Efficient Algorithms for Collective Operations with Notified Communication in Shared Windows

Muhammed Abdullah Al Ahad*, Christian Simmendinger†, Roman Iakymchuk*,
Tiberiu Rotaru‡, Erwin Laure*, Stefano Markidis*
*KTH Royal Institute of Technology, EECS, CST/PDC, 100 44 Stockholm, Sweden
Emails: {maaahad, riakymch, erwinl, markidis}@kth.se
† T-Systems Solutions for Research GmbH, 70563 Stuttgart, Germany
Email: Christian.Simmendinger@t-systems-sfr.com
‡ Fraunhofer ITWM, 67663 Kaiserslautern, Germany
Email: tiberiu.rotaru@itwm.fraunhofer.de

***Abstract*—Collective operations are commonly used in various parts of scientific applications. Especially in strong scaling scenarios collective operations can negatively impact the overall applications performance: while the load per rank here decreases with increasing core counts, time spent in e.g. barrier operations will increase logarithmically with the core count. In this article, we develop novel algorithmic solutions for collective operations – such as Allreduce and Allgather(V) – by leveraging notified communication in shared windows. To this end, we have developed an extension of GASPI which enables all ranks participating in a shared window to observe the entire notified communication targeted at the window. By exploring benefits of this extension, we deliver high performing implementations of Allreduce and Allgather(V) on Intel and Cray clusters. These implementations clearly achieve 2x-4x performance improvements compared to the best performing MPI implementations for various data distributions.**

***Index Terms*—Collectives, Allreduce, Allgather, AllgatherV, MPI, PGAS, GASPI, shared windows, shared notifications.**

I. Introduction

Although the Message Passing Interface (MPI) has been considered as the de facto standard, alternative models – like *Global Address Space Programming Interface (GASPI)* [1] with its open source implementation, *GPI-2* – also evolve in parallel. One of their key features is to take into account modern hardware architectures.

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. In passing we note that similar weak synchronization primitives might also appear in the upcoming MPI-4 standard [2]. 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.

As of today the GASPI programming model [1], [3] targets multi-threaded or task-based applications. However, in order to support migration of legacy applications (with a flat MPI communication model) towards GASPI, we have extended [4] the concept of shared MPI windows [5], [6] towards a notified communication model in which the processes sharing a common window become able to see all one-sided and notified communication targeted at this window. We have exposed the 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 here aim support notified, one-sided communication in GASPI. As - due to its one-sided notified communication - 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.

As a first proof-of-concept we have used this concept of node-wide visible communication for the implementation of collectives, which can make use of the underlying ideas. As test cases we choose two collectives, namely Allreduce and Allgather(V). Allreduce is a widely used operation in scientific codes, especially it is frequently invoked in iterative solvers like GMRES. Due to implicit global synchronization points [7], Allreduce can consume up to 10 % of the total execution time as in the pipe flow case on up to 512 cores using NEK5000 [8]. Allgather(V) is another appealing collective to verify our approach due to the variable size of the incoming data from every process. To sum up, the main contribution of this paper is the usage of node-wide visible, one-side notified communication for developing collectives.

This article is structured as follows: We present algorithmic and implementation details for two widely used collective operations: Allgather(V) (Section II) and Allreduce (Section III). Using the unique strength of notified communications, we present benchmark results which show the benefits of this novel approach. Finally, Section IV reviews related work,

while Section V draws conclusions and outlines future work.

II. ALLGATHER(V)

In this section we propose algorithmic solutions in details for the Allgather(V) collective operation as well as show its performance results in comparison against the state-of-the-art MPI implementations. For the sake of the following explanation, we assume that `nProc` is the number of participating processes; `nProcLocal` is the number of node-local processes; `nProcMaster` is the number of nodes; `iProc` is the global rank; and `iProcLocal` is the node-local rank of a process.

A. Algorithmic Solution

The algorithm was developed based on the assumption that every process knows the receiving data sizes from every other processes and their corresponding displacement in the receiving buffer. The displacements in the receiving buffer is the same for each process and hence a process in turn knows the position where to write to a remote process. The idea of shared windows with shared notifications has been deployed in developing the algorithm for `gaspi_allgatherv`, see Algorithm 1. The flow chart of the full procedure is depicted in Figure 1.

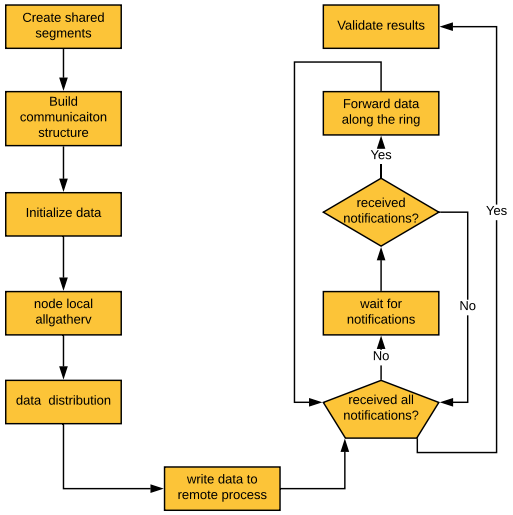


Fig. 1: Flow chart for Allgather(V).

In the following sections we guide you through the algorithm to explain the logic as well as the purpose of each code blocks.

1) Creating shared segments : init_shared_segments(): This part of the algorithm is responsible for creating local and shared data buffers. Each process owns a local buffer to initialize the data to be transferred to other processes. There will be in total five shared windows and out of them only one will be used as shared `gaspi_segment`. The size and type of each shared windows are presented in Table I.

shared windows	minimum size	type
shared_notification	$nProcLocal * sizeof(int)$	`int*`
shared_nblock_mpi_win	$nProcLocal * sizeof(int)$	`int*`
shared_nid_util_mpi_win	$2 * nProc * sizeof(int)$	`int*`
shared_data_block_mpi_win	$total\_nblock\_in\_node * sizeof(data\_bolock)$	`data_block` 2
shared_segment_ptr	$total\_vector\_length * sizeof(int)$	`gaspi_pointer_t`

TABLE I: List of shared windows used in Allgather(V).

The shared MPI window $shared\_notification$ is used to store and update received notifications by each node-local process at position $iProcLocal$ in the buffer. During the data communication step, see line 13 in Algorithm 1, each local process sums up the all value of this buffer to check whether total received notifications by a node is equal to the expected notifications. Using shared window $shared\_nblock\_mpi\_win$ a process stores its available number of data block at position $iProcLocal$ in the buffer during $LoadBalancePart\_1$, see line 4 in Algorithm 1. The purpose of $shared\_nid\_util\_mpi\_win$ is to store intermediate information for calculating the starting data block id of each process. The first half of this buffer is used during $LoadBalancePart\_1$, line 4 in Algorithm 1 and the remaining half is used during $LoadBalancePart\_2$, line 7 in Algorithm 1, to store the starting data block id of each process. To store all data block related information defined by the data structure in Listing 2, each process utilizes the shared window named $shared\_data\_block\_mpi\_win$. Offset for a process, for example local process 3 is calculated by summing up the data block contributed by each local process up to 2. Here, each local process uses the information available in the shared window $shared\_nblock\_mpi\_win$. The shared window $shared\_segment\_ptr$ is bound as a `gaspi_segment` and used as result buffer and for both sending and receiving data.

2) Initialize communication structure: init_com_structure(): In this part of the algorithm each process binds the shared buffer named $shared\_segment\_ptr$ as the `gaspi_segment` to make the buffer globally accessible by every single `GASPI` processes. This binding is performed by using the `GASPI` built-in procedure `gaspi_segment_bind`. After the `gaspi_segment` creation, each process registers the segment using `gaspi_segment_register` to only one remote process to allow the remote process to write to the segment. A process send data to a process with the same local rank to its left node. Hence, every process needs to register its `gaspi_segment` to the process with same local rank on its right node. The remote process for registering `gaspi_segment` is determined using the following formula:

$$((iProc + nProc + nProcLocal)\%nProc) \tag{1}$$

3) Initialization of data array: init_array(): Before starting the actual `Allgather(V)`, each process needs to initialize its corresponding data and needs to store it in its local buffer. To make sure that the data was initialized by each process is different and also varies in each iteration, the following formula adopts data initialization:

$$data = (iProc + 1) * (it + 1) \tag{2}$$

Algorithm 1 `Allgather(V)` using shared windows and shared notifications

```
1: init_shared_segments();
2: init_com_structure();
3: init_array();
4: LoadBalancePart_1;
5: local_allgatherv();
6: MPI_Barrier
7: LoadBalancePart_2;
8: MPI_Barrier
9: LoadBalancePart_3;
10: for all_my_data_block do
11:     write_notify(segment_id, target_process, ..., block_size, block_id);
12: end for
13: while received_notifications_by_node! = expected_notifications_for_node do
14:     waitsome_for_notifications(segment_id, notification_range, received_id, ..., GASPI_TEST);
15:     received_notifications + +;
16:     if received_block_source_node! = target_process_node then
17:         write_notify(segment_id, target_process, ..., size_of_received_data, received_notification_id, ...);
18:     end if
19: end while
```

Here it represents the current iteration.

4) Determining data block size and distributing workload for load balancing: In case of variable data sizes, there is a possibility that one process on a node may have a major parts of data and, hence, will have to do more work compared to the other node-local processes. This will definitely hamper the performance of the procedure. The best way to deal with it is to distribute the data within the node-local processes as evenly as possible. The distribution of the data in a node is calculated based on the total number of data blocks associated with the node. The number of data block in a node is calculated by defining the maximum size of a data block. The important thing regarding the maximum data block size is that, it should be large enough so that total number of data block does not exceed the maximum number of notification id provided by `GASPI`, which is $65,536$. As distributing workload among node-local processes requires the information from each node-local processes, synchronization is necessary and, hence, load balance within a node is performed in three different stages as described in the following section.

$$shared\_nblock\_mpi\_win[iProcLocal] = (int)ceil\left(\frac{(float)ivlen[iProc]}{(float)mSize}\right) \quad (3)$$

Here $mSize$ is the maximum data block size.

$$my\_working\_nblock = \left(\frac{total\_node\_block + nProcLocal - iProcLocal - 1}{nProcLocal}\right) \quad (4)$$

Here $total\_node\_block$ refers to the total number of data blocks in a node.

5) Distribution of workload among node-local processes: LoadBalance Part 1: In this part of Algorithm 1 see line 4, each process in a node computes its associated number of data blocks using (3) and stores it in $shared\_nblock\_mpi\_win$. Once every process in a node publishes its available data block, the total number of data blocks for that node can be calculated and then every process of that node will be given its working number of data block according to (4) during the $LoadBalancePart\_3$. This will ensure that no data blocks will contain data from two or more local processors. This is necessary as the position of data in the received buffer for two consecutive processes will not necessarily be contiguous.

In this implementation for `Allgather(V)`, data block is written to the remote process using the `GASPI` procedure called `gaspi_write_notify` to impose tighter synchronization on data movement. The remote process will be notified when the data write is finished and the remote process can verify whether a data block is written to its local segment through the respective wait procedure `gaspi_notify_waitsome`. Every process may expect 0 or more data blocks from remote processes, i.e notifications, so to make sure not to miss any incoming data notifications, every notification id should be unique. Moreover, `gaspi_notify_waitsome` requires us to provide the id range of expected notifications in the form of the starting notification id and the total number of notifications. To fulfill these constraints we need to provide each data block with an unique id that is continuous among all nodes and this block id will be used as the notification's id for the associated data block. This means, if the last $block\_id$ of node 0 is 10, then the $block\_id$ of the first block of the local process 0 of node 1 will be 11 and so on. This requires that every process in a node is aware of the starting id for its data block.

Every process in a node should be aware of starting id of each other remote processes as well. Whenever a process receives a data block from a remote process, the receiving

process will have to forward it to another process, hence it requires the offset and the size of the received data block. The notification will provide the process with the data block source rank and the unique global block id. But, to compute the offset and the size of the received data block, we need to map the global block id to the process local block id. Once we know the process local block id of the received data block, we can calculate the offset and the size of the received data block as we know the maximum data block size.

To reduce the run time, this part of the computation is performed in parallel in a node. Every process in a node is responsible for calculating the starting id of each other processes with the same local rank. Part of this computation is performed in lines 4-6 Algorithm 1 and the remaining is done in lines 7-8 Algorithm 1. To calculate the starting block id of a process of a particular node, at this part of the algorithm a process in a node first sums up the available data block of every processes with the same local rank of each lower node of the computing node and stores it in the shared window $shared\_nid\_util\_mpi\_win$. For example: to compute the starting id of a process in the node 3 with the global rank 10 and the local rank 1, every process in a node first sums up the data blocks representing by each remote process with the same local rank up to node 2 followed by a barrier. Code snippet of this parallel part can be viewed in Listing 1 where $tmp\_shared\_nid\_util\_mpi$ represents $shared\_nid\_util\_mpi\_win$ and $ivlen$ is the list of data size of each process.

Listing 1: `Allgather(V)` computation of starting block id of each process

```
1
2 for(int node = 0; node < nProcMaster; node++){
3    int nid_util_offset = node * nProcLocal + iProcLocal;
4    tmp_shared_nid_util_mpi[nid_util_offset] = 0;
5    for(int i = 0; i < node; i++){
6       int proc = i * nProcLocal + iProcLocal;
7       tmp_shared_nid_util_mpi[nid_util_offset] +=
             (int)ceil((float)ivlen[proc] / (float)mSize);
8    }
9 }
```

6) Local Allgather(V) : local_allgatherv(): During the local Allgather(V), each process copies its associated data from its local data buffers to the respective position in the shared buffer named $shared\_segment\_ptr$. Performing local Allgather(V) and storing data in the shared buffer makes it possible for a process to get access to the data from another process or processes available in the shared buffer.

7) Distribution of workload among node-local processes: LoadBalance Part 2: This part starts with the completion of computing the starting data block id of each process and to do so each process in a node use the information calculated in lines 4-6 in Algorithm 1.

At this point, every process at least knows the starting block id of each processes having the same local rank. By using the starting block id of itself, every process in a node publishes all of its data blocks related information to the shared window $shared\_data\_block\_mpi\_win$ of derived type `data_block`, see Listing 2. As a process in a node might be in charge of data block(s) of another process, a barrier is placed for all node-local processes after filling the shared buffer $shared\_data\_block\_mpi\_win$ to make the data available to every process in a node during the time of data movement.

Listing 2: `data_block` data type.

```
1 typedef struct {
2   int block_source_proc;      // source process of the
        data block
3   int block_id;               // unique block id
4   int block_sz;               // size of the data block in
        int
5   int offset;                 // offset related to receive
        buffer
6 } data_block;
```

8) Distribution of workload among node-local processes: LoadBalance Part 3: Once data blocks with their associated information are stored in the shared window $shared\_data\_block\_mpi\_win$ by each node-local processes, each process calculates its number of working data blocks and the offset of the first data block in the shared memory. A code snippet for this part of the load balance is illustrated in Listing 3

Listing 3: `Allgather(V)` computation of working number of block and block offset

```
1    int total_node_block = 0;
2    for(int i = 0; i < nProcLocal; i++){
3       total_node_block += tmp_shared_nblock_mpi[i];
4    }
5
6    int my_working_nblock = (total_node_block + nProcLocal
          - iProcLocal -1) / nProcLocal;
7    int my_working_block_offset = 0;
8    for (int i = 0; i < iProcLocal; i++) {
9       my_working_block_offset += (total_node_block +
             nProcLocal - i -1) / nProcLocal;
10   }
```

At this point, each local process is aware of all required information related to the node-local data blocks and hence the data blocks the process is responsible to write to the remote target process.

9) Data movement based on shared windows and shared notifications: The real data communication among remote processes happens in this part of the algorithm lines $10 - 19$ in Algorithm 1. Each process starts the data communication by writing all of the data blocks the process is responsible for to the target process one after another. Once this is accomplished, the process then waits for any incoming notifications from the remote process.

As we are using shared windows along with shared notifications, every processes in a node waits for the incoming notifications until the expected notifications for that node is fulfilled. A process in a node will catch any notification aiming for that node, does not matter which process is the target one. Handling notifications this way helps to improve the performance by getting more output from the faster process.

10) Graphical representation of the algorithm: For the simplicity of graphical representation of the algorithm, we assume each process contributes one chunk of data. Every process stores the initialized data to its corresponding local buffer as in Figure 2a. During the `local Allgather(V)` each

process copies their associated data to the respective position in the shared buffer $shared\_segment\_ptr$, see Figure 2b. After this step every local process attains access to all node-local data. Then, each process starts writing its associated data to the remote target process as depicted in Figure 2c. After posting the request to write node-local data, each process starts waiting for the incoming data from the remote process. Whenever a process receives data from a remote process, the process then forwards it to the remote target process, see Figure 2d. After receiving all data from every other process, the procedure completes, Figure 2e.

B. Performance Results

Performance tests are carried out based on three different types of initial data distribution:

1) Regular;
2) Linearly decreasing;
3) Broadcast.

If m and m_i represent the total data size and the size of data in process i, respectively, and $nProc$ is the total number of `MPI` processes, then the initial data distribution can be described as follows [9]:

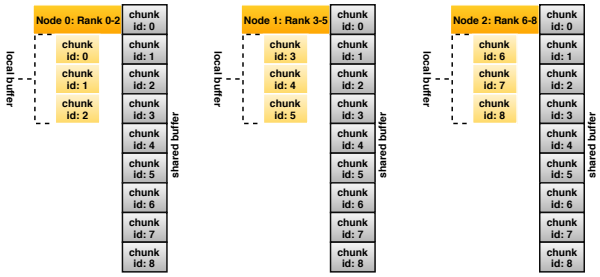
1) Regular: $m_i = c$ such that $m = nProc * c$;
2) Linearly decreasing: $m_i = 2c\frac{(nProc-1-i)}{nProc-1}$ such that $m = nProc * c$;
3) Broadcast: $m_0 = m$ and all other $m_i = 0$.

All of the tests are conducted on a Cray XC40 cluster at PDC Center for High Performance Computing. The system is equipped with the following configuration:

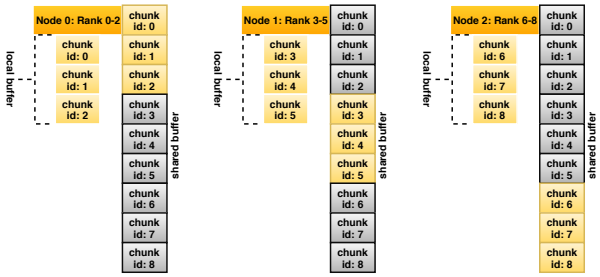
- 2 x Intel CPUs per node:
  - ☐ 9 of the cabinets have Xeon E5-2698v3 Haswell 2.3 GHz CPUs (16 cores per CPU)
  - ☐ 2 of the cabinets have Xeon E5-2695v4 Broadwell 2.1 GHz CPUs (18 cores per CPU)
- High speed network Cray Aries (Dragonfly topology).

All simulations were executed 100 times using 32 `MPI/GASPI` processes per node with up to 32 nodes. The reported time here is the median of all executions and we compare our implementation with four different `Intel` implementations of $MPI\ Allgatherv$: 1) $I\_MPI\_ADJUST\_ALLGATHERV\_1$: Recursove doubling; 2) $I\_MPI\_ADJUST\_ALLGATHERV\_2$: Bruck's; 3) $I\_MPI\_ADJUST\_ALLGATHERV\_3$: Ring; 4) $I\_MPI\_ADJUST\_ALLGATHERV\_4$: Topology aware Gatherv + Bcast.

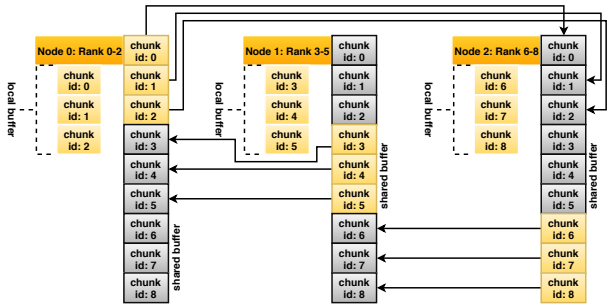
The illustration of the performance of our implementation is presented in Figure 3. As it can be seen, in all of the three cases the runtime for our implementation is almost constant despite the growth of the number of nodes, i.e. the number of processors. This is not unexpected as the main aim of this implementation is to scale the number of communications with the number of nodes instead of the number of processors. It is also noteworthy to state that compared to the `MPI` implementations we achieve at least 2x, 4x, and 3x performance boost for the regular (see Figure 2(f)), linearly



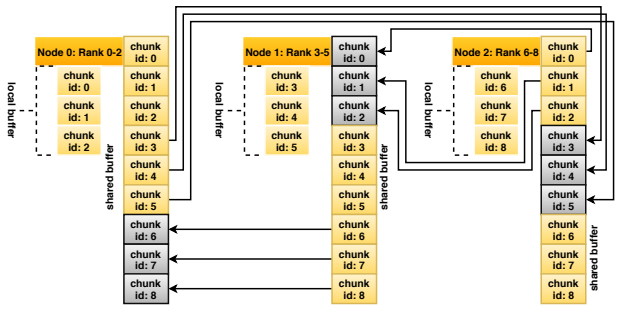
(a) Initial state of local and shared buffer



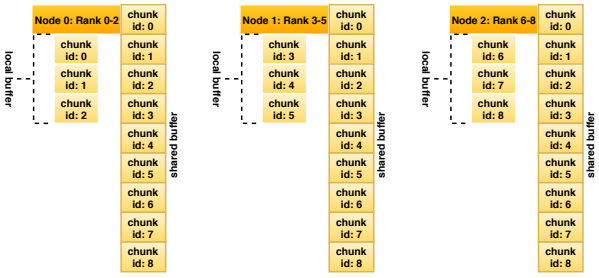
(b) State of shared buffer after local Allgather(V)



(c) Initialization of data movement



(d) Data receive and forward



(e) State of shared buffer after receiving all data

Fig. 2: Graphical representation of Allgather(V).

decreasingly (see Figure 2(g)), and broadcast data distributions (see Figure 2(h)), respectively.

III. Allreduce

In this section, we present the algorithmic solution for the Allreduce collective operation and demonstrate its performance results in comparison with the best performing MPI implementations.

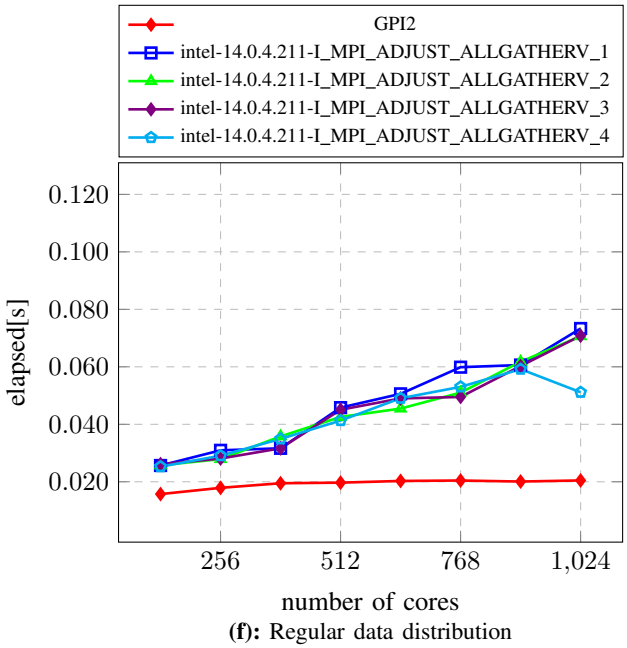
A. Algorithmic Solution

For small messages in Allreduce operations we have implemented a dissemination algorithm. In the regular dissemination algorithm, in each step $k, (0 \leq k < log_2(n)))$, process i sends a message to process $(i + 2^k)$ and receives a message from process $(i - 2^k)$, where we wrap in both directions. If the number of process n equals 2^m the process is complete after $m = log_2(n)$ steps. For nodes numbers $n \neq 2^m$, we need a modification of the above algorithm. To this we end we observe that we can use partial reductions to complete the missing information. For simplification we here consider the original state in step 0 as a partial result for step 0. We first decompose the number of nodes as $n = \sum_{k=l}^{0} c_k 2^k$, with $c_k \in \{0, 1\}$. We then complete the regular dissemination algorithm for $m = 2^l$. The last message we have send and received then will be $(i + 2^{(l-1)})$. We subsequently complement this result with the partial reductions from step $l > j \geq 0, c_j \neq 0$ which we send to node $(i + \sum_{k=l}^{j+1} c_k 2^k)$ and receive from $(i - \sum_{k=l}^{j+1} c_k 2^k)$ respectively. Again we here wrap around for both directions.

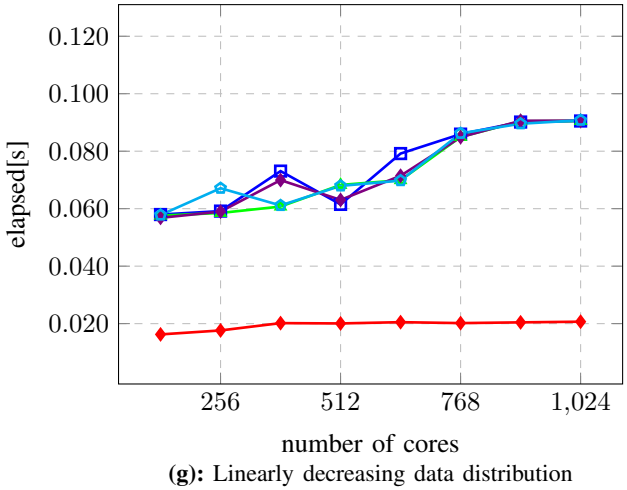
After performing all steps $l > j \geq 0, c_j \neq 0$ we have a complete result. While the algorithm requires additional buffer space of order $log_2(n) \times message\_size$, we believe this to be a minor issue as dissemination algorithms have non-optimal bandwidth term and hence are rather sub-optimal for large message size. We note that the algorithm is able to produce bit-identical results for subsequent runs, as all required partial reductions are node-locally available on the final target nodes. We also note that for large message sizes we have developed an algorithm based on notified communication in shared windows which uses pipelined rings in [10] [4].

We show the procedure for two cases, n = 7, n = 11. Colored boxes here correspond to partial reductions, with identical colors corresponding to the same reduction. We show the required sends for rank 0. While sends for identical partial reductions in 4, 5 occur sequentially (for sake of readability), the implemented algorithm sends these reductions in parallel to the ongoing dissemination.

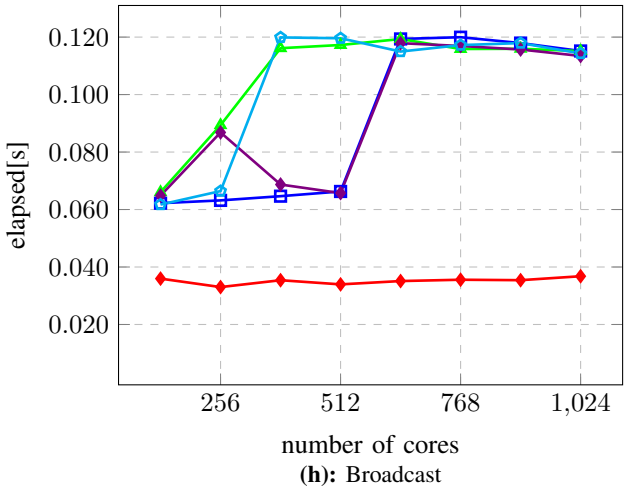
While the above algorithm already yields good scalability, it is possible to slightly improve this algorithm. We decompose the number of nodes in advance and determine the receivers of the (additional) partial reduction messages. When proceeding through the dissemination for $m = 2^l$ we can directly asynchronously send the partial results required for later stages. Assuming that the network is able to deliver these partial results out-of-band to the dissemination for $m = 2^l$, we can hide the corresponding communication in the ongoing



(f): Regular data distribution



(g): Linearly decreasing data distribution



(h): Broadcast

Fig. 3: Performance results of the Allgather(V) implementations.

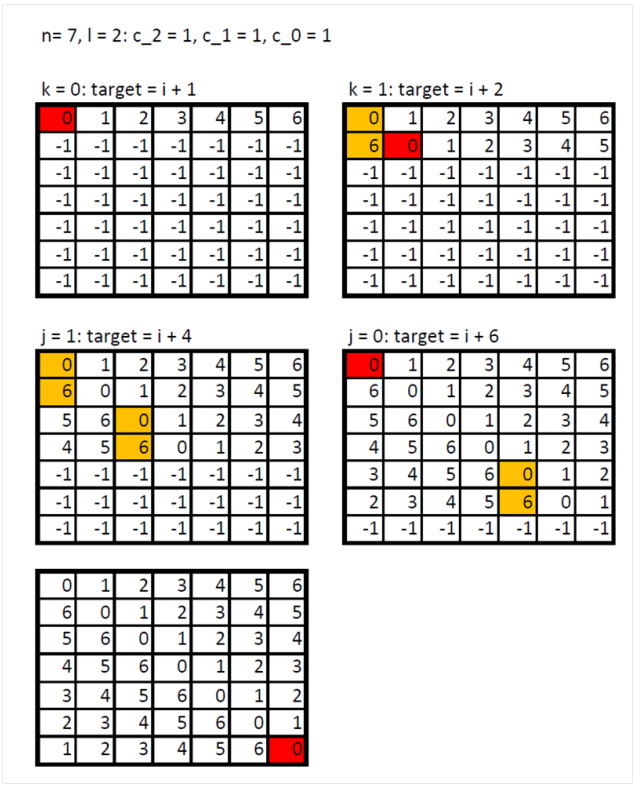


Fig. 4: Dissemination 7 nodes.

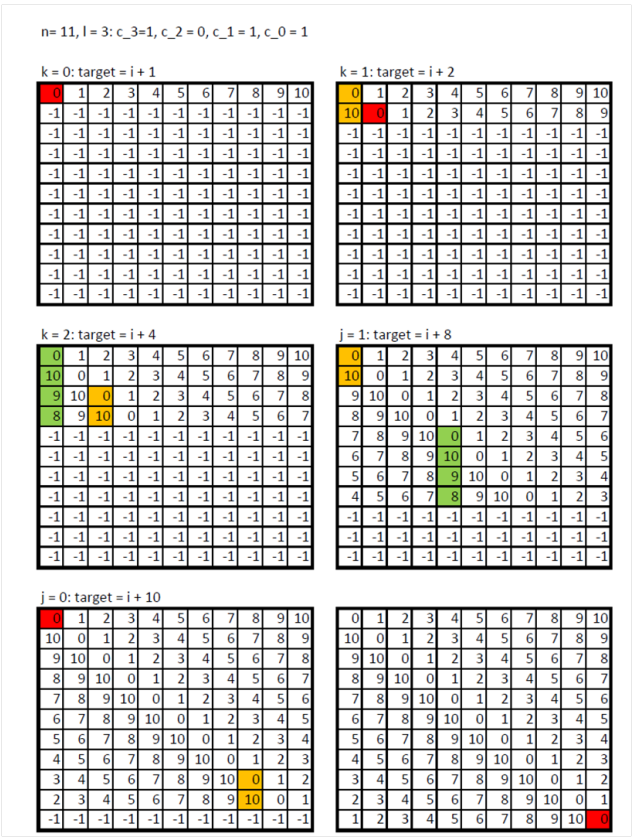


Fig. 5: Dissemination 11 nodes.

dissemination. For $n \neq 2^m$, this algorithm hence should scale with $log_2(l)$ - slightly faster than $log_2(m)$, assuming the above node decomposition).

Depending on message size we split the Allreduce load across the number of local ranks per node. Each rank then disseminates a fraction of the actual message size, given by $\frac{message\_size}{nProcLocal}$. After the initial corresponding node-local Allreduce for the respective fraction of the message, all ranks subsequently perform the above dissemination algorithm with ranks which share the same node-local rank id. The Allreduce collective is complete (for a single rank) if all partial results have been reduced for all required fractions of the message.

B. Performance Results

We have performed benchmarks for the implemented dissemination algorithm on $192 - 1920$ cores an Intel Ivy Bridge System with 12 cores per socket and an Infiniband FDR interconnect (Salomon cluster, IT4I). We have benchmarked against available implementations for Intel 5.1.2. Values for MPI_ADJUST_ALLREDUCE in the figures are: 1. Recursive doubling; 2. Rabenseifner's; 3. Reduce + Bcast; 4. Topology aware Reduce + Bcast; 5. Binomial gather + scatter; 6. Topology aware binominal gather + scatter; 7. Shumilin's ring; 8. Ring; 9. Knomial; 10. Topology aware SHM based flat; 11. Topology aware SHM based Knomial

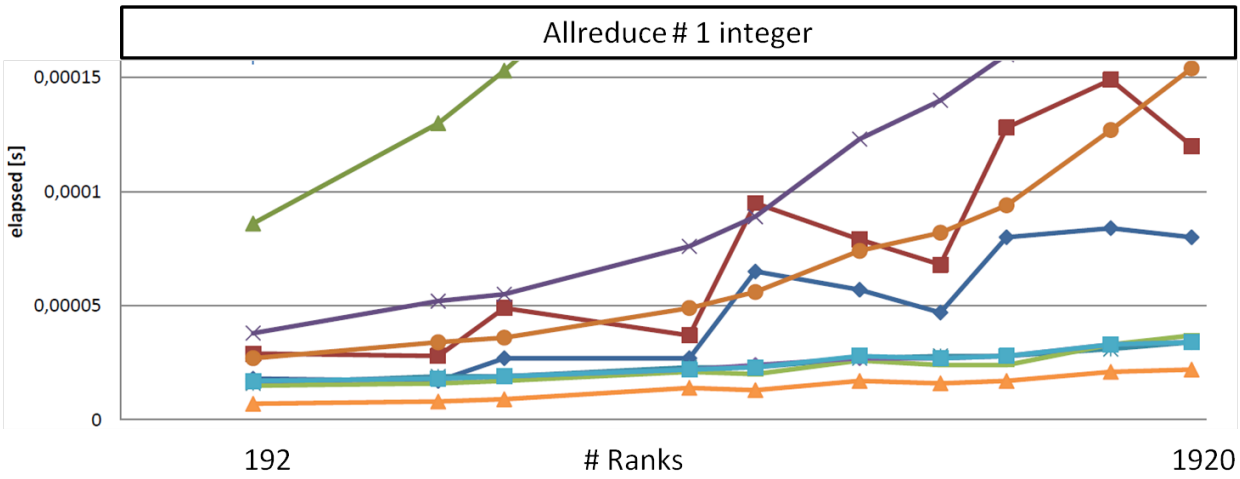
For the GPI2 implementation of GASPI (in combination with shared windows), we achieve a speedup factor of 1.5 (for large number of ranks) and 2.5 (for smaller number of ranks) vs. the best Intel MPI implementations (Intel MPI 5.1.2)

IV. Related Work

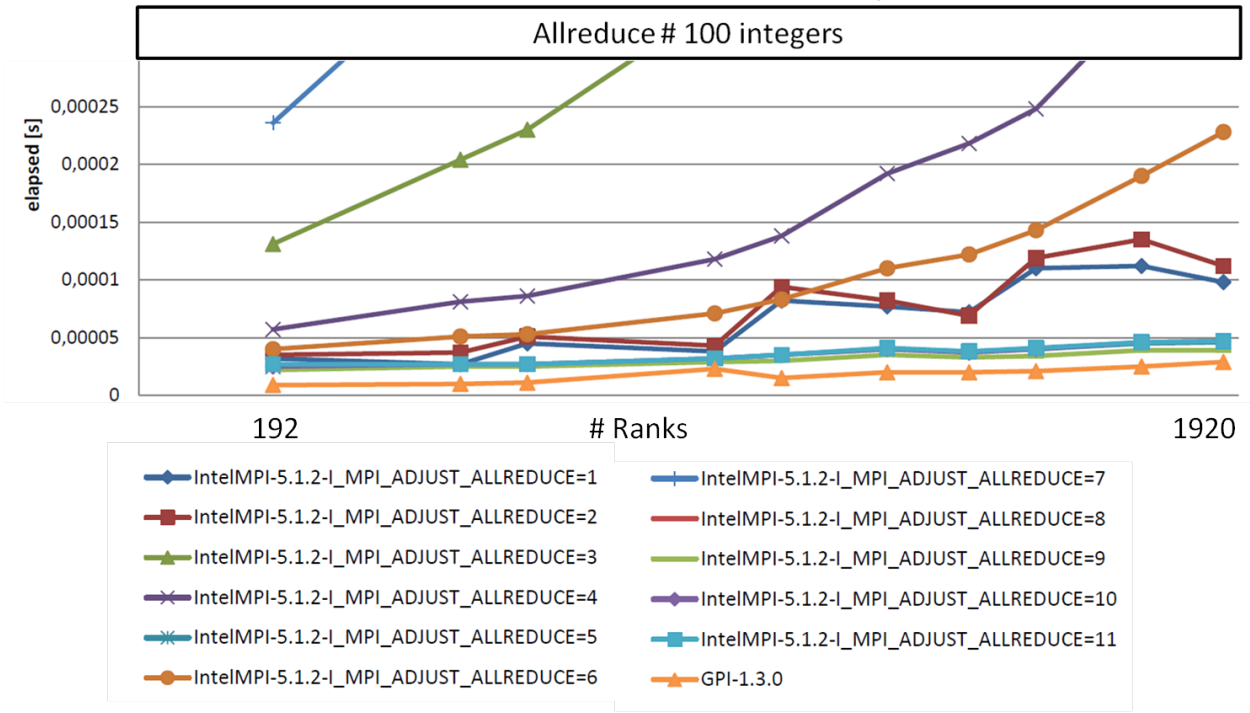
Depending on message sizes and network architecture, Allreduce implementations span a wide range of algorithms, from ring-based algorithms, binomial spanning-tree implementations [11], tree algorithms [12], or Butterfly like algorithms. [13]. In [14] the authors have designed an n-way dissemination algorithm for the GASPI API. This latter algorithm however neither leverages partial reductions of a 2-way dissemination (and their associated out-of-band delivery to late dissemination stages), nor notifed communication in shared windows.

We previously in [4], [10] designed algorithms based on pipelined rings and notified communication in shared windows. We were able to show that such an implementation can deliver roughly a 3x performance boost compared to the best Intel MPI implementations (5.1.2) on the Salomon IT4I cluster (Infiniband FDR).

We here make use of an adaption of the dissemination algorithm. The dissemination algorithm has been presented by Hensgen et al. in 1988 [15]. Due to its speed it is used in different programming APIs and libraries like the MPICH implementation of MPI for barrier implementations. In 2006 Torsten Hoefler et al. [16] have based the n-way dissemination algorithm on this work to exploit implicit parallelism of the InfiniBand network.



(a) Benchmark results of dissemination. 1 integer



(b) Benchmark results of dissemination. 100 integers

Fig. 6: Performance results of the Allreduce implementations.

V. Conclusions and Future Work

We have extended previous work on Notified Communication in Shared Windows for two important use cases, namely Allgather(V) and Allreduce. We are able to achieve high performance gains, relative to existing MPI implementations, both for the Cray Aries interconnect and Infiniband FDR.

While collective operations will very likely benefit the most from the concept of Notified Communication in Shared Windows we are currently evaluating whether an extension of this concept for message aggregation across multiple ranks will be beneficial. The key idea here is to aggregate the communication across multiple ranks which either share the same target rank or the same target window. For architectures with a high core count per node, this aggregation would both dramatically reduce the number of messages exchanged between remote nodes and also might be able to make more efficient use of network bandwidth with correspondingly aggregated larger messages in inter-node communication.

Another interesting collective is AlltoallV that involves all processes with variable data size. Currently, we are working on implementing efficient algorithmic solutions for this collectives using shared windows. Unlike Allgather(V), in case of AlltoallV, every process needs to communicate with one process from each node to transfer all node data contributions

for the target process. Moreover, in this implementation of AlltoallV, a process does not know anything about the receiving data size from other processes, hence the communication starts with sending the data size followed by waiting for remote offset for the data. Once the process receives the offset, it can send the associated data to the offset source process. Our current implementation based on the idea defined is not stable at this moment and hence we leave it for the future work.

ACKNOWLEDGEMENT

This work was funded by EU H2020 Research and Innovation programme through the INTERTWinE project (no. 671602). The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC Center for High Performance Computing.

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 Simulation Performance 2014*. Springer, 2015, pp. 17–32.

[2] R. Belli and T. Hoefler, "Notified access: Extending remote memory access programming models for producer-consumer synchronization," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 871–881.

[3] S. Markidis, I. B. Peng, J. L. Träff, A. Rougier, V. Bartsch, R. Machado, M. Rahn, A. Hart, D. Holmes, M. Bull *et al.*, "The epigram project: preparing parallel programming models for exascale," in *International Conference on High Performance Computing*. Springer, 2016, pp. 56–68.

[4] C. Simmendinger, R. Iakymchuk, L. Cebamanos, D. Akhmetova, V. Bartsch, T. Rotaru, M. Rahn, E. Laure, and S. Markidis, "Interoperability strategies for gaspi and mpi in large scale scientific applications," *IJHPCA*, accepted on June 29th, 2018. DOI of preprint: https://doi.org/10.5281/zenodo.1206292.

[5] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Mpi+ mpi: a new hybrid approach to parallel programming with mpi plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[6] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.

[7] I. B. Peng, S. Markidis, and E. Laure, "The cost of synchronizing imbalanced processes in message passing systems," in *2015 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2015, pp. 408–417.

[8] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, "Nek500 web-page," http://nek5000.mcs.anl.gov.

[9] J. l. Träff, A. Ripke, C. Siebert, P. Balaji, R. Thakur, and W. Group, "A simple, pipelined algorithm for large, irregular all-gather problems," in *Lastovetsky A., Kechadi T., Dongarra J. (eds) Recent Advances in Parallel Virtual Machine and Message Passing Interface. EuroPVM/MPI 2008. Lecture Notes in Computer Science*, vol. 5205. Springer, 2008, pp. 84–93.

[10] D. Akhmetova, L. Cebamanos, R. Iakymchuk, T. Rotaru, M. Rahn, S. Markidis, E. Laure, V. Bartsch, and C. Simmendinger, "Interoperability of gaspi and mpi in large scale scientific applications," in *Wyrzykowski R., Dongarra J., Deelman E., Karczewski K. (eds) Parallel Processing and Applied Mathematics. PPAM 2017. Lecture Notes in Computer Science*, vol. 10778. Springer, 2018, pp. 277–287.

[11] N.-F. Tzeng and H.-L. Chen, "Fast compaction in hypercubes," *IEEE Transactions on Parallel & Distributed Systems*, no. 1, pp. 50–56, 1998.

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

[13] E. D. Brooks, "The butterfly barrier," *International Journal of Parallel Programming*, vol. 15, no. 4, pp. 295–307, 1986.

[14] V. End, R. Yahyapour, C. Simmendinger, and T. Alrutz, "Adaption of the n-way dissemination algorithm for gaspi split-phase allreduce," *INFOCOMP 2015*, p. 25, 2015.

[15] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *International Journal of Parallel Programming*, vol. 17, no. 1, pp. 1–17, 1988.

[16] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, "Fast barrier synchronization for infiniband/spl trade," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 7–pp.