Efficient and Eventually Consistent Collective Operations

Roman Iakymchuk*†, Amândio Faustino‡, Andrew Emerson§, João Barreto‡, Valeria Bartsch*,
Rodrigo Rodrigues‡, José C. Monteiro‡
* Fraunhofer ITWM, 67663 Kaiserslautern, Germany
Emails: {roman.iakymchuk,valeria.bartsch}@itwm.fraunhofer.de
†Sorbonne Université, 75252 Paris, France
‡INESC-ID & IST (ULisboa), 1000-029 Lisboa, Portugal
Email: {amandio.faustino,joao.barreto,rodrigo.miragaia.rodrigues}@tecnico.ulisboa.pt, jcm@inesc-id.pt
§CINECA, 40033 Casalecchio di Reno, Italy
Email: a.emerson@cineca.it

***Abstract*—Collective operations are common features of parallel programming models that are frequently used in High-Performance (HPC) and machine/ deep learning (ML/ DL) applications. In strong scaling scenarios, collective operations can negatively impact the overall application performance: with the increase in core count, the load per rank decreases, while the time spent in collective operations increases logarithmically.**

In this article, we propose a design of eventually consistent collectives suitable for ML/ DL computations by reducing communication in Broadcast and Reduce, as well as by exploring the Stale Synchronous Parallel (SSP) synchronization model for the Allreduce collective. Moreover, we also enrich the GASPI ecosystem with frequently used classic/ consistent collective operations – such as Allreduce for large messages and AlltoAll used in an HPC code. Our implementations show promising preliminary results with significant improvements, especially for Allreduce and AlltoAll, compared to the vendor-provided MPI alternatives.

***Keywords*—Collectives, Allreduce, AlltoAll, Stale Synchronous Parallel, GASPI.**

I. Introduction

The *Global Address Space Programming Interface (GASPI)* [1] programming model has proven to be an interesting alternative to the Message Passing Interface (MPI) model, in large part due to its open source implementation, *GPI-2*[1] and support for modern hardware architectures.

In this article, our focus is on collective operations, involving all the processes, and frequently used in distributed computations from HPC to distributed machine and deep learning (ML/ DL) computations. In particular, we make a preliminary exploration of the idea that collective operations can be designed and implemented with GASPI such that a globally consistent view is dropped. The concept comes from the distributed computing community where early work on mobile computing proposed the notion of eventual consistency [2], and has been adopted by the GRID computing community where globally distributed databases allow for reads to return potentially stale data. As the concept of *eventually consistent data* is ported to HPC collectives, it brings the potential to be used in many application domains, such as machine learning (ML). In this particular domain, most ML algorithms can be classified as iterative and convergent. These algorithms start with an initial guess of a model and then improve it across several iterations until converging to a solution.

In a typical distributed implementation of ML algorithms, several workers compute adjustments in parallel to the same model. Hence, workers are usually required to work on the same iteration, which leads to several synchronization points. However, due to the convergent nature of these algorithms, consistency can be dropped to a certain extent without jeopardizing the result by allowing workers to compute less accurate adjustments using stale data from different previous iterations. Thus, this reduces the required synchronization, and allows workers to go through iterations faster.

In this paper, we make initial contributions in exploring the space of *eventually consistent collectives* by investigating a Stale Synchronous Parallel (SSP) synchronization model [3], which allows the workers to compute iterations using bounded stale data. In the SSP model, the worker can receive updates while performing computation on stale data and, thus, seamlessly overlap communication and computation. We verify this implementation on an example of Matrix Factorization trained with Stochastic Gradient Descent. Additionally, we consider a possibility to drop a certain part of the data that is below a user-defined threshold in a collective. This allows the application to proceed with computations upon arrival of a part of data instead of the full amount.

Furthermore, we also provide classic/ consistent asynchronous variants of Allreduce suitable for large messages, especially in ML/ DL codes, as well as of AlltoAll, which is a time-consuming operation within the Quantum Espresso [4] application. With this effort, we aim to extend and enhance the current set of collectives in GPI-2 (only few available) to provide developers/ users with a *library of collectives* in order to facilitate their work.

This article is structured as follows: Section II introduces the asynchronous one-sided communication in GASPI. Section III reports on our initial work on eventually consistent collectives, while Section IV presents customized consistent collectives. Section V reports performance results. Finally, Section VI reviews related work, while Section VII draws conclusions.

[1]GPI-2's repository: https://github.com/cc-hpc-itwm/GPI-2

II. GASPI's asynchronous communication model

The GASPI standard promotes the use of *one-sided communication*, where one side, the initiator, has all the relevant information for performing the data movement. The benefit of this is decoupling the data movement from the synchronization between processes. It enables the processes to put or get data from remote memory, without engaging the corresponding remote process, or having a synchronization point for every communication request. However, some form of synchronization is still needed in order to allow the remote process to be notified upon the completion of an operation. In addition, GASPI provides what is known as *weak synchronization* primitives which update a *notification* on the remote side. The notification semantics is complemented with routines that wait for the update of a single or a set of notifications. Note that similar weak synchronization primitives might also appear in the upcoming MPI-4 standard [5]. GASPI allows for a thread-safe handling of notifications, providing an atomic function for resetting a local notification. The notification procedures are one-sided and only involve the local process.

A communication strategy for parallel applications can be as important as a numerical scheme. Usually, one tries to fit a communication strategy into a given numerical scheme by taking into account the algorithmic structure and, possibly, a spectrum of target platforms. Ultimately, one wants to hide communication (performed in the background) by overlapping computation and communication. This is the underlying idea that we promote with GASPI – write as early as possible and check for the arrival of the data, i.e. the notification, as late as possible (right before the data is to be used).

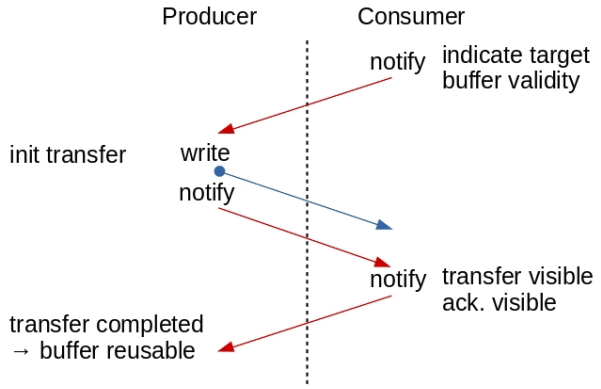

Fig. 1: Generic communication scheme in GASPI on the example of `gaspi_write`.

Figure 1 outlines the communication pattern in GASPI with the help of an example of `gaspi_write`, while Table I provides a more detailed description of the communication between producer and consumer (receiver). In this scenario, the consumer indicates the target buffer validity before the write begins. Then, the producer commences with the write of the data and issues a notification; these two can be combined into one by `gaspi_write_notify`. On the receiver side, GASPI guarantees that data is locally available whenever the corresponding notification becomes locally visible ('transfer visible'). This mechanism enables fine-grained (request based) asynchronous dataflow implementations as depicted in Figure 1. Finally, once the data arrives, the consumer acknowledges completion of the data transfer; hence, the producer can reuse the buffer.

On the producer side, the `gaspi_write_notify` – which performs the actual communication – is only allowed to be invoked if the first `gaspi_wait_some` – which waits for the initial notification from the consumer that indicates the target buffer validity – returns successfully. Therefore, the two calls can be combined together into a single operation, which would be invoked as soon as the source buffer is filled. Similarly, the `gaspi_wait_some` and the acknowledgment of completion on the consumer side can be combined into one operation which would be invoked before we want to work on the receive buffer. This reduces communication to two operations on both producer and consumer sides.

III. Eventually consistent collectives

We envision to design eventually consistent collectives by allowing the collective (the fastest processes) to not wait for the exchange of the most recent updates. Our motivation comes from the nature of many ML/ DL algorithms that can be classified as iterative and convergent. These algorithms start with an initial guess of a model and then improve it across several iterations until converging to a solution. Their convergent nature allows these algorithms to work with data that can be a bounded number of iterations out of date (referred to as the allowed slack). This tolerance to staleness is explored in the Stale Synchronous Parallel (SSP) [3] synchronization model. By allowing processes to compute iterations using bounded stale data, they can be receiving updates while performing useful computation on stale data, and thus, seamlessly overlap communication with computation. We incorporate this concept into a variant of the Allreduce collective, see Section III-A.

Another possibility is to provide pseudo eventually consistent collectives as follows. We can specify a termination criterion before the execution of collective, and then, in the call to collective, we can specify the predefined threshold as an input parameter. Hence, based on this threshold, every process or its parent can locally decide whether to stay silent or to engage, as well as how much to contribute. Consequently, the result (with respect to the threshold) can be obtained through the output status variable or at the receiver buffer after execution. This requires adding one or two additional parameters to collectives, namely threshold and status. We demonstrate this idea through examples of the Broadcast and Reduce collectives, leveraging on GASPI's API (see Section III-B).

A. Allreduce with SSP

In this section we propose an Allreduce collective following the SSP model [3]. We refer to this new collective as `allreduce_SSP`. It is presented in Algorithm 1.

We used a standard hypercube allreduce as a starting point and adapted it to the SSP model. Like most allreduce algorithms, the hypercube allreduce executes in several steps, where, in each step, communicating processes send and then wait to receive fresh contributions from every other process

TABLE I: Communication scheme in GASPI on example of `gaspi_write_notify` with the producer-consumer roles.

description/ usage	producer	consumer	description/ usage
setup phase	allocate resource exchange meta info	allocate resources exchange meta info	setup phase
check before communication	`gaspi_wait_some`	`gaspi_notify`	start as soon as the receive buffer can be overwritten
start as soon as the source buffer is filled	`gaspi_write_notify`	`gaspi_wait_some`	check before we want to work on the receive buffer or wait until its filled
check whether or wait until the source buffer can be overwritten	`gaspi_wait_some`	`gaspi_notify`	send as soon as the data arrived (acknowledgment)
shut down phase	release resources	release resources	shut down phase

before continuing to the next step. The distinguishing characteristic of the SSP model is that, instead of waiting for fresh contributions from all processes, each process only waits until local data contains contributions from all processes made at most `slack` iterations ago. For example, if the process is in iteration 5 and allows `slack` to be 1, the collective can return after using contributions from other processes that were computed in the current iteration, 5, but also from the previous iteration, 4.

Intuitively, to adapt a hypercube allreduce algorithm to support SSP, in `allreduce_SSP` processes remember the last contributions received at each step, and, provided that these contributions are not too stale, use them instead of waiting for fresh contributions. To accomplish that, we begin by identifying the contributions received at each step by a given process and reserve dedicated memory to receive the contributions from each of the steps; then, given we are using one-sided communication, processes send updated contributions by writing them in the dedicated memory of the process requiring those contributions, thus overwriting their previously sent contributions; finally, whenever a process wants to reuse the last contributions received for a given step, it will simply read from the dedicated local memory for that step.

The left hand side of Figure 2 shows the communication pattern for the hypercube algorithm using 8 processes. The grid on the left of the figure is composed of several squares, each representing a process. The y-axis of the grid corresponds to the rank of the process, and the x-axis to the step of the algorithm. At each step, processes connected by an edge in the figure exchange contributions and reduce the received contribution with the contribution they have sent, resulting in a partial reduction, which is to be communicated in the next step. We refer to this reduction as partial as it does not contain all contributions. After following this procedure for enough steps ($\lceil \log(P) \rceil$, where P is the number of processors), we obtain the final reduction result.

From this communication pattern, we see that process 0 receives the contribution from process 1 at step 0, a partial reduction from process 2 at step 1, and finally a partial reduction from process 4 at step 2. To implement this, we leverage the predictability of data transfer at each step to create a dedicated memory to receive the data that each process expects to receive, read from this memory, and repeat this process for the several steps in sequence. Before reading the data, we only need to wait for fresh updates if the latest contribution received is too stale. Otherwise, we use the latest contribution received.

Algorithm 1: allreduce_ssp

```
// Hypercube with d dimensions
Input  : new_contribution, slack
Output: reduction_result
begin
    clock ⟵ clock + 1
    min_clock_accepted = clock − slack
    part_red ⟵ new_contribution
    for 0 ≤ k < d do
        comm_proc ⟵ get_comm_process(k)
        // Send partial reduction
        send(part_red, clock, comm_proc)
        rcv_data ⟵ rcv_data_vec[comm_proc]
        if rcv_data.clock < min_clock_accepted then
            wait_for_update(comm_process)
            rcv_data ⟵ rcv_data_vec[comm_proc]
        end
        part_red ⟵ reduce(part_red, rcv_data)
    end
    reduction_result ⟵ part_red
end
```

To help in the explanation of the `allreduce_ssp` algorithm, we walk through an example of its execution, which corresponds to the diagram in the middle of Figure 2. Note that, to be able to determine the age (or iteration number) of the result of reducing contributions from different iterations, the algorithm relies on a logical clock. This clock value initially corresponds to the iteration in which the contribution was computed. Then, when two contributions are reduced together, the result of that reduction is associated with the minimum clock of both contributions. For instance, if a contribution with clock 2 is reduced with another from clock 3, the reduction result would be associated with clock 2.

In this example, we are zooming at process 0, currently with clock 3 in a scenario where slack is set to 1. The fact that slack is equal to 1 means that the process can use data from the current clock, in this case, clock 3, but it can also use data from the previous clock, clock 2. In the first step, the process sees that it already received data from the producer and uses it to move on to the second step. At the second step,

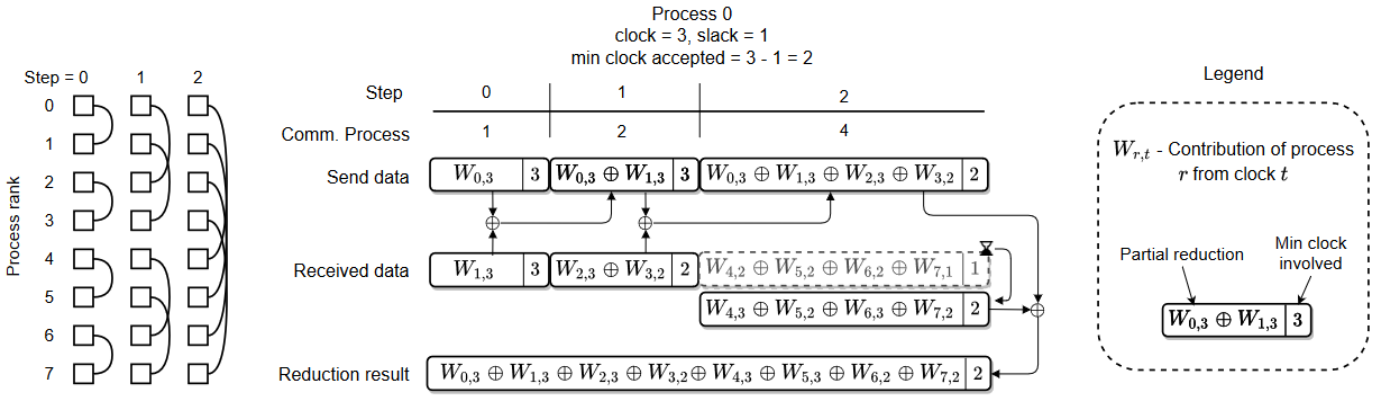

Fig. 2: On the left hand side of the figure we see the communication pattern on a hypercube using 8 processes, and in the middle of the image, we have an example of the adaptation of the hypercube to use SSP.

the process now finds received data that is stale at clock 2, but still fresh enough to be used in order to move on to the next step. Once it reaches the last step, it finds that the current received data is too stale too be used. In this case, and only in this case, the process waits until receiving a new update for the current step.

B. Broadcast and Reduce

To develop a pseudo eventually consistent Broadcast, we can rely on P-1, where P is the number of processes, `gaspi_write_notify` calls from the root or implement a classic binomial spanning tree (BST, see Figure 3) [6], specifying a certain pre-defined percentage of data to be communicated as the threshold parameter in `gaspi_bcast`. The BST is a common algorithm, which uses a binomial tree

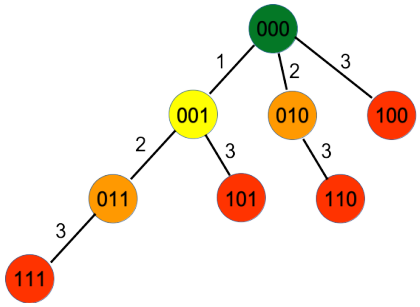

Fig. 3: Broadcast using the binomial spanning tree with eight nodes; communication stages are marked on edges with colored nodes.

structure in order to reduce the network contention. The BST implementation of `gaspi_bcast` is rather straightforward since the data is written from root/ parent to its children. The algorithm begins with determining a parent and children for each process: rank 0 is the root of the tree; the children of the process with rank p_0 are those with rank $p_0 + 2^i$, where $log(p_0) \leq i \leq \lceil log(P) \rceil$. Then, we apply the scheme in Figure 1, but only acknowledge the data transfer from the outer children to their parents.

In case of Reduce, which is the inverse of Broadcast, each child process waits for a notification from its parent indicating that the data can be sent as in Figure 1. This is crucial to avoid barriers as well as to relax the synchronization in case of multiple children writing to the parent's receive buffer simultaneously. The data is written to the segment on parent's side and reduced, and the child is also acknowledged on the completed data write. The collective continues until the root is reached and its children have contributed with their parts.

While with Broadcast, we can mostly rely on shipping less data due to the fact that all processes, but one, should receive them or at least a fraction, with Reduce we can have two strategies: one is the same as for Broadcast; the other is to still send the full amount of data but depending on the pre-defined threshold eliminate some processes. When we look on the binomial tree, Figure 3, which is not balanced, we can see that it works in stages doubling the number of involved processes on each stage. Thus, we exclude some processes depending on their id and/ or the stage id, ensuring involvement of at least the threshold amount of processes. Alternatively, we can follow the deepest path on the left hand side of the tree. A disadvantage of this approach can be a significance of the data, which in some scenarios can be on the eliminated nodes. This can be enhanced by adding weights to the data, but we omit this in our initial study.

IV. Consistent Collectives

We also propose to construct consistent collectives – such as Allreduce and AlltoAll – by relying on GASPI's API, instead of low level APIs like ibverbs. Furthermore, we outline a strategy to make Allreduce eventually consistent as well.

A. Allreduce

Since Allreduce is primarily used for large messages (from several kilobytes to hundreds of megabytes) in ML/ DL applications, our algorithm of choice is the segment pipelined ring algorithm, as it is suitable for large message sizes and promotes communication with only two neighbors. This algorithm aims to saturate bandwidth and, hence, reach high performance.

The segmented pipelined ring algorithm consists of two stages: Scatter-Reduce and Allgather. On each of these stages, a process operates with 1/P of data, sending to the next clockwise node in the ring and receiving from the previous

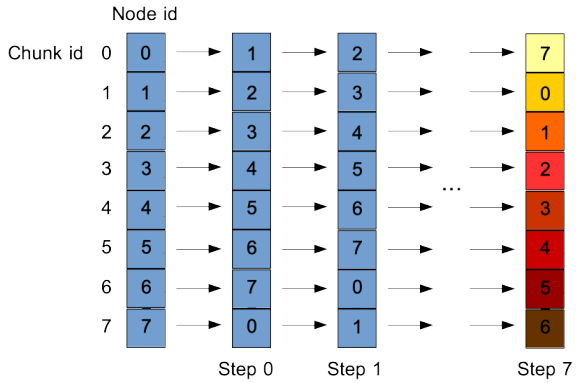

Fig. 4: Segmented pipelined ring Allreduce: Scatter-Reduce stage where each node has a complete partial result.

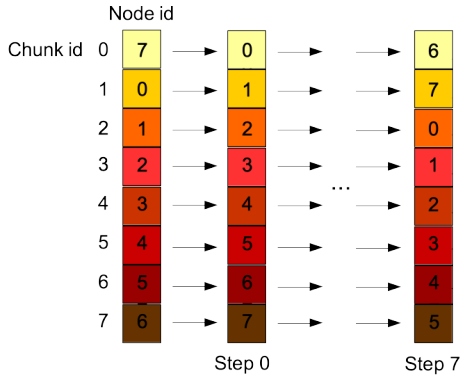

Fig. 5: Segmented pipelined ring Allreduce: Allgather stage where the partial results are broadcasted to the other nodes.

one in the ring. The Scatter-Reduce stage works as follows: in the kth step, node i will send the $i-k$th chunk and receive the $i-k-1$th chunk, reducing it into its existing data of that chunk. Hence, each of the P nodes performs a reduction of 1/P of the dataset (see Figure 4) and sends the result further. At the end of this stage, each node holds a complete result of 1/P of the data; we color this result on the plot.

In the Allgather stage, the fully accumulated partial results are distributed across all nodes, following again the pipelined ring communication with P-1 steps, as depicted in Figure 5. At the kth step, node i will send chunk $i-k+1$ and receive chunk $i-k$. After the Allgather, all nodes have access to the complete reduced dataset.

A benefit of the segmented pipelined ring algorithm is that at every stage of Allreduce each process (often) deals with its close neighbors: receiving the partial data from one and sending the partial data to another. Depending on the message size, we can also require a subspliting of messages: 1/P of the data can be divided into smaller messages to better utilize the network. As we rely upon GASPI API and not, for example, on ibverbs (which are used within GPI-2), we leave the splitting of messages to GPI-2 since it already handles this very efficiently. Hence, the GASPI implementation of Allreduce manages to use the entire memory and network bandwidth of the system. For the reduction, we used a global sum. Thus, we can hide the complete reduction effort in the communication costs. As long as the reduction effort is less time-consuming than the corresponding communication, this will also hold true for more complex reductions like user-defined reductions on user-defined data structures.

Currently, we work on extending Allreduce towards eventually consistent collectives by coupling it with a compression technique. Hence, we foresee to reduce the amount of data transferred as well as to crop some data.

B. AlltoAll

In a crucial part of the Quantum Espresso [4] application, the communication time is dominated by the `MPI_AlltoAll` collective used within a customized implementation of the Fast Fourier Transformation (FFT). In particular, `MPI_AlltoAll` consumes roughly 20-40% of FFT's total runtime.

To address this, we propose a preliminary design for an algorithmic variant of the AlltoAll collective leveraging the GASPI API. The underlying idea of this solution is to let each node write its data to the memory of all other nodes using `gaspi_write_notify` with a unique notification. Then, each node waits on the notification (`gaspi_notify_waitsome`) that some data has been written to its memory and resets this notification (`gaspi_notify_reset`). After each node has written to all nodes and has received the data and notifications from them, the AlltoAll is complete.

V. Experimental Results

We conducted our performance measurements on a set of different clusters:

- SkyLake partition at Fraunhofer with a dual Intel Xeon Gold 6132 CPU @2.6 GHz and 192 GB of memory. Nodes are connected with the 54 Gbit/s FDR Infiniband.
- Marenostrum4 cluster at BSC. Each node contains two 24-cores Intel Xeon Platinum 8160 CPUs @2.10 GHz and 96 GB of memory. Nodes are interconnected with the 100 Gbit/s Intel OmniPath HFI Silicon.
- Galileo cluster at Cineca. Each one contains two 18-cores Intel Xeon E5-2697 v4 (Broadwell) CPUs @2.30 GHz and 128 GB of memory. Nodes are interconnected through the 100 Gbit/s Intel OmniPath with OPA v10.6.

We assign one GASPI process per node (otherwise mentioned) in order to stress the communication.

a) Eventually consistent Allreduce: To experiment with our `allreduce_SSP` implementation, we use a simple Matrix Factorization algorithm using Stochastic Gradient Descent (SGD), similar to [7]. The goal of this experiment is to understand the effect of setting different values of slack on the execution time of the algorithm and its convergence. To train the model, we use the MovieLens 25M Dataset, iterate for a total of 500 iterations for the $slack = 0$ execution, and then for the other executions with different values of slack use a number of iterations necessary to achieve the same error.

Figure 6 shows the performance results of the `allreduce_SSP` evaluation using 32 GASPI processes spread across 32 nodes on MareNostrum4. The plot on the

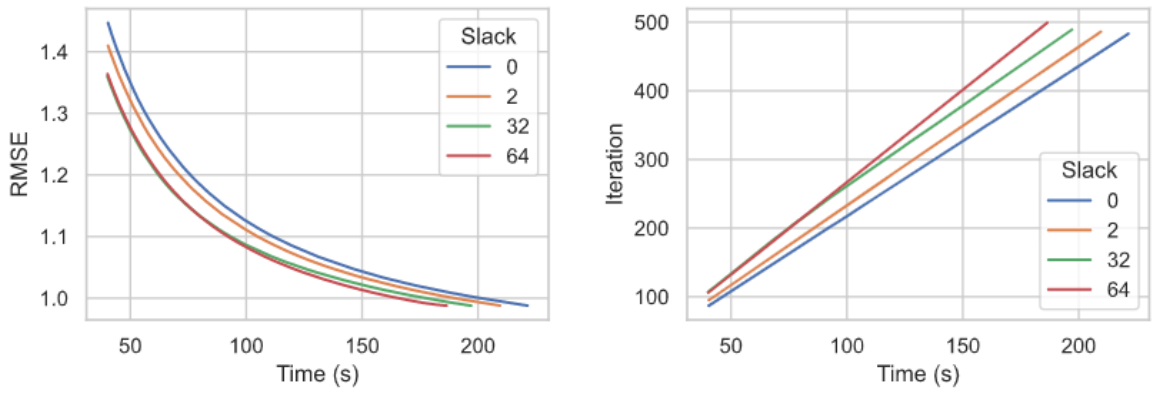

Fig. 6: Experimental results of `allreduce_SSP` impact on convergence speed of the ML algorithm (the Matrix Factorization algorithm using Stochastic Gradient Descent) on 32 nodes of the MareNostrum4 cluster.

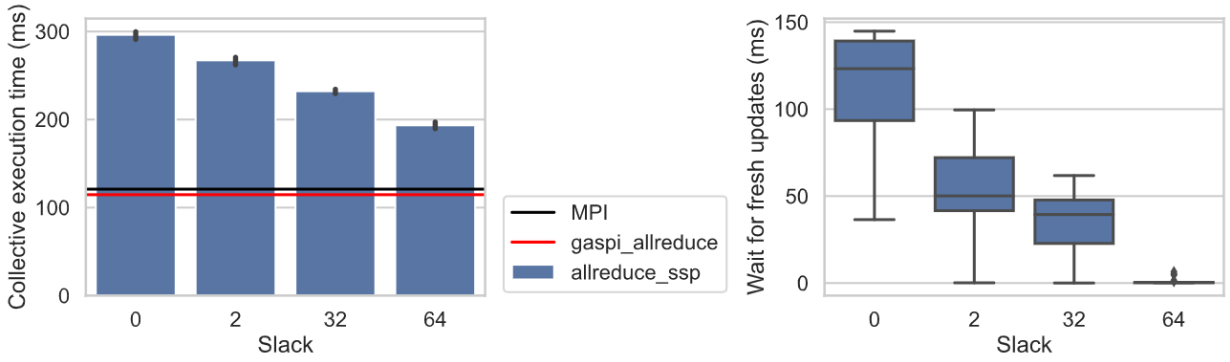

Fig. 7: Experimental results of `allreduce_SSP` collective execution speed and time spent waiting for fresh updates on 32 nodes of the MareNostrum4 cluster.

right shows the behavior of the iteration speed with respect to different values of slack. This highlights that, after a given execution time (i.e., when fixing the value on the x axis), more iterations are performed as we increase slack. This happens because of the fact that the SSP condition for advancing from one iteration to the next is more relaxed than in the traditional execution ($slack = 0$), allowing workers to lag up to slack iterations behind.

When benefiting from slack, faster processes are satisfied, on average, with potentially increasingly staler data in order to avoid waiting for new contributions. Because of this, at a certain point in their execution, processes will try to use contributions that are staler than their allowed slack. To illustrate this, we focus on the behavior of the execution using $slack = 32$ and $slack = 64$. Up to around the 100th second on the plot on the right hand side, both executions are rather similar. However, after this point we see that the $slack = 64$ execution maintains its iteration per second, while the execution with $slack = 32$ decreases its iterations per second. At this point, processes using $slack = 32$ are trying to use contributions that are staler than 32 clocks, while the execution using $slack = 64$ can continue to use staler data. Hence, this is reflected by the decrease in iteration per second in the $slack = 32$ execution.

The fact that iterations run faster may not imply that the convergence time improves since the algorithm will take more iterations to converge. However, the plot on the left in Figure 6 shows that there is an overall gain in the execution time. In particular, it shows that, for algorithms with the convergent nature such as Matrix Factorization using SGD, the Allreduce implementation following the SSP approach can increase the overall convergence speed by reaching the desired error faster. Compared to $slack = 0$, $slack = 2$ required 3 more iterations to reach the same error while being 6% faster, $slack = 32$ required 6 more iterations and was 12.3% faster, $slack = 64$ required 16 more iterations and was 19% faster.

In Figure 7, on the left plot, we see the collective execution time for the `allreduce_SSP` solution. From the plot, we can see that this solution performs significantly worse than the collective offered by MPI (default pick of Allreduce) and the `gaspi_allreduce` implementation. Note that both `gaspi_allreduce` and MPI use the Allreduce algorithms more suited for large vectors. In fact, even in the configuration for the slack value with the lowest execution time, this solution is still around 58% slower than our baseline collectives. This is due to the fact that the Hypercube algorithm shuffles the entire vector, which is better suited for small vectors, and we use vectors of a considerable size.

Regardless of the absolute performance, we see the `allreduce_SSP` collective benefits from higher values of

slack by being able to reduce, and even completely eliminate, the time waiting for fresh updates as shown in the right plot of Figure 7. As mentioned earlier, we did not expect this solution to have stellar performance. Instead, we designed it as a first attempt at adapting an existing step based Allreduce algorithm to use SSP and determine if SSP would reduce their execution time, which we confirmed.

b) Pseudo eventually consistent Broadcast: Broadcast is often used in HPC and ML/ DL code to distribute the initial data for computations. Hence, its impact on the overall application performance can be very small. In cases when Broadcast is also used within applications, especially on every iteration, careful balancing between the most relevant data and its amount can lead to the significant saving in terms of the execution time. Figure 8 shows the performance results of Broadcast using different thresholds for data, i.e., different amounts of data that is shipped. We conduct 100 executions for each message size and calculate the average time among all executions. We also compute the 95 % confidence interval that is displayed as error lines on the plots. The GASPI BST variant of Broadcast is 3.25x-3.58x faster when dealing with only a quarter of the data. Moreover, we compare these performance results against the ones with `MPI_Bcast` from Intel MPI version 2018 update 1 using their default (automatically selected) and binomial variants. These two MPI variants are clearly better compared to the `gaspi_bcast` on small data sets (up to few thousands of elements), possibly using a different binomial implementation. However, the overhead of our implementation decreases on a larger node count and for large arrays, where we can see promising performance benefits of `gaspi_bcast`. As such, we are considering to revise the BST implementation as well as to implement an alternative variant.

c) Pseudo eventually consistent Reduce: We conduct similar tests for Reduce and report the respective results in Figure 9. For each message size, we also run the benchmark 100 times and compute both the average and the 95 % confidence interval. For different message sizes, the difference between the usage of 25 % and 100 % of the data in `gaspi_reduce` increases rapidly with the message size and for 8 Mb it is roughly 5x. We compare the performance results of `gaspi_reduce` against the ones with `MPI_Reduce` from Intel MPI version 2018 update 1 using their default (automatically selected) and binomial variants. For small arrays, MPI outperforms all our variants of Reduce. However, for larger arrays, starting from 10,000 elements, the situation changes: the automatically selected variant (from the pool of 14) of `MPI_Reduce` is still (1.96x) faster, while `gaspi_reduce` is roughly by 38 % faster than the MPI binomial variant.

We also provide performance results of another `gaspi_reduce` implementation when the full amount of data is sent but only a certain (according to the predefined threshold) percentage of processes is engaged in communication, see Figure 10. Hence, the leaves that are far from the root are excluded. This implementation demonstrates slower performance compared to the previous implementation of `gaspi_reduce`, working only on a fraction of data, however it is still better then the MPI binomial variant. The lines for 75 % and 100 % show identical performance due to the fact that after 50 % it is difficult to get much improvement since 50 % of all processes are added on the last stage of the algorithm as depicted in Figure 3.

d) Consistent Allreduce: We carry out the `gaspi_allreduce` experiments on the SkyLake nodes of the Fraunhofer ITWM's cluster. We compare its performance results against the ones of `MPI_Allreduce` from Intel MPI version 2018 update 1 with the standard settings, which ensures that the optimal settings are being selected. We allocated one process per node and assigned the same amount of work (vector size) per node in order to stress communication of the collective. Figure 11 reports the timings (the average time and the 95 % confidence level among 100 executions) for arrays of doubles of sizes 10,000 and 1,000,000. Note that our implementation aims to target large vector sizes, whereas the Intel MPI library is equipped with a dozen of implementations, which in fact are all in use here, targeting various message sizes and topologies. Hence, the results of `MPI_Allreduce` are significantly better than `gaspi_allreduce` for vectors of size 10,000. However, `gaspi_allreduce` performs better for larger vectors, e.g., with 1,000,000 elements, showing performance benefits of 1.78x and 2.26x when compared to the Shumilin's ring (mpi7 on the plot) and ring (mpi8) variants of `MPI_Allreduce`, respectively; the Shumilin's ring is the best performing variant of MPI among 12 existing, while the ring supposedly implements the same segmented pipelined ring algorithm. It is worth mentioning that, compared to the implementations of the segmented pipelined Allreduce with MPI, we eliminate global synchronizations at the end of both Scatter-Reduce and Allgather phases and instead use the GASPI weak lightweight synchronization via notifications; for example, to indicate that the process is ready to receive data or to acknowledge arrival of data as depicted in Figure 1.

In addition, we conduct tests by focusing on a range of message sizes, for instance from 1,024 elements and up to eight millions with the step of 2. Figure 12 demonstrates the results of this evaluation: 1) until we reach a message size of 1,048 Kb, the MPI implementation of Allreduce is faster; 2) starting from a message size of 2,097,Kb our implementation outperforms all MPI variants and reaches its peak of 2.07x and 2.13x for ring and Shumilin's ring variants, respectively, for a message size of 67,108 Kb (or 8,388,608 elements). We believe that this trend will continue and the gap will become larger on more nodes since the MPI allreduce curves show faster growth. This is particularly thanks to the GASPI's asynchronous one-side communication with weak synchronization.

e) Consistent AlltoAll: We conducted our performance experiments and compared the GASPI implementation of AlltoAll against that of `MPI_AlltoAll` of Intel MPI v.18.0 on Cineca's Galileo cluster.

Figure 13 reports the performance results (obtained as an average over 100 runs) for various message sizes. Since we

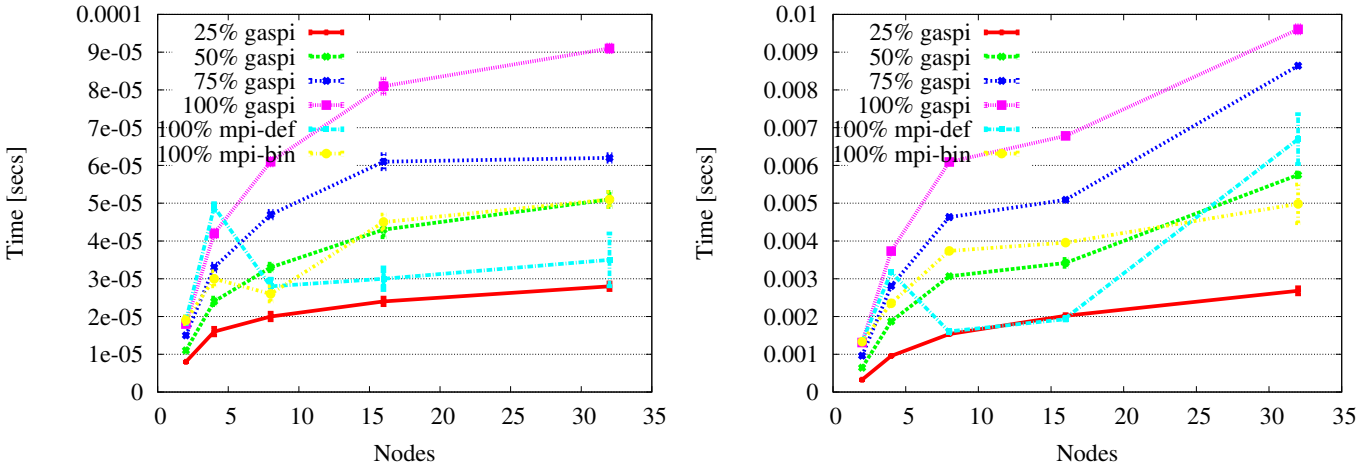

Fig. 8: Performance results of Broadcast on SkyLake nodes: for vectors of 10,000 double precision elements on the left and of 1,000,000 elements on the right. mpi-def stands for the default variant of Broadcast, while mpi-bin corresponds to the binomial variant.

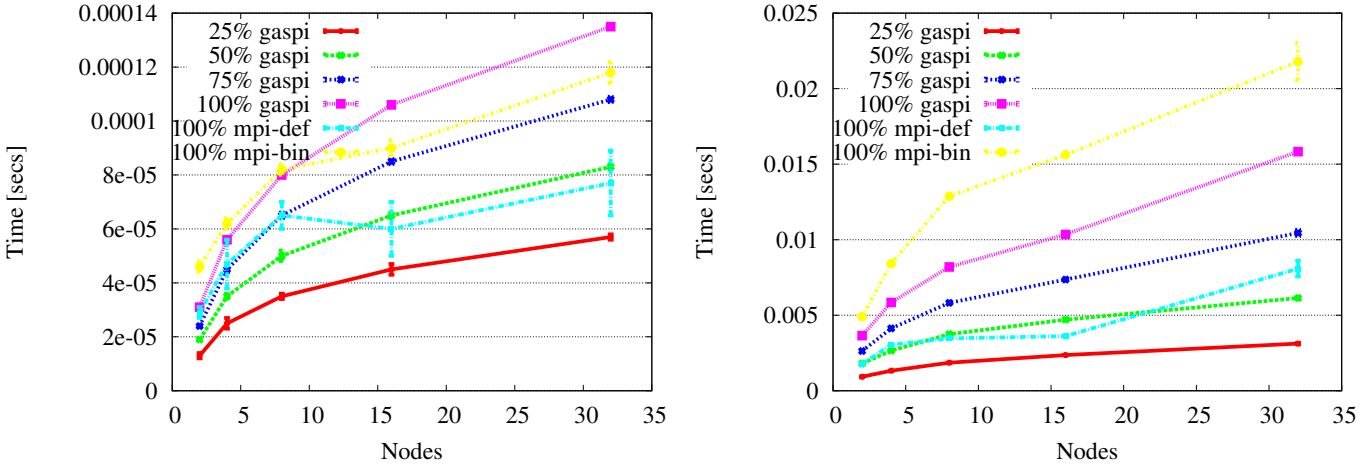

Fig. 9: Performance results of Reduce on SkyLake nodes: for vectors of 10,000 double precision elements on the left and of 1,000,000 elements on the right. mpi-def stands for the default variant of Reduce, while mpi-bin corresponds to the binomial reduction.

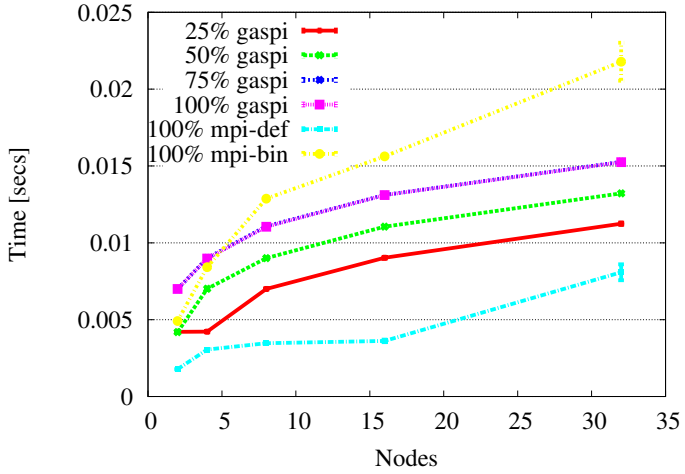

Fig. 10: Performance results of Reduce operating on the full amount of data for 1,000,000 doubles on SkyLake nodes; xx % lines corresponds to at least xx % of processes involved.

aim to have a hybrid programming model implementation, we set four GASPI/ MPI processes per node. We run our experiments using 4, 8, and 16 nodes: this is marked on each line as gaspi$x$ or mpi$x$, where $x = 4, 8,$ or 16. Note that we used Intel MPI v.18.0 with the standard settings, which ensures that the optimal settings are being selected. The performance of both implementations is similar up to a message size of about 1,024 bytes. The situation begins to change from a message size of 2,048 bytes, where our implementation of GASPI AlltoAll begins to outperform the vendor-provided MPI version, reaching the outstanding performance for a message size of 32,768 bytes; the performance gain is 2.85x, 5.14x, and 5.07x on 4, 8, and 16 nodes, accordingly.

It is important to note that the message size needed by the FFT miniapp when using `MPI_AlltoAll` is in the range of 6Kb-24Kb, i.e., in the range where the GASPI version outperforms the vendor-provided MPI implementation. Since `MPI_AlltoAll` consumes about 20-40% of FFT's total runtime, we expect a significant reduction of the total execution time in the Quantum Espresso application (whose implementation is currently in progress). A GASPI equivalent of `MPI_AlltoAllV`, also used under certain conditions in the minapp, is currently being built using the same scheme as

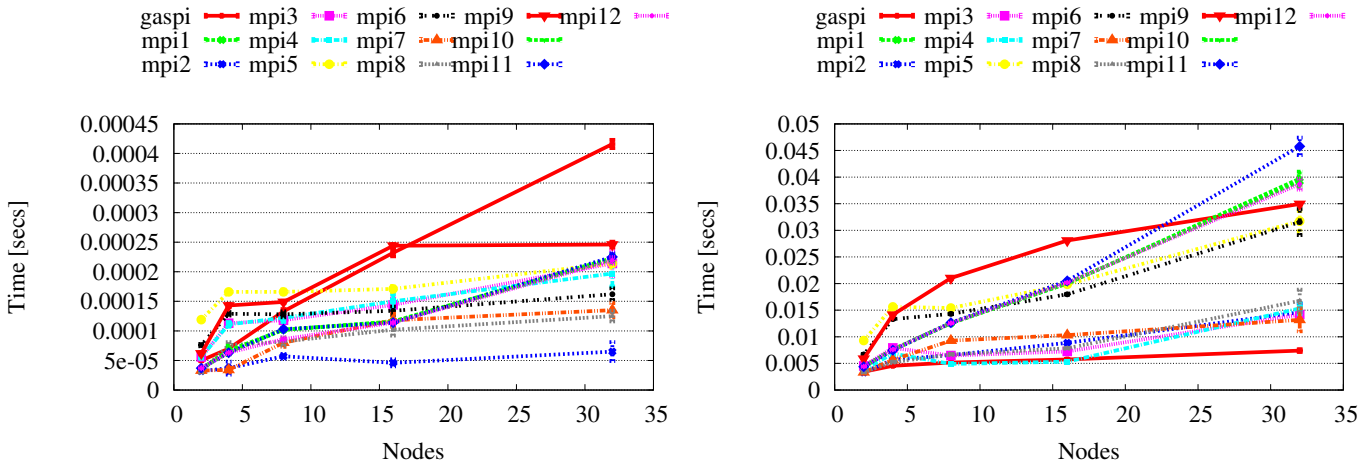

Fig. 11: Performance results of Allreduce on SkyLake nodes: for vectors of 10,000 double precision elements on the left and of for 1,000,000 elements on the right. mpi1 stands for the recursive doubling variant; mpi2 – Rabenseifner's; mpi3 – Reduce + Bcast; mpi4 – topology aware Reduce + Bcast; mpi5 – binomial gather + scatter; mpi6 – topology aware binominal gather + scatter; mpi7 – Shumilin's ring; mpi8 – ring; mpi9 – Knomial; mpi10 – topology aware SHM-based flat; mpi11 – topology aware SHM-based Knomial; mpi12 – topology aware SHM-based Knary.

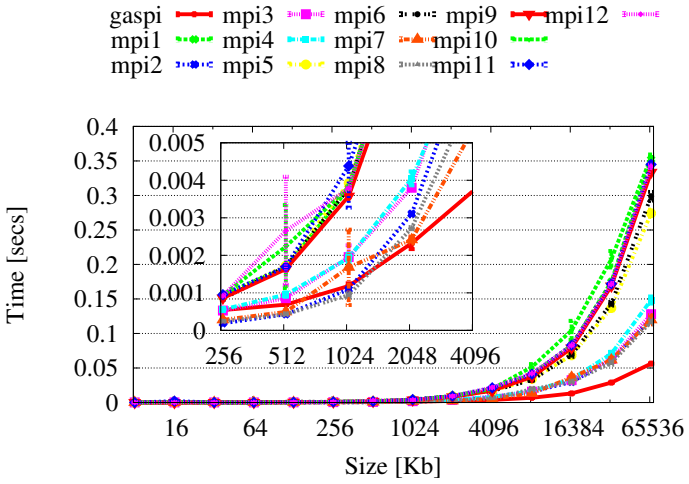

Fig. 12: Performance results of Allreduce on 32 SkyLake nodes for various message sizes.

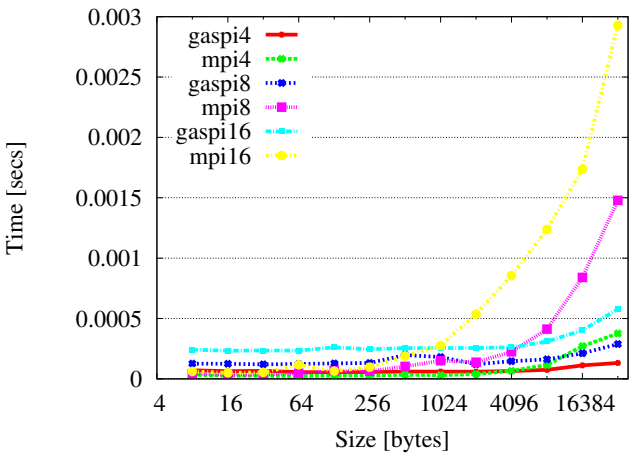

Fig. 13: Performance results of `gaspi_alltoall` compared against MPI on the Galileo cluster at CINECA; GPI-2 installation is from the next branch on GitHub, which is v1.4.0, and MPI implementation comes from the Intel MPI v18.0 library.

for the `gaspi_alltoall` collective.

VI. Related Work

Depending on message sizes and network architecture, Allreduce implementations span a wide range of algorithms, from ring-based algorithms, binomial spanning-tree implementations [6], tree algorithms [8], or Butterfly like algorithms [9]. In [10] the authors designed an n-way dissemination algorithm for the GASPI API; this algorithm is suitable for small message size. The dissemination algorithm has been presented by Hensgen et al. in 1988 [11]. Due to its speed it is used in different programming APIs and libraries like the MPICH implementation of MPI for barrier implementations. In [10], the n-way dissemination algorithm was used in a way that neither leverages partial reductions of a 2-way dissemination (and their associated out-of-band delivery to late dissemination stages), nor notified communication in shared windows.

The GASPI programming model [1] primarily targets multi-threaded or task-based applications, hence GASPI+X. However, in order to support migration of legacy applications (with a flat MPI communication model) towards GASPI, we have extended the concept of shared MPI windows [12], [13] towards a notified communication model [14], [15] in which the processes sharing a common window become able to see all one-sided and notified communication targeted at this window. We enabled communication from and to a shared memory region to all processes, which share the window. While MPI-3 readily supports this model with MPI 2-sided

and 1-sided communication, we aimed to support the notified and one-sided communication in GASPI. Since GASPI does not require a dedicated receiving process, we can avoid the detrimental effects of late receivers. Nevertheless, all processes are still able to test for completion of incoming messages without additional synchronization effort.

In [14], [15] we also designed Allreduce based on pipelined rings and notified communication in shared windows. This implementation delivered up to the 3x performance boost compared to the best Intel MPI implementations v5.1.2 on the Salomon IT4I cluster (Infiniband FDR). We extended this idea to Allgather(V) and Allreduce with an adaptation of the dissemination algorithm [16], achieving up to 2x-4x performance improvements compared to the best performing MPI implementations on the Salomon IT4I cluster and the Beskow Cray XC40 cluster at PDC, KTH.

VII. Conclusions and Future Work

In this article, we presented our ideas for adopting some classic algorithm for collective operations – like Binomial Spanning Tree and segmented pipelined ring – to implement Broadcast, Reduce, and Allreduce with GASPI. While with Broadcast and Reduce we aimed to be generic but vary the amount of data used or processes involved (mimicking eventual consistency), with Allreduce we targeted (very) large message sizes. The Allreduce implementation leads to 2x faster execution compared to a dozen of the vendor-specific implementations. We also implemented AlltoAll following a rather simple but well-performing pattern, resulting in 2.8x-5.1x performance improvements compared to the MPI's AlltoAll default variant. Furthermore, we designed and implemented in GASPI a novel Allreduce following the Stale Synchronous Parallel model (`allreduce_ssp`). `allreduce_ssp` reduces the waiting time and synchronizations by using stale contributions. This approach is suitable for ML/ DL computations. Our implementation, which is based on Hypercube, was not able to outperform the MPI standard, however we observed the desired effect of using slack in terms of faster convergence for Matrix Factorization. In order to improve `allreduce_ssp`, we consider to adapt more efficient Allreduce algorithms, e.g. the presented pipeline ring algorithm, and to explore the idea of the Parameter Server architecture, which is the setting where we usually find the SSP model. All our developments are available under the EPEEC GitHub repository: https://github.com/epeec.

Our ultimate goal is to provide a library of collectives within the GASPI ecosystem for both the HPC and ML/ DL communities by leveraging the GASPI API and focusing on the design of collectives for various data sizes and/ or application needs as in case of the `allreduce_ssp` variant. Furthermore, we are also working on the compression library and foresee to design and develop another version of eventually consistent collectives by coupling this compression library with the consistent collectives.

Acknowledgement

The research leading to these results has received funding from the European Union's Horizon2020 research and innovation programme under the EPEEC project, grant agreement No 801051, and from FCT under UIDB/50021/2020.

References

[1] C. Simmendinger, M. Rahn, and D. Grünewald, "The GASPI API: A failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures," in *Sustained Simul. Perf.* Springer, 2015, pp. 17–32.

[2] D. B. Terry, M. M. Theimer *et al.*, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *ACM SOSP*, 1995, p. 172–182.

[3] H. Cui, J. Cipar *et al.*, "Exploiting bounded staleness to speed up big data analytics," in *USENIX ATC*, 2014, pp. 37–48.

[4] P. Giannozzi, S. Baroni *et al.*, "Quantum espresso: a modular and open-source software project for quantum simulations of materials," *J. Phys.: Condens. Matter*, vol. 21, no. 39, p. 395502 (19pp), 2009.

[5] R. Belli and T. Hoefler, "Notified access: Extending remote memory access programming models for producer-consumer synchronization," in *IPDPS*. IEEE, 2015, pp. 871–881.

[6] N.-F. Tzeng and H.-L. Chen, "Fast compaction in hypercubes," *IEEE TPDS*, no. 1, pp. 50–56, 1998.

[7] J. Oh, W.-S. Han *et al.*, "Fast and robust parallel sgd matrix factorization," in *ACM SIGKDD KDD*, 2015, p. 865–874.

[8] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM TOCS*, vol. 9, no. 1, pp. 21–65, 1991.

[9] E. D. Brooks, "The butterfly barrier," *Int. J. Parallel Prog.*, vol. 15, no. 4, pp. 295–307, 1986.

[10] V. End, R. Yahyapour *et al.*, "Adaption of the n-way dissemination algorithm for gaspi split-phase allreduce," *INFOCOMP 2015*, p. 25, 2015.

[11] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *Int. J. Parallel Prog.*, vol. 17, no. 1, pp. 1–17, 1988.

[12] T. Hoefler, J. Dinan *et al.*, "Mpi+ mpi: a new hybrid approach to parallel programming with mpi plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[13] W. Gropp, T. Hoefler *et al.*, *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.

[14] D. Akhmetova, L. Cebamanos *et al.*, "Interoperability of gaspi and mpi in large scale scientific applications," in *Wyrzykowski R., Dongarra J., Deelman E., Karczewski K. (eds) PPAM 2017. Springer LNCS*, vol. 10778, 2018, pp. 277–287.

[15] C. Simmendinger, R. Iakymchuk *et al.*, "Interoperability strategies for gaspi and mpi in large scale scientific applications," *IJHPCA*, 2018.

[16] M. A. Al Ahad, C. Simmendinger *et al.*, "Efficient Algorithms for Collective Operations with Notified Communication in Shared Windows," in *IEEE/ACM PAW-ATM at SC18*, 2018, pp. 1–10.