Cluster Computing manuscript No.
(will be inserted by the editor)

Accelerating Distributed Deep Neural Network Training with Pipelined MPI Allreduce

Adrián Castelló · Enrique S. Quintana-Ortí · José Duato

Received: date / Accepted: date

Abstract TensorFlow (TF) is usually combined with the Horovod (HVD) workload distribution package to obtain a parallel tool to train deep neural network on clusters of computers. HVD in turn utilizes a blocking Allreduce primitive to share information among processes, combined with a communication thread to overlap communication with computation.

In this work, we perform a thorough experimental analysis to expose 1) the importance of selecting the best algorithm in MPI libraries to realize the Allreduce operation; and 2) the performance acceleration that can be attained when replacing a blocking Allreduce with its non-blocking counterpart (while maintaining the blocking behaviour via the appropriate synchronization mechanism). Furthermore, 3) we explore the benefits of applying pipelining to the communication exchange, demonstrating that these improvements carry over to distributed training via TF+HVD. Finally, 4) we show that pipelining can also boost performance for applications that make heavy use of other collectives, such as Broadcast and Reduce-Scatter.

Keywords Message Passing Interface (MPI) · Collective communication primitives · Allreduce · Deep learning · Distributed training

Adrián Castelló, Enrique S. Quintana-Ortí, José Duato
Universitat Politècnica de València, Valencia, Spain
E-mail: {adcastel,quintana,jduato}@disca.upv.es

Declarations

0.1 To be used for non-life science journals

Not applicable

0.2 Funding

- Project TIN2017-82972-R of the Spanish *Ministerio de Ciencia, Innovación y Universidades.*
- *Agencia Valenciana de la Innovación.*
- Juan de la Cierva-Formación project FJC2019-039222-I of the *Ministerio de Ciencia, Innovación y Universidades*
- PRACE Preparatory Access project #2010PA5531.

0.3 Conflicts of interest

Not applicable

0.4 Availability of data and material

Not applicable

0.5 Code availability

The pipelined codes may be found at `https://github.com/adcastel/collectives`

0.6 Authors' contributions

All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by Adrián Castelló and Enrique S. Quintana-Ortí. 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.

1 Introduction

The outburst of deep learning (DL) technologies in the past few years has been accelerated by the development of efficient frameworks for distributed training of deep neural networks (DNNs) on clusters. Most of these frameworks exploit *data parallelism* (DP), by partitioning (and distributing) the workload among the cluster nodes/processes across the batch dimension (i.e., the inputs or samples) [5]. In this scenario, at each iteration of training, all processes collaborate to perform a reduction of the local weights in order to produce a global update of the parameters that define the DNN model [5, 8]. This synchronous version of training thus ensures that, prior to the next training iteration (with a new batch of samples), all processes unite their state.

Data communication through the interconnection network is particularly crucial for the efficient synchronous DP training of convolutional neural networks (CNNs) on clusters of computers [8, 15]. Furthermore, data movements (across the memory hierachy as well as the nodes of a distributed platform) are a major source of energy consumption [26].

MPI (Message Passing Interface) [28] is the *de facto* standard for distributed high performance computing (HPC) applications. Therefore, it has been naturally adopted as the communication layer for distributed training frameworks such as Google's TensorFlow (TF) [1], TF+Horovod (HVD) [25], and PyTorch [24]. The MPI application programming interface (API) comprises a large variety of peer-to-peer and collective communication primitives. Among these, the DP scheme for distributed DNN training basically relies on the blocking `MPI_Allreduce` primitive, which internally reduces a collection of local values broadcasting the global result to all processes participating in the communication.

In a previous work [6], we *analyzed* the impact on performance of selecting the best algorithmic realization of `MPI_Allreduce`, depending on the message size and cluster configuration, when leveraging TF+HVD to train CNNs on a small cluster equipped with an Infiniband network. In this paper we pursue the *pipelined optimization* of this global blocking reduction via the split of the data exchange into a collection of smaller `MPI_Iallreduce` calls, which offers a better overlap of computation with communication yielding a better utilization of the network bandwidth. In particular, our present work makes the following contributions:

- We perform a complete performance evaluation of a blocking global reduction when replacing the conventional (blocking) `MPI_Allreduce` call by the alternative non-blocking `MPI_Iallreduce` primitive immediately followed by the corresponding blocking synchronization `MPI_Wait`.
- We propose two realizations of `MPI_Iallreduce` that offer significantly higher performance when applied to perform a blocking global reduction. These implementations operate by dividing the message (transparently to the user) into either a collection of messages of a specific smaller size or a fixed number of smaller messages, in both cases pipelining the transfers.
- We demonstrate that the performance acceleration from pipelining the global reduction carry over to the distributed training of representative CNNs using the TF+HVD framework, offering performance improvements that vary between 5 and 60%.
- We show that splitting the messages so as to pipeline the transfers benefits not only the global reduction primitive but also other collective operations, such as broadcast or reduce-scatter, which can improve performance for applications that make significant use of them.
- Finally, our complete experimental analyses employ three popular instances of MPI and explores the distinct algorithms for the `MPI_Iallreduce` primitive in those to detect the best operation depending on the number of nodes and message size. Furthermore, we conduct the experiments on a couple of recent Infiniband network technologies: QDR and EDR.

The rest of the paper is organized as follows. In Section 2 we provide a brief review of related work. In Section 3 we include a discussion of some popular

algorithms for implementing a global reduction together with their theoretical costs. In Section 4 we conduct an analysis of these algorithms, and in Section 5 we introduce our pipelined optimizations for the `MPI_Allreduce` collective communication primitive. In Section 6 we extend the pipelining technique to other popular collective communication primitives. Section 7, we apply these optimizations to the TF+HVD DL environment. Finally, Section 8 summarizes the conclusions of this work.

2 Related Work

2.1 MPI collective communication primitives

Since its initial appearance in the early nineties, MPI [10] has evolved to integrate new functionality in addition to many optimizations. One relevant example is the design of efficient algorithms for collective communication primitives (CCPs) in line with [9, 30]. In particular, the former work 1) formalizes the theoretical analysis of CCP, focusing on simple and effective solutions that generalize to multidimensional meshes and hypercubes; and 2) shows how the algorithms for a given CCP can be organized into parameterized families, which then expose the keys for performance. In the latter work, the authors propose new algorithms to improve the performance of CCPs, for clusters connected by switched networks, pursuing either the minimization of latency for short messages or the reduction of bandwidth use for long messages.

Other work aimed at improving the CCPs performance act at node level. In particular, in [21, 22], shared-memory is exploited to boost intra-node communication; and in [33] the reductions are performed by means of AVX-512 instructions.

There exist a few works that specifically evaluate and/or improve the MPI CCPs for DL, for example, taking into account the special characteristics of the messages that are exchanged in this type of applications [3, 4, 18, 23]. In addition, MPI-based software has been developed for distributed DNN training; for example, MVAPICH2-GDR[1] from Ohio State University or oneAPI[2] from Intel.

[1] `https://mvapich.cse.ohio-state.edu/`
[2] `https://github.com/oneapi-src`

2.2 Pipelining for MPI CCPs

MPI was originally conceived with a strong focus on the efficient exchange of small messages. In consequence, the adoption of MPI in DL forced developers to study how to reduce the overhead of MPI for large messages. A step in this direction consists in dividing the transfers into a collection of smaller messages, yielding a *pipelined* (or *segmented*) communication scheme. In [32], the authors performed a manual segmentation for `MPI_Reduce`, `MPI_Bcast`, and `MPI_Allreduce` when communicating data among a few graphics processing units (GPUs) in the same node. In [31], the authors presented a pipelined data transfer mechanism for processes running on the CPUs of a single node. In two recent works [16,17], the authors present a pipelining approach for two CCPs: `MPI_Allgather` and `MPI_Allgatherv` with message sizes up to 64 and 8 MB, respectively, in a cluster.

In [2], a variant of pipelining is obtained by slicing the network (by means of virtual LANs) in order to exploit the full network bandwidth for data broadcasting.

Compared with previous work, we address the application of pipelining to optimize a blocking Allreduce, for very large messages, of up to 1 GB, in the context of distributed DNN training on clusters of computer nodes. In addition, we demonstrate that the same technique renders appealing benefits for some collective primitives, such as `MPI_Bcast` and `MPI_Reduce_scatter`, but not for other cases, such as `MPI_Allgather`, due to their implementation in current MPI libraries.

3 MPI Algorithms for Allreduce

The term "reduction" is frequently used in DL and DNN frameworks to refer to the global update of the DNN model parameters (i.e., weights and biases) that is necessary at each iteration of the synchronous version of the training process. In practice, this reduction is performed using the conventional (blocking) `MPI_Allreduce` CCP which, depending on the MPI library, is realized via different algorithms [9, 30]. In this section we review some of the most common algorithmic realizations of `MPI_Iallreduce`, together with their theoretical cost. We note that the non-blocking primitive can be easily leveraged to mimic the behaviour of the blocking-counterpart, by simply adding a proper synchronization after it.

3.1 A family of algorithms

There exist a number of instances of the MPI library, with some prominent examples being Open-MPI,[3] MPICH,[4] MVAPICH,[5] and Intel MPI.[6] All these implementations adhere to the functionality and specification defined by the MPI API, while distinct realizations of the standard vary in the implementation of the primitives and, quite often, the performance they attain.

The MPI instances usually optimize the CCPs via the implementation of a variety of algorithms (or communication schemes), in principle selecting the most appropriate option at execution time depending, for example, on the message size, number of processes, network topology, etc. For the particular case of (the non-blocking) `MPI_Iallreduce`, the following list briefly describes some of the most popular algorithms (see [9,12,30] for additional details):

1. RDB (Recursive doubling): Initially, the processes that are a "distance" 1 apart (i.e., with rank identifiers that differ only by 1) exchange (and reduce) their data. Next, the processes that are a distance 2 apart do the same with the complete data they own after the first exchange. This is repeated for processes which are at distances 4, 8,... apart, till all processes have received (and reduced) all the data.
2. RSA (Rabenseifner's algorithm): This algorithm performs a Reduce-Scatter exchange followed by an Allgather. For this, the algorithm uses a combination of recursive-vector halving and recursive-distance doubling for the Reduce-Scatter stage, and recursive-doubling for the subsequent Allgather.
3. RNG (Ring): The message size is divided into one segment per process and each process then sends its segment to the next process, where it is reduced with the local data. Once this step is complete, the process is repeated $p-1$ times. Finally, an Allgather is applied.
4. BIN (Binomial tree): The processes first perform a common (reverse) binary (or binomial) tree-based reduction to a specific process, to then broadcast the result back to all processes using a binary tree-based broadcast.

The above-mentioned algorithms are not exclusive of `MPI_Iallreduce`. In particular, its blocking counterpart is frequently implemented using the same algorithms (together with other options) [6]. Nonetheless, one major difference between the blocking and non-blocking variants is that, in `MPI_Iallreduce`, once all operations (communication and computation) are pushed to the scheduler, the control immediately returns to the user's application process. In comparison, for the `MPI_Allreduce` case, this control is only returned when all the operations are completed by the current process. A second difference lies in the specific peer-to-peer primitives that are utilized in each case.

3.2 Theoretical cost analysis

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	RNG	$2(p-1)$	$2n\ \frac{p-1}{p}$	$n\frac{p-1}{p}$
4	BIN	$2\log p$	$2n\ \log p$	$n\ \log p$

Table 1 Theoretical cost of common algorithms for `MPI_Iallreduce`.

Let us consider a collection of $n \cdot p$ data items, evenly distributed across a platform consisting of p nodes, with a single MPI process running on each node. Furthermore, consider the link latency is given by α (in seconds) and assume the link bandwidth, denoted by β (in data items per second), is independent of the message size. Assume also that the interconnection network supports simultaneous transfers between all pairs of nodes at full link bandwidth. Finally, consider that each node can perform γ arithmetic operations per second. Table 1 then displays the theoretical cost of the afore-described algorithms for `MPI_Iallreduce` separated into their latency, bandwidth, and arithmetic components. For simplicity, in the table we assume that p is a power of 2. Otherwise, all logarithmic costs need to be rounded up to the nearest integer. When the message is large, as it is generally the case in the DNN training, the transfer cost is dominated by the bandwidth

[3] `https://www.open-mpi.org`
[4] `https://www.mpich.org`
[5] `https://mvapich.cse.ohio-state.edu`
[6] `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.html`

term and, therefore, the expressions in the table indicate that the best options are RSA and RNG.

4 Performance Analysis of MPI_Iallreduce

MPI is nowadays widely adopted for distributed programming. However, our experience with several MPI libraries is that the algorithms which are automatically selected to execute the CCPs are often suboptimal for a considerable range of message sizes and number of nodes [6]. One main reason is that, while the theoretical cost models in Table 1 *a priori* identify the best algorithm(s), the experimental results may differ significantly for a number of causes. At this point, we recognize that most MPI instances were implemented with the target of optimizing the exchange of small messages on a reduced (usually, an integer power of two) number of nodes. This does not reflect the usual scenario in the case of DNN training.

In this section we conduct a complete evaluation of the algorithms for MPI_Iallreduce in three well-know, widely-used MPI libraries, and compare the available options with the automatic selection made by the library (hereafter referred to as AUTO), so as to asses if it selection is properly done.

4.1 Experimental setup

For our evaluation, we employ OpenMPI 4.1.0rc, MPICH 3.3.1, and Intel MPI 2020. The cluster platform for these experiments consists of 9 nodes equipped with 2 Intel Xeon Gold 5120 CPUs (14 cores each, for a total of 28 cores per node) at 2.20 GHz, 192 GB of DDR4 RAM, and an NVIDIA V100-PCIe GPU with 32 GB of HBM2 memory. The nodes are connected via an InfiniBand EDR network with a link latency of 0.5 μs and a link bandwidth of 100 Gbps.

In all experiments in this paper, the performance reported for a given algorithm is calculated as the *throughput rate* that is obtained by dividing the size of the message n (in bytes) with the time required to complete the global reduction. The tests are repeated a large number of times and the execution time corresponds to the average cost. *As we are dealing with non-blocking primitives, each call is paired with its corresponding synchronization* (MPI_Wait) *in order to ensure that the execution is complete prior to measuring the execution time.* About the selected metric, on the negative side, the actual number of transferred bytes during the reduction is considerably larger. Therefore, the transfer rate represented with this metric is in general quite below the actual network bandwidth. On the positive side, the metric is inversely proportional to the execution time for all algorithms and libraries and, therefore, sets the grounds for a fair comparison of these.

4.2 Analysis

In Figure 1, we display the throughput rates (measured in millions of bytes per second, or MB/s) attained by the algorithms for MPI_Iallreduce. This comparison includes the distinct algorithms in OpenMPI, MPICH, and Intel MPI, and is conducted using 8 and 9 MPI processes/nodes. (The latter configuration is chosen to evaluate the effect of executing the algorithms with a number of nodes that is not an integer power of two.)

The first factor exposed by the charts in Figure 1 is the distinct number of algorithms depending on the specific MPI library: While OpenMPI and MPICH integrate a reduced set of options (4 and 5, respectively), Intel MPI spans a much larger number of possibilities (a total of 9).

This study reveals that the best option is in general given by the RNG algorithm in OpenMPI. Although Intel MPI also implements that particular algorithm, it does not achieve a comparable throughput. In contrast, MPICH does not offer a realization of this algorithm. Analyzing the AUTO selections, all three MPI instances select RSA (one of the two best option for large messages, from the theoretical point of view; see Table 1). However, this choice corresponds to the actual optimal option only for MPICH. This experiment clearly shows that a correct selection of the algorithm can significantly improve performance. Taking into account the considerably higher performance attained with OpenMPI, for brevity in the remainder of the paper we exclude MPICH and Intel MPI from the discussion. The complete results using these two other libraries can be found in Appendix A.

5 Pipelined MPI_Iallreduce

Although the MPI_Iallreduce CCP is usually viewed as a "monolithic" operation, its realization comprises

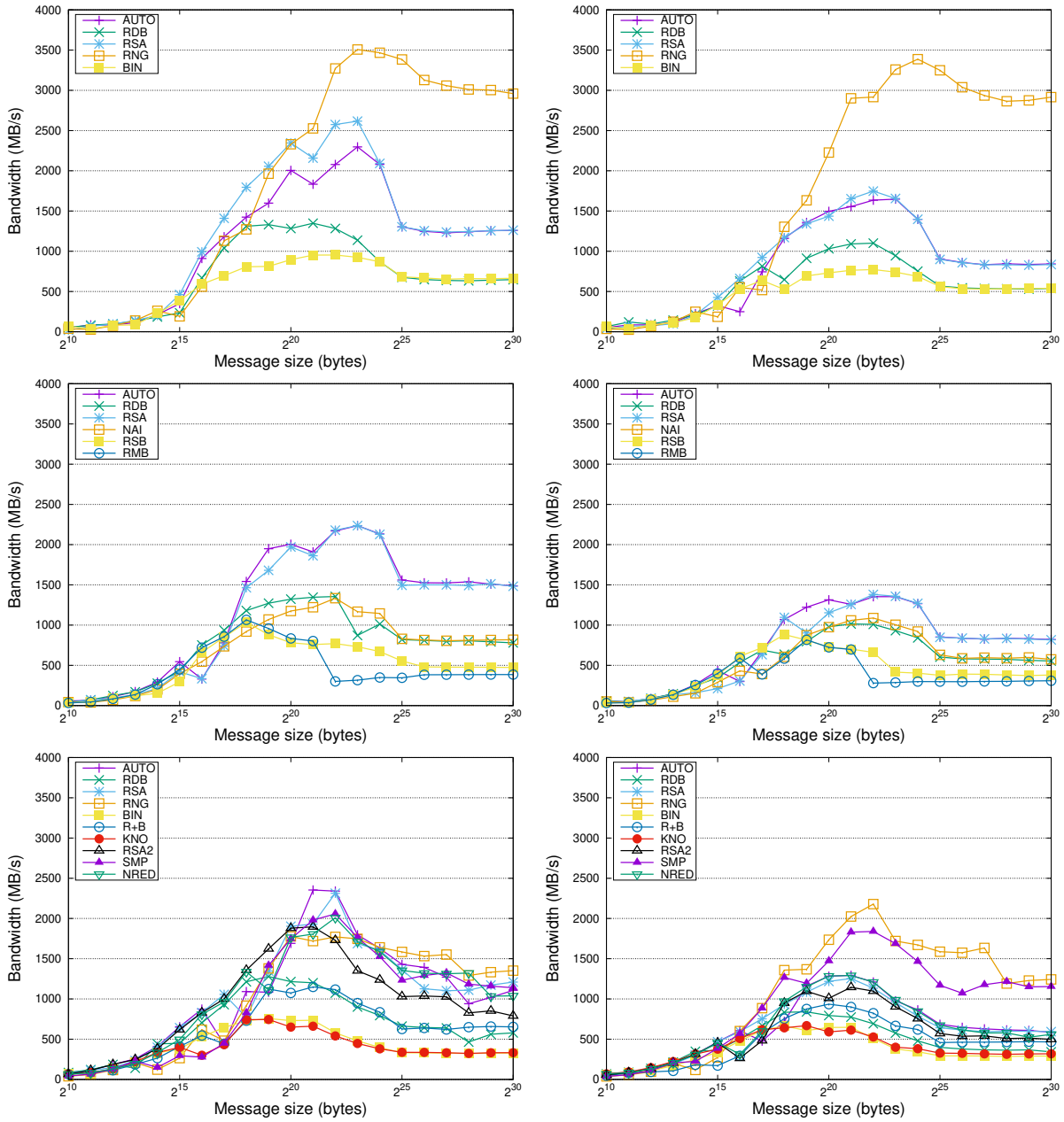

Figure 1 Performance of the algorithms in OpenMPI, MPICH and Intel MPI for `MPI_Iallreduce` (top, middle and bottom, respectively) using 8 and 9 nodes/processes (left and right, respectively).

a sequence of intertwined communication exchanges and arithmetic computations. Specifically, this CCP consists of a set of calls to the non-blocking peer-to-peer primitives in the MPI API for sending and receiving data, `MPI_Isend` and `MPI_Irecv` respectively, plus a few simple arithmetic computations. All these are passed to a communication runtime scheduler (and the associated thread) so that the application thread which invoked `MPI_Iallreduce` can continue with the execution of the user's code while the non-blocking data transfers are performed.

The possibility of overlapping communication and computation, in addition to a potential infra-utilization of the network bandwidth, offer appealing niches for performance improvement, especially for large messages. In this section, we explain how address-

```
int MPI_Piallreduce_fsize(TYPE * in, TYPE * out,
  size_t s, MPI_Comm  comm, int segsize,
  MPI_Request * request, MPI_Status * status)
{
  int h;
  int numseg = ( s <= segsize) ? 1 :  s/segsize;
  if (numseg > 1 &&  s % segsize != 0) numseg++;

  size_t sent = 0;
  for(h=0;h<numseg-1;h++)
  {
    MPI_Iallreduce( &in[h*segsize],
        &out[h*segsize], segsize, MPI_TYPE,
        MPI_OP, comm, &request[h] );
    sent+=segsize;
  }
  h=numseg-1;
  MPI_Iallreduce( &in[h*segsize],
    &out[h*segsize], s-sent, MPI_TYPE,
    MPI_OP, comm, &request[h] );
  return numsegs;
}
```

Listing 1 Pipelined global reduction with fixed message size.

```
int MPI_Piallreduce_fconc(TYPE * in, TYPE * out,
  size_t s, MPI_Comm  comm, int numseg,
  MPI_Request * request, MPI_Status * status)
{
  int h;
  if(s < numseg){numseg=1;}
  size_t segsize = s/numseg;

  size_t sent = 0;
  for(h=0;h<numseg-1;h++)
  {
    MPI_Iallreduce( &in[h*segsize],
        &out[h*segsize], segsize, MPI_TYPE,
        MPI_OP, comm, &request[h] );
    sent+=segsize;
  }
  h=numseg-1;
  MPI_Iallreduce( &in[h*segsize],
    &out[h*segsize], s-sent, MPI_TYPE,
    MPI_OP, comm, &request[h] );
  return numseg;
}
```

Listing 2 Pipelined global reduction with fixed degree of concurrency.

ing these weakness via the concurrent execution of multiple `MPI_Iallreduce`s improves performance.

5.1 Pipelining with fixed message size

In order to obtain a pipelined variant of the global reduction, we can divide (that is, partition or segment) the original exchange into several non-blocking calls, as shown in Listing 1. There, we split the message, consisting of `s` bytes, into several segments (or chunks) of `segsize` bytes each (except, possibly, for the last one), and perform $\texttt{numseg} = \lceil \texttt{s}/\texttt{segsize} \rceil$ consecutive invocations to the non-blocking `MPI_Iallreduce` CCP, that is one per segment, to initiate the exchange of the corresponding segment. Note that the routine returns the number of segments so that a conventional blocking behaviour can be achieved by simply invoking the synchronizing `MPI_Waitall`.

Figure 2 reports the effect of pipelining with a fixed message size, applied to the AUTO and RNG algorithms for both 8 and 9 nodes. For the AUTO option (RSA for OpenMPI), these results show a performance gain for the pipelined variants of about 250 MB/s for message sizes larger than 32 MB and 8 processes. From that size and beyond, dividing the message into 32 MB segments is one of the best options. For AUTO and 9 processes, the performance gain of the pipelined realizations is slightly higher: from 800 MB/s up to more than 1,200 MB/s. For the best algorithm (RNG for OpenMPI), the pipelined variants generally outperform AUTO by a factor of 2×. In addition, dividing the message into several segments allows to maintain the asymptotic exchange throughput at 4,000 MB/s, whereas the original primitive falls from a peak of 3,500 MB/s to 3,000 MB/s or less for large messages.

5.2 Pipelining with a fixed degree of concurrency

An alternative to obtain a pipelined realization of `MPI_Iallreduce` is to divide the message into a fixed number of smaller messages. For that purpose, we employ the code in Listing 2, where we perform the global reduction of a message, of `s` bytes, by means of `numseg` calls to the non-blocking `MPI_Iallreduce` CCP, one per segment of $\texttt{segsize} = \texttt{s}/\texttt{numseg}$ bytes (except for the last one, which has to take into account the possibility of the message size not being an integer multiple of the number of segments). Note that the number of primitives which are concurrently executed in this scheme is fixed. The routine also returns the number of segments so that a blocking behaviour to allow a straight-forward realization of a blocking behaviour via the invocation to `MPI_Waitall`.

Figure 3 illustrates the impact of pipelining with a fixed degree of concurrency. The two top charts in the figure confirm that the AUTO algorithm (for OpenMPI, RSA) only benefits from this type of pipelining for messages of size larger than 32 MB. In contrast, the pipelined RNG algorithm (see the bottom two charts) already improves the performance for messages of size larger than 2 MB. In addition, these charts expose that it is possible to accelerate

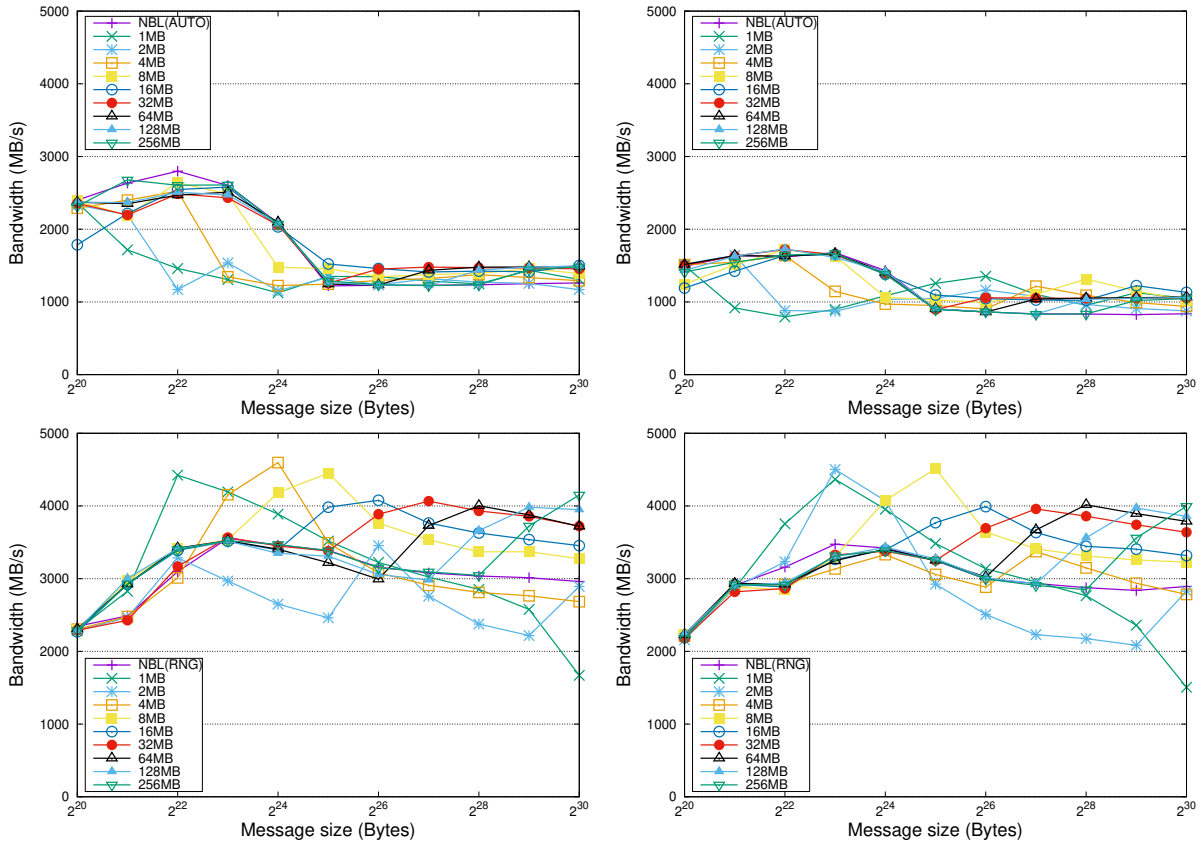

Figure 2 Performance of the AUTO and RNG algorithms in OpenMPI (top and bottom, respectively) using 8 and 9 nodes/processes (left and right, respectively). The NBL(AUTO) and NBL(RNG) labels correspond to the evaluation of the original `MPI_Iallreduce` primitive; the labels of type XMB indicate the pipelined variant `MPI_Piallreduce_fsize` with segment size `segsize`=X MB.

the throughput by up to 25% applying this type of pipelining.

Taking into account that the RNG algorithm outperforms AUTO by a factor of two, pipelining the RNG algorithm for large messages improves the communication performance of the automatic selection from 1,400 to 4,000 MB/s for 8 processes and from 800 to 4,000 MB/s for 9 processes.

Both pipelining techniques have demonstrated significant performance advantage with respect the conventional `MPI_Iallreduce`. However, when dealing with an application where the message may vary in size from one transfer to another (as it is the case of DNN training), it is more convenient to apply the pipelining with a fixed degree of concurrency. In this scenario, a solution based on a fixed-size segment may result in an elevate number of messages if the size is too small, constraining the performance as it is shown in Figure 2. In consequence, in the remainder of this paper we will consider only the variant that applies pipelining with a fixed degree of concurrency.

5.3 Identification of the source of gains

We next present a simple experiment that demonstrates the communication-computation overlap that takes place when splitting the reduction primitive into multiple smaller calls. For this purpose, we have modified OpenMPI to eliminate the arithmetic computations that occur inside the `MPI_Iallreduce` primitive by omitting the submission of the corresponding arithmetic tasks to the communication scheduler.

Figure 4 shows the throughput rate for the RNG algorithms with and without the arithmetic operations which are necessary for the reduction. The lines labeled with the NOOP suffix correspond to the modified OpenMPI routine without arithmetic operations. The lines with labels NBL and NBLx4

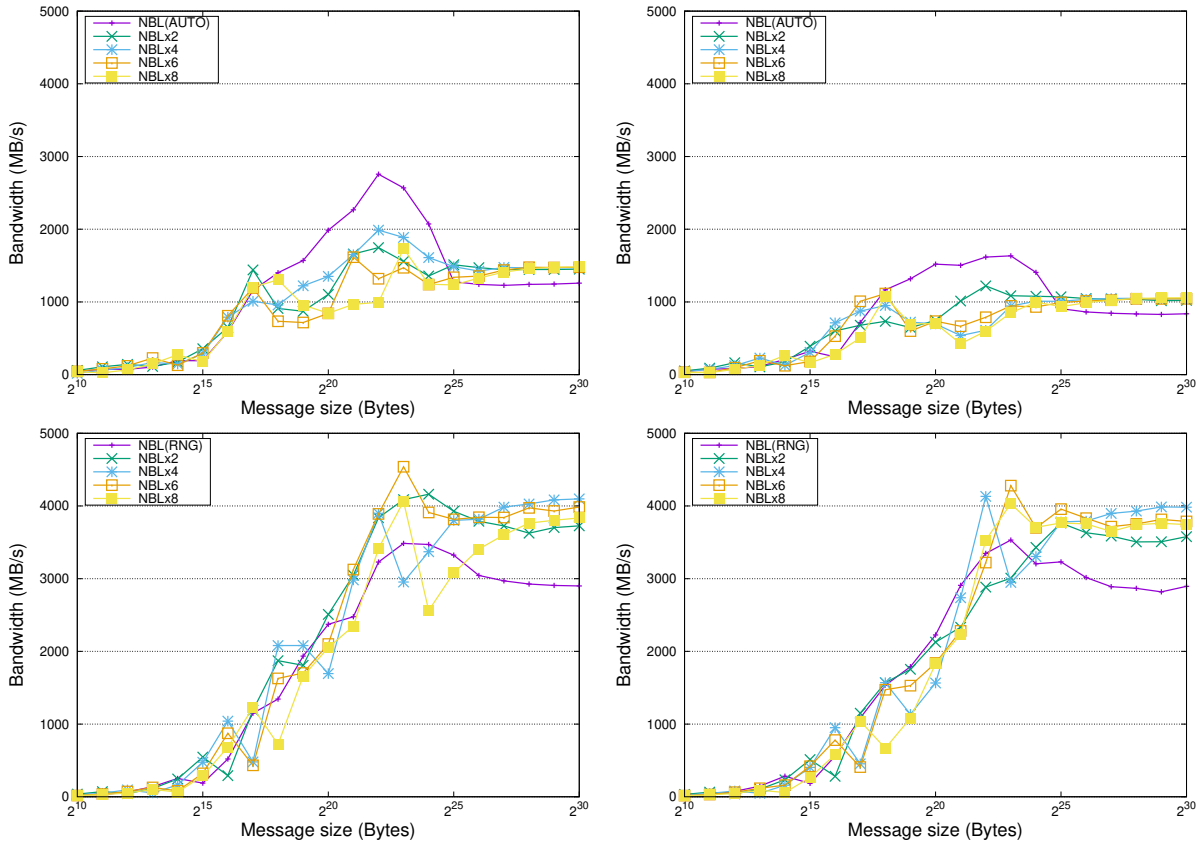

Figure 3 Performance of the AUTO and RNG algorithms in OpenMPI (top and bottom, respectively) using 8 and 9 nodes/processes (left and right, respectively). The NBL(AUTO) and NBL(RNG) labels correspond to the evaluation of the original `MPI_Iallreduce` primitive; the labels of type NBLxY indicate the pipelined variant `MPI_Piallreduce_fconc` with `numseg`=Y segments.

distinguish between the conventional and pipelined variants. The direct comparison between the lines with the NOOP suffix and the conventional variant (label "NBL(RNG)") exposes the contribution of the arithmetic to the cost of the reduction operation, which represents between 28 to 37% of the total execution time.

Now, by applying pipelining, the difference with respect to the NOOP versions is reduced, and the fraction of time with non-overlapped communication-computation diminishes to less than 14%: compare the lines with the NOOP suffix and the pipelined variant (label "NBLx4(RNG)". In summary, although the pipelining does not totally overlap computation with communication, it reduces the impact of arithmetic on the global cost of `MPI_Iallreduce` significantly.

6 Extension of Pipelining to other MPI Collectives

While our work was, so far, mainly focused in `MPI_Allreduce`, the MPI standard defines several other collective primitives that involve multiple processes inside a communicator. In this section, we evaluate the possibility of improving the performance of these other primitives by pipelining their data transfers.

6.1 `MPI_Allgather`

`MPI_Allgather` performs an all-to-all communication where each MPI process broadcasts its portion of the final result to the other processes. The logical communication pattern is thus similar to that present in `MPI_Allreduce`, as each MPI process contributes a piece of the final result, except in that

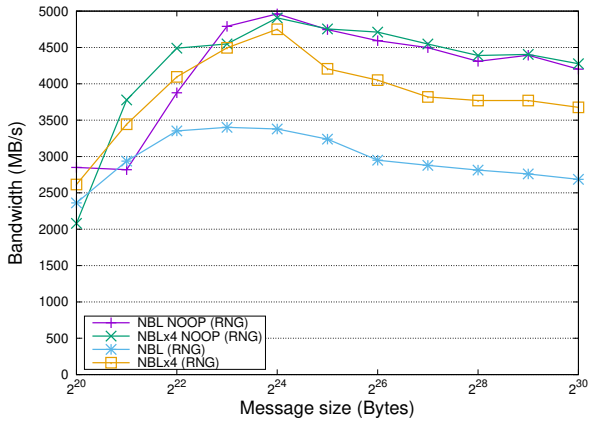

Figure 4 Performance of the RNG algorithm in Open-MPI using 8 nodes/processes. The NBL NOOP suffix identifies the modified variant of `MPI_Iallreduce` without arithmetic operations; the labels of type NBL and NBLx4 respectively indicate the conventional variant `MPI_Piallreduce_fconc` and the pipelined alternative with `numseg`=4 segments.

there is no reduction performed during the data recollection.

Figure 5 shows the result of applying pipelining to the `MPI_Iallgather` collective. All variants there rely on a LIN-based algorithm, which turns out to be the best option for this primitive. Unfortunately, for this particular primitive applying, a pipelined communication scheme does not render any performance improvement. Even worse, in the case of 8 MPI processes, the message pipelining reduces the performance for a certain range of message sizes. The reason for this lies in the particular "linear" implementation of this primitive in OpenMPI, which implies that segmenting the data transfers increments the number of messages but does not render a higher degree of parallelism among data transfers.

6.2 `MPI_Bcast`

`MPI_Bcast` is a popular collective primitive where a message is broadcast from one single process to all other processes participating in the communicator.

Figure 6 highlights the effect on performance of exploiting pipelining in the asychronous variant of this collective. In this case, the AUTO algorithm follows a CHAIN implementation which is already segmented internally. Nonetheless, adding an extra level of segmentation increases the performance from the initial 2,400 and 1,900 MB/s to 3,500 and 3,250 MB/s for 7 and 8 processes, respectively. In contrast with the `MPI_Iallgather` case, for the broadcast the communication has a single original (root) process. In consequence, splitting the message into smaller chunks augments the number of concurrent data transfers.

6.3 `MPI_Reduce_scatter`

`MPI_Reduce_scatter`, as its name indicates, combines a first stage that reduces the contents of a data array into a single process, to then split (and scatter) the result of the reduction among all processes in the second stage.

Figure 7 displays the effect of applying a pipelined scheme to the asynchronous version of this collective. The results expose two different scenarios: For messages of size smaller than 16 MB (2^{24} bytes), the best option uses a single segment (i.e., no pipelining). In contrast, for messages larger than that threshold, segmenting the communication improves the total performance by about 100 MB/s. As was the case for `MPI_Allreduce`, the performance gain comes from the reduction stage, but the effect is mostly blurred by the scatter stage if the data size is small. Note that OpenMPI implements only one algorithm for this primivite and, therefore, the AUTO option simply relies on that.

6.4 Other collectives

The MPI standard also defines other relevant collective primitives: `MPI_Alltoall`, `MPI_Gather`, `MPI_Scatter`, and `MPI_Reduce`. In this work though, we do not consider them because of their similarities with the four analyzed primitives. Specifically, `MPI_Alltoall` and `MPI_Gather` are highly related with `MPI_Allgather` in the communication pattern; `MPI_Scatter` is a "reversed" realization of `MPI_Gather`, and therefore it is also linked with `MPI_Allgather`; finally, `MPI_Reduce` is a "chopped" version of `MPI_Allreduce`, where there is a single reduction point and, given that the performance gains for the latter mostly come reduction part, we may expect a similar behaviour.

Some of the collective communication primitives reviewed in this section are heavily leveraged in distributed training of DNNs. Specifically, `MPI_Allgather`, `MPI_Bcast`, `MPI_Reduce_scatter` and `MPI_Allreduce` are employed for distributed DL frameworks that exploit model parallelism instead of the conventional data-parallel approach [7].

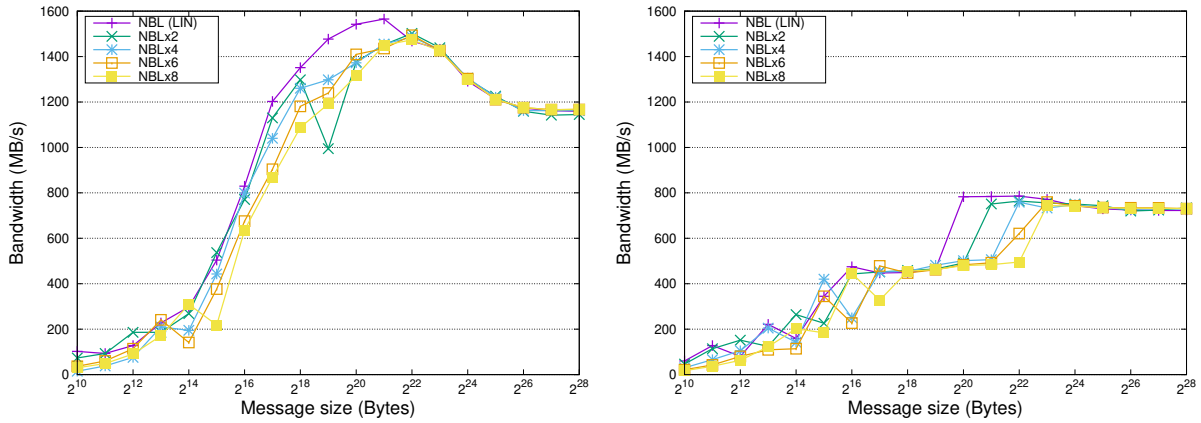

Figure 5 Performance of the LIN algorithm for `MPI_Iallgather` in OpenMPI using 7 and 8 nodes/processes (left and right, respectively). The labels of type NBLxY indicate the pipelined variant `MPI_Piallgather_fconc` with `numseg`=Y segments.

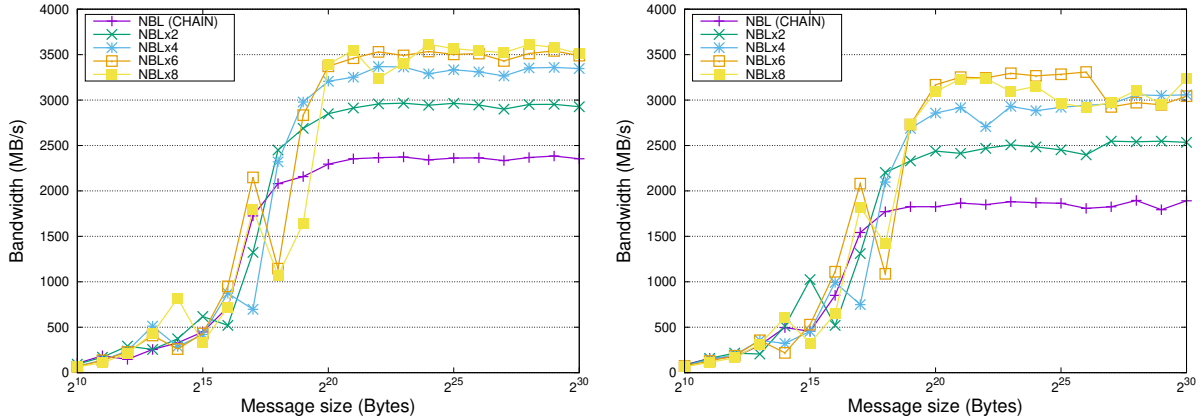

Figure 6 Performance of the CHAIN algorithm for `MPI_Ibcast` in OpenMPI using 7 and 8 nodes/processes (left and right, respectively). The labels of type NBLxY indicate the pipelined variant `MPI_Pibcast_fconc` with `numseg`=Y segments.

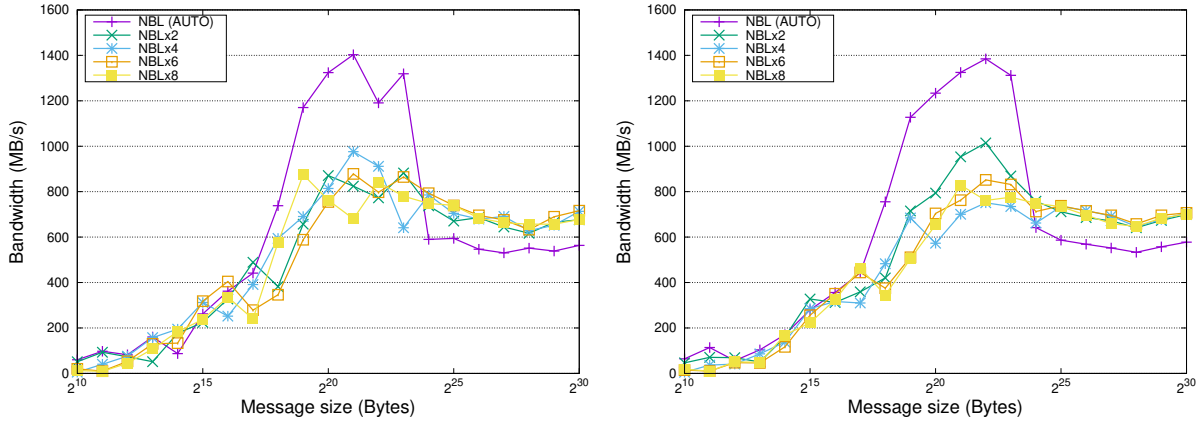

Figure 7 Performance of the AUTO algorithm for `MPI_Ireduce_scatter` in OpenMPI using 7 and 8 nodes/processes (left and right, respectively). The labels of type NBLxY indicate the pipelined variant `MPI_Pireduce_scatter_fconc` with `numseg`=Y segments.

7 Acceleration of Distributed DNN Training

In this section we assess the benefits of the pipelined communication schemes proposed in our work when applied to accelerate distributed training of DNNs on a cluster of computer nodes, possibly enhanced with GPUs.

7.1 Experimental training setup

In order to illustrate the advantage of the new CCPs, we employ TF 2.1.0 with Horovod (HVD) 0.20.3 as the target framework for distributed CNN training. In the experiments, HVD is compiled and linked with OpenMPI 4.1.0 and CUDA support. For the evaluation involving GPUs, we also consider NVIDIA NCCL 2.7.8 as an alternative communication layer. For the computations, we use Intel MKL 2020 in the case of CPUs and CUDA 10.2 and cuDNN 8.0 for GPUs. All the results are obtained using the TF benchmark suite [14] executed with Python3.7.

To close this short review of the training setup, we consider a testbed consisting of four CNN models: AlexNet [20], ResNet50 [13], VGG11 [27], and ResNet110; and two datasets: Cifar10 [19] and ImageNet [11].

Table 2 characterizes each DNN model-dataset combination, indicating the total number of layers, the amount of model parameters, and the floating point operations (flops) per training iteration. (The latter parameters is actually a function of the batch size b.) For simplicity, we only report the number of flops for the forward pass; the total amount flops for a complete forward-backward iteration of training is roughly obtained by multiplying this number by 3.

Model	# of layers	Params.	flops $\times b$
AlexNet	22	62,378,344	2,270,512,192
ResNet50	176	25,636,712	8,178,368,512
VGG11	30	132,863,336	15,218,180,096
ResNet110	393	1,747,898	506,832,128

Table 2 Characterization of the DNN models+datasets. AlexNet, ResNet50 and VGG11 employ ImageNet and ResNet110 is for Cifar10

Figure 8 reports the number of messages and their size for the model-dataset pairs. We can clearly observe there that each scenario exhibits quite different transfer requirements. At this point it is necessary to point out that, in practice, frameworks such as HVD group several consecutive "small" data transfers into a single one to reduce latency. Therefore, the number of messages per forward-backward pass may be smaller than the amount of convolutional and fully-connected layers for a DNN model. (Concretely, HVD groups small data transfers into 64 MB data transfers.) Grouping messages into larger data transfers favors pipelining, as it increases performance for messages of that size.

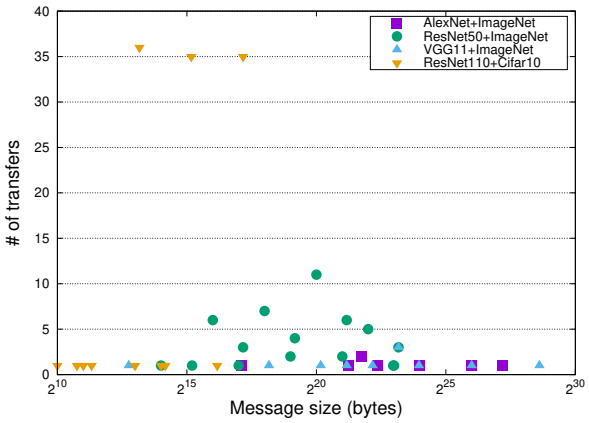

Figure 8 Number of transfers clustered by message size for the DNN models+datasets.

7.2 Brief discussion of TF+HVD

At each iteration, HVD exploits DP by distributing a global batch size of $b \cdot p$ samples among the p processes/nodes; TF then performs the local computations corresponding to the forward pass and backward propagation [29]; and HVD orchestrates the global reduction necessary for the update of the DNN model (that is, weight and biases) prior to the next iteration.

The original version of HVD relies on the blocking `MPI_Allreduce` primitive to perform the global reduction. Furthermore, HVD off-loads the execution of these primitives to a communication thread, which allows to overlap the operations that are necessary for the computational parts of the backward propagation (performed by the application processes/-threads and TF) with the communication for the global reduction [8, 25].

In order to test our (non-blocking) pipelined CCP, we developed a modified version of HVD that replaces the blocking collective with an adaptation of

the code shown in Listing 2, followed by a wait synchronization. The numerical behavior and, therefore, the convergence rate of our TF+modified HVD version is equivalent to that of the original TF+HVD framework. In consequence, we can directly compare the execution time of the two solutions for a specific number of training epochs (10 in the experiments). To avoid variations, the framework is initialized with the same random seeds, yielding the same starting DNN model for the training process and, consequently, the same training convergence (except for the effect of rounding errors).

7.3 Cluster of multicore processors

Figure 9 illustrates the acceleration (or speed-up) attained when replacing the conventional invocation to (the blocking) `MPI_Allreduce` in HVD, with the blocking optimal algorithm for OpenMPI (that is, `RNG`), the non-blocking counterpart, and the pipelined variant (with the fixed concurrency degree set to 4). The local batch size b for this particular experiment is set to either 32 or 64. We note that, increasing the batch size generally reduces the contribution of the communication to the training cost per iteration, but may increase the total time due to a decay in the convergence rate.

The results in Figure 9 offer three main conclusions: 1) The non-blocking `MPI_Iallreduce` outperforms its blocking counterpart, likely due to implementation decisions of the internal peer-to-peer calls; 2) the `RNG` algorithm offers considerably higher performance than the `AUTO` selection for this scenario; and 3) applying pipelining significantly improves the training performance of the framework.

In general, the performance gains vary depending on the contribution of communication to the total training cost, which in turn strongly depends on the CNN model (and, of course, the cluster hardware configuration). For example, applying the pipelining techniques to a communication-bound testbed, such as ResNet110 with Cifar10, improves the performance by a factor that is between 50% and 60%. Conversely, when the computation dominates the total execution time (as, for instance, is the case for ResNet50 and VGG11 when trained with ImageNet), the performance gain is more modest, in the range between 5% and 22%.

7.4 Cluster of multicore processors with GPU accelerators

In the last experiment, we investigate the impact of pipelining on the training throughput when the target cluster is equipped with graphics accelerators. For this particular case, in the comparison we also consider an alternative where the communication layer provided by NVIDIA's proprietary NCCL library instead of MPI. NCCL provides highly efficient primitives for Infiniband, with direct access to the GPU memory and, therefore, is very difficult to outperform by a "general-purpose" library such as MPI. Nonetheless, we intend to verify whether it is possible to reduce the difference between MPI and NCCL by pipelining the communications.

Figure 10 reports the training throughput rate (meassured in terms of number of images processed per second, or images/s) and speed-up with respect the conventional blocking implementation in HVD based on `MPI_Allreduce`. In this study, we only consider the `RNG` algorithm since this has been identified as the best MPI-based option for this scenario. As it already occurred when the experiment did not exploit the GPUs, for those cases where the communication is the bottleneck (e.g., AlexNet and VGG11 with ImageNet), NCCL clearly outperforms any of the MPI configurations. Conversely, for compute-bound training scenarios (e.g., ResNet50 with ImageNet and ResNet110 with Cifar10 and a large batch size), the difference between NCCL and MPI is negligible. The performance improvement when pipelining is applied yields a speed-up of up to 60% in the best case. If we compare the pipelined `MPI_Iallreduce` version against NCCL, we observe that it performs close to the NVIDIA solution, with the difference narrowing as the batch size is increased. At this point we repeat that augmenting the batch size may affect the convergence and accuracy of the training process, which often asks for a very fine-grain, application-dependent, tuning of the learning rate that needs to be dynamically varied as the training process evolves. Therefore, this is a complex technique which requires special knowledge and care.

8 Conclusions

We have reported notable improvements on the performance of the blocking Allreduce via 1) an adequate selection of the underneath algorithm; 2) the

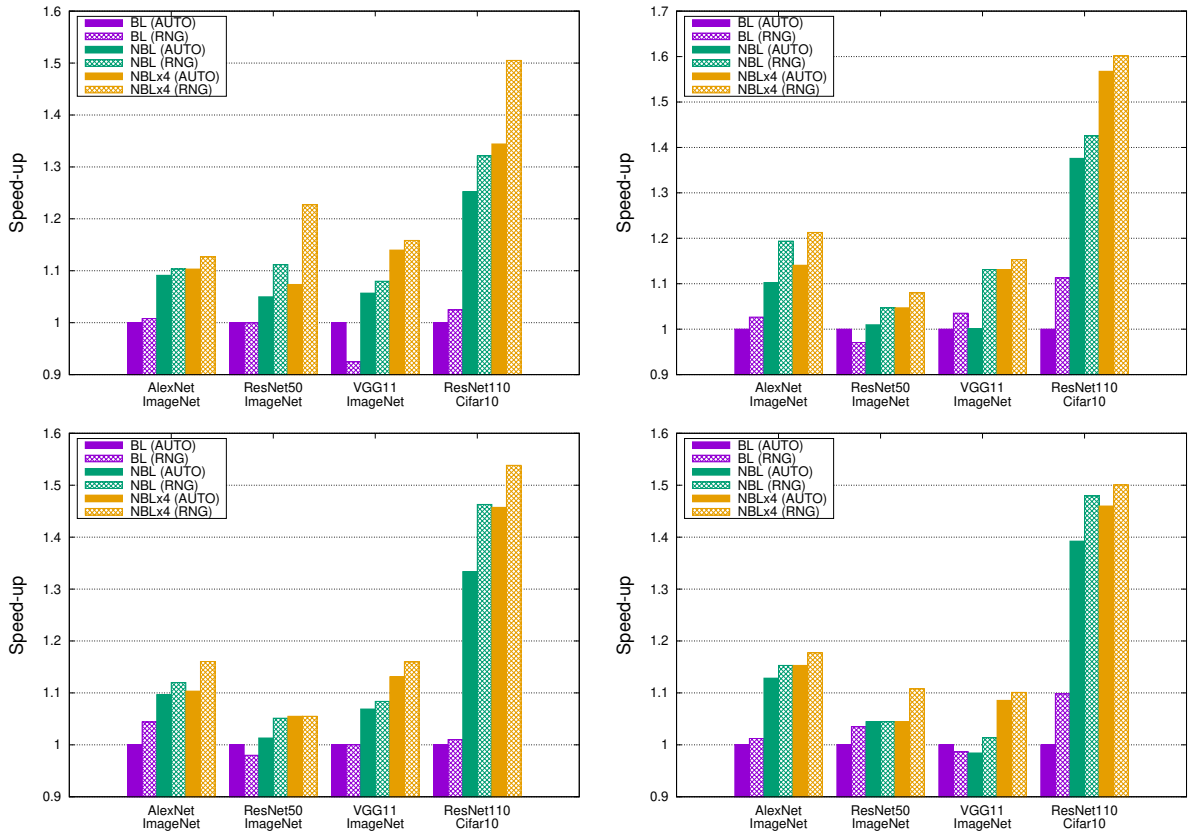

Figure 9 Speed-up of TF+HVD when the blocking AUTO algorithm for `MPI_Allreduce` (BL(AUTO)) is replaced with either the blocking RNG algorithm (BL(RNG)), the non-blocking algorithms (NBL(AUTO/RNG)), or the pipelined variants of the latter (NBLx4(AUTO/RNG)), using batch sizes of 32 and 64 (top and bottom, respectively), and 8 and 9 nodes/processes (left and right, respectively).

use of its non-blocking counterpart followed by a synchronization primitive, to preserve the global blocking behaviour; and 3) pipelining the original call into a collection of smaller collectives that accommodates a better overlap of computation with communication, yielding a better utilization of the network bandwidth. In general, the `RNG` algorithm tends to be the best `Allreduce` option for the target data-parallel TF+HVD framework but this is rarely selected as default algorithm and, in the case of MPICH, not even implemented. Furthermore, we have demonstrated that the performance advantages of the segmentation/pipelining techniques carry over to other relevant collective primitives.

These benefits have been demonstrated for MPI via a experimental analysis but, more importantly, also for a relevant framework for distributed training of DNNs: TF+HVD. For those training testbeds where the communication plays a key role, on clusters of multicore processors, the proposed optimizations of the MPI layer yield an acceleration of the training performance for TF+HDV of up to 22% for AlexNet and ResNet50; 15% for VGG11; and 50–60% for ResNet110 with respect to the configuration automatically selected by MPI. For platforms equipped with GPUs, NVIDIA's NCCL is still offers the best communication layer, outperforming any of the MPI-based solutions. However, the techniques that have been proposed in this work help to close the performance gap between NCCL and MPI by a significant margin. In general, we can expect that the segmentation/pipelining approach will benefit many distributed applications that makes heavy use of collective primitives.

Acknowledgments

This research was partially sponsored by project TIN-2017-82972-R of the Spanish *Ministerio de Cien-*

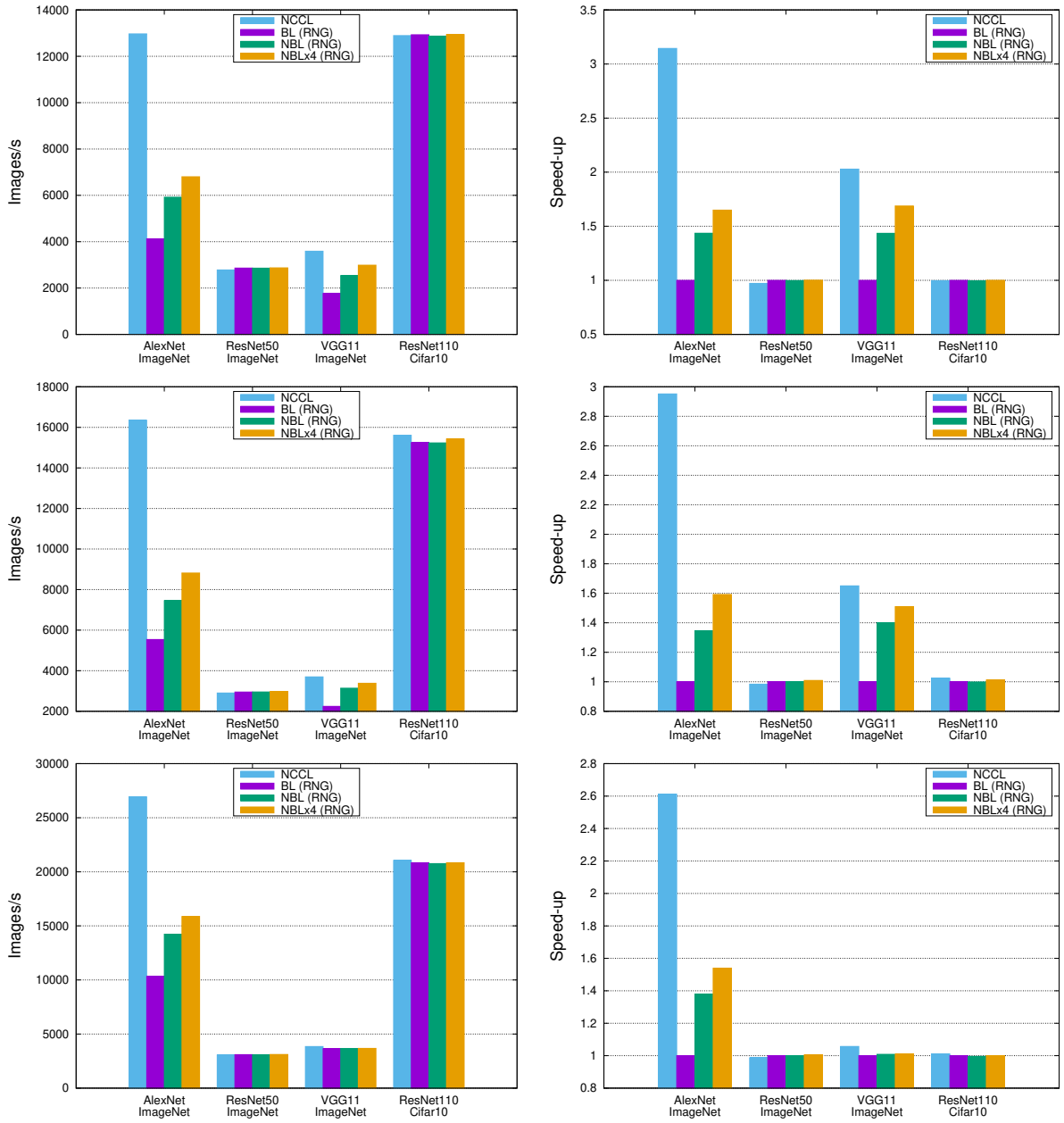

Figure 10 Training throughput (in images/s) and speed-up (left and right, respectively) of TF+HVD with 8 GPUs/nodes when the blocking AUTO algorithm for `MPI_Allreduce` (BL(AUTO)) is replaced with either the blocking RNG algorithm (BL(RNG)), the non-blocking algorithms (NBL(AUTO/RNG)), or the pipelined variants of the latter (NBLx4(AUTO/RNG)), using batch sizes of 96, 128 and 256 (top, middle and bottom, respectively). For reference, we also include the results when the communication layer is provided by NCCL.

cia, Innovación y Universidades and the *Agencia Valenciana de la Innovación*. Adrián Castelló was supported by the Juan de la Cierva-Formación project FJC2019-039222-I of the *Ministerio de Ciencia, Innovación y Universidades*. Part of this work was executed on the Marconi100 supercomputing facility from CINECA Interuniversity Consortium - HPC Department via the PRACE Preparatory Access project #2010PA5531.

References

1. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.
2. Izzat Alsmadi, Abdallah Khreishah, and Dianxiang Xu. Network slicing to improve multicasting in hpc clusters. *Cluster Computing*, 21(3):1493–1506, 2018.
3. Ammar Ahmad Awan, Jeroen Bedorf, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Scalable distributed DNN training using TensorFlow and CUDA-aware MPI: Characterization, designs, and performance evaluation, 2018. arXiv 1810.11112.
4. Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Optimized broadcast for deep learning workloads on dense-GPU InfiniBand clusters: MPI or NCCL? In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–9, 2018.
5. Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys*, 52(4):65:1–65:43, August 2019.
6. Adrián Castelló, Mar Catalán, Manuel F. Dolz, José I. Mestre, Enrique S. Quintana-Ortí, and José Duato. Evaluation of MPI Allreduce for distributed training of convolutional neural networks. In *29th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2021.
7. Adrián Castelló, Manuel F. Dolz, Enrique S. Quintana-Ortí, and José Duato. Analysis of model parallelism for distributed neural networks. In *Proceedings of the 26th European MPI Users' Group Meeting*, EuroMPI '19, New York, NY, USA, 2019. Association for Computing Machinery.
8. Adrián Castelló, Manuel F. Dolz, Enrique S. Quintana-Ortí, and José Duato. Theoretical scalability analysis of distributed deep convolutional neural networks. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 534–541, May 2019.
9. Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: Theory, practice, and experience. *Concurrency & Computation: Practice & Experience*, 19(13):1749–1783, Sept. 2007.
10. Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.
11. Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.
12. Khalid Hasanov and Alexey Lastovetsky. Hierarchical redesign of classic MPI reduction algorithms. *J. Supercomputing*, 73(2):713–725, Feb. 2017.
13. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
14. Google Inc. Tensorflow benchmarks. `https://github.com/tensorflow/benchmarks`.
15. Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers, 2020. arXiv 2007.00072.
16. Qiao Kang, Jesper Larsson Träff, Reda Al-Bahrani, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. Full-duplex inter-group all-to-all broadcast algorithms with optimal bandwidth. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–10, 2018.
17. Qiao Kang, Jesper Larsson Träff, Reda Al-Bahrani, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. Scalable algorithms for MPI intergroup Allgather and Allgatherv. *Parallel Computing*, 85:220–230, 2019.
18. Youngrang Kim, Hyeonseong Choi, Jaehwan Lee, Jik-Soo Kim, Hyunseung Jei, and Hongchan Roh. Towards an optimized distributed deep learning framework for a heterogeneous multi-gpu cluster. *Cluster Computing*, 23(3):2287–2300, 2020.
19. Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Department of Computer Sciences, University of Toronto, 2009.
20. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.
21. Mikhail Kurnosov and Elizaveta Tokmasheva. Shared memory based mpi broadcast algorithms for numa systems. In *Russian Supercomputing Days*, pages 473–485. Springer, 2020.
22. Shigang Li, Torsten Hoefler, Chungjin Hu, and Marc Snir. Improved mpi collectives for mpi processes in shared address spaces. *Cluster computing*, 17(4):1139–1155, 2014.
23. Truong Thao Nguyen, Mohamed Wahib, and Ryousei Takano. Hierarchical distributed-memory multi-leader MPI_Allreduce for deep learning workloads. In *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 216–222. IEEE, 2018.
24. Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.
25. Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow, 2018. arXiv 1802.05799.
26. John Shalf. HPC interconnects at the end of Moore's Law. In *2019 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3, 2019.

27. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014. arXiv 1409.1556.
28. Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.
29. Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017.
30. Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *Int. J. High Performance Computing & Appl.*, 19(1):49–66, Feb. 2005.
31. Joachim Worringen. Pipelining and overlapping for MPI collective operations. In *28th Annual IEEE International Conference on Local Computer Networks, 2003*, pages 548–557. IEEE, 2003.
32. Yiyang Zhao, Linnan Wang, Wei Wu, George Bosilca, Richard Vuduc, Jinmian Ye, Wenqi Tang, and Zenglin Xu. Efficient communications in training large scale neural networks. In *Proceedings of the on Thematic Workshops of ACM Multimedia 2017*, pages 110–116, 2017.
33. Dong Zhong, Qinglei Cao, George Bosilca, and Jack Dongarra. Using advanced vector extensions avx-512 for mpi reductions. In *27th European MPI Users' Group Meeting*, pages 1–10, 2020.

A Complementary Experiments

In this appendix, we assess the benefits of pipelining for a variety of experimental configurations: MPI library, network technology, network topology, and processor family. In all plots, the BL(AUTO), NBL(AUTO) and NBL(RNG) labels correspond to the evaluation of the conventional `MPI_Allreduce` collective and the original (non-blocking) `MPI_Iallreduce`; the labels of type NBLxY indicate the pipelined variant `MPI_Siallreduce_fconc` with `numseg`=Y segments.

A.1 MPI libraries

The target platform and libraries utilized in this first subsection are the same described in subsection 7.1.

Figure 11 shows that the performance gains attained with pipelining are considerable for both the AUTO and BEST algorithms in MPICH and Intel MPI. (Note that for MPICH, AUTO corresponds to the RSA algorithm as this is the best option. In consequence, we only provide one row of charts for that MPI library.)

Figure 12 shows that pipelining also improves performance for legacy library versions. In this case, the results were obtained with OpenMPI 4.0. Compared with the version evaluated earlier in the paper (4.1), OpenMPI 4.0 does not allow to select a concrete algorithm for the `MPI_Iallreduce` CCP.

A.2 Infiniband QDR

Figure 13 demonstrates that the efficiency of pipelining is also visible in case the nodes of the target platform are connected via an older Infiniband QDR switch. In this case, the experiments were executed on a cluster with 15 nodes, each equipped with two Intel Xeon E5645 Westmere processors (6 cores each) and 48 GB of DDR3 RAM memory. The MPI libraries selected for this experiment were OpenMPI 4.0.1, MPICH 3.3.2, Intel MPI 2019.

A.3 Network topology and processor family

Finally, Figure 14 reports the result of applying pipelining in the Marconi100 Supercomputer.[7] This last experiment was executed with up to 32 nodes, where each cluster node was equipped with two IBM POWER9 AC922 processors (3.1 GHz, 16 cores/processor) and 256 GB RAM each. (A node of Marconi also includes four NVIDIA Volta V100 GPUs, connected with NVLINK 2.0, and 16 GB.) The Infiniband network in this system is configured with a DragonFly topology (in all other experiments the topology was a Fat-Tree). The results confirm that a significant benefit can, again, be gained from pipelining for large messages.

[7] `https://www.hpc.cineca.it/hardware/marconi`

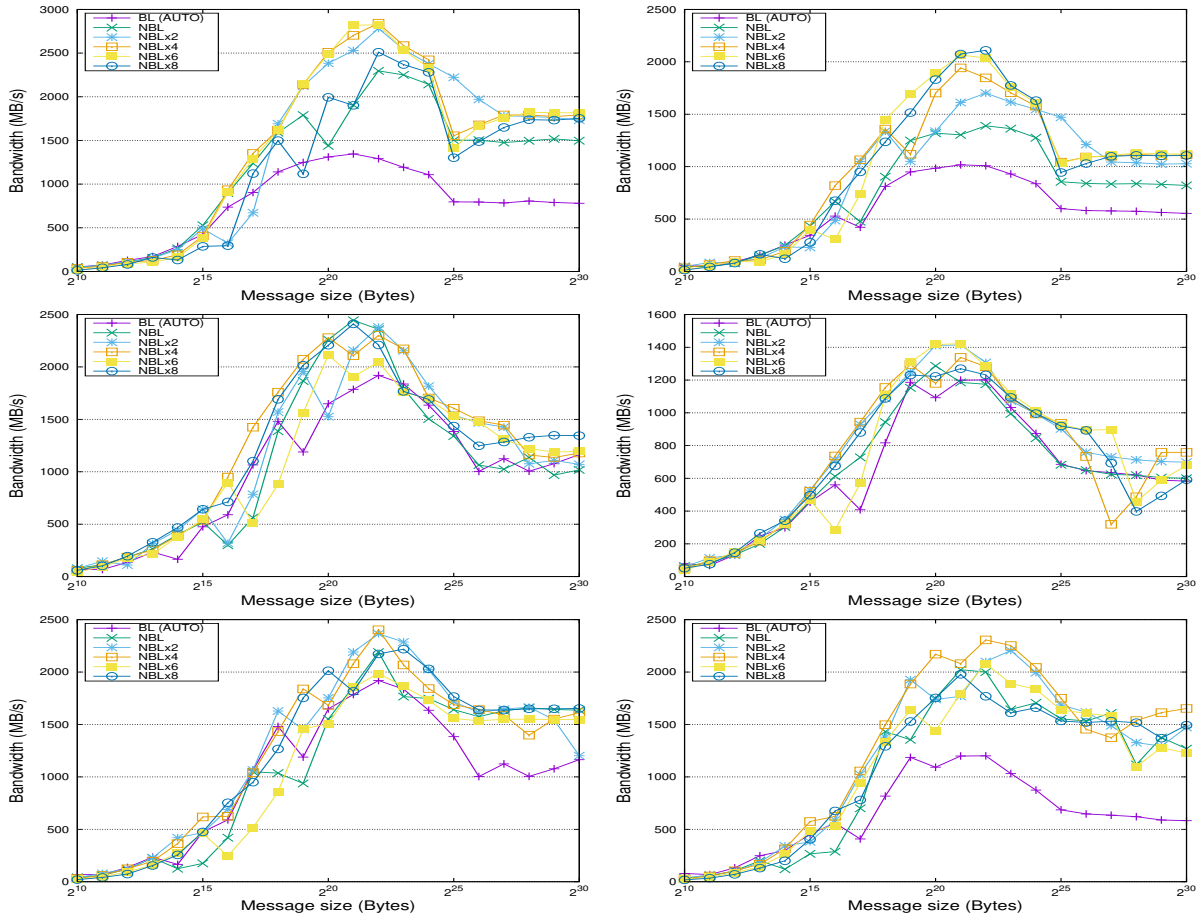

Figure 11 Performance of the AUTO algorithm in MPICH (top), and the AUTO and RNG algorithms in Intel MPI (middle and bottom, respectively) using 8 and 9 nodes/processes (left and right, respectively).

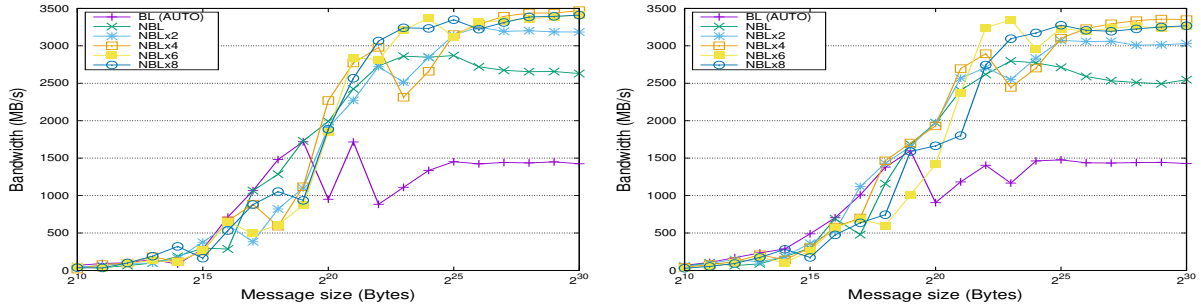

Figure 12 Performance of the AUTO algorithm in OpenMPI (4.0) using 8 and 9 nodes/processes (left and right, respectively).

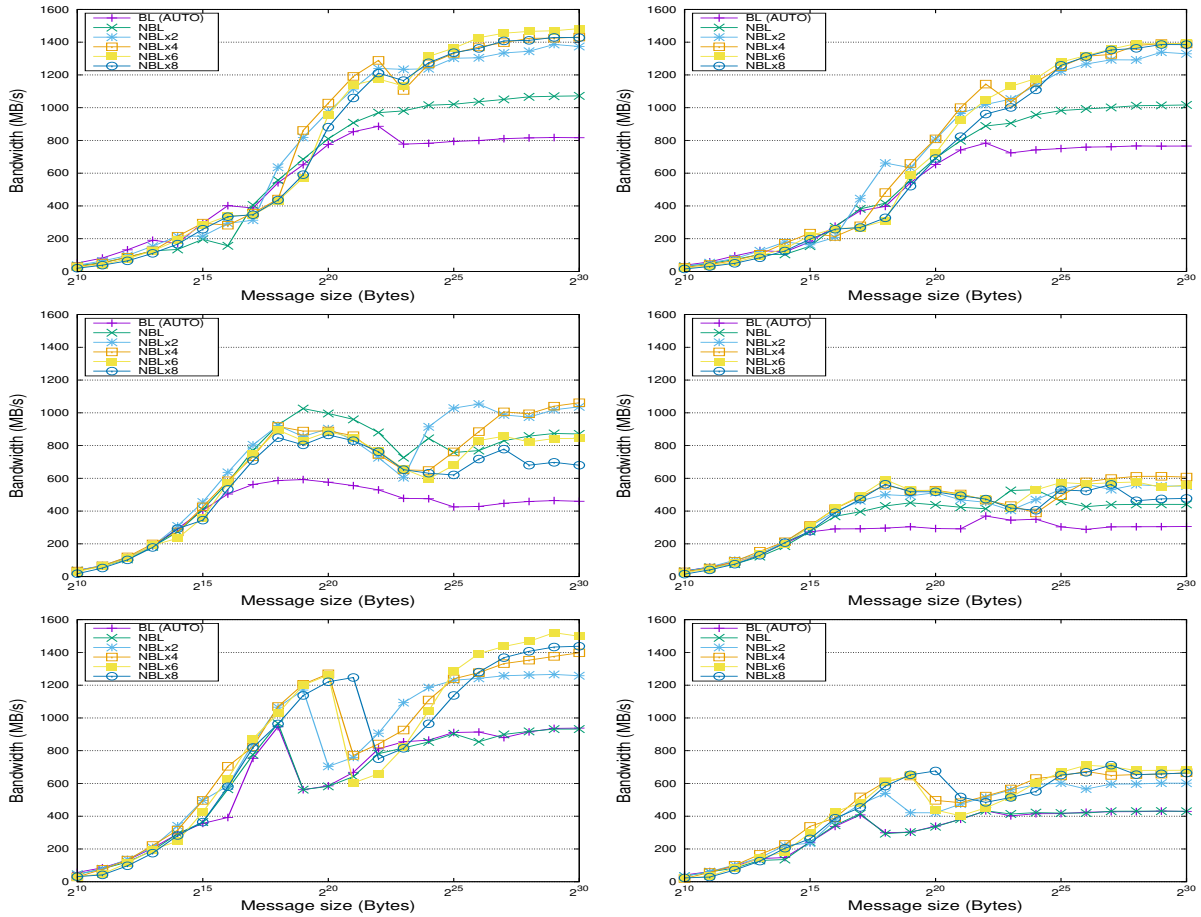

Figure 13 Performance of the AUTO algorithm in OpenMPI 4.0 (top), and the AUTO and RNG algorithms in MPICH and Intel MPI (middle and bottom, respectively) using 8 and 15 nodes/processes (left and right, respectively).

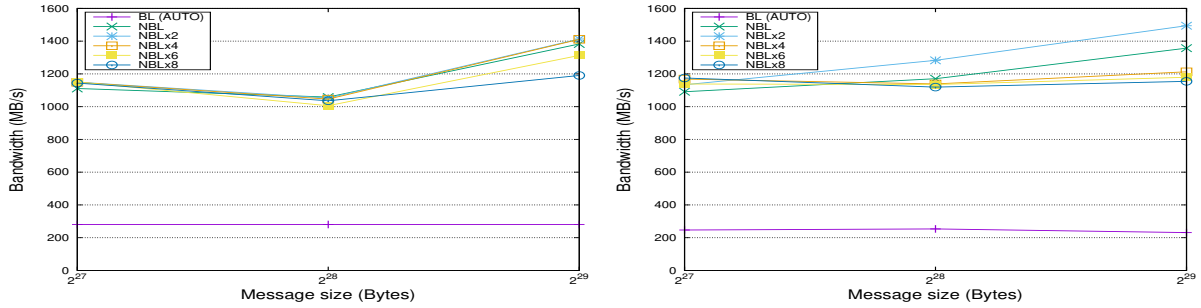

Figure 14 Performance of the AUTO algorithm in OpenMPI (4.0) for large messages using 30 and 32 processes (left and right, respectively).