OPT2024: 16th Annual Workshop on Optimization for Machine Learning

ACCO: Accumulate while you Communicate, Hiding Communications in Distributed LLM Training

Adel Nabli[1,2] ADEL.NABLI@SORBONNE-UNIVERSITE.FR
Louis Fournier[1]
Pierre Erbacher[1]
Louis Serrano[1]
Eugene Belilovsky[2]
Edouard Oyallon[1]
[1] *Sorbonne Université, CNRS, ISIR, Paris - France*
[2] *Mila - Quebec AI Institute, Concordia University, Montréal - Québec*

Abstract

Training Large Language Models (LLMs) relies heavily on distributed implementations, employing multiple GPUs to compute stochastic gradients on model replicas in parallel. However, synchronizing gradients in data parallel settings induces a communication overhead increasing with the number of distributed workers, impeding the efficiency gains of parallelization. To address this challenge, local optimization algorithms such as the ones used in Federated Learning have emerged. While effective in minimizing communication overhead, they incur significant memory costs, hindering scalability: in addition to extra momentum variables, optimizer's states cannot be partitioned among workers as communications are only allowed between rounds of local optimization steps. To conceal communication costs, we propose instead to synchronize delayed gradients *while* computing new ones between each model's update. Accumulating local gradients on the workers until the communication finishes naturally reduces the idle time of GPUs and even allows the use of heterogeneous hardware. However, we show that the one-step delay inherent in parallel execution of gradient computations and communications has drastic impacts on Transformers' convergence. To compensate this delay we introduce a novel technique, **AC**cumulate while **CO**mmunicate (`ACCO`), a memory-efficient optimization algorithm tailored for distributed training of LLMs which leads to training dynamics aligned with standard distributed optimization. Compared to ZeRO, our implementation and experiments on several LLMs pre-training and fine-tuning tasks demonstrates that `ACCO` reduces the learning time up to 87% and successfully allows both sharding optimizer states across workers and the use of heterogeneous hardware.

1. Introduction

Training modern Large Language Models (LLMs) with billions of parameters requires thousands of GPUs running in parallel [67]. This is done by relying on a distributed version of the backpropagation algorithm [30] with a gradient-based optimizer such as Adam [25] or AdamW [34]. However at this scale, the communication overhead necessary to synchronize gradients between workers in the data parallel setting can dominate the time to compute the model updates [47], and it has been estimated that it will remain the case even if models and hardware evolve [50], hindering the benefits of parallelization. Moreover, as all workers are synchronized through gradient communication, the training only proceeds at the speed of the slowest machine (straggler) [11, 39].

To alleviate this issue, distributed optimization algorithms reducing the amount of communication between workers have been developed, such as local optimization methods [63, 71] which are especially used in Federated Learning [27, 38]. These methods authorize performing multiple optimization steps *locally* before communicating and synchronizing the distributed workers, reducing the communication overhead. As communication rounds can last longer than a local gradient computation (see Fig. 1), they also naturally allow to hide the cost of communications in the training time by running them in parallel to several consecutive local computation steps [59, 65, 70, 78]. Moreover, on heterogeneous hardware, the number of computation steps can be tuned locally to the worker's speed so that slow ones compute less than fast ones, maxing out workers' usage [10, 36].

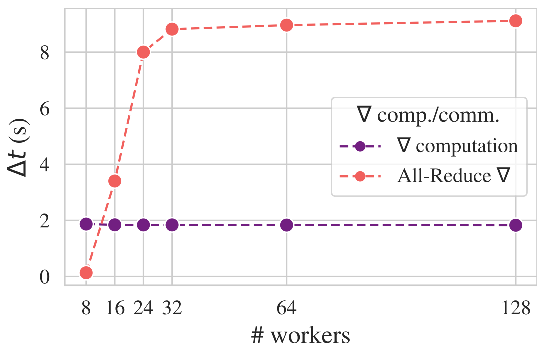


Figure 1: Time spent computing and averaging gradients of a Llama-2 7B model depending on the number of workers (GPUs).

However, this comes at a drastic memory cost. Indeed, in the standard data parallel setting, most of the memory consumption of model states comes from storing the optimizer's parameters, especially when training with mixed precision. To avoid the replication of redundant optimizer states across the workers, methods such as ZeRO [52] shard them. Due to limited GPU memory and large models' size, all frameworks used in practice nowadays to train LLMs at scale use a form of partitioning method [2, 55]. However these sharding methods rely heavily on the fact that *each* mini-batch gradient is averaged over all the workers during the backward step. This is no longer the case with local optimization algorithms: if it were, then an averaging would happen at each step, defeating the purpose of the local method. This forces each worker to host a full copy of the optimizer's parameters, increasing the memory requirements. Moreover, to prevent local steps from reducing the accuracy of the resulting model, local methods often introduce an outer optimizer step at each communication, which comes with additional momentum terms [65, 71], leading to significant memory overheads as shown in Tab. 1. This raises the following question:

Is it possible to design a memory-efficient optimization algorithm that hides the communication cost of distributed training of LLMs and accommodates heterogeneous hardware?

To completely hide the communication cost while being memory-efficient, making sharded optimizers compatible with the idea of overlapping gradient computations and communications seems natural. The concept of running two parallel processes is already present in the sharded optimization literature, but for a different purpose. ZeRO-Offload [57] introduces the ”Delayed Parameter Update” (DPU) which allows running the optimizer on the CPU while computing and averaging gradients on the GPU. By running these processes in parallel, the gradients computed during one step are on a version of the model parameters that are no longer up to date, as they have been updated by the optimizer concurrently. In practice, this one-step staleness hurts convergence, and the method can only be used after sufficiently many warmup steps of non-delayed optimization [57].

Contributions. We introduce **AC**cumulate while **CO**mmunicate (`ACCO`), a memory-efficient optimization algorithm that **(1)** allows to shard the optimizer parameters across workers, **(2)** overlaps gradients computations and communications, hiding the communication overhead while **(3)**

Table 1: Comparison of characteristics and memory consumption between several methods. Ψ: number of parameters in the model. N: number of workers. K: memory multiplier of the optimizer (Adam). While no additional momentum is required for our method, we still need a communication buffer.

Method	No comm. overhead	Handle hetero. hardware	Sharded Opt.	No add. momentum	Memory consumed per worker	$K=12,\ N=64,$ $\Psi=7.5$B
Baseline DDP [30]	✗	✗	✗	✓	$(2{+}2{+}K)\times\Psi$	120 GB
ZeRO-1 [52]	✗	✗	✓	✓	$(2{+}2{+}\frac{K}{N})\times\Psi$	31 GB
SlowMo [71]	∼	✗	✗	✗	$(2{+}2{+}2{\times}2{+}K)\times\Psi$	150 GB
CO2 [65]	✓	✗	✗	✗	$(2{+}2{+}4{\times}2{+}K)\times\Psi$	180 GB
ACCO (Ours)	✓	✓	✓	✓	$(2{+}2{+}2{+}\frac{K}{N})\times\Psi$	46 GB

maximizing GPU usage, even with heterogeneous hardware. **(4)** We introduce a novel method to compensate for the one-step delay induced by parallel execution of the gradient computations and communications, removing the need for warmup steps and **(5)** perfectly matching the training dynamic of standard distributed optimization. Our experiments across multiple LLMs training and fine-tuning tasks consistently show that ACCO allows for significant time gains.

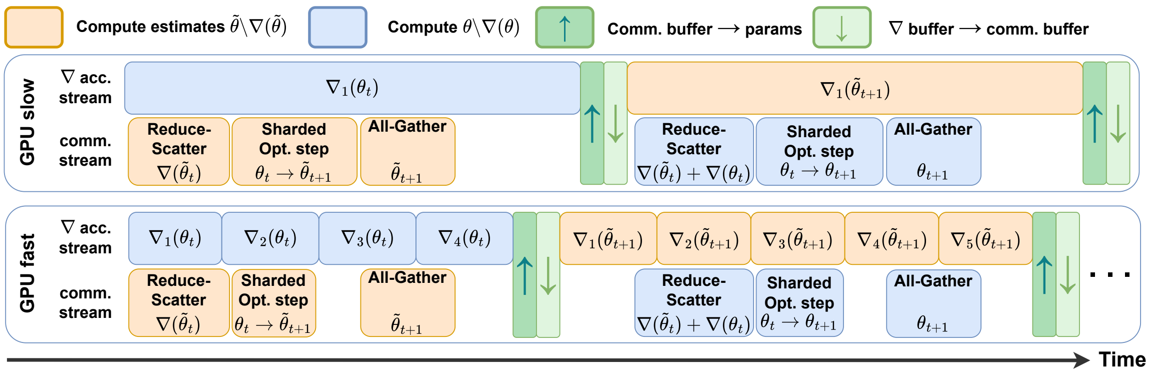

Compute estimates $\tilde{\theta}\backslash\nabla(\tilde{\theta})$
Compute $\theta\backslash\nabla(\theta)$
Comm. buffer ⟶ params
∇ buffer ⟶ comm. buffer
GPU slow
GPU fast
∇ acc. stream
comm. stream
$\nabla_1(\theta_t)$
$\nabla_1(\tilde{\theta}_{t+1})$
Reduce-Scatter $\nabla(\tilde{\theta}_t)$
Sharded Opt. step $\theta_t \to \tilde{\theta}_{t+1}$
All-Gather $\tilde{\theta}_{t+1}$
Reduce-Scatter $\nabla(\tilde{\theta}_t)+\nabla(\theta_t)$
Sharded Opt. step $\theta_t \to \theta_{t+1}$
All-Gather θ_{t+1}
$\nabla_2(\theta_t)$
$\nabla_3(\theta_t)$
$\nabla_4(\theta_t)$
$\nabla_2(\tilde{\theta}_{t+1})$
$\nabla_3(\tilde{\theta}_{t+1})$
$\nabla_4(\tilde{\theta}_{t+1})$
$\nabla_5(\tilde{\theta}_{t+1})$
Time

Figure 2: ACCO with a slow and a fast worker running in parallel, showing no idle time on both and hiding communications. The delayed update is compensated by splitting the mini-batch in two, leading to two steps in our timeline. The first uses half of the mini-batch to estimate ”next step” parameters, and the second uses the full mini-batch to update them.

2. Method

We describe our method, including the approach to compensate for the delayed update. The algorithm will be described from the point of view of each worker $i \in \{1, ..., N\}$.

Delayed Parameter Update. First, we explain the presence of a delay by re-purposing the ”Delayed Parameter Update” (DPU) [57] to fit in our framework. Contrary to the original DPU, we run gradient communications in the same stream as the optimizer step, in parallel to the gradient computations. To prevent GPU i from being idle at step t, gradients are accumulated over as many mini-batches $N_i^{(t)} \geq 1$ as necessary until the communication process finishes, which varies depending on the speed of the worker as shown in Fig. 2. Each worker i starts from the same neural network parameters $\theta^{(0)} \in \mathbb{R}^d$. $F : \mathbb{R}^d \to \mathbb{R}$ is the differentiable loss computed by our workers. A random mini-batch (modeled through the random variable $\xi \in \Xi$ following some law $\mathcal{P}$) is drawn from the local data shard $\mathcal{D}_i$ to initialize the stochastic gradient ${g_i}^{(-1)} = \nabla F(\theta^{(0)}, \xi_i^{(0)})$ and $N_i^{(-1)} = 1$. Then, for $t \in [\![0, T]\!]$ we repeat the following step, with the left and right sides running in parallel:

$$g_i^{(t)} = \sum_{k=1}^{N_i^{(t)}} \nabla F(\theta^{(t)}, \xi_{i,k}^{(t)}) \quad , \quad \theta^{(t+1)} = \texttt{Opt}\left(\theta^{(t)}, \frac{\sum_i g_i^{(t-1)}}{\sum_i {N_i}^{(t-1)}}\right) , \tag{DPU}$$

where $\texttt{Opt}$ is the optimizer of our choice (*e.g.* Adam or AdamW for LLM training). Note that the right side combines both the gradient averaging (communications) and the optimizer step, which runs in parallel to the gradient computations to the left. Remark that, except at the first step $t = 0$, the gradients used by $\texttt{Opt}$ are computed on parameters $\theta^{(t-1)}$ which differ from $\theta^{(t)}$, the ones we apply them to. This is inherently due to the parallel nature of our execution, and what we denote by "delayed update". We show in Sec. 3 that this has drastic impacts on the convergence in practice.

Toward ACCO. To counter this, we estimate what *would* be the parameters $\theta^{(t+2)}$ in addition to computing $\theta^{(t+1)}$. This allows the gradients at the next round to be computed on these estimates rather than the parameters of the last step. We denote this rule by "Weight Prediction" (WP). We initialize a common $\theta^{(0)}$, ${\tilde{g}_i}^{(0)} = \nabla F(\theta^{(0)}, \xi_i^{(0)})$, $N_i^{(0)} = 1$ and $\tilde{\theta}^{(1)} = \texttt{Est}(\bullet)$, where $\texttt{Est}$ is our estimation function that could take any argument at this point. This leads to the following:

$$\tilde{g}_i^{(t+1)} = \sum_{k=1}^{N_i^{(t+1)}} \nabla F(\tilde{\theta}^{(t+1)}, \xi_{i,k}^{(t+1)}) \, , \, \theta^{(t+1)} = \texttt{Opt}\left(\theta^{(t)}, \frac{\sum_i \tilde{g}_i^{(t)}}{\sum_i {N_i}^{(t)}}\right) \, , \, \tilde{\theta}^{(t+2)} = \texttt{Est}(\bullet) \, . \tag{WP}$$

Thanks to $\texttt{Est}$, the optimizer now applies to the parameters $\theta^{(t)}$ the gradients that were computed on an *estimated version* $\tilde{\theta}^{(t)}$, compensating the one-step delay. Akin to the idea of [6] to counter delays in pipelining, a simple estimation function could be to re-use the gradients just received and apply a second optimizer step, *i.e.* using $\tilde{\theta}^{(t+2)} = \texttt{Opt}\left(\theta^{(t+1)}, \frac{\sum_i \tilde{g}_i^{(t)}}{\sum_i {N_i}^{(t)}}\right)$. We investigate this method (denoted by ACCO-wp) in Sec. 3, but found that its training dynamic differs from the baseline, whereas ACCO, the algorithm we present next, perfectly matches it. The crux of ACCO is

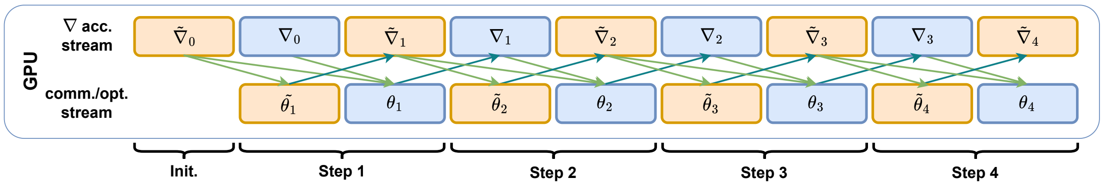


Figure 3: Illustration of ACCO's two-stage mechanism (1)-(2) to compensate the delayed updates.

to split the computation of the mini-batch gradients into two successive stages, where the first half of the mini-batch is used to estimate $\tilde{\theta}^{(t+1)}$ while $\theta^{(t+1)}$ is computed using the full mini-batch. This is motivated by the fact that gradient accumulation is often used to reach the extremely large batch sizes required to train LLMs [80], and if gradients are computed *sequentially* on a worker, we can leverage this to produce our estimate. Thus, starting with an initialized $\theta^{(0)}$, $\tilde{g}_i{}^{(0)} = \nabla F(\theta^{(0)}, \xi_i^{(0)})$ and $N_i^{(0)} = 1$, the two stages illustrated in Fig. 3 are (left and right side running in parallel):

$$g_i^{(t)} = \sum_{k=1}^{N_i^{(t)}} \nabla F(\theta^{(t)}, \xi_{i,k}^{(t)}) \quad , \quad \tilde{\theta}^{(t+1)} = \texttt{Opt}\left(\theta^{(t)}, \frac{\sum_i \tilde{g}_i^{(t)}}{\sum_i \tilde{N}_i{}^{(t)}}\right), \tag{1}$$

$$\tilde{g}_i{}^{(t+1)} = \sum_{k=1}^{\tilde{N}_i{}^{(t)}} \nabla F(\tilde{\theta}^{(t+1)}, \tilde{\xi}_{i,k}^{(t+1)}) \quad , \quad \theta^{(t+1)} = \texttt{Opt}\left(\theta^{(t)}, \frac{\sum_i g_i^{(t)} + \tilde{g}_i^{(t)}}{\sum_i N_i^{(t)} + \tilde{N}_i{}^{(t)}}\right). \tag{2}$$

We describe the different components of our two-stage mechanism as follows:

(1) The gradient computation stream uses the second half of the mini-batch to compute the gradients $g_i^{(t)}$ with respect to parameters $\theta^{(t)}$ while the communication stream estimates what would be the next steps parameters $\tilde{\theta}^{(t+1)}$ using the estimated gradients $\tilde{g}_i^{(t)}$.

(2) The computation stream uses the first half of the mini-batch to estimate what would be the gradients $\tilde{g}_i^{(t+1)}$ of the next parameters $\theta^{(t+1)}$ using estimated parameters $\tilde{\theta}^{(t+1)}$ while the communication stream computes $\theta^{(t+1)}$ using the full mini-batch. Note that it starts from the same version of the parameters $\theta^{(t)}$ as in step (1). The first half $\tilde{g}_i^{(t)}$ was estimated at step (2) of the *last round*, while the second half $g_i^{(t)}$ was just computed in (1).

3. Experiments

First we experiment with small language models on the TinyStories dataset [12] to demonstrate the impact of the delay on Transformers' convergence and the benefits of ACCO. Then, we confirm the performances of our method with larger models by pre-training on the OpenWebText dataset [18] and instruction fine-tuning on the Alpaca dataset [66]. Each distributed worker is hosted on a single GPU. Details on our experimental settings, ACCO's pseudo code and profiling of our implementation can be found in the Appendix, as well as experiment results on heterogeneous hardware.

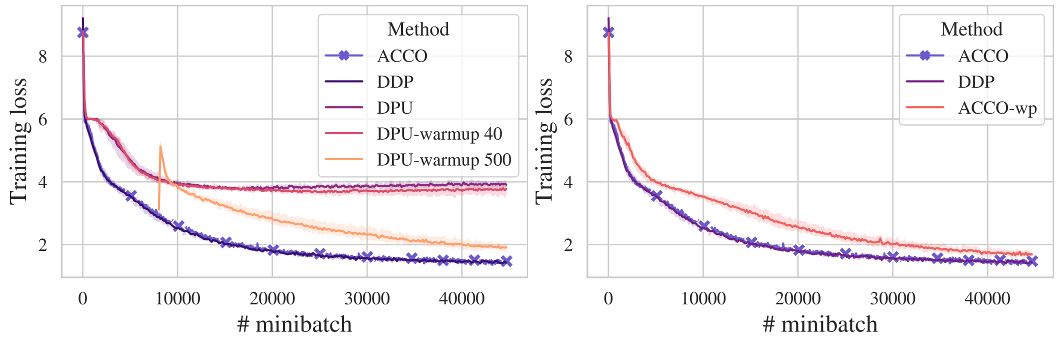


Figure 4: Comparison between DPU, ACCO and its Weight Prediction version on TinyStories.

Impact of delayed updates. We run three variants of DPU [57] as described in Sec. 2: **(1)** with no warmup, **(2)** with 40 warmup steps of non-delayed optimization step before switching to DPU (recommended recipe in [57]), and **(3)** with 500 steps of warmup. We report in Fig. 4 our training losses on 8 distributed workers averaged over 3 runs. We remark that using delayed updates can greatly hurt convergence, especially when no or too few warmup steps are performed. We also remark that, while ACCO perfectly matches the DDP baseline at all times, ACCO-wp displays worse behavior, especially at the beginning of the training.

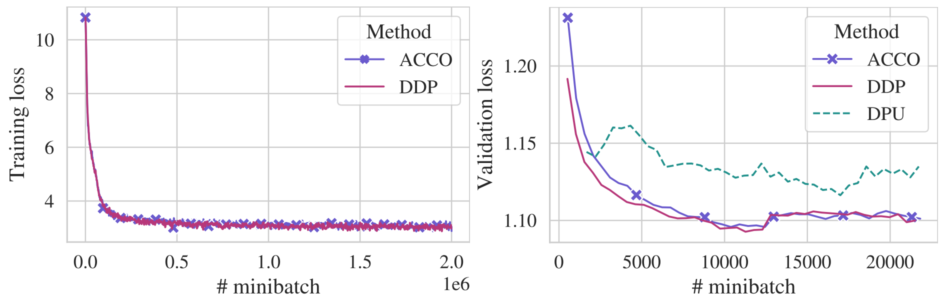


Figure 5: Loss for the pre-training task (left) and fine-tuning task (right) with larger models.

Passing the scaling test. We used the GPT-Neo architecture [5] with 125 million parameters and compared ACCO and DDP with 32 workers on a pre-training task for 50B tokens on the OpenWebText dataset [18]. We also fine-tuned on the Alpaca dataset [66] a GPT-Neo 2.7B model [5] pre-trained on the Pile dataset [17]. For that, we used two configurations: 8 A100-80G on a single node, and 8 A100-80G distributed equally across 2 nodes. We confirm in Fig. 5 that ACCO matches the training dynamic of the baseline, but Tab. 2 displays a significant speedup for our method.

Table 2: Pre-training and finetuning time speedup with ACCO against DDP on various setups.

Stage	Model	GPUs	#tokens	DDP	ACCO	(ΔT)
Pre-training	GPT-Neo-125M	1x8	6B	4h41min	4h25min	(-5.69%)
		4x8	50B	14h41min	10h55min	(-25.65%)
Finetuning	GPT-Neo-2.7B	1x8	80M	43min	25min	(-41.86%)
		2x4	80M	3h46min	29min	(-87.17%)

Conclusion

We propose ACCO, a novel algorithm that allows for parallel computation and communication of gradients while partitioning the optimizer states. Our two-stage mechanism compensates for the delayed update inherent to this parallel setting, ensuring consistent convergence dynamics with the standard optimization algorithm for large-scale distributed LLM training. We empirically display the benefits of our methods over several pre-training and finetuning tasks, reporting drastically reduced training times compared to our baseline, especially in multi-node settings or with heterogeneous devices.

References

[1] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL https://proceedings.neurips.cc/paper_files/paper/2011/file/f0e52b27a7a5d6a1a87373dffa53dbe5-Paper.pdf.

[2] Alex Andonian, Quentin Anthony, Stella Biderman, Sid Black, Preetham Gali, Leo Gao, Eric Hallahan, Josh Levy-Kramer, Connor Leahy, Lucas Nestler, Kip Parker, Michael Pieler, Jason Phang, Shivanshu Purohit, Hailey Schoelkopf, Dashiell Stander, Tri Songz, Curt Tigges, Benjamin Thérien, Phil Wang, and Samuel Weinbach. GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch, 9 2023. URL https://www.github.com/eleutherai/gpt-neox.

[3] By Mahmoud Assran, Arda Aytekin, Hamid Reza Feyzmahdavian, Mikael Johansson, and Michael G. Rabbat. Advances in asynchronous parallel and distributed optimization. *Proceedings of the IEEE*, 108(11):2013–2031, 2020. doi: 10.1109/JPROC.2020.3026619.

[4] Mahmoud Assran, Nicolas Loizou, Nicolas Ballas, and Mike Rabbat. Stochastic gradient push for distributed deep learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 344–353. PMLR, 09–15 Jun 2019.

[5] Sid Black, Gao Leo, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. URL https://doi.org/10.5281/zenodo.5297715.

[6] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform, 2019.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

[8] Xiangyi Chen, Xiaoyun Li, and P. Li. Toward communication efficient adaptive gradient method. *Proceedings of the 2020 ACM-IMS on Foundations of Data Science Conference*, 2020. URL https://api.semanticscholar.org/CorpusID:224805256.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[10] Michael Diskin, Alexey Bukhtiyarov, Max Ryabinin, Lucile Saulnier, Quentin Lhoest, Anton Sinitsin, Dmitry Popov, Dmitriy Pyrkin, Maxim Kashirin, Alexander Borzunov, Albert Villanova del Moral, Denis Mazur, Ilia Kobelev, Yacine Jernite, Thomas Wolf, and Gennady Pekhimenko. Distributed deep learning in open collaborations. In A. Beygelzimer, Y. Dauphin,

P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=FYHktcK-7v.

[11] Sanghamitra Dutta, Jianyu Wang, and Gauri Joshi. Slow and stale gradients can win the race. *IEEE Journal on Selected Areas in Information Theory*, 2(3):1012–1024, 2021. doi: 10.1109/JSAIT.2021.3103770.

[12] Ronen Eldan and Yuanzhi Li. Tinystories: How small can language models be and still speak coherent english?, 2023.

[13] Mathieu Even, Raphaël Berthier, Francis Bach, Nicolas Flammarion, Hadrien Hendrikx, Pierre Gaillard, Laurent Massoulié, and Adrien Taylor. A continuized view on nesterov acceleration for stochastic gradient descent and randomized gossip. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[14] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[15] Hamid Reza Feyzmahdavian and Mikael Johansson. Asynchronous iterations in optimization: New sequence results and sharper algorithmic guarantees. *Journal of Machine Learning Research*, 24(158):1–75, 2023. URL http://jmlr.org/papers/v24/22-0555.html.

[16] Louis Fournier and Edouard Oyallon. Cyclic data parallelism for efficient parallelism of deep neural networks, 2024.

[17] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[18] Aaron Gokaslan, Vanya Cohen, Ellie Pavlick, and Stefanie Tellex. Openwebtext corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[19] Eduard Gorbunov, Alexander Rogozin, Aleksandr Beznosikov, Darina Dvinskikh, and Alexander Gasnikov. *Recent Theoretical Advances in Decentralized Distributed Convex Optimization*, pages 253–325. Springer International Publishing, Cham, 2022. ISBN 978-3-031-00832-0. doi: 10.1007/978-3-031-00832-0_8.

[20] 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.

[21] Jörn-Henrik Jacobsen, Arnold W.M. Smeulders, and Edouard Oyallon. i-revnet: Deep invertible networks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=HJsjkMb0Z.

[22] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of bfloat16 for deep learning training, 2019.

[23] Sai Praneeth Karimireddy, Martin Jaggi, Satyen Kale, Mehryar Mohri, Sashank J. Reddi, Sebastian U. Stich, and Ananda Theertha Suresh. Mime: Mimicking centralized stochastic algorithms in federated learning. *ArXiv*, abs/2008.03606, 2020.

[24] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. torchgpipe: On-the-fly pipeline parallelism for training giant models, 2020.

[25] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, San Diega, CA, USA, 2015.

[26] Anastasia Koloskova, Sebastian U. Stich, and Martin Jaggi. Sharper convergence guarantees for asynchronous sgd for distributed and federated learning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9781713871088.

[27] Jakub Konecný, H. B. McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *ArXiv*, abs/1610.02527, 2016. URL https://api.semanticscholar.org/CorpusID:2549272.

[28] Atli Kosson, Vitaliy Chiley, Abhinav Venigalla, Joel Hestness, and Urs Köster. Pipelined backpropagation at scale: Training large models without batches, 2021.

[29] Dmitry Kovalev, Adil Salim, and Peter Richtarik. Optimal and practical algorithms for smooth and strongly convex decentralized optimization. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 18342–18352. Curran Associates, Inc., 2020.

[30] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12): 3005–3018, aug 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415530. URL https://doi.org/10.14778/3415478.3415530.

[31] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization for heterogeneous networks. In *ICML Workshop on Adaptive & Multitask Learning: Algorithms & Systems*, 2019. URL https://openreview.net/forum?id=SkgwE5Ss3N.

[32] Tao Lin, Sebastian U. Stich, Kumar Kshitij Patel, and Martin Jaggi. Don't use large mini-batches, use local sgd. In *International Conference on Learning Representations*, 2020.

[33] Yuliang Liu, Shenggui Li, Jiarui Fang, Yanjun Shao, Boyuan Yao, and Yang You. Colossal-auto: Unified automation of parallelization and activation checkpoint for large-scale models, 2023.

[34] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

[35] Karttikeya Mangalam, Haoqi Fan, Yanghao Li, Chao-Yuan Wu, Bo Xiong, Christoph Feichtenhofer, and Jitendra Malik. Reversible vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10830–10840, 2022.

[36] Artavazd Maranjyan, Mher Safaryan, and Peter Richtárik. Gradskip: Communication-accelerated local gradient methods with better computational complexity, 2022.

[37] Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, page 456–464, USA, 2010. Association for Computational Linguistics. ISBN 1932432655.

[38] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR, 20–22 Apr 2017. URL https://proceedings.mlr.press/v54/mcmahan17a.html.

[39] Konstantin Mishchenko, Francis Bach, Mathieu Even, and Blake Woodworth. Asynchronous SGD beats minibatch SGD under arbitrary delays. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=4XP0ZuQKXmV.

[40] Konstantin Mishchenko, Grigory Malinovsky, Sebastian Stich, and Peter Richtárik. Proxskip: Yes! local gradient steps provably lead to communication acceleration! finally! *arXiv preprint arXiv:2202.09357*, 2022.

[41] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony begets momentum, with an application to deep learning. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, page 997–1004. IEEE Press, 2016. doi: 10.1109/ALLERTON.2016.7852343. URL https://doi.org/10.1109/ALLERTON.2016.7852343.

[42] Adel Nabli and Edouard Oyallon. DADAO: Decoupled accelerated decentralized asynchronous optimization. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 25604–25626. PMLR, 23–29 Jul 2023.

[43] Adel Nabli, Eugene Belilovsky, and Edouard Oyallon. $\textbf{A}^2\textbf{CiD}^2$: Accelerating asynchronous communication in decentralized deep learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=YE04aRkeZb.

[44] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[45] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[46] John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Mike Rabbat, Mani Malek, and Dzmitry Huba. Federated learning with buffered asynchronous aggregation. In Gustau Camps-Valls, Francisco J. R. Ruiz, and Isabel Valera, editors, *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pages 3581–3607. PMLR, 28–30 Mar 2022. URL https://proceedings.mlr.press/v151/nguyen22b.html.

[47] Jose Javier Gonzalez Ortiz, Jonathan Frankle, Mike Rabbat, Ari Morcos, and Nicolas Ballas. Trade-offs of local sgd at scale: An empirical study. In *NeurIPS 2020 OptML Workshop*, 2021.

[48] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernandez. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1525–1534, Berlin, Germany, August 2016. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P16-1144.

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: an imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2019. Curran Associates Inc.

[50] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D. Sinclair. Computation vs. communication scaling for future transformers on future hardware, 2023.

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

[52] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

[53] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

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

[55] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3406703.

[56] Sashank J. Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and Hugh Brendan McMahan. Adaptive federated optimization. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=LkFG3lB13U5.

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

[58] Kevin Scaman, Francis Bach, Sébastien Bubeck, Yin Tat Lee, and Laurent Massoulié. Optimal algorithms for smooth and strongly convex distributed optimization in networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3027–3036. PMLR, 06–11 Aug 2017.

[59] Shuheng Shen, Linli Xu, Jingchang Liu, Xianfeng Liang, and Yifei Cheng. Faster distributed deep net training: computation and communication decoupled stochastic gradient descent. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, page 4582–4589. AAAI Press, 2019. ISBN 9780999241141.

[60] 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.

[61] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[62] Zhuoqing Song, Lei Shi, Shi Pu, and Ming Yan. Optimal gradient tracking for decentralized optimization. *Mathematical Programming*, Jul 2023. ISSN 1436-4646. doi: 10.1007/s10107-023-01997-7.

[63] Sebastian U. Stich. Local SGD converges fast and communicates little. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=S1g2JnRcFX.

[64] Sebastian U. Stich and Sai Praneeth Karimireddy. The error-feedback framework: better rates for sgd with delayed gradients and compressed updates. *Journal of Machine Learning Research*, 21(1), jan 2020. ISSN 1532-4435.

[65] 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. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ZO5cn4IfaN.

[66] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[67] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[69] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. Zero++: Extremely efficient collective communication for giant model training, 2023.

[70] Jianyu Wang, Hao Liang, and Gauri Joshi. Overlap local-sgd: An algorithmic approach to hide communication delays in distributed sgd. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, May 2020. doi: 10.1109/icassp40776.2020.9053834. URL http://dx.doi.org/10.1109/ICASSP40776.2020.9053834.

[71] Jianyu Wang, Vinayak Tantia, Nicolas Ballas, and Michael Rabbat. Slowmo: Improving communication-efficient distributed sgd with slow momentum. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SkxJ8REYPH.

[72] Blake Woodworth, Kumar Kshitij Patel, Sebastian Stich, Zhen Dai, Brian Bullins, Brendan Mcmahan, Ohad Shamir, and Nathan Srebro. Is local SGD better than minibatch SGD? In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 10334–10343. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/woodworth20a.html.

[73] Xuyang Wu, Sindri Magnusson, Hamid Reza Feyzmahdavian, and Mikael Johansson. Delay-adaptive step-sizes for asynchronous learning. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 24093–24113. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/wu22g.html.

[74] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Asynchronous federated optimization. In *NeurIPS 2020 OptML Workshop*, 2020.

[75] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training, 2020.

[76] Kun Yuan, Qing Ling, and Wotao Yin. On the convergence of decentralized gradient descent. *SIAM Journal on Optimization*, 26(3):1835–1854, 2016. doi: 10.1137/130943170.

[77] Jian Zhang and Ioannis Mitliagkas. Yellowfin and the art of momentum tuning. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 289–308, 2019. URL https://proceedings.mlsys.org/paper_files/paper/2019/file/b205b525b7ce002baae53228bab6d26b-Paper.pdf.

[78] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 685–693, Cambridge, MA, USA, 2015. MIT Press.

[79] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. Mics: Near-linear scaling for training gigantic model on public cloud, 2022.

[80] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2023.

[81] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023.

[82] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation. In *Proceedings of the*

34th International Conference on Machine Learning - Volume 70, ICML'17, page 4120–4129. JMLR.org, 2017.

[83] Fan Zhou and Guojing Cong. On the convergence properties of a k-step averaging stochastic gradient descent algorithm for nonconvex optimization. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 3219–3227. International Joint Conferences on Artificial Intelligence Organization, 7 2018. doi: 10.24963/ijcai.2018/447. URL https://doi.org/10.24963/ijcai.2018/447.

[84] Huiping Zhuang, Zhiping Lin, and Kar-Ann Toh. Accumulated decoupled learning: Mitigating gradient staleness in inter-layer model parallelization. *arXiv preprint arXiv:2012.03747*, 2020.

[85] Huiping Zhuang, Yi Wang, Qinglai Liu, and Zhiping Lin. Fully decoupled neural network learning using delayed gradients. *IEEE transactions on neural networks and learning systems*, 33(10):6013–6020, 2021.

[86] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. Parallelized stochastic gradient descent. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010. URL https://proceedings.neurips.cc/paper_files/paper/2010/file/abea47ba24142ed16b7d8fbf2c740e0d-Paper.pdf.

Appendix A. Related Work

Local optimization methods. Local optimization methods allow to perform several local model updates between periodic averaging. With the SGD optimizer, these algorithms predate the deep learning era [37, 86], and their convergence properties are still investigated nowadays [40, 63, 72, 83]. Due to their practical and efficient communication scheme, they have since been used for the Distributed Training of Deep Neural Networks (DNNs) with methods such as EASGD [78], SlowMo [71] or Post-local SGD [32, 47], and are ubiquitous in Federated Learning [27, 31, 38], broadening the choice of optimizers beyond SGD [8, 23, 56]. By overlapping communications over consecutive steps of local computations, they allow to hide communication bottlenecks, resulting in algorithms such as Overlap local-SGD [70], COCO-SGD [59] or CO2 [65]. Moreover, with heterogeneous hardware, they can adapt their local computation rate to their hardware capacity [10, 36]. However this comes at the price of additional memory requirements: due to their local nature, not only do these methods prevent the use of sharded optimizers such as ZeRO [52], but they also introduce additional control variables [40, 65, 71], hindering their scalability as shown in Tab. 1. Moreover, catering for heterogeneous hardware is not straightforward, as using different numbers of local updates leads to models shifting at different speeds, requiring extra care to counter this effect [36]. On the contrary, ACCO does not lead to such disparities: it just affects *how* the required batch size is reached.

Overlap decentralized optimization. The communication complexity being a core concern in decentralized optimization [19, 76], strategies have been devised to reduce communication overheads. For synchronous methods, works focus on designing algorithms with accelerated communication rates, leveraging Chebyshev polynomials [29, 58, 62]. For the asynchronous ones, they rely on the properties of the graph resistance [13, 42, 43]. Alternatively, some approaches overlap gradient and communication steps, either explicitly [4], or by modeling them with independent stochastic processes [42, 43]. However, none of these works focus on memory efficiency. Thus, they introduce additional variables and do not consider sharding the optimizer states. Moreover, they do not study optimizers other than SGD, and extending their beneficial properties to adaptive methods commonly used for DNN training such as Adam is still an ongoing research topic [3].

Memory-efficient distributed training of LLMs. The activation memory overhead required for training Transformers [68] can be mitigated for an extra computational cost by reconstructing the input with reversible architectures [21, 35], or recomputing the activations via checkpointing [7]. Efficient LLM training also combines parallelism methods. Classical data parallelism (DP) [9] suffers both from a high communication volume and a linear increase in memory due to the model replicas. ZeRO-DP [53] and Fully-Sharded DP [81] avoid this issue by sharding the model states (i.e., the optimizer states, gradients, and parameters) between workers. This comes at the cost of further increasing the synchronization between workers and the communication volume, which can be mitigated by compression [69], memory trade-offs [79], or delayed gradients [16]. The memory can be even more reduced using expensive CPU-GPU communications to unload states on the CPU [54, 57]. On the other hand, model parallelism partitions the DNN components for parallelization, either with tensor parallelism [60] by slicing a layer's computation on several workers, or with pipeline parallelism, which divides a model into sets of layers trained in parallel on mini-batch slices. Popularized by [20], this method leaves some workers idling and an inefficient memory overhead [14]. Allowing delay in the gradients avoids worker idleness [44, 84] but exacerbates the memory overhead, which can be partially mitigated with gradient accumulation [45, 85] and activation checkpointing [24, 33]. Combining these frameworks results in the effective 3D parallelism [61].

Delayed updates. Delays being intrinsic to distributed asynchronous optimization, there is a rich literature studying them. In the case of distributed SGD in a parameter server setting, while early analysis showed convergence rates depending on the *maximal* delay [1, 64], recent lines of work improved these dependencies [15, 26, 73], proving that asynchronous SGD beats standard mini-batch SGD even with unbounded delays [39]. However, they only study plain SGD, which is hardly used for DNN training. In this context, some work focused on the interplay between SGD with momentum and delays [41, 77], while delay compensation schemes such as re-scaling updates [74, 82] or buffering them [46] were proposed for Federated Learning. But still, they only study versions of SGD and not adaptive methods commonly used for LLMs trainingsuch as Adam [25] or AdamW [34]. Closer to our work, DPU was introduced as a memory-efficient way to train LLMs by running the optimizer on the CPU while gradients are computed on the GPU [57], inducing a one-step delay between the gradients computed and the corresponding optimizer step. To mitigate it, they advise starting training by warming up for several steps with a standard method with no delay. Perhaps surprisingly, we find in our experiments that this one-step delay has a noticeable influence on the convergence of LLMs training, even when using warmup steps. Contrary to DPU, we remove the need for them, with no impact on the convergence of our training. Moreover, as it is not its purpose, DPU still runs communications in the gradient computation stream, and is thus impacted both by the communication overhead of scaling and hardware heterogeneity. Finally, in pipeline parallelism, gradient delays also affect computation, and weight prediction methods have been proposed to mitigate the effect of staleness, by predicting the future weights using the optimizer's momentum [6]. More elaborate predictions have been proposed for SGD to further reduce the impact of the delay [28, 75].

Appendix B. Experimental Details and Further Results

B.1. Experimental setup

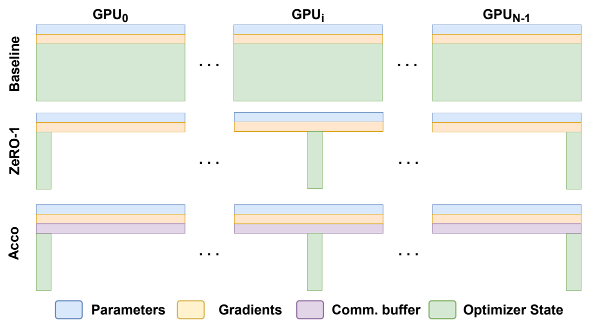


Figure 6: Memory requirements of ACCO vs DDP and ZeRO-1, see Tab.1 for quantitative details.

All of our experiments are performed on our local cluster of NVIDIA A100-80GB GPUs with 8 GPUs per node and an Omni-PAth interconnection network at 100 Gb/s for inter-node connections, intra-node connections being done with NVLink 300 GB/s. Each distributed worker is hosted on a single GPU. Our implementation is in Pytorch [49], and we verified that our code for ACCO does indeed produces two different CUDA streams running in parallel for the computations and communications using NVIDIA's Nsight System to profile it, as shown in Fig. 10. We trained all our models with AdamW [34], using mixed precision: our model parameters, gradient accumulation buffer, and communication buffers are in `bfloat16`

[22] while our sharded optimizer states are in single precision, as shown in Fig. 6. We compared our algorithm ACCO to several baselines in different settings, including Pytorch's Distributed Data Parallel (DDP) method [30] with ZeRO-1 [52].

B.2. Details on Fig. 1

In Fig. 1, we empirically motivate the need for methods mitigating communication overhead in Distributed Data Parallel (DDP) [30]. Our goal is to illustrate that the time spent communicating gradients can quickly trump the one used for computing them when using DDP to train LLMs. For that, we measure the time necessary to perform a forward and backward pass on a Llama-2 model [67] with 7B parameters hosted on a single GPU, using a batch size maxing out its memory. We compare this to the time necessary to compute an All-Reduce on those gradients with the NCCL backend, varying the number of distributed workers. We observe in Fig. 1 that when we communicate outside of a GPU node in our cluster, the time needed to average the gradients across workers can take more than *four times* the one spent on the whole forward and backward step. As DDP only partially hides communications during the backward [30], this means that our GPUs remain idle the majority of the time when we use more than 24 distributed workers, motivating the need for methods leveraging this time to compute instead.

B.3. Pre-training on TinyStories

We experiment with small language models on the TinyStories dataset [12], using the configuration available on the Huggingface Hub [1] and following the training hyper-parameters of their paper [12] to the best of our abilities. Hence, we use a 36M parameters GPT-Neo based [5] decoder-only transformer architecture. To match the 10k vocabulary they used, we trained our own BPE tokenizer on the TinyStories dataset. For our experiments, we used up to 8 workers on a single node.

B.4. Pre-training on OpenWebText

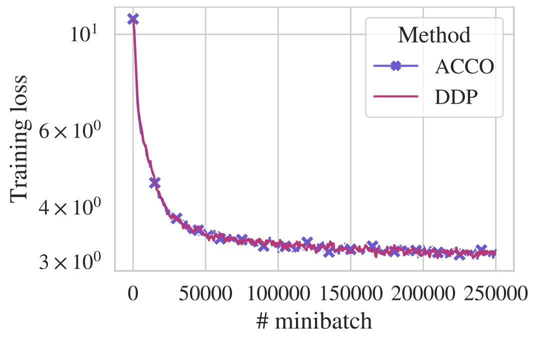
To assess how ACCO scales with larger models and more data, we pre-trained a model equivalent to GPT-2 [51] with both ACCO and DDP. Specifically, we used the GPT-Neo architecture [5] with 125 million parameters and the OpenWebText dataset [18], which contains 40 GB of text. We used the GPT-Neo tokenizer, pre-trained on the Pile dataset [17]. The models were trained on sequences of 1024 tokens, with documents concatenated using end-of-sequence tokens. The configuration used to instantiate the GPT-Neo 125M is available on the Huggingface Hub[2]. We only changed the "max_position_embeddings" parameter from 2048 to 1024. We used the OpenWebText dataset available on Huggingface[3]. To assess the impact of using different hardware, the experiment was repeated on 2 different clusters. The first was conducted on 8 H100-PCIe 80GB on a single node, and report results in Fig. 7. The

Table 3: Perplexity of our trained LLMs

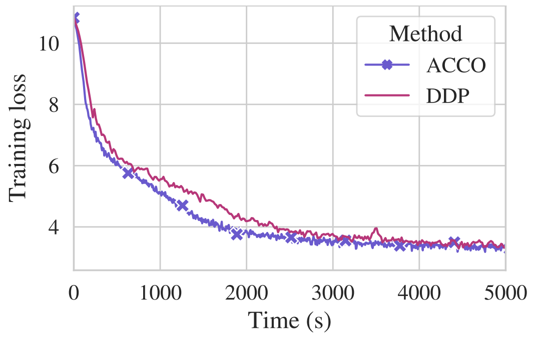
Method	LAMBADA (ppl ↓)	OpenWebText (ppl ↓)
ACCO 1x8	47.1	24.2
DDP 1x8	47.5	24.3
ACCO 4x8	45.5	22.5
DDP 4x8	44.1	21.7

1. Tiny Stories Available at: https://huggingface.co/datasets/roneneldan/TinyStories
2. GPT-neo 125M Configuration Available at: https://huggingface.co/EleutherAI/gpt-neo-125m/blob/main/config.json
3. OpenWebText Dataset Available at: https://huggingface.co/datasets/Skylion007/openwebtext

second was on 32 A100-80G GPU distributed on 4 nodes. We maxed out the memory of our GPUs with a local mini-batch size of 24. To reach a sufficiently large overall batch size, we used 1 step of gradient accumulation for DDP, and none for `ACCO` as our method naturally accumulates over 1 step, resulting for the first and second experiments in respectively 400K and 1.5M tokens per effective batch for both `ACCO` and DDP. In Tab. 2, we report additional experimental details, and notice that training with `ACCO` allows for significant time gains, which is additionally illustrated in Fig. 5. Moreover, to prevent GPUs from idling while waiting for communications, `ACCO` adaptively scheduled 315 supplementary accumulation steps over the whole training.



(*a*) Training loss during training on OpenWebText with 8 H100 GPUs and 6B tokens.



(*b*) Focus on the first part of the training with 32 A100-80GB GPUs w.r.t time.

Figure 7: Training curves for `ACCO` and DDP on OpenWebText.

Tab. 3 reports the perplexity of trained language models with both methods, which is a commonly used metric to evaluate pre-trained language models, as it quantifies the uncertainty of a model at predicting the next token. We evaluate the perplexity of language models on LAMBADA [48] and a test split of OpenWebText, and report similar results for both methods.

Table 4: Training hyperparameters for ACCO and DDP configurations.

Hyperparameter	**8 H100**	**32 A100**
mini-batch_size	24	24
n_grad_accumulation	ACCO: -DDP: 1	ACCO: -DDP: 1
sequence_len	1024	1024
#tokens_batch	400K	1.5M
optimizer	AdamW	AdamW
learning_rate	6e-4	6e-4
weight_decay	0.1	0.1
adam_beta1	0.9	0.9
adam_beta2	0.95	0.95
nb_steps_tot	50000	50000
scheduler	cosine	cosine
n_warmup_steps	0	0

B.5. Instruction Fine-Tuning

In previous sections, we compared `ACCO` against DDP in the pre-training stage. To further validate our algorithm, we additionally fine-tuned a pre-trained model on supervised instruction data. We consider the GPT-Neo 2.7B model [5] pre-trained on the Pile dataset [17] and finetuned it on the Alpaca dataset [66] containing 52k pairs of instruction/answer. We used the pre-trained GPT-neo 2.7B available on the Huggingface Hub[4] and the associated tokenizer. We used the Alpaca dataset available on Huggingface[5]. We fine-tuned the model using two configurations: 8 A100-80G on a single node, and 8 A100-80G distributed equally across 2 nodes. Samples are padded to match the longest sequence in the mini-batch. We fixed the mini-batch size at 4, leading to a total batch size of 128 for all methods. For DDP and DPU, we used a gradient accumulation of 4, while for `ACCO` , a gradient accumulation of 2 to account for the `ACCO` accumulation described in Sec. 2. The learning rate was set to 2×10^{-5} for all methods with a warmup of 50 steps, for DPU.

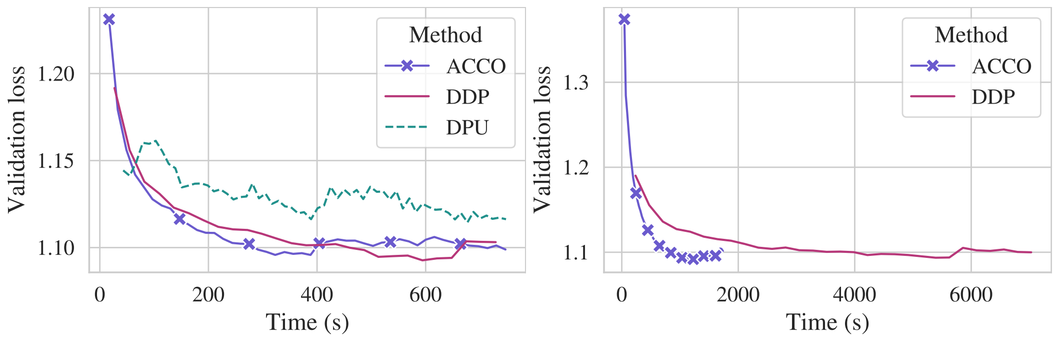


Figure 8: Validation curve with 8 workers on 1 node **(left)**, and 4 workers/node on 2 nodes **(right)**.

In this setting, padding to the longest sequence in the mini-batch induces more variability in the number of tokens per mini-batch. This results in more variability in the computational load for each worker, leading to increased wait times for synchronization. We observe in Fig. 8 that `ACCO` hits a lower validation loss faster than DDP on both 1 node and 2 nodes settings. Note that the difference between `ACCO` and DDP is accentuated when workers are distributed on multiple nodes. In 5, we observe that `ACCO` is less data efficient at the beginning of training, as evidenced by a higher loss compared to DDP for the same number of seen tokens. This is likely due to the fact that `ACCO` favors using tokens to increase the batch size to hide communication delays, meaning that fewer optimizer steps are performed per token compared to DDP. However, both algorithms converge to very similar loss values by the end of the training.

Appendix C. Experiment Using Heterogeneous Devices

To witness the impact of using heterogeneous devices, we run our algorithm `ACCO` and compared it to the DDP baseline in a four workers setting, with one of the GPU four times slower than the other three, as shown in Fig. 9. As we experiment on a cluster of A100 GPUs, we simulated the heterogeneity of the hardware by using the `time.sleep()` python command. First, we measured the time that a standard forward-backward step takes in our homogeneous cluster, and put to sleep

4. GPT-neo 2.7B Available at: https://huggingface.co/EleutherAI/gpt-neo-2.7B
5. Alpaca Dataset Available at: https://huggingface.co/datasets/tatsu-lab/alpaca

Table 5: Finetuning hyperparameters for ACCO, DDP and DPU configurations.

Hyperparameter	ACCO	DDP	DPU
mini_batch_size	4	4	4
n_grad_accumulation	2	4	4
total batch_size	128	128	128
optimizer	AdamW	AdamW	AdamW
learning_rate	2e-5	2e-5	2e-5
weight_decay	0.0	0.0	0.0
adam_beta1	0.9	0.9	0.9
adam_beta2	0.95	0.95	0.95
nb_steps_tot	50000	50000	50000
scheduler	cosine	cosine	cosine
n_warmup_steps	0	0	50

one of the four GPUs for three times this amount after each forward-backward pass. In this context, DDP is only as fast as the slowest worker, meaning that 3 of the 4 workers are idle a third of the time. With our method, the other workers accumulate during the time they are waiting for the slow one to finish. This means that ACCO allows to compute gradients for large batch sizes faster than standard baselines, resulting in faster convergence in terms of wall-clock time, as displayed in Fig. 9.

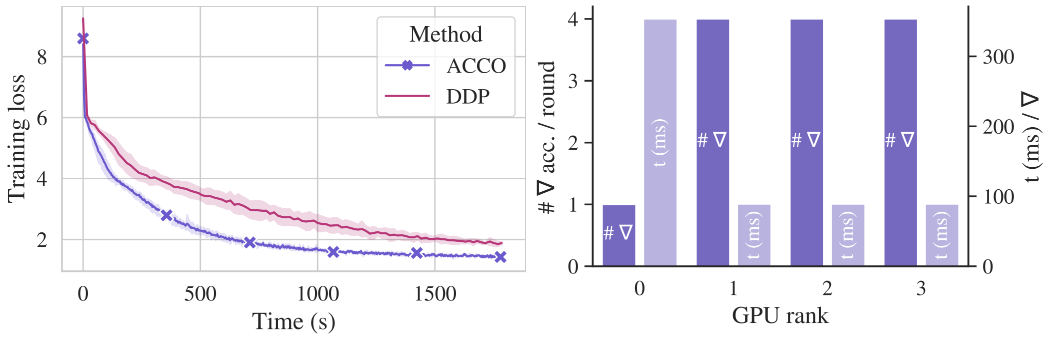


Figure 9: Training curves with 3 normal workers and 1 slow worker (4× slower).

Appendix D. Limitations

Experiments mainly on one cluster environment. Due to the lack of variety in the compute environments we have access to, the majority of our experiments were performed on a single cluster, described in Appendix B. This is a communication-constrained setting, as our hardware is not the most cutting-edge in that regard as discussed in Appendix B. This particularly flatters our method in comparison to DDP, as it accentuates the impact of the communication overhead in the wall clock time. However, to mitigate this one-sidedness, we also run a small pre-training study on one of the fastest hardware available today, and report in Tab. 2 that even in that case, ACCO leads to a 5% time gain.

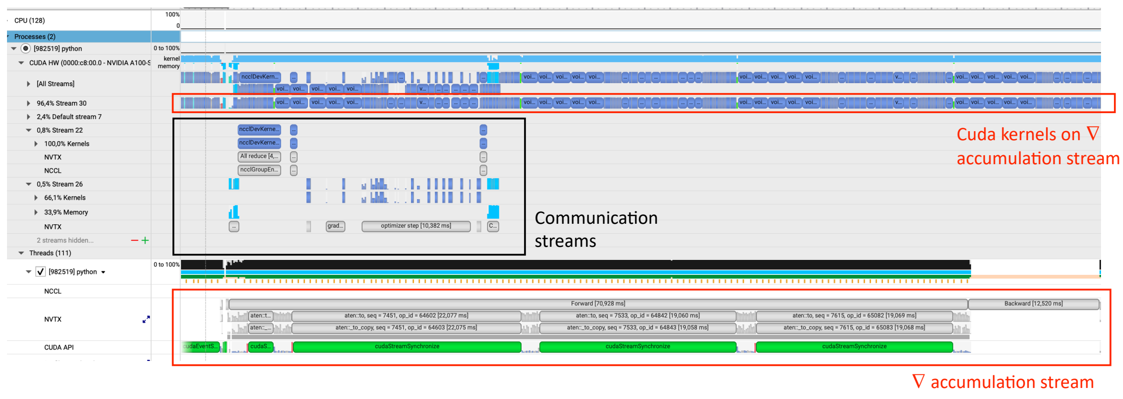
Communication cost only *hidden*, not reduced. While local optimization methods tackle the communication overhead problem with scarce communications, here we only hide them. Thus, our method does not lead to energy savings, nor question the cost of highly synchronized infrastructure. However, `ACCO` naturally maximizes the hardware throughput, allowing to reduce their use time.

Further memory savings avenue not explored. Due to the parallel nature of `ACCO`, removing the reliance on communication and gradient buffers seems hardly possible, questioning the feasibility of further memory savings if all executions are kept on the GPU. But, akin to ZeRO-Offload [57], the communication and optimizer stream could entirely be run on CPU, which would allow significant memory gains. We did not experiment with this idea, and let this consideration for future work.

Appendix E. Implementation Details

E.1. Profiling Results

E.2. Algorithm presentation



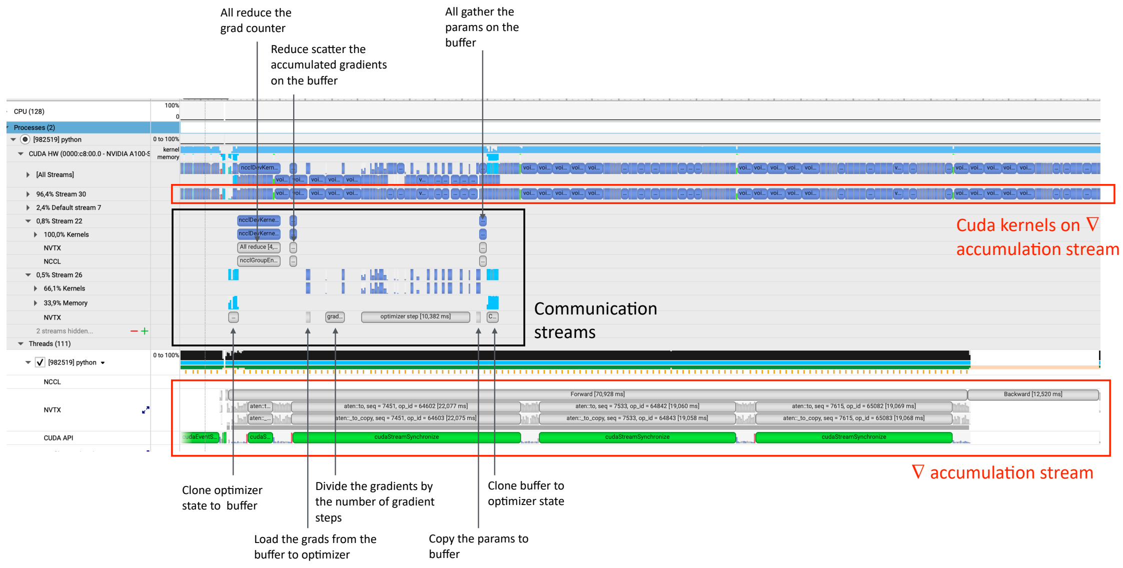

All reduce the grad counter
Reduce scatter the accumulated gradients on the buffer
All gather the params on the buffer
Cuda kernels on ∇ accumulation stream
Communication streams
∇ accumulation stream
Clone optimizer state to buffer
Divide the gradients by the number of gradient steps
Clone buffer to optimizer state
Load the grads from the buffer to optimizer
Copy the params to buffer

Figure 10: Nsight system profile of our implementation of `ACCO`: our two steams do run in parallel. In this Figure, the computation take more time than the communication because we only profiled small scale experiments with 8 workers, and small number of parameters (36M as we profiled on our TinyStories [12] setting). This changes when using larger models and more workers, as seen in Fig.1.

Algorithm 1 Training with ACCO in parallel for a worker i

```
1:  Input: Model with differentiable loss $F$, number of models $N$, initial parameters $\theta^{(0)}$, training steps $T$, dataset shards $\mathcal{D}_i$.
2:  Initialize: gradients $g_i^{(-1)} = \nabla F(\theta^{(0)}, \xi_i^{(0)})$ and number of gradients $N_i^{(-1)} = 1$
3:  # Computation CUDA stream
4:  while $t < T$ do
5:      Stage 1.
6:      while not Ready_for_Stage_2 do
7:          $\xi_i^{(t)} \leftarrow \mathcal{D}_i$
8:          $g_i^{(t)} \leftarrow g_i^{(t)} + \nabla F(\theta^{(t)}, \xi_i^{(t)})$
9:          $N_i^{(t)} \leftarrow N_i^{(t)} + 1$
10:     $\tilde{\theta}^{(t+1)} \leftarrow$ Buffer$_i$
11:     Buffer$_i \leftarrow (N_i^{(t)}, g_i^{(t)})$
12:     Stage 2.
13:     while not Ready_for_Stage_1 do
14:         $\xi_i^{(t)} \leftarrow \mathcal{D}_i$
15:         $\tilde{g}_i^{(t)} \leftarrow \tilde{g}_i^{(t)} + \nabla F(\tilde{\theta}^{(t+1)}, \xi_i^{(t)})$
16:         $\tilde{N}_i^{(t)} \leftarrow \tilde{N}_i^{(t)} + 1$
17:         $t \leftarrow t + 1$
18:     $\theta^{(t+1)} \leftarrow$ Buffer$_i$
19:     Buffer$_i \leftarrow (\tilde{N}_i^{(t)}, \tilde{g}_i^{(t)})$
20:
21: # Communication CUDA stream
22: while True do
23:     Stage 1.
24:     $(\tilde{N}_i^{(t)}, \tilde{g}_i^{(t)}) \leftarrow$ Buffer$_i$
25:     $\sum_i \tilde{N}_i^{(t)} \leftarrow$ All_Reduce$(\tilde{N}_i^{(t)})$
26:     Shard$_i\left(\sum_i g_i^{(t)}\right) \leftarrow$ Reduce_Scatter$(\tilde{g}_i^{(t)})$
27:     Shard$_i\left(\tilde{\theta}^{(t+1)}\right) \leftarrow$ ShardedOpt$\left(\text{Shard}_i\left(\theta^{(t)}\right), \text{Shard}_i\left(\frac{\sum_i \tilde{g}_i^{(t)}}{\sum_i \tilde{N}_i^{(t)}}\right)\right)$
28:     Buffer$_i \leftarrow$ All_Gather(Shard$_i\left(\tilde{\theta}^{(t+1)}\right)$)
29:     $N_i^{(t)} \leftarrow 0$
30:     Ready_for_Stage_2 $\leftarrow$ True
31:     Ready_for_Stage_1 $\leftarrow$ False
32:     Stage 2.
33:     $(N_i^{(t)}, g_i^{(t)}) \leftarrow$ Buffer$_i$
34:     $\sum_i N_i^{(t)} + \tilde{N}_i^{(t)} \leftarrow$ All_Reduce$(N_i^{(t)} + \sum_i \tilde{N}_i^{(t)})$
35:     Shard$_i\left(\sum_i g_i^{(t)} + \tilde{g}_i^{(t)}\right) \leftarrow$ Reduce_Scatter$(g_i^{(t)} + \sum_i \tilde{g}_i^{(t)})$
36:     Shard$_i\left(\theta^{(t+1)}\right) \leftarrow$ ShardedOpt$\left(\text{Shard}_i\left(\theta^{(t)}\right), \text{Shard}_i\left(\frac{\sum_i g_i^{(t)} + \tilde{g}_i^{(t)}}{\sum_i N_i^{(t)} + \tilde{N}_i^{(t)}}\right)\right)$
37:     Buffer$_i \leftarrow$ All_Gather(Shard$_i\left(\theta^{(t+1)}\right)$)
38:     $\tilde{N}_i^{(t)} \leftarrow 0$
39:     Ready_for_Stage_1 $\leftarrow$ True
40:     Ready_for_Stage_2 $\leftarrow$ False
```