

Tabatabaee, S. M., Le Boudec, J.-Y., and Boyer, M. 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.

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

Tarnawski, J., Narayanan, D., and Phanishayee, A. Piper: Multidimensional planner for DNN parallelization. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 24829–24840, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/d01eeca8b24321cd2fe89dd85b9beb51-Abstract.html.

Thorpe, J., Zhao, P., Eyolfson, J., Qiao, Y., Jia, Z., Zhang, M., Netravali, R., and Xu, G. H. Bamboo: Making preemptible instances resilient for affordable training of large dnns, 2022.

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

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

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

Vogels, T., Karimireddy, S. P., and Jaggi, M. Powersgd: Practical low-rank gradient compression for distributed optimization. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (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.

Wang, B. and Komatsuzaki, A. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 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 quantization with guarantees. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=QDPonrGtl1.

Wang, S., Bai, Y., and Pekhimenko, G. BPPSA: scaling back-propagation by parallel scan algorithm. In Dhillon, I. S., Papailiopoulos, D. S., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020. URL https://proceedings.mlsys.org/book/317.pdf.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. 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, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6.

Yang, B., Zhang, J., Li, J., Ré, C., Aberger, C. R., and Sa, C. D. Pipemare: Asynchronous pipeline parallel dnn training. *ArXiv*, abs/1910.05124, 2019.

You, Y., Li, J., Reddi, S. J., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., and Hsieh, C. Large batch optimization for deep learning: Training BERT in 76 minutes. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=Syx4wnEtvH.

Yuan, B., He, Y., Davis, J. Q., Zhang, T., Dao, T., Chen, B., Liang, P., Re, C., and Zhang, C. Decentralized training of foundation models in heterogeneous environments. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing*

Systems, 2022. URL https://openreview.net/forum?id=UHoGOaGjEq.

Zhai, X., Kolesnikov, A., Houlsby, N., and Beyer, L. Scaling vision transformers. *ArXiv preprint*, abs/2106.04560, 2021. URL https://arxiv.org/abs/2106.04560.

Zhang, P. and Gao, Y. Matrix multiplication on high-density multi-gpu architectures: Theoretical and experimental investigations. In Kunkel, J. M. and Ludwig, T. (eds.), *High Performance Computing - 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings*, volume 9137 of *Lecture Notes in Computer Science*, pp. 17–30. Springer, 2015. doi: 10.1007/978-3-319-20119-1_2. URL https://doi.org/10.1007/978-3-319-20119-1_2.

Zhang, X., Wang, J., Joshi, G., and Joe-Wong, C. Machine learning on volatile instances. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pp. 139–148. IEEE, 2020.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., Gonzalez, J. E., and Stoica, I. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning, 2022. URL https://arxiv.org/abs/2201.12023.

Supplementary Material

This part of the paper is organized as follows:

- Appendix A overviews several common questions about the details of our study and addresses the limitations of SWARM parallelism;
- In Appendix B, we list further related works on topics relevant to the problem setting we study;
- In Appendix C and Appendix D, we give a more formal description and outline the details of stochastic wiring and adaptive rebalancing, accordingly;
- In Appendix E, we outline the relation between training with SWARM and using methods for offloading.
- Appendix F and Appendix G contain additional details of our experimental setup, whereas Appendix H reports further experiments on specific aspects and components of SWARM parallelism;
- Lastly, we investigate compression-aware architectures in Appendix I and evaluate their impact in a practical setting in Appendix J.

A. Answers to Common Questions

Why not just use data parallelism with offloading? Regular data parallelism requires all-reduce steps where peers exchange gradients, which can be prohibitively expensive for large models. For example, a 1 billion 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 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 communications can be done in parallel and has a size of parameter count divided by n; in total, n messages are required. Thus, for 1B parameters in 16-bit precision, a total of 2 GB need to be synchronized for both the forward and backward pass. For low-bandwidth devices with 100 Mb/s speed, this would entail an overhead of 5.5 minutes per forward/backward pass, which is difficult to overlap with computation. This is exacerbated further, because all-to-all communication latency is determined by the slowest peer. Thus, sharded data parallelism can be particularly inefficient for setups where peers have different network bandwidths.

Should I use SWARM in a supercomputer? By default, SWARM is worse than traditional parallelism due to its extra complexity (see experiments in Section 4.2). However, SWARM can be useful in case of supercomputers that have heterogeneous devices.

ZeRO-Offload allows one to train 13B parameters on a single V100, so why do I need SWARM? Using ZeRO-Offload can slow down training due to the slow data transfer between external memory and the accelerator. Training with SWARM can *accelerate* training while also allowing to train larger models; see Appendix E for a detailed comparison.

Is it worth using preemptible instances and SWARM from an economic standpoint? Due to a significantly smaller cost per hour, one can leverage a larger amount of computation when using spot instances compared to on-demand cloud VMs or dedicated HPC setups. See Appendix J and Table 9 for a comparison of both hourly and total costs for an example large-scale pretraining task.

When should I avoid using SWARM? SWARM is efficient at training compute-intensive models with more than 1B parameters. For smaller models, a sharded data-parallel approach can be more optimal. For homogeneous HPC environments, standard sharded data-parallel or pipeline-parallel training will be more efficient than SWARM, because the rebalancing is not required. For HPC environments that are so extensive that the failure of a node is likely, the practicality 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.

Can I use SWARM without layer sharing or quantization? Yes, SWARM can still be effective in these scenarios. Our bandwidth experiments in the main part of the work give an estimate of its network overhead. By using no quantization, which means using regular 16-bit activations, the network overhead increases approximately by a factor of two. Without layer sharing, the overhead within each pipeline stage to synchronize the gradients 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 and 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 (for example, CNNs vs. RNNs vs. Transformers).

How many pipeline stages can SWARM have? While its design allows for any number of stages, using long pipelines can result in a reduced training throughput. Similarly to regular pipeline parallelism, SWARM suffers from the pipeline "bubble" problem (Huang et al., 2019): at the beginning of the initial batch processing, peers near the end of the pipeline will be waiting for inputs. Likewise, early layers will be idle after processing the final microbatch. In theory, this can be mitigated with asynchronous updates (Narayanan et al., 2019; Yang et al., 2019), but we did not investigate them in this work due to potential convergence issues.

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 can work without any issues. The key factors defining the training run state at a given SGD step are the model parameters, the optimizer statistics, the data loader state, and the step number (required for proper scheduling). The up-to-date parameters and optimizer statistics, as well as the step number, are naturally located on all active nodes of a given stage, since they are required for training. Thus, when a peer joins the network, it can download the checkpoint corresponding to the current training state from other peers.

As we mention in Section 3.2, peer failures do not affect forward and backward passes as long as there is at least one peer at the required stage: because of rewiring, it is possible to resend activations or gradients to another worker that has identical model weights by construction. Similarly, the data loader state can be recomputed from the last known SGD step. However, we do not track the order of examples sampled within the same batch; because of the i.i.d. assumption in the large-scale training setup, the distribution of gradients is expected to be the same. Hence, if the peer leaves from the pipeline stage, other workers can compute gradients and replace those accumulated by the disconnected peer, so that the number of examples for an SGD step stays the same.

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 the GPU is running the model, regardless of efficiency. Since any realistic deep learning workload cannot achieve 100% peak FLOP/s, 20% GPU idle time for SWARM means that it can reach $\approx$ 0.8x the training throughput compared to training with an infinitely fast network. As a rule of thumb, one can say that SWARM will run at a 20% slower speed than systems described by Narayanan et al. (2021) using the infrastructure that is several times cheaper.

B. Additional Related Work

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" parameters 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 n devices such that each device stores $\frac{1}{n}$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 so that only the sharded memory remains ($\frac{1}{n}$th 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 finetuning 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., 2021), 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.

Optimal scheduling for distributed training. When the configuration of each peer is known, it is possible to significantly optimize the pipeline scheduling by going beyond the greedy approach with global optimization techniques (Zheng et al., 2022; Tarnawski et al., 2021), even with heterogeneous hardware (Yuan et al., 2022). However, we consider a setup in which this is not possible: preemptible and volunteer peers can join at any point of the experiment, and dynamically rescheduling and orchestrating them in a centralized manner is technically difficult because of the communication and reliability constraints.

Elastic training. To train with a dynamic number of workers, deep learning practitioners have developed elastic training algorithms (TorchElastic; ElasticHorovod). If a worker leaves or fails during training, these algorithms rebalance the load between the remaining nodes and continue the training procedure (Harlap et al., 2017; Ryabinin et al., 2021). If new workers join during training, they get the latest model parameters from their peers and train alongside them.

Asynchronous training. Another important problem is distributed training on devices with uneven performance. One way to solve this problem is to use asynchronous training, where nodes compute gradients at their own pace and aggregate them using a parameter server (Recht et al., 2011; Kijsipongse et al., 2018) or a decentralized network (Lian et al., 2017). This idea allows full utilization of each device, but may reduce the convergence rate due to "stale" gradients (Recht et al., 2011; Aji & Heafield, 2019). 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 points while maintaining the per-iteration convergence.

C. 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 random devices from each pipeline stage, such that the workload of each device is proportional to its performance. The performance of the peer is measured as an exponentially weighted average of its response time, and all peers serving a specific stage are stored in a priority queue. We formally describe the components of stochastic wiring in Algorithm 1.

From a system design perspective, each worker 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) (IWRR) scheduling to dynamically assign microbatches to peers based on each peer's training throughput ("samples per second") in a balanced way.

An 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.

D. Description and Complexity of Adaptive Rebalancing

Algorithm 2 contains the formal definition of the adaptive rebalancing procedure. As described previously, each worker of SWARM that hosts model layers continuously updates the information about its load in parallel with processing the incoming requests. Each T seconds, the peers measure the total load for all stages of the pipeline, and the peer with the lowest queue size from the stage with the minimum load moves to the stage with the maximum load. In principle, the algorithm could be extended to support moving multiple peers simultaneously; however, as we have shown in Section 4.4, even in the current form the algorithm bridges most of the gap between the optimally balanced pipeline and the system without any rebalancing.

Algorithm 1 Pseudocode of stochastic wiring

```
input the number of pipeline stages $N$, the set of active
      servers $S$, smoothing parameter $\gamma$, initial priority $\epsilon$
▷ Initialization
ema = dict()
queues = list()
for i $\in 1, \ldots, N$ do
    queues.append(PriorityQueue())
end for
def add_server(server):
    ema[server] = $\varepsilon$
    for i $\in$ get_blocks_served_by(server):
        queues[i].update(server, priority=$\varepsilon$)
def ban_server(server) :
    for i $\in$ get_blocks_served_by(server):
        queues[i].update(server, priority=$\infty$)
def choose_server(i):
    server, priority = queues[i].top()
    new_priority = priority + ema[server]
    for j $\in$ get_blocks_served_by(server) :
        queues[j].update(server, priority=new_priority)
    return server
▷ Forward pass with stochastic wiring
def forward(inputs):
    layer_index = 0
    while layer_index $< N$:
        server = choose_server(layer_index)
        t = get_current_time()
        try:
            inputs = server.forward(inputs)
            layer_index = layer_index + 1
            $\Delta t$ = get_current_time() - t
            ema[server] = $\gamma \cdot \Delta t + (1 - \gamma)\cdot$ ema[server]
        catch (ServerFault, Timeout):
            ban_server(server)
    return inputs
```

The complexity of Algorithm 2 can be estimated as follows: for M as the highest number of peers over all stages, we have $O(M)$ operations in Lines 9–11 and Lines 22–24, and all other operations take constant time for a single stage. These operations are nested in the loop over all stages, which means that the total complexity of the algorithm is $O(MS)$. For practical numbers of both peers (e.g., < 10,000) and stages (fewer than 100), this incurs a negligible overhead on performance, as all communication and computation is done in parallel with the actual forward and backward passes.

Also, notice that only one migrating peer needs to stop processing requests and download the weights and optimizer statistics of the pipeline stage it starts serving: this means that the overall network load of this procedure is relatively small, as all DHT requests handle scalar data and do not exceed the number of active peers for each worker.

In practice, the algorithm handles slight deviations in local time and network/DHT latencies by allowing the peers to wait for straggling nodes in Line 9 for a predefined timeout. If a node does not join the rebalancing procedure by reporting its load in time or joins the network too late, it is omitted from the current iteration.

Algorithm 2 Adaptive rebalancing for SWARM parallelism

```
input peer index $i$, current peer stage $s_{cur}$, total number of
    stages $S$, rebalancing period $T$
 1: while active do
 2:   Sleep for $T$ seconds
 3:   Measure $q_i$ as the local request queue size
 4:   Write $(i, q_i)$ as the key-subkey pair to DHT[$s_{cur}$]
 5:   Initialize minimum and maximum load stages:
      $s_{min} = s_{max} := -1$,
 6:   $l_{min} := \infty, l_{max} := -\infty$
 7:   for $s$ in $1, \ldots, S$ do
 8:     Initialize the load buffer $L = 0$
 9:     for $(j, q_j)$ in DHT[$s$] do
10:       $L := L + q_j$
11:     end for
12:     if $L > L_{max}$ then
13:       $s_{max} := s,\ L_{max} := L$
14:     end if
15:     if $L < L_{min}$ then
16:       $s_{min} := s,\ L_{min} := L$
17:     end if
18:   end for
19:   if $s_{cur} = s_{min}$ then
20:     // Migrate to the maximum load stage
21:     Initialize the minimum load peer $i_{min} :=$
        $-1, q_{min} := \infty$
22:     for $(j, q_j)$ in DHT[$s$] do
23:       if $q_j < q_{min}$ then
24:         $i_{min} := j,\ q_{min} := q_j$
25:       end if
26:     end for
27:     if $i_{min} = i$ then
28:       // This peer should migrate
29:       $s_{cur} := s_{max}$
30:       Download up-to-date parameters from peers in
          $s_{max}$
31:     end if
32:   end if
33: end while
```

E. 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 the semi-synchronous training proposed in 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 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 reach the required global batch size (which is a hyperparameter). While different peers may accumulate a different number of samples, they will all use the same gradient after averaging. Any peer that fails or does not meet this condition is considered a straggler and must reload its state from neighbors before it can resume training. This procedure ensures that all surviving peers use non-stale aggregated gradients over the specified batch size when performing 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 was shown not to affect convergence (Ren et al., 2021; Stich & Karimireddy, 2020; Arjevani et al., 2020), one can disable this functionality and make SWARM fully equivalent to standard training.

Naturally, these guarantees come at the cost of reduced hardware 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. Additional Details for Section 4.1

We benchmark four versions of the Transformer layer:

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

In Table 6, we report FLOP and parameter counts of each version based on the expressions from (Kaplan et al., 2020). For simplicity, we set up each experiment with 12 Transformer layers using 12 servers (4 for "Ours") with a single V100-PCIE GPU each. The servers communicate at 500Mbps under 3–6ms latency.

Due to a 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. If we colocate 3 "GPT-3" layers per pipeline stage, the GPU utilization can further improved to $> 90\%$.

The time reported in Section 4.1 is the time required to run forward and backward pass for all layers with a batch of 1x512 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 on different servers. Each layer is a `TransformerEncoderLayer` from PyTorch 1.7.0 (Paszke et al., 2019) wrapped with activation checkpointing. We use `hivemind==0.8.15` (Ryabinin & Gusev, 2020) with a single synchronous trainer based on the BERT training code from the Transformers library (Wolf et al., 2020). 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 that the TCP buffer does not have to warm up during each call. Also, our implementation performs quantization asynchronously with communication and other computations.

Table 6: Parameter and FLOP counts of each architecture.

Architecture	Parameters	FLOP count
"base"	7.08M	2.2×10^{10}
"xxlarge"	201M	6.2×10^{11}
"GPT-3"	1.81B	5.5×10^{12}
"Ours"	201M	1.8×10^{12}

G. Additional Details for Section 4.3

We use the standard Transformer architecture with two modifications: Rotary Positional Embeddings (Su et al., 2021) and GeGLU activations (Shazeer, 2020). Similarly to other models trained on Pile (Gao et al., 2020; Wang & Komatsuzaki, 2021), we use the tokenizer of GPT-2 (Radford et al., 2019). Following (Li et al., 2021), we linearly increase training sequence length during the initial phase. More specifically, we begin training with sequences of up to 256 tokens and increase them to the maximum length of 2048 over the first $12,000$ optimizer steps. We train the model with LAMB (You et al., 2020), following the configuration from the original paper for a batch size of 16384. On top of that, we set $\eta = 10^{-3}$ and $\beta_2 = 0.95$ to account for the increased model size.

H. Additional Scaling Evaluation

In this experiment, we investigate the influence of the number of nodes training with SWARM parallelism on the throughput of the pipeline. Specifically, we measure the performance of training the same model as in Section 4.3 in several configurations that differ in the size of the data-parallel group at each pipeline stage, with the number of single-GPU instances ranging from 8 to 128 (the highest quantity of preemptible nodes that we could reliably maintain for a long time). To isolate the effect of worker heterogeneity, here we use only the T4 accelerators and measure the average performance over 30 minutes of training.

Figure 7 shows the results of our evaluation. It can be seen that the training performance exhibits an approximately linear scaling pattern, which can be explained by the high efficiency of both the stochastic wiring strategy and the auxiliary training components such as the DHT and the All-Reduce protocol used for gradient averaging.

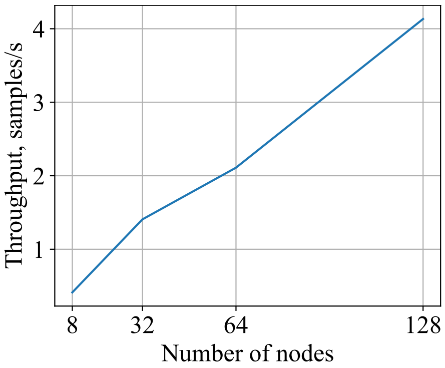


Figure 7: Scaling of SWARM parallelism throughput with the number of nodes.

I. Compression-Aware Architectures

Since pipeline parallelism has several distinct points of communication, the network overhead can be reduced considerably by reducing the size of data at these communication points. To exploit this, we develop compression-aware architectures that apply extreme compression at these points. We study two distinct communication bottleneck layers: (1) compression through a linear bottleneck layer, and (2) compression through a bottleneck induced by the maxout activation function (Goodfellow et al., 2013). We also study how compressing the activations and gradients at the communication points to 8 bits affects the predictive performance.

I.1. Description

Fully connected layers (baseline): Fully connected layers in models such as Transformers consist of a multilayer perceptron with a single hidden layer and a nonlinear activation function. Without biases and with a residual connection (He et al., 2016) 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 $\mathbf{x} \in \mathbb{R}^{b\times s\times m}$, $\mathbf{w}_1 \in \mathbb{R}^{m\times h}$, $\mathbf{w}_2 \in \mathbb{R}T^{h\times m}$, and $\sigma(\cdot)$ is a nonlinear activation function such as ReLU (Krizhevsky et al., 2012); b, s, m, and h are the batch, sequence, model, and hidden dimensions of the neural network. To compress the output of the MLP layer, we want to apply a compression layer between two consecutive stages. For example, if we have 24 layers and 4 stages, we need 3 compression layers at layers 6, 12, and 18.

Quantized activations: A natural way to reduce the communication intensity is to send activations and gradients with respect to activations in reduced precision. However, simply casting tensors to a lower precision may slow down convergence and cause instabilities. Instead, we use dynamic 8-bit quantization with blockwise scaling from (Dettmers et al., 2022). This technique reduces communication by ≈2x and ≈4x for half and full precision, respectively.

On the other hand, quantizing and dequantizing activations can add compute overhead on every microbatch processed. Our implementation circumvents that overhead by performing quantization asynchronously on the CPU. However, this is not required, as blockwise (de)quantization takes less than 1% of total computation time: see Appendix J for details.

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

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

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 layer normalization (Ba et al., 2016) to ensure training without divergence. The parameter matrix $\mathbf{w}_c$ resides in one stage and its outputs are transferred to the next stage that holds the parameters $\mathbf{w}_d$, 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 (less than 1% for typical sizes, see Appendix J).

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 a factor of k by computing the maximum value for each non-overlapping window of k features. We use maxout compression as follows:

$$\begin{gathered}\text{Maxout}(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_d) = \\ \text{LayerNorm}(\text{maxout}_k(\text{LayerNorm}(\text{MLP}(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_2))))\mathbf{w}_{\mathbf{d}},\end{gathered}$$

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

I.2. Evaluating the Speed-Quality Tradeoff

While compression techniques reduce the communication overhead, they might also degrade the perplexity reached in a certain time and the final perplexity after a specific number of steps. To study these tradeoffs, we train a Transformer language model with adaptive inputs (Baevski & Auli, 2019) on the WikiText-103 dataset and measure how compression-aware architecture variants affect convergence.

Our setup follows that of (Baevski & Auli, 2019) with one difference: we use a sequence length of 2048 instead of 3072 to fit this model into our smaller GPUs. To measure the time to solution, we look at the number of iterations it takes to converge to the training perplexity of **22**. We evaluate the baseline model and three compression-aware modifications from Section I.1: bottleneck, maxout, and block-wise dynamic 8-bit quantization, each with 2 pipeline stages and each a compression factor of 2x.

The results can be seen in Table 7. We can see that 8-bit compression does not degrade the time to 22 perplexity and maintains close to the final perplexity of the baseline. The compression-aware bottleneck and maxout architectures perform equal to each other, but degrade final perplexity slightly and increase time to a perplexity of 22 by 26–28%.

Using these results, one can determine which method is optimal for their hardware setup. For instance, training with maxout with 2 pipeline stages needs 28% more steps, but

Table 7: Performance of compression methods for a Transformer language model with adaptive inputs on WikiText-103. The asterisk denotes that the difference is not statistically significant.

Method	Ppl after 286K steps	Steps to ppl 22	Data transfer	Extra compute	
				Absolute	Relative
No compression	21.02	1x	1x	0	None
8-bit compression	21.13	0.97x*	0.5x	1.2ms	None (overlapped)
Bottleneck	21.76	1.26x	0.5x	1.96ms	$\leq 1\%$
Maxout	21.83	1.28x	0.5x	2.04ms	$\leq 1\%$

accelerates the communication phase by 2x. If communication is the limiting factor, using maxout or bottleneck compression layers will offer *improved* time to perplexity despite the performance degradation. However, the same two techniques would result in slower training in a setup where network bandwidth is unlimited.

In turn, 8-bit quantization reduces communication cost without slowing down per-iteration convergence, making it a "safe bet" for situations where the per-iteration convergence must be preserved. In our large-scale experiments (Section 4.3), we opt to using quantization since it was enough to fully saturate the GPUs. If network bandwidth is still a limiting factor, one can combine quantization with bottleneck or maxout compression to further reduce communication.

I.3. Additional Experiments

The additional experiments in this section have two purposes: (1) to evaluate how compression methods vary with the number of stages and (2) to evaluate an additional setting that is closer to modern pretraining setups such as GPT-2/3.

While (1) has further implications for scaling, (2) is helpful to account for confounding factors that might have been overlooked in the main experiments on WikiText-103. The WikiText-103 baseline uses non-BPE vocabulary, a long sequence length, and uses adaptive inputs (Baevski & Auli, 2019), all of which are not frequently used in modern pretrained Transformers since GPT-2 (Radford et al., 2019).

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 hyperparameters: sequence size 512, 16 layers with model dimension 1024, and hidden dimension 4096 for a total of 253M parameters. We use byte pair encoding (Sennrich et al., 2016; Radford et al., 2019) with a vocabulary size of 50264 symbols. We do not use dropout or other regularization, since our models underfit. We run these experiments in Fairseq (Ott et al., 2019).

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

Results: The results of our compression-aware architectures are shown in Table 8. 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 suggests compression-aware architectures do not degrade the out-of-distribution performance more than the in-distribution performance. As such, the maxout compression is an effective technique to reduce the bandwidth requirements of pipeline parallel training further.

While the 8-bit blockwise quantization can only compress the activations by a factor of two (16-bit → 8-bit), it does not affect the quality as much when compared to the baseline. As such, the 8-bit quantization appears to be a reliable default choice to reduce the communication overhead for pipeline parallelism.

When considered 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-bandwidth peers. Thus, compression-aware architectures improve the accessibility and affordability of training large models outside HPC environments.

J. Time To Solution

In this section, we evaluate the compression-aware techniques proposed in Appendix I.1 from a practitioner's point of view. A natural way to compare these techniques is in terms of "the time to solution", i.e., the wall-clock time it takes to achieve the desired validation objective. In practice, this time depends on three main factors: the compression strategy, the distributed training algorithm, and the computational infrastructure.

In order to disentangle these factors, we first address the relationship between the training algorithm and the infrastructure. As we discuss in Section 3.2 (and later in Appendix E),

Table 8: Results of language models trained on the OpenWebText Corpus (OWT). The 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. PTB=Penn Treebank, 1BW=Billion word corpus.

			Validation perplexity					
Model	Stages	Compression	OWT	LAMBADA	WikiText-2	WikiText-103	PTB	1BW
Baseline	–	–	19.7	86.4	56.2	35.4	133.0	80.9
8-bit Quantization	2	2x	19.6	89.1	**56.0**	**35.0**	132.7	79.8
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**
8-bit Quantization	4	2x	**19.7**	**87.9**	**56.3**	**35.2**	**133.9**	**79.8**
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**

Table 9: Training time and costs.

Setup	Time, hours	Cost, $	
		Hourly	Total
$8 \times V100$, reliable	175.4	7.834	1374
$8 \times V100$, preemptible	192.6	5.383	1037
$32 \times T4$, preemptible	140.8	3.536	497.8

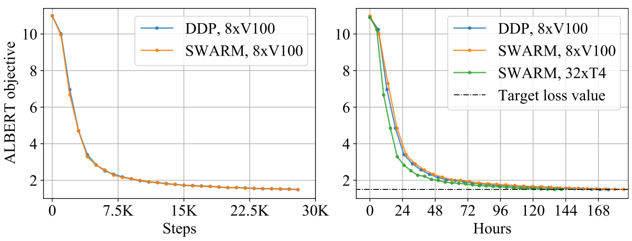


Figure 8: Convergence curves of ALBERT with SWARM and standard data-parallel training.

SWARM parallelism has the same per-iteration behavior as other synchronous methods. Theoretically, the choice of an optimal training system should come down to whichever algorithm has the highest training throughput.

To verify this argument in practice, we compare the per-iteration and per-hour performance of SWARM against fully synchronous training. For this experiment, we train the ALBERT model (Lan et al., 2020) on the WikiText-103 dataset (Merity et al., 2017). We use the ALBERT-Large architecture with 4 layer groups that correspond to 4 SWARM stages *without the architecture modifications from Appendix I*. We follow the exact hyperparameters from the original paper: for example, we use the LAMB optimizer (You et al., 2020) with the batch size of 4096 and the sequence length of 512. We train this model in three setups: traditional distributed training with 8 V100 workers, SWARM with 8 preemptible V100 GPUs, and SWARM with 32 preemptible T4 workers.

To quantify the time to solution, we measure the wall time required to achieve the ALBERT objective equal to **1.5**. Additionally, we report the per-hour cost of each experimental setup and the total cost of achieving a loss of 1.5 using public cloud provider pricing estimates in Table 9.

Figure 8 demonstrates that SWARM matches the per-iteration learning curves of traditional distributed training (PyTorch DistributedDataParallel) up to the variation comparable to caused by changing the random seed. However, SWARM parallelism can achieve the loss of 1.5 more cost-efficiently and faster by using preemptible instances. In turn, *when forced to use homogeneous and reliable GPUs*, SWARM would have slightly inferior performance compared to conventional algorithms, which was first demonstrated in Section 4.2.