OpenDiLoCo: An Open-Source Framework for Globally Distributed Low-Communication Training

Sami Jaghouar [1] Johannes Hagemann [1]

Abstract

OpenDiLoCo is an open-source implementation and replication of the Distributed Low-Communication (DiLoCo) training method for large language models. We provide a reproducible implementation of the DiLoCo experiments, offering it within a scalable, decentralized training framework using the Hivemind library. We demonstrate its effectiveness by training a model across two continents and four countries. Additionally, we conduct an analytical evaluation of its practicality, focusing on the algorithm's compute efficiency and scalability in the number of workers. Our findings indicate that while DiLoCo can be effective in specific scenarios, it is not necessarily a low-communication replacement for Distributed Data Parallel training due to its lower compute efficiency over a smaller number of steps.

1. Introduction

Large language models (LLMs) have revolutionized numerous applications of machine learning, yet training these models requires substantial computational resources typically concentrated in a single, well-connected cluster to efficiently parallelize workloads for distributed model training (Hagemann et al., 2023). Novel approaches, such as DiLoCo by Douillard et al., address these challenges by enabling efficient training across multiple, poorly connected devices. Their approach dramatically reduces the need for frequent communication, making it feasible to train LLMs on a global scale.

We reproduce DiLoCo's results in an open manner and implement them in a real-world setting using the Hivemind library (team, 2020), showcasing its applications and analyzing its compute efficiency. In summary, the contributions of our work are as follows:

- **Open Reproduction of DiLoCo Experiments:** We replicate the DiLoCo experiments for a large language model pre-training and validate their results in a reproducible manner.
- **Open-Source Implementation:** We provide implementations of DiLoCo built on top of the Hivemind library alongside a concise 180-line PyTorch one, significantly lowering the barrier for performing decentralized training.
- **Global Decentralized Training:** We demonstrate our approach in a real-world decentralized training setting executed across two continents and four countries and achieve 90-95% compute utilization.
- **Analytical Insights and Ablations:** We conduct an ablation study of DiLoCo, focusing on the algorithm's scalability in the number of workers and compute efficiency.

We publish the full data of our experiments, the Hivemind as well as the PyTorch distributed training code implementation of OpenDiLoCo on GitHub at github.com/PrimeIntellect-ai/OpenDiLoCo.

2. Implementation

DiLoCo is a local SGD algorithm (Stich, 2019) that leverages two distinct optimization processes: an inner optimizer and an outer optimizer. The inner optimizer, AdamW (Loshchilov & Hutter, 2017), performs local updates on individual workers, while the outer optimizer, SGD with Nesterov momentum (Nesterov, 1983), synchronizes the workers using pseudo-gradients calculated by subtracting the locally updated weight $\theta(t+h)$ from the original weight $\theta(t)$.

This local SGD approach significantly reduces the frequency of communication (up to 500 times), thus lowering the bandwidth requirements for distributed training.

[1]Prime Intellect, Inc.. Correspondence to: Johannes Hagemann <johannes@primeintellect.ai>.

Accepted to the Workshop on Efficient Systems for Foundation Models at the International Conference on Machine Learning (ES-FoMo@ICML2024).

General Implementation Details Our implementation of DiLoCo instantiates the two optimizers (inner and outer) and creates two copies of the model: the main model $\theta(t+h)$, which will be updated by the inner optimizer, and a copy of the original weights, $\theta(t)$, which is needed to compute the pseudo-gradient. The inner optimizer is called at the end of each step, while the outer optimizer is called periodically. Both of our implementations compute the pseudo-gradients manually and store them in fp32 inside the PyTorch gradient buffer (within $param.grad$) of the model.

In mixed precision training (Micikevicius et al., 2017) with fp16, a gradient scaler is used to improve the dynamic range of the gradients while avoiding underflow and overflow. The gradient scaler should be called during the inner optimization step but not during the outer one because the pseudo-gradients are calculated manually in fp32.

We offer two open source DiLoCo implementations, one reference implementation using `torch.distributed` and an implementation built using the Hivemind library for a more practical decentralized training setting.

Implementation with `torch.distributed` The following details our PyTorch implementation, utilizing the `torch.distributed` package with NCCL for the communication backend. This implementation was used for our main experiments.

```
for batch, step in enumerate(train_loader):
    ... # loss calculation
    inner_optimizer.step()
    if real_step % local_steps == 0:
        for old_param, param in \
                zip(original_params, model.parameters()):

            param.grad = old_param - param.data
            dist.all_reduce(
                tensor=param.grad,
                op=dist.ReduceOp.AVG
            )
            param.data = old_param
        outer_optimizer.step()
    original_params = [
        p.detach().clone() for p in model.parameters()
    ]
```

Figure 1. **Pseudo-Code for Outer Optimizer in OpenDiLoCo.**

We highlight the most important outer optimization part in Figure 1.

```
from hivemind.dht.dht import DHT
from open_diloco import DiLoCoOptimizer

optimizer = DiLoCoOptimizer(
    bs, # batch size
    ls, # learning rate scheduler
    DHT(), # distributed hash table for coordination
    i_opt, # inner optimizer
    o_opt, # outer optimizer
    m.params() # model parameters
)

for batch in train_dataloader:

    model(batch).loss.backward()
    optimizer.step()
    # the outer step, including peer synchronization
    # and communication, is triggered automatically
    # after all local steps
    optimizer.zero_grad()
```

Figure 2. **OpenDiLoCo - Hivemind API.**

Hivemind Implementation Our second implementation is built on top of the Hivemind framework. Instead of using `torch.distributed` for the worker communication, Hivemind utilizes a distributed hash table (DHT) spread across each worker to communicate metadata and synchronize them. This DHT is implemented using the open-source libp2p project [1]. Hivemind provides an optimized all-reduce algorithm designed for execution on a pool of poorly connected workers.

Unlike the `torch.distributed` implementation, our Hivemind implementation wraps both optimizers into a single optimizer class, making it compatible with popular training codebases that assume a single optimizer, such as the Hugging Face Trainer. This allows for the use of OpenDiLoCo via a simple Hivemind-compatible API by instantiating a customizable DiLoCoOptimizer, as shown in Figure 2.

Our integration with Hivemind enables a real-world decentralized training setup for DiLoCo, making many of its inherent properties usable, such as:

- **On/Off ramping of resources:** The amount of available compute may not be constant, with new devices and clusters coming and going.
- **Fault tolerance:** For decentralized training, some devices may be less reliable than others. Through Hivemind's fault-tolerant training, a device could become unavailable at any time without stopping the training process.

[1] https://github.com/libp2p/libp2p

- **Peer-to-Peer:** There is no master node. All communication is done in a peer-to-peer fashion.

Contrary to the `torch.distributed` implementation, our Hivemind one wraps both optimizers into a one optimizer class, making it more user friendly and compatible with other open source frameworks.

3. Experiment

Experiment Setup Our OpenDiLoCo experiment setup largely follows the main experiments from Douillard et al.. We conduct various experiments using a model with 150 million parameters on a language modeling task using the C4 dataset (Raffel et al., 2019). The hyperparameters are consistent with DiLoCo across experiments: an inner learning rate of $4e^{-4}$, $1{,}000$ warm-up steps, 0.1 weight decay, a batch size of 512, a sequence length of $1{,}024$, a learning rate for the Nesterov outer optimizer of 0.7, and Nesterov momentum of 0.9. Similarly, we run the experiments for a total of $88{,}000$ steps.
The one difference in our experiment setup is that we choose the Llama (Touvron et al., 2023) model architecture for our experiments, due to its recent popularity, while the original DiLoCo authors used the Chinchilla architecture (Hoffmann et al., 2022). These two architectures are generally quite similar but have slight differences. For instance, Llama uses the SwiGLU activation function (Shazeer, 2020) for the MLP and has a dimension of $\frac{2}{3}4d$ instead of $4d$. For more details about the model configuration, see Appendix A.

In addition to the DiLoCo experiments, we conduct experiments with a varying number of workers to analyze if diminishing returns occur before reaching the 8 workers reported in the DiLoCo work and to generally measure the FLOP efficiency of the algorithm.

We also run experiments in a real-world decentralized training setup, training across workers from four different countries simultaneously.

Our baseline experiments follow a similar setup as Douillard et al.. We use two baselines: the first is a weak baseline that runs without DiLoCo and without replicas for $88{,}000$ steps. The second is a stronger baseline, which uses an $8\times$ larger batch size with data parallelism, maintaining a similar compute budget as our DiLoCo experiment but with significantly larger communication requirements.

Main Results Figure 3 shows our main experimental results. It demonstrates that DiLoCo with 8 replicas significantly outperforms the baseline without any replicas and matches the performance of the stronger baseline with $500\times$ larger communication requirements, as indicated by the final perplexity results in Table 1. These findings are consistent with the main experimental results of Douillard et al.. One noticeable difference is that in Douillard et al.'s experiments, the DiLoCo run is already approaching and surpassing the stronger baseline at around $64{,}000$ steps, while our DiLoCo training run only starts to exactly match the performance of the strong baseline at the end of the training at $88{,}000$ steps. This difference might be due to the fact that their experiments start from a checkpoint with $24{,}000$ pre-training steps, while ours start from scratch.

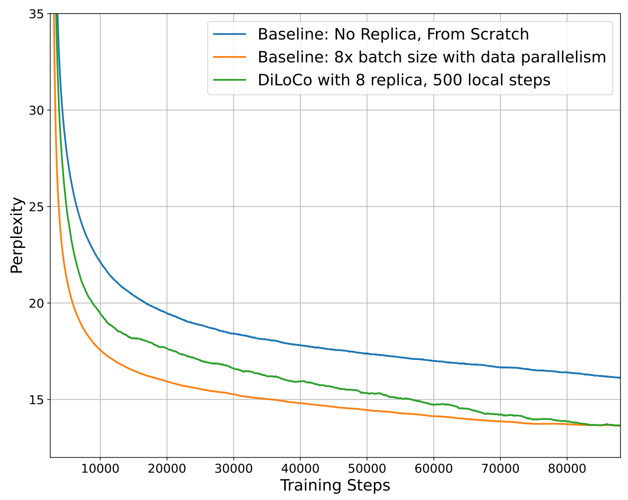

Figure 3. **Main result:** 150 million parameter Llama model pretraining with 8 DiLoCo workers yields significantly lower perplexity than the baseline without DiLoCo, and even compared to the baseline using 8 times larger batch size with the same compute budget, while communicating 500 times less.

DiLoCo Number of Worker Ablation To determine the compute efficiency of DiLoCo, we conduct an ablation study on the number of workers, as shown in Figure 4. These experiments are set up identically to our main experiment, with the only difference being a reduction in the local step size from 500 to 50.

Our results demonstrate a steady improvement in perplexity as the number of workers in DiLoCo increases.

Furthermore, Figure 7 presents the same ablation as Figure 4, but with the x-axis representing global steps instead of local steps. This provides a more accurate approximation of DiLoCo's FLOP efficiency by comparing the total compute spent on the model. These results reinforce our previous observation: DiLoCo with more than one worker is initially not as compute efficient as the same number of global steps on a single machine or when using Distributed Data Parallel training. DiLoCo may only achieve comparable FLOP efficiency after a large number of steps due to slower initial convergence, as shown in our main experiment in Figure 3.

Model	Communication	Time	Compute & Data	Perplexity
Baseline, no replica, from scratch	0	1×	1×	16.17
Baseline, 8× batch size with DP	$8 \times N$	1×	8×	13.68
DiLoCo, 8 replica, 500 local steps	$8 \times \frac{N}{H}$	1×	8×	**13.73**

Table 1. **Final Evaluation Perplexity Comparison**: We compare our two baselines *vs* DiLoCo with 8 replicas for a 150 million parameter model pre-training across their communication cost, time spent, compute & data used and final perplexity after 88,000 steps, similar to Douillard et al.. For the same time and amount of compute, we can compare the second baseline and DiLoCo. The former communicates gradients at each time step (N total steps), while DiLoCo communicates $H = 500$ times less.

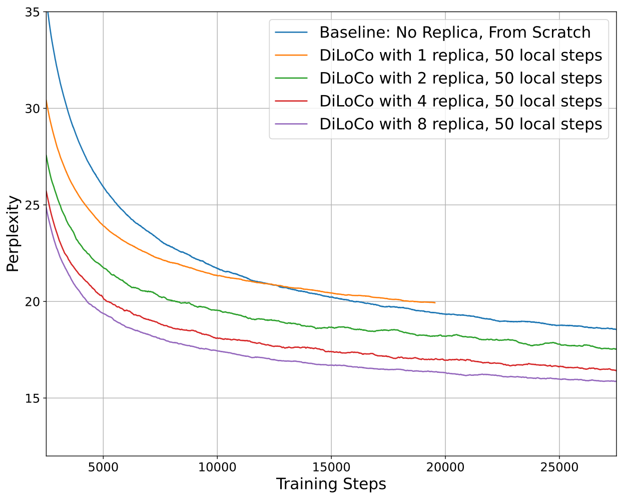

Figure 4. **Ablation Study on the Number of Workers in DiLoCo:** Performance comparison of DiLoCo with different numbers of workers and 50 local steps against the baseline without DiLoCo. Due to compute constraints, these ablation experiments were not extended to 88,000 steps like the other experiments.

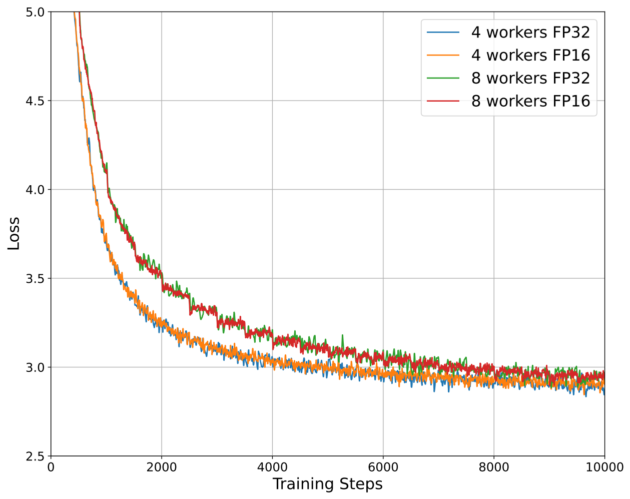

Figure 5. **FP16 vs FP32 All-Reduce Ablation:** The first group is 4 workers and 50 local steps, the second group is 8 workers and 500 local steps.

Practical Usage According to our main experimental results in Figure 3, eight DiLoCo workers yield a final perplexity comparable to that of DDP after 88,000 steps. However, training for only 44,000 steps with eight workers results in a significantly worse performing model than DDP with the same number of global steps, making four DiLoCo workers a more efficient choice in this case. Our interpretation suggests that while training with eight DiLoCo workers ultimately results in a stronger model, increasing the number of workers does not accelerate the initial convergence phase as data parallelism would.

All-Reduce in FP16 Our main experiments perform the all-reduce operation of the pseudo gradients in FP32, following the original methodology outlined in the DiLoCo paper. We repeated the DiLoCo experiment, this time using FP16 for the pseudo gradient.

Figure 5 shows there is no noticeable impact on performance both with 8 workers and 500 local steps and 4 workers and 50 local steps, indicating that FP16 all-reduce is effective for use with DiLoCo and can halve the communication time required for the all-reduce operation.

Globally Distributed Training Setting To showcase the functionality of decentralized training with OpenDiLoCo executed across four countries, we utilize our Hivemind implementation. We use four DiLoCo workers, each with one H100 GPU, located in Canada, Finland, Poland, and the United States, respectively. Figure 6 shows the network bandwidth between the workers, which varies between 80-300 Mbit/s. We use a 150 million parameter model with 500 local steps, as in our main experiment. We start our training from a checkpoint pre-trained with 19k steps obtained from the baseline of our main experiment.

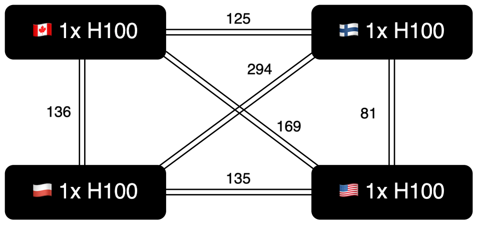

Figure 6. **Network Bandwidth between Workers:** Average bidirectional network bandwidth between the four different workers in our decentralized training setup (**in Mbit/s**). The GPUs are located in four different countries and hosted by different cloud providers: Canada (Oblivus), Finland (DataCrunch), Poland (Scaleway), and the USA (Lambda Labs). Measured using the iperf package [3].

Through the large number of local steps, the four workers run independently for around 57 minutes before communicating for gradient averaging. For the outer optimizer step, our experiment shows an average all-reduce time between the workers of 180 seconds. Additionally, we observed variations in the training speed between our different cloud instances. Although all workers had the same GPU type, we could not control for configuration variables such as the number of CPU cores and the amount of RAM, which led to slightly different training times for the 500 inner steps. Since our current DiLoCo implementation requires every worker to complete the same number of local steps, this resulted in an additional waiting time of up to 200 seconds for the fastest worker.
Nevertheless, due to the significant reduction in communication time, the all-reduce bottleneck only accounts for 5.2% of the training time, minimally impacting the overall training speed. An additional 5.8% of the training time is spent idling by the fastest worker in our scenario. In future work, we will address this issue by exploring DiLoCo in an asynchronous setting, as done by Liu et al..

4. Conclusion

We successfully reproduced the main experiment results from DiLoCo and demonstrate its application in a real-world decentralized training setting. We train a large language model using our OpenDiLoCo implementation across 2 continents and 4 countries and achieve 90-95% compute utilization through the low-communication training approach.

While scaling DiLoCo to more than eight workers is a promising research direction for enabling effective, low-communication training across globally distributed GPUs, our ablation study shows using eight workers does not yet match the computational efficiency of Distributed Data Parallel (DDP) when running for a shorter amount of steps. However, DiLoCo exhibit strong performance with two or four replicas comparable to DDP and this already opens up practical applications.

For future work, more compute-efficient methods need to be developed for decentralized training, which also improve the scalability to support a significantly larger number of distributed workers. Additionally, efforts will be directed towards scaling OpenDiLoCo to test the algorithm's scaling behavior on larger model sizes, further enhancing its applicability and efficiency in real-world scenarios.

Acknowledgements

We want to thank Max Ryabinin for his guidance and help with the Hivemind library. His insights have been very helpful for our project.

We would also like to thank Arthur Douillard for his work on DiLoCo and for helping us figure out the details of reproducing the original experiments.

References

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.

Hagemann, J., Weinbach, S., Dobler, K., Schall, M., and de Melo, G. Efficient parallelization layouts for large-scale distributed model training, 2023.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Rae, J. W., Vinyals, O., and Sifre, L. Training compute-optimal large language models, 2022.

Liu, B., Chhaparia, R., Douillard, A., Kale, S., Rusu, A. A., Shen, J., Szlam, A., and Ranzato, M. Asynchronous local-sgd training for language modeling, 2024.

Loshchilov, I. and Hutter, F. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017. URL http://arxiv.org/abs/1711.05101.

Micikevicius, P., Narang, S., Alben, J., Diamos, G. F., Elsen, E., García, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training. *CoRR*, abs/1710.03740, 2017. URL http://arxiv.org/abs/1710.03740.

Nesterov, Y. A method for solving the convex programming problem with convergence rate $o(1/k^2). Proceedings of the USSR Academy of Sciences, 269: 543--547, 1983. URL.$

Raffel, C., Shazeer, N. M., 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, 2019. URL https://api.semanticscholar.org/CorpusID:204838007.

Shazeer, N. Glu variants improve transformer, 2020.

Stich, S. U. Local sgd converges fast and communicates little, 2019.

team, L. Hivemind: a Library for Decentralized Deep Learning. https://github.com/learning-at-home/hivemind, 2020.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023.

A. Model Configuration

Model Parameters	150M
Number of layers	12
Hidden dim	1024
Number of heads	16
K/V size	64
Vocab size	32,000
Inner learning rate (AdamW)	$4e^{-4}$
Number of warmup steps	1,000
Weight decay	0.1
Batch Size	512
Sequence length	1,024
Outer Nesterov learning rate	0.7
Outer Nesterov momentum	0.9

Table 2. **Model Configuration** for the DiLoCo experiments. The models are based on the Llama architecture (Touvron et al., 2023).

B. FLOP Efficiency Comparison

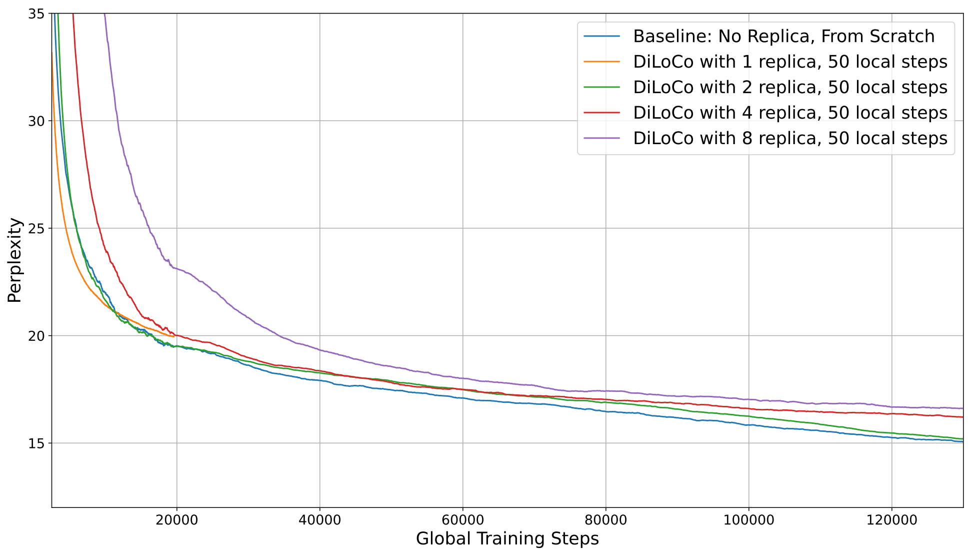

Figure 7. **Ablation Study on FLOP Efficiency Relative to Number of Workers in DiLoCo:** This figure compares the performance of DiLoCo with different numbers of workers and 50 local steps against the baseline without DiLoCo. The x-axis shows the global steps instead of local steps, providing a better approximation of DiLoCo's FLOP efficiency by comparing the total amount of compute spent on the model.