CommBench: Micro-Benchmarking Hierarchical Networks with Multi-GPU, Multi-NIC Nodes

Mert Hidayetoglu
Stanford University
California, USA

Simon Garcia de Gonzalo
Sandia National Laboratories
New Mexico, USA

Elliott Slaughter
SLAC National Accelerator Laboratory
California, USA

Yu Li
University of Illinois at Urbana-Champaign
Illinois, USA

Christopher Zimmer
Oak Ridge National Laboratory
Tennessee, USA

Tekin Bicer
Argonne National Laboratory
Illinois, USA

Bin Ren
William & Mary
Virginia, USA

William Gropp
University of Illinois at Urbana-Champaign
Illinois, USA

Wen-mei Hwu
Nvidia Research / University of Illinois at Urbana-Champaign
Illinois, USA

Alex Aiken
Stanford University
California, USA

ABSTRACT

Modern high-performance computing systems have multiple GPUs and network interface cards (NICs) per node. The resulting network architectures have multilevel hierarchies of subnetworks with different interconnect and software technologies. These systems offer multiple vendor-provided communication capabilities and library implementations (IPC, MPI, NCCL, RCCL, OneCCL) with APIs providing varying levels of performance across the different levels. Understanding this performance is currently difficult because of the wide range of architectures and programming models (CUDA, HIP, OneAPI).

We present CommBench, a library with cross-system portability and a high-level API that enables developers to easily build microbenchmarks relevant to their use cases and gain insight into the performance (bandwidth & latency) of multiple implementation libraries on different networks. We demonstrate CommBench with three sets of microbenchmarks that profile the performance of six systems. Our experimental results reveal the effect of multiple NICs on optimizing the bandwidth across nodes and also present the performance characteristics of four available communication libraries within and across nodes of NVIDIA, AMD, and Intel GPU networks.

ACM Reference Format:
Mert Hidayetoglu, Simon Garcia de Gonzalo, Elliott Slaughter, Yu Li, Christopher Zimmer, Tekin Bicer, Bin Ren, William Gropp, Wen-mei Hwu, and Alex Aiken. 2024. CommBench: Micro-Benchmarking Hierarchical Networks with Multi-GPU, Multi-NIC Nodes. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3650200.3656591

ICS '24, June 04–07, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656591

1 INTRODUCTION

Communication networks on high-performance computing (HPC) systems are complex and diverse. The challenge for application and library developers when targeting such machines is to understand a network's characteristics and the resulting performance implications to design or select tailored communication strategies for their programs [3, 16].

In particular, HPC network architectures have become more hierarchical in an effort to sustain high bandwidth, low latency, and energy-efficient communication as aggregate compute has grown. HPC systems today emphasize *fat* node designs with large numbers of accelerators per node connected by a fast internal network as well as the traditional network across nodes [5, 8, 9, 37]. Applications that exploit these hierarchical networks can achieve significantly higher overall performance [23, 33].

Traditional HPC network benchmark suites perform point-to-point (P2P) and collective, e.g., scatter—point-to-all and all-to-all, communications. These tests assume no hierarchy—all points are peers. In hierarchical systems, the performance of such tests depends on the locality of the endpoints in the communication hierarchy. For example, point-to-point tests give different results for endpoints within the same node vs different nodes.

Our key insight is that in characterizing the performance of a hierarchical network, we should use groups of processors corresponding to the levels of that hierarchy. In particular, in addition to using either individual processors or all processors, we should also cover intermediate-sized groups of processors, such as all processors in a node. We can then develop cross-group communication

patterns to evaluate performance across groups of communicating processors that accurately represent the underlying network hierarchy. To this end, we introduce CommBench, an extensible framework for constructing benchmarks of nontrivial communication patterns, stress-testing communication layers, and identifying optimal communication configurations in hierarchical machines.

A significant challenge is that each system has different numbers of GPUs and *network interface cards* (NICs) per node, and thus each system exhibits a different physical topology. Figure 1 shows example systems such as Frontier and Aurora, which consist of dual-die GPUs where each die is connected to an intra-node network in a heterogeneous way, yielding non-obvious performance behavior that can be explored with CommBench. To accommodate such level of diversity in the systems of interest, we provide a unified parameterization of the group-to-group patterns and make them portable and scalable across systems with various architectures.

Moreover, communication libraries often exhibit different logical connectivity between processors than the underlying physical hardware connectivity. As a result, we observe different performance behaviors across implementations of the message-passing interface (MPI) [11, 13, 20, 29] and other collective communication libraries [27], such as varying group-to-group performance even with the same physical topology. CommBench proposes several group-to-group patterns for straightforward comparison and summary of the differences in performance between different libraries. For additional patterns that fall outside our proposed patterns, CommBench provides an API so that the developers can build custom microbenchmarks for their own communication patterns.

To evaluate CommBench, we design portable micrbenchmarks and assess the communication performance of six HPC systems with multi-GPU and multi-NIC nodes —Delta [10], Summit [37], Perlmutter [5], Frontier [9], Aurora [8] and NVIDIA's DGX-100— using MPI (multiple versions), NCCL, RCCL, and IPC capabilities. Our key finding is that the performance of libraries on hierarchical networks vary significantly across systems. Therefore, portable micro-benchmarking across systems is crucial for porting libraries and applications in a performant way. To address the problem, this paper makes the following main contributions:

- We propose group communication benchmarks for isolating performance characteristics at specific levels of the networking hierarchy.
- We present CommBench[1], a portable network micro-benchmarking framework with a flexible API for building nontrivial communication patterns and measuring their performance with various libraries.
- We explore the hierarchical communication characteristics of multi-NIC nodes and propose an analytical model for logical GPU-to-NIC topologies based on group-to-group patterns. We confirm our model with CommBench on six state-of-the-art HPC systems.

Our evaluation reveals performance characteristics by gradually increasing the load on GPU-to-NIC communications in a parameterized way. To compare the empirical results of our group-to-group communication patterns with theoretical limits, we derive analytical models for hierarchical topologies. The proposed abstractions allowed us to port i) multi-step, ii) group-to-group, and iii) application-specific microbenchmarks in significantly different hierarchical network designs.

[1]https://github.com/merthidayetoglu/CommBench

Table 1: Number of CPUs, GPUs, and NICs per node on test systems.

System	CPUs	GPUs	NICs
Delta	1 AMD EPYC	4 Nvidia A100	1 Slingshot-10
Perlmutter	1 AMD EPYC	4 Nvidia A100	4 Slingshot-11
Summit	2 IBM POWER	6 Nvida V100	2 InfiniBand
Frontier	1 AMD EPYC	4 AMD MI250x*	4 Slingshot-11
DGX-A100	2 AMD EPYC	8 Nvidia A100	8 InfiniBand
Aurora	2 Intel Xeon	6 Intel PVC*	8 Slingshot-11

*Each AMD MI250x [30] and Intel PVC [26] involves two processor dies referred to as "graphics compute dies" or "tiles", which we count as separate GPUs in the rest of the paper.

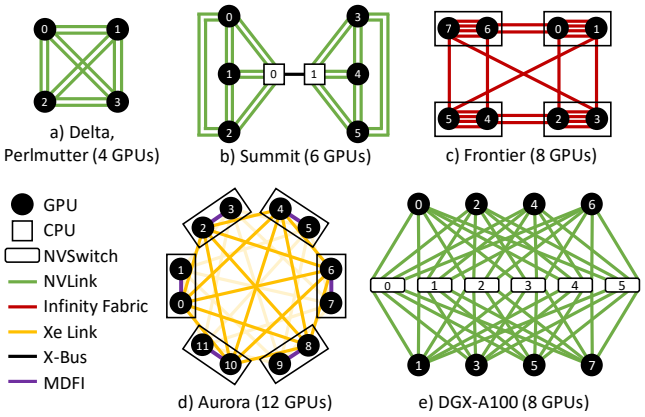

Figure 1: High-bandwidth intranode interconnects between GPUs. Nodes form uniform (e.g., (a) and (e)) and nonuniform (e.g., (b)–(d)) network interconnects with propriety links. Unidirectional bandwidth per link; NVlink and Infinity Fabric: 50 GB/s, Xe Link: 20 GB/s, X-Bus and MDFI: 64 GB/s.

2 OVERVIEW OF HIERARCHICAL NETWORKS

This section dissects the hierarchical network architecture of six current HPC systems that are summarized in Table 1.

2.1 Intra-Node Network Architecture

Modern systems involve heterogeneous node architectures with intra-node networks that are composed of i) a high-bandwidth interconnect for communication across GPUs (see Figure 1) and ii) a GPU-to-NIC interconnect for enabling communication across nodes (see Figure 2). We will first dissect the former and then the latter to understand contemporary HPC networks.

2.1.1 High-Bandwidth Links. The systems of interest (Table 1) involve various numbers of GPUs in each node. These GPUs are connected with a high-bandwidth (sub)network with propriety

links, as depicted in Figure 1. The communication bandwidth & latency across GPUs depends on the link topology, which may differ significantly across systems.

Some systems have uniform topologies such as all-to-all (Figure 1 (a) Delta and Perlmutter) or star (Figure 1 (e) DGX-A100), which are easier to understand: In uniform networks, there is no hierarchy among GPUs, i.e., they are connected to each other with the same number and type of links and hence with the same bandwidth and latency. The performance of heterogeneous interconnects is less obvious. For example, GPUs in Figure 1 (b) Summit, (c) Frontier, and (d) Aurora nodes are non-uniformly connected with different numbers and types of links. For reasoning about these networks, we consider conceptual affinity groups, discusssed next.

In hierarchical networks, communication among GPUs with a closer affinity has a lower cost than that of "distant" GPUs. The levels of affinity corresponds to the levels of the hierarchy. For example, Figure 1 (b) Summit, (c) Frontier, and (d) Aurora form two-level hierarchies with different affinity groups. In (b), the groups correspond to the half nodes, i.e., GPUs (0, 1, 2) and (3, 4, 5), where the bandwidth is higher within groups than across groups. In (c) and (d), the groups correspond to GPU pairs co-located in a single box, i.e., (0, 1), (2, 3), (4, 5), and so on. However, the boxes in (c) are connected in nonobvious ways: The number of communication links is embedded in the vendor's low-level software and not part of the public interface.

2.1.2 GPU-to-NIC Associations. When GPUs communicate across nodes, they do so through the NICs. Therefore, understanding GPU-to-NIC associations is crucial for understanding inter-node communication. Figure 2 shows the physical and logical topologies between GPUs and NICs in each node of our systems. The physical topology refers to the hardware connections between devices, often with PCIe links and switches, while the logical topology refers to the software bindings between GPUs and NICs for moving data into or out of a node. For example, we have found that communication libraries associate a single specific NIC with each GPU, even when the GPU has the same physical connection to multiple NICs.

The logical GPU-to-NIC bindings vary across systems depending on the number of GPUs and NICs and their affinity in the subnetwork. These bindings are determined by the communication library implementation (e.g., MPI or NCCL) and in our testing were found to be static[2]. In our experiments, we use the default associations: (a), (c), and (e) are packed, (d) is round-robin, and (b) and (f) are bijective, i.e., one-to-one.

2.2 Network Hierarchy Across Nodes

Communication across nodes takes place on an external network interconnect, e.g., InfiniBand and Slingshot, where each node is connected to the network through multiple NICs [7, 38]. The external network switches deliver data from the NIC associated with the sender GPU to the NIC associated with the receiving GPU. The topology of the network varies across systems, depending on their scale. For example, Summit (4,608 nodes) has a fat tree topology, where the bandwidth across nodes is uniform. To reduce the cable cost, newer systems such as Frontier (9,472 nodes) and Aurora (10,872 nodes) have three-hop dragonfly networks that introduce additional hierarchy [22]. In this work, we focus on the immediate effects of GPU-to-NIC associations in multi-node communication, and therefore we carried out our experiments up to a small number of nodes which are placed in the close vicinity by the scheduler.

[2]Except a special case that is explained in Section 5.2.1

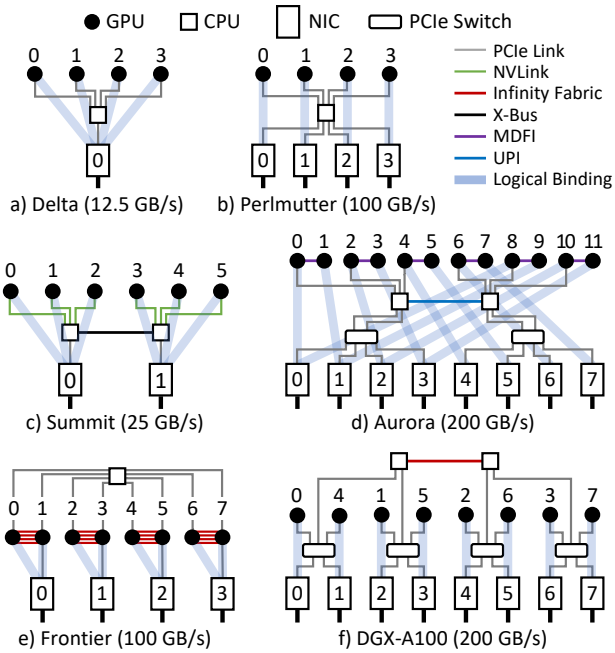

Figure 2: Interconnect between GPUs and NICs within nodes. All devices are physically connected, but each GPU uses a single NIC for P2P communicating across nodes. In our experiments, we use the default bindings as shown in (a)–(f). Machines peak bandwidth is based on the number and bandwidth of NICs per node (12.5–200 GB/s).

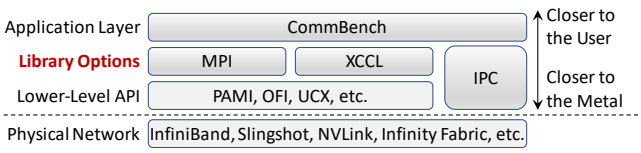

Figure 3: Overview of the communication software stack.

3 SOFTWARE

3.1 Overview

CommBench offers a system-agnostic API to build custom microbenchmarks. For ease of portability, several standard communication libraries are supported by CommBench.

3.2 Integrated Communication Libraries

CommBench is intended to measure performance as seen from an end-user application. Therefore we have integrated the most popular communication libraries (MPI, NCCL / RCCL (XCCL), and IPC) used by many applications.

For building custom microbenchmarks, CommBench relies on P2P communication functions: e.g., `MPI_Isend` / `MPI_Irecv` for MPI, and `ncclSend` / `ncclRecv` for NCCL. These functions have different GPU-aware and non-blocking protocols, and are implemented with different lower-level APIs for the networks within and across nodes. The communication software stack is deep and diverse as depicted in Figure 3, and the implementation on a specific fabric is handled by lower-level interfaces that are closer to the hardware, which are tested indirectly by CommBench.

Within nodes, libraries often use vendor-provided IPC mechanisms for message passing through the high-bandwidth links (Section 2.1.1). Nevertheless, we observe in our evaluation that higher-level library implementations are sometimes inefficient or have significant software overhead. For accurate measurements of intra-node networks, we also expose vendor-provided IPC mechanisms directly in CommBench.

3.3 Portability Across GPU Vendors

CommBench is portable across CPUs and GPUs from multiple vendors: there are OneAPI, CUDA, and HIP versions for programming GPUs of Intel, Nvidia, and AMD, respectively. The choice of port and whether CommBench uses CPU or GPU communication are both made at compile-time. According to the selection of library and port, CommBench implements each P2P communication with one of the nine capabilities listed in Table 2.

Table 2: Integrated libraries (columns) and ports (rows).

	OneAPI	CUDA	HIP	(default)
MPI	GPU MPI	GPU MPI	GPU MPI	CPU MPI
XCCL	OneCCL[3]	NCCL	RCCL	n/a
IPC	ZE IPC	CUDA IPC	HIP IPC	n/a

4 MICROBENCHMARK IMPLEMENTATION

CommBench is designed around an API for composing communication patterns succinctly. A microbencmark is composed of P2P communications. There may be a single step with concurrent communications, or multiple steps, where each step depends on the previous one. Listing 1 outlines the CommBench API for constructing a single step.

4.1 CommBench API

Each benchmarking step requires three things: 1) a persistent communicator that memoizes and executes the desired communication pattern, 2) creation of the communication pattern using individual P2P communications, and 3) validation through isolated measurements.

The persistent communicator is realized with the `Comm` object defined in Listing 1, Line 5. The communication registry is made by the `add` function on Line 7. The intended use case for CommBench is for the user to supply the desired communication pattern and then call `start` (Line 9), which kicks off all registered communications at once, maximizing usage of the machine's bandwidth. The `start` call is nonblocking and all buffers supplied to CommBench could be in use until the corresponding `wait` (Line 11) call completes. After the `wait` call, the buffers can be safely reused. For measuring the time of a step, we provide an integrated `measure` API (Line 13) which executes the communications multiple times and reports the minimum, maximum, average, and median times over a specified number of iterations.

We use the CommBench API for implementing the following microbenchmarks.

[3]OneCCL currently does not support non-blocking P2P functions and therefore not applicable to the results in this paper.

Listing 1: API for registering each microbenchmark step.

```
1  // Data type is templatised as T
2  template typename<T>
3  class Comm {
4    // Create a benchmark step with a library of choice.
5    Comm(Library);
6    // Register a P2P communication into the step.
7    void add(T *sendbuf, T *recvbuf, size_t count, int sendid, int
       recvid);
8    // Launch the registered communications and return.
9    void start();
10   // Block until the completion of communications.
11   void wait();
12   // Measure over many iterations and report statistics.
13   void measure(int warmup, int numiter);
14 }
```

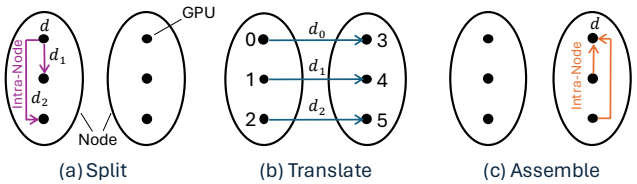

Figure 4: Striping of P2P data across GPUs for maximizing the bandwidth across nodes. It takes three steps to a) split the original data d into three stripes—d_0, d_1, d_2, b) translate the stripes across nodes using all GPUs, and c) assemble of the original data at the receiving GPU.

4.2 Striping Data Across Nodes

The point-to-point functions of libraries utilize only a single NIC, although there are multiple NICs per node in current systems. Therefore, conventional point-to-point benchmarks do not measure the full potential bandwidth across nodes. We propose a striping microbenchmark for measuring the point-to-point bandwidth across multi-NIC nodes.

This microbenchmark consists of three consecutive steps depicted in Figures 4 (a)–(c). To accommodate any node type, the code is parameterized for any message size (d) and number of GPUs per node ($g = 3$). The communication steps are programmed separately in Listing 2. In Lines 2–4, the implementation library for each step is selected (IPC within nodes and NCCL across nodes). Lines 8–17 register the three communication patterns. Line 19 creates a vector of communicators that represents the communication sequence, where each step depends on the previous one. The `measure_async` function in Line 21 executes the communication steps asynchronously while preserving the data dependencies across steps, which we explain next.

Listing 2: Program for the striping microbenchmark.

```
// Create three steps and choose implementation libraries
Comm<T> split(IPC);
Comm<T> translate(NCCL);
Comm<T> assemble(IPC);
// Allocate temporary buffer for staging data.
T *temp;
allocate(temp, d/g);
// Register the split step (g-1 intra-node P2P).
for (int i = 1; i < g; i++)
  split.add(sendbuf+i*d/g, temp, d/g, 0, i);
// Register the translate step (g inter-node P2P).
translate.add(sendbuf+i*d/g, recvbuf, d/g, 0, g);
for (int i = 1; i < g; i++)
  translate.add(temp, temp, d/g, i, g + i);
// Register the assemble step (g-1 intra-node P2P).
for (int i = 1; i < g; i++)
  assemble.add(temp, recvbuf+i*d/g, d/g, g+i, g);
// Setup microbenchmark sequence with three steps.
vector<Bench<T>> stripe = {split, translate, assemble};
// Warmup over 10 rounds and measure over 20 rounds.
measure_async(stripe, 10, 20, d);
```

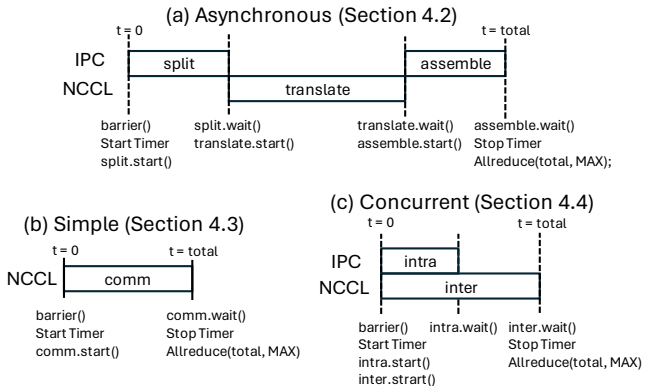

Figure 5: Synchronization schemes for scheduling steps of the proposed microbenchmarks from a process' perspective. The horizontal axis represents time and each box corresponds to a communication step. The vertical dashed lines show the earliest moment of return to the indicated functions on each GPU. For accurate measurements, the `start` and `wait` functions must be called from all processes in the shown order. The end-to-end time is the maximum of total time taken on all processes.

In multi-step microbenchmarks, we assume each step depends on its predecessor. A naive way of preserving such dependencies is a lock-step global execution with a barrier (global synchronization) between each step. However, including the barriers would not only cause idle time, and hence inefficiency, but also yield inaccurate latency measurements, especially on large numbers of nodes. CommBench does not use global synchronization in its execution. Instead, it uses finer-level synchronization functions (`start` and `wait`) that are explained in Section 4.1. Figure 5 (a) depicts such a communication sequence for the striping example. To preserve data dependencies, a GPU must wait for completion of a current step before starting the next step. In this case, a GPU waits only if it has an outstanding dependency on another GPU in the current step for preserving the data dependencies across steps. If there is no dependency, the GPU moves on to execute the subsequent step.

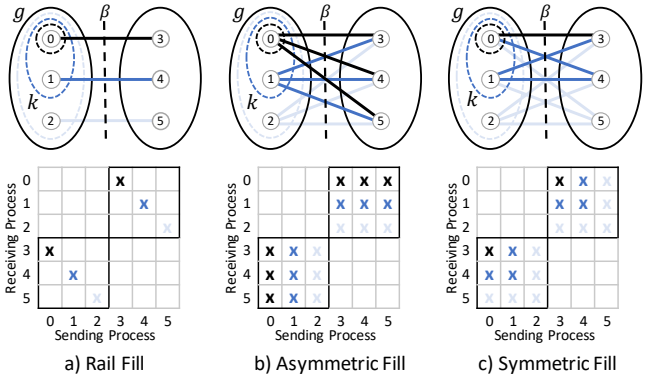

Figure 6: The (a) rail, (b) asymmetric, and (c) symmetric pattern families across two groups with (top) bipartite graph and (bottom) sparse communication matrix representations. The configuration parameters (n, g, k) are $n = 2$, $g = 3$ and k is 1 (black), 2 (blue), or 3 (light blue).

4.3 Group Communication Patterns

We propose group communication patterns to measure the performance across groups of processors at a specific level of the communication hierarchy in isolation.

Group communication patterns are useful for stressing the network across a set of processors at a specific level of the communication hierarchy, such as a node. We define three families of built-in patterns for varying the communication workload gradually across two nodes rail, asymmetric, and symmetric (Figure 6). Then, we design scaling patterns for multiple nodes with various directions of data movement. Group communication patterns reveal the effect of 1) static and dynamic (if any) associations between GPUs and NICs, 2) hardware limits in isolation (e.g., switch, link, NIC), and 3) the software overhead of libraries (e.g., MPI, NCCL).

4.3.1 Pattern Parameterization. We propose the parameterized group-to-group patterns as shown in Figure 6 (a)–(c). These are bipartite patterns between GPUs in different groups. The configuration parameters are g and n, where g is the node size in terms of number of GPUs, and n is the number of nodes.

Figures 6 (a)–(c) show the communications across groups with parameters $g = 3$ and $n = 2$ for the (a) *rail*, (b) *asymmetric*, and (c) *symmetric* group-to-group patterns.[4] To provide more diversity of patterns, we define an additional parameter $k \leq g$ that represents a *subgroup* within a group. Figure 6 shows the family of patterns for varying k.

The rail pattern generalizes the P2P pattern between GPUs in corresponding positions of two or more nodes. Selecting $k = 1$ for the rail pattern recovers exactly an internode P2P pattern. By choosing $k = 1, 2, 3$ in this example, the rail pattern tests the capacity

[4]Note that for $g = 3$ the groups are selected to be the GPUs on a single node in this example. In general, groups are selected so that the GPUs within a group have the closest possible affinity.

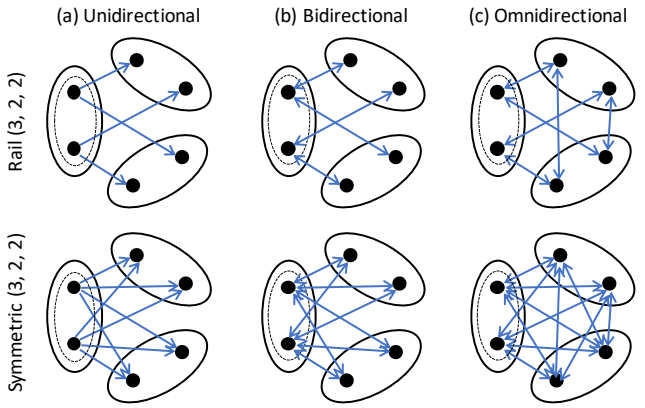

Figure 7: (a) Unidirectional, (b) bidirectional, and (c) omnidirectional data movement across multiple groups with rail (3, 2, 2) and symmetric (3, 2, 2) patterns.

of one-to-one communication between two nodes with different numbers of simultaneously participating pairs of GPUs.

The asymmetric pattern maps k GPUs in the first group to all g GPUs in the other group. For example, when $k = 1$ the pattern is equivalent to a one-to-g pattern, where the sender and receiver GPUs are in different nodes. By increasing k, we activate GPUs incrementally to increase the workload. When $k = g$, the asymmetric pattern converges to the symmetric pattern shown in Figure 6 (b)–(c).

In the asymmetric pattern, the parameter k is only used to limit the number of GPUs in the first group participating in the communication. However, for the rail family and the symmetric family, k is used to limit the number of GPUs in both nodes participating in the communication.

The bottom of Figure 6 shows the communication matrix corresponding to the group communication pattern above, where each entry corresponds to a P2P communication originating from the sending process to the receiving process. The off-diagonal blocks show the inter-node communication pattern. These patterns are registered into a single communicator and executed as depicted in Figure 5 (b).

4.3.2 Direction of Data Movement. A further refinement is to consider the direction of data movement. We consider (a) *unidirectional*, (b) *bidirectional*, and (c) *omnidirectional* communication patterns across multiple groups as seen in Figure 7 with our rail and symmetric patterns. The unidirectional patterns assume that there is a primary group that sends data to all the rest of the groups. The bidirectional pattern is the same as the unidirectional pattern except that the communications are in both directions. Omnidirectional communication captures patterns patterns where all groups communicate with all other groups, rather than having one group that communicates either unidirectionally or bidirectionally with the other groups. Since the asymmetric pattern is defined only on two groups, there is no sensible definition of the omnidirectional asymmetric pattern. For the rest of the paper, a group communication pattern is described by the three parameters (n, g, k) and a direction.

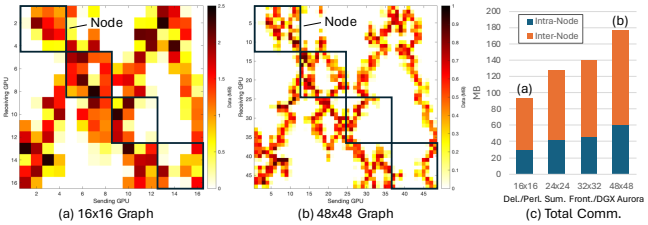

Figure 8: Application-specific (MemXCT) communication patterns for (a) 16 GPUs and (b) 48 GPUs. The individual message sizes shrink whereas (c) the total data movement grows with the number of GPUs. We replicated this pattern for microbenchmarking on four nodes of each system.

4.4 Application Case Study

Many applications involve irregular communications, where each GPU communicates with a sparse subset of GPUs and the message lengths vary. Our final custom microbenchmark replicates irregular communication patterns from an application, MemXCT [17], as a complement to the regular communication patterns discussed above.

This microbenchmark is composed of concurrent P2P communications across GPUs as a result of a distributed sparse matrix multiplication. The communication pattern across GPUs depend on the sparsity pattern of the matrix and it's partitioning. For microbenchmarking, we chose the ADS4 dataset given in the application repository[5]. We extracted communication patterns for four nodes of each system which corresponds to 16, 24, 32, and 48 GPUs. The resulting patterns across 16 and 48 GPUs are shown in Figure 8 (a) and (b), respectively.

Internally, CommBench stores a distributed sparse matrix that tracks the communication pattern as it is being created. Invocation of each `add` function corresponds to adding an entry to the sparse communication matrix. This matrix does not store any communication beyond metadata needed to track the data dependencies across GPUs.

For separate measurements within and across nodes, we register the P2P communications into separate communicators. Nevertheless, these steps are independent of each other, and therefore can be run concurrently to hide one behind another. Concurrent execution is expressed using the synchronization functions as depicted in Figure 5 (c). The concurrent execution waits for completion of whichever step takes the most time—inter-node communication in this case as seen in Figure 8 (c).

5 EVALUATION

To cover a wide variety of contemporary communication architectures, we perform experiments on the six systems discussed in Section 2. Our experiments use the default software versions installed on each system; the specific version will be listd in the artifact description appendix. For MPI, Summit uses Spectrum, Delta and DGX-A100 use OpenMPI, and rest of the systems use vendor-modified MPICH implementations. To place an MPI rank r on a GPU, we place it on node $\lfloor r/g \rfloor$ and on GPU with index $(r \bmod g)$ as

[5]Application code: https://github.com/merthidayetoglu/MemXCT-GPU

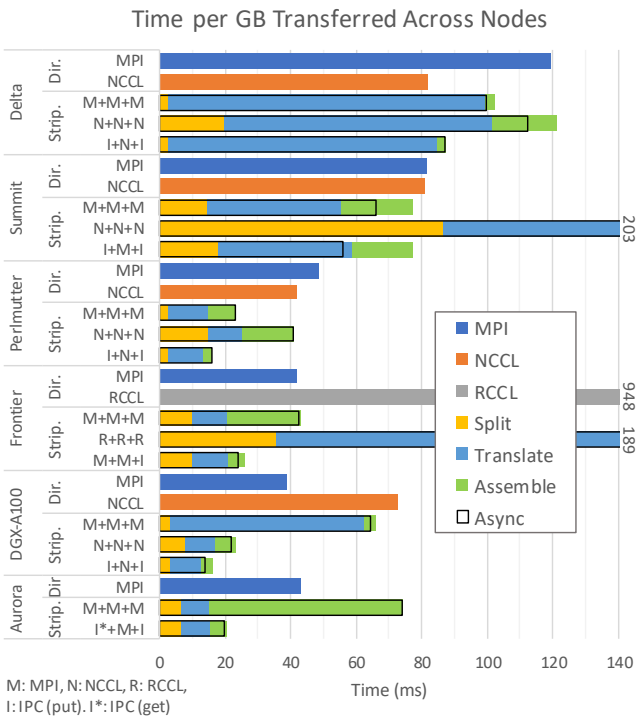

Figure 9: Time for moving one GB from GPU to GPU across two nodes of six systems. Striping utilizes multiple NICs with three steps: split (intra-node), translate (across nodes), and assemble (intra-node). We obtain the optimal performance with a mix of libraries across steps, e.g., I+N+I means that we use IPC within nodes and NCCL across nodes. As a baseline, we also report the direct (unstriped) P2P functions of MPI, NCCL, and RCCL across nodes.

shown in Figure 1, where solid black circles are GPUs and the numbers are $(r \bmod g)$. We worked with facility staff and administrators of these systems to ensure we used the best available configurations for our benchmarks.

5.1 Striping Data Movement Across Nodes

We first run the striping (Section 4.2) microbenchmark on all systems for testing available libraries within and across nodes. Our results are summarized in Figure 9. We observe four common behaviors across systems:

(1) Mixed-library implementations improve over uniform implementations in all cases of striped data movement. The uniform implementations with MPI (M+M+M), NCCL (N+N+N), and RCCL (R+R+R) are not efficient on at least one level of the network hierarchy. The optimal mixture of libraries uses the most efficient protocol for each level and achieves a (geometric) average 2.17× speedup.

(2) Asynchronous communication improves the performance of striping. The stacked bars represent the minimum time for each communication step in isolation, and the hollow black frames represent the end-to-end time of multiple steps. CommBench executes the steps back-to-back asynchronously, i.e., without any barrier, while respecting point-to-point data dependencies across steps. As a result, asynchronous is up to 20% faster and does not yield a slowdown in any case.

(3) The P2P implementations of NCCL are anamolously slow within nodes when multiple nodes are involved in a communication. This problem does not arise with MPI, IPC, or on a single-node NCCL execution. Therefore, on average, MPI and IPC implementations are 4.78× faster than NCCL within nodes. We also found that the RCCL implementation is not performant. We have confirmed that RCCL implements a TCP protocol with Slingshot-11 NICs that underutilizes the high-bandwidth fabrics that connect GPUs and nodes (see Section 5.2.2).

(4) MPICH uses get protocol in one-sided IPC communications within nodes. Subscribing them to a single stream or copy engine on the receiving GPU causes serialization in the assembly step. CommBench's IPC implementation uses put protocol by default, initiating the assembly step from multiple GPUs and obtains substantial speedup (e.g., 11× on Aurora). We also incorporated get protocol to overcome serialization on the split step on Aurora.

The striping microbenchmark exposes other inefficiencies, or at least surprising asymmetries, in communication software. For example, the split and assemble steps have symmetric-opposite patterns as seen in Figure 4: split moves data from one GPU to other GPUs, while assemble moves data from other GPUs to one GPU within a node. Despite using the same communication links, they obtain different bandwidths, most significantly on Perlmutter, Frontier, and Aurora, with up to a 11× difference.

We have verified all findings independent of CommBench to assure that our tool does not introduce any significant artifacts. We cannot speculate as to causes or solutions as we do not have full access to the library implementations. The main point of our microbenchmarks is to expose the performance characteristics so that system administrators and application developers will be aware of them.

5.2 Group Communications Across Nodes

We characterize the multi-NIC performance with our group-to-group patterns—specifically using the rail and asymmetric families as shown in Figure 6 (a)–(b), respectively—across two physical nodes. We characterize the multi-NIC utilization by varying the subgroup size k (see Section 4.3.1) and model the bandwidth in terms of the number of NICs involved in the communication across nodes.

Our evaluation, Figure 10–11, shows the bandwidth and latency across nodes when we set the following values for (n, g, k): $(2, 4, k)$ for Delta and Perlmutter, $(2, 6, k)$ for Summit, $(2, 8, k)$ for Frontier, and DGX-A100, and $(2, 12, k)$ for Aurora. Since we do not have direct control over NICs with the high-level communication libraries that we test, we vary the number k of GPUs to empirically determine their logical bindings. We first present the bandwidth results with large messages (larger than 16 MB) in Figure 10 and then latency results with small messages (4 bytes) in Figure 11.

5.2.1 Modeling Bandwidth Across Nodes. We use our group-to-group patterns to characterize the GPU-to-NIC behavior. On our

Figure 10: Bisection bandwidth profiles across two nodes were measured with blue (CPU-Only MPI), orange (GPU-Aware MPI), yellow (NCCL), and purple (RCCL) bars. Hollow bars show the proposed model in Equations (1)–(2). Group-to-group patterns gradually change the workload across nodes to expose hardware differences across systems, testing libraries' performance portability and helping developers make choices for moving their applications across systems.

test systems, we observe that GPUs are assigned to NICs with packed and round-robin schemes. We can model the bandwidth across nodes with packed scheme as:

$$f_{\text{packed}} = f_{\text{NIC}} \left(1 + \frac{\text{ReLU}(k-p)}{p}\right), \tag{1}$$

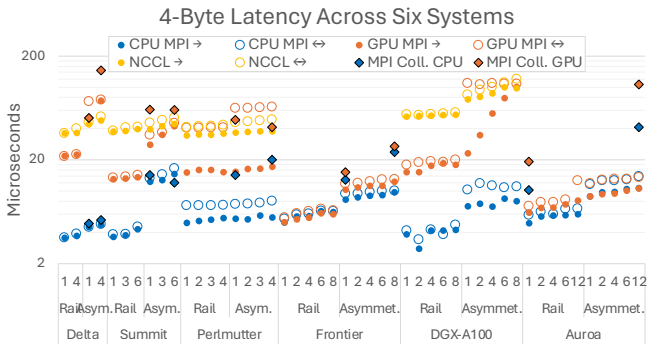

Figure 11: Group-to-group latency across two nodes with the rail and asymmetric patterns. The horizontal index represents the variable k. The comparisons with corresponding MPI collective functions are marked with diamonds.

where k is the active GPUs, p is the number of GPU assigned to each NIC, f_{NIC} is the theoretical NIC bandwidth, and ReLU is the rectified linear unit activation function, i.e., $\text{ReLU}(x) = \max\{0, x\}$. We confirm the proposed model empirical results in Figure 10 (a)–(e). On the other hand, (f) Aurora employs a round-robin scheme that can be modeled as

$$f_{\text{round-robin}} = f_{\text{NIC}} \frac{k}{\lceil k/r \rceil}, \tag{2}$$

where r in this case is the number of active NICs in each node. We activate all (eight) NICs per node, resulting in the bandwidth profile in Figure 10 (f).

The models given in (1) and (2) assume the static GPU-to-NIC associations in Figure 2 (a)–(f). The static models break down if the associations are determined dynamically, as in (f) DGX-A100 when we enable dynamic behaviour with NCCL's hardware-specific plugin, where the logical topology changes depending on the workload. The speedup is provided by Mellanox propriety software (SHARP [12]); we were able to expose this behaviour using the unidirectional asymmetric group-to-group patterns.

5.2.2 Underperforming Cases. The measurements in Figure 10 quantify the performance of the libraries relative to our models of peak performance. The figure also exposes some severely underperforming cases that we suggest developers avoid.

The first case (Figure 10 (d), purple bars that are barely seen) is the RCCL library, which is a port of NCCL for AMD GPUs on Frontier. As mentioned earlier, we confirmed that the RCCL does not have native implementation for Slingshot-11 NICs and therefore it falls back to the TCP protocol. Hence, RCCL gives the correct result but with peak bandwidth of no more than a few GB/s and high latency.

The other underperforming case is GPU-aware MPI on DGX-A100 (Figure 10 (e), orange bars). We confirmed with the system admin of this particular machine that it was configured for machine learning frameworks such as PyTorch, which relies on NCCL and hence the GPU-aware MPI is not tuned for multiple NICs. CommBench makes it possible to reliably detect and isolate such configuration issues.

Table 3: Utilization of rated GPU memory bandwidth for self-communication of 1 GB.

	Del.	Sum.	Perl.	Front.	DGX	Aur.
MPI	85%	11%	0.5%	0.3%	0.2%	0.2%
XCCL*	4.3%	15%	28%	1.7%	4.3%	n/a
IPC	85%	88%	66%	77%	85%	66%

*RCCL for Frontier, n/a for Aurora (see footnote 3), NCCL otherwise.

5.2.3 CPU vs. GPU Performance. In Figures 10–11, we observe a general trend that CPU-to-CPU communication across nodes with MPI has higher bandwidth and lower latency (shown in blue) than those of GPU-Aware MPI communication (shown in orange) on machines where the CPUs are directly connected to NICs. The exceptions are Frontier and DGX-A100, where GPUs are directly connected to NICs, as seen in Figure 2 (e)–(f). As a result GPUs on these systems obtain a higher communication bandwidth than that of CPUs as seen in Figure 10 (d)–(e), respectively.

5.2.4 MPI vs. NCCL Performance. When available, NCCL usually obtains higher bandwidth than MPI on GPUs, nevertheless, we observe a higher latency with NCCL, compared to with MPI. The latency of NCCL is more than 40 microseconds across systems as shown with the yellow marks in Figure 11, which is significantly higher than MPI, which is approximately 6 microseconds on Frontier and 8 microseconds on Aurora. On the other hand, RCCL's TCP implementation has about 12 ms latency on Frontier (not plotted in the figure). These observations matches with the existing literature [35].

5.2.5 Group-to-Group vs. MPI Collectives. Group-to-group collective patterns in Figure 11 measure the lower bound for the MPI collectives, because the group benchmarks measure the portion of time across nodes only, whereas MPI benchmarks measure the end-to-end collective functions, i.e., both within nodes and across nodes.

To validate the lower-bound property of our benchmark, we compare our group-to-group benchmarks with traditional MPI collective functions. We compare `MPI_Scatter` and `MPI_Alltoall` (represented with diamond marks) with our asymmetric family patterns with unidirectional ($k = 1$) and bidirectional ($k = g$) patterns, respectively. The group-to-group benchmarks characterize the network performance more accurately than MPI collective functions, because the former only measure the communication across the targeted interconnect (across nodes) whereas the latter performs additional (intra-node) communications.

5.3 Self Communication

The P2P bandwidth for self communication (i.e., from one buffer to another within the same GPU's memory) is supposed to be comparable with the processor's memory bandwidth[6]. However, we occasionally observe significantly lower bandwidth than expected. Table 3 shows the utilization of the rated GPU memory bandwidth with different libraries.

[6]Theoretical GPU memory bandwidths are; Summit: 900 GB/s, Delta, Perlmutter, and DGX-A100: 1.55 TB/s, Frontier & Aurora: 1.64 TB/s.

5.4 Scaling of Group Communications

We use CommBench on Frontier and Aurora using the rail and symmetric patterns on multiple nodes. We use configurations (n, 8, 8) for (a) Frontier and (n, 12, 12) for (b) Aurora and stress the external network by increasing n up to eight nodes, employing 64 and 96 GPUs, respectively, as shown in Figure 12. We employ MPI for measuring the bandwidth on both CPUs and GPUs.

On both systems, the bidirectional rail pattern achieves the highest bandwidth, approximately 175 GB/s/node on Frontier's GPUs (orange) and 275 GB/s/node on Aurora's CPUs (blue) in Figure 12. As a result, both systems utilize 90% of the theoretical bandwidth, although Frontier with GPU and Aurora with CPU due to their direct connections to NICs. The symmetric pattern obtains greater bandwidth when a higher number of nodes participate in communication due to a better saturation of the overall network. On the other hand, omnidirectional bandwidth shows the opposite due to the contention on switches that connect the nodes.

5.5 Application-Specific Microbenchmark

We introduce a case study in Section 4.4. In this application-specific microbenchmark, the communication pattern is irregular and intra- and inter-node portions are first measured in isolation and then the intra- and inter-node communications are run concurrently. The results are shown in Figure 13. In concurrent execution, the intra-node cost is hidden behind the communication across nodes in all cases except Aurora: Communications within and across nodes slow each other down, although they occur on separate networks.

We use an MPI collective function (`MPI_Alltoallv`) as a sanity check for this microbenchmark. The collective function expresses a nonuniform all-to-all communications such as this application-specific communication graph. NCCL has no equivalent collective function, which means users must implement such nonuniform patterns themselves if they wish to use NCCL. CommBench makes it much easier to write such application-specific patterns, particularly

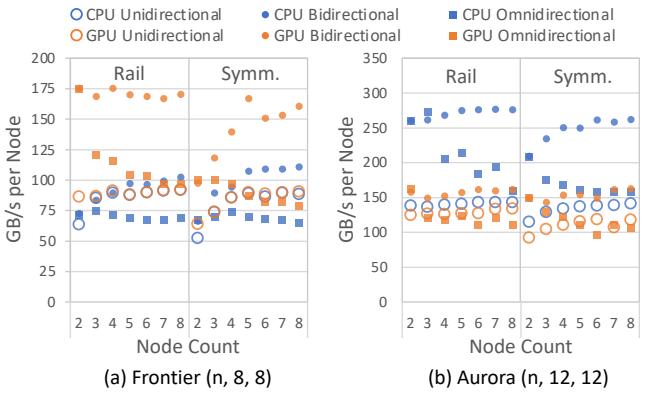

Figure 12: Bandwidth per node with unidirectional (hollow circles), bidirectional (solid circles), and omnidirectional (solid squares) scaling patterns (shown in Figure 7) across multiple nodes of (a) Frontier and (b) Aurora. Blue benchmarks represents CPU-Only MPI and orange benchmarks represents GPU-Aware MPI measurements.

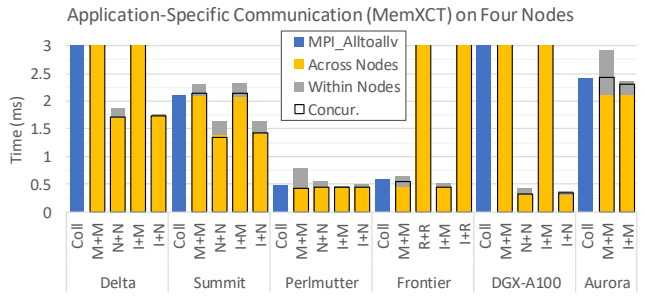

Figure 13: Time spent for application-specific microbenchmark on four nodes of each system (lower is better). CommBench executes different libraries within and across nodes concurrently, detects underperformed cases, and exposes performance portability across systems.

if one wants to exploit combinations of different libraries for the best performance.

6 RELATED WORK

There are multiple previous efforts on benchmarking HPC networks [2, 6, 21, 25, 34]. To the best of our knowledge, none consider group communication patterns that characterize multi-GPU, multi-NIC behavior with multiple libraries.

Prior research has explored bandwidth saturation with respect to the number of processors in CPU-based systems [14, 15, 18]. We follow a similar approach, focusing on hierarchical systems with multiple GPUs and NICs to saturate the bandwidth within and across nodes. We also explore the logical topology between GPUs and NICs.

Previous work has investigated understanding and modeling inter-GPU communication in large-scale HPC systems, examining data movement variations between multiple GPUs [4] and irregular P2P communications [24]. However, these studies primarily focused on MPI as the sole communication layer and relied heavily on CPU involvement. Furthermore, characterizing interconnect heterogeneity [30] has mostly targeted single systems, and microbenchmarks [31] exploring transfer behavior across data placements have been limited to CUDA primitives. CommBench introduces group-to-group patterns and empirically tests them on a variety of HPC systems while offering the flexibility to employ different communication libraries.

coNCePTual [28] is a DSL for designing benchmarks that stress communication layers, focusing on fine-grain control over application properties such as buffer lifetimes. coNCePTual does not focus on support for collective, hierarchical communications, nor does it attempt to elucidate the performance characteristics of hierarchical networks.

The current practice of system administrators and users is to run standard benchmarks provided by the MPI and NCCL distributions, such as MVAPICH benchmark (OSU Benchmarks [1]) and NCCL tests. These tests report the performance of P2P and collective communications that are offered by optimized collectives within communication layers [19, 32, 36], but lack an API for user-defined, application-specific communication patterns. We address this limitation by providing the CommBench API, enabling users to easily express and benchmark desired communication patterns across various communication layers.

7 CONCLUSION

Contemporary HPC systems comprise fat nodes with multiple GPUs and NICs that form complex network hierarchies that traditional collective benchmarks do not adequately characterize. To understand the performance of multilevel networks, we propose extended group-to-group benchmarking patterns to target specific levels of the network hierarchy. We implement these patterns with CommBench, a framework for composing and benchmarking user-defined communication patterns with multiple GPU communication libraries. We evaluate CommBench on six state-of-the-art systems. Our benchmarks reveal the performance characteristics of these systems; for example, we identified three multi-NIC scaling behaviors in packed, round-robin, and dynamic schemes, exposing the logical binding of GPUs to NICs that is not normally visible to the user. Depending on the system, library choice, and underlying group-to-group pattern, we saturate between 50%–90% of the theoretical bandwidth available in each configuration. Since we can stress specific communication channels in our approach, we consistently measure higher bandwidth (up to 30%) and lower latency (up to 3×) with group-to-group patterns compared to traditional collective patterns. CommBench's portability and flexibility make benchmarking of modern communication networks more comprehensive, more detailed, and easier.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was done on a pre-production supercomputer with early versions of the Aurora software development kit. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357. This research used the Delta advanced computing and data resource which is supported by the National Science Foundation (award OAC 2005572) and the State of Illinois. Delta is a joint effort of the University of Illinois Urbana-Champaign and its National Center for Supercomputing Applications. This work is partially supported by Laboratory Directed Research and Development (Project Number: 2023-0104) funding from Argonne National Laboratory. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research used resources of the National

Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility using NERSC award ASCR-ERCAP0029675.

REFERENCES

[1] [n. d.]. https://mvapich.cse.ohio-state.edu/benchmarks/

[2] Brian W Barrett and K Scott Hemmert. 2009. An application based MPI message throughput benchmark. In *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 1–8.

[3] Amanda Bienz, William D Gropp, and Luke N Olson. 2020. Reducing communication in algebraic multigrid with multi-step node aware communication. *The International Journal of High Performance Computing Applications* 34, 5 (2020), 547–561.

[4] Amanda Bienz, Luke N Olson, William D Gropp, and Shelby Lockhart. 2021. Modeling data movement performance on heterogeneous architectures. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[5] National Energy Research Scientific Computing Center. 2022. Perlmutter Architecture. https://docs.nersc.gov/systems/perlmutter/architecture/

[6] Sudheer Chunduri, Taylor Groves, Peter Mendygral, Brian Austin, Jacob Balma, et al. 2019. GPCNeT: Designing a benchmark suite for inducing and measuring contention in HPC networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–33.

[7] Daniele De Sensi, Salvatore Di Girolamo, Kim H McMahon, Duncan Roweth, and Torsten Hoefler. 2020. An in-depth analysis of the slingshot interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[8] Argonne Leadership Computing Facility. 2022. Aurora. https://www.alcf.anl.gov/support-center/aurora-sunspot

[9] Oak Ridge Leadership Computing Facility. 2022. Frontier User Guide - System Overview. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#id2

[10] National Center for Supercomputing Applications. 2022. Delta. https://www.ncsa.illinois.edu/research/project-highlights/delta/

[11] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, et al. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 97–104.

[12] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, et al. 2016. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, 1–10.

[13] William Gropp and Ewing Lusk. 1996. User's Guide for MPICH, a Portable Implementation of MPI.

[14] William Gropp, Luke N. Olson, and Philipp Samfass. 2016. Modeling MPI Communication Performance on SMP Nodes: Is It Time to Retire the Ping Pong Test. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI 2016)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/2966884.2966919

[15] William D Gropp. 2019. Using node and socket information to implement MPI Cartesian topologies. *Parallel Comput.* 85 (2019), 98–108.

[16] Mert Hidayetoğlu, Tekin Bicer, Simon Garcia De Gonzalo, Bin Ren, Vincent De Andrade, et al. 2020. Petascale XCT: 3D image reconstruction with hierarchical communications on multi-GPU nodes. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[17] Mert Hidayetoğlu, Tekin Biçer, Simon Garcia de Gonzalo, Bin Ren, Doğa Gürsoy, et al. 2021. MemXCT: design, optimization, scaling, and reproducibility of x-ray tomography imaging. *IEEE Transactions on Parallel and Distributed Systems* 33, 9 (2021), 2014–2031.

[18] Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Netgauge: A network performance measurement framework. In *High Performance Computing and Communications: Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007. Proceedings 3*. Springer, 659–671.

[19] Ben Huang, Michael Bauer, and Michael Katchabaw. 2005. Hpcbench-a Linux-based network benchmark for high performance networks. In *19th international symposium on high performance computing systems and applications (HPCS'05)*. IEEE, 65–71.

[20] IBM. 2017. IBM Spectrum MPI - Overview. https://www.ibm.com/products/spectrum-mpi

[21] Nikhil Jain, Abhinav Bhatele, Sam White, Todd Gamblin, and Laxmikant V Kale. 2016. Evaluating HPC networks via simulation of parallel workloads. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 154–165.

[22] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. 2008. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 77–88.

[23] Shelby Lockhart, Amanda Bienz, William Gropp, and Luke Olson. 2022. Performance analysis and optimal node-aware communication for enlarged conjugate gradient methods. *ACM Transactions on Parallel Computing* (2022).

[24] Shelby Lockhart, Amanda Bienz, William D Gropp, and Luke N Olson. 2022. Characterizing the Performance of Node-Aware Strategies for Irregular Point-to-Point Communication on Heterogeneous Architectures. *arXiv:2209.06141* (2022).

[25] Adam Moody. 2009. *Contention-free routing for shift-based communication in MPI applications on large-scale InfiniBand clusters*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[26] Samuel K. Moore. 2022. Behind Intel's HPC chip that will break the exascale barrier. https://spectrum.ieee.org/intel-s-exascale-supercomputer-chip-is-a-master-class-in-3d-integration

[27] NVIDIA. 2016. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl

[28] Scott Pakin. 2007. The Design and Implementation of a Domain-Specific Language for Network Performance Testing. *IEEE Transactions on Parallel and Distributed Systems* 18, 10 (2007), 1436–1449.

[29] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. 2021. The MVAPICH project: Transforming research into high-performance MPI library for HPC community. *Journal of Computational Science* 52 (2021), 101208.

[30] Carl Pearson. 2023. Interconnect Bandwidth Heterogeneity on AMD MI250x and Infinity Fabric. *arXiv:2302.14827* (2023).

[31] Carl Pearson, Abdul Dakkak, Sarah Hashash, Cheng Li, I-Hsin Chung, et al. 2019. Evaluating characteristics of CUDA communication primitives on high-bandwidth interconnects. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 209–218.

[32] Ralf Reussner, Peter Sanders, and Jesper Larsson Träff. 2002. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming* 10, 1 (2002), 55–65.

[33] Piyush Sao, Ramakrishnan Kannan, Xiaoye Sherry Li, and Richard Vuduc. 2019. A communication-avoiding 3D sparse triangular solver. In *Proceedings of the ACM International Conference on Supercomputing*. 127–137.

[34] Mohak Shroff and Robert A Van De Geijn. 1999. CollMark: MPI collective communication benchmark. In *International Conference on Supercomputing*, Vol. 2000. 10.

[35] Christopher M Siefert, Carl Pearson, Stephen L Olivier, Andrey Prokopenko, Jonathan Hu, et al. 2023. Latency and Bandwidth Microbenchmarks of US Department of Energy Systems in the June 2023 Top 500 List. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 1298–1305.

[36] Muhammet Abdullah Soytürk, Palwisha Akhtar, Erhan Tezcan, and Didem Unat. 2022. Monitoring collective communication among GPUs. In *Euro-Par 2021: Parallel Processing Workshops: Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers*. Springer, 41–52.

[37] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, et al. 2018. The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 661–672. https://doi.org/10.1109/SC.2018.00055

[38] Christopher Zimmer, Scott Atchley, Ramesh Pankajakshan, Brian E Smith, Ian Karlin, et al. 2019. An evaluation of the CORAL interconnects. In *SC19: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–18.