Cartesian Collective Communication

Jesper Larsson Träff
TU Wien, Faculty of Informatics
Vienna, Austria
traff@par.tuwien.ac.at

Sascha Hunold
TU Wien, Faculty of Informatics
Vienna, Austria
hunold@par.tuwien.ac.at

ABSTRACT

We introduce *Cartesian Collective Communication* as sparse, collective communication defined on processes (processors) organized into d-dimensional tori or meshes. Processes specify local neighborhoods, e.g., stencil patterns, by lists of relative Cartesian coordinate offsets. The Cartesian collective operations perform data exchanges (and reductions) over the set of all neighborhoods such that each process communicates with the processes in its local neighborhood. The key requirement is that local neighborhoods must be structurally identical (isomorphic). This makes it possible for processes to compute correct, deadlock-free, efficient communication schedules for the collective operations locally without any interaction with other processes. Cartesian Collective Communication substantially extends collective neighborhood communication on Cartesian communicators as defined by the MPI standard, and is a restricted form of neighborhood collective communication on general, distributed graph topologies.

We show that the restriction to isomorphic neighborhoods permits communication improvements beyond what is possible for unrestricted graph topologies by presenting non-trivial message-combining algorithms that reduce communication latency for Cartesian alltoall and allgather collective operations. For both types of communication, the required communication schedules can be computed in linear time in the size of the input neighborhood. Our benchmarks show that we can for small data block sizes substantially outperform the general MPI neighborhood collectives implementing the same communication pattern.

We discuss different possibilities for supporting Cartesian Collective Communication in MPI. Our library is implemented on top of MPI and uses the same signatures for the collective communication operations as the MPI (neighborhood) collectives. Our implementation requires essentially only one single, new communicator creation function, but even this might not be needed for implementation in an MPI library.

CCS CONCEPTS

• **Computing methodologies** → *Massively parallel algorithms*; *Parallel programming languages.*

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337848

ACM Reference Format:
Jesper Larsson Träff and Sascha Hunold. 2019. Cartesian Collective Communication. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3337821.3337848

1 INTRODUCTION

Many parallel computations are structured as stencil operations where each element of some regular, d-dimensional structure (e.g., 2- or 3-dimensional matrix) is repeatedly updated using values of a small set of neighboring elements with all element neighborhoods having the same structure (e.g., 5-point, 9-point, 27-point, 13-point, 125-point, uxx-stencils). On distributed memory systems this typically leads to communication patterns where all processes (processors) repeatedly need to exchange data with a small number of neighboring processes. With processes organized in a d-dimensional mesh- or torus pattern, communication patterns become stencils similar to the stencils describing the element updates. In each exchange operation, all processes communicate with other processes, all following (almost) the same pattern determined by the computational stencil and the regular mesh or torus organization of the processes.

The Message-Passing Interface (MPI) [11] provides functionality to support such computations. Cartesian topologies logically organize processes into d-dimensional meshes or tori, meaning that processes can be referred to by coordinate as well as by rank, and define *fixed* (inflexible), local process neighborhoods consisting of all distance one neighbors. MPI also provides the possibility for the MPI implementation to map (reorder) the logical process layout to characteristics of the physical machine for good communication performance with distance one neighbors; but current MPI libraries do not exploit these possibilities [6]. Neighborhood collective operations of the alltoall and the allgather kind make it possible to perform stencil type exchanges, ideally supported efficiently by the MPI library implementation. In short, a standard 5-point stencil update over a large matrix can be implemented concisely with the collective `MPI_Neighbor_alltoallv` operation, with `MPI_Cart_create` to define the logical process organization. Such is the intention of MPI.

Unfortunately, Cartesian MPI communicators have severe shortcomings that limit their usefulness. An obvious such shortcoming is the rigid, implicit definition of neighborhoods to consist of only the immediate, $2d$ distance one neighbors (excluding the process itself, d being the number of dimensions). Thus, stencil operations (e.g., 9-point updates in the 2-dimensional case) which also require exchange with processes on the mesh or torus diagonals cannot be implemented with the MPI neighborhood collectives on Cartesian communicators alone, since the diagonal processes are not

part of the default neighborhoods. Also, Cartesian process reordering, should it be supported by a good MPI library implementation, cannot take diagonals into account to improve communication performance with these neighbors. Thus, for more general stencil patterns, the application programmer would have to define neighborhoods using the general, distributed graph topology functionality [8], e.g., the operations `MPI_Dist_graph_create` or `MPI_Dist_graph_create_adjacent`. However, the important information that all processes have structurally similar neighborhoods as defined by the stencil cannot readily be conveyed to the MPI library with these interfaces, unless the application programmer observes a certain discipline, meaning that optimizations relying on such assumptions, e.g., message-combining to avoid direct diagonal communication, cannot be applied by the MPI library. Neighborhoods defined by general MPI graph topologies are weighted, which can possibly be used for better process remapping. In contrast, the simple Cartesian neighborhoods associate no weights with neighbors.

We discuss possible ways to overcome these shortcomings in MPI in order to provide more efficient support for *Cartesian Collective Communication* in which processes are organized as d-dimensional meshes or tori (similar to `MPI_Cart_create`), and neighborhoods are defined by lists of relative coordinate offsets for the neighboring processes, with the proviso that all processes employ the *exact same relative neighborhoods*. On this assumption, it is straight-forward to devise correct and deadlock-free algorithms for the sparse collective operations [16]. The main contribution of the present paper is to give non-trivial, *message-combining algorithms* for both alltoall and allgather type sparse collective operations that work for any sparse, isomorphic neighborhoods. We benchmark these algorithms using symmetric and asymmetric stencils in comparison to MPI neighborhood collective operations. For smaller message-block sizes, where communication latency is a dominating term, we can demonstrate the expected reduction in completion times compared to the neighborhood collectives in three standard MPI libraries on two different systems, including 1024 nodes on the Cray-*Titan*. This vindicates the potential of message-combining for small-block application characteristics, and demonstrates shortcomings of current implementations of the MPI neighborhood collectives.

1.1 Related Work

The MPI neighborhood collectives [11] were intended to provide support for sparse collective communication both for general communication graphs (no structural restrictions) [8], but also for highly structured graphs like stencils. The generality of the interface, however, makes it difficult to formulate worst-case efficient algorithms and implementations, beyond the trivial direct delivery implementations that may suffer from high latencies (for small problems) and congestion in the communication network. Optimization principles for unrestricted neighborhoods have been discussed in, e.g., [9, 10]. The latter paper [10] shows how to exploit message-combining to reduce latency in communication graphs with sufficiently large, common neighborhoods between processes. The techniques in [10] would give rise to different and weaker message-combining than the algorithms developed in the following, at much higher preprocessing cost. Further, recent work in this direction can be found in [4].

For the stencil graph case, there seems to be little work, be it on the use of neighborhood collectives in actual stencil computations [5], or on communication optimizations of the operations, despite various well-known and often applied message-combining tricks. Early work can be found in [2]. The shortcomings of MPI for stencil operations with collective operations are well-known, see, e.g., [16]. In contrast, recent literature on improving stencil computations on shared-memory and GPU systems, also for complex, non-regular stencils is extensive, see, e.g., [18].

The message-combining algorithms and implementations presented in this paper exploit MPI derived datatypes to eliminate explicit copies between intermediate communication buffers, similarly to the usage in [17], and are in this sense *zero-copy* [7]. For global collective allgather and alltoall operations, there has been considerable work on good algorithms for mesh and torus networks, typically employing message-combining, see, e.g., [13, 14]. In the limit, where the neighborhoods cover the whole set of processes, our algorithms will be weaker (more communications rounds); on the other hand, the case of small, irregular neighborhoods has not been addressed in this work.

2 A CARTESIAN COLLECTIVE COMMUNICATION INTERFACE

We now motivate and discuss a library for supporting *Cartesian Collective Communication* in MPI. The concrete proposal extends and simplifies an earlier suggestion for *isomorphic*, sparse collective communication [16]. We present the interface functions as actually implemented in the library accompanying this paper and used later for benchmarking the sparse, collective communication operations.

Let p be the number of MPI processes, and assume that they are organized in a d-dimensional mesh or torus with dimension sizes $p_0, p_1, \ldots, p_{d-1}$ with $\Pi_{i=0}^{d-1} p_i = p$. Each MPI process with rank $r, 0 \leq r < p$ is additionally identified by a coordinate vector $(r_0, r_1, \ldots r_{d-1})$ with $0 \leq r_i < p_i$ for $i = 0, \ldots, d-1$. A (sparse) *t-neighborhood* for process r is a collection of t *target* processes to which process r shall send data, given as a(n ordered) sequence (list) of t *relative coordinate vectors* $N^0, N^1, \ldots N^{t-1}$. Each neighbor vector N^i has the form $(n_0^i, n_1^i, \ldots, n_{d-1}^i)$ for arbitrary integer offsets n_j^i (positive or negative). A t-neighborhood is allowed to have repetitions of relative coordinate vectors. A process is a neighbor of itself if relative coordinate vector $(0, 0, \ldots, 0)$ is in the t-neighborhood. A set of *identical* t-neighborhoods for a set of processes is said to be *Cartesian*. A *Cartesian Collective Communication* operation is a collective operation over p processes with Cartesian neighborhoods. Each process $(r_0, r_1, \ldots, r_{d-1})$ with t-neighborhood $N^0, N^1, \ldots, N^{s-1}$ shall send data to the t *target processes* $((r_0 + n_0^i) \bmod p_0, (r_1 + n_1^i) \bmod p_1, \ldots, (r_{d-1} + n_{d-1}^i) \bmod p_{d-1})$, assuming that all dimensions are *periodic*; details for non-periodic meshes are not discussed further here. Since neighborhoods are Cartesian (identical), it follows that the process will then need to receive data from t *source* processes $((r_0 - n_0^i) \bmod p_0, (r_1 - n_1^i) \bmod p_1, \ldots, (r_{d-1} - n_{d-1}^i) \bmod p_{d-1})$. Note that different target neighbors may map to the same process.

Listing 1: The collective Cartesian neighborhood setup function. All calling processes in the communicator `comm` must supply exactly the same list of relative target coordinates.

```
int Cart_neighborhood_create(MPI_Comm comm,
  int d,
  const int dimensions[], const int periods[],
  int t, const int targetrelative[],
  const int weight[], MPI_Info info, int reorder,
  MPI_Comm *cartcomm);
```

In our library, Cartesian neighborhoods are defined using the MPI-like function shown in Listing 1. The `Cart_neighborhood_create` operation must be called by all processes in the communicator `comm` with the same parameters as the `MPI_Cart_create` operation, namely the dimension d of the Cartesian torus or mesh for the processes with dimension sizes and periodicity along each coordinate. The t-neighborhood is supplied by an additional, flattened list (array) of d-dimensional relative coordinate vectors. Each neighbor can be assigned a weight; unweighted neighborhoods are identified by giving `MPI_UNWEIGHTED` as argument. Weighted neighborhoods can be taken into account if process remapping is to be attempted by setting the `reorder` flag true. A new (Cartesian) communicator `cartcomm` with this information attached (and preprocessed) is returned. For a call to `Cart_neighborhood_create` to be correct, the neighborhood must be Cartesian, that is all calling processes must supply exactly the same list of t relative coordinate vectors.

The Cartesian Collective Communication operations considered here are of the allgather and the alltoall type. In an allgather operation (`Cart_allgather`, `Cart_allgatherv`, `Cart_allgatherw`)) each process r shall send the same *data block* to all of its t target neighbors. Per symmetry, process r will receive a (personal) data block from each of its t source neighbors. In the alltoall operations (`Cart_alltoall`, `Cart_alltoallv`, `Cart_alltoallw`), each process will have a possibly different (personalized) data block to each of its t target neighbors, and will likewise receive a data block from each of its source neighbors. The operations are collective and blocking in the MPI sense, meaning that all processes must partake and that a process can return when the called operation is completed from that process' point of view. The signatures of the operations are exactly as the corresponding MPI neighborhood collectives [11, Chapter 7.6] (which, we note, have the same signatures as the corresponding standard collective operations) with the same arguments and conventions for element datatypes and buffer handling (preallocated send and receive buffers with blocks stored in target and source neighbor order, matching type signatures). Both regular and irregular variants are implemented. In addition to MPI, our library also implements a `Cart_allgatherw` operation that allows different datatypes for different source blocks.

In order to later provide for non-blocking, persistent versions of the Cartesian collectives (as currently discussed in the MPI Forum), we also specify initialization calls for each of the allgather and alltoall operations. We use these later as handles for precomputing communication schedules for reuse. These functions take exactly the same arguments as the corresponding collective operations (`Cart_allgather_init`, ...).

Listing 2: Support functionality for neighborhood manipulation on Cartesian communicators.

```
int Cart_relative_rank(MPI_Comm cartcomm,
    const int relative[], int *rank);
int Cart_relative_shift(MPI_Comm cartcomm,
    const int relative[],
    int *inrank, int *outrank);
int Cart_relative_coord(MPI_Comm cartcomm,
    const int rank, int relative[]);

int Cart_neighbor_count(MPI_Comm cartcomm, int *t);
int Cart_neighbor_get(MPI_Comm cartcomm,
    const int maxin,
    int source[], int sourceweight[],
    const int maxout,
    int target[], int targetweight[]);
```

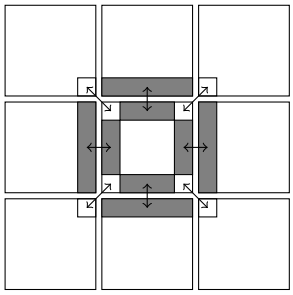


Figure 1: Process neighborhood and communication pattern for two dimensional, 9-point stencil computation.

Helper functionality to translate between relative coordinate vectors and ranks is defined on Cartesian communicators to aid applications in setting up neighborhoods. Query functions similar to the MPI query functions for distributed graphs return the neighborhood for the calling process as lists (of ranks) of source and target processes. Some of the interfaces are shown in Listing 2.

2.1 Cartesian Collective Communication in Use

The intended usage of MPI neighborhood collectives over Cartesian communicators is evidently to aid implementation of standard $2d+1$-point stencil computations. An example and discussion of the use of `MPI_Neighbor_alltoallw` for 5-point stencils can be found in [5].

As explained, Cartesian MPI communicators do not suffice for the implementation of 3^d-point stencils. Figure 1 illustrates the communication pattern for a 9-point stencil with a halo (ghost cell region) of depth 1. Listing 3 outlines how this can now be implemented with Cartesian Collective Communication. The `ROW` datatype could be just `MPI_DOUBLE`, assuming that the matrix is stored in row-major order. The `COL` datatype could be an MPI vector type, and the corner again an `MPI_DOUBLE` describing a simple 1×1 cell of the matrix. For deeper ghost regions, or complex, higher-order stencils [1, 12], the corner type would have more

Listing 3: Process neighbor communication for 9-point stencil computation showing both the neighborhood and the communication setup. The stencil updates themselves require `Cart_alltoallw` in each iteration. `ROW`, `COL`, and `COR` are MPI datatypes describing rows, columns, and corners of the two dimensional matrix, respectively.

```
double matrix[n+2][n+2];
int t = 8;

target[t] = [0,1, 0,-1, -1,0, 1,0,
             -1,1, 1,1, 1,-1, -1,-1];
Cart_neighborhood_create(comm,2,...,
                         t,target,...,&cart);

sendcount[0] = n; // upper row out
senddisp[0] = 1*(n+2)+1; sendtype[0] = ROW;
recvcount[0] = n; // lower halo in
recvdisp[0] = (n+1)*(n+2)+1; recvtype[0] = ROW;
//...
sendcount[2] = 1; // left column out
senddisp[2] = 1*(n+2)+1; sendtype[2] = COL;
//...
sendcount[4] = 1; // left-upper corner out
senddisp[4] = 1*(n+2)+1; sendtype[4] = COR;
// ...
// byte offsets
for (i=0; i<t; i++) senddisp[i] *= sizeof(double);

Cart_alltoallw_init(
  matrix,sendcount,senddisp,sendtype,
  matrix,recvcount,recvdisp,recvtype,cart);

while (iterate) {
  // compute ...

  // update
  Cart_alltoallw(
    matrix,sendcount,senddisp,sendtype,
    matrix,recvcount,recvdisp,recvtype,cart);
}
```

structure, see, e.g., the discussion in [16]. The example shows that in order to avoid communication out of intermediate buffers, an own datatype is required for each neighbor process. The `Cart_alltoallw` is therefore needed. For the same reason, we think that `Cart_allgatherw` would be an important addition to MPI.

2.2 MPI Cartesian Collective Communication

In order to support Cartesian Collective Communication in MPI, the information that the provided t-neighborhoods are isomorphic has to be communicated to the MPI library implementation which can then invoke the corresponding, efficient algorithms (see Section 3).

One possibility, as chosen here, is to provide an explicit communicator creation function `Cart_neighborhood_create` which stores all required information, including the algorithm selection as part of a new communicator type. The proposed helper functionality, as well as an `MPI_Neighbor_allgatherw` operation, will be beneficial as well, and complement missing functionality in the MPI standard. Cartesian reduction operations could also be considered as discussed in [16].

It is, however, also possible to rely almost entirely on already existing MPI functionality and support Cartesian Collective Communication without change to the MPI interface. A Cartesian neighborhood (set of identical t neighborhoods for the p MPI processes) defines a virtual communication topology, and can therefore be represented as a *distributed graph topology* in MPI. The distributed graph topology is created from the lists of actual neighbors for all processes by calling either `MPI_Dist_graph_create` or `MPI_Dist_graph_create_adjacent` on the d-dimensional Cartesian communicator used to define the isomorphic neighborhoods. For instance, the `Cart_neighbor_get` function (Listing 2) returns the list of neighbors in the format required for the `MPI_Dist_graph_create_adjacent` call. The MPI library implementations of `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent` can easily detect and reconstruct the t-neighborhood for each process and perform the algorithm selection at communicator creation time. First, by a broadcast of the number of neighbors, t, from some root process, all processes check whether they have the same number of neighbors. If this is the case, the root process broadcasts its relative t-neighborhood in some sorted order, and the processes check whether the isomorphic condition holds (the own neighborhood must be identical to the neighborhood of the root). If that is the case, the specialized algorithms for Cartesian Collective Communication of the next section can be employed and preselected. No additional information (e.g., in the info object) need to be provided. The check is obviously cheap, broadcast of information of size $O(t)$.

By this observation, a logical step to clean up the MPI standard would be to entirely disallow neighborhood communication for Cartesian communicators, which would remove the discrepancies between Cartesian and distributed graph communicators discussed in the introduction. Since the signatures of the MPI neighborhood collectives are identical to the standard collectives, a next logical step would be to eliminate the specific neighborhood collective interfaces by overloading the standard collective interfaces, and let the semantics depend on the type of communicator (standard or distributed graph communicator). Such overloading is already done for collective operations on inter-communicators [11, Section 6.6] and is thus in the spirit of the standard. Doing this would necessitate a new function `Dist_graph_comm` for getting a plain communicator without neighborhood information from a distributed graph communicator, though, in order to be able to perform global collective operations also on distributed graph topologies. If the distributed graph communicator was created from a Cartesian communicator, this plain communicator should also be Cartesian and thus support the coordinate helper functionality defined for Cartesian communicators.

3 ALGORITHMS FOR CARTESIAN COLLECTIVE COMMUNICATION

By the requirement that all processes in a Cartesian Collective have the same t-neighborhoods (given as lists of relative coordinate vectors), it is straightforward to implement the sparse allgather and alltoall communication operations in t send-receive communication rounds. An implementation is shown in Listing 4, and uses the

Listing 4: Trivial, pseudo-MPI, t-round implementation of the Cartesian alltoall operation using the relative shift operation.

```
int Cart_alltoall(sendbuf,...,recvbuf,...,comm)
{
  for (i=0; i<t; i++) {
    // N[i] i'th relative target vector
    Cart_relative_shift(comm,N[i],
                        &source,&target);
    MPI_Sendrecv(sendbuf[i],...,target,
                 recvbuf[i],...,source,comm);
  }
}
```

coordinate shift function defined in Listing 2 to find the source and target ranks for each communication round. The implementation is correct and deadlock-free by the assumption on neighborhoods. The ith target $N[i]$ of the calling process with coordinates R is the process with coordinates $R + N[i]$, and this process will have $(R + N[i]) - N[i] = R$ as it's ith source.

The trivial algorithm always takes at most t communication rounds (blocks for neighbors where $N[i] = (0, 0, \ldots, 0)$ are just copied). Depending on the structure of the neighborhood, this can sometimes be reduced by combining data blocks that "travel in the same direction", similar to message-combining algorithms for alltoall communication as presented in, e.g., [3, 17]. Although this may increase the number of times a data block is sent, it can reduce the number of communication rounds (significantly, as will be seen), and thus save considerably in collective completion time when the data blocks are small.

In the message-combining algorithms that will be developed in the following, we use a straightforward, coordinate-wise path expansion to route data blocks to their target. Let $N[i] = (n_0, n_1, \ldots, n_{d-2}, n_{d-1})$ be the ith relative neighbor in a t-neighborhood N. We route the data block to $N[i]$ in d communication rounds via the intermediate, relative processes $(n_0, 0, \ldots, 0, 0)$, $(n_0, n_1, \ldots, 0)$, $\ldots$, $(n_0, n_1, \ldots, n_{d-2}, 0)$. Thus, the data block $N[i]$ may be sent up to d times, namely for each coordinate $n_j \neq 0$. The message-combining algorithms work in d phases corresponding to the d dimensions with an extra phase for data blocks destined to the process itself (if $(0, 0, \ldots, 0)$ is part of the t-neighborhood N). In phase k, each process r will receive data blocks from processes having r as target for for their kth coordinate, that is from all relative processes $(0, 0, \ldots, -n_k, \ldots, 0)$. Process r likewise sends data blocks to the processes with relative coordinate $(0, 0, \ldots, n_k, \ldots, 0)$. Each phase has as many communication rounds as there are different, non-zero kth coordinates n_k in N which can be determined by bucket (counting) sorting the neighborhood in each phase in order of the kth coordinate. If the coordinates n_k are not too large (bounded by some constant), this takes $O(t)$ time per phase for a total of $O(dt)$ operations which is proportional to the size of the neighborhood. Since all processes have the same relative t-neighborhoods, they all execute the exact same sequence of send-receive communication rounds.

In the two next subsections we work out the (implementation) details for the alltoall and the allgather Cartesian collectives. As already implicitly assumed, we allow bidirectional, send-receive communication between any processes at a cost that is proportional to the size of the data blocks being sent and received. The analysis of the message-combining algorithms focus only on the number *communication rounds* needed, since each send-receive operation incurs a non-negligible latency (start-up time), and the *volume* of data, especially the number of times each individual data block is sent.

Listing 5: Executing a $d+1$ phase schedule. The datatypes for the communication rounds over all phases are stored in the sendtype and recvtype arrays, the ranks of the intermediate processes in the sendrank and recvrank arrays.

```
for (i=0,k=0; k<d+1; k++) {
  MPI_Request request[2*phase[k]];
  for (j=0; j<phase[k]; j++) {
    MPI_Irecv(MPI_BOTTOM,1,recvtype[i],
              recvrank[i],CARTTAG,
              cartcomm,&request[2*j]);
    MPI_Isend(MPI_BOTTOM,1,sendtype[i],
              sendrank[i],CARTTAG,
              cartcomm,&request[2*j+1]);
    i++;
  }
  MPI_Waitall(j,request,MPI_STATUSES_IGNORE);
}
```

The algorithms precompute *communication schedules* as rounds of send-receive communication operations for each of the $d + 1$ communication phases. The blocks to be sent and received in each round are grouped together, stored in MPI datatype arrays, and communicated as a single unit, without any need for explicit packing or unpacking of blocks in contiguous buffers. The ranks of the neighboring processes for each round are likewise stored in arrays, and together the rank and datatype arrays represent the schedule, together with the division of the rounds into phases. This zero-copy execution of a schedule is shown in Listing 5. Since all communication operations within a phase are independent, non-blocking communication is used. Schedule precomputation is intended to be done explicitly by the initialization operations.

The algorithms establish the following result.

Proposition 3.1. *Let N be a d-dimensional t-neighborhood. Correct, message-combining schedules for alltoall and allgather Cartesian Collective Communication with d communication phases and possibly one non-communication phase for performing local data block copies can be computed in $O(td)$ operations, locally per process without any communication.*

The schedule computation time is optimal, since a t-neighborhood in d dimensions takes $O(td)$ space.

3.1 Cartesian Alltoall

Message-combining Cartesian alltoall instantiates the outline given above as follows. Each process has an individual data block for each relative neighbor $N[i]$ in the t-neighborhood. The block to $N[i]$ is sent along as many intermediate processes (hops) as there are non-zero coordinates in $N[i]$. We let $N[i]_k$ denote the kth coordinate

Algorithm 1 Computation of message-combining alltoall schedule for a process with coordinate R represented as sequence of send- and receive ranks and sets of data blocks of size m. Data blocks alternate between a temporary buffer and the receive buffer given in the alltoall call depending on the number of hops of each block; for brevity we assume that blocks are initially stored in the temporary buffer. A last non-communication phase may be needed for copying a block from send buffer to receive buffer if the zero-vector is part of the neighborhood, and is not shown here. The function TypeApp appends a block description consisting of the address of the block and its number of elements to a user-defined datatype.

```
1:  procedure AlltoallSchedule(d, t, N, m)
2:      for i = 0, 1, ..., t − 1 do
3:          hops[i] ← z_i
4:      end for
5:      j, V ← 0, 0                                  ▷ Round and volume count
6:      for k = 0, 1, ..., d − 1 do
7:          BucketSort(t, N, k, order)               ▷ on coordinate k
8:          for i = 0, 1, ..., t − 1 do
9:              i' ← order[i]                        ▷ Neighbors in sorted order
10:             if N[i']_k ≠ 0 then
11:                 if odd(hops[i']) then
12:                     TypeApp((sendtype[j], (tempbuf[i'], m))
13:                     TypeApp((recvtype[j], (recvbuf[i'], m))
14:                 else
15:                     TypeApp((sendtype[j], (recvbuf[i'], m))
16:                     TypeApp((recvtype[j], (tempbuf[i'], m))
17:                 end if
18:                 hops[i'] ← hops[i'] − 1          ▷ One hop further
19:                 if i = t − 1 ∨ N[i']_k ≠ N[order[i + 1]]_k then
20:                     sendrank[j] ← R + N[i']^0_k
21:                     recvrank[j] ← R − N[i']^0_k
22:                     j ← j + 1                    ▷ Next round
23:                 end if
24:                 V ← V + 1
25:             end if
26:         end for
27:         phase[k] ← j                             ▷ Next phase
28:         if k > 0 then
29:             phase[k] ← phase[k] − phase[k − 1]
30:         end if
31:     end for
32: end procedure
```

of $N[i]$, $0 \leq k < d$, and $N[i]^0_k$ a vector of zero coordinates, except for the kth coordinate which is equal to $N[i]_k$. We let z_i be the number of non-zero coordinates (hops) in $N[i]$. Block i is always kept in a buffer indexed by i, that is in a communication round in phase k where $N[i]_k \neq 0$, each processor sends and receives a block for position i. In order to avoid having to copy blocks in and out of the same receive buffer, two buffers are maintained such that a process in a communication rounds sends block i from one buffer and receives block i into the other buffer. The buffers are chosen such that block i initially starts from the sendbuffers of the processes, alternates between a temporary and the receive buffer, and eventually ends up at the correct position i in the receive buffer.

The algorithm for computing the message-combining alltoall schedule is shown as Algorithm 1. In each communication round for phase k, all blocks in the neighborhood having the same kth coordinate are sent and received together. Packing and unpacking these blocks together explicitly is avoided by recording the block indices in an MPI derived, structured datatype (a datatype describing a layout consisting of several blocks of possibly different structure and number of elements). The function TypeApp appends a new entry consisting of a start address and a block size to a structured datatype, here either `sendtype`$[i]$ or `recvtype`$[i]$. In the written-out MPI program, these datatype are all committed by the end of the schedule construction such that the can be used in the execution of the schedule shown in Listing 5. Note that each process sends and receives the same number of data blocks in each communication round when executing the schedule.

The algorithm also computes the *per process communication volume* V of the schedule which is the number of times blocks from a process are sent over intermediate processes before ending up at the target neighbor. By symmetry, the same number of blocks are received for each process, and the total communication volume in number of blocks is pV, and pVm in number of units (Bytes).

For the trivial algorithm shown in Listing 4, the communication volume is $V = t$.

Proposition 3.2. *Let N be a d-dimensional t-neighborhood, z_i the number of non-zeroes in $N[i], 0 \leq i < t$, and C_k the number of different, non-zero kth coordinates in $N, 0 \leq k < d$. The message-combining, Cartesian alltoall schedule runs in d communication phases with $C = \sum_{i=0}^{d} C_i$ communication rounds, and per process communication volume $V = \sum_{i=0}^{t-1} z_i$.*

The message combining algorithm can be faster than the trivial algorithm when $C < t$, and the communication volume per process Vm for data blocks of size m is not too large. If this is not the case, it is better to use the trivial algorithm.

If we assume simple, linear communication costs with latency α and transfer time per unit β, the message-combining schedule is preferable if $C\alpha + \beta Vm < t(\alpha + \beta m)$, that is if $m < \frac{\alpha}{\beta}\frac{t-C}{V-t}$. This cut-off threshold can be computed from the t-neighborhood N in $O(dt)$ time as part of the schedule computation.

Example. Consider a d-dimensional neighborhood consisting of all vectors where each coordinate can be either of n different values including zero, e.g., for $n = 3$ with values $\{-1, 0, 1\}$ and $d = 3$, a 27-point stencil or *Moore neighborhood* of radius one [15]. The neighborhood is described by $t = n^d$ coordinate vectors. The message-combining, per process alltoall communication volume is $V = \sum_{j=1}^{d} j(n-1)^j \binom{d}{j}$: There are $(n-1)^j \binom{d}{j}$ vectors with j non-zero coordinates, and for each of these the data block travel j hops. Concrete volumes for some d and n values are computed in Table 1.

3.2 Cartesian Allgather

In the Cartesian allgather operation, the same data block has to be routed to all neighbors in the t-neighborhood. Consider a single process. In the first phase, the block is sent to processes along the first coordinate in as many rounds as there are different first coordinates.

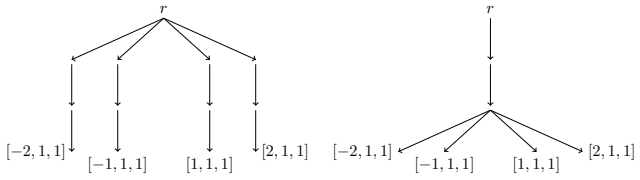


Figure 2: Allgather trees for three dimensional 4-neighborhood consisting of relative offsets $[-2,1,1]$, $[-1,1,1]$, $[1,1,1]$, $[2,1,1]$ as constructed in either increasing (left) or decreasing (right) coordinate order.

In the second round the block is sent along the second coordinate, in the third round along the third coordinate, and so forth. Thus, the communication pattern for a single process is a rooted tree over intermediate processes. The *per process communication volume* is the number of edges in the tree. In contrast to the alltoall operation, the communication volume depends on the dimension order in which the tree is constructed. Figure 2 shows two possible trees for the 3-dimensional neighborhood $N = [(-2,1,1),(-1,1,1),(1,1,1),(2,1,1)]$, one constructed in increasing dimension order, the other in decreasing order. The communication volumes are $V = 12$ and $V = 7$, respectively. Let again C_k be the number of different kth coordinates in the vectors of the given t neighborhood. Without claim of optimality, we construct the allgather trees in order of increasing C_k values, and thus prefer the right tree in Figure 2.

Algorithm 2 Recursive construction of the sorted allgather tree for t-neighborhood N (in increasing dimension order). The function is called starting from dimension $k = 0$.

```
1:  function AllgatherTree(d, t, N, k)
2:      BucketSort(t, N, k, N')                    ▷ On coordinate k
3:      T ← CreateNode(t, N')
4:                          ▷ Tree node with neighbors for all subtrees
5:      if k < d then
6:          s ← 0                                          ▷ First child
7:          for i = 0, 1, ..., t − 1 do
8:              if i = t − 1 ∨ N'[i]_k ≠ N'[i + 1]_k then
9:                  T' ←
10:                 AllgatherTree(d, i + 1 − s, N'[s, ..., i], k + 1)
11:                 AddChild(T, T')
12:                 s ← i + 1
13:             end if
14:         end for
15:     end if
16:     return T
17: end function
```

The message-combining Cartesian allgather schedule routes the blocks from all processes at the same time, with the same tree used for each process. Then, when a process sends a block destined to some $N[i]$, the process will likewise receive a block with the same index i for which the process is an intermediate process. Tree routing is done in the same dimension order as used for the tree construction. At the beginning of phase k each process will have collected blocks destined to all subtrees at level k, $k = 0, 1, \ldots, d-1$, and these blocks are sent along the next dimension in as many rounds as there are different, non-zero kth coordinates in the neighborhood. Let $N[i]_k$ be a kth coordinate representing the root of some subtree. This block is sent together with all other subtree root blocks j for which $N[j]_k = N[i]_k$. Finding all blocks with the same kth coordinate for a round in phase k can be done in $O(t)$ time if the edges of each tree node are kept in sorted dimension order. This can be accomplished by bucket sorting edges as the tree is constructed and can be done in $O(dt)$ total work. The tree construction algorithm shown as Algorithm 2 indicates how. Here, tree construction is done in increasing dimension order; the sorting of the dimensions in increasing C_k order would have to be done in advance, and does not change anything essential.

The schedule can now be computed by a breadth-first traversal of T. At level k, the blocks to be sent in each round are picked by examining all children at that level, grouping those together having the same kth coordinate, and for each such child choosing a representative block index i. This can be done in $O(t)$ time per level by scanning through the neighbors attached to the tree nodes at that level. The block indices for sending and receiving in that round are these representative indices. As in the alltoall schedule, all such indices for a round are put together in send- and receive datatypes, and alternation between a temporary and the receive buffer is done similarly to avoid explicit packing and unpacking. The number of hops that a block travels is the depth of the subtree where the block is first chosen as representative.

All in all, the complete allgather schedule can be constructed in $O(dt)$ operations for any t-neighborhood N as claimed in Proposition 3.1.

Proposition 3.3. *Let N be a d-dimensional t-neighborhood, and C_k the number of different kth coordinates in N, $0 \leq k < d$. Let T be the allgather tree for N. The message-combining, Cartesian allgather schedule runs in d communication phases with $C = \sum_{i=0}^{d} C_i$ communication rounds, and per process communication volume V equal to the number of edges in T.*

Example. Consider again the Moore neighborhood in d dimensions with $n-1$ different, non-zero coordinates in each dimension. The message-combining allgather communication volume is $V = \sum_{j=1}^{d}(n-1)^j\binom{d}{j} = n^d - 1$: Each data block sent to a neighbor that can be reached in j hops, of which there are $(n-1)^j\binom{d}{j}$, is sent to one or more neighbors one hop further. Concrete volumes for some values of d and n are shown in Table 1. The allgather communication volume is much smaller than for alltoall (by a factor j in each term of the sum). Also notable is that the message-combining per process communication volume matches exactly the number of rounds for the trivial algorithm, and thus the message-combining algorithm can be expected to be the faster than the trivial algorithm, regardless of the block size m.

3.3 Irregular Cartesian Alltoall and Allgather

The structure of the communication schedules for both alltoall and allgather does actually not depend on the sizes of the blocks, but solely on the structure of the neighborhood, e.g., the values z_i and C_k. In all rounds, all processes send and receive the same number of blocks with the same indices, meaning that in each round, each

process sends and receives the same amount of data, irregardless of whether the blocks all have the same or possibly different sizes. The schedule computation described therefore immediately works also for the irregular operations. In that case, different rounds within a phase may have different amounts of data, and if all rounds in a phase are executed concurrently (with non-blocking send and receive operations), the round with the largest amount of data will dominate.

3.4 Optimality of Schedules

The message-combining algorithms use a simple, dimension-wise path expansion to route the blocks, with the number of rounds determined by the total number of intermediate coordinates in these paths $C = \sum_{i=0}^{d} C_i$. Other path expansions might lead to smaller number of rounds, and/or a smaller volume, depending on the given neighborhood. Finding a set of basis vectors such that each $N[i]$ can be written as a sum of some of the basis vectors and such that, e.g., $\frac{t-C}{V-t}$ is maximized, seems a hard optimization problem. We have not attempted to solve this problem here and rely on the set of basis vectors $N[i]_k^0$.

With the path expansion chosen here, the alltoall schedule is optimal for non-overlapping target blocks, since the volume V does not depend on the dimension order. This is not the case for the allgather schedule, where the number of edges in the constructed tree depends on the dimension order. Our construction explores the dimensions in order of increasing C_k, but we do not know whether this is optimal. Trying all possible orders would take $O(d!(td + c))$ and thus not be feasible for very large d. Also, we do not know whether the problem is actually NP-hard for arbitrary t-neighborhoods.

When the target blocks are overlapping, as is the case for the corners in the stencil communication pattern shown in Figure 1, the alltoall schedule is not optimal in volume of communicated units, since overlapping parts (corners) will be sent multiple times. In the stencil case, a better schedule would be a combination of one irregular alltoall schedule for rows and columns plus four allgather schedules for the corners. Expressing this in an MPI like interface would require either new, very complex interfaces, or a way to combine schedules in the form of more expressive setup functions. The representation of schedules as arrays of datatypes and ranks would make such a combination both easy and execution efficient.

4 EXPERIMENTAL RESULTS

We have implemented the interface for Cartesian Collective Communication as described in Section 2 with both the trivial, t communication round and the $d + 1$ communication phase, message-combining algorithms described in Section 3[1] We aim to demonstrate the advantages of both interface and algorithms by comparing to the MPI neighborhood collective communication operations, in both blocking and non-blocking variants, when using the same neighborhoods.

4.1 Experimental Setup

4.1.1 Test Cases. We use neighborhoods generalizing the message exchanges required in d dimensional, 3^d-point stencils, e.g., 9-point, 27-point, etc. stencil computations. Neighborhoods are parameterized by their dimension d (we experiment with $d = 2, 3, 4, 5$), its number of neighbors per dimension n (we experiment with $n = 3, 4, 5$) and its offset f which gives the relative offset of the first neighbor in each dimension (we experiment with $f = -1$, which leads to asymmetric neighborhoods for $n = 4, 5$). As an example with $d = 2$, taking $n = 3$, $f = -1$ would specify the 9-point Moore neighborhood $[(-1,-1),(-1,0),(-1,1),(0,-1),(0,0),(0,1),(1,-1),(1,0),(1,1)]$, and taking $n = 4$ would add neighbors $(-1,2)$, $(0,2)$, $(1,2)$, $(2,-1)$, $(2,0)$, $(2,1)$, $(2,2)$ and make the neighborhood asymmetric (and non-Moore).

The message-combining algorithms are expected to have most effect for small block sizes, where latency is dominating, and gradually become worse than the trivial algorithms as the block size increases, and depend strongly on the amount of message-combining that can be done (rounds and volume). We report results for data blocks of size $m = 1, 10, 100$; the unit is `MPI_INT`. To be able to estimate the cut-off block size where message-combining becomes worse than the trivial algorithm, the number of communication rounds for both algorithms (t and C), as well as the communication volumes V for the alltoall and allgather message-combining algorithms, and the ratio $\frac{t-C}{V-t}$ are shown in Table 1 for all stencils.

4.1.2 Systems and Data Processing. The first set of experiments has been conducted on *Hydra*, an Intel Skylake-OmniPath cluster with 36 32-core nodes. On this cluster, we have used Open MPI 3.1.0 and Intel MPI 2018. We also report results from 1024 16-core nodes on the Cray *Titan* system with Cray MPI, see Table 2.

For comparing the performance of the different collective operations, we measure the time of each operation in isolation for a given number of repetitions. Since the recorded times are of different scale, we normalize all numbers to the default, blocking MPI neighborhood call. Finally, we compute the mean and the 95% confidence interval over the measurements (cf. Appendix A).

On *Hydra*, we repeated the measurement for each collective 100 times for $m = 1$, 30 times for $m = 10$, and 10 times otherwise. This worked well with Open MPI and Intel MPI, if we disabled the shared memory device (`shm`) of Intel MPI. With `shm` enabled, the first 15–20 measurements were much slower than the following ones. For consistency, we only used the OmniPath fabric. On *Titan*, however, we had to increase the repetition count to 300, 50, and 40, respectively, due to very large variations in the measured times. We provide more details on which measurements are reported in Appendix A.

4.2 Results

Figures 3–5 depict the results for the `Cart_alltoall` operation normalized to the time of `MPI_Neighbor_alltoall`. Even for the small $d = 3, n = 3$ neighborhood, the message-combining implementation can outperform the MPI neighborhood collectives, as can be seen from the result for $m = 100$ (for $m = 1$, the times are so small and close that the relative distance is not significant). As the number of neighbors per dimension increases to $n = 5$, the improvements for $m = 1, m = 10$ become even more prominent, whereas for $m = 100$ the larger volume start to dominate. For the large $d = 5, n = 5$ neighborhood, the differences are huge for the Open MPI and Intel MPI cases, a factor of more than, e.g.,

[1] All implementations can be found at https://github.com/parlab-tuwien/mpi-cartesian-collectives.

Table 1: Number of communication rounds, communication volume, and cut-off threshold for the message-combining alltoall algorithm (m should be smaller than a constant α/β depending on the communication network times the cut-off ratio). Note that for the neighborhoods used here, the allgather message-combining communication volume exactly matches the volume for the trivial algorithm, but the number of rounds is exponentially smaller.

d	2			3			4			5		
n	3	4	5	3	4	5	3	4	5	3	4	5
$t = n^d - 1$	8	15	24	26	63	124	80	255	624	242	1023	3124
$C = d(n-1)$	4	6	8	6	9	12	8	12	16	10	15	20
Allgather V	8	15	24	26	63	124	80	255	624	242	1023	3124
Alltoall V	12	24	40	54	144	300	216	768	2000	810	3840	12500
Alltoall $\frac{t-C}{V-t}$	1.167	1.250	1.133	0.778	0.688	0.646	0.541	0.477	0.443	0.411	0.358	0.331

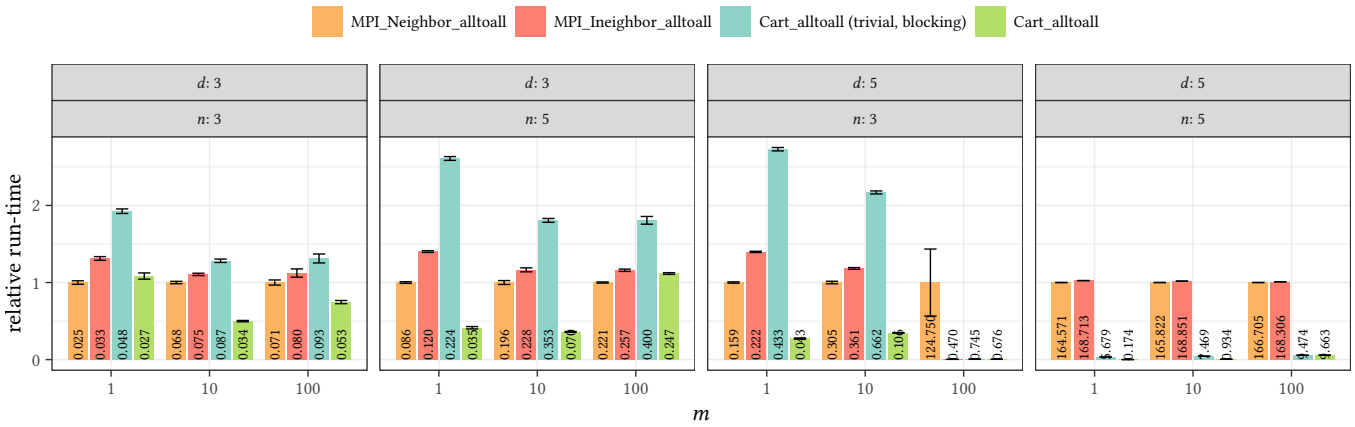


Figure 3: Relative performance of trivial and message-combining `Cart_alltoall` implementations. Baseline: `MPI_Neighbor_alltoall`; absolute run-times given in ms; 36×32 **processes, Open MPI 3.1.0, *Hydra*.**

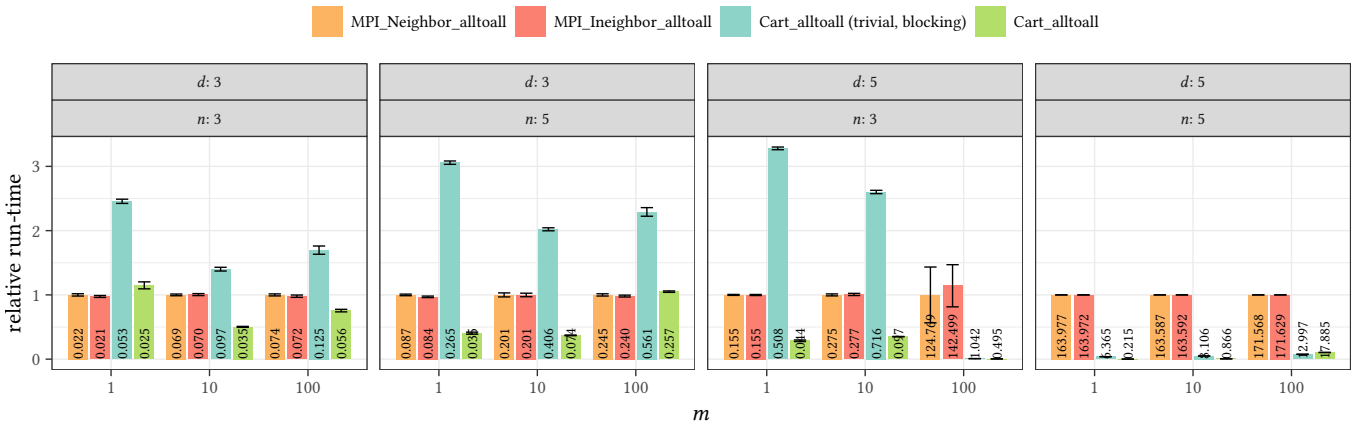


Figure 4: Relative performance of trivial and message-combining `Cart_alltoall` implementations. Baseline: `MPI_Neighbor_alltoall`; absolute run-times given in ms; 32×32 **processes, Intel MPI 2018, *Hydra*.**

17 for Open MPI for $m = 100$. This rather indicates a problem with the MPI library implementations of `MPI_Neighbor_alltoall` which has to do with both the number of neighbors and the block size, and can be seen clearly in Figure 4: For $d = 5, n = 3$ with $m = 10$, the improvement with the message-combining algorithm is a reasonable factor of 3, but with $m = 100$ the factor is 250, and for $d = 5, n = 5$, where the number of neighbors has increased from $3^5 = 243$ to $5^5 = 3125$, the factor is 190 for $m = 10$. The behavior of the trivial algorithm for `Cart_alltoall` of Listing 4 implemented with blocking MPI send-receive calls is, surprisingly,

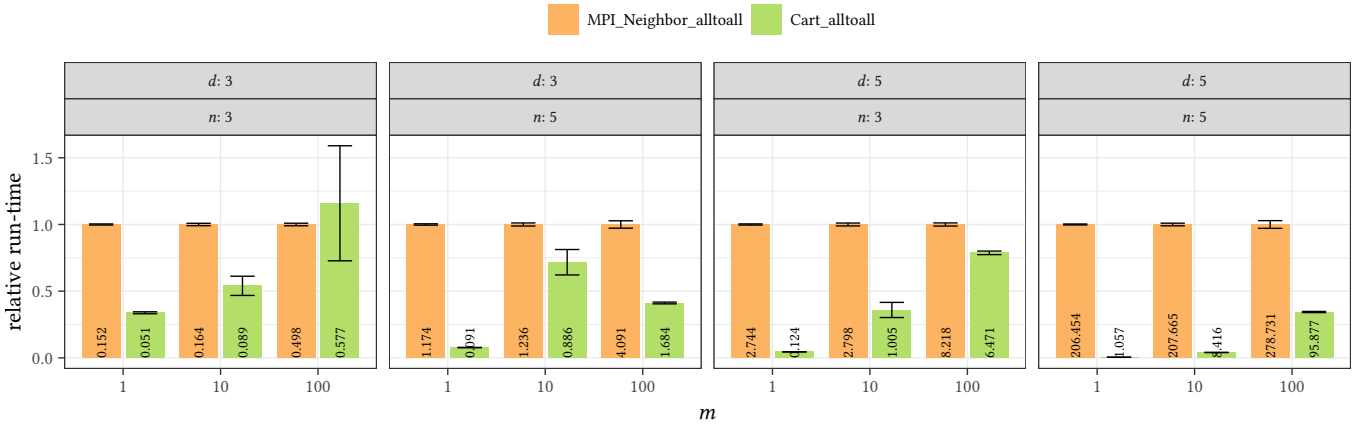


Figure 5: Relative performance of trivial and message-combining `Cart_alltoall` implementations. Baseline: `MPI_Neighbor_alltoall`; absolute run-times given in ms; 1024×16 **processes, Cray MPI, *Titan*.**

Table 2: Systems used in the experiments.

Name	Hardware	MPI Libraries	Compiler
Hydra	36 × Dual Intel Xeon Gold 6130 (16 cores) @ 2.1 GHz, Intel OmniPath	Open MPI 3.1.0 Intel MPI 2018	gcc 6.3.0 icc 18.0.5
Titan	Cray XK7, Opteron 6274 (16 cores) @ 2.2 GHz, Cray Gemini	cray-mpich/7.6.3	PGI 18.4.0

a factor of 2 to 3 slower than the library `MPI_Neighbor_alltoall`. The non-blocking `MPI_Ineighbor_alltoall` is a little slower than the blocking `MPI_Neighbor_alltoall` operation for Open MPI, except for the case $d = 5, n = 3, m = 100$ which may correspond better to the performance that MPI should deliver. For Intel MPI, blocking and non-blocking neighborhood collectives are on par. The results for Cray MPI in Figure 5 are more in line with our expectations, showing an improvement with the message-combining algorithm of a factor of 3 for $d = 5, n = 5$ with $m = 100$.

The results for the `Cart_allgather` and the irregular `Cart_alltoallv` operations can be seen in Figure 6 for the large $d = 5, n = 5$ neighborhoods. Again, the result on *Hydra* with Open MPI for the `MPI_Neighbor_allgather` operation is problematic (much too high). The interesting observation here is the improvement of the message-combining allgather algorithm over the trivial implementation by a factor of about 3 with $m = 100$. For the irregular `Cart_alltoallv` operation, the block sizes are chosen to resemble the different amounts of data for the different neighbors, e.g., corners and row/columns in Figure 1. Let $m = 1, 10, 100$ be the basic block size as in the regular experiments. The size of each concrete data block is chosen to be $m^{(d-z)}$ for a neighbor vector having z non-zero coordinates, $0 \leq z \leq d$, except when $z = 0$ for which the block size is set to 0 (no data to the process itself). The result with Cray MPI shows a large message-combining improvement of a factor of 6 with $m = 10$.

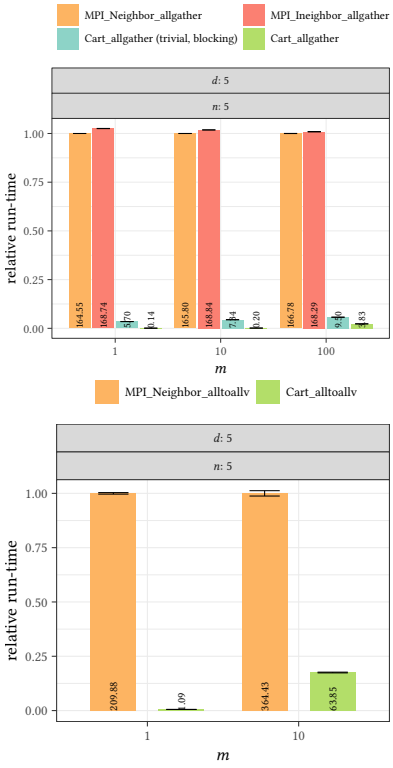
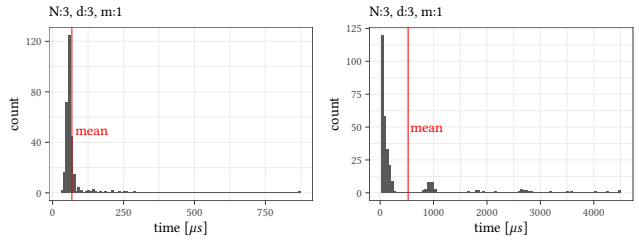


Figure 6: Relative performance of trivial and message-combining `Cart_allgather` (top) and `Cart_alltoallv` (bottom) implementations. Baseline: `MPI_Neighbor_allgather` or `MPI_Neighbor_alltoallv`; absolute run-times given in ms; **left:** 36×32 **processes, Open MPI 3.1.0, *Hydra*; right:** 1024×16 **processes, Cray MPI, *Titan*.**

5 SUMMARY

We introduced Cartesian Collective Communication to express collective communication with arbitrary (isomorphic) stencil patterns. This significantly extends on the limited capabilities of neighborhood communication on MPI Cartesian communicators. Our library implementation consists in essentially one new function not



(a) 128 × 16 **processes** **(b)** 1024 × 16 **processes**

Figure 7: Histograms showing the distribution of run-times for `Cart_alltoall` obtained on *Titan*.

in MPI for creating Cartesian Collective Communication graphs, but we also indicated how Cartesian Collective Communication could be supported in an MPI library implementation essentially without change to the MPI specification. We presented message-combining algorithms for both alltoall and allgather type Cartesian collectives based on dimension-wise message forwarding, which we implemented on top of MPI. Our experiments show that message-combining can lead to significant performance improvements for small(er) Cartesian Communication problems. We think that the idea of specifying (identical) neighborhoods relative to some underlying regular structure other than d-dimensional tori or meshes can be generalized, prove useful for implementing other kinds of stencil computations, and likewise give rise to good, message-combining algorithms. As outlined, combining schedules to avoid communicating overlapping data blocks (corner and row/columns in Figure 1) would be a worthwhile, non-trivial addition to the interface that could possible lead to further performance improvements.

ACKNOWLEDGMENTS

This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] Protonu Basu, Mary W. Hall, Samuel Williams, Brian van Straalen, Leonid Oliker, and Phillip Colella. 2015. Compiler-Directed Transformation for Higher-Order Stencils. In *International Parallel and Distributed Processing Symposium, (IPDPS)*. 313–323.

[2] Rajesh Bordawekar, Alok N. Choudhary, and J. Ramanujam. 1996. Automatic Optimization of Communication in Compiling Out-of-Core Stencil Codes. In *10th International Conference on Supercomputing (ICS)*. 366–373.

[3] Jehoshua Bruck, Ching-Tien Ho, Schlomo Kipnis, Eli Upfal, and D. 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.

[4] S. Mahdieh Ghazimirsaeed, Seyed H. Mirsadeghi, and Ahmad Afsahi. 2019. An Efficient Collaborative Communication Mechanism for MPI Neighborhood Collectives. In *33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 781–792.

[5] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. 2014. *Using Advanced MPI*. MIT Press.

[6] William D. Gropp. 2019. Using Node and Socket Information to Implement MPI Cartesian Topologies. *Parallel Computing* 85 (2019), 98–108.

[7] Torsten Hoefler and Steven Gottlieb. 2010. Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient Using MPI Datatypes. In *Recent Advances in Message Passing Interface. 17th European MPI Users' Group Meeting (Lecture Notes in Computer Science)*, Vol. 6305. Springer, 132–141.

[8] Torsten Hoefler, Rolf Rabenseifner, Hubert Ritzdorf, Bronis R. de Supinski, Rajeev Thakur, and Jesper Larsson Träff. 2011. The Scalable Process Topology Interface of MPI 2.2. *Concurrency and Computation: Practice and Experience* 23 (2011), 293–310.

[9] Torsten Hoefler and Timo Schneider. 2012. Optimization principles for collective neighborhood communications. In *IEEE/ACM Conference on High Performance Computing Networking, Storage and Analysis (SC)*. 98:1–98:10.

[10] Seyed Hessamedin Mirsadegh, Jesper Larsson Träff, Pavan Balaji, and Ahmad Afsahi. 2017. Exploiting Common Neighborhoods to Optimize MPI Neighborhood Collectives. In *24th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 348–357.

[11] MPI Forum. 2015. *MPI: A Message-Passing Interface Standard. Version 3.1.* www.mpi-forum.org.

[12] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. 2015. Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model. In *29th ACM International Conference on Supercomputing (ICS)*. 207–216.

[13] Young-Joo Suh and Kang G. Shin. 2001. All-to-All Personalized Communication in Multidimensional Torus and Mesh Networks. *IEEE Transactions on Parallel and Distributed Systems* 12, 1 (2001), 38–59.

[14] Young-Joo Suh and Sudhakar Yalamanchili. 1998. All-To-All Communication with Minimum Start-Up Costs in 2D/3D Tori and Meshes. *IEEE Transactions on Parallel and Distributed Systems* 9, 5 (1998), 442–458.

[15] Tommaso Toffoli and Norman Margolus. 1987. *Cellular Automata Machines: A New Environment for Modeling.* MIT Press.

[16] Jesper Larsson Träff, Felix Donatus Lübbe, Antoine Rougier, and Sascha Hunold. 2015. Isomorphic, Sparse MPI-like Collective Communication Operations for Parallel Stencil Computations. In *22nd European MPI Users' Group Meeting (EuroMPI)*. ACM, 10:1–10:10.

[17] 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.

[18] Charles Yount, Alejandro Duran, and Josh Tobin. 2019. Multi-level spatial and temporal tiling for efficient HPC stencil computation on many-core processors with large shared caches. *Future Generation Computer Systems* 92 (2019), 903–919.

A PROCESSING OF MEASUREMENT DATA

It turned out to be non-obvious how to present the performance results. At first, we intended to show either the median or the mean value of all measurements. The problem was that both measures were very unstable for two reasons: (1) some measurements were large outliers (e.g., 1000 times larger than the smallest measurement), which often makes the mean less informative, and (2) the measurements sometimes follow a bimodal distribution, where both modes are significantly large, and thus, the median would sometimes jump. Since it is hard to trace the root cause of the problem (e.g., network congestion, cross-cabinet traffic, issues with the MPI implementation, or the use of non-blocking communication, etc.), we decided to take only a subset of the measurements. In particular, we report data only for both the first and the second quartile for *Hydra*, as on this machines the values were commonly stable. On *Titan*, however, we report averages only on the smallest third of all measurements. Figure 7 shows two histograms presenting the distribution of measurements on *Hydra*. We can observe in Figure 7a that for 128 × 16 processes the variance is relatively small. In contrast, Figure 7b presents one case where the variance is significantly larger than before, this time with 1024 × 16 processes. We contend that this problem is not caused by the algorithmic structure, but by the way the run-time system and the parallel machine execute our algorithm. It seems that our algorithm is sensitive to system noise when running on a larger number of compute nodes.