

PICO: Performance Insights for Collective Operations

Saverio Pasqualoni^{*,†}, Tommaso Bonato[‡], Lorenzo Piarulli^{*},
Torsten Hoefler[‡], Marco Canini[†], Daniele De Sensi^{*}

^{*}Sapienza University of Rome

[†]KAUST

[‡]ETH Zurich

{saverio.pasqualoni,marco}@kaust.edu.sa, {piarulli,desensi}@di.uniroma1.it, {tommaso.bonato,torsten.hoefler}@inf.ethz.ch

Abstract—Collective operations are cornerstones of both HPC applications and large-scale AI training and inference, yet benchmarking them in a systematic and reproducible way remains difficult on modern systems due to the complexity of their hardware and software stacks. Existing suites primarily report end-to-end timings and offer limited support for controlled algorithm and configuration selection, fine-grained profiling, and capturing the runtime environment. We present PICO (Performance Insights for Collective Operations), an open-source framework that decouples portable experiment setup from platform execution, provides a backend-adaptive parameter selection interface across MPI and NCCL, supplies plain-MPI reference collective implementations, optionally instrumentable, and records the system configuration for reproducible comparisons. Evaluated on three major supercomputers, PICO shows that default collective algorithms and transport settings can be up to $5\times$ slower than the best available choice. It provides diagnostic evidence by isolating topology sensitive algorithmic choices and, through instrumentation, reveals detailed algorithmic breakdowns. To assess end-to-end effects of benchmark-informed tuning and evaluate application-level impacts, we replay open-source LLM training traces in ATLAHS simulator with optimized collective profiles identified by PICO, achieving reductions in training times of up to 44%.

Index Terms—High performance computing, Performance analysis, Computer networks, Message passing, Software Tools

I. INTRODUCTION

While recent High Performance Computing (HPC) systems have surpassed exascale performance [1, 2], the growing disparity between the available computational resources and data movement capabilities remains a critical obstacle. Processor performance continues to improve at a faster pace than memory and interconnect technologies, increasing the relative cost of communication. As HPC systems scale to ever-larger sizes, the efficiency of data transfers increasingly determines both application performance and overall system scalability [3, 4, 5].

Collective operations, being among the most communication-sensitive components of distributed-memory applications, are particularly affected by this trend and face increasing performance challenges on large-scale HPC clusters [6]. Together with point-to-point communication, they form the backbone of traditional HPC workloads as well as AI training and inference tasks [7]. At large scale, the cost of collective communication often dominates application runtime [8, 7, 9], making the design and optimization of efficient collective algorithms a key priority [10, 11, 12, 13, 14, 15].

Understanding and optimizing collective communication performance is difficult because collectives intertwine computation, memory movement, and network transfer, making it non-trivial to attribute bottlenecks to a specific subsystem [16, 17]. This attribution problem is amplified by today’s heterogeneous environments: systems combine scale-up and scale-out fabrics with diverse topologies, multiple user-level communication libraries (e.g., MPI implementations and *CCL [18, 19, 20, 7, 21]), and evolving network software stacks and APIs such as OFI and UCX/UCC [22, 23, 24]. Moreover, performance is shaped by time-varying runtime conditions, such as congestion [25], load-balancing [26] allocation policies [27], and task-to-node mappings [28], which introduce variance that is hard to predict, reproduce, or control.

Crucially, each collective operation can be implemented by multiple different algorithms, each one optimal for different combinations of node count and message sizes [29, 21]: libraries often select among their available implementations with predetermined general heuristics. Moreover, performance gaps between different communication libraries, namely here GPU-aware MPI and *CCL libraries, can invert depending on runtime conditions, with one library outperforming the other depending on message size and node count [21]. As a result, fair evaluation and reproducible characterization of collective algorithms require benchmarking methods that capture both system context and algorithm-level behavior.

Existing benchmarking tools such as OMB [30], NCCL Tests [31], Intel IMB [32], ReproMPI [33], and CommBench [34] are effective for reporting end-to-end collective performance, but they only partially address the needs of modern systems. They generally lack fine-grained phase/step profiling and do not support straightforward, controlled comparisons of similar algorithms across different libraries. They also do not systematically record experimental conditions, such as node allocations, environment variables, software stack versions, and relevant hardware configuration, which complicates rigorous, reproducible evaluation.

For these reasons, we introduce **PICO (Performance Insights for Collective Operations)**, an open-source¹, modular, and extensible framework for benchmarking and diagnosing the performance of collective operations across multiple communication libraries. PICO contributes with (i) step-level

¹<https://github.com/HLC-Lab/pico>

instrumentation to attribute time to algorithmic phases, (ii) metadata-rich run capture to enable reproducibility and regression diagnosis, and (iii) backend-neutral reference collectives to isolate algorithmic differences from backend effects in a (iv) unified experiment specification spanning diverse backends (e.g., MPI and NCCL) for portable benchmarking.

We use PICO to analyze collective performance on different supercomputers: LUMI [35], Leonardo [36] and MareNostrum 5 [37]. We find that default collective algorithm selections can be 30–40% slower than the best available alternative, and in the worst case deliver only $0.2\times$ of optimal performance. We leverage PICO’s comprehensive metadata capture to support a post-mortem analysis and identify root causes of diverging scaling behaviors of similar Broadcast algorithms, with the slower variant exhibiting a $2.5\times$ slowdown. Using PICO’s fine-grained instrumentation, we localize performance losses to different algorithmic phases, separating network-limited regimes from reduction and memory-movement limits.

Finally, to quantify end-to-end application impact, we replay open-source traces from LLaMA 7B and Mistral MoE using the ATLAHS simulator [38] with optimized collective profiles informed by PICO, achieving projected runtime reductions of up to 44%.

II. MOTIVATION AND REQUIREMENTS

Systematic, fair, and reproducible measurement of collective operations performance across modern HPC systems is increasingly difficult due to the tight coupling of the structured heterogeneity of hardware environments and the growing complexity of software stacks. The main challenges faced are:

C1 On the hardware side, modern platforms are typically built from multi-GPU nodes connected by high-bandwidth *scale-up* fabrics (e.g., NVLink [39], InfinityFabric [40], UALink [41], UnifiedBuffer [42], or Scale-Up Ethernet (SUE) [43]), whose bandwidth can exceed the *scale-out* interconnect by up to an order of magnitude [21]. At the same time, the size of scale-up domains is rapidly expanding (e.g., up to 72 GPUs in NVL72 [44] and even larger domains up to 384 GPUs [42]). GPUs across nodes communicate over scale-out networks such as InfiniBand [45], Slingshot [46], or Ultra-Ethernet [47], frequently deployed with tapered topologies (e.g., Dragonfly/Dragonfly+ [48, 35, 49, 46, 36, 50] or tapered fat-trees [37, 51, 52]). Thus effective bandwidth and latency depends on whether communicating endpoints share the same scale-up domain and, for scale-out, whether they are within the same local switch domain or traverse oversubscribed global links [21, 46]. Such non-uniform communication costs violate the homogeneous-link assumptions behind many traditional collective designs and motivate hierarchical and topology-aware collectives that explicitly exploit intra-node vs inter-node structure [53, 12, 54, 55].

C2 On the software side, as illustrated in Fig. 1, collective execution spans multiple layers (MPI implementations and *CCL libraries, as well as network stacks such as OFI/libfabric and UCX [24, 23]), each with tunable parameters and evolving behavior across versions. End-to-end timings include costs

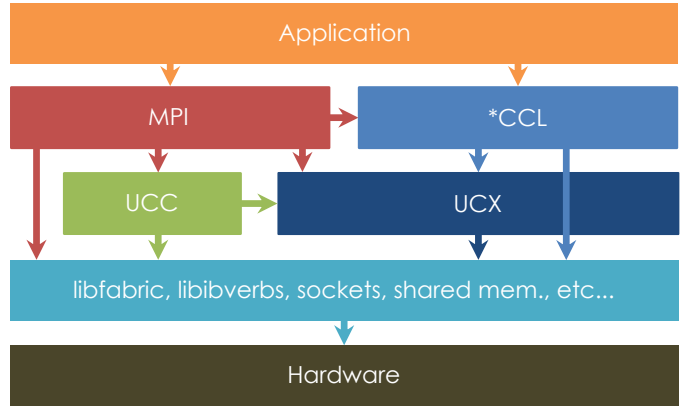


Fig. 1. Simplified software stack hierarchy. Applications interface with high-level communication libraries (MPI, *CCL), while the high-level library typically relies on middleware layers, such as UCX or libfabric to access transport layer interfaces. Legacy versions of MPI and some *CCL libraries access directly the transport layer interfaces without relying on middleware APIs.

from network transfer, memory staging/movement, and reduction/computation, each dependent on one or more layer of the software stack. Moreover, results are sensitive to time-varying runtime conditions (e.g., congestion [25], allocation policies [27], and task-to-node mappings [28]) as well as version/parameter changes across layers, which complicates reproducibility and regression diagnosis.

C3 Beyond system heterogeneity, *measurement methodology* itself is a source of systematic biases in collective benchmarking [56]: precise process synchronization methods are required to ensure accurate results, but achieving this precision is a non-trivial task. A common approach is the use of barriers, but such constructs don’t ensure that every single process enters the measured portion of the code at the same time: a process may exit the barrier before others, distorting measured runtimes [57]. In this regard the choice of barrier algorithm is important: some algorithms cause more skewing than others, with linear approaches (e.g. ring) being the worst due to their long propagation delay. An alternative approach to the use of barriers is represented by window-based schemes, where processes agree on a future start time. Those methods can reduce some barrier-induced artifacts, but they shift the problem to clock synchronization and drift [56, 57].

A. State-of-the-Art Analysis

Standard benchmark suites, including OSU-MicroBenchmark (OMB) [30], Intel MPI Benchmark (IMB) [32], and NCCL Tests [31], are effective at reporting end-to-end latency/bandwidth, but they are not designed for controlled, reproducible diagnosis. In particular, they provide limited support for (i) portable algorithm selection and cross-library baselining (often relying on manual environment-variable configuration), (ii) per step/phase measurements beyond aggregate timings, and (iii) lack mechanisms to record and store system’s information and ensure reproducibility of the experiments.

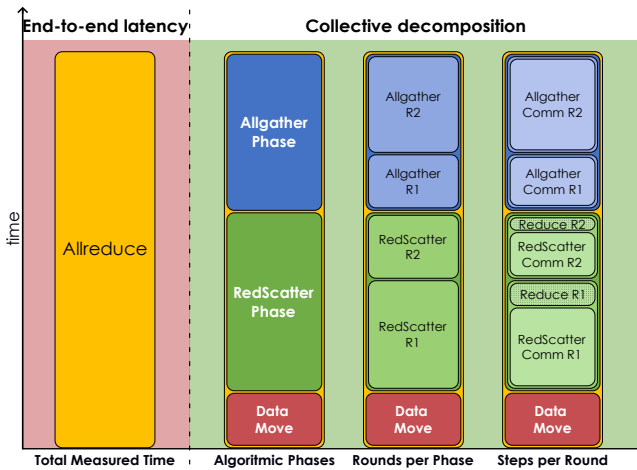


Fig. 2. End-to-end benchmarks report a single latency for a collective (left), but execution decomposes into phases, rounds, and steps that mix communication, reduction/computation, and data movement (right). PICO targets this gap by enabling optional phase/step attribution and controlled baselines.

This growing need for *communication observability* is also reflected in industry efforts. NVIDIA recently introduced NCCL Inspector [58], a profiler-plugin that provides low-overhead, always-on, per-communicator and per-collective performance and metadata logging during real distributed AI workload runs, explicitly to help diagnose issues such as congestion and to correlate dips in compute throughput with collective performance. While valuable, such tooling is library-specific (NCCL-focused) and targets in-workload monitoring rather than portable, controlled benchmarking and backend-neutral baselining across stacks.

ReproMPI [33] is a micro-benchmarking framework whose primary goal is correctness of measurements. It offers multiple synchronization methods and provides reference implementations of different collective algorithms but unfortunately it does not track the state of the system nor it aims to provide cross-stack compatibility and per step/phase measurements.

CommBench [34] takes a significant step toward portability by introducing a library-agnostic API spanning MPI and *CCL collectives, but it does not target the same diagnostic and reproducibility objectives: it does not provide fine-grained observability of collective behaviours, metadata logging to track the state of the system is relatively minimal, and its usage model often requires writing low-level benchmarking logic, increasing manual effort for exploratory evaluation [34].

Netgauge [59] introduces a modular benchmarking design that decouples benchmark “patterns” from communication “modules,” enabling extensible, comparable network/protocol measurements; however it is not designed for collective communications benchmarking.

Table I summarizes this comparison, distinguishing capabilities that are natively supported as part of a tool’s integrated workflow (✓) from those that are achievable only through external scripting, or ad hoc modifications (⊖) and those that are not supported at all (×).

TABLE I
QUALITATIVE COVERAGE OF REQUIREMENTS (SEC. II-B).
✓: BUILT-IN; ⊖: PARTIAL/MANUAL; ×: NOT TARGETED.

	OMB	IMB	NCCL-T/I	CommBench	NetGauge	ReproMPI	PICO
R1 Fine grained profiling	⊖	×	✓	×	×	⊖	✓
R2 Backend-neutral references	×	×	×	×	×	⊖	✓
R3 Portable spec & control	⊖	⊖	×	✓	✓	⊖	✓
R4 Automation & amm. usability	⊖	⊖	⊖	✓	✓	⊖	✓
R5 Metadata-rich reproducibility	×	×	×	×	⊖	⊖	✓
R6 Extensibility across stacks	⊖	×	×	✓	⊖	×	✓

B. Design requirements

The central challenge in modern collective benchmarking is moving from end-to-end reporting to *diagnosis*: controlled experiments that can explain *why* a collective is underperforming and *which* algorithmic step or subsystem dominates. Fig. 2 illustrates this gap: many microbenchmarks typically report a single end-to-end latency, while the collective’s execution decomposes into phases, rounds, and steps (where a round is an iteration within a phase, and a step is a sub-operation within a round) that mix communication, reduction/computation, and data movement. Accordingly, our primary objective is *fine-grained profiling* (R1), enabled by *backend-neutral, instrumentable baselines* (R2) and a *portable experiment specification* (R3). The remaining requirements (R4–R6) ensure reproducibility and practicality at scale.

R1 The framework must support fine-grained profiling at the granularity of algorithm phases, rounds and steps, enabling attribution of time to network transfer, memory movement/staging, and reduction/computation where applicable. Collective implementations must therefore be able to delineate regions of interest via explicit annotations that map measurements to semantically meaningful regions of the collective. Instrumentation must be *optional* and impose no measurable overhead, within experimental noise, when disabled.

R2 To enable controlled comparisons and meaningful attribution, the framework must include *backend-neutral reference collectives* (e.g., plain-MPI implementations ported from major libraries) that isolate algorithmic differences from backend/transport effects (Sec. IV-B) and can be instrumented at fine boundaries, providing consistent portable references for profiling without requiring modifications to implementation-specific internal code.

R3 The framework must provide a portable, declarative specification for experiments (e.g., collective type, message sizes, scale, algorithm choice, and relevant backend parameters) so that the same experiment can be executed and compared across platforms and stacks with minimal platform-specific modifications. The specification must serve as a stable control interface that (i) selects among internal algorithm choices exposed by a given stack and (ii) exposes a set of relevant configuration

parameters (Sec. IV-A), enabling controlled baselining across libraries without the need for specific per-experiment scripts.

R4 The framework must support large-scale benchmarking campaigns while providing structured bookkeeping and post-processing. It must front-load complexity into an infrequent platform-setup step (e.g., module/environment configuration, backend discovery, scheduler integration), after which experiments can be executed from a portable specification. The system must automatically apply requested algorithm/knob settings, submit and manage jobs, collect results, and produce outputs in a standardized schema that support systematic comparison across runs, enabling recurring workflows such as tuning studies and regression checks.

R5 Each run must capture sufficient metadata to reproduce and audit results and to diagnose regressions, including software stack versions (and build identifiers), selected backends/transports, relevant environment variables, hardware details (e.g., GPU/NIC model), and allocation/mapping context (e.g., node list and rank placement). Without this context, performance differences across runs, hardware or software updates are difficult to interpret. To keep large campaigns practical, the framework should support configurable metadata verbosity so that per-test metadata volume can be reduced when only minimal context is needed.

R6 The framework must accommodate evolving communication stacks by providing a clear extension interface for adding new backends (MPI implementations or *CCL libraries) and new collectives/algorithms: extensions to the framework should preserve a consistent end-to-end workflow. Moreover when a backend does not support a feature (e.g., algorithm selection or exposure to network layer parameters), the framework should degrade gracefully and still execute the experiment with a well-defined subset of functionality, without requiring all backends to implement the same sets of features.

C. Workflows and Usability

We selected these requirements to address the needs of three types of users. First, *researchers* and *collective algorithm developers* require backend-neutral baselines and fine-grained instrumentation capabilities (R1–R3) to directly evaluate algorithmic differences and optimizations while ensuring reproducibility (R5). Second, *application developers and users* require automated exploration and structured result management to make tuning campaigns feasible and repeatable (R4), together with sufficient metadata to interpret variability across allocations and software stacks (R5–R6). Third, *system administrators* require a repeatable workflow for regression testing across upgrades and configuration changes (R4–R5).

III. ARCHITECTURE

PICO’s core design principle is to decouple *what to run* (a portable experiment description) from *how to run it* on a given platform (reusable environment descriptors), while producing outputs that are both comparable and diagnosable. The resulting end-to-end workflow, as shown in Fig. 3, follows

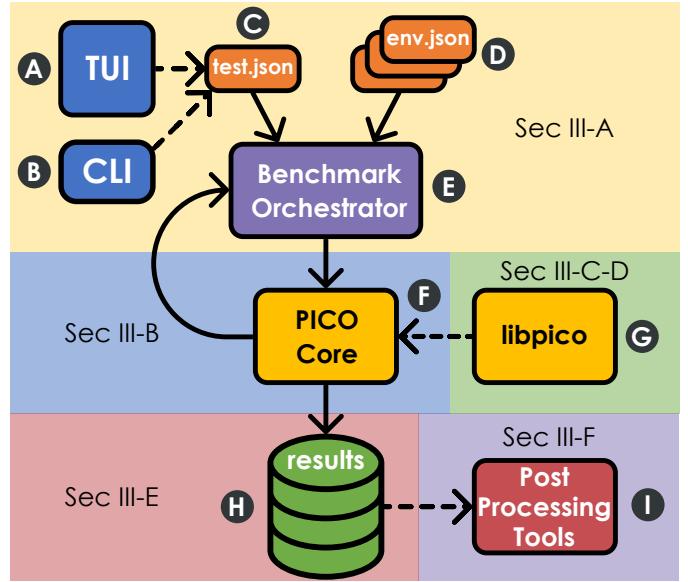


Fig. 3. High-level PICO workflow. Users define experiments via a portable test descriptor (`test.json`) paired with a platform environment descriptor (`env.json`). The benchmark orchestrator resolves the descriptors, builds and launches `pico_core` (and optional `libpico` baselines), and collects standardized results and metadata for post-processing and diagnosis.

a simple pipeline: test and environment specifications are defined via descriptive files (C and D), an orchestrator script (E) sets up the environment according to the test description and launches instances of a benchmarking core program (F) responsible for executing test instances and store performance data and metadata (H) in a structured format. Post processing and visualization tools (I) are used to analyze the data itself. The benchmarking core can optionally use a library (G) containing reference implementations and instrumentation primitives.

A. Experiment specification and control plane

PICO’s control plane provides a stable, portable interface for defining experiments and expressing backend control (algorithm choice and relevant parameters) through declarative descriptors. The framework translates this intent to backend-specific mechanisms without requiring per-experiment scripting. A platform descriptor (`env.json`; Fig. 3D) records platform-specific capabilities and control mappings, while a portable test descriptor (`test.json`; Fig. 3C) records experiment intent. Together, they realize R3 by making experiments executable and comparable across platforms with minimal intervention, and they enable R4 by allowing the orchestrator (Fig. 3E) to execute large campaigns directly from descriptors.

To improve usability, set up configuration complexity is front-loaded into the infrequent creation of `env.json` descriptors. Those files define the local environment: available communication stacks (MPI implementations and selected *CCL libraries), module/environment setup, scheduler and launcher templates (e.g., SLURM defaults), and backend-

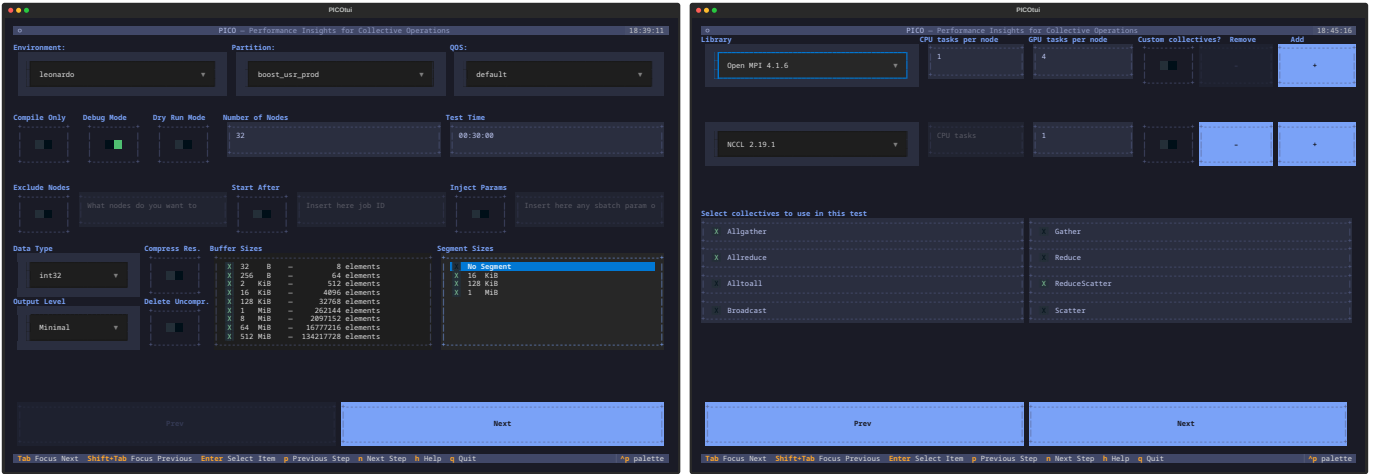


Fig. 4. PICO terminal user interface (TUI) for interactive experiment specification. The TUI exposes available backends, collectives, algorithms, and supported control parameters for the current platform descriptor, and produces validated `test.json` files that can be executed by the orchestrator.

specific control mappings (e.g., which algorithm selectors and transport knobs are exposed and how to apply them). Given this platform context, users define a `test.json` that specifies the experiment in a backend-agnostic form (collective, message sizes, scale, and requested algorithm/parameter settings). Crucially, `test.json` does not encode cluster-dependent scripts; instead it encodes *control intent* (e.g., “use algorithm X”, “set parameter Y”) that PICO resolves using `env.json`.

PICO provides both a terminal user interface (TUI; Fig. 3A) and a command-line interface (CLI; Fig. 3B) as front-ends to the same specification model. PICO’s TUI, shown in Fig. 4, is designed to guide the user into the discovery of available libraries and parameters: it presents what controls are available for the selected backend, applies defaults and validation, and outputs a self-contained `test.json` with the desired test configuration.

B. Execution engine and backend adapters

PICO’s execution engine, PICO core (Fig. 3F), runs on compute nodes inside the allocated job. PICO core is responsible for the timing-critical portion of benchmarking: initializing the communication context, applying requested controls when supported, executing the target collective(s) over the specified message sizes and scales, and emitting measurements and metadata in the standardized output format, using PICO’s internal barrier synchronization for timing alignment [16].

PICO supports heterogeneous communication stacks (R_6) through a uniform backend adapter interface, with backend availability selected at compile time (e.g., `#ifdef NCCL/CUDA` to support for GPU collectives). Each compiled-in adapter implements: (i) context initialization, (ii) mapping of abstract controls from `test.json` to backend-specific knobs when exposed, and (iii) collective execution and timing.

C. Backend-neutral baselines

`libpico` (Fig. 3G) is a user-space library that provides reference implementations of collective algorithms. In the

current version, `libpico` focuses on MPI: it provides plain-MPI implementations (built on point-to-point primitives and adapted from Open MPI and MPICH) so algorithmic choices can be evaluated without, necessarily, relying on library internal collectives. This enables controlled, backend-independent comparisons of algorithmic behavior under identical experimental conditions. Sec. IV-B will demonstrate the practical value of stable reference baseline to isolate algorithmic effects.

Importantly, `libpico` was designed to be extensible and allow developer to write and test directly new algorithms. It can be extended to support other communication libraries by implementing the corresponding backend signature and registering the implementation within `pico_core`.

```

1  int allreduce(const void *sbuf, void *rbuf, int count,
2              MPI_Datatype dt, MPI_Op op, MPI_Comm comm) {
3      PICO_TAG_BEGIN("init:mem-move");
4      /* staging, temporary buffers allocations and copies */
5      PICO_TAG_END("init:mem-move");
6
7      PICO_TAG_BEGIN("phase:redscat");
8      for (int step = 0; step < steps; step++) {
9          PICO_TAG_BEGIN("redscat:comm", step);
10         err = MPI_Sendrecv(...);
11         PICO_TAG_END("redscat:comm", step);
12
13         PICO_TAG_BEGIN("redscat:reduction", step);
14         err = MPI_Reduce_local(...);
15         PICO_TAG_END("redscat:reduction", step);
16     }
17     PICO_TAG_END("phase:redscat");
18
19     PICO_TAG_BEGIN("phase:allgather");
20     for (int step = steps-1; step >= 0; step--) {
21         PICO_TAG_BEGIN("allgather:comm", steps-1-step);
22         err = MPI_Sendrecv(...);
23         PICO_TAG_END("allgather:comm", steps-1-step);
24     }
25     PICO_TAG_END("phase:allgather");
26     return MPI_SUCCESS;
27 }

```

Fig. 5. Pseudo-code fragment illustrating tag-based fine-grained attribution in an instrumented Allreduce implementation: nested `PICO_TAG_BEGIN/END` markers annotate memory movement, algorithm phases, and per-step operations for fine-grained timing breakdowns.

TABLE II
RESULT DATA GRANULARITY MODES SUPPORTED BY PICO.

Mode	Description
Full Statistics	Stores all measurements for each rank and each iteration. For each iteration, stores aggregated statistics across ranks.
Minimal Summary	Records only the maximum value per iteration. Stores a single set of statistical aggregates over the iterations for the test point.
None	Only <code>stdout</code> output with no values stored.

D. Tag-based instrumentation for fine-grained attribution

To move beyond aggregate end-to-end timings, PICO supports optional, tag-based instrumentation (*R1*) for collectives implemented in `libpico` (including user-defined collectives). Tags delineate semantically meaningful regions of an implementation, such as data staging, algorithmic phases, and per-step communication/reduction, enabling fine-grained attribution of where time is spent. Because instrumentation lives in `libpico`, PICO provides this breakdown without modifying vendor communication stacks.

Instrumentation is expressed through lightweight macros (`PICO_TAG_BEGIN(...)` and `PICO_TAG_END(...)`), which can be used either flat or nested to capture hierarchical structure. The probes are optional, user-controlled, and inserted only at selected regions of interest. When enabled, PICO records timings for tagged regions and emits them using the same structured output model as other measurements; when disabled, the tag macros compile out to empty statements, preserving the behavior of standard benchmarking and leaving end-to-end tuning sweeps unaffected. The added cost per timing invocation was measured to be negligible (less than 100 ns per tagged region). Fig. 5 illustrates the instrumentation of an Allreduce, in particular: (i) a memory initialization region (lines 3–5), (ii) the two phase structure with Reduce-Scatter and Allgather of the algorithm (lines 7–17 and 19–25), and (iii) per-step regions inside each phase denoting communication and reductions (lines 9–11, 13–15, and 21–23).

E. Standardized results and metadata capture

To satisfy *R5*, PICO emits performance measurements and execution context in a standardized, human-readable output suitable for both large campaigns and post hoc diagnosis. Each campaign stores per-test measurements under a run directory, snapshots the resolved experiment specification (including the *effective* control settings applied to the backend), and maintains a lightweight index to support automated traversal, aggregation, and comparison across runs.

Each *test point* (collective type, message size, scale, backend, and control settings) is a separate record containing also timing data and identifiers; the schema is backend-agnostic and encodes both the *requested* configuration (from `test.json`) and the *effective* configuration after platform resolution (via `env.json`), preserving comparability even when controls are

unsupported or mapped differently across stacks. To balance diagnostic depth and campaign scale, PICO supports configurable result granularity.

In addition, PICO records run context alongside performance data, including software stack versions/build identifiers, selected backends/transport, relevant environment variables and tuning knobs, hardware characteristics (e.g., GPU/NIC model), and allocation/mapping context (e.g., node list and rank placement). Metadata capture supports configurable verbosity so users can retain minimal context for broad sweeps while enabling richer capture for focused diagnostic runs.

F. Analysis and diagnosis toolkit

To help interpret collective performance on tapered, non-uniform interconnects, PICO provides a lightweight network traffic tracer that estimates how traffic is distributed across the cluster’s different topology domains (e.g., Dragonfly groups). The tracer takes as input (i) allocation and rank-placement metadata captured per run (e.g., node list and rank mapping; *R5*) and (ii) a topology description for the target system (e.g., node to switch-group membership and link hierarchy). It separates pairs of communicating ranks into categories based on their physical allocation (e.g., intra-node, intra-switch and inter-group) and returns an estimate of the utilization of network links by different collective algorithms. These estimates enable users to correlate observed performance with expected link congestion when comparing algorithms that trade local aggregation for reduced global traffic, or when diagnosing performance shifts caused by placement changes or topology-aware transport settings. It provides a topology-level estimate only, not a packet-accurate simulation of congestion, adaptive routing, or protocol behavior.

For convenience and amortized usability (*R4*), PICO provides scripts that generate standard plots directly from the result schema, including heatmaps (e.g., message size vs. scale), line plots, and box/bar summaries across algorithms or backends. Because plots are derived from the same indexed schema used for campaign execution, visualization remains consistent across runs and can be integrated into automated tuning and regression pipelines.

IV. EVALUATION

This section evaluates PICO through a set of case studies designed to validate the requirements in Section II and to demonstrate practical utility beyond end-to-end benchmarking. We organize the evaluation around four questions.

- 1) How often does the library’s algorithm selection deviate from the best-performing choice for a given message size and process topology, and can PICO orchestrate controlled tuning campaigns? (Sec. IV-A)
- 2) Can `libpico` reference implementations isolate algorithmic effects and explain cross-platform differences? (Sec. IV-B)
- 3) Can a more detailed instrumentation of a collective algorithm reveal actionable bottlenecks that are opaque under aggregate timings? (Sec. IV-C)

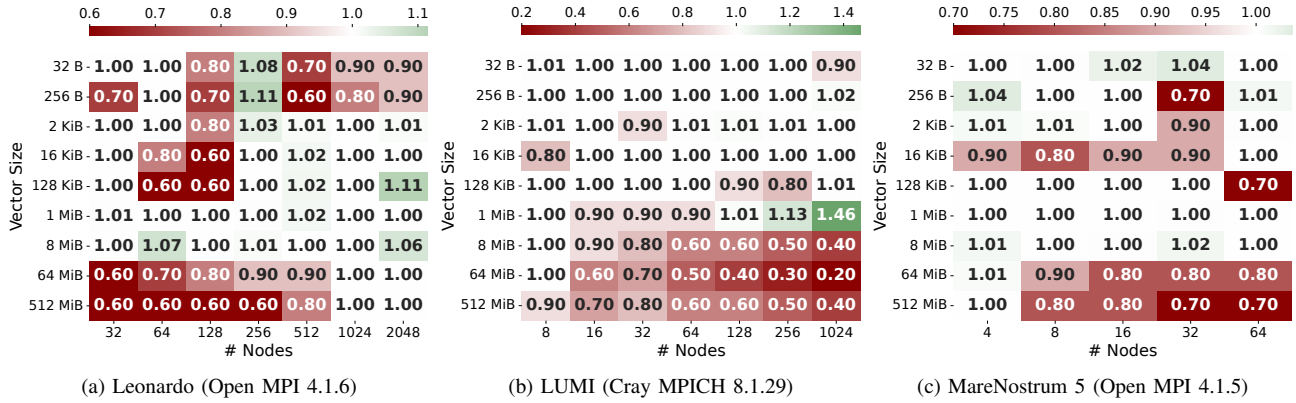


Fig. 6. Median best-to-default *latency ratio* r over the algorithm choices exposed by the communication library. In particular $r = \frac{t_{best}}{t_{def}}$ where t_{def} is the algorithm automatically selected by the library and t_{best} is the best performing algorithm not selected by default. Values $\tilde{r} < 1$ indicate suboptimal choices.

4) Do benchmark-informed choices translate into application-level implications under realistic communication traces? (Sec. IV-D)

Each case study shows what PICO makes possible *because* it was designed around the design goals described in Sec. II-B.

A. Collective Tuning

Using PICO we conducted systematic sweeps of `MPI_Allreduce` across three major European supercomputers, Leonardo (Open MPI 4.1.6), LUMI (Cray MPICH 8.1.29), and MareNostrum 5 (Open MPI 4.1.5), varying only the collective algorithm choice exposed by each communication stack while keeping all other settings fixed. Fig. 6 summarizes the outcome of these sweeps using the *best-to-default latency ratio* $r = \frac{t_{best}}{t_{def}}$, where t_{def} is the median runtime obtained under the backend default algorithm and t_{best} is the minimum median runtime among the *non-default* algorithms exposed by the backend for the same test point (message size, scale, and stack). Thus, $r < 1$ indicates that the default choice is suboptimal (a faster non-default alternative exists), while $r > 1$ indicates that the default is best among the exposed choices. Across all three systems, the heatmaps exhibit structured regions, often at larger scales and for specific message sizes, where defaults fall short of the best alternative by roughly 30–40%, and in the most pronounced case achieving only 20% of the optimal performance.

Default selection heuristics are typically engineered to be conservative and broadly portable; as a result, they may fail to capture the platform-specific characteristics that matter in practice. This motivates systematic tuning of collective communication algorithms and their selection parameters. PICO’s outputs can be used as empirical basis to tune libraries algorithmic choices: Open MPI supports overriding algorithm selection using `coll_tuned` dynamic decision files [60], and analogous mechanisms exist across other MPI implementations and **CCL* libraries (via configuration files or environment variables).

Additionally, sub-optimality is not limited to algorithm choice: performance can change dramatically with back-

end/transport parameters that are easy to overlook if they are not made explicit in the experiment specification. To illustrate this sensitivity, we fix the `MPI_Allreduce` algorithm to Ring on Leonardo at 32 nodes (removing algorithmic variability) and vary only `UCX_MAX_RNDV_RAILS`, a UCX parameter that caps the number of network rails used by the rendezvous protocol for large-message transfers. Fig. 7 reports execution times normalized to the default `UCX_MAX_RNDV_RAILS=2`. For large messages in the rendezvous regime, increasing the rail limit to 4 reduces runtime up to 10%, whereas smaller messages (typically in the eager regime) are largely unaffected.

This result reinforces two practical points. First, meaningful tuning requires dealing with both algorithm selection and backend configuration (*R3*): even a strong algorithm can appear weak under an unfavorable transport setting. Second, reproducibility and regression diagnosis depend on recording the *effective* configuration used in each run (*R5*). PICO captures both requested settings and platform configuration defaults, making it straightforward to rerun controlled A/B tests in which only a single knob changes and to interpret performance differences.

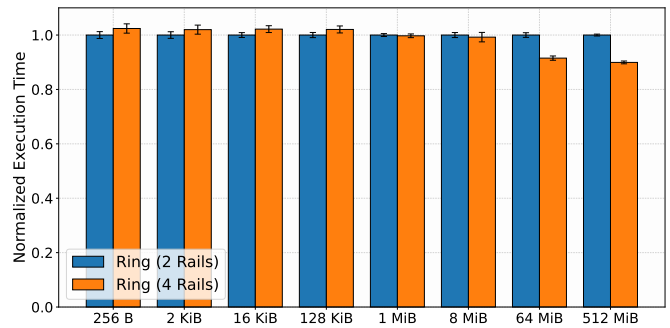


Fig. 7. Ring `MPI_Allreduce` on Leonardo (32 nodes; Open MPI 4.1.6; UCX 1.15.0). Latency normalized to the default `UCX_MAX_RNDV_RAILS=2` (lower is better). Setting `UCX_MAX_RNDV_RAILS=4` (orange) yields up to a 10% improvement over the default (blue).

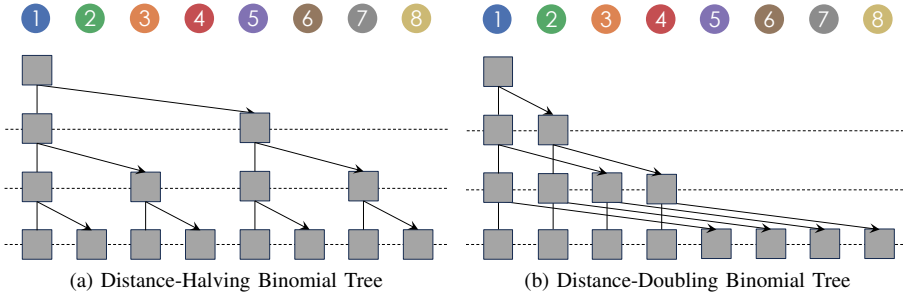


Fig. 8. Binomial-tree broadcast schedules with different partner ordering: (a) distance-halving vs. (b) distance-doubling. Both complete in $\log_2(p)$ rounds and transmit the same total volume, but differ in how communication distance evolves across steps. In particular, the distance halving approach maximizes communication locality at later communication rounds, when the overall communication volume is greater.

B. Algorithmic differences

Performance models are an essential tool for understanding collective behavior: they provide analytic guidance on how step count, overall communication volume, and reduction costs scale with process count and vector size. As discussed in Sec. II, modern hierarchical systems can exhibit strong topology-dependent bottlenecks. Accurate performance prediction therefore benefits from refinements that track traffic flow and link saturation [61, 25]. Our goal is not to argue against modeling, but to show why microbenchmarking remains a practical necessity: even when two algorithms are equivalent under a cost-model, their performance can differ substantially on different topologies and allocations.

We illustrate this by comparing two broadcast algorithms on Leonardo: *distance-doubling* binomial tree broadcast (Open MPI’s binomial broadcast implementation [62]) and *distance-halving* binomial tree broadcast (MPICH binomial tree implementation). The two communication schedules, illustrated in Fig. 8, appear indistinguishable under a classic α - β modeling: both complete in $\log_2(p)$ rounds and transmit the same total communication volume. However, distance-doubling keeps communication local in the early rounds and defers longer-distance exchanges to later rounds. By contrast, distance-halving performs longer-distance exchanges earlier and becomes progressively more local in later rounds. On a topology with non-uniform link costs or tapered global bandwidth, this ordering can shift how many exchanges traverse local versus global links at each step, and thus where congestion pressure concentrates [10].

To quantify this effect, PICO’s network tracer reveals (Fig. 9) that, for the same 128 nodes allocation on Leonardo’s Dragonfly network, distance-doubling algorithm sends nearly all volume inter-group (external 122-n, internal 5-n, where n is the size of the send buffer in Bytes), whereas distance-halving keeps 90-n Bytes intra-group, reducing inter-group traffic to 37-n. This effect arises naturally from the interaction of rank placement and algorithm’s communication schedule, illustrating why topology- and placement-aware diagnosis is valuable in realistic runs. Fig. 10 reports the measured execution times

bcast	
Algorithm:	binomial_doubling
Internal bytes:	5 n bytes
External bytes:	122 n bytes
Total bytes:	127 n bytes
Algorithm:	binomial_halving
Internal bytes:	90 n bytes
External bytes:	37 n bytes
Total bytes:	127 n bytes

Fig. 9. Network volume estimates of distance-halving and distance-doubling broadcast on a 128 nodes allocation on Leonardo. Distance-halving broadcast induces only 29% of total communication volume to across-group (*external*) links, while distance-doubling 96%.

of the two algorithms. The curves are nearly identical for small messages (up to 16 KiB), but diverge sharply once large-message transfers dominate. At 512 MiB, the distance-doubling algorithm is $\times 2.5$ slower at 757 ms compared to 304 ms of the distance-halving one. Moreover, we can notice how the Open MPI internal Binomial algorithm appears to be almost one order of magnitude slower at 1.9 s, indicating inefficiencies in its implementation regardless of the algorithm of choice.

Thus, cost models capture step/volume trade-offs, but they may not distinguish between algorithms that are equivalent in those metrics unless topology and placement effects are explicitly modeled. PICO complements modeling by enabling backend-neutral comparisons (R2) while also providing placement-aware diagnosis thanks to its rich metadata gathering (R5). In practice, these capabilities make it possible to identify when a default algorithm over-stresses links on hierarchical networks and to justify an alternative schedule with both structural evidence (tracer) and measured performance.

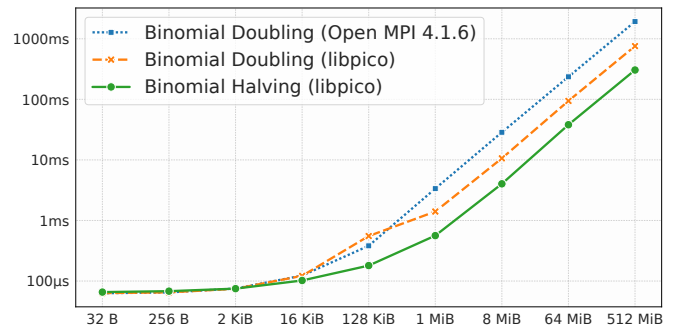


Fig. 10. Distance-doubling and distance-halving MPI_Bcast on Leonardo, Open MPI (4.1.6). Latency vs. message size for 128 nodes, 4 processes per node (log-log axes); 512MiB: libpico 757ms (doubling) vs. 304ms (halving). Open MPI (doubling) 1.9s is one order of magnitude slower.

C. Fine grained instrumentation

Collective communications are composed of multiple algorithmic steps, each one stressing different hardware resources (NIC/fabric, caches/DRAM and CPU/GPU arithmetic units).

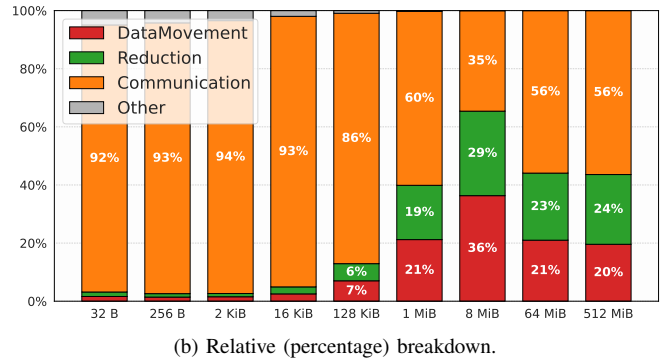
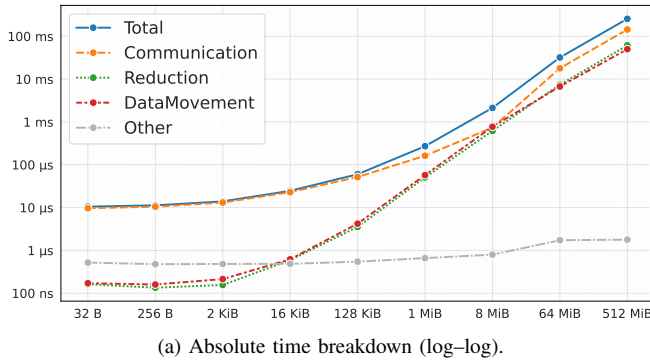


Fig. 11. Instrumented Rabenseifner Allreduce on 8 nodes (Leonardo, Open MPI 4.1.6; `libpico` implementation). (a) Absolute runtime breakdown into tagged components: Communication (network transfer), Reduction (compute), Data Movement (staging/copies to work buffers), and residual Other overhead. (b) Same data expressed as percentage shares, highlighting the shift from communication-dominated behavior at small messages to substantial data-movement and reduction contributions at larger messages.

Crucially, the different performance characteristics of these resources shape how the cost of each step scales with message size and node count, meaning that the aggregate end-to-end timing of a collective can hide where inefficiencies originate.

An instrumented benchmark run of the `libpico` reference Rabenseifner implementation on an 8-node allocation on Leonardo (Fig. 11) reveals several non-obvious behaviors. Fig. 11a shows the *aggregate* runtime (blue) together with the tagged components—network communication (orange), reduction computation (green), and intra-node data movement (red; e.g., staging and copies to working buffers), while Fig. 11b reports their relative weight.

We first consider the overall trend in Fig. 11a. For small message sizes (up to 128 KiB), the communication curve is nearly indistinguishable from the aggregate curve, suggesting that end-to-end performance is dominated by network communication. In particular, for message sizes up to 2 KiB the total runtime is nearly constant ($10 \mu\text{s}$ at 32 B, $11 \mu\text{s}$ at 256 B, and $10 \mu\text{s}$ at 2 KiB), consistent with a latency-dominated regime where fixed network startup costs outweigh bandwidth effects. However, the tagged breakdown shows that this “communication-dominated” interpretation does not hold uniformly as message size grows. After 128 KiB, Fig. 11b shows that the relative cost of communication drops sharply (from nearly 95% to 35%) before increasing again to 56% at 64 MiB and 512 MiB. This non-monotonic trend indicates that the scaling driver of the collective changes with message size: once per-message latency is amortized, the overall runtime is no longer governed by network transfer alone. The missing fraction is largely absorbed by *intra-node data movement* and *reduction*. As message size grows into the MiB range, the data-movement component rises substantially (staging and copies to work buffers), and reduction becomes a first-class contributor as more bytes must be combined at each step.

This breakdown can be interpreted through the intuition of the roofline model [63]: performance is limited by whichever resource is currently most constraining. While strictly speaking a collective communication is not a single kernel, its com-

ponents can be interpreted as kernels with different limiting resources. The communication component is primarily limited by network latency for small messages and bandwidth for large ones, whereas the staging and reduction components are limited by local memory bandwidth and compute throughput (we assume local memory latency to be negligible). From this perspective, the non-monotonic behavior observed in Fig. 11a–11b can be explained as follows: as message size grows, the dominant limiter shifts from network-latency to local data movement and reduction, effectively moving the collective onto a memory-bandwidth roof even though the operation is “communication-heavy” at a high level. At very large messages, the network bandwidth becomes the dominant limit, but the persistent contribution of data movement and reduction indicates that local memory bandwidth and compute throughput still cap end-to-end gains.

The crossover points at which this shift occurs, as well as the magnitude of the shift, depend on the machine’s hardware characteristics. Consequently, the same end-to-end Allreduce curve can hide different underlying bottlenecks on different systems, reinforcing the need for fine grained profiling when tuning or diagnosing performance. Thus, PICO’s instrumentation (*R1*) on backend-neutral baselines (*R2*) exposes hidden bottleneck shifts that are opaque under end-to-end timing.

D. Simulation results with ATLAHS

We evaluate PICO on real AI workloads using trace replay to translate microbenchmark-level effects into end-to-end performance changes. Specifically, we use the ATLAHS [38], a recently published toolchain that can trace NCCL executions and replay them via GOAL traces [64] running on a number of network simulators. ATLAHS can generate different replayable traces from the same raw NCCL logs, allowing us to swap collective algorithms and protocol choice while preserving original invocation sequence and message sizes. This enables controlled what-if analysis without needing to re-run the actual workload. Additionally, this capability allows us to test collective algorithms that might not be already implemented in NCCL by simply implementing new translation units in

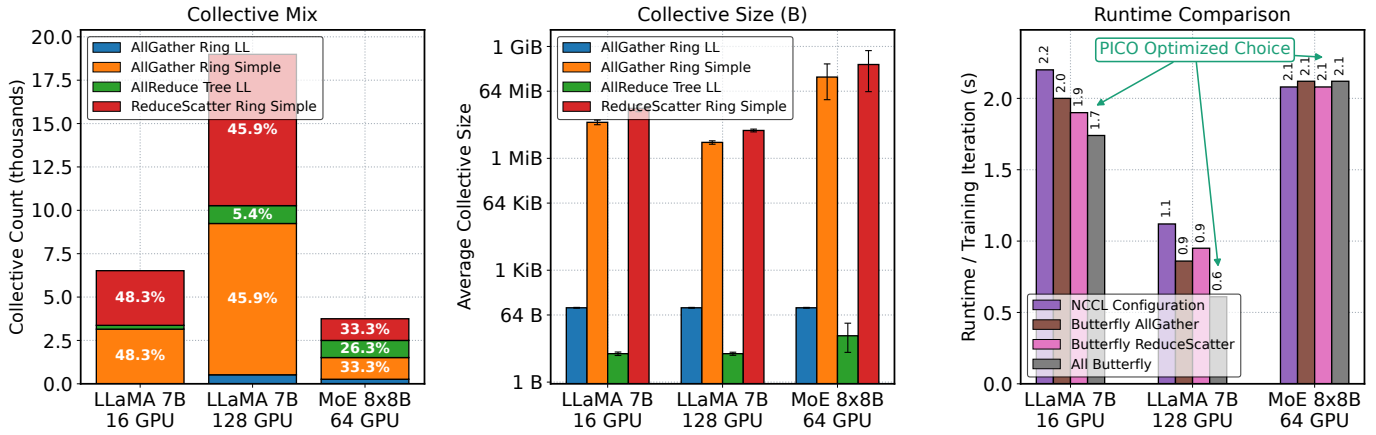


Fig. 12. ATLAHS-based trace analysis and replay for AI training workloads. Left/center: collective mix and message-size distributions extracted from NCCL traces. Right: projected per-iteration time after substituting collective algorithms/protocols while preserving the original invocation sequence and message sizes. PICO-derived profiles reduce per-iteration time of up to 44%.

the simulator toolchain. The choice of application traces was intentionally geared toward AI workloads, as they feature large, repeated collectives for which algorithm, protocol, and transport decisions are more likely to produce observable end-to-end effects. Similar sensitivity can also arise in traditional HPC applications with communication-heavy phases, such as those based on 3D FFTs and their associated all-to-all exchanges [65]; however, such traces were not available among the open-source traces considered in this study.

The traces we consider are collected with NCCL 2.22, a version that provides Ring and Tree implementations for AllReduce (i.e., Distance-Halving Reduce followed by Distance-Doubling Broadcast), but crucially only a Ring algorithm for ReduceScatter and AllGather. Newer NCCL releases have since added an additional algorithm, PAT [66], which is a Binomial Butterfly. During tracing, both the collective *algorithm* and *protocol* were recorded for each invocation, alongside several other key pieces of information such as the collective size and details about the communicator and GPU streams. While the algorithm controls the overall schedule of the collective, the protocol controls the low-level transfer/synchronization strategy; available NCCL protocols are *Simple*, which favors large-message bandwidth, and *LL*, which reduces small-message latency via flag-based synchronization [67].

Our chosen workloads are LLaMA 7B [68], run on 16 and 128 GPUs, and a Mistral MoE (Mixture-of-Experts) [69] on 64 GPUs: for clarity of exposition we will refer to those traces as L16, L128 and MoE. Analyzing number and types of collective invocations present in the raw traces we notice that (Fig. 12; left) for both L16, and L128 the overall majority of those collective are AllGather Ring Simple (L16 48.3%, L128 45.9%), and ReduceScatter Ring Simple (L16 48.3%, L128 45.9%), while the Allreduce Tree LL and ReduceScatter Ring LL accounted for a small minority of invocations (L16 1-3%, L128 3-6%). In the MoE trace we notice an overall lower number of invocations almost equally distributed between Allreduce Tree LL, ReduceScatter Ring

Simple and Allgather Ring Simple. Taking into consideration size distribution (Fig. 12; center) it appears that: (i) Allreduce invocations were all of relatively small size (i.e., < 1 KiB), (ii) AllGather and ReduceScatter invocations had a median size of 3-6 MiB (L16) and 7-14 MiB (L128), while (iii) a significantly higher size of 33-67 MiB in the MoE trace. We note that for this analysis we purposely ignored point-to-point sends to focus purely on the collective operations.

Using PICO, we identified candidate *collective profiles* (algorithm/protocol choices) for the observed communicator and message-size distributions. In particular, for the L16 and L128 traces we identified a profile consisting of AllGather and ReduceScatter Binomial Butterfly algorithm with Simple protocol, and Allreduce Tree algorithm with LL protocol.

Fig. 12 (right) reports the resulting simulated end-to-end runtimes, reported over a single training iteration: the PICO-optimized profiles improve over native NCCL by 21% on L16 and 44% on L128. On the other hand, the optimized profile found for the MoE model reported no measurable improvements, indicating that a good profile was already in use when the trace was instrumented. This is likely because MoE has larger collectives on average, which tend to perform better with Ring implementations. For completeness, alternative suboptimal profiles were played alongside the optimal one, confirming runtime variation across algorithm/protocol choices, thus further highlighting the workload sensitivity to collective configurations.

Due to PICO’s extensible design (R6), we are able to evaluate NCCL algorithms and conduct a systematic evaluation campaign (R4) across both algorithms selection and protocol configuration (R3), observing end-to-end runtime improvements of real world applications.

V. CONCLUSIONS

We presented PICO, a lightweight and extensible framework for benchmarking collective communication operations across heterogeneous HPC and AI systems. Unlike existing tools,

PICO integrates fine-grained profiling, rich metadata collection, and automated orchestration to support reproducible, system-aware performance analysis. Through its modular architecture, PICO enables portable comparisons across MPI, *CCL, and user-defined algorithms, while its integrated post-processing tools facilitate both high-level performance summaries and detailed algorithmic phase breakdowns.

Our case studies demonstrated how PICO can reveal sub-optimal default algorithm selections and guide library tuning, highlight subtle performance trade-offs between closely related algorithms, quantify the impact of backend parameters, and identify hidden bottlenecks thanks to its reference backend-neutral implementations and instrumentation capabilities. Finally, we demonstrated that PICO tuned collective configurations can translate to real world end-to-end improvements.

These examples underscore PICO’s utility for algorithm designers, application developers, and system administrators alike. By bridging systematic benchmarking with actionable insights, PICO aims to become a foundational tool for advancing the reproducible performance analysis of collective communications in next-generation computing systems.

VI. ACKNOWLEDGMENTS

This work is supported by the European Union’s Horizon Europe under grant 101175702 (NET4EXA), by Sapienza University Grants ADAGIO and D2QNeT (*Bando per la ricerca di Ateneo 2023 and 2024*), and by a research grant from Microsoft Azure. The research was also conducted as part of the FastTrackAI project at the Singapore-ETH Centre, which was established collaboratively between ETH Zurich and the National Research Foundation, Singapore. Additionally, this research is supported by the National Research Foundation, Singapore (NRF), and the Ministry of Digital Development and Information (MDDI) under the AI Visiting Professorship (Award No. AIVP-2025-005). We acknowledge ISCRA for awarding this project access to the LEONARDO supercomputer, owned by the EuroHPC Joint Undertaking, hosted by CINECA (Italy). We acknowledge the EuroHPC Joint Undertaking, the LUMI consortium, and BSC for granting access to the LUMI and MareNostrum 5 supercomputers. These resources, hosted by CSC (Finland) and the Barcelona Supercomputing Center (Spain), were provided through the EuroHPC Regular Access program.

REFERENCES

- [1] Lawrence Livermore National Laboratory. *Lawrence Livermore National Laboratory’s El Capitan verified as world’s fastest supercomputer*. Press release. Feb. 2025. URL: <https://www.llnl.gov/article/52061/lawrence-livermore-national-laboratorys-el-capitan-verified-worlds-fastest-supercomputer>.
- [2] TOP500 Project. *TOP500 List – June 2025*. <https://top500.org/lists/top500/2025/06/>. Accessed: Jul. 18, 2025. June 2025.
- [3] Xiang-ke Liao et al. “Moving from exascale to zettascale computing: challenges and techniques”. In: *Frontiers of Information Technology & Electronic Engineering* 19.10 (2018), pp. 1236–1244. ISSN: 2095-9230. DOI: 10.1631/FITEE.1800494. URL: <https://doi.org/10.1631/FITEE.1800494>.
- [4] Robert Lucas et al. *DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) Report: Top Ten Exascale Research Challenges*. Report. U.S. Department of Energy, ASCR, 2014. DOI: 10.2172/1222713.
- [5] Ping-Jing Lu, Ming-Che Lai, and Jun-Sheng Chang. “A Survey of High-Performance Interconnection Networks in High-Performance Computer Systems”. In: *Electronics* 11.9 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11091369. URL: <https://www.mdpi.com/2079-9292/11/9/1369>.
- [6] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. Computer engineering series. WCB/McGraw-Hill, 1998. ISBN: 9780070317987. URL: <https://books.google.it/books?id=OJNQAAAAMAAJ>.
- [7] Adam Weingram et al. “xCCL: A Survey of Industry-Led Collective Communication Libraries for Deep Learning”. In: *Journal of Computer Science and Technology* 38.1 (Feb. 2023), pp. 166–195. ISSN: 1860-4749. DOI: 10.1007/s11390-023-2894-6. URL: <https://doi.org/10.1007/s11390-023-2894-6>.
- [8] Ignacio Laguna et al. “A large-scale study of MPI usage in open-source HPC applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC ’19*. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356176. URL: <https://doi.org/10.1145/3295500.3356176>.
- [9] Pavan Balaji et al. “MPI on a Million Processors”. In: *Proceedings of the 16th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Espoo, Finland: Springer-Verlag, 2009, pp. 20–30. ISBN: 9783642037696. DOI: 10.1007/978-3-642-03770-2_9. URL: https://doi.org/10.1007/978-3-642-03770-2_9.
- [10] Daniele De Sensi et al. *Swing: Short-cutting Rings for Higher Bandwidth Allreduce*. 2024. arXiv: 2401.09356 [cs.DC]. URL: <https://arxiv.org/abs/2401.09356>.
- [11] Andres Sewell et al. “Bruck Algorithm Performance Analysis for Multi-GPU All-to-All Communication”. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. HPCAsia ’24*. Nagoya, Japan: Association for Computing Machinery, 2024, pp. 127–133. ISBN: 9798400708893. DOI: 10.1145/3635035.3635047. URL: <https://doi.org/10.1145/3635035.3635047>.
- [12] Amanda Bienz, Shreeman Gautam, and Amun Kharel. *A Locality-Aware Bruck Allgather*. 2022. arXiv: 2206.

- 03564 [cs.DC]. URL: <https://arxiv.org/abs/2206.03564>.
- [13] Prithwish Basu et al. *Efficient All-to-All Collective Communication Schedules for Direct-Connect Topologies*. 2024. arXiv: 2309.13541 [cs.DC]. URL: <https://arxiv.org/abs/2309.13541>.
- [14] Paul Sack and William Gropp. “Collective Algorithms for Multiported Torus Networks”. In: *ACM Trans. Parallel Comput.* 1.2 (Feb. 2015). ISSN: 2329-4949. DOI: 10.1145/2686882. URL: <https://doi.org/10.1145/2686882>.
- [15] Dongkyun Lim and John Kim. “TidalMesh: Topology-Driven AllReduce Collective Communication for Mesh Topology”. In: *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2025, pp. 1526–1540. DOI: 10.1109/HPCA61900.2025.00114.
- [16] Sascha Hunold, Alexandra Carpen-Amarie, and Jesper Larsson Träff. “Reproducible MPI Micro-Benchmarking Isn’t As Easy As You Think”. In: *Proceedings of the 21st European MPI Users’ Group Meeting, EuroMPI/ASIA ’14*. Kyoto, Japan: Association for Computing Machinery, 2014, pp. 69–76. ISBN: 9781450328753. DOI: 10.1145/2642769.2642785. URL: <https://doi.org/10.1145/2642769.2642785>.
- [17] Yongji Wu et al. “MCCS: A Service-based Approach to Collective Communication for Multi-Tenant Cloud”. In: *Proceedings of the ACM SIGCOMM 2024 Conference*. ACM SIGCOMM ’24. Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 679–690. ISBN: 9798400706141. DOI: 10.1145/3651890.3672252. URL: <https://doi.org/10.1145/3651890.3672252>.
- [18] NVIDIA Corporation. *NVIDIA Collective Communication Library (NCCL) Documentation*. Online documentation. Accessed July 23, 2025. 2025. URL: <https://docs.nvidia.com/deeplearning/nccl/index.html>.
- [19] Advanced Micro Devices, Inc. *ROCm Communication Collectives Library (RCCL) Documentation, v2.22.3*. Online documentation. Accessed Jul. 23, 2025. 2025. URL: <https://rocm.docs.amd.com/projects/rccl>.
- [20] Zhenhao He et al. “ACCL: FPGA-Accelerated Collectives over 100 Gbps TCP-IP”. In: *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2021, pp. 33–43. DOI: 10.1109/H2RC54759.2021.00009.
- [21] Daniele De Sensi et al. “Exploring GPU-to-GPU Communication: Insights into Supercomputer Interconnects”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’24)*. Nov. 2024. DOI: 10.1109/SC41406.2024.00039.
- [22] Manjunath Gorentla Venkata et al. “Unified Collective Communication (UCC): An Unified Library for CPU, GPU, and DPU Collectives”. In: *IEEE Symposium on High-Performance Interconnects, HOTI 2024, Albuquerque, NM, USA, August 21-23, 2024*. IEEE, 2024, pp. 37–46. DOI: 10.1109/HOTI63208.2024.00018. URL: <https://doi.org/10.1109/HOTI63208.2024.00018>.
- [23] *The Unified Communication X Library*. <http://www.openucx.org>.
- [24] OpenFabrics Interfaces Working Group (OFIWG). *libfabric: Open Fabric Interfaces framework for high-performance networking*. GitHub repository. Accessed Jul. 23, 2025. 2025. URL: <https://github.com/ofiwg/libfabric>.
- [25] Yijia Zhang et al. “Quantifying the impact of network congestion on application performance and network metrics”. In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 2020, pp. 162–168. DOI: 10.1109/CLUSTER49012.2020.00026.
- [26] Tommaso Bonato et al. *REPS: Recycled Entropy Packet Spraying for Adaptive Load Balancing and Failure Mitigation*. 2025. arXiv: 2407.21625 [cs.NI]. URL: <https://arxiv.org/abs/2407.21625>.
- [27] Muhammad Shuaib Qureshi et al. “A comparative analysis of resource allocation schemes for real-time services in high-performance computing systems”. In: *International Journal of Distributed Sensor Networks* 16.8 (2020), p. 1550147720932750. DOI: 10.1177/1550147720932750. eprint: <https://doi.org/10.1177/1550147720932750>. URL: <https://doi.org/10.1177/1550147720932750>.
- [28] Andrii Kovalov et al. “Task-Node Mapping in an Arbitrary Computer Network Using SMT Solver”. In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 177–191. ISBN: 978-3-319-66845-1.
- [29] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. “Optimization of collective communication operations in MPICH”. In: *The International Journal of High Performance Computing Applications* 19.1 (2005), pp. 49–66.
- [30] The Ohio State University. *OSU Micro-Benchmarks (OMB)*. Online. Accessed Jul. 21, 2025. URL: <https://mvapich.cse.ohio-state.edu/benchmarks/>.
- [31] NVIDIA Corporation. *NCCL-Tests: Performance and correctness micro-benchmarks for NVIDIA NCCL*. GitHub repository. Accessed Jul. 21, 2025. URL: <https://github.com/NVIDIA/nccl-tests>.
- [32] Intel Corporation. *Intel® MPI Benchmarks User Guide, version 2021.2*. Online. Accessed Jul. 21, 2025. 2021. URL: <https://www.intel.com/content/www/us/en/docs/mpi-library/user-guide-benchmarks/2021-2/overview.html>.
- [33] Hunold Sascha. *ReproMPI Benchmark for MPI Collective*. <https://github.com/hunsa/reprompi>. GitHub repository. Accessed: 2025-12-19. 2025.
- [34] Mert Hidayetoglu et al. “CommBench: Micro-Benchmarking Hierarchical Networks with Multi-GPU, Multi-NIC Nodes”. In: *Proceedings of the 38th ACM*

- International Conference on Supercomputing*. 2024, pp. 426–436.
- [35] T. Zwinger, J. Heikonen, and P. Manninen. “LUMI supercomputer for European researchers”. In: *Galileo Conference: Solid Earth and Geohazards in the Exascale Era*. Barcelona, Spain, May 23–26, 2023, GC11-solidearth–25. DOI: 10.5194/egusphere-gc11-solidearth-25. URL: <https://doi.org/10.5194/egusphere-gc11-solidearth-25>.
- [36] Matteo Turisini, Giorgio Amati, and Mirko Cestari. *LEONARDO: A Pan-European Pre-Exascale Supercomputer for HPC and AI Applications*. 2023. arXiv: 2307.16885 [cs.DC]. URL: <https://arxiv.org/abs/2307.16885>.
- [37] Fabio Banchelli et al. *Introducing MareNostrum5: A European pre-exascale energy-efficient system designed to serve a broad spectrum of scientific workloads*. 2025. arXiv: 2503.09917 [cs.DC]. URL: <https://arxiv.org/abs/2503.09917>.
- [38] Siyuan Shen et al. “ATLAHS: An Application-centric Network Simulator Toolchain for AI, HPC, and Distributed Storage”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’25. Association for Computing Machinery, 2025, pp. 349–367. ISBN: 9798400714665. DOI: 10.1145/3712285.3759838. URL: <https://doi.org/10.1145/3712285.3759838>.
- [39] Rick Merritt. *What Is NVLink?* NVIDIA Official Blog. Accessed August 8, 2025. Mar. 2023. URL: <https://blogs.nvidia.com/blog/what-is-nvidia-nvlink/>.
- [40] David Schor. *ISSCC 2018: AMD’s Zeppelin; Multi-chip routing and packaging*. WikiChip Fuse blog. Accessed Aug. 11, 2025. Mar. 2018. URL: <https://fuse.wikichip.org/news/1064/isscc-2018-amds-zeppelin-multi-chip-routing-and-packaging/>.
- [41] Jon Ames and Ron Lowman. *How Ultra Ethernet and UALink Enable High-Performance, Scalable AI Networks*. Synopsys Blog. Accessed Aug. 11, 2025. Jan. 2025. URL: <https://www.synopsys.com/articles/ultra-ethernet-ualink-ai-networks.html>.
- [42] Pengfei Zuo et al. *Serving Large Language Models on Huawei CloudMatrix384*. 2025. arXiv: 2506.12708 [cs.DC]. URL: <https://arxiv.org/abs/2506.12708>.
- [43] Broadcom Inc. *Scale-Up Ethernet (SUE) Framework Specification*. Technical specification (PDF). Accessed Aug. 11, 2025. July 2025. URL: <https://docs.broadcom.com/doc/scale-up-ethernet-framework>.
- [44] NVIDIA Corporation. *NVIDIA GB200 NVL72 (Grace + Blackwell) – Rack-Scale AI System*. Product web page. Accessed Aug. 11, 2025. 2025. URL: <https://www.nvidia.com/en-us/data-center/gb200-nvl72/>.
- [45] Rajkumar Buyya, Toni Cortes, and Hai Jin. “An Introduction to the InfiniBand Architecture”. In: *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. 2002, pp. 616–632. DOI: 10.1109/9780470544839.ch42.
- [46] Daniele De Sensi et al. “An In-Depth Analysis of the Slingshot Interconnect”. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2020, pp. 1–14. DOI: 10.1109/sc41405.2020.00039. URL: <http://dx.doi.org/10.1109/SC41405.2020.00039>.
- [47] Torsten Hoefler et al. *Ultra Ethernet’s Design Principles and Architectural Innovations*. 2025. arXiv: 2508.08906 [cs.NI]. URL: <https://arxiv.org/abs/2508.08906>.
- [48] Jongryoul Kim et al. “Technology-Driven, Highly-Scalable Dragonfly Topology”. In: vol. 36. July 2008, pp. 77–88. ISBN: 978-0-7695-3174-8. DOI: 10.1109/ISCA.2008.19.
- [49] Scott Atchley et al. “Frontier: Exploring Exascale”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’23. Denver, CO, USA: Association for Computing Machinery, 2023. ISBN: 9798400701092. DOI: 10.1145/3581784.3607089. URL: <https://doi.org/10.1145/3581784.3607089>.
- [50] Alexander Shpiner et al. “Dragonfly+: Low Cost Topology for Scaling Datacenters”. In: Feb. 2017. DOI: 10.1109/HiPINEB.2017.11.
- [51] Forschungszentrum Jülich, Jülich Supercomputing Centre. *JUPITER Technical Overview*. Technical overview page. Last modified Jan 7, 2025; accessed Aug 11, 2025. 2025. URL: <https://www.fz-juelich.de/en/ias/jsc/jupiter/tech>.
- [52] Daniele De Sensi et al. *Noise in the Clouds: Influence of Network Performance Variability on Application Scalability*. 2022. arXiv: 2210.15315 [cs.DC]. URL: <https://arxiv.org/abs/2210.15315>.
- [53] Daniele De Sensi et al. “Bine Trees: Enhancing Collective Operations by Optimizing Communication Locality”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’25)*. Nov. 2025. DOI: ToAppear.
- [54] Krishna Kandalla et al. “Designing multi-leader-based Allgather algorithms for multi-core clusters”. In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5160896.
- [55] Jesper Larsson Träff. “Efficient Allgather for Regular SMP-Clusters”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Bernd Mohr et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 58–65. ISBN: 978-3-540-39112-8.
- [56] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. “Accurately Measuring Overhead, Communication Time and Progression of Blocking and Nonblocking Collective Operations at Massive Scale”. In: *International Journal of Parallel, Emergent and Distributed Systems* 25.4 (July 2010), pp. 241–258. ISSN: 1744-5779.
- [57] Sascha Hunold and Alexandra Carpen-Amarie. “On the impact of synchronizing clocks and processes on bench-

- marking MPI collectives”. In: *Proceedings of the 22nd European MPI Users’ Group Meeting*. 2015, pp. 1–10.
- [58] Sirshak Das et al. *Enhancing Communication Observability of AI Workloads with NCCL Inspector*. <https://developer.nvidia.com/blog/enhancing-communication-observability-of-ai-workloads-with-nccl-inspector/>. NVIDIA Developer Blog, Dec. 2025.
- [59] Torsten Hoefler et al. “Netgauge: A network performance measurement framework”. In: *International Conference on High Performance Computing and Communications*. Springer. 2007, pp. 659–671.
- [60] The Open MPI Community. *Open MPI 5.0: 11.10. Tuning Collectives (coll-tuned)*. Online documentation. Last updated Jul. 31, 2025; accessed Aug. 11, 2025. 2025. URL: <https://docs.open-mpi.org/en/v5.0.x/tuning-apps/coll-tuned.html>.
- [61] Rohini Uma-Vaideswaran et al. “A Peak Performance Model for All-to-all on Hierarchical Systems and Its Applications”. In: *Proceedings of the SC ’25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC Workshops ’25. Association for Computing Machinery, 2025, pp. 1442–1451. ISBN: 9798400718717. DOI: 10.1145/3731599.3767704. URL: <https://doi.org/10.1145/3731599.3767704>.
- [62] Open MPI Project. *ompi/mca/coll/base/coll_base_bcast.c*. https://github.com/open-mpi/ompi/blob/dd6a7a3ad0c37dde58da7ccabe2c54da8f51d130/ompi/mca/coll/base/coll_base_bcast.c. Commit dd6a7a3ad0c37dde58da7ccabe2c54da8f51d130, line 343. Accessed 2025-12-16. 2025.
- [63] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.
- [64] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. “Group Operation Assembly Language - A Flexible Way to Express Collective Communication”. In: *2009 International Conference on Parallel Processing*. 2009, pp. 574–581. DOI: 10.1109/ICPP.2009.70.
- [65] Lisandro Dalcin, Mikael Mortensen, and David E Keyes. *Fast parallel multidimensional FFT using advanced MPI*. 2018. arXiv: 1804.09536 [cs.DC]. URL: <https://arxiv.org/abs/1804.09536>.
- [66] Sylvain Jeugey. *PAT: a new algorithm for all-gather and reduce-scatter operations at scale*. 2025. arXiv: 2506.20252 [cs.DC]. URL: <https://arxiv.org/abs/2506.20252>.
- [67] Zhiyi Hu et al. *Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms*. 2025. arXiv: 2507.04786 [cs.DC]. URL: <https://arxiv.org/abs/2507.04786>.
- [68] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: <https://arxiv.org/abs/2302.13971>.
- [69] Albert Q. Jiang et al. *Mixtral of Experts*. 2024. arXiv: 2401.04088 [cs.LG]. URL: <https://arxiv.org/abs/2401.04088>.