Weingram A, Li Y, Qi H *et al.* xCCL: A survey of industry-led collective communication libraries for deep learning. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 38(1): 166-195 Jan. 2023. DOI: 10.1007/s11390-023-2894-6

xCCL: A Survey of Industry-Led Collective Communication Libraries for Deep Learning

Adam Weingram, Yuke Li (李雨珂), *Student Member, ACM*, Hao Qi (戚　昊), Darren Ng Liuyao Dai (代柳瑶), and Xiaoyi Lu (鲁小亿), *Member, ACM, IEEE*

Department of Computer Science and Engineering, University of California, Merced, Merced 95343, U.S.A.

E-mail: aweingram@ucmerced.edu; yli304@ucmerced.edu; hqi6@ucmerced.edu; dng350@ucmerced.edu
ldai8@ucmerced.edu; xiaoyi.lu@ucmerced.edu

Received October 8, 2022; accepted January 3, 2023.

Abstract Machine learning techniques have become ubiquitous both in industry and academic applications. Increasing model sizes and training data volumes necessitate fast and efficient distributed training approaches. Collective communications greatly simplify inter- and intra-node data transfer and are an essential part of the distributed training process as information such as gradients must be shared between processing nodes. In this paper, we survey the current state-of-the-art collective communication libraries (namely xCCL, including NCCL, oneCCL, RCCL, MSCCL, ACCL, and Gloo), with a focus on the industry-led ones for deep learning workloads. We investigate the design features of these xCCLs, discuss their use cases in the industry deep learning workloads, compare their performance with industry-made benchmarks (i.e., NCCL Tests and PARAM), and discuss key take-aways and interesting observations. We believe our survey sheds light on potential research directions of future designs for xCCLs.

Keywords collective, deep learning, distributed training, GPUDirect, RDMA (remote direct memory access)

1 Introduction

Designing high-performance communication subsystems is one of the most challenging tasks essential to achieving scalable parallel computing goals[1] as the communication performance can directly influence the execution efficiency of large-scale distributed software. Collectives are a form of organized communication that has become ubiquitous in parallel computing, distributed computing, and high-performance computing (HPC) applications. Collective communication operations, such as Broadcast and All-Reduce, can aggregate and disseminate data to multiple processes while in practice retaining a relatively simple API (Application Programming Interface). Collectives abstract away much of the complexity of managing communication; however, it is critical that both the collective communication implementation and programming model chosen are well architected, well designed, and optimized for the particular intended application.

Having existed for almost 30 years, Message Passing Interface (MPI)[①] is one of the most widely-used programming models for large-scale scientific applications that involve collective communication. Due to its high speed and portability, MPI has become the model favored in the academic community. There are various implementations of the MPI programming model, such as MPICH[②], MVAPICH[③], and Open

Survey

Special Issue in Honor of Professor Kai Hwang's 80th Birthday

This work was supported in part by the U.S. National Science Foundation under Grant No. CCF-2132049, a Google Research Award, and a Meta Faculty Research Award. This work used the Expanse cluster at SDSC (San Diego Supercomputer Center) through allocation CIS210053 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by the U.S. National Science Foundation under Grant Nos. 2138259, 2138286, 2138307, 2137603, and 2138296.

①MPI: A message-passing interface standard version 4.0. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, Jan. 2023.
②MPICH. https://www.mpich.org/, Jan. 2023.
③MVAPICH. https://mvapich.cse.ohio-state.edu/, Jan. 2023.

MPI[④]. Despite the age of MPI and the development of new collective communication models and libraries, few have been able to compete with MPI in terms of popularity and generality. Some examples of newer non-MPI libraries are OpenSHMEM (Open-source Symmetric Hierarchical MEMory)[⑤], UCX (Unified Communication X)[⑥], and UCC (Unified Collective Communication)[⑦]. Fig.1 shows an overview of the classic collective communication libraries, modern collective communication libraries, as well as related communication hardware and interconnects.

In recent years, machine learning (ML), especially deep learning (DL), has become an extremely hot topic, and there have been numerous advancements in many scientific fields such as computer vision and natural language processing. With continuously increasing data volume and model sizes, methods for reducing training and inference time have themselves become important research topics. For example, the GPT-3[2] (Generative Pre-trained Transformer 3) model contains approximately 175 billion parameters and may take multiple days (or more) to train on advanced GPU-based clusters. Long training times are often considered blockers for the practical deployment of such models. The case is similarly severe when considering industry-level large-scale ML/DL models such as deep learning recommendation models (DLRM)[3]. Therefore, it is necessary to accelerate these processes with effective use of parallel computing, and collectives have the potential to significantly influence the performance and scalability. Under the influence of ML/DL, optimizations on some collective routines are heavily investigated[4, 5]. This evolution also applies to related communication hardware and interconnects. For example, the traditional Remote Direct Memory Access (RDMA) communication mechanism has been widely used in many areas such as HPC, big data[6–10], key-value store[11–14], and high-performance cloud computing workloads[15–17]. With the advance of ML, there is an increasing demand for RDMA and GPUDirect RDMA (GDR)[18, 19]. The interconnect speed requirements can reach 400 Gbps

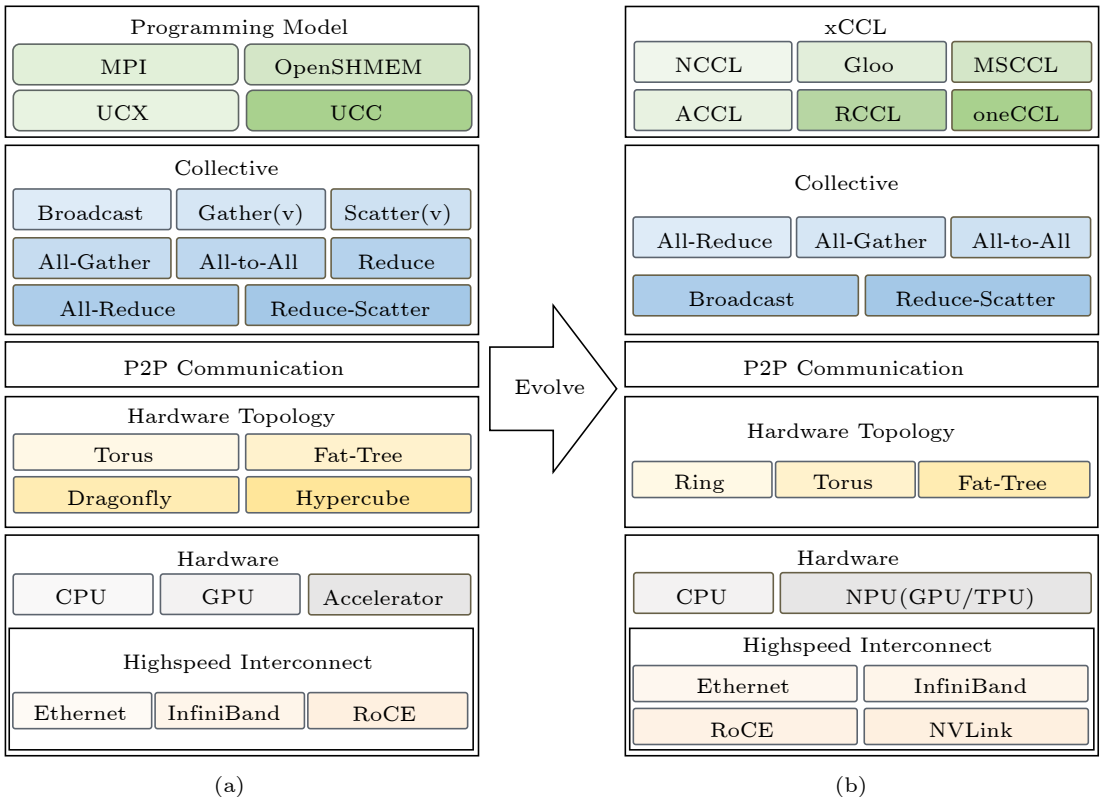


Fig.1. Overview of collective communication evolution. (a) Classic HPC scenarios. (b) Emerging deep learning scenarios.

[④]Open MPI. https://www.open-mpi.org/, Jan. 2023.
[⑤]OpenSHMEM. http://www.openshmem.org/site/, Jan. 2023.
[⑥]UCX. https://openucx.org/, Jan. 2023.
[⑦]UCC. https://ucfconsortium.org/projects/ucc/, Jan. 2023.

per port[20].

However, while MPI has enjoyed success in the academic world, it is not widely adopted in the industry. Instead, many industry-leading companies like NVIDIA and Microsoft have developed their own collective communication libraries for deep learning applications. Most notably, the NVIDIA Collective Communications Library (NCCL)[8], first released by NVIDIA in 2015, has gained enough traction to inspire other companies to develop and deploy similar collective libraries such as AMD's ROCm Collective Communication Library (RCCL)[9] and parts of Gloo[10]. In this paper, we refer to such collective communication libraries as xCCL. The evolution from MPI-dominated collectives for classic HPC scenarios to emerging hardware-accelerated collectives for deep learning scenarios is shown in Fig.1.

This momentum has motivated us to pose several research questions. 1) What makes the contemporary xCCL libraries more attractive than classic MPI designs? 2) What are the performance characteristics of each collective communication library? 3) How are these xCCL libraries designed? Are there shared design patterns, and if so, why?

To answer these questions, we survey the current state-of-the-art collective communication libraries (i.e., xCCL), with a focus on those developed for industry deep learning workloads. We investigate the features of these xCCLs, compare their performance with experiments, and discuss key takeaways and interesting observations.

The rest of this paper is organized as follows. Section 2 introduces widely used collective communication routines. Section 3 and Section 4 describe the popular physical network topologies and collective algorithms. In Section 5, we present the impact of collectives on machine learning training as well as some case studies from industry. In Section 6, we survey representative industry-developed collective communication libraries and introduce their features. In Section 7, we select several libraries and run experiments to benchmark them. We show a comparison of their performance characteristics. Section 8 will discuss some of our observations and insights. Lastly, Section 9 discusses some related work and Section 10 concludes this paper.

The main contributions of this paper are as follows.

- Summarizing and studying the collective communication operations, network topologies, and algorithms that underpin contemporary distributed deep learning training.
- Discussing industry collective communication solutions through case studies and a detailed examination of collective communication libraries.
- Comparing the performance of current collective communication libraries using industry-made benchmarks.

2 Collective Communication Routines

Collective communication operations are an essential tool used in many high-performance computing applications to move and process data within multi-process systems. Though there are many named routines as listed in Table 1, some are especially important for machine learning applications. Programmers can use individual or combinations of collective routines to build distributed training strategies. In this section, we will review routines that are implemented in contemporary collective communication libraries. A high-level review of collective algorithms is included in Section 4.

2.1 Broadcast

The Broadcast collective operation describes a process whereby the root node distributes the same data to all nodes within the system. After the Broadcast operation is complete, every node will hold the same data. Broadcast is one of the two most common collectives in DL training applications (along with All-Reduce; see Subsection 2.6) and can be used for tasks such as sending training data to all processes.

Example. Consider a system with four processes (as in Fig.2(a)): p_0, p_1, p_2, and p_3. Process p_0 holds data D. After the Broadcast collective runs (t_1), processes p_0, p_1, p_2, and p_3 will all hold data D.

2.2 All-Gather

The All-Gather collective operation results in each node receiving data from all nodes within the system.

[8]NVIDIA Collective Communication Library. https://github.com/NVIDIA/nccl, Jan. 2023.
[9]ROCm Collective Communication Library. https://rccl.readthedocs.io/en/rocm-5.2.3/, Jan. 2023.
[10]Collective Communications Library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo, Jan. 2023.

Table 1. Summary of Collective Support Within Libraries

Collective	Discussed in	MPI Function	Implemented in				
			NCCL	MSCCL	Gloo	oneCCL	ACCL
Barrier	Not discussed	`MPI_BARRIER`	No	No†	Yes	Yes	Unknown
Broadcast	Subsection 2.1	`MPI_BCAST`	Yes	No†	Yes	Yes	Yes
Reduce	Subsection 2.5	`MPI_REDUCE`	Yes	No†	Yes	Yes	Unknown
Gather	Not discussed	`MPI_GATHER`	No	No†	Yes‡	No	Unknown
Scatter	Subsection 2.3	`MPI_SCATTER`	No	No†	Yes‡	No	Unknown
All-Gather	Subsection 2.2	`MPI_ALLGATHER`	Yes	No†	Yes‡	Yes	Yes
All-to-All	Subsection 2.4	`MPI_ALLTOALL`	No	Yes	No	Yes	Unknown
All-Reduce	Subsection 2.6	`MPI_ALLREDUCE`	Yes	Yes	Yes	Yes	Yes
Reduce-Scatter	Subsection 2.7	`MPI_REDUCE_SCATTER`	Yes	No†	Yes	Yes	Yes
Scan	Not discussed	`MPI_SCAN`	No	No†	No	No	Unknown

Note: MSCCL is unique because it allows programmers to implement their own collective routines and algorithms. †: algorithm not provided but can be implemented using DSL or called via NCCL API; ‡: algorithm not supported on all accelerator types.

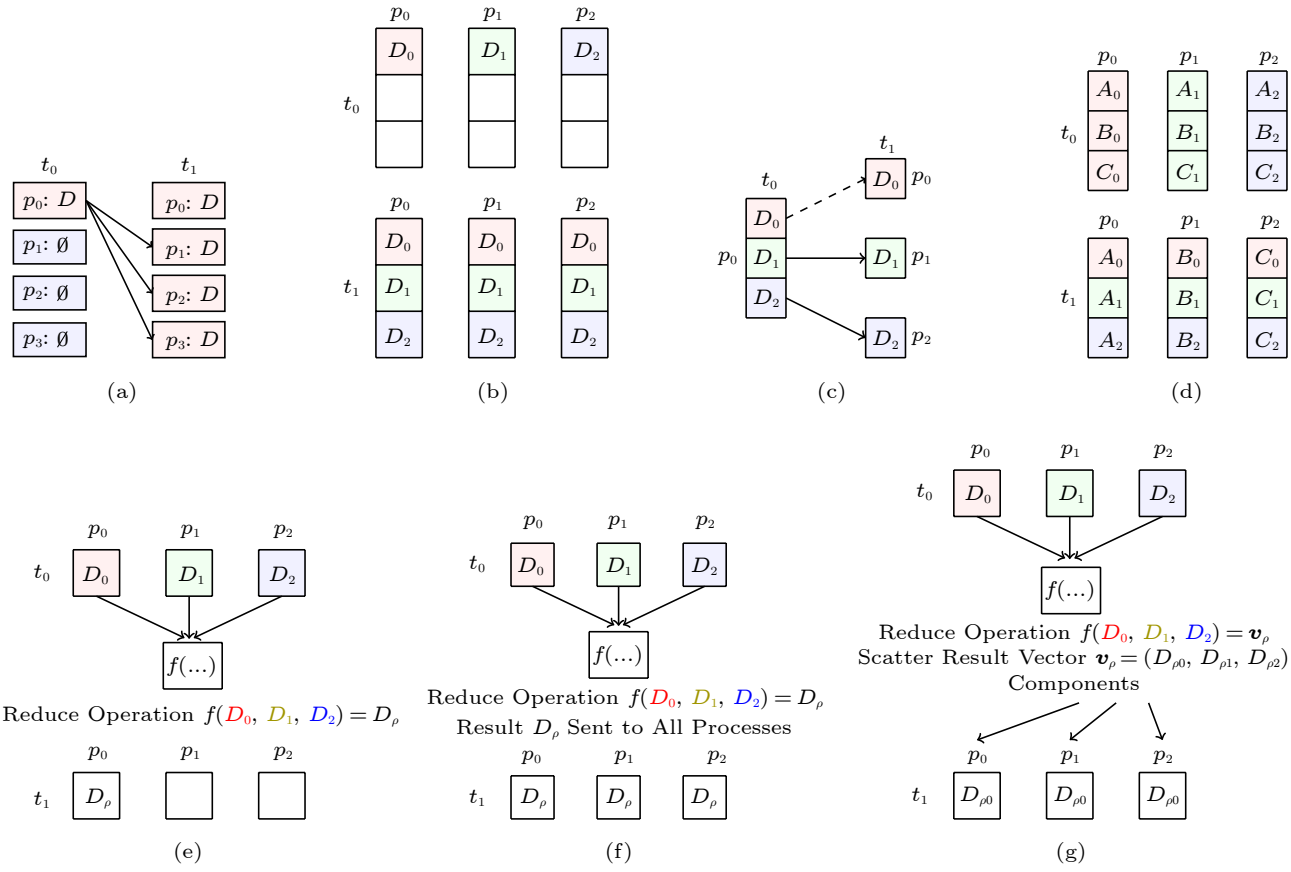


Fig.2. Overview of collective operations. (a) Broadcast. (b) All-Gather. (c) Scatter. (d) All-to-All. (e) Reduce. (f) All-Reduce. (g) Reduce-Scatter.

Essentially, All-Gather can be described as all processes performing a Broadcast operation with their respective data or as all nodes performing a Gather operation. Note that this is not necessarily how All-Gather is actually implemented.

Example. Consider a system with three processes (as in Fig.2(b)): p_0, p_1, and p_2. Process p_0 holds data D_0, process p_1 holds data D_1, and process p_2 holds data D_2. After All-Gather completes, processes p_0, p_1, and p_2 will all hold data D_0, data D_1, and data D_2.

2.3 Scatter

Unlike Broadcast, in which one node sends the same data to every other process, the Scatter collective operation involves a single process transmitting

different data to the other processes based on some splitting pattern or rule. By the traditional definition of Scatter, the rule is that the input data are divided into n pieces where n is the number of processes in the system. Each piece is then sent to its corresponding process[21].

Example. Consider a system with three processes (as in Fig.2(c)): p_0, p_1, and p_2. Process p_0 holds data vector $\boldsymbol{v}_d = (D_0, D_1, D_2)$ where D_0, D_1, and D_2 are data. When Scatter is run, vector $\boldsymbol{v}_d$ is divided into component pieces D_0, D_1, and D_2. Data D_0 remains on p_0, D_1 is sent to process p_1, and D_2 is sent to process p_2. There is a clear benefit to using Scatter over Broadcast when dividing work among processes as each process will not waste memory holding data it does not need. Network bandwidth can also be conserved by avoiding unnecessary data transfer operations[21].

2.4 All-to-All(v)

All-to-Allv (note the addition of "v") is like standard All-to-All, except that participating processes are not restricted to sending uniform data sizes and can instead send messages with variable sizes. The general All-to-All operation itself is where each process sends data to each of the other processes in the system. The resulting data layout is effectively a transpose of the layout present before the operation. The All-to-All collective is vital if high-performance switching between data and model parallelism in a deep learning training process is required because this switch can be described as a transpose.

Example. Consider a system where there are three processes (as in Fig.2(d)): p_0, p_1, and p_2. Each process holds unique data A_i, B_i, and C_i where i corresponds to the process number. After the All-to-All collective completes, process p_0 will hold data A_0, A_1, and A_2; process p_1 will hold data B_0, B_1, and B_2; and process p_2 will hold data C_0, C_1, and C_2.

2.5 Reduce

The Reduce collective refers to a process in which a single node receives data from each node in the system and applies some operation on those data, resulting in a single output. Note that this operation can be anything, provided it is associative. This allows the operation to be performed in parallel while maintaining the correctness and determinism of the program.

Example. Consider a system with three processes (as in Fig.2(e)): p_0, p_1, and p_2. Process p_0 holds D_0, p_1 holds D_1, and p_2 holds D_2. When the Reduce collective is performed, data from process p_0, data from process p_1, and data from process p_2 will be combined to produce result $f(D_0, D_1, D_2) = D_\rho$. If p_0 is set as the destination process, then result D_ρ will be sent to p_0. Note that the Reduce collective does not itself distribute result D_ρ to the other processes. Instead, one must either broadcast result D_ρ as shown in Subsection 2.1 or use the All-Reduce collective operation as explained in Subsection 2.6. If the result D_ρ must be broken up before being distributed to the other processes, either the Scatter operation can be used after Reduce, or the Reduce-Scatter operation can be used in place of both (as explained in Subsection 2.3 and Subsection 2.7 respectively).

2.6 All-Reduce

At a high level, All-Reduce collective can be described as a Reduce step followed by a Broadcast step. After the operation completes, all processes in the system will hold the result of the Reduce operation. All-Reduce is used extensively in data-parallel distributed deep learning training tasks to compute and communicate gradients during the backpropagation step.

Example. Consider a system with three processes (as in Fig.2(f)): p_0, p_1, and p_2. Process p_0 holds D_0, p_1 holds D_1, and p_2 holds D_2. When the All-Reduce collective operation is performed, data from process p_0, data from process p_1, and data from process p_2 will be combined to produce result $f(D_0, D_1, D_2) = D_\rho$. The result D_ρ is then sent to each of the processes p_0, p_1, and p_2. All-Reduce implementations are tuned for higher performance than running Reduce and Broadcast sequentially, even if both approaches result in all processes holding D_ρ.

2.7 Reduce-Scatter

As the name implies, the Reduce-Scatter collective is best described as the combination of the Reduce operation and the Scatter operation in the given order. This definition, however, is not fully descriptive as it is the result of the Reduce operation that must be divided into n pieces so that it can be distributed to the processes in the system[21].

Example. In a system with three processes as shown (as in Fig.2(g)): p_0, p_1, and p_2 where each process holds corresponding data D_0, D_1, and D_2, the Re-

duce portion of Reduce-Scatter produces an output $\boldsymbol{v}_\rho$. The components of the result D_{ρ_0}, D_{ρ_1}, and D_{ρ_2} are scattered (e.g., via the Scatter collective) to processes p_0, p_1, and p_2 respectively.

3 Network Topologies for Collectives

In MPI, the Communicator construct is an abstraction that hides the complexity of lower-level communication between processes. This makes programming much more convenient. However, the physical network topology (i.e., not just the virtual topology associated with MPI Communicators) chosen can heavily impact the performance of collective communications. This is true for traditional collective applications and there are many studies focused on designing network topology-aware collective algorithms to best take advantage of different network architectures[22]. The network topology is especially important when considering hardware accelerated collectives because poor architecture decisions have the potential to wipe out the performance gains realized by using accelerator-specific communications in communication-bound applications[23].

3.1 Hypercube

The hypercube network topology[24] consists of a set of nodes connected in a multi-dimensional cube pattern (hypercube). Increasing the number of nodes in the system will increase the dimensionality of the hypercube. Complete hypercubes must contain exactly 2^k nodes, where k is the hypercube dimensionality. As a comparison, incomplete hypercubes can have any number of nodes[25]. The simplest hypercube network is two nodes connected by a single link and is described as a 1D (one-dimensional) hypercube. A 2D hypercube is four nodes connected in a square pattern. Fig.3 shows an example of a 4D hypercube containing 16 nodes and is, as a consequence, complete.

While hypercube topologies may not be deployed as frequently as other topologies in high-performance computing (HPC) applications in favor of architectures such as Fat-Tree (see Subsection 3.4) and Dragonfly+ (see Subsection 3.5), it serves as an important reference that other more recent topologies can be compared against. Hypercube topologies are resistant to node failures due to their high connectivity; however, the same high connectivity can result in scaling issues and higher complexity for networks with larger numbers of nodes[25]. They are also unique in that incomplete hypercube topologies can exhibit different performance characteristics than complete hypercube topologies[24, 25].

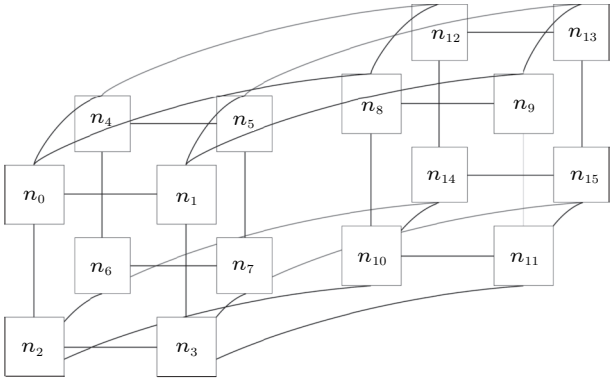


Fig.3. Example of a complete hypercube network topology with 16 nodes. Vertices represent physical nodes and edges represent physical network connections.

3.2 Ring

A ring topology[26] is a configuration where all the members are connected in a conceptually circular fashion. Hence, each member has two connections: one to each of its immediate neighbors. To communicate, packets of data are transmitted from one device to the next until reaching the destination. There are two transfer modes: the unidirectional ring network, in which packets of data travel in only one direction, and the bidirectional ring network, in which packets may travel in either direction.

The ring topology has three advantages. First, in both transmission modes of the ring topology, all data flow in only one direction, thus reducing packet collisions. Second, devices can be added easily without affecting the transmission speed. Finally, there is no need for a central server to coordinate network connectivity. At the same time, the ring topology possesses notable disadvantages. First, in the worst case, data transmitted through the network must pass through all devices, which makes data transmission less flexible. Second, the entire topology will be affected if one machine experiences a failure. Also, the channel utilization of the ring topology is inefficient for short packets, and bandwidth fragmentation may occur.

3.3 Torus

A torus topology[27] is a generalization of the ring topology to higher dimensions, where the ring topolo-

gy is viewed as a one-dimensional (1D) torus. In a 1D torus, as mentioned in Subsection 3.2, each member is connected to its two neighbors. The communication can occur bidirectionally $(-x, +x)$. In a 2D torus configuration, there are two dimensions consisting of m rows and n columns. Each member in the topology is connected to its four immediate neighbors and communication can occur in four directions $(-x, +x, -y, +y)$. Similarly, for an ND torus topology where N is the number of dimensions, each member in the topology is connected to its $2N$ neighbors. Communication is possible in 2^N directions as each member node will have two neighbors in each dimension. Some examples of torus topologies of different dimensions are shown in Fig.4 (1D) and Fig.5 (2D).

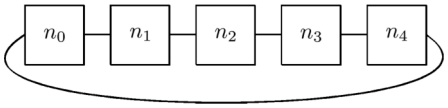


Fig.4. Example of the ring network topology. Vertices represent physical nodes and edges represent physical network connections. Node n_0 is connected to node n_1 and node n_4 (for two total connections).

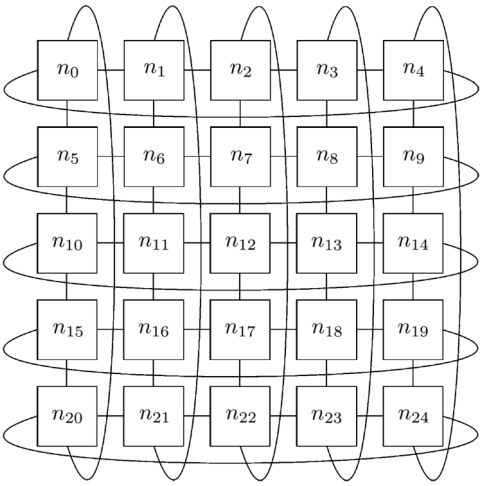


Fig.5. Example of a 2D Torus network topology. Vertices represent physical nodes and edges represent physical network connections. Node n_0 is connected to nodes n_1, n_4, n_5, and n_{20} for a total of four connections. In this case $N = 2$ dimensions, therefore, each node will have exactly $2N = 2 \times 2 = 4$ physical connections. If this particular topology were to be visualized in three-dimensional space, it would resemble a "donut" or "ring torus" shape.

One advantage of the torus topology is that it significantly decreases the topology diameter and reduces the cost to add new members. All one must do is to add additional links[28][⑪]. The torus topology can also provide higher bandwidth and lower latency than some other network topologies while still achieving high scalability. This is because the torus topology is homogeneous and member nodes can communicate with one another via multiple routes[29]. For the same reason, it is also able to consume less power[29][⑪].

The torus topology also has certain limitations. First, as dimensionality increases, the number of physical network connections necessarily increases. This means that wiring becomes more complex and deployment cost grows[29]. Second, as new member nodes are added to a given dimension, it will require more energy and take longer to communicate within the dimension as each message must travel through more nodes[29, 30]. To address this problem, a modified version of torus topology, called folded-torus topology, was developed[31].

3.4 Fat-Tree

The fat-tree topology[32] is one of the most widely used topologies for efficient data communication. Unlike traditional tree structures in computer science, the fat-tree topology resembles the trees in the real world. In a traditional tree topology, all branches have the same thickness (bandwidth) whereas in a fat-tree topology, the communication bandwidth gets larger, i.e., the fat-tree gets fatter, as one moves closer to the root. An illustration of a fat-tree topology is given in Fig.6.

In the fat-tree topology, only the leaves are used for computation and all the other nodes are strictly for communication. For example, when a leaf node wants to communicate with another leaf node, data will flow up the hierarchy recursively until a shared ancestor with the second leaf node is found. Data then flow back down the hierarchy to the second leaf node.

There are numerous advantages to use the fat-tree topology. First, the average distance between nodes grows logarithmically since it is a tree structure in nature[⑫]. In addition, it has also been proved that fat-trees are recursively scalable and partitionable with multiple well designed routing algorithms[33, 34]. Some other advantages of the fat-tree topology include its symmetry, regularity, and high connectivity[35], which

[⑪]The 3D Torus Architecture and the Eurotech Approach. https://www.hpcwire.com/2011/06/20/the_3d_torus_architecture_and_the_eurotech_approach/, Jan. 2023.
[⑫]Fat Trees. https://people.engr.ncsu.edu/efg/506/s02/lectures/notes/lec27.pdf, Jan. 2023.

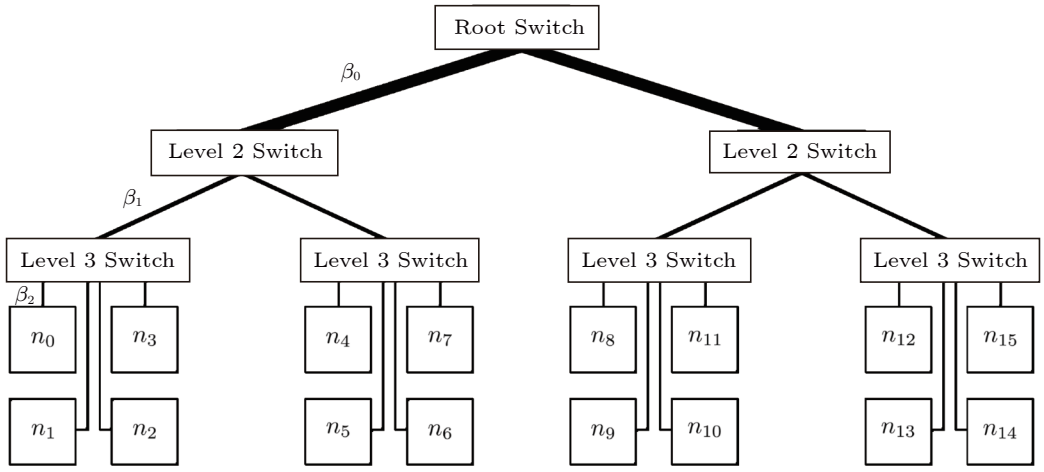


Fig.6. Visualization of a fat-tree network topology. Bandwidth is represented by link line thickness, and follows $\beta_0 > \beta_1 > \beta_2$. Nodes $n_0, ..., n_{15}$ perform computation, while switches handle communication only.

is attributable to its tree structure.

One disadvantage is the bandwidth requirement for the branches connected to the root[36]. This bandwidth requirement will get higher and higher when the fat-tree grows bigger, leading to a challenge in implementation. Another disadvantage is that due to the the structure of the fat-tree topology, it is necessary to traverse all nodes between two leaf nodes when communicating data. In this case, load balancing and scheduling becomes another challenge[37].

3.5 Dragonfly and Dragonfly+

Another commonly used topology is the dragonfly topology[38], which is a hierarchical structure made up of multiple levels (i.e., routers and groups). An example of the dragonfly topology is shown in Fig.7. At the lowest level, each router is connected to multiple (p) terminals. Above this level is the group, which is a collection of routers (a routers) which have connections to routers in the same group (local channels) and connections to routers in the other groups (h global channels). In a dragonfly topology, there will be many groups. All routers within a group work as a "virtual router" that has a very high radix (k). This radix is equal to the number of routers in the group multiplying the number of connections each router has ($k = a \times (p + h)$). All the numbers a, p, and h can be adjusted in accordance with the deployment requirements.

Modularity is one of the advantages that the dragonfly topology can provide[39]. Because the designs of intra-group connections and inter-group connections are decoupled, the wiring within a group does not affect the number of groups in the topology. In addition, this modular design also leads to the high scalability of the topology[38, 39]. The dragonfly topology is able to scale to a high number of nodes by simply increasing the effective radix, while still keeping a relatively low number of hops[38].

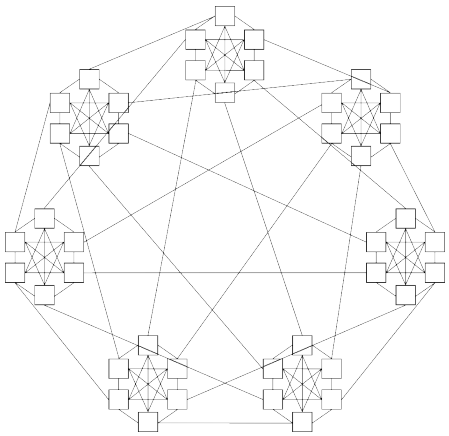


Fig.7. Example of a dragonfly network topology. In this case, all boxes represent routers that nodes (not shown) are connected to.

At the same time, this high number of connections also leads to a high construction cost for dragonfly topologies[39]. Also, these various connections can bring the topology a high path diversity, causing a low network utilization and throughput under certain traffic patterns[40]. To address this problem, an extended version called dragonfly+ has been introduced in recent years[41]. In the dragonfly+ topology, routers inside the group are connected in a Clos-like topology[32]. It also shows higher scalability and better router utilization[41].

4 Collective Communication Algorithms

Though the collective routines are presented to the programmer through a clean API, the collective communication library must implement algorithms that perform the actual intra-node and inter-node communications.

4.1 Classic Collective Communication Algorithms

In recent years, new advancements have been made in the development of collective communication algorithms. Table 2 briefly overviews the representative state-of-the-art collective communication al-

Table 2. Classic and xCCL's Collective Communication Algorithms

Category		Collective	Algorithm	Description on Suitability (e.g., Message Size, Number of Processes)
Classic		All-to-All	Bruck[42]	Short (e.g., < 32 B)
			Isend-Irecv[43]	Medium (e.g., 32 B to 32 KB)
			Pairwise-Exchange[44]	Long (2^n processes)
		All-Gather	Ring[43]	Long, medium (not 2^n processes)
			Recursive-Doubling[43]	Short, medium (2^n processes)
			Bruck[42]	Short (not 2^n processes)
		Broadcast	Binomial Tree[45]	Short (e.g., < 32 B)
			Van de Geijn[46, 47]	Long (e.g., > 32 KB)
		Reduce-Scatter	Recursive-Halving[44]	Short (commutative reduction)
			Recursive-Doubling[43]	Short (not commutative reduction)
			Pairwise-Exchange[43]	Long (e.g., $\geqslant$ 512 KB for commutative, $\geqslant$ 512 B for noncommutative)
			Binomial Tree and Linear Scatterv[43]	Medium
		Reduce	Binomial Tree[45]	Short (e.g., $\leqslant$ 2 KB)
			Rabenseifner's[48]	Long (e.g., > 2 KB)
		All-Reduce	Recursive-Doubling[44]	Short, long (user-defined reduction)
			Rabenseifner's[48]	Long (predefined reduction)
			Ring[44]	Small or medium numbers of processes
		Reduce/All-Reduce	Vector Halving and Distance Doubling[48]	Long (vectors and 2^n processes)
			Binary Blocks[48]	Not 2^n processes
xCCL	NCCL	All-Reduce	Double Binary Trees[13]	Short, medium
			Ring[14]	Long
	MSCCL	All-Reduce[15]	Ring	Medium (e.g., 32 KB to 3 MB)
			All-Pairs	Short and medium (e.g., 1 KB to 2 MB)
			Hierarchical	Short or long (e.g., < 64 MB or > 1 GB)
		All-to-All[15]	Two-Step	Long (e.g., > 2 MB)
	Gloo	All-Reduce[16]	Ring	Long
			Ring-Chunked	Long
			Halving-Doubling	2^n processes
			BCube	Short
		Reduce-Scatter[16]	Halving-Doubling	2^n processes
		Broadcast[16]	Pairwise-Exchange	Long (2^n processes)
	ACCL	All-Reduce[49]	Hybrid	Medium, long (e.g., > 16 KB)

Note: "not 2^n processes" means the number of processes is not 2^n.

[13]Massively scale your deep learning training with NCCL 2.4. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/, Jan. 2023.

[14]NCCL: Accelerated multi-GPU collective communications. https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf, Jan. 2023.

[15]MSCCL: Microsoft collective communication library. https://github.com/microsoft/msccl, Jan. 2023.

[16]Collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo, Jan. 2023.

gorithms and their features. In these algorithms: the Ring, Binomial Tree, Recursive-Doubling, and Recursive-Halving algorithms are the most widely used in HPC workloads. Hence, we will focus on these algorithms in particular.

To estimate the latency and bandwidth of collective communication algorithms, we use the $\alpha - \beta$ cost model[50]. The time taken by the bidirectional communication between processes is $\alpha + n\beta$ and the unidirectional communication is $\alpha_{\text{uni}} + n\beta_{\text{uni}}$[51]. In this function, α is the latency (startup time) per message, β is the transfer time per byte, n is the number of transferred bytes, and p is the number of processes in the communication. In the case of reduction operations, γ is the computation cost per byte for one process.

4.1.1 *Ring*

The Ring algorithm is traditionally utilized for All-Gather. The implementation of All-Gather in this method is that data are transferred around a virtual ring of processes. First, each process sends its chunk of data to the following process in the ring and receives the chunk of data from the previous process in the ring. From the second step, each process sends the data it received from the previous process in the first step to the following process. If p is the number of processes, it takes $p-1$ steps to complete the entire algorithm. If n is the total amount of data to be gathered on each process, then at every step, each process sends and receives n/p amounts of data. Therefore, the time taken by this algorithm is $T_{\text{ring}}^{\text{All-Gather}} = (p-1)\alpha + ((p-1)/p)n\beta$[43].

The Ring algorithm is also used for All-Reduce. There are two phases in All-Reduce: Reduce-Scatter and All-Gather. The Ring algorithm can be applied for All-Gather, and the Reduce-Scatter phase can be performed in the Pairwise-Exchange algorithm. When the number of processes is not a power of 2, this algorithm performs well in bandwidth utilization. Still, the latency of this algorithm grows linearly as the number of processes increases. Therefore, this algorithm is only suitable for small or medium processes or large vectors. For All-Reduce, the time taken is $T_{\text{ring}}^{\text{All-Reduce}} = 2(p-1)\alpha + 2n\beta + n\gamma - (1/p)(2n\beta + n\gamma)$[44].

4.1.2 *Binomial Tree*

The Binomial Tree algorithm is commonly used for Broadcast in MPICH. First of all, process $(\text{root} + (p/2))$ receives data from the root. From the second step, this process and the root act as new roots in their respective subtrees. This algorithm will run recursively and takes a total of $\lg p$ steps. In this algorithm, each process communicates n bytes of data at any step. Therefore, the time taken by this algorithm to perform Broadcast is $T_{\text{tree}}^{\text{Broadcast}} = (\lg p)(\alpha + n\beta)$[43]. This algorithm performs well when communicating short messages because of the logarithmic latency term. As a result, the Binomial Tree algorithm can be a good choice when working with short messages (e.g., < 12 KB) or when the number of processes is less than 8.

The Binomial Tree algorithm can also efficiently implement Reduce. The Binomial Tree algorithm takes $\lg p$ steps to complete the process, and the amount of data is n at each step. In general, the time taken by this algorithm is $T_{\text{tree}}^{\text{Reduce}} = (\lg p)(\alpha + n\beta + n\gamma)$[43]. Owing to the $\lg p$ steps, the Binomial Tree algorithm performs Reduce efficiently for short messages. For Reduce, the Binomial Tree algorithm is used for short messages (e.g., $\leqslant$ 2 KB) when the reduction operation is predefined. Because the user-defined reduction operations may pass or break up derived datatypes to do the complex Reduce-Scatter, the Binomial Tree algorithm is used for all message sizes when the reduction operations are user-defined. When executing the All-Reduce process, the algorithm first does a Reduce to rank 0 and then performs a Broadcast.

4.1.3 *Recursive-Doubling*

Recursive-Doubling is an efficient algorithm for All-Gather. In the first step, each process sends and receives data from its neighbors. In the second step, each process sends and receives data from the process that is two processes away from it. In the third step, the process exchanges data from the process that is four processes away from it, and so on. In this way, when the number of processes is a power of 2, all data communication can be completed in $\lg p$ steps. The amount of data exchanged by each process is n/p in the first step, $2n/p$ in the second step, and so on. In the last step, the amount of data is $(2^{\lg(p-1)}n)/p$. In general, the total time taken by this algorithm is $T_{\text{rec-dbl}}^{\text{All-Gather}} = \lg p\alpha + ((p-1)/p)n\beta$[43]. Due to the communication's mathematical features, Recursive-Doubling works very well for situations where the num-

ber of processes is a power of 2, but it does not work well when the number of processes is not a power of 2.

The Recursive-Doubling algorithm can be used in Reduce-Scatter, similar to the one used in All-Gather. However, more data are communicated in Reduce-Scatter than in All-Gather. In step 1 of Reduce-Scatter, the data needed for their result in each process is not exchanged, and the amount of data exchanged by each process is $(n-(n/p))$; in step 2, the data required by themselves and by the processes communicated in the previous step in each process are not exchanged, and the amount of data exchanged by each process is $(n-(2n/p))$; in step 3, it is $(n-(4n/p))$; and so on. Therefore, the time taken by this algorithm is $T_{\text{rec-dbl}}^{\text{Reduce-Scatter}} = \lg p\alpha + (\lg p - ((p-1)/p))n\beta + (\lg p - ((p-1/p))n\gamma$[43]. This algorithm works well for concise messages (e.g., < 32 B).

The Recursive-Doubling algorithm can also perform All-Reduce similarly to how it performs All-Gather, except that each communication step of the Recursive-Doubling algorithm also involves a local reduction. The Recursive-Doubling algorithm performs well for short messages and long messages with user-defined reduction operations. The time taken by this algorithm for All-Reduce is $T_{\text{rec-dbl}}^{\text{All-Reduce}} = \lg p\alpha + n\lg p\beta + n\lg p\gamma$[44].

4.1.4 *Recursive-Halving*

Similar to applying the Recursive-Doubling algorithm for All-Gather, the Recursive-Halving algorithm can be used to perform Reduce-Scatter. First, processes at a distance of $p/2$ away exchange data with each other. Each process performs both the sending and receiving operations. All processes need the sent data in the other half, and all processes need the received data in its half. The reduction operation is performed on the received data. The reduction can be made because the procedure is commutative. Second, processes at a distance of $p/4$ away exchange data with each other: each process performs both the sending and receiving operations. All processes need the sent data in the other half of the current subtree, and all processes require the received data in its half of the current subtree. The reduction operation is performed on the received data. This procedure is performed recursively, halving the data communicated at each step. The total number of steps of this process is $\lg p$. Therefore, if p is a power of 2, the time taken by this algorithm is given by $T_{\text{rec-half}}^{\text{Reduce-Scatter}} = \lg p\alpha + ((p-1)/p)n\beta + ((p-1)/p)n\gamma$[44].

4.2 xCCL Collective Communication Algorithms

In practice, xCCL will select algorithms to perform collectives based on conditions such as system configuration, network topology, and invocation circumstances to improve the performance[52]. Next, we introduce these algorithms used by xCCL.

4.2.1 *NCCL*

Ring. The Ring algorithm is used for All-Reduce in NCCL to move data across all GPUs[17]. The data are split into multiple chunks and transferred one by one during the operation. This pipeline modality reduces the idle time the GPU spends waiting for data. However, the latency of the Ring algorithm for All-Reduce increases with the number of GPU devices. Since NCCL is implemented with CUDA, one CUDA thread block is allocated to one ring direction in this library.

Double Binary Trees. Since the latency of the Ring algorithm increases with the number of GPUs, it is not suitable for communication among a large number of GPUs. The Double Binary Tree algorithm was proposed to solve this problem because of its logarithmic latency[18]. Based on the architecture of a binary tree, the leaves of one binary tree can be used as the nodes of another. Almost every rank is connected to two parents and two children ranks, except for the root ranks. Compared with the Ring algorithm, the latency of Double Binary Trees is more negligible in the NCCL test on various large machines.

4.2.2 *MSCCL*

Ring. MSCCL implements Ring for All-Reduce,

[17]NCCL: Accelerated multi-GPU collective communications. https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf, Jan. 2023.

[18]Massively scale your deep learning training with NCCL 2.4. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/, Jan. 2023.

Reduce-Scatter, and All-Gather[19]. MSCCL allocates multiple channels to one logical ring. In this way, different point-to-point connections can be implemented between the same pairs of GPUs. The protocol for scheduling a logical ring onto one channel varies according to the message size. This strategy enables the logical ring's distribution across channels and efficiently overlaps point-to-point operations.

All-Pairs. In MSCCL, three different types (input, output, and scratch) of GPU buffers are available for chunks of data. Because the Ring algorithm is unsuitable for small message sizes, MSCCL implements All-Pairs for All-Reduce when the message size is small[19]. The Ring algorithm proceeds in two steps: rank receives the chunk of data from every rank and performs computation operations, and then the chunks of the result data are broadcast to every other rank. Compared with $2n-2$ steps in the Ring algorithm, only two communication steps are used in the All-Pairs algorithm, which makes the latency of the All-Pairs algorithm lower.

Hierarchical. Different algorithms can be applied to perform All-Reduce according to the input configurations. Besides the above-mentioned algorithms, Hierarchical is another possible one in MSCCL[19]. There are four communication steps in this algorithm. The first step is to perform Reduce-Scatter within a node, the second step is to perform Reduce-Scatter across nodes, the third step is to perform All-Gather across nodes, and the final step is to perform All-Gather within a node.

Two-Step. The traditional All-to-All algorithm only implements one communication step, but the number of small chunks transferred across nodes is large. In MSCCL, a two-step All-to-All algorithm is implemented with aggregated cross-node communication to reduce the cost[19].

4.2.3 Gloo

Ring. In Gloo, the Ring algorithm is implemented the same as mentioned in Subsection 4.1.1[20].

Ring-Chunked. Based on the Ring algorithm, the Ring-Chunked algorithm divides the buffer into chunks so that each process can reduce a chunk into a local result while it is transmitting another chunk[20].

Halving-Doubling. The design of Halving-Doubling in Gloo is similar to that of the All-Reduce Recursive Halving and Doubling algorithms[20]. The Halving-Doubling algorithm uses the distance to decide the communication pair between processes. For example, each process sends and receives data buffers: from the process next to it when the communication distance is 1; from the process that is one process away from it when the distance is 2. The algorithm consists of two phases. 1) The distance doubling the Reduce-Scatter operation phase. At the result, each process holds part of the reduction results. 2) The distance halving All-Gather operation phase. At the result, all processes receive the rest parts of the reduction results from other processes.

Pairwise-Exchange. The Pairwise-Exchange algorithm is a simplified Halving-Doubling algorithm[20]. In each step, the nodes are partitioned into pairs and the message size in the communication between pairs is the same. Pairwise-Exchange is used for benchmarking purposes in Gloo.

BCube. The Ring algorithm organizes the communication structure of processes one by one as a ring. Halving-Doubling uses the distance to manage the communication of processes. Different from the above algorithms, the BCube algorithm divides the processes into groups[20]. Firstly, it performs Reduce-Scatter among processes within the group and All-Reduce among the corresponding processes from different groups. Secondly, each group performs All-Gather within the group so that every process receives the reduction results in the end.

4.2.4 ACCL

Hybrid. ACCL uses a hybrid All-Reduce algorithm to maximize bandwidth utilization[49]. Hybrid All-Reduce decouples the All-Reduce operation into several micro-operations, eliminating the meaningless micro-operations. This hybrid algorithm proceeds in three steps. In step 1, the intra-node Reduce-Scatter is performed based on the Ring algorithm; in step 2, the inter-node All-Reduce is performed based on the Halving-Doubling algorithm; in step 3, the intra-node All-Gather is performed based on the Ring algorithm.

5 Collectives and Deep Learning

Machine learning techniques are increasingly being adopted both in industry and in research to solve

[19]MSCCL: Microsoft collective communication library. https://github.com/microsoft/msccl, Jan. 2023.
[20]Collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo, Jan. 2023.

problems not easily handled by traditional methods. Machine learning can be seen in precision systems, automation, cancer detection, self driving, and more. It is well known that hardware accelerators have contributed massively to the advancement of inter-disciplinary machine learning applications.

The adaptation of GPUs for general purpose computing, often referred to as general purpose GPUs (GPGPUs), has allowed training to be conducted in a parallel fashion, drastically reducing the time required to achieve acceptable results when compared with the CPU[53][21–25]. Data volumes—and more importantly, model sizes—continue to increase, creating a need for distributed training solutions, including those that integrate tightly with hardware accelerators. There are many ways to parallelize the training process, and the collective communication paradigm provides the requisite flexibility to implement these solutions while remaining conceptually simple.

With the advent of large models, parallelizing the training process becomes a necessity. Using NCCL with TensorFlow, researchers from Uber found that VGG-16 training can be sped up by a significant 30% when leveraging RDMA networking[53]. Collectives demonstrate a considerable speed-up in training time that very large networks can achieve. Using 256 GPUs for distributed training, Meta trained a ResNet-50 model within one hour with a 90% scaling efficiency from an initial 8 GPUs[54].

Machine learning using collectives is an area of active research. Multiple industry applications have been proposed and implemented into daily workflow. In the following Subsections 5.1–5.4, we observe some representative use cases for Meta, Google, Uber, and Amazon.

5.1 Case Study with Meta Workloads

A recommendation model is a kind of ML workloads that aims to provide personalized recommendations to users. Such models are deployed often in e-commerce, social media, and advertising settings. In 2020, Meta introduced its production-scale DNN-based RMCs (recommendation model classes)[55]. These RMCs all exhibit computation and communication intensive characteristics. Additionally, since the purpose of RMCs is to provide recommendation to users, the ability to achieve a short inference time is a critical metric to evaluate their performance.

For all the requirements mentioned above, the collective plays an important role, as it can directly influence the communication time during the training and inference stages. To improve the performance of these production-scale recommendation models, Meta developed a software-hardware co-design, named Neo, which integrates their collective-related optimizations[56]. The implementation of Neo is closely related to PyTorch[57], a widely-used machine learning framework originally created by Meta. The kernel fusion introduced in Neo is open sourced as part of the Meta General Matrix Multiplication (FBGEMM) library which serves as a matrix processesing backend for PyTorch[58].

Neo is built for efficient and scalable DLRM (Deep Learning Recommendation Model) training utilizing three key techniques. The first one is 4D parallelism that combines table-wise parallelism, row-wise parallelism, column-wise parallelism, and data parallelism. It is aimed at reducing workload imbalance among GPUs to minimize the costs of conducting communication. Second, the hybrid kernel fusion technique fuses the parameter update and the embedding computation into a same CUDA kernel. Third, a new hardware platform called ZionEX was introduced. ZionEX is co-designed with the 4D parallelism technique and also optimizes the inter-node communication for distributed training.

Image recognition networks that utilize residual learning have become extremely popular since their introduction in 2016 due to the fact that they enable much deeper neural network architectures[59]. ImageNet, a database of labeled image data is a popular dataset used to train, test, and evaluate network architectures and training systems[60]. Meta was able to train a ResNet50 model on ImageNet in one hour on a distributed training system[54].

In order to achieve this level of performance, all components of the training system must be considered. Their deployment consisted of 32 nodes each with 8 GPUs for a total of 256 GPUs[54]. Nodes each

[21]MPI: A message-passing interface standard version 4.0. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, Jan. 2023.
[22]Collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo, Jan. 2023.
[23]CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit, Jan. 2023.
[24]NVIDIA NVLink. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/, Jan. 2023.
[25]NVIDIA collective communication library. https://github.com/NVIDIA/nccl, Jan. 2023.

had 20 Gbit network cards and GPUs were directly connected using NVIDIA NVLink. Collective operations were used both locally within nodes to compute local reductions and to communicate gradients between nodes. Local and inter-node communications used NCCL (see Subsection 6.1) while collectives were performed using Gloo (see Subsection 6.5)[54]. Attempting to train with many GPUs can cause communication and aggregation costs to increase to unacceptable levels. To mitigate these issues, CPU and GPU resources were balanced by splitting communication between them using the buffer size and manually selecting the most performant collective algorithms for both the specific workload being tested and the network topology[54].

5.2 Case Study with Google Workloads

DistBelief[61] is a framework for parallel distributed training of DNN models developed by Google researchers. In their paper, Dean *et al.*[61] observed that very large DNN models can benefit greatly when they are trained using many machines organized in a distributed training system. Their largest model with 1.7 billion parameters sees a speedup greater than 12x on 81 machines.

Google's TensorFlow[62] is the most popular framework for deep learning applications and the successor to DistBelief. TensorFlow's flexibility makes it widely applicable to many ML problems and countless researchers have utilized the framework. It provides tools for deploying on GPU clusters, enabling the distributed training of very large models[62]. This allows users with large-scale models to train more efficiently and significantly reduce computation time. As industry data volume and velocity both become larger, quickly training models with enormous parameter sizes increases model training productivity.

Awan *et al.* evaluated the designs and performance of TensorFlow for training multiple DNNs on distributed/HPC systems using different communication libraries[63]. In their paper, they showed that MPI-based communication solutions for TensorFlow achieve 71% scaling efficiency scaling up to 64 GPUs. In a recent study[64], the authors designed a benchmark suite to characterize TensorFlow's communication patterns and performance. Biswas *et al.* developed an RDMA-based gRPC that can adjust communication mechanisms dynamically for TensorFlow-based deep learning training workloads[65].

There is ongoing work to increase the scaling efficiency and communication efficiency of distributed training in TensorFlow, which is highly desired by the DNN community. Google's researchers and developers continue to update the communication subsystem designs in TensorFlow to work with NCCL and other optimized communication backends. For example, a communication library called NCCL Fast Socket[26] was proposed by Google to optimize NCCL collective communication performance for distributed ML training on Google Cloud.

5.3 Case Study with Uber Workloads

Uber has utilized machine learning in multiple diverse applications (e.g., UberEATS, Marketplace Forecasting, Customer Support, Ride Check, Estimated Times of Arrival, One-Click Chat, and Self-Driving Cars). Specifically, Uber's Michelangelo[27] machine learning platform runs several models for UberEATS. Search ranking, search autocomplete, and restaurant rankings are all examples of use case models that UberEATS utilizes from Michelangelo. With the scale of Uber's models increasing, distributing the training process is a practical necessity. Using Michelangelo's Data Science Workbench (DSW), large-scale distributed training and deployment of deep learning models on GPU clusters is well supported[27], even for data scientists and developers with little systems knowledge. Users can easily distribute their training processes with DSW and use Michelangelo's hyperparameter searching algorithms.

As Uber started using deep learning models for self-driving cars, the dataflow grew exponentially and required distributed training across an extensive set of GPU machines. Michelangelo's Horovod[27] was introduced to enable much faster training and research progress by implementing collective communications between GPUs in TensorFlow. In their paper[53], Sergeev and Del Balso replaced Baidu's `ring-allreduce` implementation with NCCL for communication in Horovod. The authors found that a model with a large number of parameters (e.g., VGG-16) saw a 30% speedup, and that other models (e.g., Inception V3 and ResNet-101) exceeded 90% scaling efficiency when scaling to 128 GPUs. The default

[26]NCCL fast socket. https://github.com/google/nccl-fastsocket, Jan. 2023.
[27]Scaling machine learning at Uber with Michelangelo. https://www.uber.com/blog/scaling-michelangelo, Jan. 2023.

TensorFlow distributed implementation was found to waste about half of GPU resources when training on the same 128 GPUs, while Horovod is able to reach the 88% efficiency mark. Though after Horovod's work, TensorFlow has added support for NCCL2. Horovod greatly simplifies the distributed training process for users and supports multiple popular communication libraries (e.g., MPI, NVIDIA's NCCL, Meta's Gloo, Intel's MLSL, and IBM's DDL)[66]. Horovod's GitHub also provides model examples for Keras, MXNet, PyTorch, Spark, TensorFlow, and more, making it very simple to start working quickly with various models.

5.4 Case Study with Amazon Workloads

Amazon Web Services (AWS) provides developers with a wide variety of cloud-based tools. Distributed training is an important tool for building capable large models, especially if models are large enough that training must be split across multiple devices (model parallelism). However, capable systems can be difficult and expensive to deploy. As a result, many companies look for hosted solutions. Network architectures and speeds available on public clouds are often very different than those of dedicated ML training clusters where bandwidths are more heterogeneous. On such cloud-based systems, intra-node and inter-node bandwidth can differ by a factor of 20 or more, meaning assumptions about the cost of using collectives may not longer be correct[67]. Attempting to perform collective operations that involve all ranks can be very expensive.

To help make model-parallel training faster on cloud systems, researchers at Amazon have proposed a system called MiCS[67], which reduces the impact of highly-heterogeneous network architectures on training performance. The key idea is to reduce the number of participants in collective communications and by extension reduce the data volume of the communications that take place over lower-bandwidth connections. This is possible even when model sizes surpass on-node device memory requirements by using a hierarchical communication strategy and breaking communication up into stages. First, devices perform an inter-node All-Gather operation with devices of the same respective relative rank on the other nodes. Second, nodes perform an intra-node All-Gather to complete the communication[67]. Additionally, MiCS makes use of a "2-hop" gradient synchronization process rather than the standard gradient aggregation process. This step is usually very expensive because its cost increases with a higher number of devices. Devices are split into small groups that span across nodes. Gradient partitioning and synchronization is first performed within the small groups and then globally, reducing total traffic. When compared with the existing training optimizer ZeRO[68], MiCS was 2.98x faster[67]. MiCS was also shown to have achieved 99.4% scaling efficiency in a cloud environment[67].

6 Industry Solutions—xCCL

The rise in the popularity of distributed deep learning training has contributed to growing interest in fast, efficient, and portable collective implementations. A summary of industry collective communication libraries is shown in Table 3.

6.1 NVIDIA NCCL

NVIDIA Collective Communication Library[28] is currently the most popular GPU-accelerated collective communication library. It implements collective operations for multiple GPUs across multiple nodes as well as specific point-to-point communication primitives. NCCL officially supports NVIDIA GPUs only, though there have been efforts to port it to AMD graphics cards in the form of the ROCm Collective

Table 3. Summary Comparison of Collective Communication Libraries

CCL	Accelerator	License	P2P	Differentiation
NCCL	NVIDIA GPU	**Open** (MIT)	Yes	Industry standard for GPU-based collectives
Gloo	GPU	**Open** (BSD)	No	Combined CPU/GPU DL workloads
MSCCL	GPU	**Open** (MIT)	Yes	DSL for custom collective algorithms
Intel oneCCL	Intel CPU, GPU, FPGA	**Open** (Apache 2.0)	Yes	Support for heterogeneous accelerators
ROCm	AMD GPU	**Open** (MIT)	Yes	AMD GPU support
ACCL	GPU	Proprietary	N/A	Hybrid algorithms

[28]NVIDIA collective communication library. https://github.com/NVIDIA/nccl, Jan. 2023.

Communication Library (RCCL)[29] (see Subsection 6.4).

NCCL's programming model is very similar to MPI Collectives[30]. However, NCCL is geared toward providing fast communication of messages among GPUs in dense multi-GPU systems, while MPI focuses on efficient communication across thousands of nodes in a cluster.

6.1.1 *Architecture*

NCCL aims to perform communication among GPUs using the CUDA, as shown in Fig.8(a). In NCCL collective communications, the communicators are created using CUDA before launching any collective algorithm design. After the collective algorithm is launched using point-to-point primitives, the point-to-point operation will be effectively enqueued to the given stream.

NCCL uses rings to move data across all GPUs, and therefore data are divided into chunks among all ranks in the communicator to obtain reasonably good bandwidth while lowering the latency. NCCL performs intra-node communication through PCIe, NVLink[31], and GPUDirect[32]. Inter-node communication in NCCL is via GDR. NCCL's CUDA kernels can copy data stored in the global memory of one GPU to another GPU by using GDR and GPUDirect.

6.1.2 *NCCL API*

In a similar fashion to NVIDIA's CUDA[33], NCCL was designed to be easy to program. Because NCCL provides a C API, programmers can use NCCL within existing C projects or even use C bindings in a high-level language like Python.

NCCL can be initiated with the `ncclCommInitRank`(), `ncclCommInitRankConfig`(), or `ncclCommInitAll`() function. Each gives the programmer different options for configuring ranks and communications. A communicator is required to perform any communication operation. Individual collective operations can be run using the correspondingly named API calls. All-Reduce can be run using the `ncclAllReduce`() function, Broadcast can be run using the `ncclBroadcast`() function, and so on. Programmers familiar with MPI should feel comfortable with the NCCL API. NCCL also supports point-to-point communications in the form of `ncclSend`() (sending data to a specific rank) and `ncclRecv`() (receiving data from a specific rank).

6.1.3 *Framework Support*

Because deep neural networks training is becoming too large to be performed on a single compute node, some state-of-the-art deep learning frameworks like TensorFlow[62], Caffe[69], PyTorch[57], CNTK[70], and MXNet[71] have complimented distributed train-

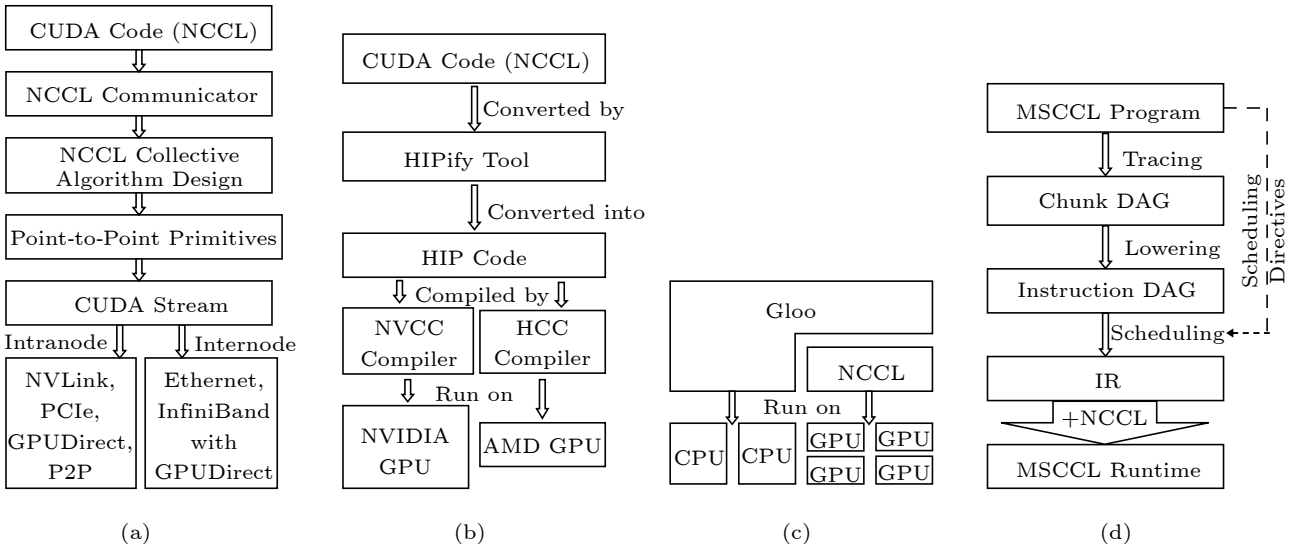


Fig.8. Overview of different architectures of xCCL from industry solutions. (a) NCCL. (b) ROCm. (c) Gloo. (d) MSCCL.

[29]ROCm communication collectives library. https://github.com/ROCmSoftwarePlatform/rccl, Jan. 2023.
[30]MPI: A message-passing interface standard version 4.0. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, Jan. 2023.
[31]NVIDIA NVLink. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/, Jan. 2023.
[32]GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, Jan. 2023.
[33]CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit, Jan. 2023.

ing on multiple nodes by using NCCL. These frameworks use NCCL to perform collective communication among all the available GPUs.

Horovod[53] enables faster, easier distributed training in TensorFlow by employing efficient inter-GPU communication with NCCL.

6.1.4 Supported Features

Communicator. NCCL assigns a unique rank between 0 and $n-1$ to each of the n CUDA devices in a communicator. Each communicator object associated with a fixed rank and CUDA device in the same NCCL communicator will be used to launch collective communications.

Stream. Point-to-point primitives and collective communication implementation perform communication and computation in a single CUDA kernel. The entire message in each communication step is divided into smaller chunks for fast synchronization. By scheduling the operation in separate CUDA streams, the NCCL call may return before the process is complete.

Topology. Based on the interconnect network, NCCL chooses from a set of topologies which include ring- and tree-based approaches.

Protocols. There are three protocols when NCCL sends data: "low latency, 8 bytes atomic store (LL)", "low latency, 128 bytes atomic store (LL128)", and Simple[34]. The bandwidth and the latency of these protocols are different because of the type of inter-node synchronization.

6.1.5 Example: Distributed Training with NCCL

Data-parallel distributed deep learning training on many GPUs is one of the more compelling use-cases for collective communications. As mentioned in Subsection 6.1.3, NCCL is used by many machine learning frameworks as a distributed training backend. Here we will examine how collectives fit into the training process and how NCCL's API makes implementing distributed training strategies simple.

In a data-parallel training arrangement, each device (GPU in the case of NCCL) holds a full model locally. The training data are split and distributed to each during each training step. NCCL's point-to-point communication can be used here. Once the data are broken up, a root process can send data using the `ncclSend`() function. Though NCCL does not provide it directly, the programmer can emulate the Scatter collective by placing `ncclSend`() in a `for` loop where the index corresponds to the target device rank. Once each device finishes receiving data using the `ncclRecv`() function, it can then run the forward pass and compute local gradients. These local gradients can be combined using an All-Reduce collective operation, or, `ncclAllReduce`() in NCCL. After the All-Reduce completes, each device will hold the updated gradients.

6.1.6 Practical Workloads and Applications

In ResNet-101's distributed training, the TensorFlow modified to use NCCL is compared with the regular distributed TensorFlow[53]. The training using NCCL was about twice as fast as standard distributed TensorFlow training. When running a distributed training job for VGG-16, NCCL leveraging RDMA networks provides a 30% improvement over NCCL using TCP networks.

6.2 Intel oneCCL

Intel oneAPI Collective Communications Library (oneCCL) is a collective communication library created with the intention of developing a single standard API that is compatible with multiple different types of hardware accelerators, ranging from CPUs to GPUs to FPGAs, in a way that makes accelerating deep learning training workloads easy. It is part of Intel's larger "oneAPI" project which incorporates a deep neural network library and a C++ standard library for accelerators, among others. The library supports Intel products such as Core CPUs, Xeon CPUs, Xeon Phi, and Intel graphics cards.

6.2.1 Architecture

Intel oneCCL is built on top of existing lower-level middleware and thus has support for InfiniBand, Ethernet, and other interconnects. More specifically it is built upon, Intel's own customized MPICH-supporting MPI library (i.e., Intel MPI Library) and libfrabric, an open-source set of libraries for fabrics. Low-level inter-node and inter-device communication is handled by these libraries for portability and interconnect support. However, oneCCL still provides a direct access to hardware (level 0) for performance critical computation and on-device communication.

[34]NCCL's environment variables. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html, Jan. 2023.

There are three key abstractions present in oneCCL that the application programmer interacts with. The first is the Communicator. Similar to other collective communication libraries such as MPI, oneCCL uses communicators for inter-rank communication and they are used to define which resources should participate in a given communication operation. However, in oneCCL, host communication and device (e.g., GPU) communication requires the use of separate communicators. Ranks in oneCCL can contain either CPUs or devices depending on the type of the communicator being considered. The second key abstraction is the Stream object, instances of which are used to pass execution context to communicator objects. Streams also contain the collective operation execution order. The third is the collective communications, the specifics of which are explained in Section 4 and Subsection 6.2.2.

6.2.2 Routines

Intel oneCCL currently has support for the All-Gather(v), All-Reduce, All-to-All(v), Barrier, Broadcast, Reduce, and Reduce-Scatter collective operations. These operations can be run asynchronously, and the status of operations can be tracked using event objects returned when operations are run. The programmer is also given some control over operation scheduling via priority fields.

6.2.3 Framework Support

Known previously as torch_ccl, PyTorch has implemented bindings for Intel oneCCL. Code to interact with oneCCL is written in Python alongside any PyTorch code, making oneCCL easily accessible to machine learning researchers who have Python-based workflows. A set of profiling tools is included to help programmers debug problems or improve the performance oft their software.

There is also a oneCCL integration available for Horovod. Unlike with PyTorch, Horovod does not expose any oneCCL details directly to the programmer. Instead, Horovod is configured to use oneCCL for collective communication using environment variables.

6.3 Alibaba ACCL

Alibaba Collective Communication Library (ACCL) is another collectives library that takes advantage of the fact that many deployments will have multiple types of fabrics available, utilize multi-rail networks, and will likely experience performance limitations primarily as a result of communication cost[49]. By focusing on support for heterogeneous interconnects, ACCL can outperform other collective communications libraries given that certain conditions are met[49]. ACCL can be used with both Tensorflow and Horovod, though it is not open source, which limits its use outside of Alibaba's cloud products.

Alibaba provides their Apsara AI Accelerator (AIACC) AI acceleration infrastructure as part of their cloud service. A recent study by Alibaba Group and Univesity of Leeds researchers found that AIACC outperformed Horovod and BytePS for certain training workloads[72]. Like ACCL, AIACC is currently proprietary.

6.3.1 Hybrid Algorithms

One important feature of ACCL is its ability to use hybrid collective algorithms. Hybrid algorithms are sets of standard collective algorithms that are combined in an effort to maximize network utilization and therefore increase overall communication performance. Choices about which algorithms to use for any given situation are made by the system using a model of the physical network derived from probing[49].

6.4 AMD RCCL

The AMD ROCm Communication Collectives Library (RCCL) is an AMD port of NCCL for communications used within single and multi-process applications running on AMD and NVIDIA GPUs. The aim of RCCL is to allow developers to run programs on both NVIDIA and AMD GPUs without rewriting the code. RCCL is a component of AMD ROCm[35] open software stack[36] and is running on the system with HIPify[37] which can convert CUDA code to HIP (Heterogeneous-Computing Interface for Portability)

[35] ROCm—Open software platform for GPU compute. https://github.com/RadeonOpenCompute/ROCm, Jan. 2023.
[36] New AMD ROCm™ information portal—ROCm v4.5 and above. https://rocmdocs.amd.com/en/latest/, Jan. 2023.
[37] HIP Programming Guide v4.5. https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html, Jan. 2023.

code automatically. RCCL supports data transfers locally over PCIe and xGMI interconnects, and over the network through InfiniBand Verbs and TCP/IP sockets. In a similar fashion to NCCL, RCCL supports GPU-to-GPU direct communication operations.

6.4.1 Architecture

RCCL includes the same collective routines as NCCL. The algorithms used in RCCL collectives are similar to those found in NCCL, and as such are implemented based on the ring and tree algorithms. Fig.8(b) shows how the CUDA code can be converted to run the application on AMD and NVIDIA GPUs: the CUDA code, for example, the NCCL library itself, is converted into the HPI code by HIPify. The HPI code can then run on NVIDIA GPUs when compiled with the NVCC compiler and can run on AMD GPU when compiled with HCC.

6.4.2 Supported Features and Workloads

Starting from PyTorch 1.8 release, the ROCm software stack—which includes the RCCL library—is provided so that developers and researchers may use PyTorch with AMD GPUs. RCCL was integrated into Tensorflow in v1.15. In both PyTorch and Tensorflow, AMD GPUs can be used for deep learning workloads such as training and inference.

Because it uses the same API as NCCL, RCCL also supports features such as communicator and topology. The stream feature in RCCL is different from the stream feature found in NCCL and uses a HIP stream instead of a CUDA stream. The multi-GPU communication in RCCL is supported by MPI. In addition, RCCL integrates NPKit[38], which is a profiling framework, to the communication routines so that RCCL can give a profiling fine-grained trace on collective routines.

6.5 Meta Gloo

Meta's Gloo[39] is a communication library for deep learning workloads which run on multiple machines. Its architecture is shown in Fig.8(c). Gloo supports both point-to-point communications and collective communications on CPUs, as well as All-Reduce, All-Gather, and Broadcast when used on GPUs.

6.5.1 Architecture

Gloo supports multi-GPU communication over interconnects such as PCIe and NVLink. Gloo supports different data transport methods for inter- and intra-node data communication, for example, TCP, RoCE, and IB for CPU-to-CPU transport and GPUDirect for GPU-to-GPU transport.

6.5.2 Supported Features and Workloads

Gloo is provided by PyTorch as a communication backend in the `torch.distributed` package. PyTorch recommends that users choose Gloo mainly for distributed training on CPU and as the fallback option for distributed training on GPU. Users can enable Gloo as the part of components of Horovod in TensorFlow, MXNet[71], and Keras[40].

Gloo uses two methods to coordinate the communications channels for CPU data transport: MPI and a custom rendezvous process[39]. Using MPI is straightforward. The MPI processes take control of the connection channels across the devices and the MPI communicator is bound to the GPU context. Another way to manage the communication across multiple machines is with Gloo's rendezvous channel setup process. Rendezvous uses a central key-value store system accessible to all processes to store the Gloo contexts. Every process has a set of keys for its peers. When a process wants to send messages to another process, it uses the key-value store system to get the information such as the corresponding IP address and port as the value.

Gloo is used as a part of the Multi-GPU communication coordination controller in Horovod, and also serves as an alternative method to manage communication and coordination among processes in Horovod.

For each cross-node All-Reduce collective operation, there are three phases that happen in order: 1) every process performs a local reduction if a process holds more than one buffer; 2) the All-Reduce collective is performed across processes; 3) like the reverse of step 1, every process broadcasts the reduction results to its buffers. Gloo provides multiple algorithm

[38]NCCL profiling kit. https://github.com/microsoft/npkit, Jan. 2023.
[39]Collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo, Jan. 2023.
[40]Keras. https://github.com/fchollet/keras, Jan. 2023.

designs for phase 2 including the Ring, Ring-chunked, Halving-Doubling, and BCube algorithms. More in-depth design details are covered in Subsection 4.1.1.

6.6 Microsoft MSCCL

The Microsoft Azure team proposed the Microsoft Collective Communication Library (MSCCL)[41] to make creating and executing custom collective communication algorithms much easier. MSCCL is made up of three components: GC3[73], TACCL[74], and SCCL[75]. GC3 provides a data-oriented domain-specific language (DSL) and a corresponding compiler to simplify GPU communication programming. TACCL is dedicated to automatically generating algorithms by guiding a synthesizer. SCCL synthesizes collective communication algorithms tailored to the hardware topology. With the three components, custom collective communication algorithms can be implemented efficiently and flexibly in MSCCL.

6.6.1 *Architecture*

For a given collective communication algorithm and physical topology, MSCCL can explore different implementations and optimizations with high-level specifications. MSCCL enables generating efficient custom communication algorithms with a chunk-oriented program, as shown in Fig.8(d). The chunk-oriented program specifies the chunk routine from source to destination. To specify chunk routing through GPUs, a DSL is used in GC3 and communication sketch is used in TACCL. After the program is created, it can be traced into a chunk-directed acyclic graph (DAG). Then the instruction DAG (distinct from the chunk DAG) is created by expanding the chunk operations into instruction operations. After that, the instruction DAG is scheduled after being compiled into an intermediate representation (IR). After the IR is generated, the MSCCL runtime executes it efficiently, since MSCCL runtime inherits NCCL's capability to set point-to-point links over various interconnects such as NVLink and PCIe.

6.6.2 *Framework Support*

Because MSCCL's API is compatible with NCCL, it is convenient to integrate the MSCCL runtime into state-of-the-art deep learning frameworks such as PyTorch by swapping out the NCCL backend with the MSCCL backend.

6.6.3 *MSCCL Runtime*

MSCCL DSL. The DSL is a chunk-oriented dataflow language that can be used to write an efficient communication kernel. The programmer specifies how chunks are routed across GPUs in this language.

MSCCL Runtime. IR is the executable file generated by MSCCL's compiler. It can be executed by the MSCCL runtime. The MSCCL runtime extends NCCL and uses NCCL's point-to-point send and receive functionality and is backward compatible with NCCL's API.

MSCCL Compiler. The MSCCL compiler traces the program to record the chunk dependencies in the chunk DAG. The compiler then performs a series of optimizations and schedules the resulting chunk DAG to thread blocks specified in the IR. The MSCCL DSL allows users to guide the compiler into optimizing and scheduling the program.

Optimization. It is important to optimize the schedules of the program to improve performance. A set of scheduling directives is used to optimize the trade-off for parallelization when scheduling instructions to multiple thread blocks. There are several aspects. 1) Multiple connections may exist in the same pair of GPUs and are labeled as channels to help distinguish different connections. The most efficient channel can then be allocated for a particular operation. 2) A transfer can be broken up into multiple smaller transfers to improve execution parallelism. 3) When multiple contiguous chunks are sent from one GPU to another, aggregating these chunks in a single transfer can reduce the latency.

6.6.4 *Practical Workloads and Applications*

MSCCL[41] has been used for inference with a public-facing language model on 8x A100 GPUs; the operations of the GPU have been accelerated by 1.22x–1.29x, depending on the input batch size. MSCCL has also been used to train a sizeable Mixture-of-Experts model on 256x A100 GPUs, providing 1.10x– 1.89x speed-up depending on the Mixture-of-Experts model architecture.

[41]MSCCL: Microsoft collective communication library. https://github.com/microsoft/msccl, Jan. 2023.

7 Experimental Comparison of Implementations

7.1 Experimental Setup

In this subsection, we run experiments on the SDSC Expanse[42] cluster to compare the performance of different collective communication libraries. The hardware details of Expanse are shown in Table 4. Taking into account availability and fairness, we select PARAM[43] from Meta, NCCL Tests[44] from NVIDIA, and OSU MPI Micro-Benchmarks (OMB)[76] as benchmarks. We choose NCCL, Gloo, MSCCL, and CUDA-aware MPI by MPICH[45,46] and UCX as the libraries of interest in our performance comparison. PARAM communication benchmarks are PyTorch-based collective benchmarks while NCCL Tests are CUDA-based collective benchmarks. We use Python 3.7, PyTorch 1.13, CUDA 11.6, NCCL 2.14.3, MSCCL 0.7.3, MPICH v4.0.2, and UCX v1.13.1. We use 16 GPUs across four nodes at most in our experiments. The rest of this section is organized as follows. We first benchmark NCCL and MSCCL with NCCL Tests, and CUDA-aware MPI with OMB. Though we use two different benchmarks to make the comparison, we keep the message size and collective routines consistent. We benchmark NCCL and Gloo with PARAM. In the experiments, all numbers are taken three times. We pick up the stable number and present the average number as the result shown below.

Table 4. SDSC Expanse Details

Specification	SDSC Expanse
GPU	4x NVIDIA V100 SMX2 (32 GB, 34.4 TFlop/s) per node
CPU	40-Core Xeon Gold 6248 2.5 GHz (384 GB DDR4 DRAM)
Interconnects	HDR InfiniBand, NVLINK 2
Topology	Hybrid Fat-Tree

7.2 NCCL Tests Benchmark with NCCL and MSCCL, and OMB with CUDA-Aware MPI

The NCCL Tests benchmark can be used to compare the latency of NCCL and MSCCL. This test includes four collectives: All-Reduce, All-Gather, All-to-All, and Broadcast. OMB can be used to compare the CUDA-aware MPI library with xCCL for the same collective routines as NCCL Tests. MSCCL is built on NCCL, and the runtime of MSCCL is an extension of NCCL. Based on the configuration, the runtime of MSCCL dynamically selects the efficient optimized algorithms or NCCL's built-in algorithms. For this reason, most tests have similar latency results for NCCL and MSCCL.

The latency of the All-Reduce collective with NCCL, MSCCL, and CUDA-aware MPI for different message sizes is shown in Fig.9(a). MSCCL can efficiently explore different algorithms, and uses the All-Pairs, Hierarchical, and Two-Step All-Reduce algorithms to support algorithmic optimizations for All-Reduce. These algorithms are described with more details in Subsection 4.2.2. In most instances, the latency of MSCCL is slightly lower than that of NCCL for both small and large message sizes. In general, the All-Reduce latency of NCCL is about 1.07 times that of MSCCL. Because MSCCL is built on top of NCCL, NCCL and MSCCL show similar performance at different numbers of GPUs. For both MSCCL and NCCL, the latency numbers for two-node tests are about three times those of single-node tests. The latency numbers of four-node tests are four times those of single-node tests. CUDA-aware MPI always has a higher latency than NCCL and MSCCL. For example, CUDA-aware MPI latency is 2x–4x slower than the one of NCCL and MSCCL when the message size is smaller than 2 MB. When the message size is larger than 2 MB, the CUDA-aware MPI latency is 4x–40x slower than the one of NCCL and MSCCL. The reason behind is that NCCL and MSCCL are optimized based on NVIDIA GPU directly on both the programming language and the algorithms.

The latency of the All-Gather collective with NCCL and MSCCL for different message sizes is shown in Fig.9(b). MSCCL and NCCL use NCCL's built-in algorithms to support the All-Gather collective communication, and the results of MSCCL are almost the same as those of NCCL. It can be observed that the latency of All-Gather is lower than that of All-Reduce. This is because All-Gather is equivalent to a

[42]Expanse SDSC. https://www.sdsc.edu/services/hpc/expanse/, Jan. 2023.
[43]PARAM. https://github.com/facebookresearch/param, Jan. 2023.
[44]NCCL tests. https://github.com/NVIDIA/nccl-tests, Jan. 2023.
[45]MPICH. https://www.mpich.org/, Jan. 2023.
[46]Official MPICH repository. https://github.com/pmodels/mpich, Jan. 2023.

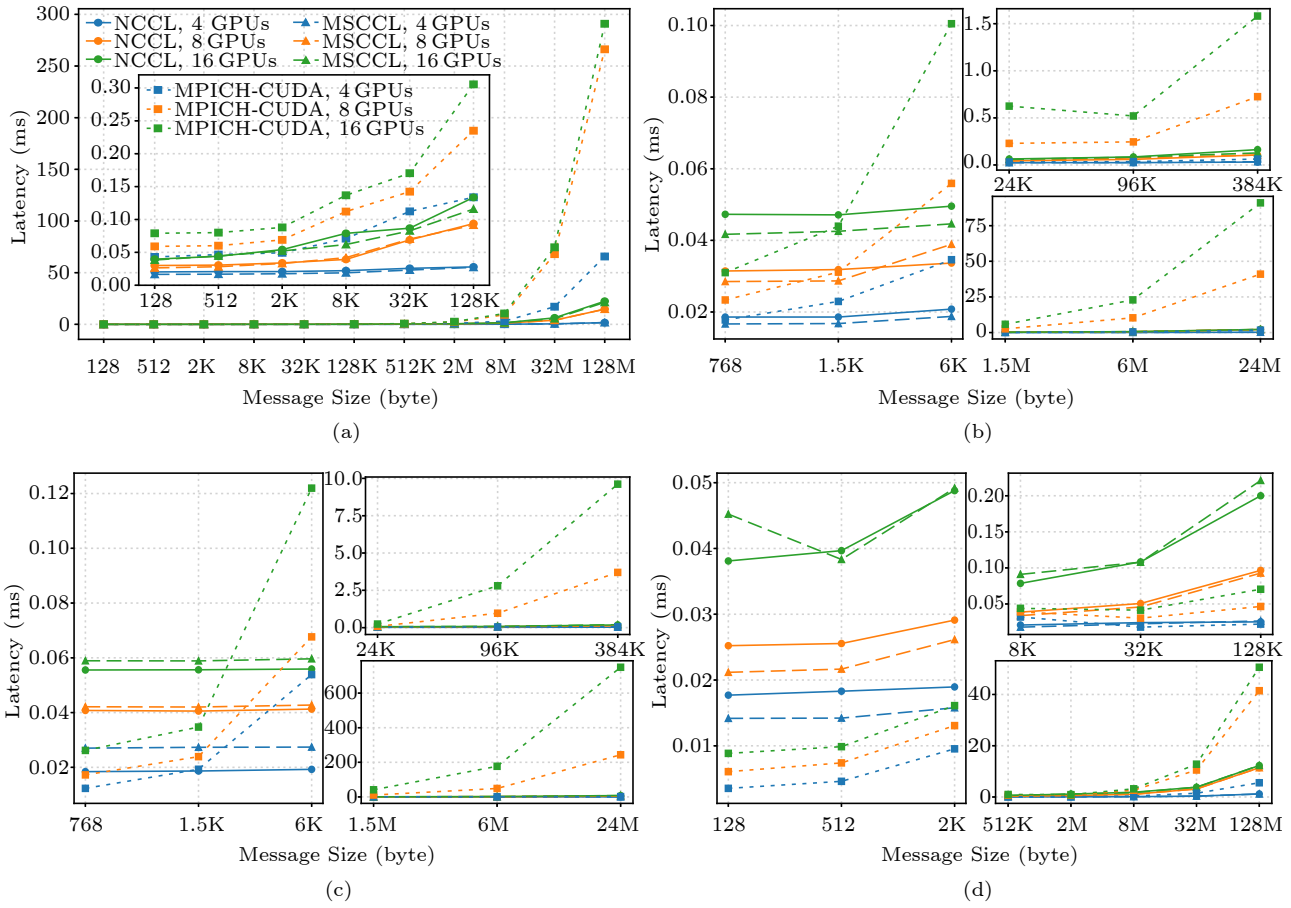


Fig.9. Latency comparison among NCCL, MSCCL, and CUDA-aware MPICH with different collectives. (a) All-Reduce. (b) All-Gather. (c) All-to-All. (d) Broadcast. NCCL and MSCCL are tested with NCCL Tests benchmark, while CUDA-aware MPICH is tested with OSU MPI Micro-Benchmarking.

Gather followed by a Broadcast, while All-Reduce can be formed by combining a reduction and a Broadcast. For small message sizes, the trend of tests for different numbers of GPUs is almost the same. When the message size gets larger, the latency results of 2 and 4 nodes are close. When the message size is smaller than 1.5 KB, CUDA-aware MPI can achieve lower latency than NCCL and MSCCL. For example, the latency of CUDA-aware MPI is 0.7x that of NCCL and MSCCL when running on 16 GPUs. When the message size becomes large, the latency of MPI can be 40x greater than that of NCCL and MSCCL when running with 16 GPUs. CUDA-aware MPI provides several algorithms and changes them automatically based on several aspects, for example, the number of nodes and the message size. Therefore, CUDA-aware MPI can achieve lower latency than NCCL and MSCCL for small message sizes.

The All-to-All routine is the third collective we compare among NCCL, MSCCL, and CUDA-aware MPI, and the latency for different message sizes is shown in Fig.9(c). We see that the overall latency results of NCCL and MSCCL among all message sizes are similar because MSCCL also uses the built-in algorithms of NCCL. Compared with the first two collectives, All-to-All's latency measurements are the largest when the message size is large. The behavior of CUDA-aware MPI is similar for All-Gather as discussed above: the latency is lower than that of both NCCL and MSCCL when the message size is smaller than 6 KB, and higher than that of both NCCL and MSCCL when the message size is large. For example, when the message size is 24 MB, CUDA-aware MPI latency is 98x slower than the ones of NCCL and MSCCL.

NCCL and MSCCL use the built-in algorithms of NCCL in Broadcast, and the latency results for different message sizes are shown in Fig.9(d). When the message size is small, the latency results scale to the number of nodes. However, when the message size be-

comes larger, the latency results for the 2-node and 4-node tests no longer increase linearly because inter-node communication becomes the bottleneck. In some cases, the latency differences between xCCL (NCCL and MSCCL) and CUDA-aware MPI are not so large. When the message size is smaller than 512 KB, CUDA-aware MPI latency is 0.2x– 0.5x of NCCL and MSCCL latency. When the message size is larger than 8 MB, CUDA-aware MPI latency is 2x– 5x slower than NCCL and MSCCL latency.

Overall, CUDA-aware MPI has higher latency than NCCL and MSCCL when the message size is larger than 1 MB. It may have lower latency than NCCL and MSCCL when the message size is smaller than 1.5 KB for All-Gather and All-to-All, and 512 KB for Broadcast.

7.3 PARAM Benchmark with NCCL and Gloo

In this subsection, we benchmark NCCL and Gloo as the communication backend with PARAM in terms of latency for three collectives: All-Reduce, All-Gather, and Broadcast. PARAM uses PyTorch as the backend engine and therefore PARAM can better reflect the performance or overhead of PyTorch with deep learning workloads. Not surprisingly, the latency of NCCL is lower than that of Gloo for all message sizes and collectives. The reason behind is that NCCL is involved in Gloo's procedure on performing collectives and they share some design concepts like communicator, as described in Subsection 6.5. Also, Gloo performs the collective operations on CPUs, which is different from NCCL that performs the collectives on GPUs. This is another reason why Gloo's performance is slower than NCCL's.

Fig.10(a) shows the latency of the All-Reduce collective with NCCL and Gloo for different message sizes. On average, the All-Reduce latency of Gloo is about 10 times to 20 times that of NCCL. In the extreme case with 16 GPUs and small message sizes, NCCL can be 40 times faster than Gloo. NCCL and Gloo also show similar latency performance when varying the number of GPUs. For small message sizes, the All-Reduce latency measured with 2-node tests is around twice that with 1-node tests. The latency measured with 4-node tests is around four times that with 1-node tests. For large message sizes, the latency numbers with 8 GPUs and 16 GPUs are similar, while the latency with 4 GPUs is much smaller.

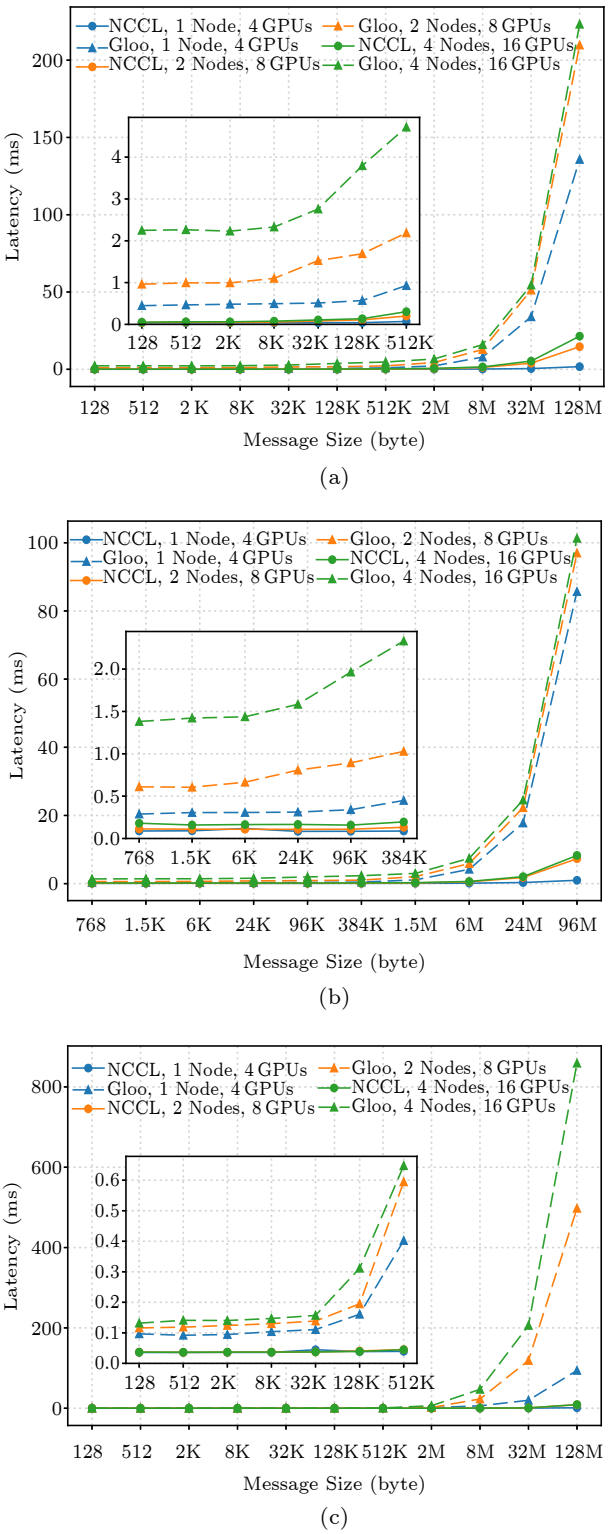


Fig.10. Latency of PARAM communication benchmarks for NCCL/Gloo with different collectives. (a) All-Reduce. (b) All-Gather. (c) Broadcast.

The reason could be that 4 GPUs residing in one node are connected with the much faster NVLink, which means there is no inter-node communication

necessary for 4-GPU tests.

The latency of the All-Gather collective with NCCL and Gloo for different message sizes is shown in Fig.10(b). The overall trend of the latency of the All-Gather collective is almost the same as that of the All-Reduce collective. The influence of the number of GPUs and message size is also very similar to the previous experiments. In the small message range, the All-Gather latency of Gloo is about three times to six times that of NCCL for different numbers of GPUs. When it goes to a large message size, Gloo is getting slower than NCCL, where its latency becomes around 10 times of NCCL's latency on average. One difference is that the latency of All-Gather is lower than that of All-Reduce, since All-Gather can be treated as a Broadcast operation from all ranks while All-Reduce can be described as a reduction and a Broadcast.

The last collective we compare between NCCL and Gloo is Broadcast. The latency for different message sizes is shown in Fig.10(c). We still see the same trend of the overall latency comparison among all message sizes. The comparison with different numbers of GPUs still shows similar trends. One thing to notice is that the NCCL's latency is very consistent among different message sizes and the number of nodes, while the Gloo's latency is less stable, especially when the message size becomes larger. For example, when running with 4 GPUs, the range of the latency for different messages from NCCL is around 0.03 ms to 1 ms but it is around 0.1 ms to 100 ms from Gloo. Compared with the first two collectives, Broadcast's latency has a wide range depending on the message size. When the message is small, the collective can finish within 1 ms because the operation is simple.

8 Discussion

Why Have xCCLs Become More Attractive Than Classic MPI in the Industry? We summarize the following possible reasons. First, due to the popularity of ML/DL in recent years, GPUs have become common in both industry and research. Therefore, the community is interested in investigating collective communication libraries for GPUs and specialized hardware, like NVIDIA's NCCL. Second, the NCCL itself is well-designed. In essence, NCCL can be treated as a simpler implementation of MPI with CUDA, which allows it to better utilize the powerful GPUs, especially for the ML/DL workloads. NCCL is easy to use, light-weight, and it provides high scalability and stable performance. Third, compared with xCCL, classic MPI communication libraries have not yet been able to make effective use of hardware acceleration, making them less attractive. For example, while many features have been added to the MPI libraries over the decades (such as GPU support), these numerous features make the libraries increasingly bloated, which harms the performance and usability of MPI.

Which Among the xCCLs Has the Best Performance? As shown and discussed in the results from Section 7, NCCL from NVIDIA shows better performance more reliably. Beyond this, researchers are still looking for opportunities where xCCL's performance can be further improved. For example, researchers from the University of California, Berkeley, found that NCCL's model parameter synchronization contains high overheads when performing distributed training and they proposed Blink[77], a set of fast collectives for distributed machine learning that reduces end-to-end training time for the image classification task up to 40%. Blink does this by dynamically generating optimal communication primitives. Another example is MSCCL. MSCCL aims to look for the best communication patterns or algorithms instead of directly using the traditional collective communication algorithms. In [74], an abstraction called communication sketch is introduced. After obtaining important information, such as hardware topology, the communication sketch will guide the synthesizer to find better algorithms to perform a certain collective. Both of these studies and our survey work shed light on potential directions of future designs for xCCLs.

What Are the Common and Distinct Design Considerations in Each xCCL? From the collective communication routine perspective, we can observe some similarities. xCCLs are evolving the classic MPI implementations into their ML versions. Among all collective routines shown in Table 1, two routines, All-Reduce and Broadcast, are supported by all xCCLs. Both collectives are useful and commonly ultilized communication patterns in ML/DL applications. Another reason is that many xCCLs are designed based on NCCL and/or use NCCL as the backend. Although they share some design considerations for supported routines, the xCCLs have diverse feature supports, which are shown in Table 3. For example, different xCCLs may adopt different open source licenses or choose to be proprietary. In terms of supported

accelerators, NCCL is the industry defacto standard for GPU-accelerated collectives, ROCm is aimed to provide support for AMD GPUs, and Intel oneCCL can support heterogeneous accelerators.

Will Current-Generation Networks Become the Bottleneck for xCCLs? From Blink[77], it is noted that communication bottlenecks between GPUs cannot be fully mitigated, for reasons such as differing server configurations and GPU job scheduling. The networking protocol can also affect scaling efficiency as TCP networking will not significantly increase performance compared with RDMA (over 90% scaling) when running Inception V3 and ResNet-101 models on Uber's Horovod. In [78], researchers argued that the network speed itself is not the bottleneck, but choosing how to utilize fast network speeds more efficiently is. The network speed is one important factor which influences the performance of xCCLs, while the real-world xCCL design is another one. The trend of xCCLs is to adopt even faster networks. In the case of ZionEX[56] from Meta, a single node accommodates 8 GPUs and each GPU has a dedicated 200 Gbps RoCE NIC. In the case of ACCL[49] from Alibaba, each node contains four Mellanox CX-5 (MT27800) 100 Gbps NICs. These real-world deployments in the industry clearly show that network speed and the appropriate software-hardware co-designs are playing significant roles in modern deep learning applications.

9 Related Work

In our survey, we provide an in-depth overview of multiple collective communication methods with a focus on industrial led collective communication solutions. Different from our survey, many other existing surveys focus more on computations, optimization, or tuning.

Many existing surveys contribute extensive overviews within collectives and tuning of parameters or optimizations. A survey of collectives provides an in-depth analysis into existing and state-of-the-art methods for optimization and tuning[79]. A paper on the implementation of collective communication on distributed-memory reviews best practices, analyzes existing algorithms, and implements tunable libraries for users[80]. A paper on the performance analysis of MPI collective operations observes and improves collective communication[81] and researchers from University of Tennessee created automatically tuned collective communications using matrix operations and Fast Fourier Transforms[82]. These surveys are unlike our survey, where we do not explicate computation acceleration and rather focus on general algorithms and industrial designs.

Distributed machine learning work also heavily relies on collective communication systems to reduce training time. A distributed machine learning survey provides an extensive overview of current methods including techniques and a review of the available systems[83]. An overview of parallel systems can guide those who are unaware of which system may suit their applications best. Recently, Wang *et al.* reviewed over 200 papers to present an overview on large-scale machine learning from a computational perspective, providing guidance in this direction[84]. The authors gave analysis on distributed deep learning, diving deep into each component that builds its structure while covering popular implementations in the community. A survey on distributed deep learning presents parallelism strategies for deep neural networks with analysis[85], while another survey takes a broader view and provides an overview on scalable deep learning systems[86]. Understanding scalability on deep learning systems is important for realizing the amount of hardware to allocate. A survey on distributed training using TensorFlow details the structure of TensorFlow for collectives and implements a design faster than Horovod-NCCL2[63]. Ouyang *et al.* provided a survey on methods to tackle communication overhead during distributed deep neural networks training from an inter-disciplinary perspective[87]. The authors focused on the structure of distributed deep neural networks and computations for collective communication, including architectures and network protocols. A performance analysis on multiple distributed deep learning frameworks (i.e., Caffe-MPI[88], CNTK[70], MXNet[71], and TensorFlow[62]) on three convolutional neural network models (i.e., AlexNet[89], GoogleNet[90] and ResNet-50[91]) focuses on collective communication bottlenecks[92]. Shi *et al.*[92] presented a very vast combination of different configurations for distributed training of Convolutional Neural Network model that provides guidance on how to select frameworks and models. The paper also reviews Convolutional Neural Network training computations.

There are also other studies to further analyze performance factors other than latency in collective communication. For example, Hoefler and Moor reported on tradeoffs between energy, memory, and runtime of different algorithms for collectives[93], al-

though it should be noted that each application of collectives will require its own specific implementation. It may not always be the case that the results can be reproduced perfectly.

10 Conclusions

This paper presented an extensive survey on industry-led collective communication libraries (xCCL) which are frequently used in distributed deep learning training workloads. We started at the physical network topology layer that underlies all communication between devices. We then discussed the data transfer algorithms used in collective routines. Next, we explored different industry solutions by comparing their feature sets and explaining real-world deep learning application use cases. We evaluated xCCL performance by running two industry-made benchmarks (NCCL Tests and PARAM). Based on our results, we explained the performance characteristics of evaluated xCCLs. We also discussed why xCCLs are gaining traction in the industry when the classic communication libraries such as MPI implementations exist. We further explained how these libraries take advantage of hardware accelerators and fast interconnects to support deep learning training workloads. Through our tests and investigation, we have determined that NCCL is currently the most mature collective communication library. We hope that future efforts will be made to explore the optimizations present in NCCL and effectively apply them in other xCCLs.

Acknowledgments On the momentous occasion of Prof. Kai Hwang' 80th birthday, we would like to express our deepest gratitude and admiration for his exceptional contributions to the field of Parallel Computing, as well as for his unwavering commitment to educating and inspiring generations of students, including ourselves. We want to thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] Hwang K, Xu Z W. Scalable Parallel Computing: Technology, Architecture, Programming. McGraw-Hill, 1998.

[2] Brown T B, Mann B, Ryder N *et al.* Language models are few-shot learners. In *Proc. the 34th Int. Conf. Neural Information Processing Systems*, Dec. 2020, pp.1877–1901.

[3] Naumov M, Mudigere D, Shi H J M *et al.* Deep learning recommendation model for personalization and recommendation systems. arXiv: 1906.00091, 2019. https://arxiv.org/abs/1906.00091, Jan. 2023.

[4] Bayatpour M, Chakraborty S, Subramoni H, Lu X Y, Panda D K. Scalable reduction collectives with data partitioning-based multi-leader design. In *Proc. the 2017 Int. Conf. High Performance Computing, Networking, Storage and Analysis* (*SC*), Nov. 2017. DOI: 10.1145/3126908.3126954.

[5] Chu C H, Lu X Y, Awan A A, Subramoni H, Hashmi J, Elton B, Panda D K. Efficient and scalable multi-source streaming broadcast on GPU clusters for deep learning. In *Proc. the 46th Int. Conf. Parallel Processing* (*ICPP*), Aug. 2017, pp.161–170. DOI: 10.1109/ICPP.2017.25.

[6] Panda D K, Lu X Y, Shankar D. High-Performance Big Data Computing. The MIT Press, 2022.

[7] Lu X Y, Islam N S, Wasi-Ur-Rahman *et al.* High-performance design of Hadoop RPC with RDMA over InfiniBand. In *Proc. the 42nd ICPP*, Oct. 2013, pp.641–650. DOI: 10.1109/ICPP.2013.78.

[8] Wasi-Ur-Rahman, Lu X Y, Islam N S, Panda D K. HOMR: A hybrid approach to exploit maximum overlapping in MapReduce over high performance interconnects. In *Proc. the 28th ACM Int. Conf. Supercomputing* (*ICS*), Jun. 2014, pp.33–42. DOI: 10.1145/2597652.2597684.

[9] Islam N S, Lu X Y, Wasi-Ur-Rahman, Panda D K. SOR-HDFS: A SEDA-based approach to maximize overlapping in RDMA-enhanced HDFS. In *Proc. the 23rd Int. Symp. High–Performance Parallel and Distributed Computing*, Jun. 2014, pp.261–264. DOI: 10.1145/2600212.2600715.

[10] Lu X Y, Shankar D, Gugnani S, Panda D K. High-performance design of Apache Spark with RDMA and its benefits on various workloads. In *Proc. the 2016 IEEE Int. Conf. Big Data*, Dec. 2016, pp.253–262. DOI: 10.1109/BigData.2016.7840611.

[11] Kalia A, Kaminsky M, Andersen D G. Using RDMA efficiently for key-value services. In *Proc. the 2014 ACM Conference on SIGCOMM*, Aug. 2014, pp.295–306. DOI: 10.1145/2619239.2626299.

[12] Shankar D, Lu X Y, Panda D K. SCOR-KV: SIMD-aware client-centric and optimistic RDMA-based key-value store for emerging CPU architectures. In *Proc. the 2019 SC*, Dec. 2019, pp.257–266. DOI: 10.1109/HiPC.2019.00040.

[13] Dragojević A, Narayanan D, Hodson O, Castro M. FaRM: Fast remote memory. In *Proc. the 11th USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2014, pp.401–414.

[14] Shankar D, Lu X Y, Islam N, Wasi-Ur-Rahman, Panda D K. High-performance hybrid key-value store on modern clusters with RDMA interconnects and SSDs: Non-blocking extensions, designs, and benefits. In *Proc. the 2016 IEEE International Parallel and Distributed Processing Symposium* (*IPDPS*), May 2016, pp.393–402. DOI: 10.1109/IPDPS.2016.112.

[15] Gugnani S, Lu X Y, Panda D K. Swift-X: Accelerating OpenStack swift with RDMA for building an efficient HPC cloud. In *Proc. the 17th IEEE/ACM International*

Symposium on Cluster, Cloud and Grid Computing, May 2017, pp.238–247. DOI: 10.1109/CCGRID.2017.103.

[16] Gugnani S, Lu X Y, Panda D K. Designing virtualization-aware and automatic topology detection schemes for accelerating Hadoop on SR-IOV-enabled clouds. In *Proc. the 2016 IEEE Int. Conf. Cloud Computing Technology and Science*, Dec. 2016, pp.152–159. DOI: 10.1109/CloudCom.2016.0037.

[17] Zhang J, Lu X Y, Panda D K. Designing locality and NUMA aware MPI runtime for nested virtualization based HPC cloud with SR-IOV enabled InfiniBand. In *Proc. the 13th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, Apr. 2017, pp.187-200. DOI: 10.1145/3050748.3050765.

[18] Chu C H, Lu X Y, Awan A A *et al*. Exploiting hardware multicast and GPUDirect RDMA for efficient broadcast. *IEEE Trans. Parallel and Distributed Systems*, 2019, 30(3): 575–588. DOI: 10.1109/TPDS.2018.2867222.

[19] Zhang J, Lu X Y, Chu C H, Panda D K. C-GDR: High-performance container-aware GPUDirect MPI communication schemes on RDMA networks. In *Proc. the 2019 IPDPS*, May 2019, pp.242–251. DOI: 10.1109/IPDPS.2019.00034.

[20] Li Y K, Qi H, Lu G, Jin F, Guo Y F, Lu X Y. Understanding hot interconnects with an extensive benchmark survey. *BenchCouncil Trans. Benchmarks, Standards and Evaluations*, 2022, 2(3): 100074. DOI: 10.1016/J.TBENCH.2022.100074.

[21] Pacheco P. An Introduction to Parallel Programming. Elsevier, 2011. DOI: 10.1016/C2009-0-18471-4.

[22] Gong Y F, He B S, Zhong J L. Network performance aware MPI collective communication operations in the cloud. *IEEE Trans. Parallel and Distributed Systems*, 2015, 26(11): 3079–3089. DOI: 10.1109/TPDS.2013.96.

[23] Brown K A, Domke J, Matsuoka S. Hardware-centric analysis of network performance for MPI applications. In *Proc. the 21st IEEE Int. Conf. Parallel and Distributed Systems (ICPADS)*, Dec. 2015, pp.692-699. DOI: 10.1109/ICPADS.2015.92.

[24] Katseff H P. Incomplete hypercubes. *IEEE Trans. Computers*, 1988, 37(5): 604–608. DOI: 10.1109/12.4611.

[25] Kalb J L, Lee D S. Network topology analysis. Technical Report SAND2008-0069. Sandia National Laboratories, Albuquerque, New Mexico, 2008. https://digital.library.unt.edu/ark:/67531/metadc845229/m2/1/high_res_d/1028919.pdf, Jan. 2023.

[26] Kim J, Kim H. Router microarchitecture and scalability of ring topology in on-chip networks. In *Proc. the 2nd Int. Workshop on Network on Chip Architectures*, Dec. 2009, pp.5–10. DOI: 10.1145/1645213.1645217.

[27] Bouknight W J, Denenberg S A, McIntyre D E, Randall J M, Sameh A H, Slotnick D L. The Illiac IV system. *Proceedings of the IEEE*, 1972, 60(4): 369–388. DOI: 10.1109/PROC.1972.8647.

[28] Cheng S H, Zhong W, Isaacs K E, Mueller K. Visualizing the topology and data traffic of multi-dimensional torus interconnect networks. *IEEE Access*, 2018, 6: 57191–57204. DOI: 10.1109/ACCESS.2018.2872344.

[29] Romanov A Y, Amerikanov A A, Lezhnev E V. Analysis of approaches for synthesis of networks-on-chip by using circulant topologies. *Journal of Physics: Conference Series*, 2018, 1050(1): 012071. DOI: 10.1088/1742-6596/1050/1/012071.

[30] Ravankar A A, Sedukhin S G. Mesh-of-Tori: A novel interconnection network for frontal plane cellular processors. In *Proc. the 1st Int. Conf. Networking and Computing*, Nov. 2010, pp.281–284. DOI: 10.1109/IC-NC.2010.30.

[31] Pham P H, Mau P, Kim C. A 64-PE folded-torus intra-chip communication fabric for guaranteed throughput in network-on-chip based applications. In *Proc. the 2009 IEEE Custom Integrated Circuits Conference*, Sept. 2009, pp.645–648. DOI: 10.1109/CICC.2009.5280748.

[32] Al-Fares M, Loukissas A, Vahdat A. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 2008, 38(4): 63–74. DOI: 10.1145/1402946.1402967.

[33] Leiserson C E, Abuhamdeh Z S, Douglas D C *et al*. The network architecture of the connection machine CM-5 (extended abstract). In *Proc. the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, Jun. 1992, pp.272–285. DOI: 10.1145/140901.141883.

[34] Valerio M, Moser L E, Melliar-Smith P M. Recursively scalable fat-trees as interconnection networks. In *Proc. the 13th IEEE Annual International Phoenix Conference on Computers and Communications*, Apr. 1994. DOI: 10.1109/PCCC.1994.504091.

[35] Nienaber W. Effective routing on fat-tree topologies [Ph. D. Thesis]. Florida State University, Tallahassee, 2014.

[36] Prisacari B, Rodriguez G, Minkenberg C, Hoefler T. Bandwidth-optimal all-to-all exchanges in fat tree networks. In *Proc. the 27th ICS*, Jun. 2013, pp.139–148. DOI: 10.1145/2464996.2465434.

[37] Li Y, Pan D. OpenFlow based load balancing for fat-tree networks with multipath support. In *Proc. the 12th IEEE International Conference on Communications*, Jun. 2013.

[38] Kim J, Dally W J, Scott S, Abts D. Technology-driven, highly-scalable dragonfly topology. In *Proc. the 2008 International Symposium on Computer Architecture*, Jun. 2008, pp.77–88. DOI: 10.1109/ISCA.2008.19.

[39] Teh M Y, Wilke J J, Bergman K, Rumley S. Design space exploration of the dragonfly topology. In *Lecture Notes in Computer Science 10524*, Kunkel J, Yokota R, Taufer M *et al*. (eds.), Springer. pp.57–74. DOI: 10.1007/978-3-319-67630-2_5.

[40] Prisacari B, Rodriguez G, Garcia M, Vallejo E, Beivide R, Minkenberg C. Performance implications of remote-only load balancing under adversarial traffic in dragonflies. In *Proc. the 8th International Workshop on Interconnection Network Architecture: On-Chip, Multi-Chip*, Jan. 2014. DOI: 10.1145/2556857.2556860.

[41] Shpiner A, Haramaty Z, Eliad S, Zdornov V, Gafni B, Zahavi E. Dragonfly+: Low cost topology for scaling datacenters. In *Proc. the 3rd IEEE International Workshop on High-Performance Interconnection Networks in the Exas-*

cale and Big-Data Era (*HiPINEB*), Feb. 2017. DOI: 10.1109/HiPINEB.2017.11.

[42] Bruck J, Ho C T, Kipnis S, Weathersby D. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Proc. the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, Aug. 1994, pp.298–309. DOI: 10.1145/181014.181756.

[43] Thakur R, Rabenseifner R, Gropp W. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 2005, 19(1): 49–66. DOI: 10.1177/1094342005051521.

[44] Pjesivac-Grbovic J. Towards automatic and adaptive optimizations of MPI collective operations [Ph.D. Thesis]. University of Tennessee, Knoxville, 2007.

[45] Huse L P. Collective communication on dedicated clusters of workstations. In *Proc. the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Sept. 1999, pp.469–476. DOI: 10.1007/3-540-48158-3_58.

[46] Barnett M, Shuler L, van De Geijn R, Gupta S, Payne D G, Watts J. Interprocessor collective communication library (InterCom). In *Proc. the IEEE Scalable High Performance Computing Conference*, May 1994, pp.357–364. DOI: 10.1109/SHPCC.1994.296665.

[47] Shroff M, Van De Geijn R A. CollMark: MPI collective communication benchmark. In *Proc. the 2000 ICS*, June 29–July 2.

[48] Rabenseifner R. Optimization of collective reduction operations. In *Proc. the 4th Int. Conf. Computational Science*, Jun. 2004. DOI: 10.1007/978-3-540-24685-5_1.

[49] Dong J B, Wang S C, Feng F *et al*. ACCL: Architecting highly scalable distributed training systems with highly efficient collective communication library. *IEEE Micro*, 2021, 41(5): 85–92. DOI: 10.1109/MM.2021.3091475.

[50] Hockney R W. The communication challenge for MPP: Intel paragon and Meiko CS-2. *Parallel Computing*, 1994, 20(3): 389–398. DOI: 10.1016/S0167-8191(06)80021-9.

[51] Benson G D, Chu C W, Huang Q, Caglar S G. A comparison of MPICH allgather algorithms on switched networks. In *Proc. the 10th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Oct. 2003, pp.335–343. DOI: 10.1007/978-3-540-39924-7_47.

[52] Almási G, Heidelberger P, Archer C J *et al*. Optimization of MPI collective communication on BlueGene/L systems. In *Proc. the 19th ICS*, Jun. 2005, pp.253–262. DOI: 10.1145/1088149.1088183.

[53] Sergeev A, Del Balso M. Horovod: Fast and easy distributed deep learning in TensorFlow. arXiv: 1802.05799, 2018. https://arxiv.org/abs/1802.05799, Jan. 2023.

[54] Goyal P, Dollár P, Girshick R *et al*. Accurate, large minibatch SGD: Training imagenet in 1 hour. arXiv: 1706.02677, 2017. https://arxiv.org/abs/1706.02677, Jan.-2023.

[55] Gupta U, Wu C, Wang X *et al*. The architectural implications of Facebook's DNN-based personalized recommendation. In *Proc. the 2020 IEEE International Symposium on High Performance Computer Architecture* (*HPCA*), Feb. 2020, pp.488–501. DOI: 10.1109/HPCA47549.2020.00047.

[56] Mudigere D, Hao Y, Huang J *et al*. Software-hardware codesign for fast and scalable training of deep learning recommendation models. In *Proc. the 49th Annual International Symposium on Computer Architecture*, Jun. 2022, pp.993–1011. DOI: 10.1145/3470496.3533727.

[57] Paszke A, Gross S, Massa F *et al*. Pytorch: An imperative style, high-performance deep learning library. In *Proc. the 33rd International Conference on Neural Information Processing Systems*, Dec. 2019.

[58] Khudia D, Huang J Y, Basu P, Deng S, Liu H, Park J, Smelyanskiy M. FBGEMM: Enabling high-performance low-precision deep learning inference. arXiv: 2101.05615, 2021. https://arxiv.org/abs/2101.05615, Jan. 2023.

[59] He K M, Zhang X Y, Ren S Q, Sun J. Deep residual learning for image recognition. In *Proc. the 2016 IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), Jun. 2016, pp.770–778. DOI: 10.1109/CVPR.2016.90.

[60] Deng J, Dong W, Socher R, Li L J, Li K, Li F F. ImageNet: A large-scale hierarchical image database. In *Proc. the 2009 CVPR*, Jun. 2009, pp.248–255. DOI: 10.1109/CVPR.2009.5206848.

[61] Dean J, Corrado G S, Monga R, Chen K, Devin M, Le Q V, Mao M Z, Ranzato M A, Senior A, Tucker P, Yang K, Ng A Y. Large scale distributed deep networks. In *Proc. the 25th Int. Conf. Neural Information Processing Systems*, Dec. 2012, pp.1223–1231.

[62] Abadi M, Barham P, Chen J *et al*. TensorFlow: A system for large-scale machine learning. In *Proc. the 12th USENIX Conference on Operating Systems Design and Implementation*, Nov. 2016, pp.265–283.

[63] Awan A A, Bédorf J, Chu C H *et al*. Scalable distributed DNN training using TensorFlow and CUDA-aware MPI: Characterization, designs, and performance evaluation. In *Proc. the 19th IEEE/ACM Int. Symp. Cluster, Cloud and Grid Computing* (*CCGRID*), May 2019, pp.498–507. DOI: 10.1109/CCGRID.2019.00064.

[64] Biswas R, Lu X Y, Panda D K. Designing a micro-benchmark suite to evaluate gRPC for TensorFlow: Early experiences. In *Proc. the 9th Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, Mar. 2018.

[65] Biswas R, Lu X Y, Panda D K. Accelerating TensorFlow with adaptive RDMA-based gRPC. In *Proc. the 25th IEEE Int. Conf. High Performance Computing* (*HiPC*), Dec. 2018, pp.2–11. DOI: 10.1109/HiPC.2018.00010.

[66] Jain A, Awan A A, Subramoni H, Panda D K. Scaling TensorFlow, PyTorch, and MXNet using MVAPICH2 for high-performance deep learning on Frontera. In *Proc. the 3rd IEEE/ACM Workshop on Deep Learning on Supercomputers* (*DLS*), Nov. 2019, pp.76–83. DOI: 10.1109/DLS49591.2019.00015.

[67] Zhang Z, Zheng S, Wang Y S *et al*. MiCS: Near-linear

scaling for training gigantic model on public cloud. *Proceedings of the VLDB Endowment*, 2022, 16(1): 37–50. DOI: 10.14778/3561261.3561265.

[68] Rajbhandari S, Rasley J, Ruwase O, He Y X. ZeRO: Memory optimizations toward training trillion parameter models. In *Proc. the 2020 SC*, Nov. 2020.

[69] Jia Y, Shelhamer E, Donahue J *et al.* Caffe: Convolutional architecture for fast feature embedding. In *Proc. the 22nd ACM International Conference on Multimedia*, Nov. 2014, pp.675–678. DOI: 10.1145/2647868.2654889.

[70] Seide F, Agarwal A. CNTK: Microsoft's open-source deep-learning toolkit. In *Proc. the 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, Aug. 2016, p.2135. DOI: 10.1145/2939672.2945397.

[71] Chen T Q, Li M, Li Y T, Lin M, Wang N Y, Wang M J, Xiao T J, Xu B, Zhang C Y, Zhang Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv: 1512.01274, 2015. https://arxiv.org/abs/1512.01274, Jan. 2023.

[72] Lin L X, Qiu S H, Yu Z Q, You L, Long X, Sun X Y, Xu J, Wang Z. AIACC-training: Optimizing distributed deep learning training through multi-streamed and concurrent gradient communications. In *Proc. the 42nd IEEE Int. Conf. Distributed Computing Systems*, Jul. 2022, pp.853–863. DOI: 10.1109/ICDCS54860.2022.00087.

[73] Cowan M, Maleki S, Musuvathi M *et al.* MSCCL: Microsoft collective communication library. arXiv: 2201.11840, 2022. https://arxiv.org/abs/2201.11840v1, Jan. 2023.

[74] Shah A, Chidambaram V, Cowan M *et al.* TACCL: Guiding collective algorithm synthesis using communication sketches. In *Proc. the 2023 USENIX Symposium on Networked Systems Design and Implementation*, April 2023.

[75] Cai Z X, Liu Z Y, Maleki S, Musuvathi M, Mytkowicz T, Nelson J, Saarikivi O. Synthesizing optimal collective algorithms. In *Proc. the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2021, pp.62–75. DOI: 10.1145/3437801.3441620.

[76] Panda D K, Tomko K, Schulz K, Majumdar A. The MVAPICH project: Evolution and sustainability of an open source production quality MPI library for HPC. In *Proc. the Workshop on Sustainable Software for Science: Practice and Experiences*, Nov. 2013.

[77] Wang G H, Venkataraman S, Phanishayee A *et al.* Blink: Fast and generic collectives for distributed ML. In *Proc. the 2020 Machine Learning and Systems 2*, Mar. 2020, pp.172–186.

[78] Zhang Z, Chang C K, Lin H B *et al.* Is network the bottleneck of distributed training? In *Proc. the 2020 Workshop on Network Meets AI & ML*, Aug. 2020, pp.8–13. DOI: 10.1145/3405671.3405810.

[79] Wickramasinghe U, Lumsdaine A. A survey of methods for collective communication optimization and tuning. arXiv: 1611.06334, 2016. https://arxiv.org/abs/1611.06334, Jan. 2023.

[80] Chan E N, Heimlich M, Purkayastha A, van de Geijn R. Collective communication: Theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 2007, 19(13): 1749–1783. DOI: 10.1002/cpe.1206.

[81] Pješivac-Grbović J, Angskun T, Bosilca G, Fagg G E, Gabriel E, Dongarra J J. Performance analysis of MPI collective operations. *Cluster Computing*, 2007, 10(2): 127–143. DOI: 10.1007/s10586-007-0012-0.

[82] Vadhiyar S S, Fagg G E, Dongarra J. Automatically tuned collective communications. In *Proc. the 2000 ACM/IEEE Conference on Supercomputing*, Nov. 2000. DOI: 10.1109/SC.2000.10024.

[83] Verbraeken J, Wolting M, Katzy J *et al.* A survey on distributed machine learning. *ACM Computing Surveys*, 2020, 53(2): 30. DOI: 10.1145/3377454.

[84] Wang M, Fu W J, He X N, Hao S J, Wu X D. A survey on large-scale machine learning. *IEEE Trans. Knowledge and Data Engineering*, 2022, 34(6): 2574–2594. DOI: 10.1109/TKDE.2020.3015777.

[85] Ben-Nun T, Hoefler T. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys*, 2019, 52(4): Article No. 65. DOI: 10.1145/3320060.

[86] Mayer R, Jacobsen H A. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys*, 2020, 53(1): Article No. 3. DOI: 10.1145/3363554.

[87] Ouyang S, Dong D Z, Xu Y M, Xiao L Q. Communication optimization strategies for distributed deep neural network training: A survey. *Journal of Parallel and Distributed Computing*, 2021, 149: 52–65. DOI: 10.1016/j.jpdc.2020.11.005.

[88] Lee S, Purushwalkam S, Cogswell M, Crandall D, Batra D. Why M heads are better than one: Training a diverse ensemble of deep networks. arXiv: 1511.06314, 2015. https://arxiv.org/abs/1511.06314, Jan. 2023.

[89] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks. In *Proc. the 25th International Conference on Neural Information Processing Systems*, Dec. 2012, pp.1097–1105.

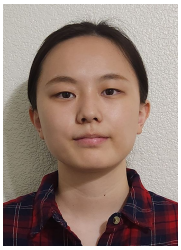
[90] Szegedy C, Liu W, Jia Y Q, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going deeper with convolutions. In *Proc. the 2015 CVPR*, Jun. 2015. DOI: 10.1109/CVPR.2015.7298594.

[91] He K M, Zhang X Y, Ren S Q, Sun J. Deep residual learning for image recognition. In *Proc. the 2016 CVPR*, Jun. 2016, pp.770–778. DOI: 10.1109/CVPR.2016.90.

[92] Shi S H, Wang Q, Chu X W. Performance modeling and evaluation of distributed deep learning frameworks on GPUs. In *Proc. the DASC/PiCom/DataCom/CyberSciTech*, Aug. 2018, pp.949–957. DOI: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.000-4.

[93] Hoefler T, Moor D. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Supercomputing Frontiers and Innovations*, 2014, 1(2): 58–75. DOI: 10.14529/jsfi140204.

Adam Weingram is a Ph.D. student in the Parallel and Distributed Systems Laboratory (PADSYS Lab) of Department of Computer Science and Engineering at the University of California, Merced (UCM). Previously, he received his B.S. degree in computer science from UCM. His research interests include systems for machine learning and applications of computer science in remote sensing.



Yuke Li is a Ph.D. student in the PADSYS Lab of Department of Computer Science and Engineering at the University of California, Merced. Previously, she received her M.S. degree and B.E. degree from The University of Edinburgh, Edinburgh, and Sun Yat-sen University (SYSU), Guangzhou, in 2020 and 2019, respectively. Her research interests include high-performance computing, MPI, RDMA, and DPU. She is a student member of ACM.

Hao Qi is a M.S. student in the PADSYS Lab of Department of Computer Science and Engineering at the University of California, Merced. Previously, he received his M.S. degree in biomedical engineering from the Ohio State University, Columbus, and B.S. degree in bioscience from Nankai University, Tianjin, in 2021 and 2019, respectively. His research interests mainly include high-performance computing, parallel computing, and system for machine learning.

Darren Ng is a M.S. student in the PADSYS Lab of Department of Computer Science and Engineering at the University of California, Merced. Previously, he received his B.E. degree in computer science from UCM. His research interests include deep learning, convolutional neural networks, and cloud computing.

Liuyao Dai is a Ph.D. student in the PADSYS Lab of Department of Computer Science and Engineering at the University of California, Merced. Previously, he received his M.S. degree in electrical engineering from Southern University of Science and Technology, Shenzhen, and B.S. degree from Huazhong University of Science and Technology, Wuhan, in 2022 and 2018, respectively. His research interests include the co-design of computer software and hardware, and GPU-based systems.

Xiaoyi Lu is an assistant professor in the Department of Computer Science and Engineering at the University of California, Merced (UC Merced), Merced. He is leading the Parallel and Distributed Systems Laboratory (PADSYS Lab) at UC Merced. His current research interests include parallel and distributed computing, high-performance interconnects, advanced I/O technologies, big data analytics, cloud computing, and deep learning system software. He has published one book and more than 100 papers in prestigious international conferences, workshops, and journals with multiple Best (Student) Paper Awards or Nominations. He has delivered more than 100 times of invited talks, tutorials, and presentations worldwide. Many of Dr. Lu's research outcomes (e.g., PMIdioBench, HiBD, MVAPICH2-Virt, DataMPI) are made publicly available to the community and are currently being used by hundreds of organizations all over the world. Dr. Lu has received a Meta/Facebook Faculty Research Award and a Google Research Award. Dr. Lu's research has also been funded by the National Science Foundation (NSF) of USA. Dr. Lu is a member of ACM and IEEE. More details about Dr. Lu can be found at http://faculty.ucmerced.edu/luxi.