Where Did My NCCL Calls Go? A Profiler Comparison

Ruben Laso
Faculty of Computer Science
University of Vienna
Vienna, Austria
0000-0003-2574-4025

Majid Salimi Beni
Faculty of Informatics
TU Wien
Vienna, Austria
0000-0002-8634-7712

Ioannis Vardas
Faculty of Informatics
TU Wien
Vienna, Austria
0000-0001-5461-556X

Siegfried Benkner
Faculty of Computer Science
University of Vienna
Vienna, Austria
0000-0002-6520-2047

Sascha Hunold
Faculty of Informatics
TU Wien
Vienna, Austria
0000-0002-5280-3855

***Abstract*—Deep learning has become an increasingly important technology, with a profound impact on science and society, where modern models need to use several devices for training and inference. Communication among devices plays a crucial role in the performance of distributed deep learning applications, where NCCL is the most frequently used library for GPU-GPU communications. However, performance analysis tools for NCCL still lack maturity, with no clearly established alternative.**

In this paper, we compare several profiling tools for NCCL based on the accuracy and completeness of the collected data and the overhead introduced. We also introduce a NCCL profiler tool called `ncclsee` that overcomes the limitations of existing tools. Our experiments with micro-benchmarking and a real-world application show that existing profilers often miss a significant number of communication events, leading to inaccurate results. In contrast, `ncclsee` records all NCCL calls in every rank, offering more accurate measurements with negligible overhead.

***Index Terms*—NCCL, Collective Communication, Profiling, Tracing, Deep Learning, GPU-GPU Communications**

I. Introduction

The performance of training a Deep Learning (DL) model on a single GPU depends on the device's computational power. In Distributed Deep Learning (DDL), however, performance also relies heavily on the efficiency of communication between devices. NCCL (NVIDIA Collective Communications Library) serves as the communication backbone for DL frameworks, such as PyTorch and TensorFlow, providing collective operations, including AllReduce and AllGather.

Studying the communication performance in DDL is essential for identifying performance bottlenecks and optimizing the communication layer [1], [2]. Similar to MPI, where a range of profiling and tracing tools such as `mpisee` [3], Vampir [4], and Score-P [5] are available, there is a need for profiling and tracing tools for NCCL.

In NCCL 2.23.4, NVIDIA incorporated a profiler plugin interface that exposes timing information from inside the library. In its early versions, the main limitation was the lack of information on GPU kernel execution times, giving an incomplete picture of the communication performance. This gap was addressed with the profiling interface v4 (NCCL 2.27) by providing kernel runtime statistics. Having a profiling interface led to the development of profiling tools such as Inspector [6], and Google's Collective Communication Analyzer (CoMMA) [7]. Since NCCL 2.27, NVIDIA Nsight Systems (NSys) [8] also provided both tracing and profiling information and now supports more advanced features in recent versions. Despite recent progress, existing NCCL profiling tools still exhibit various limitations, including missing collective events and incomplete per-rank coverage, as shown in Fig. 1.

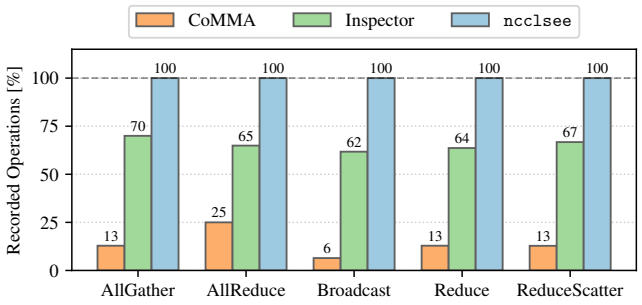


Fig. 1: Percentage of collective operations recorded by different profilers across 5 runs of `ncclbench` with several message sizes from 1 KiB to 1 GiB (`int32`). A value of 100 % means that all calls have been recorded across all processes. 2×4 GPUs on *Leonardo*.

In this paper, we perform an analysis of the existing NCCL profiling tools, focusing on the amount of collected data and the accuracy of the reported communication times. Additionally, we introduce `ncclsee`, a lightweight profiling tool that addresses some of the limitations found in existing tools, while providing more accurate measurements.

II. Background and related work

Performance analysis of communication libraries has been an active research area for years, especially in the context of MPI. With the adoption of GPUs for general-purpose computing, and more specifically distributed deep learning (DDL), profiling and tracing tools for GPU-GPU communication libraries, such as NCCL, have started to emerge.

Regarding MPI, several profilers and tools have been developed over the years. Most of these tools use the PMPI interface to intercept MPI calls and record the start and end times of each MPI operation, message sizes, communicators, etc. The most well-known MPI profilers include mpiP [9] and IPM [10]; however, these are not actively maintained anymore. Score-P [5] supports several programming models, including MPI and CUDA, with two modes of operation: profiling and tracing. Vampir [4] is a commercial visualization tool that supports traces in several formats (including those generated by Score-P), but operates on full traces, requiring large storage space. More recently, `mpisee` [3] has been proposed as a lightweight MPI profiler that focuses on communicator-centric profiling, providing fine-grained insights into MPI applications that use several communicators.

Despite being relatively new, NCCL and other GPU-GPU communication libraries (e.g., RCCL and MSCCL) [11] had a significant impact on AI workloads, gathering the attention of the research community. Nevertheless, the development of NCCL profilers and tracers is still in its early stages.

The tools we analyze in this paper are built on top of the NCCL Profiler Plugin API, introduced in NCCL 2.23. This API allows external plugins to receive callbacks for various communication events within NCCL, such as *Group*, *Collective*, *Kernel*, and *Proxy* events. Using this API, the profilers can assign timestamps to each event and infer the duration of NCCL operations. A major limitation until the introduction of the profiling interface v4 (NCCL 2.27) was the lack of information on GPU kernel execution times, crucial for understanding the performance of communication operations. Up to that point, communication cost was estimated based on CPU-side timestamps and the time spent in proxy operations. Having access to GPU kernel execution times allows for a more accurate measurement of the time spent in communication, as it captures the actual time taken by the GPU to execute the kernels associated with these operations.

In the NCCL repository, two examples of profilers are provided: Inspector and CoMMA. Additionally, NVIDIA Nsight Systems (NSys) also provides support for NCCL profiling.

a) Inspector: It is a tracing tool that uses the NCCL profiling interface v5 to generate a structured JSON output containing per-collective runtime metrics. Such metrics include information about the communicator, operation type, message size, data type, algorithm and protocol used, etc. Then the information is written to a file periodically, using a polling mechanism. However, traces might be incomplete if the polling interval is too large, as some operations might be overwritten before being logged.

b) CoMMA: Developed by Google, CoMMA can act as both a tracer and a profiler. It is built on top of the NCCL profiling interface v3 and uses proxy operations to estimate the duration of collective operations. However, this approach has important limitations. First, it does not capture GPU kernel execution times, affecting the accuracy of collected data. Second, proxy operations are used only for inter-node communications; using them in intra-node communication requires disabling some paths (e.g., P2P and SHM), leading to significant performance degradation. Also, in multi-node scenarios, only one GPU per node is responsible for proxy operations, resulting in incomplete traces.

c) NSys: Despite having basic NCCL profiling capabilities through NVTX annotations since version 2021.3, NSys only incorporated advanced NCCL tracing capabilities in version 2025.6.1, requiring NCCL 2.28. It captures system-wide traces, including CPU and GPU activities, and provides detailed information about NCCL operations, including kernel execution times. However, it generates large files, making it impractical for long-running DDL training sessions.

To the best of our knowledge, there is no prior work that analyzes and compares existing NCCL profilers and tracers. In this paper, our aim is to fill this gap by analyzing existing tools, focusing on the accuracy of the collected data. Furthermore, we introduce `ncclsee`, a lightweight profiling tool that addresses some limitations found in existing tools.

III. `ncclsee` Profiler

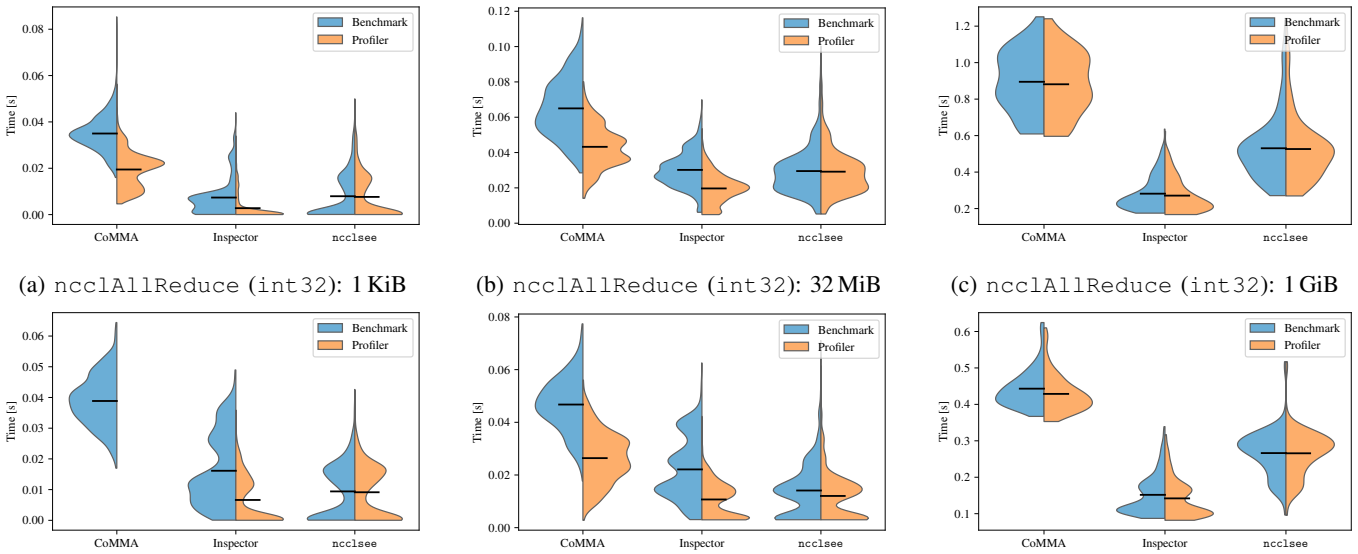
`ncclsee`[1] started on top of NCCL's profiling interface v3 [12], initially relying on CUPTI to gather GPU kernel execution times. With the introduction of profiling interface v4, which provides kernel runtime statistics, `ncclsee` was updated to leverage this new information, eliminating the need for CUPTI integration.

The main goal of `ncclsee` is to provide accurate profiling data with low overhead, focusing on not missing any event provided by the NCCL profiling interface. To achieve this, `ncclsee` implements a buffering mechanism that temporarily stores profiling data in memory before writing it to disk. The buffer size is adjustable via environment variables to control memory overhead, with the trade-off that smaller buffers imply more frequent I/O operations.

`ncclsee` supports three modes for data reporting: *profiling*, *tracing*, and *full*. In profiling mode, `ncclsee` aggregates data for each unique combination of communicator, collective operation, message size, and data type. This mode provides a high-level overview of the communication performance with minimal output size. In tracing mode, `ncclsee` reports the timings of each collective call in parseable JSON Trace Event Format[2], including details like number of ranks, algorithm, protocol, etc. This allows for a detailed analysis of the communication patterns at the cost of larger output files. In full mode, `ncclsee` will report the information in both profiling and tracing modes.

[1] Available at: https://github.com/parlab-tuwien/ncclsee

[2] Traces can be visualized using `chrome://tracing` or the Perfetto UI.



(a) `ncclAllReduce` (`int32`): 1 KiB (b) `ncclAllReduce` (`int32`): 32 MiB (c) `ncclAllReduce` (`int32`): 1 GiB

(d) `ncclAllGather` (`int32`): 1 KiB (e) `ncclBroadcast` (`int32`): 32 MiB (f) `ncclAllGather` (`int32`): 1 GiB

Fig. 2: Distributions of execution times reported by `ncclbench` and the profilers across 5 runs (200 calls per run), with black lines indicating the mean. 2×4 GPUs on *Leonardo* (non-contiguous nodes).

IV. Experimental Results

Our evaluation focuses on three NCCL profilers: Inspector, CoMMA, and `ncclsee`. For comparison, we run a set of micro-benchmarks and a real-world DL training workload. We analyze the accuracy of the collected data, the number of recorded events, and the overhead introduced by each profiler.

A. Experimental Setup

All experiments were performed on the *Leonardo* supercomputer, where each node consists of four NVIDIA A100 GPUs interconnected via 800 Gibit/s NVLink 3.0 and a 32-core Intel Xeon 8358 CPU. The nodes are interconnected through 200 Gibit/s InfiniBand. The software stack includes Open MPI 4.1.6, CUDA 12.2, and NCCL 2.28.9 (`dbc86fd`). The versions of the profilers we used are Inspector (`dbc86fd`), CoMMA (`1b2b374`), and `ncclsee` (`1fb9b07`).

B. Micro-benchmarks

Micro-benchmarks provide a controlled, close-to-ideal scenario (usually distant from real application execution) useful for evaluating profiler accuracy. We run a set of micro-benchmarks and compare the distribution of the communication times reported by the benchmark and the profilers, where more accurate profilers should report distributions closer to those of the benchmark.

Instead of using NCCL Tests, we developed a suite called `ncclbench`.[3] There are three key differences between them:

- Precision: NCCL Tests includes calls to `MPI_Barrier` within the timing region, distorting the results. In `ncclbench`, these are placed outside the timing region.
- Granularity: NCCL Tests records the time of the entire set of iterations and reports only the average time per call, while `ncclbench` reports the time of each individual call, allowing for more detailed analyses.
- Flexibility: `ncclbench` allows benchmarking both by number of iterations as in NCCL Tests, but also by time, executing the specified NCCL call repeatedly until a certain time budget (e.g., 3 seconds) is met.

We run `ncclbench` with eight GPUs across two non-contiguous nodes, testing the functions `ncclAllGather`, `ncclAllReduce`, `ncclBroadcast`, `ncclReduce`, and `ncclReduceScatter` with message sizes 1 KiB, 32 KiB, 1 MiB, 32 MiB, and 1 GiB using `int32`.

In Fig. 2, we present the distributions of execution times reported by `ncclbench` and the profilers for three representative cases: small (1 KiB), medium (32 MiB), and large (1 GiB) message sizes.

Overall, `ncclsee` provides the most accurate results, closely matching the measurements from the benchmark in all scenarios. Inspector, and to a larger extent CoMMA, tend to underestimate the execution times, especially for small and medium message sizes (Figs. 2a, 2b, 2d and 2e). For large message sizes (Figs. 2c and 2f), all profilers perform similarly well, with `ncclsee` still being the closest to the benchmark.

The inaccuracies by Inspector and CoMMA are likely due to missing events in their traces, that is, not all NCCL operations are recorded for all GPUs. As shown in Fig. 1, Inspector records 65 % to 70 % of the calls across all GPUs, usually missing several calls per process. In extreme cases, Inspector misses all calls for some processes, failing to produce an output file for those processes. CoMMA records from 6 % to 25 % of the events, as it has information only from a

[3] Available at: https://github.com/parlab-tuwien/lib-ncclbench

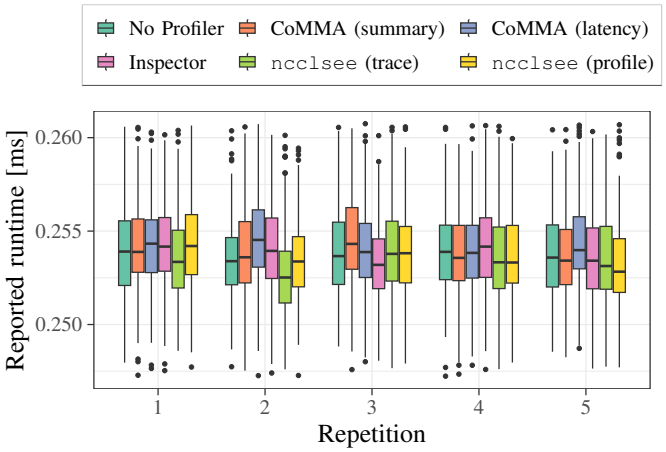


Fig. 3: Overhead comparison of various profiling/tracing plug-ins for NCCL. The y-axis shows the run-time distribution of `ncclAllReduce` as measured by `ncclbench` for 1 MiB of `int32`. 2×4 GPUs on *Leonardo* (contiguous nodes).

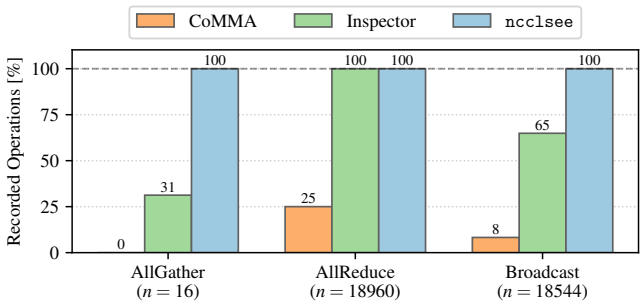
single process per node (due to relying on proxy operations). Notably, CoMMA reports all calls for `ncclAllReduce` for the processes it monitored in our experiments, but misses a significant number of calls for other collectives. In contrast, `ncclsee` records all the events across all tested scenarios.

C. Overhead Comparison

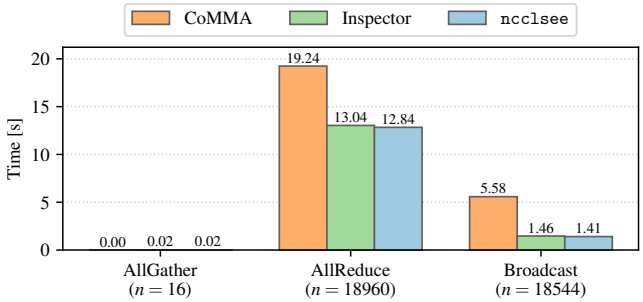
To evaluate the overhead introduced by each profiler, we repeated the micro-benchmark experiments described in Section IV-B on contiguous nodes of *Leonardo*. This setup minimizes the communication time, which also reduces variability in the measurements, allowing us to better observe the overhead introduced by the profilers. Fig. 3 shows the execution time distributions for `ncclAllReduce` with 1 MiB of `int32`. As observed, the communication times reported by `ncclbench` with and without profilers are very similar, differing in the range of microseconds, indicating a negligible overhead for all profilers.

D. Real-world DL Training

We further evaluate the profilers using a real-world DL training workload. We train the DenseNet121 model [13] for 5 epochs using PyTorch 2.0.0, in distributed data-parallel mode with 4×4 GPUs on *Leonardo* (non-contiguous nodes). Results are shown in Fig. 4. Regarding the number of recorded collective operations (Fig. 4a), the trends observed in the micro-benchmarks are confirmed. Most notably, `ncclsee` and Inspector report 100 % of the calls to `ncclAllReduce`, while CoMMA stays at 25 %. For other operations, Inspector and CoMMA miss a significant number of calls. In terms of communication time (Fig. 4b), both `ncclsee` and Inspector report similar times. However, in this scenario, CoMMA overestimates the communication time (24.82 s) to an extent that it exceeds the total training time (19.85 s).



(a) Percentage of collective operations recorded across all GPUs.



(b) Communication time reported.

Fig. 4: Results when profiling 5 epochs of DenseNet121 model. 4×4 GPUs on *Leonardo* (non-contiguous nodes).

V. Conclusions

In this paper, we presented a comparison of existing NCCL profiling tools, focusing on their accuracy, the number of recorded events, and the overhead they introduce. Our evaluation, conducted using micro-benchmarks and a real-world DL training workload, revealed that profilers based on the NCCL profiling interface can collect information with negligible overhead. However, we found that existing profilers such as Inspector and CoMMA often miss a significant number of calls, leading to inaccurate estimations of communication times. In contrast, our proposed tool, `ncclsee`, consistently captured all calls and provided accurate measurements.

Acknowledgments

We acknowledge EuroHPC Joint Undertaking for awarding us access to Leonardo hosted by CINECA, Italy.

Part of the software presented in this paper was developed using the Chameleon testbed supported by the National Science Foundation (US).

References

[1] Z. Hu, S. Shen, T. Bonato, S. Jeaugey, C. Alexander, E. Spada, J. Dinan, J. R. Hammond, and T. Hoefler, "Demystifying NCCL: an in-depth analysis of GPU communication protocols and algorithms," in *IEEE Symposium on High-Performance Interconnects, HOTI*. IEEE, 2025, pp. 48–59. [Online]. Available: https://doi.org/10.1109/HOTI66940.2025.00024

[2] M. Salimi Beni, R. Laso, B. Cosenza, S. Benkner, and S. Hunold, "Exploring NCCL tuning strategies for distributed deep learning," in *2025 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2025 - Workshops*. IEEE, 2025, pp. 59–62. [Online]. Available: https://doi.org/10.1109/IPDPSW66978.2025.00015

[3] I. Vardas, J. Larsson Träff, R. Laso, and S. Hunold, "Mpisee: Communicator-centric profiling of MPI applications," *Concurrency and Computation: Practice and Experience*, vol. 37, no. 15-17, p. e70158, 2025. [Online]. Available: https://doi.org/10.1002/cpe.70158

[4] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir performance analysis tool-set," in *Tools for High Performance Computing - Proc. 2nd International Workshop on Parallel Tools for High Performance Computing*. Springer, 2008, pp. 139–155. [Online]. Available: https://doi.org/10.1007/978-3-540-68564-7_9

[5] D. an Mey, S. Biersdorff, C. H. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. Shende, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A unified performance measurement system for petascale applications," in *Competence in High Performance Computing 2010 - Proceedings of an International Conference on Competence in High Performance Computing*. Springer, 2010, pp. 85–97. [Online]. Available: https://doi.org/10.1007/978-3-642-24025-6_8

[6] S. Das, J. Sewall, G. Congiu, P. Shamis, and G. Prasad, "Enhancing communication observability of AI workloads with NCCL Inspector," NVIDIA Technical Blog, 12 2025. [Online]. Available: https://developer.nvidia.com/blog/enhancing-communication-observability-of-ai-workloads-with-nccl-inspector/

[7] Google, "CoMMA: Collective communication analyzer," https://github.com/google/CoMMA, 2025, [Accessed: 2026-01-22].

[8] NVIDIA Corporation, "NVIDIA Nsight Systems," https://developer.nvidia.com/nsight-systems, 2026, [Accessed: 2026-01-22].

[9] J. Vetter and C. Chambreau, "mpiP: Lightweight, Scalable MPI Profiling," https://github.com/LLNL/mpiP, 2006.

[10] J. Borrill, J. Carter, L. Oliker, and D. Skinner, "Integrated performance monitoring of a cosmology application on leading HEC platforms," in *Proceedings of the 2005 International Conference on Parallel Processing, ICPP'05*. IEEE, 2005, pp. 119–128. [Online]. Available: https://doi.org/10.1109/ICPP.2005.47

[11] A. Weingram, Y. Li, H. Qi, D. Ng, L. Dai, and X. Lu, "xCCL: A survey of industry-led collective communication libraries for deep learning," *J. Comput. Sci. Technol.*, vol. 38, no. 1, pp. 166–195, 2023. [Online]. Available: https://doi.org/10.1007/S11390-023-2894-6

[12] I. Vardas, R. Laso Rodriguez, and M. Salimi Beni, "ncclsee: A lightweight profiling tool for NCCL," in *ASHPC25: Austrian-Slovenian HPC Meeting 2025*, 2025, p. 39. [Online]. Available: https://doi.org/10.34726/10426

[13] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. IEEE Computer Society, 2017, pp. 2261–2269. [Online]. Available: https://doi.org/10.1109/CVPR.2017.243