Computing
https://doi.org/10.1007/s00607-021-01029-2

SPECIAL ISSUE ARTICLE

Analyzing the impact of the MPI allreduce in distributed training of convolutional neural networks

Adrián Castelló[1] · Mar Catalán[2] · Manuel F. Dolz[2] · Enrique S. Quintana-Ortí[1] · José Duato[1]

Received: 30 April 2021 / Accepted: 22 October 2021

Abstract

For many distributed applications, data communication poses an important bottleneck from the points of view of performance and energy consumption. As more cores are integrated per node, in general the global performance of the system increases yet eventually becomes limited by the interconnection network. This is the case for distributed data-parallel training of convolutional neural networks (CNNs), which usually proceeds on a cluster with a small to moderate number of nodes. In this paper, we analyze the performance of the Allreduce collective communication primitive, a key to the efficient data-parallel distributed training of CNNs. Our study targets the distinct realizations of this primitive in three high performance instances of Message Passing Interface (MPI), namely MPICH, OpenMPI, and IntelMPI, and employs a cluster equipped with state-of-the-art processor and network technologies. In addition, we apply the insights gained from the experimental analysis to the optimization of the TensorFlow framework when running on top of Horovod. Our study reveals that a careful selection of the most convenient MPI library and Allreduce (ARD) realization accelerates the training throughput by a factor of 1.2× compared with the default algorithm in the same MPI library, and up to 2.8× when comparing distinct MPI libraries in a number of relevant combinations of CNN model+dataset.

✉ Adrián Castelló
adcastel@disca.upv.es

Mar Catalán
catalama@uji.es

Manuel F. Dolz
dolzm@uji.es

Enrique S. Quintana-Ortí
quintana@disca.upv.es

José Duato
jduato@disca.upv.es

1 Universitat Politècnica de València, Valencia, Spain

2 Universitat Jaume I, Castellón de la Plana, Castellón, Spain

Published online: 10 January 2022

Keywords Message passing interface (MPI) · Collective communication primitives · Allreduce · Deep learning · Distributed training

Mathematics Subject Classification 6804

1 Introduction

Computationally demanding applications are frequently executed in large high performance computing (HPC) facilities, in order to tackle their time complexity via additional hardware resources. However, this type of acceleration is often limited by non-negligible overheads introduced by data movement. For many distributed algorithms, how and when the data are moved, between processes running in different nodes over an interconnection network, determines the global performance [6]. Moreover, data movement is also a significant contributor to energy consumption in current facilities [14]. For distributed training of convolutional neural networks (CNNs) [3,9], communication is a crucial factor, with a potentially high impact on the performance of this process, when conducted on a cluster of computers.

TensorFlow (TF)[1] [1], and PyTorch[2] [10] are nowadays the two most widely-used *distributed* training frameworks for CNNs. When executed on a cluster, they both exploit data parallelism by partitioning and distributing the workload among the processes/cluster nodes across the batch dimension (i.e., the training samples) [2]. As a result, at each iteration of the training procedure, once the gradient of each layer is computed, all processes have to combine (specifically, reduce via addition) their new model parameter values, yielding the same initial state for the next training step [2].

MPI (Message Passing Interface) [15] is leveraged by distributed training frameworks such as TF, TF+Horovod[3] [13], and PyTorch as the underlying communication layer. Among different communication primitives, the MPI application programming interface (API) comprises the `MPI_Allreduce` primitive for the reduce+broadcast communication needed in distributed data-parallel training of CNNs.

In this paper, we extend our previous work in [4] with a complete evaluation of `MPI_Allreduce` for three popular instances of MPI, analyzing the impact of this primitive on the distributed training of CNNs, using a top-of-the-shelf cluster with nodes connected via an EDR Infiniband interconnection network. In addition, we complete this study by targeting a variety of scenarios including four CNN models and two datasets with distinct batch sizes. In particular, our work makes the following contributions:

- We identify the Allreduce (ARD) algorithms underlying the realizations of `MPI_Allreduce` in MPICH, OpenMPI, and IntelMPI.
- We demonstrate the performance gap between the theoretical communication throughput of these algorithms, and the real execution performance. Moreover, we highlight some details that may cause the deviation.

[1] https://www.tensorflow.org.

[2] https://pytorch.org.

[3] https://github.com/horovod/horovod.

- We perform a complete evaluation of the ARD algorithms on a small cluster consisting of 8 nodes, equipped with Intel Xeon Gold 5120 processors connected via a EDR high performance network.
- We illustrate the practical benefits of a careful selection of the communication library and ARD algorithm for the acceleration of distributed data-parallel CNN training using TF+Horovod for the distributed CNN scenarios.

In this paper, we overcome the limitations presented in our previous work by re-visiting the MPI libraries using an state-of-the-art cluster. In [4], a modern technology was simulated by applying a scaling factor to the experimental results in order to "accelerate" the computation power of the nodes so that the balance between communication-computation was compensated. In this work, we directly apply the ARD optimization without acting over the experimental results.

Although graphic processor units (GPUs) are commonly used for distributed training of CNNs, our experimental analysis is focused on "non-accelerated" clusters. For this purpose, we spawn one MPI-rank per node and exploit the inter-node parallelism via OpenMP threads. This approach avoids the growth in the communication overhead that arises when increasing the number of nodes, and allows to augment the batch size per process without exceeding the memory capacity of the nodes. At this point, we emphasize the interest of companies like Facebook to exploit idle workload cycles in their HPC facilities, which leave a significant number of (general-purpose) multicore CPUs to perform distributed training during off-peak periods [8].

The rest of the paper is organized as follows. In Sect. 2 we review some previous works and compare them with the contributions presented in this study. In Sect. 3 we re-visit the family of ARD algorithms, together with a theoretical analysis of their arithmetic and communication costs. In addition, we expose the difference in performance attained by the real execution of this primitive in comparison with the theoretical models. In Sect. 4, we carry out a complete experimental evaluation of the realization of these algorithms in the three target MPI libraries. In Sect. 5 we extend our experimental analysis to the distributed training of CNNs using TF+Horovod. In Sect. 6 we discuss other aspects that are not directly tackled in this paper. Finally, in Sect. 7 we close the paper with a few concluding remarks and a sketch of future research lines.

2 Related work

The performance optimization of the MPI collectives has been a constant research and development topic since the formulation of the standard and the first prototype implementations. From the point of view of performance modeling, the works in [5] and [12] tried to offer accurate cost models for different collective communication primitives. While the former describes several algorithmic implementations for the MPI collective primitives depending of the message size, the latter goes beyond the theory and analyzes the library source code.

From the software point of view, there exist other works that aim at improving the performance of MPI collectives. In [16], the authors evaluate distinct algorithmic

realizations of collective communications to estimate the best option depending on the number of nodes and the message size. This knowledge is then applied to the selection module of the MPICH library. The authors in [7] present a hierarchical approach for reduction-based collective primitives, presenting notable performance improvements in small- and medium-size clusters.

Manual fine-tuning for collective communication via multilayer perceptrons was also analyzed in [18], using earlier versions of OpenMPI (1.4) and MPICH2. In [17], the same authors improve the performance of the reduction collective by targeting the intra-node parallelization. Both works were applied to the batch pattern training algorithm and also improve the performance by reducing the number of collective communications (as Horovod does).

Our work does not aim to model the performance of the MPI algorithms. Instead, our goal is to highlight the significant gap between theory and practice for several well-known algorithms. In this sense, we align our efforts with the work done in [17,18], with state-of-the-art MPI library implementations (as [18] suggests), and apply the insights gained from our study to evaluate the performance of distributed CNN training using TF+Horovod.

3 Allreduce algorithms

3.1 A family of Allreduce algorithms

MPI is the *de facto* standard API for message-passing in distributed-memory systems, yet the standard only specifies the functionality that must be covered by the realization and the interface of the routines (primitives). Over the past decades, a fair number of MPI libraries have been developed, following the evolution of network technology and software, resulting in MPICH,[4] OpenMPI,[5] IntelMPI,[6] and MVAPICH[7] as some of the most relevant instances of the API.

There exist a variety of algorithms that can be used to implement an ARD collective communication. Most MPI libraries aim to optimize performance by selecting among these algorithmic variants at execution time, depending on features such as the message size, the number of MPI ranks (processes), the network topology, etc. The most popular algorithms for ARD include (see [5,7,16]):

1. `RDB` (Recursive doubling): Initially, the processes that are a distance 1 apart exchange (and reduce) their data. Next, those processes that are a distance 2 apart do the same with the complete data they own after the first exchange. This is repeated for the processes which are at distance 4 apart, then those at distance 8 apart,. . . till all the processes have received all the data.
2. `RSA` (Rabenseifner's algorithm): This option performs a Reduce-Scatter followed by an Allgather exchange [16].

[4] https://www.mpich.org.

[5] https://www.open-mpi.org.

[6] https://software.intel.com/content/www/us/en/develop/tools/mpi-library.html.

[7] https://mvapich.cse.ohio-state.edu.

Table 1 Theoretical costs of common ARD algorithms

Id.	Alg.	Latency ($\times\alpha$)	Bandwidth ($\times\beta^{-1}$)	Arithmetic ($\times\gamma^{-1}$)
1	RDB	$\log p$	$n \log p$	$n \log p$
2	RSA	$2 \log p$	$2n \frac{p-1}{p}$	$n \frac{p-1}{p}$
3	LIN	$2(p-1)$	$2n (p-1)$	$n (p-1)$
4	RNG	$2(p-1)$	$2n \frac{p-1}{p}$	$n \frac{p-1}{p}$
5	SRG	$(2p+s-3)$	$(p+s-2)\frac{n}{s}$ $+(p-1)\frac{n}{p}$	$(p+s-2)\frac{n}{s}$

3. LIN (Linear): This basic scheme initially reduces the data into a root process, using $p-1$ point-to-point (P2P) messages, to then broadcast the result from there, using $p-1$ additional P2P messages.
4. RNG (Ring): A pairwise-exchange algorithm [16] is used for the Reduce-Scatter phase, and a ring algorithm is applied for the Allgather.
5. SRG (Segmented ring): This is a segmented variant of RNG that divides the messages into s segments.

3.2 Theoretical cost models for ARD

Let us consider an ARD primitive executed over $n \cdot p$ "data items", evenly distributed across a platform consisting of p nodes, with a single MPI rank or process per node. Moreover, assume that the network links are characterized by a latency α (in seconds) and a bandwidth β (in data items per second); and assume also that the interconnection network supports simultaneous transfers between all pairs of nodes at full link bandwidth. Finally, consider that each node can operate at a rate of γ additions per second. With these premises, Table 1 displays the theoretical costs of the ARD algorithms, divided into their latency, bandwidth, and arithmetic components. (For simplicity, in the table we assume that p is a power of 2.)

3.3 Theoretical cost analysis for ARD

The formulae in Table 1 offer a straight-forward tool to expose the theoretical properties and behaviour of the distinct ARD algorithms. To illustrate this, consider a cluster characterized by the following parameters: $\alpha = 2\ \mu$s, $\beta = 11,770$ MB/s (that is, $11,770 \cdot 10^6$ bytes/s),[8] and $\gamma = 8 \cdot 10^9$ (32-bit floating point) operations per second or FP32 flops/s. (These values were experimentally determined for the cluster employed in the practical evaluation in this paper; see Sect. 4.) Furthermore, consider this cluster comprises p nodes, and the vectors to be reduced consist of n FP32 numbers in each cluster node. In Fig. 1, we report the performance of the ARD algorithms, measured in MiB/s, when fixing three of the following parameters: message dimension $n = 2^{20}$

[8] For β we can transform data items/s into MB/s by simply taking into account the storage requirements of data type. The same holds to transform the arithmetic rate γ from arithmetic operations/s into bytes/s.

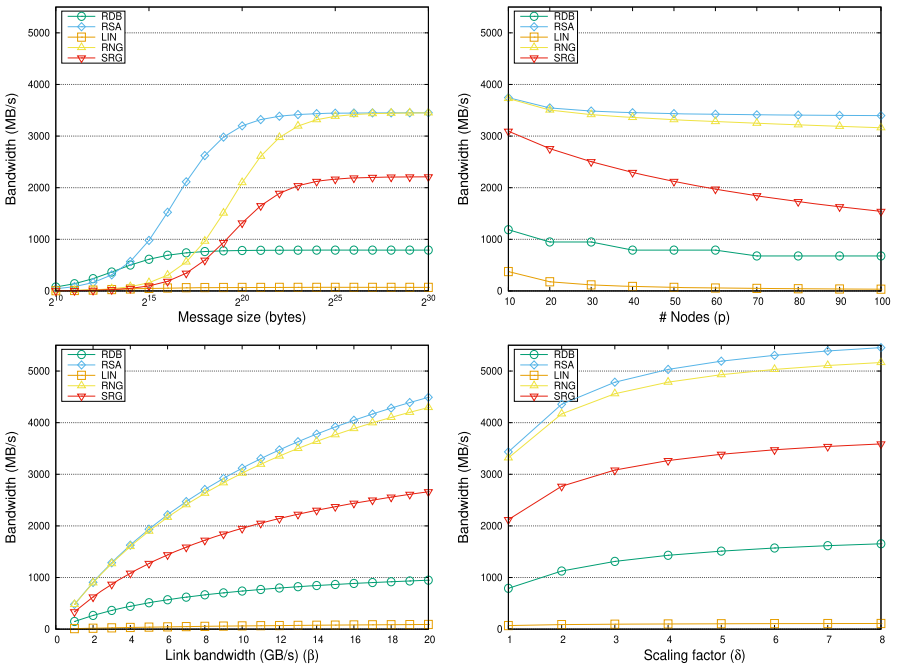

Fig. 1 Estimation of cost given by the theoretical models for the ARD algorithms. In the top-left plot, the number of nodes $p = 50$ (one MPI-rank per node); in the top-right plot, the message size $n = 16 \cdot 2^{20}$ bytes. In the bottom-right plot, δ is a factor that multiplies the reference value $\gamma = 8 \cdot 10^9$ FP32 flops/s. For SRG, $s = 64$

bytes, number of nodes $p = 50$, link bandwidth $\beta = 11,770$ MiB/s, and scaling factor $\delta = 1$; while varying the remaining one.

In all the experiments in this paper, the throughput rates for the ARD algorithms (in MiB/s) are computed by dividing the message size n (in bytes) by the time required to complete the reduction. However, the actual number of bytes that are transferred during an ARD is considerably larger than n; see Table 1. On the one hand, this explains the lower performance of the ARD collective compared with the theoretical P2P bandwidth. On the other hand, compared with an evaluation based on the standard time metric, this MiB/s rate sets an upper bound on the ARD performance, facilitating the identification of the asymptotic throughput.

This first study exposes that, for these particular values of link latency/bandwidth, arithmetic rate, and number of nodes/processes, the best algorithm largely depends on the message size. In addition, RSA and RNG show higher scalability with the number of nodes. Also, these two variants report significant gains when increasing the link bandwidth and the arithmetic capacity. Therefore, at least in theory, they are the best candidates to deliver the highest performance in the distributed training of CNNs.

3.4 Experimental cost analysis for ARD

Although the theoretical costs of the ARD algorithms provide some useful hints, Fig. 2 demonstrates that there exist important deviations between the theoretical behaviour

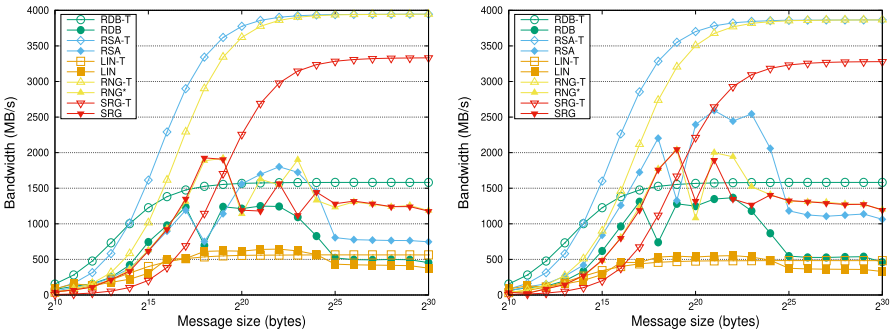

Fig. 2 Theoretical cost (lines labelled with the "-T" suffix) and real performance of the ARD algorithms in OpenMPI using 7 and 8 MPI-ranks (one per node) (left and right, respectively)

and the real performance of the ARD algorithms implemented in OpenMPI.[9] (A similar observation holds for other MPI instances.) This test demonstrates the need for a careful study of all the ARD alternatives before selecting the most convenient realization.

Several aspects can be highlighted from the analysis of the plots in Fig. 2:

- RSA (chosen as the baseline or default option by almost all MPI libraries) is rarely the best option for messages of size larger than 32 MiB;
- the most appropriate algorithm varies depending on the numbers of processes; and
- RNG and SRG are clearly the best options for large messages, of size greater than 2^{25} bytes.

The following reasons explain some of the deviations between theory and practice:

- The theoretical models assume that the bandwidth link works at full throughput independently of the message size.
- The theoretical models do not take into account software/hardware limitations (e.g., number and size of buffers in the underlying implementation).
- In theory, the models assume a perfect overlap between communications inside a process (thus, e.g., the cost of IRecv+Send is estimated as that of a Send); however, the practice may differ from that perfect implementation.

Let us analyze in more detail the last factor. For that purpose, consider the results in Fig. 3, where the line labeled as "THEO" shows the theoretical cost of the algorithm; the line "REAL" is the actual performance offered by the implementation; and the lines labeled as "OVLP" and "NOVL" respectively correspond to the theoretical costs if either the internal communications can be fully overlapped or there is no overlap. For the RSA algorithm, in the left-hand side plot of the figure, we observe that OVLP (where all the communication is overlapped) is the best case for messages of up to 2^{24} bytes. Beyond that size, any of the approximations yield a higher transfer rate than the theoretical cost. It is also notable the poor performance attained by the implementation, which only outperforms the non-overlapping code for messages of size larger than 2^{20} bytes. For RNG, in the right-hand side plot, THEO lies in between the rates offered by OVLP and NOVL, and REAL is closer to NOVL. This implies that, although the implementation aims to overlap most computation and communication, this is not possible.

[9] The cluster that was used for this experiment will be presented later, in Sect. 4.

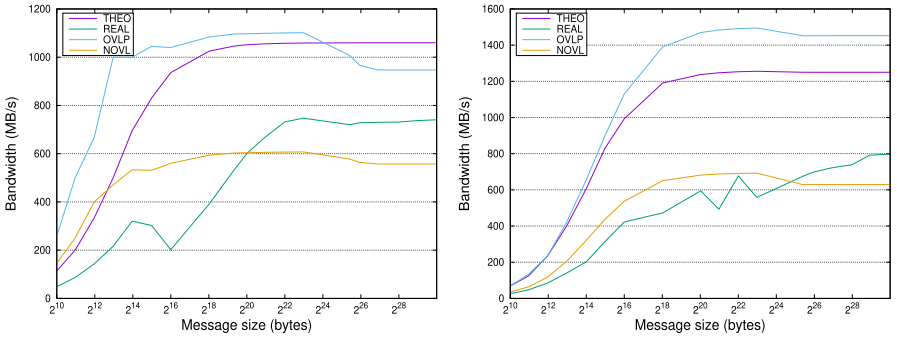

Fig. 3 Theoretic and real performance of the ARD algorithms on OpenMPI for RSA and RNG (left and right, respectively) using 8 MPI-ranks (one per node)

Table 2 ARD algorithms in MPICH, OpenMPI, and IntelMPI. The asterisk in the RNG-based algorithms indicates that, although OpenMPI classifies two of its variants as "ring" algorithms, the actual implementation corresponds to a different communication scheme

Id.	Alg.	MPICH	OpenMPI	IntelMPI
1	RDB	✓	✓	✓
2	RSA	✓	✓	✓
3	LIN	–	✓	–
4	RNG	–	✓*	✓
5	SRG	–	✓*	–

This initial evaluation exposes the importance of analyzing in detail the performance of the distinct ARD algorithms, prior to utilizing a particular instance from an application that heavily depends on this type of collective communication, as is the case of distributed CNN training.

We close this section by noting that a deeper study of the factors that underlie this theory-vs-real performance deviation is out of scope for this work.

4 Experimental selection of `MPI_Allreduce` in MPI libraries

In this section we evaluate the ARD algorithms that are implemented in the three MPI libraries targeted in this work: MPICH 3.3.1, OpenMPI 4.1, and IntelMPI 2020. For the ARD primitives, these instances of the API instantiate the algorithms listed in Table 2.

4.1 Characterization of the cluster

The experiments in this work were conducted on the ALTEC cluster, a platform consisting of 8 nodes, equipped each with two Intel Xeon Gold 5120 CPU processors (14 cores, running with a nominal frequency of 2.20 GHz), and connected via an Infiniband EDR network. A single MPI rank (process) was mapped per node in all cases, and we repeated the experiments using 7 and 8 nodes. The first cluster configuration

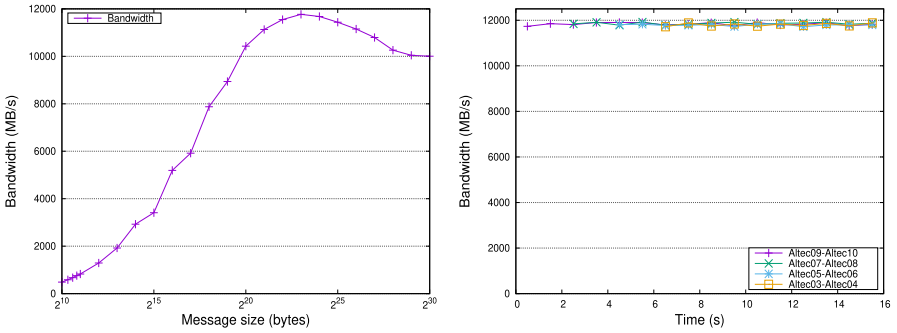

Fig. 4 P2P network performance in the ALTEC cluster measured using the ping-pong benchmarking test in OpenMPI on a single pair of nodes (left); and four pairs of nodes, with each pair starting the test in 2-second intervals with a fixed message size $n = 16$ MiB (right)

was selected because some of the ARD algorithms largely benefit from the number of MPI ranks being an integer power of two. However, this is not always possible or convenient when running an application, such as a distributed deep learning framework, on a real cluster facility.

In order to report the actual performance of the network links and, at the same time, expose some of the reasons for the deviation of the algorithm costs discussed in the previous section, we performed a simple ping-pong benchmark. The left-hand side plot in Fig. 4 offers the results from this P2P characterization test using OpenMPI, showing that the Infiniband EDR network delivers a sustained link bandwidth $\beta \approx 11.7$ GB/s when the message size is larger than 4 MiB (that is, $4 \cdot 2^{20}$ bytes). This ping-pong test also offered an estimated link latency of $\alpha \approx 2$ μs. Similar values were obtained when the ping-pong test was run on top of the other two MPI libraries. Here, the drop in link bandwidth for messages of size larger than 32 MiB is caused by the internal implementation of the MPI libraries, in aspects such as buffering. In order to approximate γ, we used a straightforward code that performed the summation of two long vectors with FP32 numbers (similar to the STREAM benchmark[10]).

The right-hand side plot in Fig. 4 demonstrates that the network can sustain the P2P bandwidth when up to 4 pairs of nodes run the ping-pong test (and, therefore, exchange messages) simultaneously. This is a fundamental assumption of the theoretical models presented in Sect. 3.2.

4.2 Individual evaluation

When the user provides no specific indication, the MPICH, OpenMPI, and IntelMPI libraries choose the ARD algorithm that is employed at runtime, depending only on the message size and number of processes. This selection can be quite simple (e.g., taking into account the theoretical cost,) or rather evolved (e.g., made via test-and-error approaches resulting in a long list of if-else statements).

[10] https://www.cs.virginia.edu/stream/.

In this subsection, we evaluate the ARD algorithms available in the three target MPI libraries. For MPICH, this includes only two of the algorithms listed in Table 2 (ids. 1, 2); OpenMPI implements all the algorithms ; and IntelMPI comprises three of the ARD algorithms in the table (ids. 1, 2, 4) plus the following 9 additional variants (numbered with ids. 6–14 next):

6. Shumilin's ring (SHR),
7. Reduce+Broadcast (R+B),
8. Topology-aware Reduce+Broadcast (TA-R+B),
9. Binomial Gather+Scatter (BGS),
10. Topology-aware binominal Gather+Scatter (TA-BGS),
11. Knomial (KNO) ,
12. Topology-aware SHM-based flat (TA-SHM BF),
13. Topology-aware SHM-based Knomial (TA-SHM KNO), and
14. Topology-aware SHM-based Knary (TA-SHM KNA).

Several of these variants correspond to segmented algorithms, for which IntelMPI sets the segmentation parameter to $s = 64$. Furthermore, the SRG algorithm in OpenMPI employs a non-segmented ring for messages of up to 8 MiB and, from that point, selects s to set the segment size to 1 MiB.

The ARD algorithm can be selected via environment variables for MPICH or, alternatively, via command arguments passed to `mpirun` in the case of OpenMPI and IntelMPI. Without an explicit indication from the user, the library follows its internal selection mechanism. In our plots, this default selection is labeled as "AUTO".

Figure 5 reports the results from this individual evaluation of the MPI libraries leading to the following observations:

- MPICH: AUTO selects RDB, which turns to be a suboptimal option as it delivers a much lower performance for messages larger than 2^{15} bytes (32 KiB) for both 7 and 8 nodes.
- OpenMPI: AUTO selects the RSA algorithm in both scenarios, which corresponds to the best for some message sizes. However, it is suboptimal compared with SRG or RNG for message sizes larger than 2^{25} bytes and 8 nodes. These two algorithms are also the best choice for message sizes from 2^{17} to 2^{19} and 7 nodes.
- IntelMPI: For 8 nodes, AUTO makes a fairly good selection, except for message sizes larger than 2^{25}, where this should be changed to RNG. For 7 nodes, RNG delivers the best performance while, unfortunately, AUTO adopts RSA.

As a short summary from this study, we conclude that, for this particular cluster configurations, MPICH misses the optimal configuration for medium and large message sizes, independently of the number of nodes; OpenMPI fails in selecting RSA for RNG or SRG for the largest messages; and IntelMPI offers a mixed optimization level: good for 8 nodes but rather suboptimal for 7 nodes.

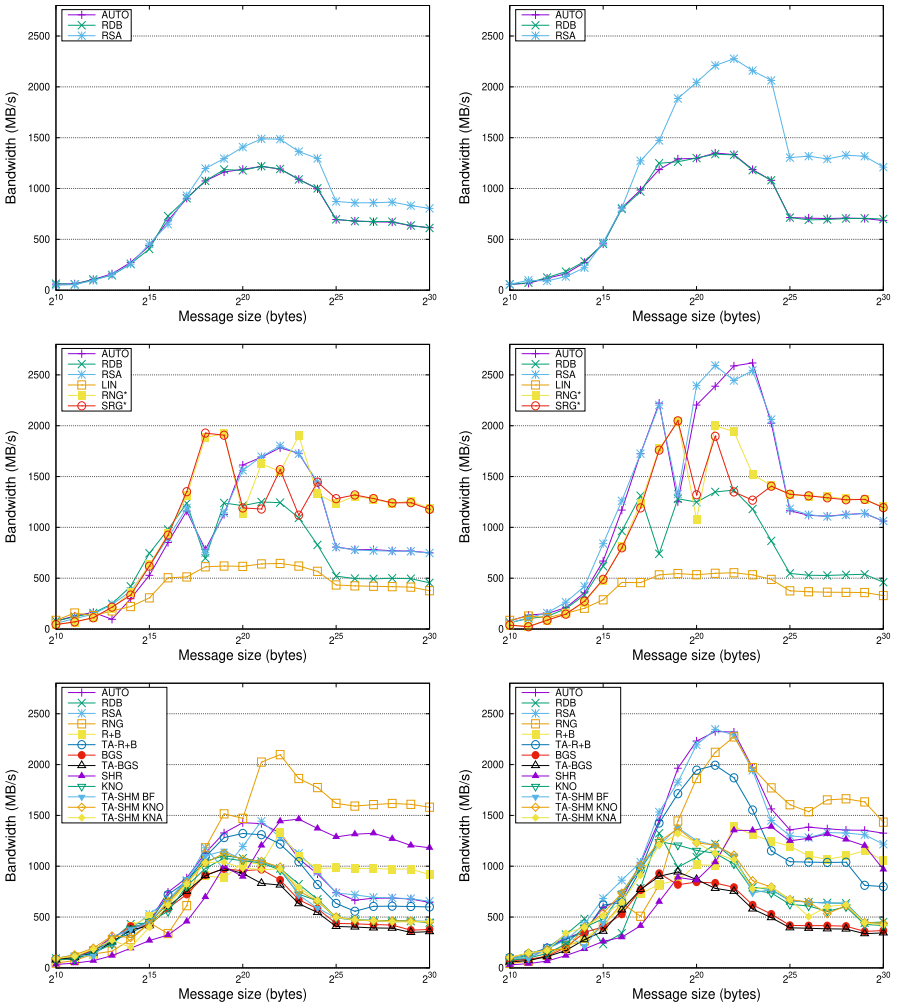

Fig. 5 Evaluation of the ARD algorithms in MPICH (top), OpenMPI (middle) and IntelMPI (bottom) using 7 and 8 MPI-ranks (one per node) of the ALTEC cluster (left and right, respectively)

4.3 Global comparison

In this final part, we illustrate the performance differences comparing the AUTO and the BEST algorithm, for the three target MPI instances. For this experiment, BEST corresponds to the best option for each message size and number of processes.

Figure 6 displays the results of this global evaluation, offering two main conclusions:

– There are large gaps between the performance of BEST and AUTO for MPICH when $p = 7, 8$, as well as IntelMPI and OpenMPI when $p = 7$. The differences are significantly narrower for OpenMPI and IntelMPI when $p = 8$. This exposes an

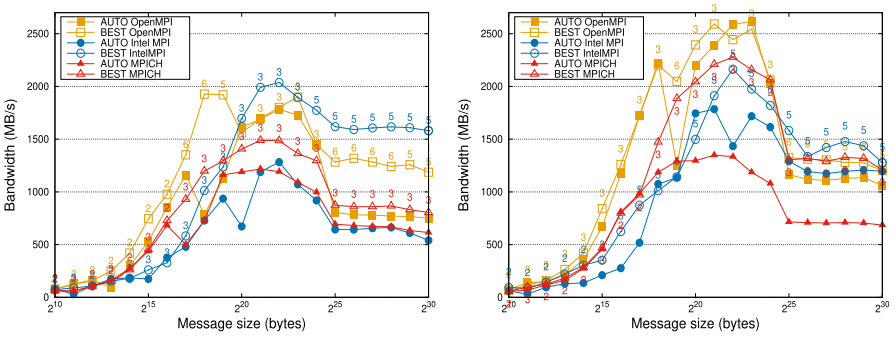

Fig. 6 Comparison of the AUTO and BEST ARD algorithms in MPICH, OpenMPI, and IntelMPI using 7 and 8 MPI-ranks (one per node) of the ALTEC cluster (left and right, respectively). The labels in the BEST lines indicate the id of the ARD algorithm that offers the best performance for that particular library and message size

appealing opportunity to improve the performance of `MPI_Allreduce` for IntelMPI and OpenMPI when utilizing 7 nodes, and for MPICH when using 8 nodes.

– For $p = 8$, MPICH-AUTO is optimal for small messages (up to 2^{17} bytes) but fails for other cases. IntelMPI-RNG delivers a high MiB/s rate for $p = 7$, and for the larger message sizes with $p = 8$. However, this algorithm is not selected as AUTO. OpenMPI-RNG/-SRG performs best for message sizes up to 2^{19}; from 2^{25} bytes when $p = 7$; and from 2^{25} bytes with $p = 8$. Again, AUTO adopts the RSA algorithm, being suboptimal on these message sizes window.

5 Impact of ARD in distributed DNN training

In this section, we analyze the gains that can be attained via a careful selection of the ARD algorithms on the distributed data-parallel training of CNNs. To this end, we employ Google's TF framework 2.3.0 for deep learning, running on top of Horovod 0.19, which provides a workload distribution tool that scales TF to run on a multi-node cluster. In addition, we consider 4 different CNN models —namely AlexNet, ResNet50, ResNet110, and VGG11— and two well-known datasets: ImageNet and Cifar10. The following configurations are selected for the evaluation:

– AlexNet+ImageNet. This is a classic CNN model characterized by a reduced number of convolutions followed by three dense layers.
– ResNet50+ImageNet. This benchmark from MLPerf [11] is composed of a series of residual blocks combining convolutions, batch normalization layers, and ReLU functions.
– ResNet110+Cifar10. This combination explores the behavior of a large DNN model with a "small" dataset (compared with ImageNet).
– VGG11+ImageNet. This model is computationally very intensive, which paves the road to analyzing a type of scenario where communications play a less significant role.

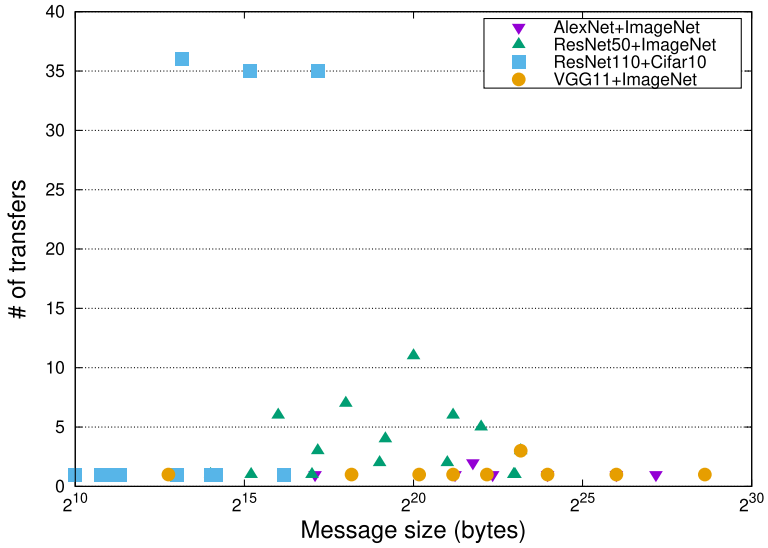

Fig. 7 Message sizes and number of transfers of each size for four of the selected configurations

Figure 7 represents the communication requirements for four configurations in terms of message sizes and number number of messages of each size. These data demonstrate the need to steer our experimental evaluation to cover a range of message sizes that expands from 1 KiB to 1 GiB.

5.1 Hiding the communication cost with Horovod

Horovod relies on MPI for the data exchanges and therefore utilizes the underlying ARD algorithms to realize the reduce+broadcast exchanges that are necessary during the distributed data-parallel training of CNN models. In addition, Horovod overlaps communication and arithmetic via an auxiliary communication thread as follows: Consider Fig. 8, which illustrates the data dependencies appearing during a single forward pass (FP) and backward pass (GC for Gradient computation and WU for Weight update) for a DNN consisting of L layers. In Horovod, the use of the communication thread overlaps the global reduction AR^l, for a certain layer l, with the computations corresponding to GC^{l-1}, GC^{l-2},…, GC^1. Furthermore, Horovod decides whether to aggregate the data corresponding to several consecutive layers into a single ARD communication operation of a larger dimension.

5.2 Evaluation in ALTEC

As an initial experiment, Fig. 9 displays the performance, in terms of images per second, attained by AUTO and BEST in the three target libraries. This simple test exposes the distinct behaviour of the different DNN model+dataset cases and, at the same time, clearly indicates that there is not a single optimal choice. The results also demonstrate

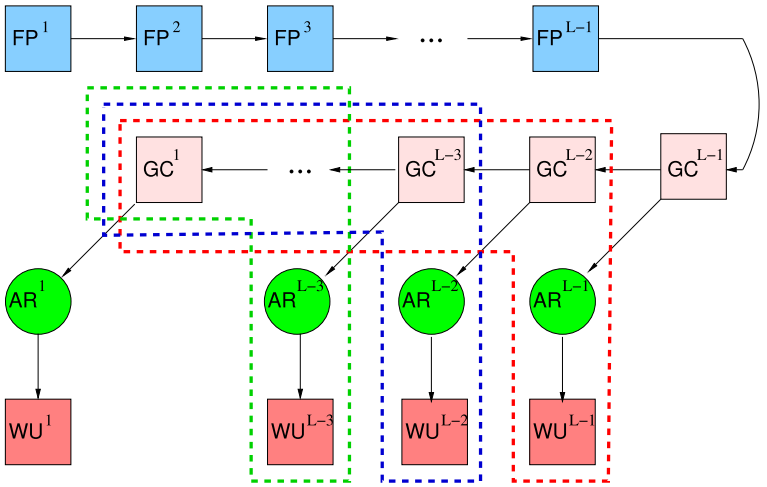

Fig. 8 Data dependencies in the training. The colored boxes correspond to the computational stages: FP, GC and WU; the circles denote ARD exchanges AR; and the arrows indicate dependencies. The colored dashed lines mark operations which can be overlapped

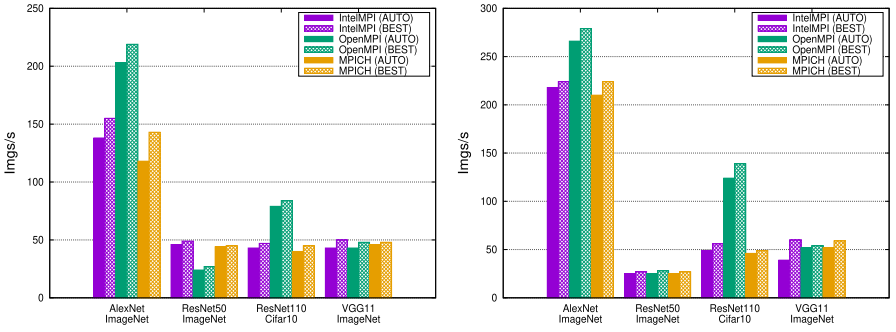

Fig. 9 Images per second of the CNN using 8 MPI-ranks (one per node) of the ALTEC cluster with $b = 16$ and 32 (left and right, respectively)

the need to optimize for each configuration: Due to the reduced model (in terms of model parameters and layers), AlexNet+ImageNet presents a communication-bound scenario. VGG11+ImageNet is at the opposite extreme, corresponding to a compute-bound case. ResNet50+ImageNet and ResNet110+Cifar10 lie in between, becoming compute-bound as the batch size b is increased.

Figure 10 reports the benefits of enforcing a specific selection of BEST instead of AUTO from within the `MPI_Allreduce` primitive in MPICH, OpenMPI, and IntelMPI, for the training of the selected DNN models and datasets via TF+Horovod, using 8 nodes of ALTEC cluster with a batch size b comprising 16, 32, and 64 images. For a complete comparison, the BEST algorithm is compared with the AUTO for each individual library (e.g., OpenMPI-BEST against OpenMPI-AUTO) in the left-hand column of plots, and taking the IntelMPI-AUTO algorithm as the baseline in the right-hand column of plots.

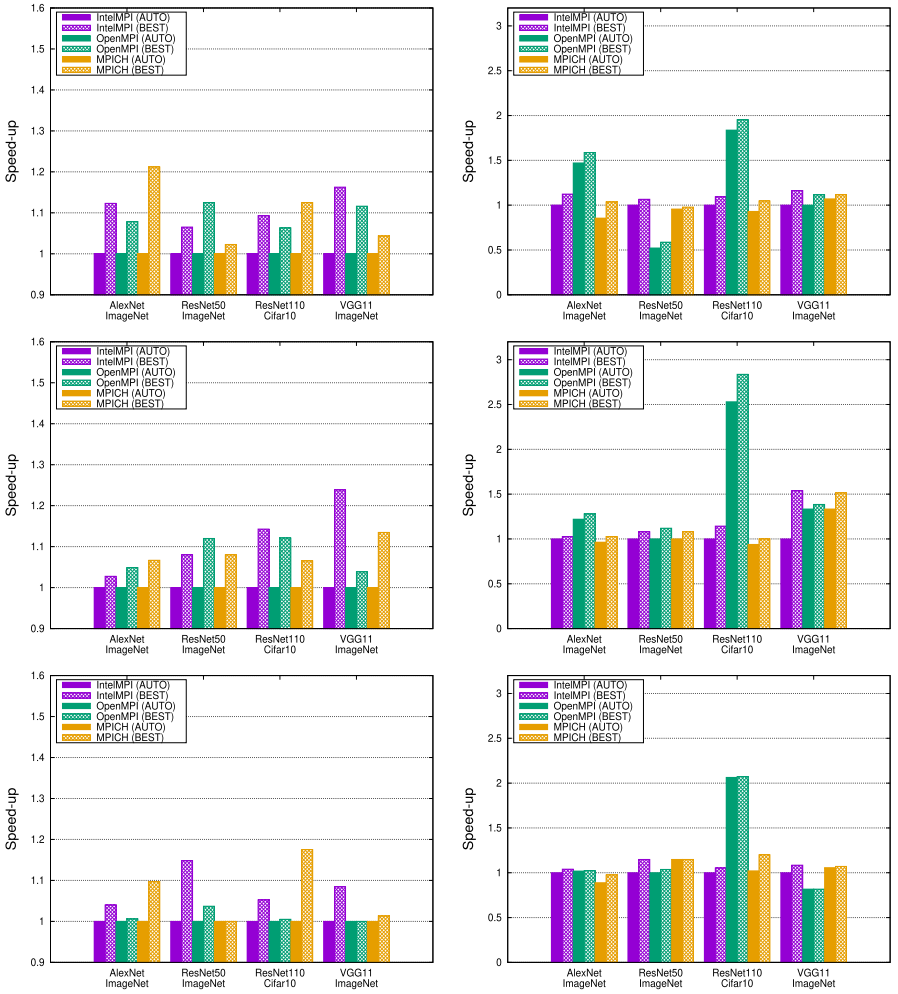

Fig. 10 Performance of TF+Horovod using 8 MPI-ranks (one per node) of the ALTEC cluster with $b = 16$, 32, and 64 (from top to bottom), with the results normalized with respect to either the AUTO algorithm of the corresponding library (left) or the AUTO algorithm in IntelMPI (right)

The first observation from this experiment is the need to carefully selecting the best option from within all the ARD possibilities (including both algorithm and library) for a concrete scenario before executing the CNN training: the benefits of a careful choice of the optimal algorithm within a particular library may render a performance boost of up to 40% in IntelMPI and MPICH, and up to 11% with OpenMPI. In general, the gains narrow as the training becomes more compute-bound (in particular, when increasing the batch size) and the impact of the communication decreases. Note, however, that there are limits to the dimension of the batch; see Sect. 5.3. Overall, OpenMPI outperforms its counterparts for almost all possible scenarios. As an exception to this general

Table 3 Performance comparison in terms of images per second of 8 CPUs against 8 GPUs using OpenMPI 4.1 and NCCL

Hardware	AlexNet ImageNet	*ResNet*50 *ImageNet*	ResNet110 Cifar10	VGG11 ImageNet
CPU (AUTO)	330	27	196	58
CPU (BEST)	337	31	197	64
GPU (AUTO)	10,156	3054	20,786	3657
GPU (NCCL)	26,928	3092	21,059	3846

rule, IntelMPI is the best choice for a few cases, in particular, ResNet50+ImageNet and VGG11+ImageNet with $b = 16, 64$, and VGG11+ImageNet with $b = 32$.

5.3 Limitations of the analysis

The insights gained from the experimental evaluation in this section can vary depending on two factors with an impact on the computation-communication balance, namely the batch size and the interconnection network. We next discuss these in detail:

- Although the arithmetic cost grows linearly with the batch size b, for a distributed data-parallel scheme the communication cost is largely independent of b (but grows with p). Therefore, for a cluster with a fixed number of nodes, the practical contribution of the communication overhead to the total training cost can be reduced by increasing the batch size. Unfortunately, there is a fundamental limit to the largest batch size that can be used in neural network training and, already for modest batch sizes, there appears a significant decline in the convergence rate of the training process. This issue can be tacked to a certain point, via the integration of very sophisticated, case-specific algorithmic techniques; see, e.g., [19].
- The Top500 list[11] from November 2020 comprises 25 systems connected via Infiniband HDR and 61 with Infiniband EDR. These two interconnection networks from Mellanox respectively present link speeds of 50 Gbps, and 25 Gbps per lane. If we compare this performance evolution (2× per interconnection generation) and the year gap between them (4 years), it is clear that it is much slower than the evolution in terms of computational power of CPUs/GPUs. Therefore, if the growth of both elements follows the same trend, the optimization (via the selection of the best algorithm or via the implementation of new communication patterns) is crucial for future applications.

6 Discussion

In this paper, we highlight the importance of the selection of MPI communication library as well as, the appropriated algorithm implementation of the `MPI_Allreduce` communication collective in order to accelerate distributed CNN training. We focus on

[11] https://www.top500.org.

general-purpose clusters (CPU only) because they still represent an important niche of computing resources which can be dedicated to the distributed CNN training. Although the experiments are done in a relative "small" cluster, the results expose how the appropriate choice of the software stack benefits performances. These analyses should be performed by all hardware-software combinations with the aim of achieving the best performances in all cases.

Although this study does not include clusters accelerated with GPUs, it is convenient to comment on the significant performance difference between CPUs and GPUs. Table 3 compares the throughput (in terms of images per second) when using OpenMPI and NVIDIA NCCL. The CPU rows show the results when using the AUTO and the BEST algorithm with a batch size of 64 images. The rows labeled as GPU (AUTO) and (NCCL) employ either the CUDA-aware OpenMPI 4.1 library or the NCCL communication library developed by NVIDIA, in both cases using a batch size of 256.

The adoption of GPU accelerators roughly increases the performance by a factor that ranges between 30× and 106×, though increases come with a considerable acquisition cost and power consumption footprint.

7 Conclusions and future work

We have conducted a complete experimental analysis of all the realizations of the ARD collective communication primitive in three popular MPI libraries: MPICH, OpenMPI, and IntelMPI using a cluster equipped with the state-of-the-art processor and network technologies. This study yields a number of relevant insights:

- There is a significant gap between the theoretical cost models for the ARD algorithms and their practical implementation in current MPI libraries. We have highlighted a few aspects that negatively affect the accuracy of the theoretical models.
- For some combinations of message size/number of nodes, the three MPI libraries make a poor selection of the best ARD algorithm, offering ample space for optimization.
- In general, all three libraries automatically select the RSA algorithm for ARD (the best algorithm, in theory), instead of the ring-based algorithms which the experiments show is a better option in a relevant variety of cases.
- When the number of processes participating in the communication is not an integer power-of-two, the overall communication throughput drops significantly, except for the ring-based solutions, which are independent of this feature.
- As the arithmetic capacity of the cluster nodes raises, the interconnection network poses a performance bottleneck. For distributed data-parallel training of CNNs, this can be tackled via either integrating a faster computer network or augmenting the batch size. However, it is surely beneficial to optimize the communication layer by choosing the best MPI library (in case this is possible) and/or select the appropriate ARD algorithm.
- Overall, when this last type of optimization is applied to distributed CNN training, the performance improvement is up to 20% compared with the automatic selection

done in the same MPI instance, and up to 280% if we compare the distinct MPI libraries.

Acknowledgements This research was partially sponsored by projects TIN2017-82972-R of *Ministerio de Ciencia, Innovación y Universidades* and Prometeo/2019/109 of the *Generalitat Valenciana*. Adrián Castelló was supported by the Juan de la Cierva-Formación project FJC2019-039222-I of the *Ministerio de Ciencia, Innovación y Universidades*. Manuel F. Dolz was also supported by the Plan GenT project CDEIGENT/2018/014 of the *Generalitat Valenciana*.

Author Contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by Adrián Castelló, Mar Catalán, and Manuel F. Dolz. The first draft of the manuscript was written by Adrián Castelló and was reviewed by Enrique S. Quintana-Ortí and Jose Duato. All authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding – Project TIN2017-82972-R of the Spanish *Ministerio de Ciencia, Innovación y Universidades*. – *Agencia Valenciana de la Innovación*.

Availability of data and material Not applicable

Declarations

Conflicts of interest Not applicable.

To be used for non-life science journals Not applicable.

Code availability The allreduce test codes can be found at https://github.com/adcastel/collectives.

References

1. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X(2016) Tensorflow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp 265–283, Savannah, GA, Nov. USENIX Association
2. Ben-Nun T, Hoefler T (2019) Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Computing Surveys, 52(4):65:1–65:43
3. Castelló A, Dolz MF, Quintana-Ortí ES, Duato J (2019) Theoretical scalability analysis of distributed deep convolutional neural networks. In 2nd High Performance Machine Learning Workshop – HPML'2019, pp 534–541
4. Castelló A, Catalán M, Dolz MF, Mestre JI, Quintana-Ortí ES, Duato J (2021) Evaluation of mpi allreduce for distributed training of convolutional neural networks. In 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pages 109–116
5. Chan E, Heimlich M, Purkayastha A, van de Geijn R (2007) Collective communication: Theory, practice, and experience. Concurr Comput: Practice Experience 19(13):1749–1783
6. Demmel J (2012) Communication avoiding algorithms. In Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC '12, pp 1942-2000, USA. IEEE Computer Society
7. Hasanov K, Lastovetsky A (2017) Hierarchical redesign of classic MPI reduction algorithms. J Supercomput 73(2):713–725
8. Hazelwood K, Bird S, Brooks D, Chintala S, Diril U, Dzhulgakov D, Fawzy M, Jia B, Jia Y, Kalro A, Law J, Lee K, Lu J, Noordhuis P, Smelyanskiy M, Xiong L, Wang X (2018) Applied machine learning

at Facebook: A datacenter infrastructure perspective. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp 620–629
9. Ivanov A, Dryden N, Ben-Nun T, Li S, Torsten H (2020) A case study on optimizing transformers, Data movement is all you need
10. Ketkar N (2017) Introduction to pytorch. In Deep learning with python, pages 195–208. Springer
11. attson P, Cheng C, Coleman C, Diamos G, Micikevicius P, Patterson D, Tang H, Wei GY, Bailis P, Bittorf V, Brooks D, Chen D, Dutta D, Gupta U, Hazelwood KM, Hock A, Huang X, Jia B, Kang D, Kanter D, Kumar N, Liao J, Ma G, Narayanan D, Oguntebi T, Pekhimenko G, Pentecost L, Reddi VJ, Robie T, John TS, Wu C-J, Xu L, Young C, Zaharia M (2019) Mlperf training benchmark. CoRR, arXiv:1910.01500
12. Nuriyev E, Lastovetsky A (2021) A new model-based approach to performance comparison of mpi collective algorithms. In Victor Malyshkin, editor, Parallel Computing Technologies, pp 11–25, Cham, Springer International Publishing
13. Sergeev A, Balso MD (2018) Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799
14. Shalf J (2019) HPC Interconnects at the End of Moore's Law. In 2019 Optical Fiber Communications Conference and Exhibition (OFC), pp 1–3
15. Snir M, Otto SW, Steven HL, Walker DW, Dongarra J (1996) The Complete Reference. The MIT Press, MPI
16. Thakur R, Rabenseifner R, Gropp W (2005) Optimization of collective communication operations in MPICH. Int J High Performance Comput Appl 19(1):49–66
17. Turchenko V, Bosilca G, Bouteiller A, Dongarra J (2013) Efficient parallelization of batch pattern training algorithm on many-core and cluster architectures. In 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), vol. 2, pp 692–698. IEEE
18. Turchenko V, Grandinetti L, Bosilca G, Dongarra JJ (2010) Improvement of parallelization efficiency of batch pattern bp training algorithm using open mpi. Procedia Computer Science 1(1):525–533
19. You Y, Gitman I, Ginsburg B (2017) Scaling SGD batch size to 32k for ImageNet training, arXiv:1708.03888