OPT2025: 17th Annual Workshop on Optimization for Machine Learning

Partial Parameter Updates for Efficient Distributed Training

Anastasiia Filippova A_FILIPPOVA@APPLE.COM
Angelos Katharopoulos A_KATHAROPOULOS@APPLE.COM
David Grangier GRANGIER@APPLE.COM
Ronan Collobert COLLOBERT@APPLE.COM
Apple

Abstract

We introduce a memory- and compute-efficient method for low-communication distributed training. Existing methods reduce communication by performing multiple local updates between infrequent global synchronizations. We demonstrate that their efficiency can be significantly improved by restricting backpropagation: instead of updating all the parameters, each node updates only a fixed subset while keeping the remainder frozen during local steps. This constraint substantially reduces peak memory usage and training FLOPs, while a full forward pass over all parameters eliminates the need for cross-node activation exchange. Experiments on a 1.3B-parameter language model trained across 32 nodes show that our method matches the perplexity of prior low-communication approaches under identical token and bandwidth budgets while reducing training FLOPs and peak memory.

1. Introduction

Recent research has shown that scaling language models (LLMs) consistently improves generalization and downstream capabilities [14, 25]. At scale, training is typically achieved by distributing data across many compute nodes and synchronizing gradients at every optimization step. This synchronization demands high-bandwidth interconnects, limiting large-scale training to high-end clusters with large number of well-interconnected nodes–a resource still accessible to only a small fraction of the machine learning community. In this paper, we explore strategies for training large language models on less powerful hardware with limited memory and bandwidth. Our approach reduces both memory footprint and total training FLOPs compared to existing low-communication methods, enabling efficient large-scale training in resource-constrained environments. To reduce the reliance on high-bandwidth interconnects, prior work has explored decreasing the volume of data transferred between nodes through gradient sparsification, compression, or quantization [1, 4, 13]. Another line of research, which our work builds upon, lowers communication overhead by reducing the frequency of gradient synchronization. First introduced in the federated learning setting [17], this approach allows each model replica to perform multiple local update before a global optimization step [15, 23, 27]. DiLoCo [6] applies this dual optimization scheme to LLM training, reducing bandwidth requirements by orders of magnitude compared to standard every-step gradient reduction. Streaming DiLoCo [7] extends this idea by synchronizing only a subset of parameters at a time, thereby lowering both peak bandwidth and memory usage. Building on these advances, our method further reduce memory footprint and total training FLOPs, making faster low-communication training feasible on sets of low-memory devices (e.g., RTX 3090 GPUs with 16GB of RAM).

In this work, we introduce a method for efficient, low-memory, low-communication distributed training. Our approach can be viewed as a form of block coordinate optimization: each node backpropagates through and updates only a fixed slice of the parameters, treating the remainder as constant. After several local steps, the parameter differences are all-reduced across nodes, followed by an outer optimizer step, as in prior works. By restricting both backpropagation and optimizer updates to the active slice, our approach reduces peak memory usage and total training FLOPs relative to Streaming DiLoCo, while matching its bandwidth requirements and final performance (measured by test perplexity).

Our contributions are as follows: **(i)** We introduce an efficient algorithm for low-communication distributed data-parallel training that performs local updates on a node-specific subset of parameters, thereby reducing both memory usage and computational cost. **(ii)** We empirically validate the effectiveness of our method by training a 1.3B-parameter language model on 32 nodes, achieving perplexity comparable to prior low-communication training approaches under the same token and bandwidth budgets, while using 15% fewer FLOPs and up to 47% less memory (depending on configuration) (Figure 1a and 1b). **(iii)** We demonstrate that in simulated low-bandwidth settings, our method converges substantially faster than DDP (Figure 1c).

2. Method

In this section, we formalize our proposed method for low-communication training. Section 2.1 provides a brief overview of language modeling and distributed data parallelism in both high-bandwidth and bandwidth-limited settings. Section 2.2 introduces the core idea behind our method, followed by its training procedure and implementation details.

2.1. Background

Language Modeling. Let $\mathcal{D}$ be a dataset of token sequences $\mathbf{x} = (x_1, \ldots, x_S)$ with $x_s \in \mathcal{V}$, where $\mathcal{V}$ is the vocabulary and S is the sequence length. Language modeling is about modeling the data distribution $p(\mathbf{x})$, that can be factorized as $p(\mathbf{x}) = \prod_{s=1}^{S-1} p(x_{s+1} \mid \mathbf{x}_{1:s}; \theta)$, where θ denotes the model parameters. The parameters usually are estimated by solving: $\theta^\star = \arg\min_\theta \mathbb{E}_{\mathbf{x}\sim\mathcal{D}} \mathcal{L}(\mathbf{x}; \theta)$. where the loss is the negative log-likelihood of next-token prediction. In practice, this objective is minimized using stochastic gradient descent, where at each step the gradient $\nabla_\theta \mathcal{L}(\mathbf{X}; \theta)$ is computed on a mini-batch of sequences $\mathbf{X}$.

Distributed Data Parallelism (DDP). To scale the optimization problem (2.1), a common approach is to partition $\mathcal{D}$ across K compute nodes, assigning each node k a shard $\mathcal{D}_k$. At training step t, node k computes the gradient $g_k^{(t)} = \nabla_\theta \mathcal{L}(\mathbf{X}_k^{(t)}; \theta^{(t)})$, on its local mini-batch $\mathbf{X}_k^{(t)} \sim \mathcal{D}_k$. Gradients are aggregated via all-reduce to form $g^{(t)} = \frac{1}{K} \sum_{k=1}^{K} g_k^{(t)}$, which is then used to update the model parameters on every node. Model parameters, optimizer state, and gradients may be either fully replicated or partitioned across nodes to reduce memory usage [30, 34].

Low-communication Distributed Data Parallelism. Because standard DDP aggregates gradients at every step, it is impractical on hardware lacking high-bandwidth, low-latency interconnects. Low-communication distributed training methods relax this by reducing the synchronization frequency: each node k performs H local updates

$$\theta_k^{(t,h+1)} \leftarrow \textsc{InnerOpt}\big(g_k^{(t,h)};\, \theta_k^{(t,h)}\big), \qquad h = 0, \ldots, H-1, \tag{1}$$

without inter-node communication. After H steps, each node computes its parameter delta and performs an all-reduce to obtain the averaged delta

$$\Delta\theta^{(t)} \;=\; \frac{1}{K}\sum_{k=1}^{K}\big(\theta_k^{(t,H)} - \theta_k^{(t)}\big), \tag{2}$$

Global parameters are then updated via an *outer optimizer*:

$$\theta^{(t+1,0)} \;\leftarrow\; \textsc{OuterOpt}\big(\Delta\theta^{(t)},\, \theta^{(t)}\big), \tag{3}$$

where InnerOpt$(\cdot)$ and OuterOpt$(\cdot)$ can be arbitrary optimizers. In practice, the outer step either applies the averaged delta directly [17] or treats it as a pseudo-gradient in SGD [23, 27]. For LLM training, DiLoCo [6] showed that using AdamW as the inner optimizer and Nesterov SGD as the outer optimizer yielded lower validation loss than alternative configurations.

Memory Usage and Computational Cost In low-bandwidth settings, model parameters, optimizer state, and gradients typically cannot be sharded and must therefore be fully replicated on each node. If the outer optimizer maintains momentum, its state must also reside in memory. Synchronizing only a subset of model parameters at a time—while keeping the remainder on the host—can reduce the outer optimizer's memory usage [7], yet the overall footprint remains substantial. For example, even a relatively small 1.3B-parameter model requires about 18 GB of GPU memory solely for model weights, gradients, and optimizer moments (Fig. 1c, Sec. 3.2). Our goal in this paper is to reduce memory usage and computational cost without increasing communication overhead or degrading performance. Making low-communication distributed training more memory-efficient would enable large-scale training across consumer devices without relying on high-speed interconnects.

2.2. Low-communication DDP with Partial Parameter Updates

Our method can be viewed as a distributed variant of block coordinate descent. On each node k, we define a fixed index set $I_k^{\text{train}} \subseteq \{1,\dots,|\theta|\}$, such that parameters $\theta_k[i]$ with $i \in I_k^{\text{train}}$ are updated locally. The complement $I_k^{\text{frozen}} = \{1,\dots,|\theta|\} \setminus I_k^{\text{train}}$ corresponds to parameters that remain unchanged on node k. For a mini-batch $\mathbf{X}_k^{(t,h)}$ at local step h, the forward pass proceeds as usual, $f(\theta_k^{(t,h)}, \mathbf{X}_k^{(t,h)})$. During backpropagation, gradient computation is restricted to I_k^{train} (Algorithm 1, line 8), and only the corresponding parameters are updated (line 9). After H local steps, each node computes a delta for its trainable parameters and zeros elsewhere (line 11). These deltas are summed across nodes and normalized elementwise by a fixed count vector $\mathbf{m} \in \{1,\dots,K\}^{|\theta|}$, where $\mathbf{m}[i]$ records how many nodes update parameter $\theta[i]$ (line 13). The normalized update is then applied in the outer optimizer step (line 14). The full training procedure is summarized in Algorithm 1.

Updating partly disjoint parameter sets on each node is intuitive for limiting divergence across local replicas, since each parameter $\theta[i]$ is optimized by fewer than K nodes during local steps. This design also offers two benefits: (i) reduced per-node memory usage, as no gradient buffers or optimizer state are allocated for parameters in I_k^{frozen} (Figure 1a), and (ii) lower training FLOPs, since gradients for $\theta[i]$ with $i \in I_k^{\text{frozen}}$ are never computed (Figure 1c). In Section 3.3, we demonstrate that despite fewer updates per parameter than full-model baselines, our method achieves comparable test perplexity.

Algorithm 1 Distributed Training with Partial Updates

Inputs: outer rounds T, local steps H, number of nodes K
Notation: I_k^{train}; $I_k^{\text{frozen}} = \{1, \dots, |\theta|\} \setminus I_k^{\text{train}}$, count vector $\mathbf{m} \in \{1, \dots, K\}^{|\theta|}$
for $t = 1 \dots T$ **do**
  $\theta_k^t \leftarrow \theta^{t-1}$
  for $k = 1 \dots K$ **do** # Execute in parallel on K nodes
    for $h = 1 \dots H$ **do** # Perform H local steps independently on each node
      $\mathbf{X}_k^{(t,h)} \leftarrow \mathcal{D}_k$
      $g_k^{(t,h)} \leftarrow \nabla_{\theta[i],\, i \in I_k^{\text{train}}} \mathcal{L}_k(\theta_k^{(t,h)}; \mathbf{X}_k^{(t,h)})$ # Backpropagate only through I_k^{train}
      $\theta_k^{(t,h+1)}[I_k^{\text{train}}] \leftarrow \text{INNEROPT}(\theta_k^{(t,h)}[I_k^{\text{train}}],\, g_k^{(t,h)})$
    end for
    $\Delta_k^{(t)}[i] = \begin{cases} \theta_k^{(t,H)}[i] - \theta^{(t-1)}[i], & i \in I_k^{\text{train}} \\ 0, & \text{otherwise} \end{cases}$
  end for
  $\Delta^{(t)}[i] = \frac{1}{\mathbf{m}[i]} \sum_{k=1}^{K} \Delta_k^{(t)}[i], \quad i = 1, \dots, |\theta|$ # Elementwise normalize by count vector $\mathbf{m}$
  $\theta^{(t)} \leftarrow \text{OUTEROPT}(\theta^{(t-1)}, \Delta^{(t)})$
end for

2.2.1. PARAMETER SLICING

We consider two variants for assigning trainable parameters across nodes, controlled by a slicing hyperparameter N with $K \bmod N = 0$ and slice index $n = k \bmod N$ on node k. This leads to the following update count vector in Eq. (14): $\mathbf{m}[i] = \begin{cases} \frac{K}{N}, & i \in I^{\text{train}} \\ K, & \text{otherwise} \end{cases}$.

MLP-only slicing Embedding, attention, and normalization parameters remain fully trainable on all nodes. Only the MLP layers are sliced, so that each node updates only a portion of the MLP weights while keeping the rest frozen. In each MLP block, let $\mathbf{W} \in \mathbb{R}^{d \times 4d}$ be the up-projection and $\mathbf{V} \in \mathbb{R}^{4d \times d}$ the down-projection, such that: $\text{MLP}(\mathbf{x}) = \mathbf{V}^\top\big(\text{ReLU}(\mathbf{W}^\top \mathbf{x})\big)$. We partition $\mathbf{W}$ column-wise into $\{\mathbf{W}_1, \dots, \mathbf{W}_N\}$, where $\mathbf{W}_n \in \mathbb{R}^{d \times 4d/N}$, and split $\mathbf{V}$ row-wise into $\{\mathbf{V}_1, \dots, \mathbf{V}_N\}$, where $\mathbf{V}_n \in \mathbb{R}^{4d/N \times d}$. This gives: $\text{MLP}(\mathbf{x}) = \sum_{n=1}^{N} \mathbf{V}_n^\top\big(\text{ReLU}(\mathbf{W}_n^\top \mathbf{x})\big)$. Each node k trains all non-MLP parameters, as well as the MLP slice $\{\mathbf{W}_n, \mathbf{V}_n\}$ corresponding to its assigned index $n = k \bmod N$, the remaining MLP slices are frozen.

MLP + attention-heads slicing In addition to MLP-only slicing, we also partition attention heads so that each node trains only a subset of heads while freezing the rest. Let the multi-head attention mechanism [26] use projections: $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times (h \cdot d_h)}$, where h is the number of heads and d_h the per-head dimension (so that $d = h \cdot d_h$). We split the h heads into N disjoint groups of size h/N (assuming $h \bmod N = 0$). For node k, the assigned slice index is $n = k \bmod N$, with head group $\mathcal{H}_n = \{n \cdot (h/N), \dots, (n+1) \cdot (h/N) - 1\}$. The attention projection matrices are then sliced column-wise according to $\mathcal{H}_n : \mathbf{W}_Q^{(n)} = \mathbf{W}_Q[:, \mathcal{H}_n], \mathbf{W}_K^{(n)} = \mathbf{W}_K[:, \mathcal{H}_n], \mathbf{W}_V^{(n)} = \mathbf{W}_V[:, \mathcal{H}_n]$. With this decomposition, the attention block becomes a sum over head-groups, and node k updates only the parameters of its assigned group n, while freezing the rest.

3. Experiments

In this section, we experimentally analyze the performance of our proposed method. We benchmark our approach against Streaming DiLoCo [7] and standard distributed data-parallel (DDP) training.

3.1. Experimental Setup

We use the RedPajama dataset [29]. All experiments use sequences of $1{,}024$ tokens. We use a Transformer [26] with 1.3B parameters, consisting of 24 layers and a hidden dimension of 2048. We employ rotary positional encoding [22] and a SentencePiece tokenizer [11] with a vocabulary size of $32{,}000$. All methods are trained with the AdamW optimizer [16] using $\beta_1 = 0.9$, $\beta_2 = 0.99$, and a weight decay of 0.1. The learning rate schedule linearly warms up to 3×10^{-4} over the first $1{,}500$ steps, followed by cosine decay. For both our method and Streaming DiLoCo, we use SGD with Nesterov momentum $m = 0.9$ [18] as the outer optimizer, with learning rate 4×10^{-1}. We use batch size 512 that is distributed across 32 NVIDIA H100 GPUs with 80GB memory (16 per GPU).

Comparison to Streaming DiLoCo Our algorithm (Section 2.2) is orthogonal to the streaming synchronization pattern proposed by Douillard et al. [7]. To ensure fair comparison we reuse this in our experiments (App. B.2).

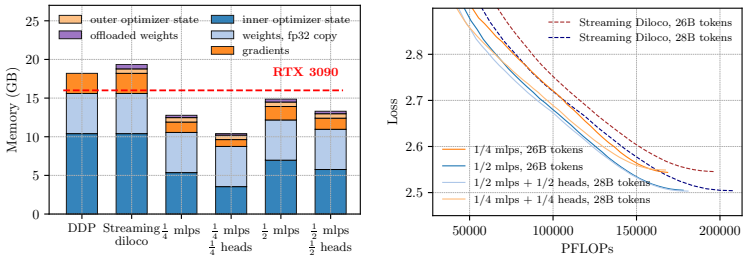
3.2. Memory Usage

We assume training in BF16 with full activation recomputation during the backward pass. In mixed-precision training, it is standard to maintain a master copy of model parameters in single precision (FP32) for updates, while casting parameters on the fly to BF16 for computation. Gradients are stored in BF16, while the optimizer state remains in FP32. In low-bandwidth training with an outer optimizer, one must also account for offloaded weights and momentum buffers. When synchronization is performed in a streaming manner (i.e., grouped communication as in [7]), the additional memory overhead from this state is relatively small (see Figure 1a). Thus, with activation recomputation, peak memory usage is primarily determined by: (i) optimizer state, (ii) weights, (iii) gradients, (iv) outer optimizer state (if used), and (v) offloaded parameters (if any), as illustrated in Figure 1a.

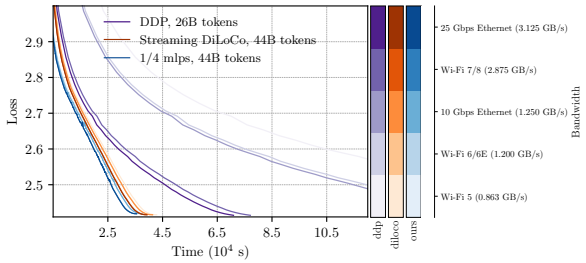
3.3. Results

Peak Memory Footprint Figure 1a demonstrates that our method requires significantly less memory than Streaming DiLoCo and DDP. This reduction comes from the fact that we do not train a large portion of parameters (detailed in Table **??**), which means we neither maintain optimizer state nor store gradients for these parameters. For instance, $1/4$ mlps + $1/4$ heads configuration of our method uses 47% less memory compared to full model training, while achieving similar test loss. This allows us to fit training with activation checkpointing of a 1.3B model using devices with less than 16GB of RAM.

Compute Efficiency We compare our method to Streaming DiLoCo in terms of training FLOPs. Figure 1c shows test loss as a function of total training FLOPs. For this comparison, we trained the $1/4$-MLP, $1/2$-MLP, and Streaming DiLoCo configurations with the Chinchilla-optimal token budget (26B). To match the performance of the $1/2$-MLP configuration, we slightly increased the token budget for Streaming DiLoCo to 28B. We also trained the $1/N$-MLP+$1/N$-heads configurations



(a) Memory usage, GB (b) Training FLOPs vs test loss



(c) Time vs test loss

Figure 1: **Less memory, fewer FLOPs, same performance.** Comparison of memory usage, total training FLOPs, and training time between our approach, Streaming DiLoCo, and DDP for 1.3B language model training across 32 compute nodes. In each Transformer layer we either slice only the MLPs ($\frac{1}{N}$ MLPs) or slice both MLPs and attention heads ($\frac{1}{N}$ MLPs, $\frac{1}{N}$ heads). In both cases, only $1/N$ of the parameters in the corresponding projections are trained on each node (see Sec.2.2.1). **(a)** Estimated memory usage for DDP, Streaming DiLoCo, and our four variants (Sec.3.2). **(b)** Total training FLOPs versus test perplexity for our method compared to Streaming DiLoCo (Sec.3.3). **(c)** Training time comparison between our method, Streaming DiLoCo, and standard DDP (synchronizing every step) under different bandwidth constraints. (Sec. 3.3).

on 28B tokens to match the performance of their corresponding $1/N$-MLP runs. Across these performance-matched comparisons, our method consistently required 15% fewer total FLOPs.

Convergence Speed Under Bandwidth Constraints We compare our approach against Streaming DiLoCo and standard DDP in terms of total training time (Figure 1c). To simulate training under different bandwidth constraints, we first measure the time for a single training step on one GPU, and then add the estimated communication time for ring all-reduce [19] (App. B.1). It is important to note that, due to infrequent synchronization, low-bandwidth methods require more tokens (and thus more computation) to reach the same performance as DDP. In Figure 1c, both Streaming DiLoCo and our method were trained on twice the number of tokens as DDP (39B vs. 26B). However, even under the assumption that peak bandwidth is fully reachable (in practice, effective bandwidth is always lower), training a 1.3B model on 32 nodes connected via a Thunderbolt 3 ring with DDP requires more than twice the training time to match the performance of low-communication approaches. Finally, it is worth noting that prior research only reported results for up to 8 nodes, which explains why this degradation in performance was not observed in Douillard et al. [6].

4. Conclusion

We introduced a simple yet effective strategy for reducing memory and compute costs in low-communication distributed training by updating only a subset of parameters on each node. Our method achieves substantial memory and FLOPs reductions without increasing communication or degrading performance. These findings open up multiple directions for future research, including large-scale training in bandwidth- and memory-constrained environments.

References

[1] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. *Advances in neural information processing systems*, 2017.

[2] Matt Beton, Seth Howes, Alex Cheema, and Mohamed Baioumy. Improving the efficiency of distributed training using sparse parameter averaging. In *ICLR 2025 Workshop on Modularity for Collaborative, Decentralized, and Continual Deep Learning*, 2025.

[3] Alon Cohen, Amit Daniely, Yoel Drori, Tomer Koren, and Mariano Schain. Asynchronous stochastic optimization robust to arbitrary delays. *Advances in Neural Information Processing Systems*, 2021.

[4] Tim Dettmers. 8-bit approximations for parallelism in deep learning. *arXiv preprint arXiv:1511.04561*, 2015.

[5] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. *arXiv preprint arXiv:2110.02861*, 2021.

[6] Arthur Douillard, Qixuan Feng, Andrei A Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc'Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. Diloco: Distributed low-communication training of language models. *arXiv preprint arXiv:2311.08105*, 2023.

[7] Arthur Douillard, Yanislav Donchev, Keith Rush, Satyen Kale, Zachary Charles, Zachary Garrett, Gabriel Teston, Dave Lacey, Ross McIlroy, Jiajun Shen, et al. Streaming diloco with overlapping communication: Towards a distributed free lunch. *arXiv preprint arXiv:2501.18512*, 2025.

[8] Yizhou Han, Chaohao Yang, Congliang Chen, Xingjian Wang, and Ruoyu Sun. Q-adam-mini: Memory-efficient 8-bit quantized optimizer for large language model training. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*, 2025.

[9] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[10] Satyen Kale, Arthur Douillard, and Yanislav Donchev. Eager updates for overlapped communication and computation in diloco. *arXiv preprint arXiv:2502.12996*, 2025.

[11] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2018.

[12] Bingrui Li, Jianfei Chen, and Jun Zhu. Memory efficient optimizers with 4-bit states. *Advances in Neural Information Processing Systems*, 36, 2023.

[13] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *International Conference on Learning Representations*, 2018.

[14] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[15] Bo Liu, Rachita Chhaparia, Arthur Douillard, Satyen Kale, Andrei A Rusu, Jiajun Shen, Arthur Szlam, and Marc'Aurelio Ranzato. Asynchronous local-sgd training for language modeling. *arXiv preprint arXiv:2401.09135*, 2024.

[16] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2017.

[17] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*. PMLR, 2017.

[18] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*. Springer Science & Business Media, 2013.

[19] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 2009.

[20] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. Swarm parallelism: Training large models can be surprisingly communication-efficient. In *International Conference on Machine Learning*. PMLR, 2023.

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

[22] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 2024.

[23] Tao Sun, Dongsheng Li, and Bao Wang. Decentralized federated averaging. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[24] Weigao Sun, Zhen Qin, Weixuan Sun, Shidi Li, Dong Li, Xuyang Shen, Yu Qiao, and Yiran Zhong. Co2: Efficient distributed training with full communication-computation overlap. *arXiv preprint arXiv:2401.16265*, 2024.

[25] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[26] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[27] Jianyu Wang, Vinayak Tantia, Nicolas Ballas, and Michael Rabbat. Slowmo: Improving communication-efficient distributed sgd with slow momentum. *arXiv preprint arXiv:1910.00643*, 2019.

[28] Jue Wang, Binhang Yuan, Luka Rimanic, Yongjun He, Tri Dao, Beidi Chen, Christopher Re, and Ce Zhang. Fine-tuning language models over slow networks using activation compression with guarantees. *arXiv preprint arXiv:2206.01299*, 2022.

[29] Maurice Weber, Dan Fu, Quentin Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, et al. Redpajama: an open dataset for training large language models. *Advances in neural information processing systems*, 2024.

[30] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. Automatic cross-replica sharding of weight update in data-parallel training. *arXiv preprint arXiv:2004.13336*, 2020.

[31] Binhang Yuan, Yongjun He, Jared Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy S Liang, Christopher Re, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments. *Advances in Neural Information Processing Systems*, 2022.

[32] Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, Diederik P Kingma, Yinyu Ye, Zhi-Quan Luo, and Ruoyu Sun. Adam-mini: Use fewer learning rates to gain more. *arXiv preprint arXiv:2406.16793*, 2024.

[33] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

[34] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

Appendix A. Related Work

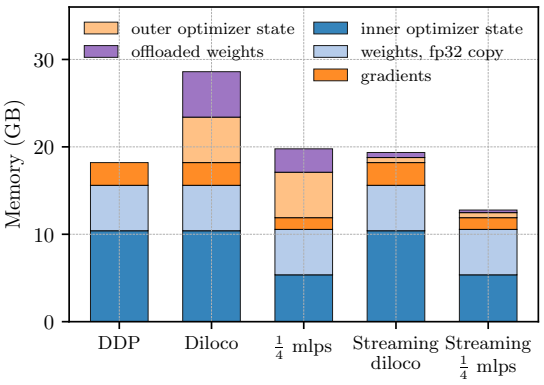
We review prior work in two areas most relevant to our contributions: methods for low-communication distributed training and approaches for improving memory and computational efficiency during training. For the latter, we focus on memory-efficient optimizers and tensor parallelism, which are most directly related to our method.

Low-Communication Training Communication overhead in distributed data-parallel training has been tackled in three main ways: reducing the volume of data exchanged between nodes with gradient compression or quantization [1, 4, 12, 13], hiding latency by overlapping communication with computation [3, 10, 24], and lowering frequency of communication by performing multiple local updates between synchronizations [6, 7, 17, 23, 27]. We show that the latter can be made substantially more memory- and compute-efficient by restricting backpropagation to partial parameter subsets. The three strategies are complementary, and compression or overlap techniques can be applied together with our method to further reduce communication costs. More recently, Beton et al. [2] proposed sparse parameter synchronization, which reduces communication by synchronizing only a random fraction of parameters at each step. While this lowers divergence across nodes, all parameters are still updated on every device, meaning each node must store the full optimizer state and perform full backpropagation. In contrast, our method updates only a fixed subset of parameters per node, which directly reduces both memory and compute.

Another line of work studies pipeline parallelism in slow-network settings [9], which requires inter-stage communication of activations in every step. To mitigate this communication overhead, recent methods propose compressing or quantizing activations [20, 28, 31]. Unlike these approaches, which still depend on activation exchange, our method operates purely in the data-parallel regime but can be combined with pipeline parallelism and activation compression in large-scale settings.

Memory and Compute Efficiency A large fraction of GPU memory during training is occupied by optimizer states, particularly for adaptive methods such as Adam [16], which maintain first- and second-order moments for every parameter. The main savings of our approach come from the fact that each node only updates a subset of parameters. As a result, momentum states for the remaining parameters do not need to be stored locally, yielding substantial memory savings. This is especially important in low-communication settings, where sharding optimizer states across devices is impractical due to the communication overhead it introduces. Several methods aim to reduce optimizer state memory directly. One strategy is to quantize optimizer states to lower precision, for example 8-bit quantization [5, 12]. Another is to apply low-rank projections to compress gradients and optimizer states [33]. Parameter grouping has also been explored: Zhang et al. [32] maintain a single momentum vector per block of parameters, while Han et al. [8] combine grouping with quantization. Such efficient optimizers are orthogonal to our method and could be combined with it for further savings.

Another line of work distributes compute and memory through tensor parallelism, where large matrix multiplications are partitioned across GPUs and results are gathered after each operation [21]. Our method is conceptually related, but applies slicing only in the backward pass: each device updates a portion of the parameter matrix while still executing the full forward computation. In contrast to tensor parallelism, our approach avoids frequent all-to-all communication and therefore does not depend on high-bandwidth interconnects.



(a) Memory usage, GB

Method	Test perplexity
DiLoCo	12.78
Streaming DiLoCo	12.74
Ours ($\frac{1}{4}$ mlps)	12.73
Ours (streaming $\frac{1}{4}$ mlps)	12.72

(b) Test perplexity

Figure 2: **Streaming synchronization.** Comparison of memory usage and test perplexity with and without streaming synchronization for DiLoCo and our approach in training a 1.3B parameter language model across 32 compute nodes. In all experiments, the 24 layers were grouped sequentially into groups of 3, yielding 8 layer groups plus a ninth group for the embedding and outer normalization layers. **(a)** Estimated memory usage per GPU (Sec. 3.2). **(b)** Final test perplexity after training on 26B tokens.

Appendix B. Experiments

B.1. Communication Overhead

We compare our method with Streaming DiLoCo and standard DDP by simulating total training time under bandwidth-constrained conditions. While low-communication methods require more training tokens to achieve the same performance as Distributed Data Parallel (DDP), they are significantly faster in terms of wall-clock time on slow networks.

Our simulation model deliberately favors DDP by assuming perfect overlap between computation and communication, giving a per-step runtime of $T = \max\big(T_{\text{comm}}, T_{\text{comp}}\big)$. In contrast, for low-communication methods we assume no overlap: $T = T_{\text{comm}} + T_{\text{comp}}$.

$$T_{\text{comm}} \approx \frac{2M(K-1)}{K \cdot B}.$$

This estimate assumes that peak bandwidth is fully utilized which may not hold in real-world environments.

For a 1.3B parameter model trained in `bf16`, the total gradient size is approximately:

$$M = 1.3 \cdot 10^9 \cdot 2 \text{ bytes} = 2.6 \text{ GB}.$$

Assuming $K = 32$ training nodes connected via Wi-Fi 8 (with a peak bandwidth of $B = 23$ Gbps $= 2.875$ GB/s), the estimated communication time per step becomes:

$$T_{\text{comm}} \approx \frac{2 \cdot 2.6 \text{ GB} \cdot 31}{32 \cdot 2.875 \text{ GB/s}} \approx \frac{161.2}{92} \approx 1.75 \text{ seconds}.$$

In comparison, the measured step time on an H100 GPU with batch size 16 is approximately 0.44 seconds, showing that every step gradient synchronization would dominate the runtime even under relatively high wireless bandwidth (e.g., Wi-Fi 8).

In contrast, our method and Streaming DiLoCo synchronize gradients infrequently—once every $S = 100$ steps in our experiments. This reduces the average communication overhead to:

$$T_{\text{comm-low}} = \frac{T_{\text{comm}}}{S} \approx \frac{1.75}{100} = 0.0175 \text{ seconds}.$$

Douillard et al. [7] propose overlapping communication and computation via gradient delaying to further reduce communication overhead. However, even without such optimizations, we show in Figure 1c that our method and Streaming DiLoCo achieve significantly faster training under limited bandwidth compared to every step synchronization. For example, under a Wi-Fi 8 connection, DDP training takes roughly three times longer than low-communication approaches—even under idealized assumptions of peak bandwidth utilization. While techniques like gradient delaying can further reduce communication cost, they are orthogonal to the main contributions of this work.

B.2. Memory overhead

One way to reduce peak memory usage in low-communication distributed training is to lower the memory consumed by the outer optimizer state and offloaded parameters. When parameters are synchronized in groups with multiple local steps in between, it becomes unnecessary to keep the full optimizer state in memory at every step. Instead, it is sufficient to load the state and parameters corresponding to the active group. In Douillard et al. [7], parameters are grouped by transformer layers, either sequentially or using a strided pattern. We adopt this streaming synchronization strategy in all our experiments to ensure a fair comparison. However, our method is orthogonal to this idea: our primary goal is to reduce the memory footprint of the inner optimizer state and gradients. Streaming synchronization can be applied in addition to our method to further reduce memory usage (Figure 2a). As observed in Douillard et al. [7], synchronization in groups does not lead to any noticeable performance difference, either for our approach or for DiLoCo (Table 2b).