

Optimal, Non-pipelined Reduce-scatter and Allreduce Algorithms with an Application to All-to-all Communication

JESPER LARSSON TRÄFF, Faculty of Informatics, Institute of Computer Engineering, Parallel Computing, TU Wien, Vienna (Wien), Austria

The reduce-scatter collective operation in which p processors in a network of processors collectively reduce p input vectors into a result vector which is partitioned into p blocks over the processors is important both in its own right and as building block for other collective operations. We present a surprisingly simple, but non-trivial algorithm for solving this problem optimally in $\lceil \log_2 p \rceil$ communication rounds with each processor sending, receiving and reducing exactly $p - 1$ blocks of vector elements, provided that the binary reduction operator is commutative. We combine this with a similarly simple, well-known allgather algorithm to get a volume optimal algorithm for the allreduce collective operation where the result vector is replicated on all processors. The communication pattern is a simple, $\lceil \log_2 p \rceil$ -regular, circulant graph also used elsewhere. A simple requirement on the skip sequence of the circulant graph gives freedom in the choice of communication pattern which may be valuable for practical implementation. The algorithms can readily be implemented and used for the collective operations `MPI_Reduce_scatter_block`, `MPI_Reduce_scatter` and `MPI_Allreduce` as specified in the MPI standard. For completeness, we also give a $\lceil \log_2 p \rceil$ communication round optimal allreduce algorithm that can be used to implement `MPI_Allreduce` and is suited for smaller input vectors. Finally, we observe that the reduce-scatter algorithm can be used as a template for round-optimal all-to-all communication with smaller communication volume, for some p , than the well-known algorithm by Bruck et al. (1997) to implement the collective `MPI_Alltoall` operation. We provide experimental evidence on a small cluster that the algorithms can perform better than or comparably to other implemented MPI library algorithms.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; • **Computer systems organization** → **Interconnection architectures**; • **Theory of computation** → **Massively parallel algorithms**;

Additional Key Words and Phrases: Collective communication, reduction problems, reduce-scatter, allreduce, all-to-all, Message-Passing Interface (MPI)

ACM Reference Format:

Jesper Larsson Träff. 2025. Optimal, Non-pipelined Reduce-scatter and Allreduce Algorithms with an Application to All-to-all Communication. *ACM Trans. Parallel Comput.* 12, 4, Article 10 (December 2025), 23 pages. <https://doi.org/10.1145/3760789>

1 Introduction

Collective *combine* or *reduction operations* in which sets of processors (processes) cooperate to globally combine or reduce sets of input vectors and distribute the result in various ways across

Author's Contact Information: Jesper Larsson Träff, Faculty of Informatics, Institute of Computer Engineering, Parallel Computing, TU Wien, Vienna (Wien), Austria; e-mail: traff@par.tuwien.ac.at.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2329-4949/2025/12-ART10

<https://doi.org/10.1145/3760789>

the processors are important algorithmic building blocks for applications on large-scale, parallel computing systems: For recent adaptations and applications of mostly known algorithms, see, e.g., References [10, 16, 17].

Given p consecutively ranked processors r , $0 \leq r < p$ each with an input vector V_r , with the same number of elements for all processors, and an associative, binary operator $\oplus$ that can element-wise combine two vectors, the global *combine* or *reduction problem* is to compute $W = \oplus_{r=0}^{p-1} V_r$. By the associativity of $\oplus$, brackets can be left out. If the operator is also commutative, the order of the input vectors does not matter. The result vector W can be stored at either a designated root processor, at all processors, or be partitioned into p blocks of elements with block $W[r]$ of the result stored at processor r . Blocks may have the same or different numbers of vector elements.

With the **message-passing interface (MPI)** [15], these problems are solved by the collective operations MPI_Reduce (reduction to a designated root process), MPI_Allreduce (result vector replicated on all processes), MPI_Reduce_scatter_block (result vector partitioned into blocks having exactly the same number of elements) and MPI_Reduce_scatter (result vector partitioned into blocks of possibly different number of elements). These operations are also supported with other communication interfaces and libraries [32]. The commonly used term *reduce-scatter* for global reduction with partitioned result is unfortunate, since it suggests that the problem is solved in two stages as a reduction to a root processor followed by a scatter operation for partitioning the result vector. Good algorithms do not take this detour, but solve the problem directly. A better intuition is therefore to view the operation as p disjoint, simultaneous, rooted reductions with each processor r being the root in a reduction of the input blocks with index r . The reduce-scatter algorithm to be presented in the following follows this intuition, but the processors cooperate subtly in the reduction of the blocks. We will alternatively refer to the reduce-scatter operation as *disjoint all-reduce*.

Our algorithms work uniformly for any number of processors p . We assume the operator $\oplus$ to be commutative. The processors communicate in a $\lfloor \log_2 p \rfloor$ -regular circulant graph pattern where each processor has $\lfloor \log_2 p \rfloor$ incoming and $\lfloor \log_2 p \rfloor$ outgoing neighbor processors. Communication is assumed to be only one-ported (a processor can be engaged in one communication operation at a time), but to allow a processor to send a block to some processor and at the same time receive a block from some other processor [2, 9]. This *simultaneous send/receive* model corresponds to what the combined MPI_Sendrecv operation of MPI is meant to accomplish [15].

The reduce-scatter (disjoint all-reduce) and allreduce operations have been intensively researched and many algorithms and implementations, taking aspects of the communication system (different topologies, different, hierarchical organizations) into account have been proposed. A primary starting point for the algorithms of this article is the well known and often used power-of-two, straight doubling, circulant graph, dissemination allgather (concatenation, all-to-all broadcast, all-broadcast) algorithm by Bruck et al. [9], which in contrast to hypercube algorithms works for any number of processors p . The algorithms by Bar-Noy and others for allreduce (computing census functions) [3, 8] that use a cleverly adjusted doubling scheme have likewise been a reference point for subsequent algorithms and generalizations, both for the allreduce and for the reduce-scatter (disjoint all-reduce) operation [6, 29].

Well-known algorithms assuming either a ring or a fully connected communication network can solve the problem in $p - 1$ communication rounds, in which each processor sends and receives a partial block result $W[i]$ to and from a preceding and a succeeding processor and performs a reduction on the received partial result block, see for instance [11, 14, 18]. With a ring, the $\oplus$ operator must be commutative, whereas with a fully connected network, the algorithm can also work for non-commutative operators [14]. These algorithms are optimal in the number of blocks that are sent and received per processor, namely $p - 1$, but have a linear number of communication

rounds which is very unattractive for small block sizes and large numbers of processors p . The lower bound on the number of communication rounds is clearly $\lceil \log_2 p \rceil$, as is well known [9] and follows easily from an information dissemination argument.

The reduce-scatter problem can be solved in $\log_2 p$ communication rounds with the optimal number $p-1$ of sent, received and reduced partial result blocks with a $\log_2 p$ -dimensional hypercube or butterfly communication pattern. Likewise, the allreduce problem can be solved with twice as many communication rounds, partial result blocks and send-receive operations. A drawback of these algorithms is that they do not readily extend to arbitrary numbers of processors beyond only powers of two [31]. This problem has often been addressed and extensions that are better than the trivial reduction to the nearest power of two have been proposed and implemented [19, 24]. Hypercube or butterfly pattern algorithms with some care work also for non-commutative operators.

For the reduce (reduction to root) and allreduce operations, pipelined fixed-degree (binary) trees are also used. Such algorithms are not too difficult to implement and can work for any number of processors, sometimes also for non-commutative operators (depending on how trees are constructed and numbered), but have the disadvantage of losing effective bandwidth proportional to the arity of the trees. Likewise, there is a latency penalty proportional to the size of the pipeline block size, which can in addition be difficult to select well. Some of these problems can be alleviated by using two trees simultaneously [22].

A standard observation is that allreduce can be accomplished by performing a reduce-scatter (disjoint all-reduce) operation followed by an allgather (all-broadcast) operation. Lower bound arguments in References [4, 18] show that when the total number $p(p-1)$ of required applications of the $\oplus$ operator to blocks of elements are evenly shared by the processors, it is required to send and receive $2(p-1)$ partial result blocks per processor. At least $\lceil \log_2 p \rceil$ communication rounds are required. Using the reduce-scatter (disjoint all-reduce) and allgather (all-broadcast) algorithms of the present article, the bound on the number of blocks is achieved with $2\lceil \log_2 p \rceil$ communication rounds. Non-trivial, direct, uniform, one phase algorithms with $\lceil \log_2 p \rceil$ communication rounds which reach the lower bound on the number of rounds at the expense of a communication volume of $\lceil \log_2 p \rceil m$ for inputs of size m were given in Reference [3, 8, 29]. We include the algorithm from Reference [29] here to complement the combined reduce-scatter and allgather algorithm, since this algorithm [29] can perform better for small(er) input vectors. It furthermore has interesting applications, one of which (the *allgather-merge* collective operation) we sketch here. No algorithm that reaches both the lower bound on number of communication rounds and on communication volume at the same time is known.

We finally observe that all-to-all communication can be accomplished by a (commutative) reduce-scatter operation by taking concatenation as the operator, and present a corresponding algorithm. Surprisingly, for some p (that are not powers of two), this algorithm has lower communication volume (total number of blocks communicated) than the well-known, power-of-two distance doubling all-to-all algorithm by Bruck et al. [9] often implemented in MPI libraries [20, 23]. All algorithms presented here have been implemented in and for MPI [15] as written, and can perform better than or comparably to standard MPI libraries in many situations (system, MPI library, problem sizes, number and distribution of processes) as indicated by an experimental evaluation on a small high-performance cluster.

2 The Algorithms

We are given p consecutively ranked processors r , $0 \leq r < p$, each of which can in a communication step simultaneously send a block and receive a block from two other, possibly different processors. The reduce-scatter (disjoint all-reduce) and allreduce algorithms uniformly follow a communication pattern in which each processor r in a communication round k sends a block of elements to a

ALGORITHM 1: The p -block reduce-scatter (disjoint all-reduce) algorithm for processor r , $0 \leq r < p$. Each processor has an input vector V partitioned into p blocks of elements. Processor r eventually computes in W the complete reduction over the r th input blocks, that is, for processor r , $W_r = \bigoplus_{i=0}^{p-1} V_i[r]$, where $\oplus$ is the commutative operator for pairwise reduction of blocks.

```

procedure DISJOINTALLREDUCE( $V[p]$ ,  $W$ ,  $\oplus$ )
   $W \leftarrow V[r]$ 
  for all  $i = 1, \dots, p-1$  do  $R[i] \leftarrow V[(r+i) \bmod p]$ 
  end for ▷  $R[0]$  will be kept in  $W$ 
   $s \leftarrow p$ 
  while  $s > 1$  do
     $s', s \leftarrow s, \lceil s/2 \rceil$  ▷ Halve and round up
     $t, f \leftarrow (r+s) \bmod p, (r-s+p) \bmod p$  ▷ To- and from processors
    Send( $R[s \dots s'-1], t$ ) || Recv( $T[0 \dots s'-1-s], f$ )
     $W \leftarrow W \oplus T[0]$ 
    for all  $i = 1, \dots, s'-s-1$  do  $R[i] \leftarrow R[i] \oplus T[i]$ 
    end for
  end while
end procedure

```

to-processor $(r+s_k) \bmod p$ and receives a block of elements from a from-processor $(r-s_k+p) \bmod p$ for certain skips (or jumps) s_k . A graph $C_p^{s_0, s_1, \dots, s_{q-1}}$ with vertices $r, 0 \leq r < p$ and directed edges $(r \pm s_k + p) \bmod p$ is called a *circulant graph* with *skips* (jumps) $s_k, k = 0, \dots, q-1$ (sometimes a *loop network*, see Reference [5]). For the following algorithms, the skips s_k are at first chosen by repeated halving of $p = s_q$ with rounding up until $s_0 = 1$, that is $s_k = \lceil s_{k+1}/2 \rceil$ for $k = q-1, \dots, 0$. The number of such roughly halving steps required is clearly $q = \lceil \log_2 p \rceil$.

2.1 A Simple, Uniform Reduce-Scatter Algorithm

For the reduce-scatter problem, each processor takes an input vector V_r of elements and each V_r is partitioned in the same way into p disjoint blocks of elements such that each processor has p input blocks, each with a given, known number of elements. The number of elements per block may be the same (as in the MPI_Reduce_scatter_block operation) or may be different (as in the MPI_Reduce_scatter operation). The input blocks for processor $r, 0 \leq r < p$ are indexed as $V_r[i], 0 \leq i < p$. The reduce-scatter operation computes for each processor r the complete sum (under $\oplus$)

$$W_r = \bigoplus_{i=0}^{p-1} V_i[r]$$

of the r th block over all inputs. A partial result block is a sum of input blocks for a subset of processors. Per specification of the problem, each processor initially has a partial result for all p blocks (namely, each of the input blocks). A reduce-scatter (disjoint all-reduce) algorithm will gradually combine partial result blocks, until in the end, each processor r will have its complete sum over all r th input blocks.

Algorithm 1 shows such a complete reduce-scatter (disjoint all-reduce) algorithm for processor r . The processors each perform the same number of communication rounds, starting with $s = p$ and roughly halving s with rounding up until $s = 1$. In each round, the processors communicate simultaneously with their to- and from-processors in the circulant graph, namely, for processor r , $(r+s) \bmod p$ and $(r-s+p) \bmod p$. Each processor maintains partial result blocks $R[i]$ for some i that will contribute towards the final result W both for processor r itself and for other processors ranked after r (modulo p). The partial result blocks are maintained in the same way for all processors

such that for processor r , $R[i]$, $0 \leq i < p$ is a partial result block that will contribute to the final result at processor $(r + i) \bmod p$. This is achieved initially by a rotated copy of the input blocks $V[(r + i) \bmod p]$ into $R[i]$. We use **for all** to indicate that this can be done (in parallel) in any order.

In each communication round where s is roughly halved, with s' being the previous value of s , a consecutive sequence of partial result blocks $R[s \dots s' - 1]$ is sent to the to-processor $(r + s) \bmod p$ and a consecutive sequence with the same number of blocks is received from the from-processor $(r - s + p) \bmod p$ and summed into $R[0 \dots s' - 1 - s]$ using the $\oplus$ operator. Thus, for each i , $1 \leq i < p$, the partial result for block $R[i]$ is sent once as part of a consecutive sequence of blocks. Block $R[0]$ is kept as W and will eventually store the result of the complete reduction for block r (to save a final copy of $R[0]$ into W).

THEOREM 1. *On p input vectors partitioned into p blocks, Algorithm 1 solves the reduce-scatter (disjoint all-reduce) problem in $\lceil \log_2 p \rceil$ simultaneous send-receive communication rounds. Each processor sends and receives exactly $p - 1$ partial result blocks of elements and performs exactly $p - 1$ applications of the commutative reduction operator $\oplus$ on partial result blocks.*

PROOF. The communication round complexity is obvious, since $q = \lceil \log_2 p \rceil$ roughly halving steps are needed for the **while**-loop to terminate. Let s_{k+1} be the value of s before the k th round, $k = q - 1, q - 2, \dots, 1$ (numbered downwards from $q - 1$). Starting with $s_q = p$, by the repeated halving, clearly $s_q = p > s_{q-1} > \dots > s_1 > s_0 = 1$. In round k , each processor sends and receives $s' - s = s_{k+1} - s_k$ partial result blocks and applies the $\oplus$ operator to this number of blocks. Since $\sum_{k=q-1}^0 (s_{k+1} - s_k) = s_q - s_0 = p - 1$, the bounds on the communication and computation volume follow.

The algorithm maintains for each processor r the invariant that for $0 \leq i < s_k$, $R[i]$ (with $W = R[0]$) stores a partial result block over a subtree T_i of processors rooted at i with subtrees T_i and T_j being disjoint for $i \neq j$, but spanning all i , $0 \leq i < p$. In other words, each processor r maintains a spanning forest over all i , $0 \leq i < p$. The invariant holds before the first iteration of the **while**-loop since initially each T_i is a singleton storing the input block $R[i] = V[(r + i) \bmod p]$. After the last iteration where $s_0 = 1$ the invariant implies that for each processor r , $R[0] = W$ is the result of a reduction over a spanning tree T_0 of the blocks $V_{(r-i+p) \bmod p}[r]$.

In round k , subtrees T_j represented by $R[j]$, $s_k \leq j < s_{k+1}$ are hooked into subtrees T_i , $0 \leq i < s_{k+1} - s_k$ with $j = i + s_k$, $T_{i+s_k} \xrightarrow{s_k} T_i$, by an edge labeled with the skip s_k , and the partial results represented by the subtrees $R[i]$, $0 \leq i < s_k - s_{k+1}$ are combined with the partial results $R[j]$ sent from processor $(r - s_k + p) \bmod p$. The T_i trees indeed remain trees (no cycles introduced), since $s_{k+1} - s_k = s' - s \leq s = s_k$ by the repeated halving scheme (either $s_{k+1} = s_k + s_k$ or $s_{k+1} + 1 = s_k + s_k$).

Partial results $R[j]$, $1 \leq j < p$ are sent once, and $W = R[0]$ never. Thus, the invariant is maintained, and in each round, the number of disjoint subtrees decreases by $s' - s$. $\square$

An indirect proof of the existence of an algorithm with the properties of Theorem 1 follows from the observations in [27, Section 2.1] which was the incentive for the much simpler, fully explicit Algorithm 1.

Example: It is illustrative to trace how Algorithm 1 builds the reduction tree for the r th input blocks. For instance, for $p = 22$, the roughly halving skips are $s = 11, 6, 3, 2, 1$. In each communication round, a tree T_{i+s} is hooked into tree T_i for $0 \leq i < s' - s$ by an edge corresponding to skip s . When the algorithm terminates with $s = 1$, a tree as depicted in Figure 1 has been constructed (implicitly) by each processor. Now, for a given processor, say $r = p - 1 = 21$, the order in which input blocks from the other processors are reduced can be found by a depth-first traversal of this tree, subtracting the node label i from r (modulo p) to get the index of the contributed

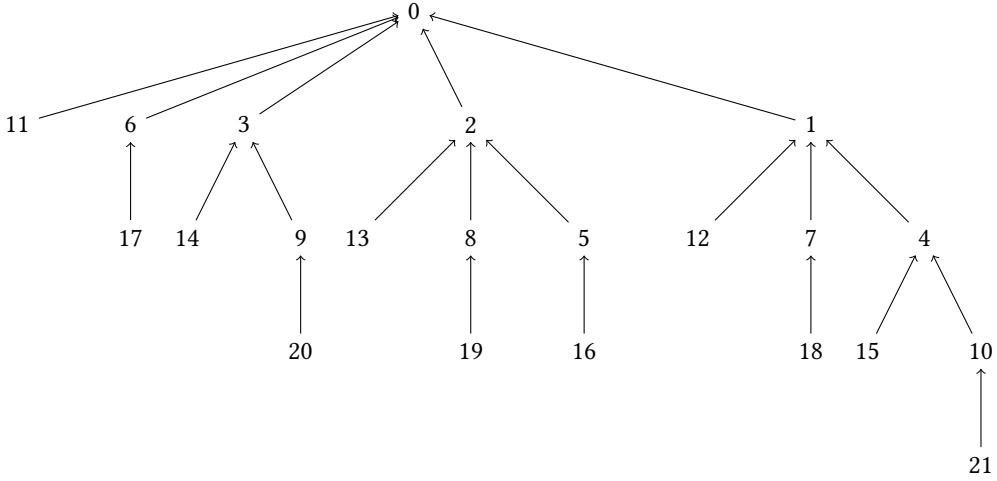


Fig. 1. The reduction tree implicitly constructed by each processor by Algorithm 1 for $p = 22$. The total path length for this tree is $P = 45$: There are 5 processors at distance 1, 9 processors at distance 2, 6 processors at distance 3 and one processor at distance 4.

input block $V_{(r-i+p) \bmod p}[r]$. Processor $r = 21$ therefore receives partial results from processor $21 - 11 = 10, 21 - 6 = 15, 21 - 3 = 18, 21 - 2 = 19$ and finally $21 - 1 = 20$. Let x_i denote the input block of processor i , $x_i = V_i[r]$. Processor $r = 21$ then computes

$$\begin{aligned}
 W &= x_{21} \oplus \\
 &\quad x_{10} \oplus \\
 &\quad (x_{15} \oplus x_4) \oplus \\
 &\quad (x_{18} \oplus x_7 \oplus (x_{12} \oplus x_1)) \oplus \\
 &\quad (x_{19} \oplus x_8 \oplus (x_{13} \oplus x_2) \oplus (x_{16} \oplus x_5)) \oplus \\
 &\quad (x_{20} \oplus x_9 \oplus (x_{14} \oplus x_3) \oplus (x_{17} \oplus x_6 \oplus (x_{11} \oplus x_0))) \\
 &= \bigoplus_{i=0}^{p-1} x_i
 \end{aligned}$$

where each line shows the received partial sums in the five communication rounds. The last equality holds by commutativity of $\oplus$.

If we assign time costs to the simultaneous send-receive operations, the total time of the algorithm for p processors and vectors of m elements (and blocks of size m/p) can be estimated.

COROLLARY 2. *In a homogeneous, linear-affine transmission cost model where simultaneous sending and receiving of blocks of m' elements by all processors in a communication round is charged $\alpha + \beta m'$ time units and pairwise reduction of two m' -element vectors by the binary $\oplus$ operator is charged $\gamma m'$ time units, the reduce-scatter (disjoint all-reduce) problem on input vectors of m elements uniformly partitioned into blocks of m/p elements is solved in time*

$$T(m, p) = \alpha \lceil \log_2 p \rceil + \beta \frac{p-1}{p} m + \gamma \frac{p-1}{p} m$$

by Algorithm 1. The time for the initial copy of the m input elements into $R[i]$ adds another term of at most γm .

The proof of Theorem 1 did not use any particular properties of the roughly halving scheme for computing s except that $s' - s \leq s$. Algorithm 1 can therefore work correctly for any sequence of $q \geq \lceil \log_2 p \rceil$ skips $s_k, k = 0, 1, \dots, q-1$ (with $s_q = p$ and $s_0 = 1$) satisfying

$$s_{k+1} - s_k \leq s_k \quad (1)$$

This condition guarantees that any $i, 0 \leq i < p$ can be written as a sum of different skips s_k which means that there is a path from any processor i to processor r consisting of at most q edges corresponding to different skips.

LEMMA 1. *Let $s_k, k = 0, 1, \dots, q-1$ be a sequence of increasing skips with $s_q = p$ and satisfying Requirement (1). Then any $i, 0 \leq i < p$ can be written as a sum of at most q different skips.*

PROOF. The proof is by induction on the sequence of skips. Assume the claim holds for any i with $0 \leq i < s_k$. Consider an i with $0 \leq i < s_{k+1}$. If $0 \leq i < s_k$, the claim holds by assumption. Consider therefore an i with $s_k \leq i < s_{k+1}$. If $i = s_k$, there is again nothing to prove, the claim holds with skip s_k . Otherwise, $0 < i - s_k < s_{k+1} - s_k \leq s_k$ by Requirement (1) and the claim follows by the induction hypothesis with s_k as the last skip. $\square$

COROLLARY 3. *The reduce-scatter problem can be solved correctly with optimal communication volume of $p-1$ partial result blocks in q communication rounds on any circulant graph $C_p^{s_0, s_1, \dots, s_{q-1}}$ with skips $1 = s_0 < s_1 < \dots < s_{q-1} < s_q = p$ satisfying Requirement (1).*

The flexibility in the specific choice of skip sequence for Algorithm 1 can provide interesting potential for tuning. Different circulant graphs may be more or less suited to be embedded into the concrete communication network partition. The difference between consecutive skips determines the maximum message size communicated, which can be controlled by the choice of skip sequence. Finding the best skip sequence in practice in a(ny) concrete setting on a given system is a challenging, experimental question.

Examples: The reduce-scatter (disjoint all-reduce) problem is solved on a fully connected network in $p-1$ communication steps by taking $s_k = 1, 2, \dots, p-1$. In this case, the algorithm can easily be made to work also for non-commutative operators and corresponds to the folklore algorithm also stated in Reference [14]. A straight power-of-two doubling scheme with $s_k = 2^k, k = 0, 1, \dots, q-1$ as used by Bruck et al. [9] will lead to another $\lceil \log_2 p \rceil$ round algorithm. Instead of halving, we get algorithms with $O(\log p)$ communication rounds by taking $s_k = f s_{k+1}$ by some constant factor $1/2 < f < 1$ in each round. We can get an algorithm running in a square root of p number of rounds by doubling s_k until $s_k \geq \lceil \sqrt{p} \rceil$ and then incrementing the skips by $\lceil \sqrt{p} \rceil$ as long as $s_k < p$. The number of blocks per round could be upper bounded by a constant by taking $s_{k+1} - s_k = c$ as long as $s_k > c$. And so on.

Algorithm 1 does not make any assumptions on the way input and result vectors are partitioned into disjoint blocks of elements, except for requiring that all input vectors are partitioned in the same way by all processors. Thus, the number of elements in block $V_r[i]$ and block $V_r[j]$ may differ, but the algorithm will work correctly and meaningfully as long as the number of elements in $V_i[r]$ is equal to the number of elements in $V_j[r]$ for all r . Let m be the total number of elements over all blocks $V_r[i]$ (which is the same for all processors r). In the extreme case, all elements are concentrated in one block only, and partial results of all m elements will be sent and/or received and reduced in every communication round. This leads to the following observation.

COROLLARY 4. *On p input vectors of m elements partitioned into p blocks of possibly different numbers of elements, Algorithm 1 solves the reduce-scatter (disjoint all-reduce) problem in time at*

most $\lceil \log_2 p \rceil (\alpha + \beta m + \gamma m)$, assuming a homogeneous, linear-affine transmission cost model with constant latency α and transmission and computation cost per unit β, γ , respectively.

Algorithm 1 can therefore sensibly be used also for the MPI_Reduce_scatter operation as long as the sizes of the input blocks do not differ too much, and in the extreme case of only one block, also for MPI_Reduce (reduction to root) as long as the number of elements is not too large compared with p . For large, irregular reduce-scatter problems where the sizes of the blocks for the processors can differ significantly, pipelined algorithms, also using a circulant graph communication pattern, can perform much better, depending only linearly on the total problem size m , see Reference [27].

As the example showed, the applications of $\oplus$ are not done in rank order, and therefore exploits the commutativity of the $\oplus$ operator. However, all processors perform the reductions in the same relative order by using the same reduction tree whose structure is determined by the skip sequence. It is therefore possible to permute the input blocks over the processors in advance to ensure that the result block for each processor is reduced in a specific, required order. Unfortunately, this entails an expensive all-to-all (p)redistribution step, and will therefore only make sense for very small problems.

2.2 An Allreduce Algorithm for Larger Problems

It is well-known and easy to see that the allreduce problem can be solved by a reduce-scatter (disjoint all-reduce) operation followed by an allgather (all-broadcast) operation that gathers together all result blocks at all processors (equivalently, broadcasts a block from all processors simultaneously). An allgather operation can, as classically shown in Reference [9], easily be implemented by a doubling scheme, essentially the reduce-scatter algorithm, Algorithm 1 run in reverse. A variant that reuses the skip sequence of the reduce-scatter algorithm is shown as Algorithm 2. The skip sequence $s_k, k = 0, 1, \dots, s_{q-1}$ must fulfill Requirement (1).

THEOREM 5. *On p input vectors partitioned into p blocks, Algorithm 2 solves the allreduce problem in $2\lceil \log_2 p \rceil$ send-receive communication rounds using a circulant graph with $q = \lceil \log_2 p \rceil$ skips. Each processor sends and receives exactly $2(p-1)$ blocks of elements and performs exactly $p-1$ applications of the commutative reduction operator $\oplus$ on blocks of elements.*

The bound on the number of blocks communicated and the number of reductions is optimal, see the arguments in References [4, 18]. The number of communication rounds is a factor of 2 off from optimal; the lower bound is $\lceil \log_2 p \rceil$ communication rounds.

COROLLARY 6. *In a homogeneous, linear-affine transmission cost model where simultaneous, bidirectional sending and receiving of blocks of m' elements by all processors in a communication round can be charged $\alpha + \beta m'$ time, and pairwise reduction of two m' -element vectors by the binary $\oplus$ operator takes time $\gamma m'$, the allreduce problem on input vectors of m elements is solved in time*

$$T(m, p) = 2\alpha \lceil \log_2 p \rceil + (2\beta + \gamma) \frac{p-1}{p} m$$

by Algorithm 2 by dividing the input into blocks of (roughly) m/p elements. The initial and final copy to and from buffer R adds a term of at most $2\gamma m$.

2.3 An Allreduce Algorithm for Small Problems

Algorithm 2 solves the allreduce problem optimally with respect to communication volume, but takes $2q$ communication rounds for any q skip circulant graph fulfilling Requirement (1). This is undesirable for small problems where the communication latency often determines the overall running time. A natural approach for achieving the optimal $q = \lceil \log_2 p \rceil$ rounds would aim at

ALGORITHM 2: The allreduce algorithm for processor r , $0 \leq r < p$ derived from the reduce-scatter algorithm by addition of an allgather phase that collects all result blocks on all processors. Each processor has an input vector V that can be partitioned into p blocks of elements. Each processor r returns in W the resulting reduction over all input vectors. The commutative operator for pairwise reduction of blocks is $\oplus$. A skip sequence $s_k, k = 0, 1, \dots, q-1$ of $q \geq \lceil \log_2 p \rceil$ skips fulfilling Requirement (1) is given or computed on the fly.

```

procedure ALLREDUCE( $V, W, \oplus$ )
    ▷ Assume  $V, W$  both partitioned into  $p$  blocks  $V[i], W[i]$ 
    for all  $i = 0, \dots, p-1$  do  $R[i] \leftarrow V[(r+i) \bmod p]$ 
    end for
    for  $k = q-1, \dots, 0$  do ▷ Disjoint all-reduce phase
         $t, f \leftarrow (r + s_k) \bmod p, (r - s_k + p) \bmod p$ 
         $\text{Send}(R[s_k \dots s_{k+1} - 1], t) \parallel \text{Recv}(T[0 \dots s_{k+1} - 1 - s_k], f)$ 
        for all  $i = 0, \dots, s_{k+1} - s_k$  do  $R[i] \leftarrow R[i] \oplus T[i]$ 
        end for
    end for
    for  $k = 0, 1, \dots, q-1$  do ▷ Allgather phase
         $f, t \leftarrow (r + s_k) \bmod p, (r - s_k + p) \bmod p$ 
         $\text{Send}(R[0 \dots s_{k+1} - 1 - s_k], t) \parallel \text{Recv}(R[s_k \dots s_{k+1} - 1], f)$ 
    end for
    for all  $i = 0, \dots, p-1$  do  $W[(r+i) \bmod p] \leftarrow R[i]$ 
    end for
end procedure

```

maintaining the invariant that after round $k, k = 0, 1, \dots, q-1$, processor r has computed a sum (under $\oplus$) of s_{k+1} successive input vectors (modulo p) starting from processor r itself, and use either a straight doubling or a roughly halving circulant graph with $\lceil \log_2 p \rceil$ skips. This, unfortunately, looks impossible. For a roughly halving circulant graph, $s_k + s_k = s_{k+1} + 1 > s_{k+1}$ when s_{k+1} is odd, so combining the partial sums from processor r and processor $(r + s_k) \bmod p$ would lead to a partial sum with one summand too much. With a straight doubling circulant graph, also $s_{q-1} + s_{q-1} > s_q = p$ when p is not a power of two, so also here the final partial sum would include some (many) input vectors twice (this could work for idempotent operators [29]).

The idea can be rescued by using a slightly adjusted, roughly halving circulant graph as communication structure. Let V_r be the input for processor r and $\oplus$ the commutative operator. Instead of the above, we try to maintain the invariant that

$$W = \bigoplus_{i=1}^{s_{k+1}-1} V_{(r+i) \bmod p}$$

after round k , that is, each processor maintains a sum of $s_{k+1} - 1$ successive input vectors (under $\oplus$) as partial result, *excluding* the contribution from processor r itself, a decisive idea which can be attributed to [3, 8]. With this, after the last round $k = q-1$, the final result for each processor r can be computed easily and correctly as $V_r \oplus W$. The invariant also holds, formally, before the first round by initializing W to the neutral element $\perp$ under $\oplus$. Or better, by simply not storing anything in W (a neutral element may not exist or be known). In the first round $k = 0$, each processor r would then have to receive the input $V_{(r+1) \bmod p}$ from processor $(r+1) \bmod p$ directly into W . This is correct since $s_1 = 2$ for any roughly halving circulant graph.

Now, for the rounds $k = 1, \dots, q-1$, we can maintain the invariant by the following observations.

- If $s_{k+1} = s_k + s_k$ (i.e., s_{k+1} is even), processor r can receive the partial result $V_{(r+s_k) \bmod p} \oplus (\bigoplus_{i=1}^{s_k-1} V_{(r+s_k+i) \bmod p})$ from processor $(r + s_k) \bmod p$, computed by this processor before

ALGORITHM 3: The allreduce algorithm for processor r , $0 \leq r < p$ with adjusted roughly halving skips s_k , $k = 0, 1, \dots, q - 1$. The invariant is that after round k , $W = \bigoplus_{i=1}^{s_{k+1}-1} V_{(r+i) \bmod p}$.

```

procedure ALLREDUCE( $V, W, \oplus$ )
  if  $p = 1$  then                                     ▷ For good measure, handle  $p = 1$  case
     $W \leftarrow V$ 
    return
  end if                                              ▷ Now  $p > 1$ 
   $t, f \leftarrow (r - s_0 + p) \bmod p, (r + s_0) \bmod p$    ▷ To- and from-processors for first round
  Send( $V, t$ ) || Recv( $W, f$ )                          ▷ First communication round,  $p > 1$ 
  for  $k = 1, \dots, q - 1$  do
     $\varepsilon \leftarrow s_{k+1} \wedge (1)_2$                        ▷ Adjustment  $\varepsilon = 1$  if  $s_{k+1}$  is odd,  $\varepsilon = 0$  if  $s_{k+1}$  is even
     $t, f \leftarrow (r - s_k + \varepsilon + p) \bmod p, (r + s_k - \varepsilon) \bmod p$    ▷ To- and from-processors
    if  $\varepsilon = 1$  then                                   ▷  $s_{k+1}$  odd
      Send( $W, t$ ) || Recv( $T, f$ )
    else                                                ▷  $s_{k+1}$  even
       $W' \leftarrow V \oplus W$ 
      Send( $W', t$ ) || Recv( $T, f$ )
    end if
     $W \leftarrow W \oplus T$ 
  end for
   $W \leftarrow V \oplus W$ 
end procedure

```

sending, and compute

$$\begin{aligned}
 W &= (\bigoplus_{i=1}^{s_k-1} V_{(r+i) \bmod p}) \oplus V_{(r+s_k) \bmod p} \oplus (\bigoplus_{i=1}^{s_{k+1}-1} V_{(r+s_k+i) \bmod p}) \\
 &= (\bigoplus_{i=1}^{s_k-1} V_{(r+i) \bmod p}) \oplus (\bigoplus_{i=0}^{s_{k+1}-1} V_{(r+s_k+i) \bmod p}) \\
 &= \bigoplus_{i=1}^{s_{k+1}-1} V_{(r+i) \bmod p} \quad .
 \end{aligned}$$

– If $s_{k+1} + 1 = s_k + s_k$ and therefore $s_{k+1} = s_k + s_k - 1$ (i.e., s_{k+1} is odd), processor r can instead receive the invariant partial result $\bigoplus_{i=1}^{s_{k+1}-1} V_{(r+s_k+i) \bmod p}$ from processor $(r + s_k - 1) \bmod p$, and compute

$$\begin{aligned}
 W &= (\bigoplus_{i=1}^{s_k-1} V_{(r+i) \bmod p}) \oplus (\bigoplus_{i=1}^{s_{k+1}-1} V_{(r+s_k-1+i) \bmod p}) \\
 &= (\bigoplus_{i=1}^{s_k-1} V_{(r+i) \bmod p}) \oplus (\bigoplus_{i=0}^{s_{k+1}-2} V_{(r+s_k+i) \bmod p}) \\
 &= \bigoplus_{i=1}^{s_{k+1}-1} V_{(r+i) \bmod p} \quad .
 \end{aligned}$$

In both cases, the invariant is maintained.

An explicit algorithm, originally given in Reference [29], is shown as Algorithm 3. The roughly halving skips s_k are adjusted subtracting the ε correction term which is taken as the parity of skip s_{k+1} .

We summarize the explanation in the following theorem.

THEOREM 7. *The allreduce problem is solved in $q = \lceil \log_2 p \rceil$ send-receive communication rounds by Algorithm 3 using an adjusted, roughly halving skip circulant graph. The number of reduction operations is at most $2(q - 1) + 1 = 2q - 1$, namely q plus the number of times s_{k+1} is even.*

COROLLARY 8. *In a homogeneous, linear-affine transmission cost model with constant latency α and transmission and computation cost per unit β, γ , respectively, the allreduce problem is solved*

in time

$$T(m, p) \leq \lceil \log_2 p \rceil (\alpha + \beta m) + (2\lceil \log_2 p \rceil - 1)(\gamma m)$$

by Algorithm 3.

Algorithm 3 has the drawback that sometimes more than q applications of $\oplus$ are performed (but only up to $2q - 1$). Depending on p , the number of extra applications beyond q can sometimes be reduced. First, note that if all s_k (except $s_1 = 2$) are odd, there are no extra applications. This happens when $p = 2^{q-1} + 1$. If from some s_k , all following, larger skips are even, V_r can be summed into W once and the originally intended invariant $W = \bigoplus_{i=0}^{s_{k+1}-1} V_{(r+i) \bmod p}$ maintained for the remaining rounds. From this s_k on, there are no extra applications of $\oplus$. In particular, if $p = 2^q$, exactly q applications are performed with this modification.

Algorithm 3 has an interesting application to the so called *allgather-merge* collective operation in which each processor contributes an input block of elements in sorted order and the output for each processor is the concatenation of all input blocks in sorted order (p -way merge of the input blocks). The operation was used as a building block for sorting algorithms in Reference [1], but is of independent interest, even for the case where each processor provides only one input element [21]. It is easy to see that Algorithm 3 can be used for the allgather-merge operation by taking $\oplus$ to be a two-way merge operation. The merge time for round k is $O(s_{k+1} \frac{m}{p})$ for input blocks of roughly m/p elements for a total of $O(m)$ over all rounds. This is asymptotically better than the obvious solution of an allgather followed by a p -way merge which takes $O(m \log p)$ time steps. Algorithm 3 extends the hypercube solution described in [1] to any number of processors.

3 An Application to All-to-All Communication

Reduction and disjoint all-reduction can solve the problems of gathering individual data blocks from each processor to a single root processor, and of gathering individual data blocks from each processor to each of the processors, namely by taking the $\oplus$ operator as concatenation of two sequences of blocks, and adapting the corresponding reduction algorithm. In other words, a reduce-scatter (disjoint all-reduce) algorithm can in principle be specialized to an algorithm for all-to-all communication. Concatenation is not commutative, but since blocks will be kept disjoint in some order, this at the most necessitates processor local reordering of blocks to get them into a desired, prescribed order. Also, since blocks need to be kept disjoint (no reduction into single blocks), extra space must be carefully managed, also to avoid as much as possible costly rearrangement of blocks by the processors. With these observations in mind, we now specialize Algorithm 1 to an algorithm for all-to-all communication. The resulting algorithm is shown as Algorithm 4 (and further improved into Algorithm 5 shown in Appendix A), and can readily be used to implement the MPI_Alltoall collective operation for blocks that all have the same number of elements. By further specialization, which we do not show here, namely by taking all blocks except the block for the root processor to have no elements, an algorithm for the MPI_Gather operation can also be derived. The input S for each processor r is divided into p blocks such that $S[i]$ is the block from processor r to processor i , $0 \leq i < p$. For processor r , the resulting block from processor i , $0 \leq i < p$ is stored in $R[i]$.

Algorithm 4 follows the same circulant graph communication pattern as Algorithm 1, determined by a skip sequence s_k fulfilling Requirement (1). It goes through q communication rounds, which is the optimal lower bound on the number of communication rounds when $q = \lceil \log_2 p \rceil$ [9] as is the case for the roughly halving circulant graph.

The reduce-scatter (disjoint all-reduce) algorithm Algorithm 1 implicitly constructs a tree at each processor r that captures how partial results are reduced together by the $\oplus$ operator as explained

ALGORITHM 4: Algorithm for the all-to-all operation for processor r , $0 \leq r < p$ using a circulant graph with skips s_k fulfilling Requirement (1). Processor r has p input blocks to each of the other processors stored as $S[i]$ and receives a distinct block from each of the other processors to be stored as $R[i]$, $0 \leq i < p$. The algorithm uses $R[i]$ to store intermediate blocks to be sent further on in the circulant graph and one extra intermediate communication buffer T sufficiently large to store p blocks.

```

procedure ALLALL( $S[p]$ ,  $R[p]$ )
     $R[r] \leftarrow S[r]$                                 ▷ Block from processor  $r$  to itself
     $l[0], u[0], B, B' \leftarrow 0, p, 1, 1$             ▷ First segment bounds
    for  $k = q - 1, \dots, 0$  do
         $j \leftarrow 0$                                 ▷ Pack  $j$  blocks
        for  $i = s_k, \dots, s_{k+1} - 1$  do            ▷ It holds that  $l[0] = 0$  and  $u[0] = s_{k+1}$ 
             $T[j] \leftarrow S[(r + i) \bmod p]$ 
             $j \leftarrow j + 1$ 
        end for
        for  $b = 1, \dots, B - 1$  do
            for  $i = l[b] + s_k, \dots, u[b] - 1$  do
                 $T[j] \leftarrow R[(r - i + p) \bmod p]$ 
                 $j \leftarrow j + 1$ 
            end for
        end for
         $t, f \leftarrow (r + s_k) \bmod p, (r - s_k + p) \bmod p$     ▷ To- and from processors
        Send( $T[0 \dots j - 1], t, C_p^s$ ) || Recv( $T[j \dots j + j - 1], f, C_p^s$ )
        for  $b = 0, \dots, B - 1$  do
            if  $l[b] + s_k < u[b]$  then                    ▷ Split segment for next round
                 $l[B'], bu[B'] \leftarrow l[b] + s_k, u[b]$ 
                 $B' \leftarrow B' + 1$ 
                 $u[b] \leftarrow l[b] + s_k$ 
            end if
        end for
        for  $b = B, \dots, B' - 1$  do                    ▷ Unpack another  $j$  blocks
            for  $i = l[b], \dots, u[b] - 1$  do
                 $R[(r - i + p) \bmod p] \leftarrow T[j]$ 
                 $j \leftarrow j + 1$ 
            end for
        end for
         $B \leftarrow B'$ 
    end for
end procedure

```

by Theorem 1. A concrete example is shown in Figure 1. Algorithm 4 performs no reductions, but maintains sequences of blocks corresponding to the trees being built. In each round, the same number of subtree blocks are sent and received by each processor r . The blocks of the subtrees are stored temporarily in R , such that at the end of the last communication round, $R[i]$, $0 \leq i < p$ is indeed the block from processor i to processor r . An intermediate buffer T is used for received and sent blocks that are packed to and from the corresponding positions in R .

In the first round, processor r receives $s_q - s_{q-1}$ blocks from processor $(r - s_{q-1} + p) \bmod p$. These blocks are roots of (singleton) subtrees and can be stored in $R[(r - i + p) \bmod p]$, $s_{q-1} \leq i < s_q$, since the same block indices are being sent to processor $(r + s_{q-1}) \bmod p$, but directly out of the send buffer S . This saves the initial, rotated copy of S into R as needed in Algorithm 1 and

Algorithm 2. In particular, the first of these blocks $R[(r - s_{q-1} + p) \bmod p]$ is the block from processor $(r - s_{q-q} + p) \bmod p$ to processor r and therefore already in the correct, final position in R . For the next rounds, the algorithm maintains a list of segments of blocks of subtrees. A segment is a consecutive range of indices of subtree blocks and defined by lower and upper bounds $[l, u[$. The segment for the first round has $l = 0$ and $u = s_q = p$. The subtree blocks that are sent and received are $R[(r - i + p) \bmod p]$ with $l + s_q \leq i < u$. For each such segment for round k , the lower bound l will be a sum of different $s_{k'}$ with $k < k' < q$, such that the first block with index $l + s_k$ is the final, correct block from processor $(r - (l + s_k) + p) \bmod p$ to processor r . After round k , each segment $[l, u[$ is split into (at most) two segments, namely $[l, l + s_k[$ and $[l + s_k, u[$. The first such segment is for new subtree roots, and the second for the children of these subtree roots. The number of segments is recorded in B which at most doubles in each communication round. In each communication round, all subtree blocks are packed and unpacked via the intermediate T buffer from and to R , also the blocks that are going to be sent again in the next round. An improved packing and unpacking scheme which saves roughly half the copies from T into R in each round, is implemented in Appendix A.

The tree constructed by Algorithm 4 is actually a shortest path tree which captures how the block from processor i to processor r is sent along a shortest path from i to r using different skips s_k . Since this holds for all blocks, the communication volume incurred by Algorithm 4 per processor is the sum of the lengths of the shorts paths from all processors i to r . This is clearly a lower bound for any all-to-all algorithm that can at best use shortest path routing in the given communication graph. In the following, P will denote the total path length (in number of communication edges) of a shortest path tree to a processor r from all other processors. In circulant graphs, all processors have isomorphic shortest path trees, and therefore the same, total shortest path length.

We summarize the explanation and discussion as follows.

THEOREM 9. *Algorithm 4 can solve the all-to-all communication problem in $\lceil \log_2 p \rceil$ send-receive communication rounds on inputs of p distinct blocks per processor to be sent to each of the other p processors using a roughly halving circulant graph. The total number of blocks sent and received per processor is exactly the total single source shortest path length P in a circulant graph with skips computed by roughly halving.*

Since blocks are sent along shortest paths from source to destination processor, and thus several times, the algorithm performs well only when blocks are small enough that the communication latency dominates the communication volume.

COROLLARY 10. *In a homogeneous, linear-affine transmission cost model where concurrent, bidirectional sending and receiving blocks of m' elements by all processors in a communication round can be charged $\alpha + \beta m'$ time, the all-to-all communication problem on input vectors of m elements uniformly partitioned into blocks of roughly m/p elements is solved in time*

$$T(m, p) = \alpha \lceil \log_2 p \rceil + \beta Pm/p$$

by Algorithm 4, where P is the sum of the lengths of all shortest paths to processor 0 from all other processors $1, \dots, p-1$ in a p -processor circulant graph with roughly halving skips.

As for Algorithm 1 and Algorithm 2, also Algorithm 4 (and Algorithm 5) works for any circulant graph fulfilling Requirement (1). This provides tuning potential for concrete implementations, different from what the algorithm by Bruck et al. [9] offers, by choosing different skip sequences with different distances between subsequent skips $s_{k+1} - s_k$. The classical straight-doubling circulant graph all-to-all algorithm by Bruck et al. [9] has the same properties, in particular, also here blocks are routed along shortest paths. However, the two types of circulant graphs, straight doubling as

Table 1. The Sum P of the Lengths of the Shortest Paths (in Number of Communication Edges) from $r = 0$ to $r = 1, \dots, p - 1$ in Straight Doubling and Roughly Halving Circulant Graphs

p	q	Straight doubling P	Roughly halving P
1024	10	5120	5120
1025	11	5121	4757
1026	11	5123	4765
1027	11	5125	4772
1028	11	5128	4780
1029	11	5130	4787
1030	11	5133	4795
1031	11	5136	4803
16384	14	114688	114688
16385	15	114689	108753
16386	15	114691	108765
16387	15	114693	108776
16388	15	114696	108788
16389	15	114698	108795
16390	15	114701	108807
16391	15	114704	108817

By the symmetry of the circulant graphs, P is the same for all processors.

used by Bruck [9] and roughly halving as used here, have different total shortest path lengths. A few examples with p around 2^{10} and p around 2^{14} are shown in Table 1. The reduction tree in Figure 1 has total shortest path length 45 as can easily be counted. When $p = 2^q$ is a power-of-two, the two types of circulant graphs coincide, but otherwise, the roughly halving skip pattern is significantly better, in the examples by about 10%. This reduction in communication volume may have a performance impact. An analytical expression for the total shortest path length for the roughly halving circulant graph is not known (to the author).

4 Implementation in and for MPI

Algorithms 1, 2, 3, and Algorithm 4 can all readily be implemented in MPI [15] with MPI_Sendrecv for the bidirectional, combined Send() || Recv() operation. Standard considerations as when implementing the doubling allgather algorithm of Bruck et al. [9] apply, see for instance [7, 25]. In particular, the doubling and halving schemes lead to latency contention and communication redundancy when run as written on clustered, hierarchical systems with constrained per node bandwidth. There are various ways to alleviate these problems which we do not consider in the implementations here [28].

Algorithms 1, 2, and 4 can compute the required skips on the fly in constant time per communication rounds by halving with rounding up. Algorithm 3 must precompute the $\lceil \log_2 p \rceil$ skips. For the reduction algorithms, all partial result blocks are kept in consecutive buffers and no reordering of blocks is needed in the $\lceil \log_2 p \rceil$ communication rounds. Reduction and copy operations can therefore be done as bulk operations over many blocks. For the all-to-all algorithm shown as Algorithm 4 and Algorithm 5, blocks do not form consecutive segments, therefore individual blocks are copied to and from the intermediate result buffer T . This is an inherent property (drawback) of message-combining all-to-all algorithms which can only to some extent be alleviated [9, 30], for instance with “tunneling” as elaborated in Algorithm 5 in Appendix A. Still, the implementations

are not “zero-copy” in the sense explained in Reference [30] since explicit, process-local copying between buffers S , T and R is performed.

5 Experimental Results

We have implemented Algorithms 1, 2, 3, and 5 for `MPI_Reduce_scatter_block`, `MPI_Allreduce` and `MPI_Alltoall` in MPI as written and discussed in the previous sections (these implementations are available from the author). Apart from the algorithmic properties (rounds and communication volume), there are other, system related factors that determine achieved and observable performance. We have conducted initial experiments to explore some of these. Our experimental system is a medium sized 36×32 processor cluster with 36 dual-socket compute nodes, each with two Intel(R) Xeon(R) Gold 6130F 16-core CPUs. The compute nodes are interconnected via dual Intel Omnipath interconnects each with a bandwidth of 100 Gbytes/s. We report results only with the apparently best performing MPI library available to us (with respect to collective operations) which is the IntelMPI 2021.13 library, and observe that MPI libraries differ both in concrete performance and in scalability characteristics. The implementations and benchmarks were compiled with the `-O3` option.

The applied benchmarking methodology and procedure influence observed performance [12, 13, 26]. For each of the algorithms, we choose an MPI process configuration over the compute nodes, and perform experiments with a range of problem sizes (here multiples of powers of 10 `MPI_DOUBLE` elements). For each experiment with a given input problem size, the MPI processes are synchronized as good as it goes with `MPI_Barrier`, and the duration of the collective operation is measured by each process. The time of the last process to complete is taken as the time of the experiment. Experiments are, for each problem size, repeated (here 40 times) and the best seen completion time (of a slowest process) is recorded and plotted as a function of the problem size (per process). In all experiments, we use `MPI_DOUBLE` as the element type, and for the reduction operations, reduce with `MPI_SUM` as the binary operator. The chosen process configurations are $p = 32 \times 1$ and $p = 32 \times 32$ MPI processes (power of two number of processes), and $p = 36 \times 1$, $p = 36 \times 31$ and $p = 36 \times 32$ MPI processes (non-powers of two). In the process configurations with only one process per compute node, $p = 32 \times 1$ and $p = 36 \times 1$, all communication is via the network, and here the simple, linear-affine communication cost model may apply reasonably well. In the process configurations with full(er) nodes, the 32 or 31 MPI processes per compute node will contend for the communication networks in some communication rounds, and a proportional serialization slowdown can be expected. There are different standard ways to deal with serialization [28], but we here benchmark the algorithms as they are. We would in these cases expect a well tuned MPI library function to perform (much) better (as long as reasonably good algorithms have been used).

We now discuss the results achieved with the IntelMPI 2021.13 library. With `MPI_Reduce_scatter_block`, `MPI_Allreduce` and `MPI_Alltoall` being among the most often used and important collective operations, we would expect good quality, well performing implementations of these in MPI libraries, making the library native implementations hard to beat. We plot the measured operation completion times as explained against the problem size (in bytes) for the chosen MPI process configurations. Both axes are in log-scale and show the performance characteristics over a wide range of problem sizes but blur finer details.

The results for the reduce-scatter (disjoint all-reduce) operation for the five different MPI process configurations are shown in Figure 2 and in Figure 3. Algorithm 1 was implemented in MPI as written, using a roughly halving circulant graph, and benchmarked against the library native `MPI_Reduce_scatter_block` operation. Benchmarking was done with problem sizes up to about $m = 10\,000\,000$ `MPI_DOUBLE` elements. The block size given as parameter to the `MPI_Reduce_scatter_block` operation is m/p . For most of the problem size range, Algorithm 1 performs consistently better

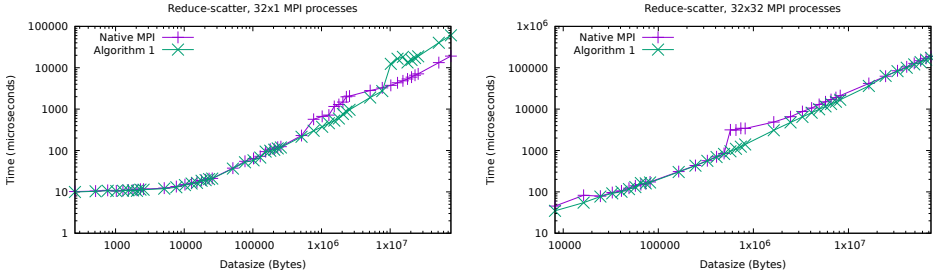


Fig. 2. Results for the MPI_Reduce_scatter_block operation, natively and by Algorithm 1, with IntelMPI 2021.13 with the $p = 32 \times 1$ and $p = 32 \times 32$ MPI process configurations.

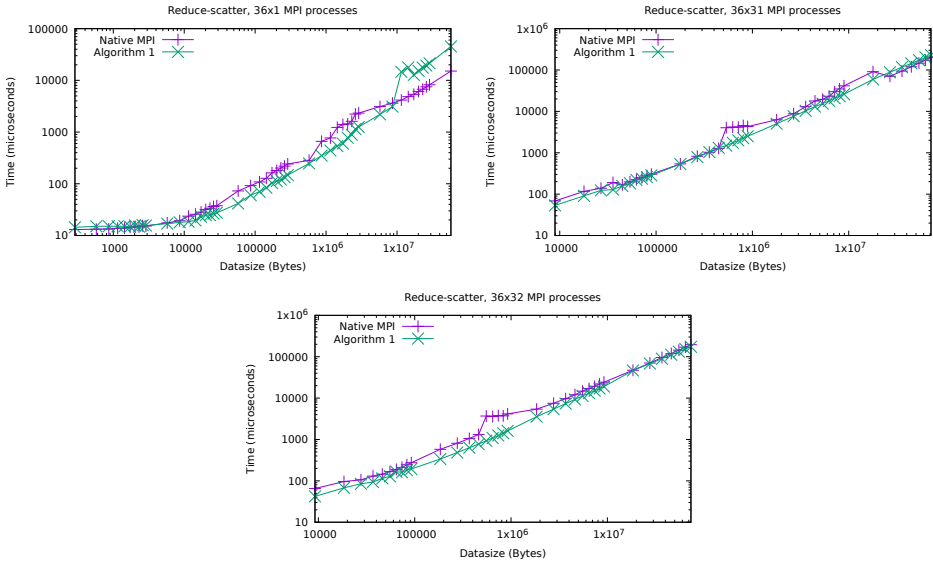


Fig. 3. Results for the MPI_Reduce_scatter_block operation, natively and by Algorithm 1, with IntelMPI 2021.13 with the $p = 36 \times 1$, $p = 36 \times 31$ and $p = 36 \times 32$ MPI process configurations.

than the native MPI_Reduce_scatter_block operation by up to a factor of two. In the configurations with only one MPI process per node, $p = 32 \times 1$ and $p = 36 \times 1$, there is a large, more than linear increase in running time around $m = 1\,000\,000$ MPI_DOUBLE elements, which is not in accordance with the performance estimate of Corollary 6. Why the native MPI_Sendrecv operations behaves in this manner (or whether this has other reasons) is unknown (but possibly related to the used Omnipath driver). Experiments with different circulant graph skip sequences did not improve performance, but can move the point where the performance degradation happens. It is surprising that Algorithm 1 performs so well, in comparison, for the configurations with full compute nodes, since nothing was done to make the implementation deal with the serialization and bandwidth issues between full compute nodes [28].

The results for the allreduce operation for the five different MPI process configurations are shown in Figures 4 and 5. Here, Algorithm 2 for larger problems is benchmarked against the MPI library native MPI_Allreduce operation. For the configuration with one MPI process per node, $p = 32 \times 1$

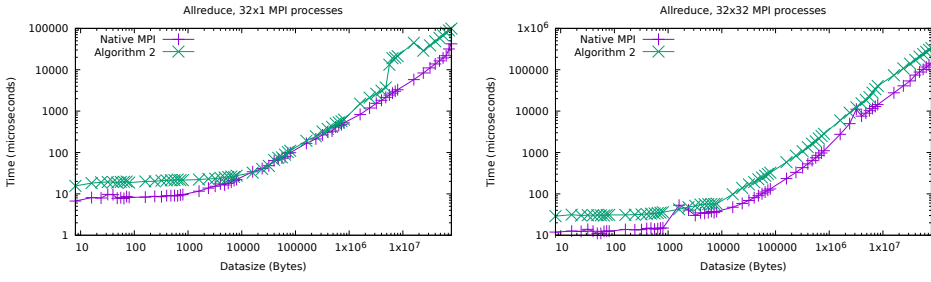


Fig. 4. Results for the MPI_Allreduce operation, natively and by Algorithm 2 with IntelMPI 2021.13 with the $p = 32 \times 1$ and $p = 32 \times 32$ MPI process configurations.

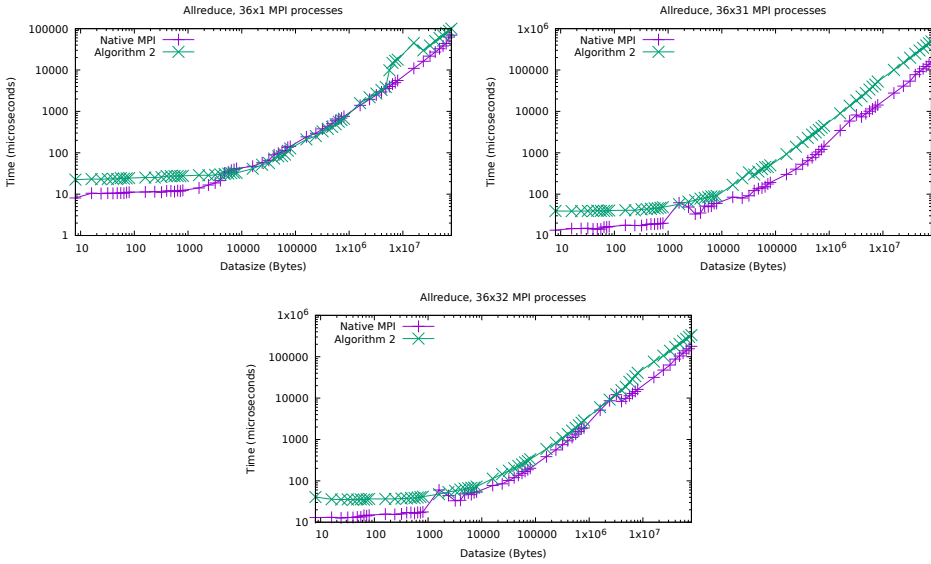


Fig. 5. Results for the MPI_Allreduce operation, natively and by Algorithm 2 with IntelMPI 2021.13 with the $p = 36 \times 1$, $p = 36 \times 31$ and $p = 36 \times 32$ MPI process configurations.

and $p = 36 \times 1$, there are ranges around $m = 10\,000$ and up to $m = 100\,000$ MPI_DOUBLE elements where Algorithm 2 can outperform the native MPI_Allreduce function, but also here, there are problem sizes at which the performance degrades unexpectedly. For instance, with $p = 36 \times 1$ in Figure 5 the measured time for $m = 100\,000$ MPI_DOUBLE elements is $662\mu s$ and for $m = 200\,000$ MPI_DOUBLE elements already $1579\mu s$. Even more dramatically, for $m = 600\,000$ MPI_DOUBLE elements, the time is $3877\mu s$ and for $m = 700\,000$ MPI_DOUBLE elements jumps to $14842\mu s$. This reproducible behavior is in need of investigation.

We also implemented Algorithm 3 for the allreduce operation for small problems, and benchmark with $m \leq 10\,000$ MPI_DOUBLE elements. Results for 32 compute nodes are shown in Figure 6, and results for 36 compute nodes in Figure 7. Overall, Algorithm 3 is not competitive with the library native MPI_Allreduce implementation. The differences for the smallest problem sizes, $m \leq 100$ MPI_DOUBLE elements are only a few microseconds, though, and can possibly be eliminated in by a more efficient implementation with less overhead (in terms of constants) inside an MPI library.

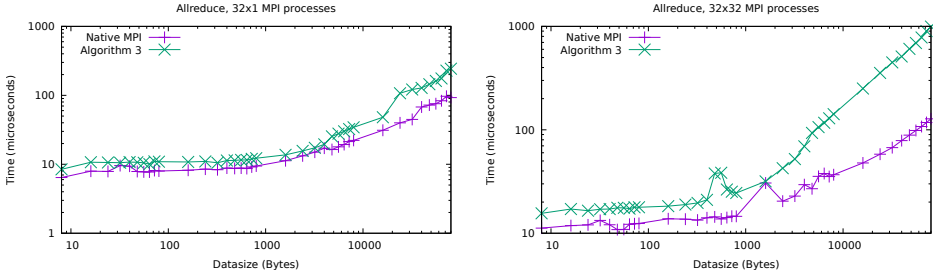


Fig. 6. Results for the MPI_Allreduce operation, natively, and by Algorithm 3 with IntelMPI 2021.13 with the $p = 32 \times 1$ and $p = 32 \times 32$ MPI process configurations.

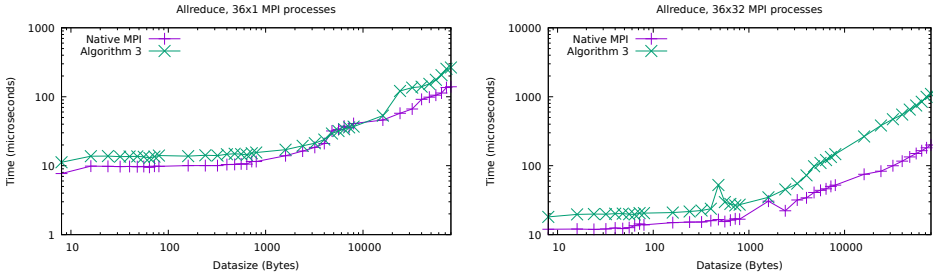


Fig. 7. Results for the MPI_Allreduce operation, natively, and by Algorithm 3 with IntelMPI 2021.13 with the $p = 36 \times 1$ and $p = 36 \times 32$ MPI process configurations.

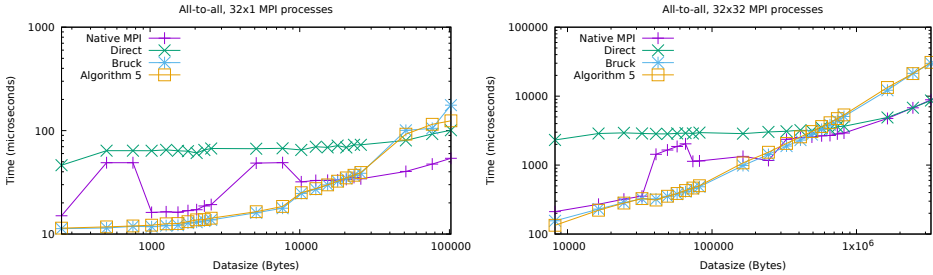


Fig. 8. Results for the MPI_Alltoall operation, natively, by Algorithm 5, by a “Bruck algorithm” and by a direct, $p - 1$ round algorithm with IntelMPI 2021.13 with the $p = 32 \times 1$ and $p = 32 \times 32$ MPI process configurations.

The results for the all-to-all operation for the five different MPI process configurations are shown in Figures 8 and 9. Since Algorithm 4 and all related, message-combining all-to-all algorithms are relevant only for smaller problem sizes [9], we show results for $m/p \leq 400$ MPI_DOUBLE elements. We compare our implementation of Algorithm 5 to our own implementation of the logarithmic round, doubling skips circulant graph algorithm of Bruck et al. [9], also applying “tunneling”, against a direct $p - 1$ round algorithm where each MPI process sends and receives its blocks directly to and from all other processes and against the MPI library native implementation of the MPI_Alltoall collective. For very small problems, but in all process configurations, both of the logarithmic round algorithms can significantly outperform the native MPI_Alltoall implementation,

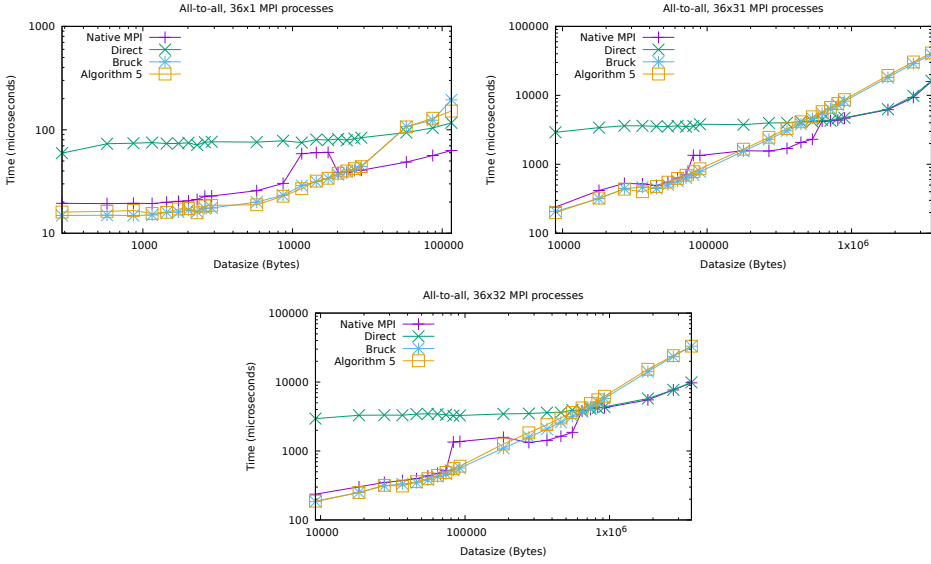


Fig. 9. Results for the MPI_Alltoall operation, natively, by Algorithm 5, by a “Bruck algorithm” and by a direct, $p - 1$ round algorithm with IntelMPI 2021.13 with the $p = 36 \times 1$, $p = 36 \times 31$ and $p = 36 \times 32$ MPI process configurations.

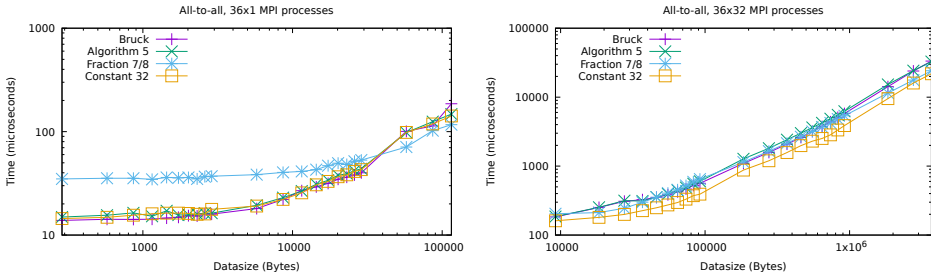


Fig. 10. Results with different skip sequences for Algorithm 5 in comparison to a “Bruck algorithm” with IntelMPI 2021.13 with the $p = 36 \times 1$ and $p = 36 \times 32$ MPI process configurations. Additional results are shown for skip sequences with $s_k = \frac{7}{8}s_{k+1}$ and with constant difference $s_{k+1} - s_k = 32$.

which apparently also use a message-combining, logarithmic round algorithm [23]. Our Algorithm 5 performs on par with the Bruck algorithm implementation. The reduced total shortest path length of Table 1 does not lead to a visible improvement in these experiments. In the configurations with one MPI process per node, $p = 32 \times 1$ and $p = 36 \times 1$, there is again a large, much more than linear increase in running time (here around $m/p = 100$ MPI_DOUBLE elements) both for Algorithm 5 and the Bruck implementation. The library native MPI_Alltoall implementation shows many undesirable, but reproducible performance phenomena (erratic, non-linear running time as function of m/p).

For Algorithm 5, it turned out that the freedom in the choice of skip sequence for the circulant graph can make significant differences in performance. In Figure 10, we run Algorithm 5 additionally with a skip sequence where $s_k = \frac{7}{8}s_{k+1}$, and with a fixed difference skip sequence where $s_{k+1} - s_k =$

Table 2. The Estimated Running Times of the p -processor, Roughly Halving Circulant Graph Algorithms Described and Discussed in this Article in a Linear-Affine Cost Model where Sending and Receiving m Elements Takes Time $\alpha + \beta m$ and Reduction of Two m Element Vectors Time γm

Operation	MPI collective	Estimated Time	Conditions
reduce-scatter (disjoint all-reduce)	MPI_Reduce_scatter_block	$\alpha \lceil \log_2 p \rceil + (\beta + \gamma) \frac{p-1}{p} m$	
allgather	MPI_Allgather	$\alpha \lceil \log_2 p \rceil + \beta \frac{p-1}{p} m$	
allreduce	MPI_Allreduce	$\lceil \log_2 p \rceil (\alpha + \beta m) + (2 \lceil \log_2 p \rceil - 1) \gamma m$	Small m
allreduce	MPI_Allreduce	$2\alpha \lceil \log_2 p \rceil + (2\beta + \gamma) \frac{p-1}{p} m$	Larger m
allgather-merge		$\alpha \lceil \log_2 p \rceil + \beta \frac{p-1}{p} m + \gamma m$	
reduce	MPI_Reduce	$\lceil \log_2 p \rceil (\alpha + \beta m + \gamma m)$	Small m
		$2\alpha \lceil \log_2 p \rceil + (2\beta + \gamma) \frac{p-1}{p} m$	Larger m
broadcast	MPI_Bcast	$\lceil \log_2 p \rceil (\alpha + \beta m)$	Small m
gather	MPI_Gather	$\alpha \lceil \log_2 p \rceil + \beta \frac{p-1}{p} m$	
scatter	MPI_Scatter	$\alpha \lceil \log_2 p \rceil + \beta \frac{p-1}{p} m$	
all-to-all	MPI_Alltoall	$\alpha \lceil \log_2 p \rceil + \beta P m / p$	Small m

For the all-to-all algorithm, P is the total single source shortest path length in the p -processor circulant graph. The results hold uniformly for any number of processors p .

32 (as long as $s_k > 32$), and compare that to the Bruck implementation which does not have this potential for adaptation. Especially for the $p = 36 \times 32$ configuration, the running time is improved over the baseline by a factor of about two, without sacrificing performance on any of the problem sizes.

6 Summary

This article gave a very simple and easily implementable algorithm for the reduce-scatter (disjoint all-reduce) operation as found in message-passing frameworks like MPI [15] that works for any number of processors and MPI processes. The algorithm is optimal in number of communication rounds and in communication and computation volume. The algorithm was used as building block for an allreduce algorithm that is likewise optimal in communication and computation volume, and an all-to-all algorithm that is optimal in the number of communication rounds and better than previous algorithms [9, 30] in terms of message-volume for non-powers of two numbers of processors. These algorithms work for any circulant graph with a skip sequence fulfilling Requirement (1), which provides room for tuning and tradeoffs. As contrast, we gave an algorithm for the allreduce operation that is optimal in the number of communication rounds [3, 8, 29], but not in the communication volume, and works for circulant graphs with a skip sequence defined by repeated halving of the number of processes. We implemented all algorithms in MPI and performed a series of indicative experiments on a small cluster, showing that in particular the reduce-scatter and the all-to-all algorithms can have considerable potential for implementation in MPI libraries.

By specialization of the algorithms, round and volume-optimal algorithms for the rooted, regular scatter and gather problems can easily be derived (MPI_Scatter, MPI_Gather). We also indicated that the algorithms can be used for reduction to root (MPI_Reduce), and by implication, for broadcast (MPI_Bcast), and may be attractive for small problem sizes. Circulant graphs are thus a universal scheme for collective operations as found in MPI and elsewhere. The achieved results with roughly halving circulant graphs in a linear-affine transmission cost model are summarized in Table 2.

Acknowledgments

The author thanks friends, colleagues and students for discussions and comments, as well as the anonymous reviewers for constructive suggestions that significantly contributed to shape the final version of this paper. The author also thanks Mike Wilkins, Yanfei Guo and Sean Rhee of Argonne National Laboratory for productive interaction on the implementation of some of the algorithms of this paper on real, large-scale high-performance systems. The students in the Summer 2025 HPC class at TU Wien are honorably mentioned for implementing and evaluating the collective allgather-merge operation using MPI_Allgather, well-known allgather algorithms [9, 29] as well as Algorithm 3 as templates.

References

- [1] Michael Axtmann and Peter Sanders. 2017. Robust massively parallel sorting. In *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 83–97.
- [2] Amotz Bar-Noy and Shlomo Kipnis. 1994. Broadcasting multiple messages in simultaneous send/receive systems. *Discrete Applied Mathematics* 55, 2 (1994), 95–105.
- [3] Amotz Bar-Noy, Shlomo Kipnis, and Baruch Schieber. 1993. An optimal algorithm for computing census functions in message-passing systems. *Parallel Processing Letters* 3, 1 (1993), 19–23.
- [4] Michael Barnett, Richard J. Littlefield, David G. Payne, and Robert A. van de Geijn. 1995. Global combine algorithms for $2 - D$ meshes with wormhole routing. *Journal of Parallel and Distributed Computing* 24, 2 (1995), 191–201.
- [5] J.-C. Bermond, F. Comellas, and D. F. Hsu. 1995. Distributed loop computer networks: A survey. *Journal of Parallel and Distributed Computing* 24, 1 (1995), 2–10.
- [6] Massimo Bernaschi, Giulio Iannello, and Mario Lauria. 2003. Efficient implementation of reduce-scatter in MPI. *Journal of Systems Architecture* 49, 3 (2003), 89–108.
- [7] Amanda Bienz, Shreeman Gautam, and Amun Kharel. 2022. A locality-aware bruck allgather. In *29th European MPI Users' Group Meeting (EuroMPI/USA)*. ACM, 18–26.
- [8] Jehoshua Bruck and Ching-Tien Ho. 1993. Efficient global combine operations in multi-port message-passing systems. *Parallel Processing Letters* 3, 4 (1993), 335–346.
- [9] Jehoshua Bruck, Ching-Tien Ho, Schlomo Kipnis, Eli Upfal, and Derrick Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems* 8, 11 (1997), 1143–1156.
- [10] Adrián Castelló, Mar Catalán, Manuel F. Dolz, Enrique S. Quintana-Ortí, and José Duato. 2023. Analyzing the impact of the MPI allreduce in distributed training convolutional neural networks. *Computing* 105, 5 (2023), 1101–1119.
- [11] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert A. van de Geijn. 2007. Collective communication: Theory, practice, and experience. *Concurrency and Computation: Practice and Experience* 19, 13 (2007), 1749–1783.
- [12] William Gropp and Ewing Lusk. 1999. Reproducible measurements of MPI performance characteristics. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 1697)*. Springer, 11–18.
- [13] Sascha Hunold and Alexandra Carpen-Amarie. 2016. Reproducible MPI benchmarking is still not as easy as you think. *IEEE Transactions on Parallel and Distributed Systems* 27, 12 (2016), 3617–3630.
- [14] Giulio Iannello. 1997. Efficient algorithms for the reduce-scatter operation in LogGP. *IEEE Transactions on Parallel and Distributed Systems* 8, 9 (1997), 970–982.
- [15] MPI Forum. 2023. *MPI: A Message-Passing Interface Standard. Version 4.1*. Retrieved from www.mpi-forum.org
- [16] Truong Thao Nguyen, Mohamed Wahib, and Ryousei Takano. 2021. Efficient MPI-allreduce for large-scale deep learning on GPU-clusters. *Concurrency and Computation: Practice and Experience* 33, 12 (2021), 1–20.
- [17] Emin Nuriyev, Ravi Reddy Manumachu, Samar Aseeri, Mahendra K. Verma, and Alexey L. Lastovetsky. 2024. SUARA: A scalable universal allreduce communication algorithm for acceleration of parallel deep learning applications. *Journal of Parallel and Distributed Computing* 183 (2024), 104767.
- [18] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing* 69, 2 (2009), 117–124.
- [19] Rolf Rabenseifner and Jesper Larsson Träff. 2004. More efficient reduction algorithms for message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 3241)*. Springer, 36–46.
- [20] Hubert Ritzdorf and Jesper Larsson Träff. 2006. Collective operations in NEC's high-performance MPI libraries. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*. 100.

- [21] Paul Sack and William Gropp. 2010. A scalable MPI_Comm_split algorithm for exascale computing. In *Recent Advances in the Message Passing Interface. 17th European MPI Users' Group Meeting (EuroMPI) (Lecture Notes in Computer Science, Vol. 6305)*. Springer, 1–10.
- [22] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. 2009. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing* 35, 12 (2009), 581–594.
- [23] Rajeev Thakur, William D. Gropp, and Rolf Rabenseifner. 2005. Improving the performance of collective operations in MPICH. *International Journal on High Performance Computing Applications* 19, 1 (2005), 49–66.
- [24] Jesper Larsson Träff. 2005. An improved algorithm for (non-commutative) reduce-scatter with an application. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 3666)*. Springer, 129–137.
- [25] Jesper Larsson Träff. 2006. Efficient allgather for regular SMP-clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 4192)*. Springer, 58–65.
- [26] Jesper Larsson Träff. 2012. mpicoscope: Towards an MPI benchmark tool for performance guideline verification. In *Recent Advances in Message Passing Interface. 19th European MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 7490)*. Springer, 100–109.
- [27] Jesper Larsson Träff. 2025. Optimal broadcast schedules in logarithmic time with applications to broadcast, reduction, all-broadcast and all-reduction. *ACM Transactions on Parallel Computing* 12, 3 (2025), 1–21.
- [28] Jesper Larsson Träff and Sascha Hunold. 2020. Decomposing MPI collectives for exploiting multi-lane communication. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 270–280.
- [29] Jesper Larsson Träff, Sascha Hunold, Nikolaus Manes Funk, and Ioannis Vardas. 2023. Uniform algorithms for reduce-scatter and (most) other collectives for MPI. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 284–294.
- [30] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. 2014. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *28th ACM International Conference on Supercomputing (ICS)*. ACM, 135–144.
- [31] Robert van de Geijn. 1994. On global combine operations. *Journal of Parallel and Distributed Computing* 22, 2 (1994), 324–328.
- [32] Adam Weingram, Yuke Li, Hao Qi, Darren Ng, Liuyao Dai, and Xiaoyi Lu. 2023. xCCL: A survey of industry-led collective communication libraries for deep learning. *Journal of Computer Science and Technology* 38, 1 (2023), 166–195.

Appendix

A The All-to-All Algorithm with Tunneling

Algorithm 4 in each communication round copies all received blocks into intermediate positions in the receive buffer R and for the next round copies blocks again out of the R buffer. For blocks received in one round which are going to be sent in the next round, this entails an extra, avoidable copy operation and is, in this sense, inefficient. This inefficiency is removed by “tunneling” blocks that are going to be used in the next communication round directly from their position in the receive buffer T' to their new position in the send buffer T . Packing, unpacking and tunneling is all done before the communication operation. This somewhat more elaborate version of the algorithm is shown explicitly as Algorithm 5, and is the version that was implemented and used in Section 5.

ALGORITHM 5: Algorithm for the all-to-all operation for processor r , $0 \leq r < p$ using a circulant graph with skips s_k fulfilling Requirement (1). Processor r has p input blocks to each of the other processors stored as $S[i]$ and receives a distinct block from each of the other processors to be stored as $R[i]$, $0 \leq i < p$. The algorithm uses $R[i]$ to store intermediate blocks to be sent further on in the circulant graph and extra intermediate communication buffers T, T' sufficient to store p blocks. Received blocks in T' to be used in the next round are tunneled directly into T .

```

procedure ALLALL( $S[p], R[p]$ )
   $R[r] \leftarrow S[r]$                                 ▷ Block from processor  $r$  to itself
   $l[0], u[0], B, B' \leftarrow 0, p, 1, 1$            ▷ First segment bounds
  for  $k = q - 1, \dots, 0$  do
     $j \leftarrow 0$                                 ▷ Pack  $j$  blocks
    for  $i = s_k, \dots, s_{k+1}$  do                 ▷ It holds that  $l[0] = 0$  and  $u[0] = s_{k+1}$ 
       $T[j], j \leftarrow S[(r + i) \bmod p], j + 1$ 
    end for
    for  $b = 1, \dots, B - 1$  do
      for  $i = l[b] + s_k, \dots, u[b] - 1$  do
         $T[j], j \leftarrow R[(r - i + p) \bmod p], j + 1$ 
      end for
    end for
     $j' \leftarrow 0$                                 ▷ Unpack and tunnel  $j'$  blocks
    for  $b = B, \dots, B' - 1$  do
      for  $i = l[b], \dots, \min(l[b] + s_k, u[b]) - 1$  do
         $R[(r - i + p) \bmod p], j' \leftarrow T'[j'], j' + 1$ 
      end for
      for  $i = l[b] + s_k, \dots, u[b] - 1$  do
         $T[j] \leftarrow T'[j']$ 
         $j, j' \leftarrow j + 1, j' + 1$ 
      end for
    end for
     $B \leftarrow B'$                                 ▷ All blocks done
     $t, f \leftarrow (r + s_k) \bmod p, (r - s_k + p) \bmod p$   ▷ To- and from processors
    Send( $T[0 \dots j - 1], t$ ) || Recv( $T'[0 \dots j - 1], f$ )
    for  $b = 0, \dots, B - 1$  do
      if  $l[b] + s_k < u[b]$  then                 ▷ Split segment for next round
         $l[B'], u[B'] \leftarrow l[b] + s_k, u[b]$ 
         $B' \leftarrow B' + 1$ 
         $u[b] \leftarrow l[b] + s_k$ 
      end if
    end for
  end for
   $j' \leftarrow 0$                                 ▷ Unpack all blocks for last round
  for  $b = B, \dots, B' - 1$  do
    for  $i = l[b], \dots, u[b] - 1$  do
       $R[(r - i + p) \bmod p], j' \leftarrow T'[j'], j' + 1$ 
    end for
  end for
end procedure

```

Received 13 February 2025; revised 30 May 2025; accepted 25 July 2025