École Polytechnique Fédérale de Lausanne

Optional Master Research Project, *January 2025*

DiLoCo-SWARM

Mika Senghaas
mika.senghaas@epfl.ch

Supervisors
Martijn De Vos, Akash Dhasade, Rishi Sharma

Professor
Anne-Marie Kermarrec

Laboratory
Scalable Computing Systems (SaCS)

DiLoCo-SWARM

Mika Senghaas [1] **Martijn De Vos** [2] **Akash Dhasad** [2] **Rishi Sharma** [2]

Abstract

This work proposes *DiLoCo-SWARM*, a decentralized training method that integrates infrequent gradient synchronization from DiLoCo into SWARM's fault-tolerant data-pipeline parallelism. Our experiments show that DiLoCo-SWARM matches the generalization performance of traditional SWARM while reducing synchronization frequency by up to 50x. We analytically show that DiLoCo-SWARM scales efficiently to larger models, with communication savings increasing as model size grows.

1. Introduction

Modern foundation models contain billions of parameters and are trained on trillions of tokens (Chowdhery et al., 2022; Brown et al., 2020; Dubey et al., 2024; Gemini, 2024). Achieving this scale necessitates orchestrating thousands of GPUs to distribute computation (Dubey et al., 2024; DeepSeek-AI, 2024). However, conventional parallelization techniques rely heavily on high-bandwidth interconnects to avoid communication bottlenecks. As a result, cutting-edge models are primarily trained in high-performance clusters (HPCs). Operating these clusters incurs significant costs, thereby restricting research abilities on foundation models to a select group of corporate and state actors (Jaghouar et al., 2024a).

Decentralized training has emerged as a promising alternative. By leveraging low-cost, globally distributed compute resources, decentralized training promises to democratize access to model development. However, this paradigm shift introduces substantial technical challenges, particularly in accommodating latency and bandwidth limitations, heterogeneous compute environments and node failures.

While decentralized training efforts cannot yet match the scale and efficiency of centralized training, the field has made significant progress in recent years. INTELLECT-1 (Jaghouar et al., 2024a), a 10B parameter model trained collaboratively across the globe, demonstrates this most recently. The model builds on DiLoCo (Douillard et al., 2023), a data parallel method that reduces communication overhead by synchronizing gradients infrequently through a dual optimization scheme. DiLoCo achieves competitive performance with traditional data parallelization but only synchronizes gradients every 500 steps instead of after each step. However, DiLoCo requires each node to store a full model replica, limiting participation to high-performance hardware when training large models.

SWARM (Ryabinin et al., 2023) offers a promising, though less explored, alternative for larger-scale decentralized training. SWARM combines both data and pipeline parallelism and introduces several novel techniques to enable fault-tolerant training in heterogeneous compute and network conditions. By partitioning the model across nodes, SWARM can train larger models and allow devices with limited resources to participate. However, in its original formulation, SWARM requires frequent gradient synchronization within pipeline stages.

This work proposes *DiLoCo-SWARM*, a novel approach that integrates DiLoCo-style infrequent gradient synchronization into SWARM parallelism. Our key contributions are:

1. **Convergence.** We demonstrate that SWARM parallelism can effectively integrate DiLoCo and achieve comparable validation perplexity to a synchronous SWARM baseline, while requiring 50x fewer gradient synchronization.

2. **Communication Efficiency.** We show that DiLoCo-SWARM achieves increasing communication savings over SWARM as the model size grows.

3. **Robustness.** We validate the robustness of DiLoCo-SWARM to synchronization frequency, and model sizes, underlining its robustness at scale.

We open-source the experiment code and results on GitHub and Weights & Biases, along with a distilled version of the training logic in a GitHub Gist with only a few hundred lines of PyTorch code. We aim for these resources to foster further open-source research on scaling DiLoCo-SWARM.

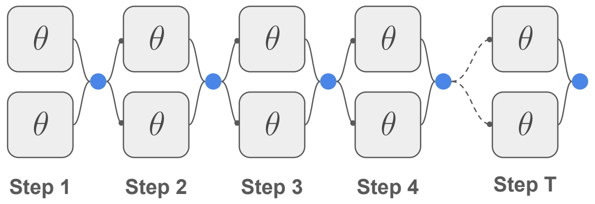
[1]Author [2]Supervisor. Correspondence to: Mika Senghaas <mika.senghaas@epfl.ch>.

2. Background

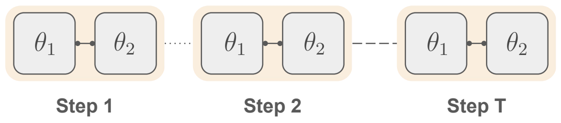
This section outlines the foundations of distributed and decentralized learning, detailing core parallelization techniques, key challenges, and recent solutions, with a focus on the DiLoCo and SWARM methods that underpin our proposed approach.

2.1. Distributed Training

In distributed training, n nodes collaboratively train a *model* parameterized by weights θ, on a *dataset* of samples $D = \{(\mathbf{x}_1, \mathbf{y}_1), \dots\}$. This collaboration requires periodic communication of intermediate results between nodes.



(a) **DP Communication**. Two nodes in a DP group hold the full model θ and a local data shard D_i. In each step, they synchronize local gradients via `AllReduce` (●).



(b) **PP Communication**. Two nodes in a PP group (▨) hold a unique model shard θ_j and communicate activations and gradients (point-to-point) to update their respective model shard.

Figure 1: **DP and PP.** Illustration of communication patterns for $n = 2$ DP and PP groups, unrolled over T training steps. Nodes are represented as boxes, and communication as solid lines. The dotted lines indicate no communication.

In *Data Parallelism* (DP) (Dean et al., 2012), the dataset is partitioned into n shards, with each node holding a unique shard D_i and a complete model replica θ. Training proceeds as follows: Each node samples a batch from its local data shard, and computes local gradients through a forward and backward pass. Before updating the model, nodes synchronize gradients to ensure identical updates across all nodes (Algorithm 1). Typically, the synchronization is performed using an `AllReduce` operation. Figure 1a illustrates this communication pattern unrolled over T training steps.

In *Pipeline Parallelism* (PP) (Huang et al., 2019), the model is partitioned into n stages, with each stage holding a unique subset of parameters θ_i. The full model is then represented as a pipeline, where each stage processes a portion of the input sequentially. Communication occurs in two phases during training: activations are sent forward through the pipeline during the forward pass, and gradients are sent backward during the backward pass (Figure 1b). Importantly, PP relies solely on point-to-point communication between adjacent stages without all-to-all synchronization.

Algorithm 1 Data Parallel Gradient Synchronization

```
1: Input: Local Dataset $D_i$, Model $\theta^{(t-1)}$, Optimizer OPT, Loss $\mathcal{L}$
2: Sample batch: $x_i \sim D_i$
3: Compute gradient: $g_i \leftarrow \nabla_{\theta^{(t-1)}} \mathcal{L}(x_i; \theta^{(t-1)})$
4: Sync gradients: $g \leftarrow \frac{1}{n} \sum_i^n g_i$        ▷ AllReduce
5: Update model: $\theta^{(t)} \leftarrow$ OPT$(\theta^{(t-1)}, g)$
```

Despite its simplicity, PP is challenging to optimize in practice due to synchronization delays that cause GPU idle time. For example, the first stage must wait for the full forward and backward pass through the entire pipeline before it can process the next input. Efficient PP implementations, therefore, rely on techniques such as micro-batching and advanced scheduling to minimize idle time and maximize throughput (Harlap et al., 2018; Huang et al., 2019). Further, for large models, the computation-to-communication ratio grows in favor of PP communication, allowing to "hide" communication latency behind computation (Fernandez et al., 2024).

Both parallelization techniques may be combined. We will refer to this form of hybrid parallelism as *Data-Pipeline Parallelism (DPP)*. In this approach, both the dataset and model are partitioned into $\{D_i\}_{i\in[m]}$ and $\{\theta_j\}_{j\in[n]}$, resulting in a two-dimensional grid of $m \times n$ nodes, where the (i, j)-th node is responsible for data shard D_i and model shard θ_j. Figure 2 illustrates the communication pattern for a 2×2 DPP system. PP and DP communication is interleaved: First, nodes within each PP group communicate activations and gradients to accumulate local gradients independently. Then, nodes synchronize local gradients within each DP group before updating their model shard.

All described parallelization techniques produce equivalent gradient updates and converge to identical model parameters $\theta^{(t)}$. However, they differ fundamentally in how computation and communication are distributed, which impacts efficiency depending on model size, hardware, and network configuration (Hagemann et al., 2024; Fernandez et al., 2024).

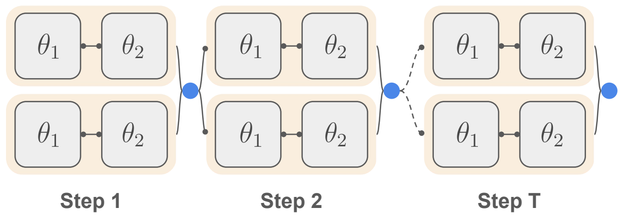


Figure 2: **DPP.** Illustration of communication patterns in a hybrid parallelism (DPP) setup with a 2×2 grid of nodes, unrolled over T training steps. PP and DP communication are interleaved: PP groups accumulate local gradients independently (🟧). Then, DP groups synchronize gradients before updating the model (●).

2.2. Decentralized Learning

Decentralized learning builds on foundational work in federated learning (FL) (McMahan et al., 2016), which enables learning on a network of mobile devices that retain private access to their own data and periodically share local model updates with a central server for aggregation (McMahan et al., 2016; Wang et al., 2020). More recently, decentralized learning has shifted toward large-scale training on preemptible, low-cost hardware, often referred to as *volunteer computing*. While many FL concepts remain relevant, modern research in decentralized learning focuses on improving training efficiency under the unique constraints of decentralized environments: orders-of-magnitude slower interconnect, network and device heterogeneity, and instances joining and leaving training at any time.

Recent research addresses these challenges: Zhang et al. first explored fault-tolerant training on pre-emptible instances, which SWARM extended to support hybrid parallelism in heterogeneous environments (Ryabinin et al., 2023). Another key area of research concerns reducing the communication volume of decentralized training to combat the slow interconnect. DiLoCo (Douillard et al., 2023) reduces the gradient synchronization frequency via a dual optimization scheme, while DeMo (Peng et al., 2024) synchronizes only fast-moving momentum components. SPARTA (EXO, 2025) proposes to only communicate a small random subset of gradients at each step.

Efforts to scale decentralized training to production systems are ongoing. Notable examples include collaborative training platforms such as Transformers Together (Diskin et al., 2021) and Petals (Borzunov et al., 2023), which enable distributed inference for large language models. The recent INTELLECT-1 run (Jaghouar et al., 2024a) demonstrated the feasibility of decentralized training at scale by collaboratively training a 10-billion-parameter model across the globe.

2.3. DiLoCo

In decentralized training, synchronizing gradients at every step, as in traditional DP (Algorithm 1), can be prohibitive due to slow interconnects. DiLoCo (Douillard et al., 2023) addresses this limitation by reducing synchronization frequency through a local-global optimization scheme, shown in Algorithm 2. Each node trains locally for a fixed number of steps, H, using a local optimizer. After H steps, nodes compute their pseudo-gradient (the difference between the initial and updated parameters) and synchronize it using a global optimizer to update a shared model.

Algorithm 2 DiLoCo Gradient Synchronization

```
**Input:** Local dataset $D_i$, Model $\theta^{(t-1)}$, Optimizers $\texttt{OPT}_L$, $\texttt{OPT}_G$, Loss $\mathcal{L}$, Num. Local Steps $H$
Copy global model: $\theta_i^{(t-1)} \leftarrow \theta^{(t-1)}$
**for** $H$ steps **do**
    Sample batch: $x \sim D_i$
    Compute gradient: $g_i \leftarrow \nabla_{\theta_i}\mathcal{L}(x_i; \theta_i^{(t-1)})$
    Update local model: $\theta_i^{(t-1)} \leftarrow \texttt{OPT}_L(\theta_i^{(t-1)}, g_i)$
Compute pseudo-gradient: $\Delta_i \leftarrow \theta_i^{(t-1)} - \theta^{(t-1)}$
Sync pseudo-gradients: $\Delta \leftarrow \frac{1}{n}\sum_{i=1}^{n}\Delta_i$ ▷ AllReduce
Update global model: $\theta^{(t)} \leftarrow \texttt{OPT}_G(\theta^{(t-1)}, \Delta)$
```

Figure 3 illustrates the communication pattern of DiLoCo in contrast to DP (Figure 1a). Nodes only synchronize pseudo-gradients every H steps. Empirical results show that DiLoCo with $H = 500$ matches the generalization performance of a synchronous DP baseline while reducing communication by a factor of 500 (Douillard et al., 2023; Jaghouar et al., 2024b). However, like DP, DiLoCo is limited by the memory capacity of individual nodes, making it unsuitable for models that exceed local memory. Scaling DiLoCo requires techniques such as gradient quantization (Jaghouar et al., 2024a) or parameter offloading (Cui et al., 2016).

2.4. SWARM

While DiLoCo reduces communication frequency, it does not address memory limitations. Sharding the model across nodes, as in pipeline parallelism (PP), is a natural solution. However, traditional PP is poorly suited to decentralized environments due to its sequential nature, which causes significant idle time when nodes have heterogeneous performance. Additionally, a single node failure can stall the entire pipeline.

SWARM parallelism (Ryabinin et al., 2023) addresses these limitations by introducing a fault-tolerant DPP approach. Rather than relying on a fixed grid of nodes,

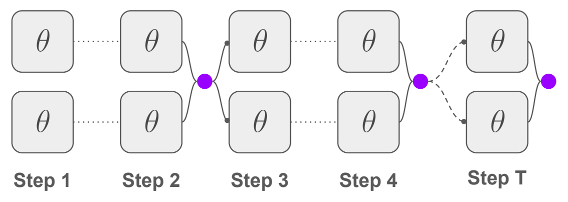


Figure 3: **DiLoCo.** Comparison of communication patterns for a $n = 2$ DiLoCo group, unrolled over T training steps. Unlike traditional DP, DiLoCo synchronizes pseudo-gradients (●) every $H = 2$ steps, reducing total communication by a factor of $1/2$.

SWARM constructs stochastic pipelines dynamically. During the forward pass, each node forwards activations to a randomly selected peer in the next stage, with probability proportional to the peer's throughput. The backward pass follows the same path in reverse to ensure gradient consistency. Once all micro-batches are processed, nodes within each stage form DP groups and synchronize local gradients before updating their respective model shard.

This *stochastic wiring* approach has two key advantages: (1) it naturally balances workloads in heterogeneous environments by routing activations to faster nodes, and (2) it enables the pipeline to recover from node failures by re-routing activations or gradients to available nodes. As long as each stage has at least one active node, SWARM remains functional.

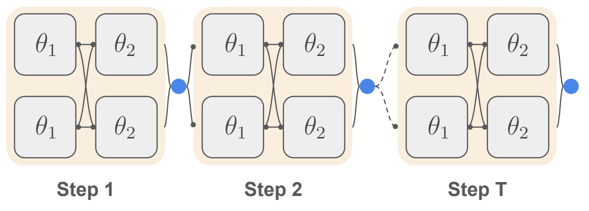


Figure 4: **SWARM.** Illustration of communication in a 2×2 SWARM. As in DPP, PP (▬) and DP (●) communication is interleaved. Crucially, any node may communicate with any other node in adjacent stages, increasing fault-tolerance and throughput in heterogeneous environments.

3. DiLoCo-SWARM

SWARM enables large-scale, fault-tolerant data-pipeline parallel (DPP) training in decentralized settings by dynamically constructing pipelines. However, its data-parallel component still requires frequent gradient synchronization, resulting in high communication costs. In contrast, DiLoCo (Douillard et al., 2023) reduces synchronization frequency through local-global optimization but is limited by memory constraints in large models.

This motivates *DiLoCo-SWARM*, a decentralized training method that combines the communication efficiency of DiLoCo with the scalability and fault-tolerance of SWARM. By replacing SWARM's data-parallel groups with DiLoCo groups, DiLoCo-SWARM significantly reduces the frequency of costly all-to-all gradient synchronization within SWARM stages (Figure 5).

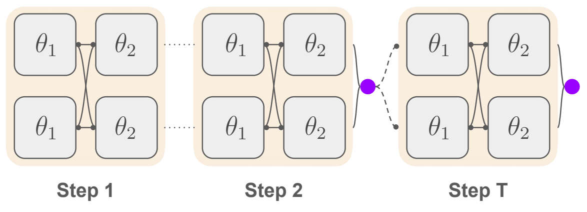


Figure 5: **DiLoCo-SWARM.** The communication patterns of DiLoCo-SWARM for 4 nodes in a 2×2 grid. DiLoCo-SWARM replaces SWARM's frequent gradient synchronization with less frequent DiLoCo-style synchronization of pseudo-gradients (●).

3.1. Algorithm

Algorithm 3 describes the algorithmic details of DiLoCo-SWARM. For simplicity, we assume a grid of $m \times n$ nodes, though the method generalizes to any number of nodes per stage.

Each node begins with a copy of the global model shard $\theta_j^{(t-1)}$. During the inner optimization loop, nodes perform H local steps, stochastically forwarding activations and gradients to adjacent stages and updating their local model shard $\theta_{i,j}$ using a local optimizer. After H steps, nodes compute pseudo-gradients $\Delta_{i,j}$ and synchronize within their stage to update the global model shard $\theta_j^{(t)}$ using a global optimizer.

DiLoCo-SWARM combines SWARM's stochastic wiring during local steps with DiLoCo to synchronize within SWARM stages. In doing so, the method reduces the points of gradient synchronization, resulting in more communication-efficient training. In DiLoCo, gradient synchronization is the only form of communication, so reducing its frequency by a factor of H results in a proportional reduction in communication cost. In DiLoCo-SWARM, gradient synchronization is only a fraction of the total communication cost, which comprises of both gradient synchronization and pipeline communication. Therefore, the communication cost reduction depends on the fraction of the total communication cost incurred by gradient synchronization. Furthermore, due to stochastic wiring, gradients may depend on batches sampled from different data shards, resulting in data mixing across nodes.

Algorithm 3 DiLoCo-SWARM

```
1: Input: Data shard $D_i$, Model shard $f^{(j)}$, Global model shard parameters $\theta_j^{(t-1)}$, Loss $\mathcal{L}$, Optimizers $\texttt{OPT}_L$ and $\texttt{OPT}_G$, Local Steps $H$
2: procedure FwdBwd
3:     if $j = 1$ then                                        ▷ First stage
4:         Sample batch: $x_i \sim D_i$
5:         Forward: $a_{ij} \leftarrow f^{(j)}_{\theta_{ij}}(x_i)$
6:         Send forward: $\texttt{send}(a_{ij}, j+1)$
7:         Receive backward: $g_{i,j+1} \leftarrow \texttt{recv}(j+1)$
8:         Compute gradient: $g_{ij} \leftarrow g_{i,j+1} \cdot \nabla_{\theta_{ij}} f^{(j)}_{\theta_{ij}}(a_{ij})$
9:     else if $j = n$ then                                   ▷ Last stage
10:        Receive forward: $a_{i,j-1} \leftarrow \texttt{recv}(j-1)$
11:        Forward: $a_{ij} \leftarrow f^{(j)}_{\theta_{ij}}(a_{i,j-1})$
12:        Compute gradient: $g_{ij} \leftarrow \nabla_{\theta_{ij}} \mathcal{L}(a_{ij})$
13:        Send backward: $\texttt{send}(g_{ij}, j-1)$
14:    else                                                   ▷ Intermediate stages
15:        Receive forward: $a_{ij-1} \leftarrow \texttt{recv}(j-1)$
16:        Forward: $a_{ij} \leftarrow f^{(j)}_{\theta_{ij}}(a_{i,j-1})$
17:        Send forward: $\texttt{send}(a_{ij}, j+1)$
18:        Receive backward: $g_{i,j+1} \leftarrow \texttt{recv}(j+1)$
19:        Compute gradient: $g_{ij} \leftarrow g_{i,j+1} \cdot \nabla_{\theta_{ij}} f^{(j)}_{\theta_{ij}}(a_{ij})$
20:        Send backward: $\texttt{send}(g_{ij}, j-1)$
    return $g_{ij}$
21: Copy global model: $\theta_{ij}^{(t-1)} \leftarrow \theta_j^{(t-1)}$
22: for $H$ steps do
23:     Compute gradients: $g_{i,j} \leftarrow \texttt{FwdBwd}()$
24:     Update local model: $\theta_{ij}^{(t-1)} \leftarrow \texttt{OPT}_L(\theta_{ij}^{(t-1)}, g_{i,j})$
25: Compute pseudo-gradient: $\Delta_{i,j} \leftarrow \theta_{ij}^{(t-1)} - \theta_j^{(t-1)}$
26: Sync pseudo-gradients: $\Delta_j \leftarrow \frac{1}{m}\sum_i^m \Delta_{i,j}$
27: Update global model: $\theta_j^{(t)} \leftarrow \texttt{OPT}_G(\theta_j^{(t-1)}, \Delta_j)$
```

Note: The `recv` and `send` functions perform many-to-one and one-to-many communication with adjacent stages using stochastic wiring. Details are abstracted away for brevity.

3.2. Implementation

We release a complete implementation of the proposed method, along with a distilled, single-file training script that supports DP, PP, DPP, SWARM, and DiLoCo-SWARM. The implementation is designed for simplicity, relying solely on PyTorch's `torch.distributed` package for communication.

The training script is minimal, consisting of only a few hundred lines of code, making it an accessible resource for researchers to explore and experiment with various distributed training techniques. It supports emulation of multiple nodes via threading and integrates with HuggingFace for model and dataset loading.

While the implementation prioritizes readability over efficiency, it provides a foundation for future work on decentralized training systems. Inspired by open-source efforts like NanoGPT (Karpathy, Andrej, 2024), we aim to foster further research and experimentation in decentralized learning.

4. Experiments

In this section, we validate the effectiveness of DiLoCo-SWARM through a series of experiments on a language modeling task. Our experiments evaluate the impact of DiLoCo-style gradient synchronization within SWARM on convergence, communication cost, and scalability across model sizes. The setup largely follows the experimental protocol of DiLoCo (Douillard et al., 2023), with adaptations for the SWARM framework.

4.1. Setup

We use the FineWeb-Edu dataset (Penedo et al., 2024), a large pre-training corpus of educational web content, for all experiments. As our model, we chose the GPT-2 family (Radford, A. and Metz, L. and Chintala, S., 2019), following the original architecture hyperparameters, as summarized in Table 1. To avoid additional gradient communication in pipelined settings, we do not share the weights of the embedding matrix and language modeling head. Unless otherwise stated, we use GPT-2 Small with roughly 180M parameters. We always train from randomly initialized weights.

Hyperparameters are adapted from the tuning done in DiLoCo (Douillard et al., 2023): The default local optimizer is AdamW (Loshchilov & Hutter, 2019) with a linearly warmed up learning rate of $4 \cdot 10^{-4}$, and a weight decay of 0.01. When DiLoCo is active, we use a Nesterov optimizer with a learning rate of 0.7 and a momentum of 0.9 as the global optimizer. A detailed description of the hyperparameters is provided in Table 3 in the Appendix.

Our main evaluation metric is the validation perplexity on a held-out set of 10M tokens from FineWeb-Edu. We evaluate during and after training to show the convergence against the number of training steps, as well as the total communication cost.

All experiments were conducted on a single node with eight co-located H100 GPUs on Prime Intellect Compute.

4.2. Main Experiment: DiLoCo-SWARM

Our main experiment compares DiLoCo-SWARM to two baselines: a single-node setup and a standard SWARM configuration. The weak baseline trains on a single GPU for

Model	Layers	Heads	Hidden Size	Params
Tiny	4	4	128	$\sim$14M
Small	12	12	768	$\sim$180M
Medium	24	16	1024	$\sim$405M
Large	36	20	1280	$\sim$800M

Table 1: **Model Sizes.** GPT-2 configurations used in experiments. Parameter counts reflect non-shared embedding and head weights.

2,000 steps, with a batch size of 512 and a sequence length of 1,024, resulting in a data budget of 1B tokens. The strong baseline is a 4x2 SWARM setup (eight workers distributed across two pipeline stages), which performs step-synchronous gradient synchronization. DiLoCo-SWARM follows the same 4x2 setup but replaces step-synchronous gradient synchronization with DiLoCo-style synchronization every 50 steps. This reduces the communication cost within data-parallel groups by a factor of 50.

Figure 6a shows that DiLoCo-SWARM closely matches the strong SWARM baseline, with a final validation perplexity of 30.15 compared to 30.22. This confirms that infrequent gradient synchronization does not negatively impact convergence.

Figure 6b shows the total communication cost for the same experiment. Surprisingly, the total communication cost is only moderately reduced in DiLoCo-SWARM compared to SWARM. This finding points at an important difference between DiLoCo and DiLoCo-SWARM. In DiLoCo, synchronizing gradients every H steps translates to a proportional reduction in communication cost. DiLoCo-SWARM, in contrast, combines pipeline and gradient synchronization communication, and the former is unaffected by DiLoCo. Therefore, the communication cost reduction depends on the fraction of the total communication cost incurred by gradient synchronization. Figure 7 shows the fraction of the communication cost for 4x2 SWARM training increasingly larger models. It illustrates that DP communication cost scales faster and dominates the total communication cost for larger models. This is due to the fact that increasing the model size only effects the pipeline communication cost through the hidden dimension H (sub-linear), while the cost for gradient synchronization is directly proportional to all model parameters. This finding suggests that DiLoCo-SWARM is more effective in reducing the communication cost for larger models.

4.3. Communication Frequency Ablation

We investigate the impact of varying the synchronization frequency within pipeline stages on convergence. Specifically, we train five models with synchronization intervals of $\{10, 20, 50, 100, 200\}$ steps, using the same setup as the main experiment.

Table 2 shows that generalization performance improves with more frequent synchronization. Synchronizing every 10 steps achieves the best validation perplexity of 27.95 but the performance degrades only mildly when decreasing the frequency. It is worth noting that this is more frequent compared to the 500 steps reported in DiLoCo. We attribute this difference to a significantly larger number of training steps in DiLoCo's setup, allowing for less frequent synchronization. We believe that with more training steps, a similar frequency is feasible with DiLoCo-SWARM.

Freq. (steps)	Val. PPL	Δ (Abs./Rel.)
10	27.95	-
20	28.61	+0.66 / +2.36%
50	30.15	+2.20 / +7.87%
100	30.49	+2.54 / +9.09%
200	31.27	+3.32 / +11.88%

Table 2: **Communication Frequency Ablation.** Validation perplexity at different synchronization intervals.

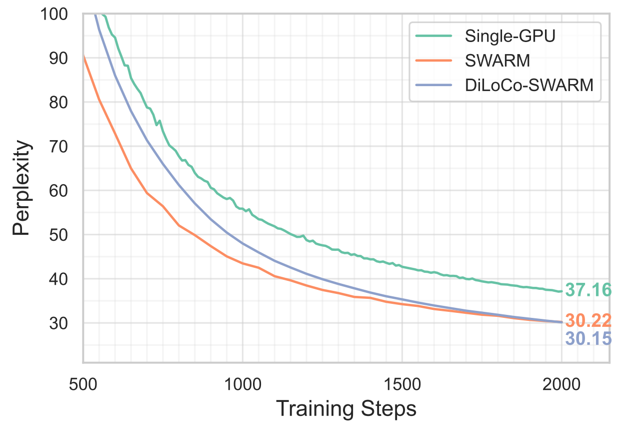
4.4. Model Size Ablation

Finally, we assess the scalability of DiLoCo-SWARM by training four GPT-2 model sizes (Tiny, Small, Medium, and Large) under the same conditions. Figure 8 shows that DiLoCo-SWARM consistently improves validation perplexity over the single-node baseline across all tested model sizes. For all, but the GPT-2 Large model, DiLoCo-SWARM outperforms the baseline by roughly 18%. We attribute the GPT-2 Large outlier to untuned hyperparameters, and hypothesize that similar results could be obtained with more careful tuning. Overall, we conclude that DiLoCo-style gradient synchronization scales effectively to larger models for the range of model sizes tested.

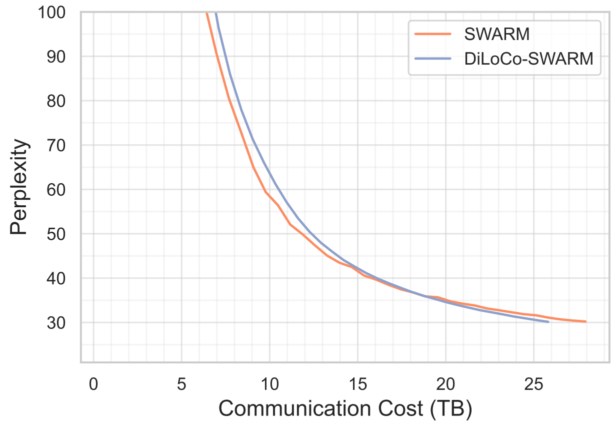
5. Conclusion

In this work, we introduced DiLoCo-SWARM, a decentralized training method that combines the communication efficiency of gradient synchronization of DiLoCo with the scalability and fault-tolerance of SWARM parallelism. By combining both methods, DiLoCo-SWARM reduces communication costs without compromising convergence performance.

Our experimental results on language modeling tasks using the GPT-2 family of models demonstrate that DiLoCo-SWARM matches the generalization performance of traditional SWARM while reducing gradient synchronization frequency by 50x. We analytically show that DiLoCo-



(a) Validation perplexity vs. training steps.



(b) Validation perplexity vs. communication cost.

Figure 6: **Main Result.** DiLoCo-SWARM matches the strong baseline in generalization performance, achieving a final validation perplexity of 30.15. Despite synchronizing gradients 50x fewer times, DiLoCo-SWARM incurs only a moderate reduction in total communication cost due to the dominance of pipeline communication in small models.

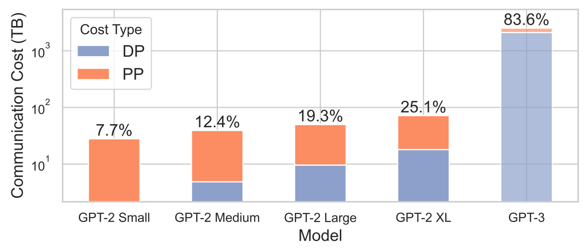


Figure 7: **Communication Cost Scaling** We show analytically derived (log) communication cost incurred by DP and PP communication for 4x2 SWARMs in the common training setup. PP communication dominates the total communication for smaller models, but DP communication dominates for larger models.

SWARM shows stronger communication savings for larger models, where data-parallel communication becomes the dominant cost. These findings suggest that DiLoCo-style synchronization is compatible with SWARM and can be a practical solution for scaling decentralized training to larger models.

6. Future Work

While this work demonstrates the feasibility of DiLoCo-SWARM, several limitations remain, suggesting directions for future research.

6.1. Hyperparameter Tuning

Our experiments assumed that the hyperparameters from DiLoCo (Douillard et al., 2023) directly apply to DiLoCo-SWARM. While these hyperparameters provide a reasonable starting point, they may not be optimal for the different dataset, model architecture, and training setup used in this work. Future research should include a thorough hyperparameter sweep, focusing on key parameters such as batch size, learning rates, and learning rate schedules. Given the compute budget constraints, hyperparameter tuning should first be conducted on smaller models and then extrapolated to larger scales.

6.2. Training Duration

The experiments in this work were conducted on the GPT-2 family of models, which are small by today's standards. Additionally, models were trained on less than 10 billion tokens, meaning they were still in the early stages of training. In contrast, prior DiLoCo and SWARM experiments trained models on over 50 billion tokens, requiring multiple days of GPU time. To obtain more robust results on late-stage training performance, future experiments should consider increasing the data budget and training duration.

6.3. Decentralized Setting

All experiments were conducted on co-located GPUs with fast and reliable interconnectivity. However, one of the primary motivators for SWARM is to enable decentralized training across heterogeneous and unreliable hardware with slower network connections. While our findings suggest that DiLoCo-style gradient synchronization is compatible with SWARM, they remain to be validated in a truly decentralized setting with more pipeline stages and nodes per stage. Future work should explore DiLoCo-SWARM in real-world decentralized environments. This likely requires

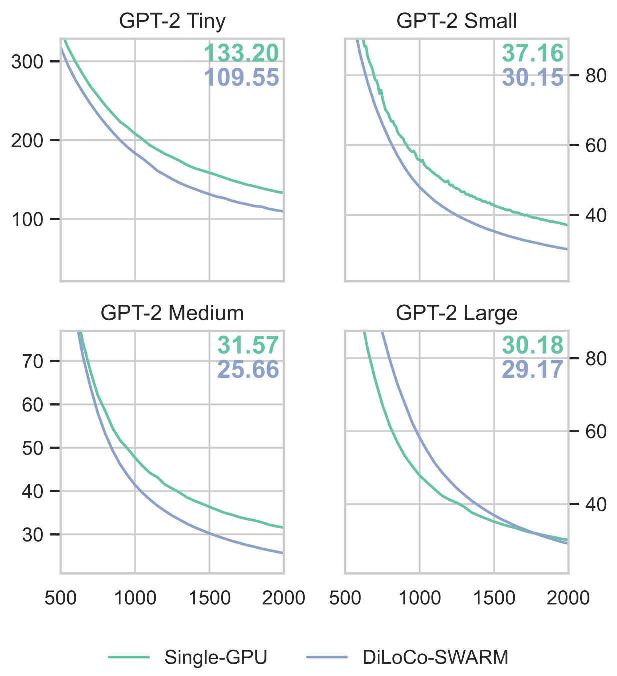


Figure 8: **Scalability with Model Size.** DiLoCo-SWARM improves generalization performance over the single-node baseline across different GPT-2 model sizes.

further optimizing the implementation of both DiLoCo-SWARM, specifically SWARM's fault-tolerant communication mechanisms. These optimizations are necessary to conduct experiments at larger scales to harvest the full benefits of DiLoCo-SWARM.

Acknowledgements

This work was developed in collaboration with the Scalable Computing Systems (SaCS) lab at EPFL and Prime Intellect. I want to thank Martijn, Akash and Rishi from the SaCS lab for the fruitful discussions in our weekly meetings, as well as useful feedback during the writing of this paper. Further, I thank the whole Prime Intellect research team, in particular Sami and Johannes, for their constant support, and for providing the compute infrastructure for the experiments.

References

Borzunov, A., Baranchuk, D., Dettmers, T., Riabinin, M., Belkada, Y., Chumachenko, A., Samygin, P., and Raffel, C. Petals: Collaborative Inference and Fine-tuning of Large Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 558–568, 2023. URL https://arxiv.org/abs/2209.01188.

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, 2020. URL https://arxiv.org/abs/2005.14165.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., and Mishra, G. PaLM: Scaling Language Modeling with Pathways, 2022. URL https://arxiv.org/abs/2204.02311.

Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., and Xing, E. P. Geeps: scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342407. doi: 10.1145/2901318.2901323. URL https://doi.org/10.1145/2901318.2901323.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q., Mao, M., Ranzato, A., Senior, A., Tucker, P., Yang, K., and Ng, A. Large Scale Distributed Deep Networks. *Advances in neural information processing systems*, 10 2012.

DeepSeek-AI. DeepSeek-V3 Technical Report, 2024. URL https://arxiv.org/abs/2412.19437.

Diskin, M., Bukhtiyarov, A., Ryabinin, M., Saulnier, L., Lhoest, Q., Sinitsin, A., Popov, D., Pyrkin, D., Kashirin, M., Borzunov, A., del Moral, A. V., Mazur, D., Kobelev, I., Jernite, Y., Wolf, T., and Pekhimenko, G. Collaborative Learning in Open Source, 2021. URL https://arxiv.org/abs/2106.10207.

Douillard, A., Feng, Q., Rusu, A. A., Chhaparia, R., Donchev, Y., Kuncoro, A., Ranzato, M., Szlam, A., and Shen, J. DiLoCo: Distributed Low-Communication Training of Language Models, 2023. URL https://arxiv.org/abs/2311.08105.

Dubey, A., Jauhri, A., Pandey, A., and .... The Llama 3 Herd of Models, 2024. URL https://arxiv.org/abs/2407.21783.

EXO. SPARTA: Distributed Training with Sparse Parameter Averaging. https://blog.exolabs.net/day-12/, 2025. Accessed: 2025-01-08.

Fernandez, J., Wehrstedt, L., Shamis, L., Elhoushi, M., Saladi, K., Bisk, Y., Strubell, E., and Kahn, J. Hardware

Scaling Trends and Diminishing Returns in Large-Scale Distributed Training, 2024. URL https://arxiv.org/abs/2411.13055.

Gemini. Gemini: A Family of Highly Capable Multimodal Models, 2024. URL https://arxiv.org/abs/2312.11805.

Hagemann, J., Weinbach, S., Dobler, K., Schall, M., and de Melo, G. Efficient Parallelization Layouts for Large-Scale Distributed Model Training, 2024. URL https://arxiv.org/abs/2311.05610.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. PipeDream: Fast and Efficient Pipeline Parallel DNN Training, 2018. URL https://arxiv.org/abs/1806.03377.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism, 2019. URL https://arxiv.org/abs/1811.06965.

Jaghouar, S., Ong, J. M., Basra, M., Obeid, F., Straube, J., Keiblinger, M., Bakouch, E., Atkins, L., Panahi, M., Goddard, C., Ryabinin, M., and Hagemann, J. INTELLECT-1 Technical Report, 2024a. URL https://arxiv.org/abs/2412.01152.

Jaghouar, S., Ong, J. M., and Hagemann, J. OpenDiLoCo: An Open-Source Framework for Globally Distributed Low-Communication Training, 2024b. URL https://arxiv.org/abs/2407.07852.

Karpathy, Andrej. NanoGPT. https://github.com/karpathy/nanoGPT, 2024.

Loshchilov, I. and Hutter, F. Decoupled Weight Decay Regularization, 2019. URL https://arxiv.org/abs/1711.05101.

McMahan, H. B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. Communication-Efficient Learning of Deep Networks from Decentralized Data, 2016. URL https://arxiv.org/abs/1602.05629.

Penedo, G., Kydlíček, H., allal, L. B., Lozhkov, A., Mitchell, M., Raffel, C., Werra, L. V., and Wolf, T. The fineweb datasets: Decanting the web for the finest text data at scale, 2024. URL https://arxiv.org/abs/2406.17557.

Peng, B., Quesnelle, J., and Kingma, D. P. DeMo: Decoupled Momentum Optimization, 2024. URL https://arxiv.org/abs/2411.19870.

Radford, A. and Metz, L. and Chintala, S. Language Models are Unsupervised Multitask Learners. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf, 2019.

Ryabinin, M., Dettmers, T., Diskin, M., and Borzunov, A. SWARM Parallelism: Training Large Models Can Be Surprisingly Communication-Efficient, 2023. URL https://arxiv.org/abs/2301.11913.

Wang, H., Yurochkin, M., Sun, Y., Papailiopoulos, D., and Khazaeni, Y. Federated Learning with Matched Averaging, 2020. URL https://arxiv.org/abs/2002.06440.

Zhang, X., Wang, J., Joshi, G., and Joe-Wong, C. Machine learning on volatile instances, 2020. URL https://arxiv.org/abs/2003.05649.

A. Appendix

A.1. Training Script Invocations

Below we show example invocations for different distributed training algorithms. All examples use a GPT-2 Small model and the FineWeb-Edu dataset as an example. For more details see the README on GitHub.

```
# Single GPU training
torchrun --nproc_per_node 1 src/train.py --swarm.num_stages 1 \
    --model @configs/model/gpt2-small.toml \
    --data @configs/data/fineweb-edu-10bt.toml
```

```
# Pipeline parallel training with 2 GPUs
torchrun --nproc_per_node 2 src/train.py --swarm.num_stages 2 \
    --model @configs/model/gpt2-small.toml \
    --data @configs/data/fineweb-edu-10bt.toml
```

```
# Data parallel training with 2 GPUs
torchrun --nproc_per_node 2 src/train.py --swarm.num_stages 2 \
    --model @configs/model/gpt2-small.toml \
    --data @configs/data/fineweb-edu-10bt.toml
```

```
# DiLoCo training with 2 GPUs
torchrun --nproc_per_node 2 src/train.py --swarm.num_stages 2 \
    --train.outer_optimizer @configs/optimizer/nesterov.toml
    --model @configs/model/gpt2-small.toml \
    --data @configs/data/fineweb-edu-10bt.toml
```

```
# SWARM training with 4 GPUs
torchrun --nproc_per_node 4 src/train.py --swarm.num_stages 2 \
    --model @configs/model/gpt2-small.toml \
    --data @configs/data/fineweb-edu-10bt.toml
```

```
# DiLoCo-SWARM training with 4 GPUs
torchrun --nproc_per_node 4 src/train.py --swarm.num_stages 2 \
    --train.outer_optimizer configs/optimizer/nesterov.toml
    --model @configs/model/gpt2-small.toml \
    --data @configs/data/fineweb-edu-10bt.toml
```

A.2. Hyperparameters

Table 3 shows the hyperparameters used throughout the experiments. The outer optimizer parameters is only used for DiLoCo-style training. For non-DiLoCo runs gradients are averaged before performing a regular local optimizer step.

Hyperparameter	**Value**	
General	Batch Size	512
	Sequence Length	1024
	Steps	2000
	Num. Warmup Steps	50
Local Optimizer	Name	AdamW
	Weight decay	0.01
	Learning Rate	4×10^{-4}
Global Optimizer	Name	Nesterov
	Learning Rate	0.7
	Momentum	0.9
SWARM	Num. Stages	2
	Num. Nodes per Stage	4

Table 3: Hyperparameters