Collectives and Communicators: A Case for Orthogonality

(Or: How to get rid of MPI neighbor and enhance Cartesian collectives)

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

Guillaume Mercier
Bordeaux Institute of Technology, CNRS, Bordeaux INP
Inria, LaBRI, UMR 5800
Talence, France
guillaume.mercier@bordeaux-inp.fr

Daniel J. Holmes
EPCC, University of Edinburgh
Edinburgh, Scotland, UK
d.holmes@epcc.ed.ac.uk

ABSTRACT

A major reason for the success of MPI as the standard for large-scale, distributed memory programming is the economy and orthogonality of key concepts. These very design principles suggest leaner and better support for stencil-like, sparse collective communication, while at the same time reducing significantly the number of concrete operation interfaces, extending the functionality that can be supported by high-quality MPI implementations, and provisioning for possible future, much more wide-ranging functionality.

As a starting point for discussion, we suggest to (re)define communicators as the sole carriers of the topological structure over processes that determines the semantics of the collective operations, and to limit the functions that can associate topological information with communicators to the functions for distributed graph topology and inter-communicator creation. As a consequence, one set of interfaces for collective communication operations (in blocking, non-blocking, and persistent variants) will suffice, explicitly eliminating the `MPI_Neighbor_` interfaces (in all variants) from the MPI standard. Topological structure will not be implied by Cartesian communicators, which in turn will have the sole function of naming processes in a (d-dimensional, Euclidean) geometric space. The geometric naming can be passed to the topology creating functions as part of the communicator, and be used for the process reordering and topological collective algorithm selection.

Concretely, at the price of only 1 essential, additional function, our suggestion can remove 10(+1) function interfaces from MPI-3, and 15 (or more) from MPI-4, while providing vastly more optimization scope for the MPI library implementation.

EuroMPI/USA 2020, 2020, Austin, Texas, USA

ACM ISBN 2…$15.00
https://doi.org/1

ACM Reference Format:
Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, and Daniel J. Holmes. 2020. Collectives and Communicators: A Case for Orthogonality: (Or: How to get rid of MPI neighbor and enhance Cartesian collectives). In *European MPI*. ACM, New York, NY, USA, 8 pages. https://doi.org/1

1 INTRODUCTION

A major reason for the success of MPI as the standard for large-scale, distributed memory programming is the economy and orthogonality of key concepts. Indeed, this economy and orthogonality is what makes the otherwise large (and growing) standard conceptually manageable and useful for practitioners. Orthogonality of concepts prevent proliferation of concrete interfaces, and the discussion in this paper outlines opportunities for actually deprecating and removing concrete interfaces from the MPI standard without sacrificing any functionality. On the contrary, the proposals provide for new functionality by completing so far undefined cases for existing interfaces, and for possible, future extensions way beyond current MPI support. Consequences for current MPI users are very modest, especially since neighborhood collectives is a recent addition to the MPI standard with a limited legacy [3, 4].

The discussion is based on the following observations. Collective communication and reduction operations are defined for "normal", fully connected communicators with the standard interfaces `MPI_Bcast`, etc. [4, Chapter 5]. For communicators with attached virtual topologies, a smaller set of collective operations with similar intentions as their counterparts for standard, intra-communicators are defined through special, concrete interfaces, `MPI_Neighbor_allgather`, etc. [4, Section 7.6]. For the bi-partite, socalled inter-communicators [4, Section 6.6], collective operations are defined and expressed through the same interfaces as for intra-communicators, but the concrete semantics is determined by the bi-partite structure of the inter-communicator (local and remote groups) [4, Chapter 5]. As can be seen, there are two schools of thought at work here. We argue for opting for one, specifically for letting *the structure of the communicator be the sole determiner of the semantics of the collective interfaces* as for the intra/inter-communicator case, and as a consequence suggest to eliminate the concrete neighbor collective interfaces from the standard (but not the functionality). Applications would have to replace `MPI_Neighbor_allgather` with `MPI_Allgather` etc., and to observe a certain

discipline in the use of communicators in contexts where neighbor collective communication is being used.

In MPI, as it is, communicators carrying process topology information come in two flavors with inconsistent expressive and semantic power, namely Cartesian communicators and distributed graph communicators [4, Chapter 7]. Topology information serves several purposes:

- It defines the process neighborhoods for the (neighbor) collective operations and determines the cost and complexity of the neighbor collectives.
- Neighborhoods are ordered, and the order of the neighbors of each process locally determines the placement of data buffers for the neighbor collectives.
- It reflects application communication patterns that can be used for optimizing process placement and ordering.
- Cartesian communicators place (name) processes in a geometric, d-dimensional point-space, and implicitly define specific, unweighted, regular process neighborhoods with a fixed order.
- Distributed graph communicators can define unrestricted weighted process neighborhoods (to reflect prevalence, "distance", frequency, volume, …, but unspecified by the MPI standard) with weights that can be taken into account in process reordering and selection of algorithms for neighborhood collective communication.

Cartesian and distributed graph communicators have different expressive power. For Cartesian communicators, defined via `MPI_Cart_create`, communication edges are for instance unweighted and undirected. Cartesian communicators in d dimensions express implicitly only the neighborhoods containing the nearest $2d$ Manhattan distance one MPI processes. These are the neighborhoods often used in 5-point stencil (in $d = 2$ dimensions, sometimes called D2Q5), 7-point stencil (in $d = 3$ dimensions, sometimes called D3Q7), in general $2d + 1$ stencil codes. The restriction to these $2d$ neighborhoods prevents the use of Cartesian communicators for implicitly and efficiently defining the neighborhoods of more complex stencils. Also, their implied, fixed order of neighbors and communication buffers can be ill-fitting with the actual data layouts of stencil-codes. Most of these problems and discrepancies are well known and have been pointed out often [6–8].

For Cartesian and distributed graph communicators, topological information does not restrict or forbid any communication between processes. Two processes that are not neighbors in either type of communicator can still communicate. This is different for inter-communicators, where both point-to-point and collective communication is restricted to be between processes in different (local and remote, from each process' point of view) groups. We actually argue for a stricter interpretation of topological information, and propose to forbid communication between processes that are not adjacent in the topological structure of the communicator. Such restrictions could (easily) be enforced by the MPI library, and might actually be a useful restriction for debugging and ensuring correctness. A strict interpretation would necessitate access to a "fully connected" communicator allowing unrestricted communication when communication between topological "non-neighbors" is needed, and we do argue for providing this functionality (which is partly present in MPI already) in a unified way.

Our contribution in this paper is a consistent proposal of solutions to these problems that strengthens the orthogonality of basic concepts in MPI. We argue for a cleaner separation of concerns, in particular, to separate the naming of processes (in some geometric, Euclidean space) from their topological (connectivity) structure. We argue for *downgrading the Cartesian functionality to a pure naming scheme*, and instead let all neighborhood topological information be handled through distributed graph communicators.

The introduction of the idea of (geometric) naming scheme could be extended beyond current MPI to cover more complex geometries, non-Euclidean geometries, tile patterns, etc., as possibly occurring in and useful for more complex applications. We do not pursue such speculations further in this paper, which focuses on concrete suggestion for current MPI versions.

2 TOPOLOGICAL CONNECTIVITY STRUCTURE

A communicator in MPI represents an ordered set of processes that can communicate with each other, in particular collectively. Standard, *intra-communicators* are "fully connected" in the sense that any process is allowed to communicate with any other process. Bi-partite communicators, in MPI termed *inter-communicators*, consist of two ordered sets of processes, but communication, in particular collective communication (and point-to-point), can only be among processes in different sets. Normal communicators with additional topological information define neighborhoods for neighbor collective communication. A topology describes a typically sparse graph of preferred communication neighbors. Topologies defined via `MPI_Cart_create` are (almost) regular (with corner cases, depending on whether the grid is periodic in some dimensions), whereas topologies defined via `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent` can describe any process communication graph over the set of processes in the communicator. Distributed graph topologies can be weighted and directed, whereas Cartesian topologies cannot.

We suggest to unify terminology and concepts, and to speak of (virtual) *communication topologies*. An intra-communicator has the topology of a *fully connected graph*, where any process can communicate with any other process, and collectives have their well-known and well-defined semantics. A *bi-partite topology* provides the functionality and semantics of an MPI inter-communicator. A *directed graph topology* restricts collective communication to the process neighborhoods (adjacent process neighbors). The neighborhood of a process in a communicator with graph topology is the processes that are adjacent in the topology.

It is an implementation issue that bi-partite topologies are handled by two communicators (local and remote), and graph topologies by information attached to an otherwise normal communicator. We suggest that the topology terminology can cover all three cases (fully connected, bi-partite, graph).

Point-to-point communication on inter-communicators is restricted. The rank in a communication operation refers to a process in the other set of processes, and communication between processes in the same set is not possible on such communicators. If this is

Listing 1: Getting the base communicator for a communicator (communication topology), whether a fully connected, inter- or distributed graph communicator.

```
int Comm_base(MPI_Comm comm, MPI_Comm *basecomm);
```

needed, the communicator for the local group of processes (that to which the process belongs) has to be extracted from the bi-partite topology. There is no such functionality in MPI, but the function `MPI_Comm_group` can return the local group, from which a communicator can be created by the `MPI_Comm_create` functionality. This is tedious. It would be more effective and possibly more efficient to provide and implement functionality to get the communicator for the local group of processes directly.

Let us define the *base communicator* of a topology as follows. The base communicator of a fully connected intra-communicator is the communicator itself. The base communicator of a bi-partite inter-communicator is the communicator for the set of processes to which the process belongs (corresponding to the local group). The base communicator of a graph topology is the intra-communicator that can be constructed from the group of processes in the graph topology. In all three cases, the base communicator of a communication topology is a fully connected intra-communicator. The functionality shown in Listing 1 extracts (a duplicate of) the base communicator for any communication topology. Base communicators preserve the mapping of MPI processes to processors. If reordering has for instance been done when a distributed graph is set up, the base communicator will also have processes mapped in that order.

For bi-partite topologies (inter-communicators), the base communicator as proposed here contains only the processes in the local group which is a proper subset of the total set of processes (in local and remote groups). The purpose of the base communicator is to enable communication between the processes that cannot communicate in the bi-partite communicator, so a base communicator consisting of all processes (in local and remote groups) would be too large. Furthermore, insisting on the base communicator to contain all processes would require a consistent decision on how the processes in local and remote groups are relatively ordered; recall that local and remote groups are process relative and different for different processes (see `MPI_Intercomm_merge`). Lastly, the operation of creating a merged communicator is potentially expensive, and it may not always be desirable to have this communicator explicitly constructed.

The query functionality `MPI_Comm_test_inter` could be deprecated, and instead the query functionality `MPI_Topo_test` should return the value `MPI_INTER` when the the communicator of the calling process is an inter-communicator topology. In line with our suggestions, the value `MPI_CART` should be deprecated, since Cartesian communicators are not carriers of topological information. They only carry naming (geometric) information.

Point-to-point communication on directed graph topologies could likewise be restricted to be among neighbors only (directed, from source to destination), which might actually be helpful for debugging to discover unintended communication. If indeed communication between processes that are not adjacent in the topology is needed and intended, the base communicator must be used, exactly as is now the case for inter-communicators for communication between processes in the same (local) group.

Other types of topologies might be conceivable, e.g., multi-partite communication domains, or regular patterns of processes like triangles, tiled patterns, etc. Supporting structured operations on such topologies by more than the straight-forward directed graph representation is by now beyond MPI.

Our crucial suggestion is to let the *semantics of the collective operation interfaces* (and perhaps also other operations, like point-to-point communication on inter-communicators) be *determined by the communication topology*. The MPI collective interfaces are suggestive of certain communication patterns (one-many, one-to-all, all-to-one, all-to-all, etc.), but the concrete semantics will be depend on the topology of the calling communicator.

On fully connected (intra-communicator) and on bi-partite (inter-communicator) topologies, the semantics of the collectives are already defined in the MPI standard. On directed graph topologies, the neighborhood collectives `MPI_Neighbor_allgather`, `MPI_Neighbor_allgatherv`, `MPI_Neighbor_alltoall`, `MPI_Neighbor_alltoallv`, and `MPI_Neighbor_alltoallw` (and their non-blocking counterparts) should be deprecated: Their functionality will be covered by the corresponding interfaces `MPI_Allgather` etc., with the actual communication pattern determined by the distributed graph of the communicator as now defined in MPI for the neighborhood collectives [4, Section 7.6]. This is possible by the observation that the signatures of the neighborhood collectives and the corresponding intra/inter-communicator collectives are the same.

Since currently no collectives with graph neighborhood semantics are defined for the broadcast, rooted gather/scatter, and reduction patterns, the existing collective interfaces gives rise to new, possibly application relevant collective communication operations on graph topologies. All these operations will be defined as collectives over all processes in the communicator, but strictly non-synchronizing. Here is a list of possible interpretations of these collective patterns for distributed graph topology communicators:

- `MPI_Bcast` broadcasts data from the named root process (all processes must give the same root argument) to the processes adjacent to the root via outgoing edges, and nothing is done for processes not in the neighborhood of the root. Alternatively: undefined.
- `MPI_Gather`, `MPI_Scatter` operations also take place only in the neighborhood of the root. The `MPI_Gather` operation gathers to the root from the adjacent neighbors via incoming edge, and `MPI_Scatter` scatters from the root to adjacent processes via outgoing edges. The order of the data on the root is determined by the order of the neighbors. Alternatively: undefined.
- `MPI_Reduce` similarly to gather/scatter, perform reduction in the neighborhood of the root, but only for commutative operators (or: the order of the neighbors determines the order of the reductions; this is a strong requirement that could potentially limit the set of possible algorithms). Alternatively: undefined.
- `MPI_Allreduce` collectively performs reductions as described for `MPI_Reduce` in the neighborhoods of all processes.

- `MPI_Reduce_scatter_block` has the combined effect of an allreduce (all processes root in their neighborhoods) and concurrent scatter operations with each of the processes as root.
- The semantics of `MPI_Reduce_scatter` are undefined for distributed graph topologies (as currently for bi-partite topologies).
- `MPI_Scan`, `MPI_Exscan` are undefined for distributed graph topologies (as currently for bi-partite topologies).

Straight-forward, point-to-point based implementations of the collectives interfaces with neighborhood topology semantics can easily be given, but we leave this as an exercise at this point. Note that for the proposed semantics for `MPI_Reduce_scatter_block` for graph topologies, `MPI_Reduce_scatter_block` is no longer equivalent to `MPI_Reduce` (to an arbitrary root) followed by an `MPI_Scatter` as is the case for intra- and inter-communicators. Rather, `MPI_Reduce_scatter` block on graph topologies would be equivalent to `MPI_Allreduce` followed `MPI_Alltoall`.

Our suggestions now save 10(+1) concrete interfaces in MPI-3.1 [4] for the blocking and non-blocking neighborhood collectives, and at least 5 more when persistence is added to MPI-4. Scope for potentially useful new functionality (especially `MPI_Allreduce` and `MPI_Reduce_scatter_block`) is introduced at no interface burden.

In stencil applications, the structure of the data for neighbors (up-down, left-right, corners) is often different, which can sometimes be handled in a direct, zero-copy way by the use of derived datatypes. For this reason, we think that `MPI_Neighbor_alltoallw` is especially useful in the stencil context (with our proposal, it the operation will be `MPI_Alltoallw`). Likewise, for the same reasons, an `Allgatherw` operation could be relevant, but this interface is currently not in MPI. It could be worthwhile to consider this extension to the MPI standard, despite the scalability and other (argument complexity) drawbacks that the `MPI_Alltoallw` operation has.

2.1 Advice to Users

When both collective neighborhood communication and standard, fully connected collectives are to be used in the same part of an application, the user must distinguish and use the appropriate communicator. The `Comm_base` call (cf. Listing 1) extracts a fully-connected base communicator from a communicator with distributed graph or bi-partite topology. Also, unrestricted point-to-point and one-sided communication between any two processes may only be allowed on the fully connected (base) communicator.

2.2 Advice to Implementers

A drawback of the suggestion is that the decoding of communicator topology type may be on the critical path for all communication operations. Explicit interfaces for the different types of topologies do not have this drawback, but instead, the number of interfaces increases combinatorially. However, some MPI libraries may already pay this small price, since inter-communicators are part of MPI and need to be checked for almost all communication operations. Therefore, the added overhead for the collectives to decide whether fully-connected or neighborhood or bi-partite, inter-communicator semantics apply should be acceptably small. Alternatively, an MPI implementation could choose to do the decoding at communicator creation time, and preselect the concrete functions implementing the (collective and point-to-point) operations at this time (by copying a function pointer table, for instance). Such an implementation could have a smaller overhead for the concrete operations. Which choice is the better needs to be determined experimentally.

2.3 The Promise of Persistence

Persistence of collective communication operations will be included in the upcoming MPI-4 standard, and can provide handles for amortizing complex algorithm selection and precomputation for the collective operations (for instance schedule computations for neighborhood collective patterns as shown in [7]). Since persistent versions will be added for all collectives including the neighborhood collectives, our proposal again reduces the actual number of new interfaces to be added in MPI-4. Since there is only one set of interfaces for collective operations with semantics determined by the structure of the communicator, only one set of persistent collective interfaces would actually be needed.

3 CARTESIAN PROCESS NAMING

Through the Cartesian process topologies, MPI provides a convenient mechanism for application programmers for organizing and naming MPI processes in arbitrary, Cartesian grids (meshes and tori) of any dimension d. Cartesian communicators are often used in stencil codes, e.g., LAMMPS; see also [2]. Cartesian communicators implicitly define a neighborhood for all processes in the grid, consisting of the immediate grid neighbors in each of the d dimensions. These $2d$ neighbors are implicitly in a fixed (immutable) order. The implicit neighborhoods in their implicit order can be used for performing neighborhood collective communication, e.g., `MPI_Neighbor_alltoallw`, on Cartesian communicators. The implicit neighborhood is also the only application-relevant information that is made (implicitly) available when the MPI library is attempting to perform process reordering for Cartesian communicators, as is possible with the current MPI interface.

Our proposal is to retain only the naming aspect of the Cartesian communicator functionality, but not the implicit definition of a communication topology (neighborhoods for the processes), and also not the option for process reordering. *Cartesian communicators shall henceforth not carry any topological information and in particular not define any implicit neighborhoods for use in neighborhood collective communication. Process reordering is not implied with any use of Cartesian naming functionality.* The following discussion will show why.

The Cartesian naming scheme in MPI was extended with MPI 3.0 [3] to provide support for certain collective communication on regular stencil patterns, by defining the implicit neighborhoods for the processes in the grid, so that neighborhood collectives can be used on a Cartesian communicator. The intention was to support stencil codes via neighborhood collectives [1]. In stencil codes all processes communicate in similar patterns with a small set of neighboring processes that is defined geometrically by referring to an underlying process grid. Stencil communication patterns are (mostly) symmetric, meaning that communication edges between neighboring processes are bi-directed.

Listing 2: Cartesian (geometric) naming function.

```
int Cart_name(MPI_Comm comm,
              int d, int order[], int periods[],
              int coord[]);
```

The neighborhood implicitly defined by `MPI_Cart_create` is the $2d$ neighborhood only. Thus, Cartesian communicators cannot be used to set up the neighborhoods for, e.g., 3^d stencil codes (like D2Q9, D3Q27), deeper stencils, asymmetric stencils, etc. For such patterns, the application programmer has to compute the neighborhoods explicitly, and define the communication topology by calling `MPI_Dist_graph_create_adjacent` or `MPI_Dist_graph_create`. Perhaps even worse, Cartesian communicators define a fixed order of the neighbors, namely dimension-wise, left-right [4, Section 7.6, page 314]. The order of the neighbors determine the order of the data blocks for the neighborhood collectives `MPI_Neighbor_allgather`, `MPI_Neighbor_alltoall`, etc. For some applications, this order may not correspond well to the actual placement of the data for the stencil code, meaning that intermediate copying will be necessary in order to use neighborhood collectives on Cartesian communicators. There is no functionality in MPI to query Cartesian neighborhoods.

Also for the reordering aspect, `MPI_Cart_create` is limited. The implicitly defined communication edges are unweighted, in contrast to the MPI distributed graph topologies. This might be unproblematic for simple, symmetric $2d+1$ stencils, where the communication load between neighbors is often the same. But for 3^d stencils (e.g., 9-point), this is typically not the case, since communication with the "corners" along the diagonals often have less volume than communication along the principal axes. This could have an effect on the best process mapping.

These issues are well known, and have often been discussed [5, 6, 8] and motivate the proposal to define graph communication topologies solely through the distributed graph functionality, `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent`. For stencil codes, the Cartesian naming scheme is indispensable for this.

The proposed naming function shown in Listing 2 like `MPI_Cart_create` takes a number of dimensions, the order (size) of each dimension, the periodicity of each, and attaches this information to the communicator, but unlike `MPI_Cart_create` does not create a new communicator and therefore cannot affect the mapping of the MPI processes to processors. Therefore, no reorder argument is necessary anymore. This function should supersede `MPI_Cart_create`. After calling `Cart_name`, processes have in addition to their rank, also a position in the d-dimensional grid given by a d-dimensional vector of coordinates. The function could take additional arguments for controlling the way processes are given coordinates, that is row-major or column-major. The naming function could be defined as having local completion semantics, but for consistency between processes, using Cartesian naming should be thought of as a (non-synchronizing) collective operation. Implementations might be completely non-synchronizing. We also suggest that the naming function returns immediately the d-dimensional coordinates of the calling processes. If `Cart_name` is called with a mesh/torus smaller than the size of the communicator, some processes will remain unnamed, and this is signalled by returning a coordinate vector with negative entries. Alternatively, it could be required that the size of the grid must match exactly the size of the communicator. Functionality similar to `MPI_Cart_sub`, but with more expressive power could easily be proposed (subgrids with offsets, for instance). We do not make any such concrete proposal here.

Listing 3: Additional Cartesian functionality, from [7].

```
// coordinate vector to rank
int Cart_vector_rank(MPI_Comm commwithcart, int vector[],
                     int *rank);

// rank to coordinate vector
int Cart_vector_coord(MPI_Comm commwithcart, int rank,
                      int vector[]);

// shift along direction vector
int Cart_vector_shift(MPI_Comm commwithcart, int vector[],
                      int *sourcerank, int *destrank);

// neighborhood of vectors to lists of ranks
int Cart_neighborhood(MPI_Comm commwithcart,
                      int t, int neighbors[],
                      int sources[], int destinations[]);
```

It is important to note that such a naming scheme can be implemented transparently and fully on top of MPI, using MPI communicator attributes to cache the necessary information for the processes; as can also more complex, geometric or non-geometric naming schemes that might be helpful in complex applications for processes to navigate and refer to other processes. The same goes for the current Cartesian support functions, and it might be worthwhile investigating whether such (application specific) "naming" should better be implemented in MPI "standard libraries" rather than being part of the MPI standard itself.

The Cartesian naming scheme does not associate implicit neighborhoods with processes. For use in collective communication operations, neighborhoods must be defined for the processes. The Cartesian naming scheme can, however, support this. The idea is that each process describes its (stencil) neighborhood by listing explicitly the d-dimensional vector offsets of its neighbors. Cartesian communicator functionality translates the coordinate lists into lists of ranks which can be used as input to the distributed graph creation routines, most conveniently `MPI_Dist_graph_create_adjacent` (which might be implemented with less overhead than `MPI_Dist_graph_create` in MPI libraries). Therefore, functionality to map between ranks and Cartesian offset vectors is convenient. A generalized shift operation that shifts along a vector (and not only along a principal axis, as does `MPI_Cart_shift`) could be helpful as well. Such interface prototypes are shown in Listing 3. All the functions are local and non-collective. These functions can all be implemented easily with existing MPI Cartesian functionality. Implementations are available, see [7].

The `Cart_neighborhood` function takes a list of t outgoing neighbors given as a flattened list of relative coordinate vectors for the outgoing edges only, and returns the process ranks of the source (incoming) and target (outgoing) neighbors. These lists of processes can be used as input to `MPI_Dist_graph_create_adjacent`. Note that the lists can be permuted if needed by the application into the most convenient order for the given data layout before calling `MPI_Dist_graph_create_adjacent`. This order of the neighboring processes will determine the way data buffers are

Listing 4: Cartesian stencil neighborhood generation functionality for different types of stencils (with possibly additional parameters).

```
int Cart_stencil_neighbors_count(MPI_Comm commwithcart,
                                 int stencil_type, ...,
                                 int *neighbors);

int Cart_stencil_neighbors(MPI_Comm commwithcart,
                           int stencil_type, ...,
                           int sources[],
                           int destinations[]);
```

used in the collectives, see [4, Section 7.6]. Our proposal entirely frees the user from relying on the implicit, fixed orders as now defined by `MPI_Cart_create`, and thus gives much more flexibility to the application programmer for finding efficient data layouts.

For standard stencils like the $2d + 1$ and 3^d stencils often used in applications, convenience functionality for creating such neighborhoods can easily be implemented, either as part of MPI, or in a library on top of MPI. Two such functions are shown in Listing 4. The stencil type parameter selects the kind of stencil, and the functions can return the number of neighbors and the lists of neighbors.

In order to use collective neighborhood communication on Cartesian neighborhoods computed with the above functionality, a communication topology must be defined. Our suggestion is that the only way to do this is by the distributed graph topology functionality `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent`, which are also the only functions that can possibly reorder processes in the resulting communicator with topology information. Thus, the lists of rank neighbors, possibly with associated lists of weights, are given as input to, e.g., an `MPI_Dist_graph_create_adjacent` call.

For the possible reordering, and for optimization of the neighborhood collective functionality, it is important that `MPI_Dist_graph_create_adjacent` (and `MPI_Dist_graph_create`) retains the Cartesian naming of the calling communicator, and that this is inherited on the resulting communicator (where reordering may have been done). The following can be done: If the calling communicator indeed has Cartesian naming, the distributed graph creation functions can easily discover whether all processes have the same kind of stencil neighborhoods, and if so use this for special, structured reordering with potentially better algorithms and results than for unstructured graphs. Also in this case, special collective algorithms for structured stencil communication can be used and preselected by the MPI library [7]. This is the main reason why it is proposed to have the `Cart_name` function as an MPI function. If entirely implemented outside of, on top of MPI, some other means would have to be found to convey this essential structural information to `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent`.

3.1 Advice to Users

Changes to existing code using Cartesian communicators and neighborhood collectives would be very small. Instead of `MPI_Cart_create`, applications now use `Cart_name` to associate the Cartesian naming scheme with the (new) communicator (or `MPI_Cart_create`, alternatively, is defined to do exactly what `Cart_name` does on a new communicator). For reordering, and for enabling the use of neighborhood collective functionality, `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent` *must* be called on a communicator with Cartesian naming. If the standard, fully connected intra-communicator collectives are also to be used in the application, this must be done via the base communicator, which can be extracted with the new `Comm_base` function, which is actually the only essentially new function in our proposal.

3.2 Advice to Implementers

Cartesian collective communication is a special case of neighborhood collective communication on general graphs. For this special case, there exist better algorithms (at least for some problem sizes) with schedules that can be computed efficiently and locally by the processes as has been explored in some detail in [7].

Thus, it is important that the information that a stencil communication graph is used is conveyed to `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent`. If the user calls these creation functions on the communicator with Cartesian naming (all processes must do so!), it is possible to find out whether all processes have structurally similar neighborhoods, since Cartesianness of neighborhoods is an easily checkable property.

To check the Cartesianness of neighborhoods, `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent` do the following. Let d be the number of dimensions in the Cartesianly named communicator, and t the maximal number of processes in any process neighborhood.

(1) Check whether the calling communicator has Cartesian naming. If so, proceed with the analysis.
(2) Process 0 broadcasts the number of its neighbors.
(3) All processes compare their own number of neighbors to that of process 0. If identical, proceed with the analysis.
(4) All processes translate their list of neighbors (sources and destinations) into vector offsets. All processes sort both lists lexicographically (or otherwise uniquely order). All processes compare source and destinations neighbors, each source should be the opposite of its corresponding destination. If so, proceed with the analysis.
(5) Process 0 broadcasts its target neighbors.
(6) All processes compare their own target neighbors to those of process 0. This tells whether the neighborhood is Cartesian.

As can be seen, the analysis to determine whether the underlying communication pattern is Cartesian (stencil) takes only a few broadcast operations of size at most dt integers.

3.3 Example Application Code: Simple Stencil

In Figure 1b, we show an iterated 9-point stencil code with convergence check. The stencil communication is done by a sparse `MPI_Alltoallw` call on the communicator with the associated graph topology (perhaps via special algorithms for this as selected by the MPI library implementation), whereas the convergence check, which is global, is done via the base communicator. Note that using the base communicator (and not the original communicator) is necessary since reordering of the processes may have taken place.

Instead of using `Cart_neighborhood` to compute the lists of neighbor ranks explicitly, the proposed (external) library function

```
MPI_Comm_size(comm,&p);

d = 2; // number of dimensions
MPI_Dims_create(p,d,order);
reorder = ...; // reorder here?
MPI_Cart_create(comm,d,order,period,reorder,&cart);

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

int target[t] = {0,1, 0,-1, -1,0,   1,0,
                 -1,1, 1,1,   1,-1, -1,-1};

int sources[t], destinations[t];

int vector[2];
for (i=0; i<8; i++) {
  MPI_Cart_coords(cart,rank,vector);
  vector[0] += target[2*i];
  vector[1] += target[2*i+1];
  MPI_Cart_rank(cart,vector,destinations[i]);
  MPI_Cart_coords(cart,rank,vector);
  vector[0] -= target[2*i];
  vector[1] -= target[2*i+1];
  MPI_Cart_rank(cart,vector,sources[i]);
}

reorder = ...; // or reorder here?
MPI_Dist_graph_create_adjacent(cart,
  t,sources,MPI_UNWEIGHTED,
  t,destinations,MPI_UNWEIGHTED,
  MPI_INFO_NULL,reorder,&cartcomm);

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);

short iterate = 1;
while (iterate) {
  // compute ...

  // update
  MPI_Neighbor_Alltoallw(
    matrix,sendcount,senddisp,sendtype,
    matrix,recvcount,recvdisp,recvtype,cartcomm);

  // converged?
  local = ...; // local check
  MPI_Allreduce(&local,&iterate,1,MPI_SHORT,MPI_LAND,
                cartcomm);
}
```

(a) CURRENT

```
MPI_Comm_size(comm,&p);
d = 2; // number of dimensions

MPI_Dims_create(p,d,order);
Cart_name(comm,d,order,period);

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

int target[t] = {0,1, 0,-1, -1,0,   1,0,
                 -1,1, 1,1,   1,-1, -1,-1};

int sources[t], destinations[t];

Cart_neighborhood(comm,t,target,
                  sources,destinations);

reorder = ...;
MPI_Dist_graph_create_adjacent(comm,
  t,sources,MPI_UNWEIGHTED,
  t,destinations,MPI_UNWEIGHTED,
  MPI_INFO_NULL,reorder,cartcomm);

Comm_base(cartcomm,&basecomm);

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);

short iterate = 1;
while (iterate) {
  // compute ...

  // update
  MPI_Alltoallw(
    matrix,sendcount,senddisp,sendtype,
    matrix,recvcount,recvdisp,recvtype,cartcomm);

  // converged?
  local = ...; // local check
  MPI_Allreduce(&local,&iterate,1,MPI_SHORT,MPI_LAND,
                basecomm);
}
```

(b) PROPOSED

Figure 1: CURRENT: The stencil example with current MPI. Note that there are two places where process reordering can be suggested to the MPI library. PROPOSED: A modernized 2-dimensional, 9-point stencil computation code on matrices of local order n showing Cartesian naming, neighborhood setup and creation of the graph topology communicator. The stencil updates themselves are done with `MPI_Alltoallw` on the graph topology communicator in each iteration, whereas convergence checking uses `MPI_Allreduce` on the fully connected base communicator. `ROW`, `COL`, and `COR` are MPI datatypes describing rows, columns, and corners of the two dimensional matrix, respectively. Since communication with the different types of neighbors is different, a possibly better process mapping could be achieved by specifying weighted neighborhoods.

`Cart_stencil_neighbors` with a suitable 9-point stencil parameter could have been used.

Figure 1a shows the 9-point stencil code, still using neighborhood collective communication, but with MPI as it is. For specifying the neighborhood, the detour (with current MPI, it is a detour; a 5-point stencil code could be written without [1], but any other stencil would need to take the detour) over `MPI_Dist_graph_create_adjacent` is necessary for defining the neighborhood, which gives two places in the code where the MPI library can attempt a process reordering.

Both places are wrong! Reordering at `MPI_Cart_create` does not have the information that a 9-point (weighted) neighborhood is going to be used. And `MPI_Dist_graph_create_adjacent` may not (implementation dependent) be using the information that neighborhoods and communication are structured along the 9-point stencil. In particular, if `MPI_Dist_graph_create_adjacent` would be called on the initial communicator `comm` (which would be fine if no reordering was done by `MPI_Cart_create`), it would be extremely difficult for the MPI library implementation to detect that a stencil pattern was defined by the distributed graph, and the library would therefore likely not be able to use efficient, message-combining algorithms for the `MPI_Neighbor_alltoallw` operation [7]. The "Advice to users" given in Section 3.1 is very important (even for current MPI, it conveys further, useful information to the distributed graph creation routines). The clumsy computation of sources and destinations rank lists could of course be wrapped in a suitable, application specific library.

3.4 Buffers in Collective Stencil Communication

In the example in Figure 1, the corners of the stencil halo that are being sent to the neighbor processes on the diagonals are overlapping with the halo rows and columns being sent to the neighbors along the principal dimensions. This redundancy/overlap of data is typical in stencil codes. For (very) large stencil communication problems, this overlap might impact performance.

A very high-quality implementation of the `MPI_Alltoallw` collective on distributed graph topology communicators with underlying Cartesian naming could solve the problem. The structure of the neighborhoods and the overlaps in the data would reveal parts of the data that has to be sent to several neighbors, and for these parts of the data use for instance the allgather schedules proposed in [7], together with alltoall communication schedules for the non-overlapping, individual parts of the data. Again, to compute such combined schedules would require the MPI implementation to have full structural information, meaning both the stencil neighborhood, the Cartesian grid, and the data (buffer addresses and datatypes).

4 SUMMARY OF A CONCRETE PROPOSAL

This paper presented a proposal for exploiting the orthogonality of concepts in MPI for cleaner support of sparse collective communication on process neighborhoods, in particular in combination with Cartesian communicators for stencil communication. The gist of the proposal is to let the topological (neighborhood) structure of the communicator determine the semantics of the collective operations through only one set of interfaces. This one set of interfaces thus covers collective communication on "normal", fully connected intra-communicators, inter-communicators, and distributed graph communicators. Cartesian communicators are suggested to be trimmed down to a naming scheme associating processes with the points on a specified grid.

At the price of only 1 essential additional function, our suggestion can remove 10(+1) functions from MPI-3, and (at least) 15(+1) from MPI-4.

Acknowledgments

The work of Daniel J. Holmes was part-funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement 801039 (the EPiGRAM-HS project). The authors thank Bill Gropp, Rolf Rabenseifner, and George Bosilca for insightful past discussions on the issues and suggestions of this paper.

REFERENCES

[1] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. 2014. *Using Advanced MPI.* MIT Press.
[2] William D. Gropp. 2019. Using Node and Socket Information to Implement MPI Cartesian Topologies. *Parallel Computing* 85 (2019), 98–108.
[3] MPI Forum. 2012. *MPI: A Message-Passing Interface Standard. Version 3.0.* www.mpi-forum.org.
[4] MPI Forum. 2015. *MPI: A Message-Passing Interface Standard. Version 3.1.* www.mpi-forum.org.
[5] Christoph Niethammer and Rolf Rabenseifner. 2019. An MPI interface for application and hardware aware Cartesian topology optimization. In *Proceedings of the 26th European MPI Users' Group Meeting (EuroMPI).* 6:1–6:8.
[6] Jesper Larsson Träff. 2003. SMP-Aware Message Passing Programming. In *Eigth International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS), 17th International Parallel and Distributed Processing Symposium (IPDPS).* 56–65.
[7] Jesper Larsson Träff and Sascha Hunold. 2019. Cartesian Collective Communication. In *48th International Conference on Parallel Processing (ICPP).* 48:1–48:11.
[8] 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.