Evaluating All-Reduce Operations in DDP Training with FABRIC: Framework for Architecture, Backend, and Reduction Interchange Comparison

Matilde Restelli*†, Ke Fan‡, Aadirupa Saha*, John K. Holmen†, and Sidharth Kumar*
*University of Illinois at Chicago, Chicago, IL, USA, {mrest, sidharth}@uic.edu, aadirupa.saha@gmail.com
†Oak Ridge National Laboratory, Oak Ridge, TN, USA, holmenjk@ornl.gov
‡Temple University, Philadelphia, PA, USA, ke.fan@temple.edu

***Abstract*—Distributed deep learning increasingly relies on large GPU systems, where Distributed Data Parallel (DDP) training replicates models across devices and synchronizes gradients before each optimizer update. As model sizes and GPU counts grow, gradient synchronization becomes a major training overhead. Although many optimized all-reduce algorithms have been proposed, they are often embedded in different communication libraries or stack layers, making controlled comparison in full training workflows difficult. We present FABRIC (Framework for Architecture, Backend, and Reduction Interchange Comparison), an open-source, portable, and extensible framework for evaluating PyTorch DDP across communication backends and collective strategies. Using DDP communication hooks, FABRIC integrates alternative all-reduce algorithms without modifying PyTorch internals or training code, enabling reproducible comparisons across models, datasets, scales, and backends including MPI, NCCL, and RCCL. FABRIC also enables researchers to add and benchmark custom communication methods. Experiments on Frontier and Polaris reveal meaningful performance differences among synchronization strategies, including hardware-topology effects. FABRIC is open-sourced at https://github.com/matirestelli/ddp-allreduce-eval-framework.**

I. Motivation and Challenges

Studying communication behavior in distributed training is important for understanding performance and scalability in real DDP workloads. Before this work, such studies were difficult to carry out in a fair and systematic way. Most evaluations relied on microbenchmarks, custom frameworks, or backend-specific implementations, since communication algorithms usually live inside backend libraries rather than being exposed through the training interface. This made it impractical to compare communication strategies, backends, and hardware platforms directly within standard PyTorch training.

We developed FABRIC to make this kind of research possible in end-to-end DDP workflows. FABRIC supports studies at the PyTorch/DDP layer [1], [2], including comparisons of standard backend all-reduce implementations, custom all-reduce methods through DDP hooks, and broader differences across communication backends and GPU platforms. It can therefore be used to compare backends such as MPI and NCCL, hardware such as AMD and NVIDIA GPUs, and communication methods under consistent conditions.

To enable this, we worked to make the framework portable across systems, backends, and GPU vendors, and to solve practical issues in PyTorch integration, software environments, and GPU-aware communication. We also wanted FABRIC to be productive to use, letting researchers swap communication algorithms, backends, and system configurations without touching PyTorch internals or training code, so new strategies can be added and benchmarked with little effort.

II. Framework

Experiments can be configured through either a Python interface or a batch/job script for supercomputing environments, as illustrated in Fig. 1. In both cases, the researcher selects the machine learning frontend (model, dataset, optimizer, and training parameters), the communication backend (MPI, NCCL, or RCCL), and the communication algorithm.

Communication algorithms are implemented as custom PyTorch DDP communication hooks [3]. During the backward pass, DDP groups gradients into buckets and normally issues an all-reduce on each one. When a hook is selected, it intercepts these gradient buckets and performs the corresponding all-reduce operation before execution continues, which is what allows alternative strategies to be substituted without modifying PyTorch internals or the training loop. The hooks currently implemented include ring and recursive doubling, both plain and with naive ZFP compression, collective-level online ZFP compression for ring and recursive doubling following Zhou et al. [4], and a hierarchical variant for NCCL and RCCL, adapted from their design, that reduces within a node before applying the same stream-pipelined ZFP compression across nodes. New strategies can be added by implementing additional hooks. Selecting `none` disables custom hooks and runs standard PyTorch DDP with the chosen backend, which serves as the baseline in our experiments.

III. Systems

To evaluate portability across GPU vendors, the framework was deployed on two contrasting leadership-class HPC systems: Frontier at the Oak Ridge Leadership Computing Facility [5] and Polaris at the Argonne Leadership Computing Facility [6]. Frontier represents the AMD ecosystem, with 9,856 compute nodes, each equipped with one AMD EPYC CPU and four AMD Instinct MI250X GPUs, whereas Polaris represents the NVIDIA ecosystem, with 560 compute nodes, each equipped with one AMD EPYC CPU and four NVIDIA

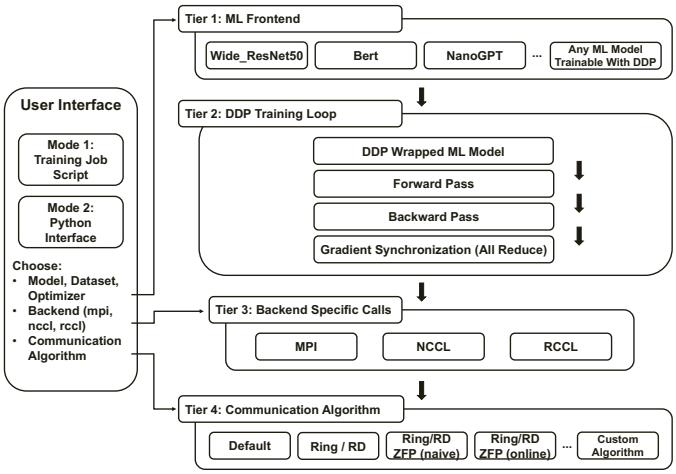


Fig. 1: Components and execution flow of FABRIC

A100 GPUs. Supporting the MPI backend on these platforms required building PyTorch from source and addressing the detection of GPU-aware MPI for Cray MPICH. Enabling ZFP-based [7], [8] compression within selected hooks further required integrating ZFP with the HIP and CUDA backends for Frontier and Polaris, respectively, together with stream-aware compression and decompression launches.

IV. Experiments

To ensure fair comparison, the DDP hyperparameters for each model were tuned once and then held fixed across all evaluated backends, communication algorithms, and platforms. Experiments were performed under both strong- and weak-scaling regimes. Strong scaling used a fixed global batch size, whereas weak scaling used a fixed local batch size per GPU. The primary performance metrics were epoch and iteration time, while model accuracy was recorded to verify correctness. To isolate communication overhead, the framework records hook work time and communication tail time.

V. Results

Using FABRIC, we identify system- and backend-dependent performance trends that warrant further study. Recursive doubling [9] is more favorable on Frontier, ring [10] on Polaris, and no single synchronization strategy is uniformly best across systems and scales.

The clearest case where a custom hook beats the baseline comes from the hierarchical ZFP online-compression algorithm on Polaris with the NCCL backend. The baseline is plain PyTorch DDP, which leaves gradient synchronization to the backend and runs whatever all-reduce it selects automatically. Across both strong and weak scaling, and at every GPU count we tested (8, 16, and 32), the hierarchical hook shortens per-epoch training time below this baseline. Since the all-reduce is the only difference between the two runs, the improvement comes from the communication strategy alone rather than any change in the model or training setup. This is exactly the kind of comparison FABRIC is built to make, one that would be hard to set up and measure fairly otherwise. Representative training-time results are shown in Fig. 2.

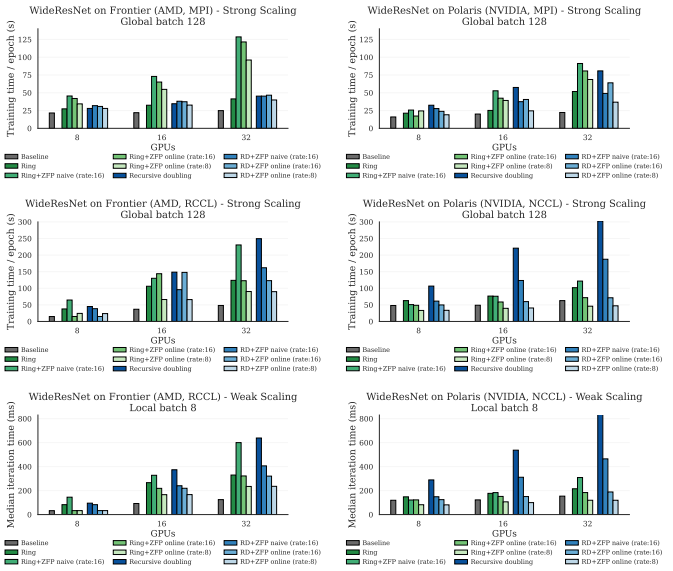


Fig. 2: WideResNet training performance across backends and scales. *Left:* Frontier (AMD). *Right:* Polaris (NVIDIA).

VI. Conclusion and Future Work

Our study shows that both the communication backend and algorithm substantially affect distributed training performance, and that this impact varies across systems, models, and scaling regimes. By exposing and measuring communication behavior, the framework helps identify bottlenecks and supports the design of improved custom communication algorithms. Future work will extend the evaluation to larger models and a broader set of communication algorithms, including approaches better optimized for NCCL and RCCL.

References

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS Autodiff Workshop*, 2017.

[2] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "PyTorch distributed: Experiences on accelerating data parallel training," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3005–3018, 2020.

[3] PyTorch, "DDP communication hooks documentation."

[4] Q. Zhou, B. Ramesh, A. Shafi, M. Abduljabbar, H. Subramoni, and D. K. Panda, "Accelerating MPI AllReduce communication with efficient GPU-based compression schemes on modern GPU clusters," in *ISC High Performance 2024*, pp. 1–12, 2024.

[5] S. Atchley *et al.*, "Frontier: Exploring Exascale The System Architecture of the First Exascale Supercomputer," in *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2023, doi: 10.1145/3581784.3607089.

[6] Argonne Leadership Computing Facility, "Polaris machine overview."

[7] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2674–2683, 2014.

[8] Lawrence Livermore National Laboratory, "zfp: Compressed floating-point and integer arrays."

[9] M. Ruefenacht, M. Bull, and S. Booth, "Generalisation of recursive doubling for AllReduce: Now with simulation," *Parallel Comput.*, vol. 69, pp. 24–44, 2017.

[10] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *J. Parallel Distrib. Comput.*, vol. 69, no. 2, pp. 117–124, 2009.