MPI collective communication through a single set of interfaces: A case for orthogonality

Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, Daniel Holmes

MPI collective communication through a single set of interfaces: A case for orthogonality*

Jesper Larsson Träff[a,*], Sascha Hunold[a], Guillaume Mercier[b], Daniel J. Holmes[c]

[a] *TU Wien, Faculty of Informatics, Favoritenstrasse 16/191-4, Vienna, 1040, Austria*
[b] *Bordeaux Institute of Technology, CNRS, Bordeaux INP, Inria, LaBRI, UMR 5800, Talence, F-33400, France*
[c] *Collis-Holmes Innovations Limited, Three Springs Cottage, Earlston, TD4 6AT, Scotland, UK*

Abstract

We present and discuss a unified view of and interface for collective communication in the MPI (Message-Passing Interface) standard that in a natural way exploits MPI's orthogonality of concepts. We observe that the currently separate and different interfaces for sparse and global collective communication can be unified under the global collective communication interfaces, and at the same time lead to leaner and stronger support for stencil-like, sparse collective communication on Cartesian communicators. Our observations not only significantly reduce the number of concrete operation interfaces, but extend the functionality that can be supported by MPI while provisioning for possible future, much more wide-ranging functionality.

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, thereby explicitly eliminating the `MPI_Neighbor_` interfaces (in all variants) from the MPI standard. Topological structure will no longer be implied by Cartesian communicators, which in turn will have the sole role of naming processes in a (d-dimensional, Euclidean) geometric point-space. The geometric naming can be passed to the topology creating functions as part of the communicator, and guide process rank reordering and topological collective algorithm selection. We also explore ramifications of our proposal for one-sided communication.

Concretely, at the price of only one essential, additional function, our suggestion eliminates 10 concrete collective function interfaces from MPI 3.1, and 15 from MPI 4.0, while providing vastly more optimization scope for the MPI library implementation. Interfaces for Cartesian communicators can likewise be simplified and/or eliminated from the MPI standard, as could the general active (post-start) synchronization mechanism for one-sided communication.

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 [2, 3].

The discussion is based on the following observations. Collective communication and reduction operations are defined for "normal", complete, fully connected communicators with the standard interfaces `MPI_Bcast`, etc. [3, 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. [3, Section 7.6]. For the bi-partite, so-called inter-communicators [3, Section 6.6], collective operations are defined and expressed through the same interfaces as for intra-communicators, but the concrete semantics is different, and determined by the bipartite structure of the inter-communicator (local and remote groups) [3, 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. As a consequence, we can eliminate the con-

*This is a revised, substantially extended version of the paper "Collectives and Communicators: A Case for Orthogonality (Or: How to get rid of MPI neighbor and enhance Cartesian collectives)" that appeared at the 27th European MPI Users' Group (Virtual) Meeting in September 2020 [1]

*Corresponding Author

Email addresses: `traff@par.tuwien.ac.at` (Jesper Larsson Träff), `hunold@par.tuwien.ac.at` (Sascha Hunold), `guillaume.mercier@bordeaux-inp.fr` (Guillaume Mercier), `danholmes@chi.scot` (Daniel J. Holmes)

crete neighbor collective interfaces from the MPI standard, and without sacrificing any functionality. On the contrary, the existing interfaces suggest new, additional collective functionality for sparse topologies. Applications that use the sparse neighborhood collectives would only 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 and normal collective communication are being used together. The burden is moderate, and overall costs tolerable since the neighborhood collectives is a recent addition to MPI with little legacy code. By extending the argument to let one-sided communication windows inherit topological structure from the underlying communicator, the concrete interface functions `MPI_Win_start`/`MPI_Win_post` for general, active one-sided synchronization could likewise be rendered superfluous.

In MPI, as it currently is [3], communicators implicitly carry information on the topology of processes, meaning that communicators determine which processes can communicate with which other processes (as in inter-communicators), and suggest preferred directions of communication (as in Cartesian and distributed graph communicators). Our proposal makes the concepts of *process topology* and *topological structure* captured by a communicator explicit. The current process topology information comes in three flavors with inconsistent expressive and semantic power, namely Cartesian communicators and distributed graph communicators [3, Chapter 7], and inter-communicators [3, Section 6.1]. 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 by identifying preferred or most common communication directions between processes.
- Cartesian communicators place (name) processes in a geometric, d-dimensional, discrete 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 [3, Section 7.5.4]) with weights that can be taken into account in process rank reordering and selection of algorithms for neighborhood collective communication.
- Inter-communicators define specific, but different collective communication patterns for the same interfaces as for intra-communicators, and disallow processes within the same group of processes to communicate.

There are several inconsistencies in the way topological information is used, and in the power of the interfaces that more or less explicitly associate topological information with communicators. Cartesian and distributed graph communicators for instance have different expressive powers. For Cartesian communicators, defined via `MPI_Cart_create`, communication edges are unweighted and undirected. Cartesian communicators in d dimensions implicitly define neighborhoods of processes containing only the nearest $2d$ MPI processes with a Manhattan distance of one. These are the neighborhoods often used in 5-point stencil (in $d = 2$ dimensions, sometimes called D2Q5 in Lattice-Boltzmann codes [4]), 7-point stencil (in $d = 3$ dimensions, sometimes called D3Q7), and in general in $2d + 1$ 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, e.g., stencils with all 3^d neighbors with Chebychev distance one (Moore- or D2Q9, D3Q27 neighborhoods [5]). 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 often been pointed out [6, 7, 8].

For Cartesian and distributed graph communicators, topological information neither restricts nor forbids any communication between processes. Two processes that are not neighbors in either type of communicator can still communicate, although communication between neighbors is preferred. Neighborhood relationships is for instance one criterion often used in MPI process rank reordering [9, 10, 11, 12, 13, 14]. 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) process groups. In our proposal, we actually argue for a stricter interpretation of topological information, and suggest 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 useful for debugging and ensuring correctness of applications. Such a strict interpretation necessitates access to a "fully connected" communicator allowing unrestricted communication when communication between topological "non-neighbors" is needed, and we 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 and inconsistencies that strengthens the orthogonality of basic concepts in MPI. We introduce explicitly a notion of *topological process structure* to be associated with communicators, and argue for a cleaner separation of concerns, in particular, to separate the *naming of processes* (in some geometric, Euclidean point-space) from the topological structure. We argue for *downgrading the Cartesian functionality to a pure naming scheme*, and instead to let all Cartesian neighborhood topological information be handled through the topological structure of communicators. We explore ramifications of this separation of concerns for communicator management (`MPI_Comm_split` etc.), and one-sided communication.

We have given prototype implementations of all functional-

ities and interfaces discussed in the paper, which is useful for experimentation and possible extensions to our proposals. Our `mpiortholib.c` library is publically available[1].

Some of the ideas can be extended further. For instance, 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, combinatorial objects, etc., as possibly occurring in and useful for more complex applications. Also the bi-partite inter-communicators could possibly be generalized to multi-partite topologies with possibilities for new, extended (collective) functionality. We do not pursue such speculations further in this paper, which focuses on concrete suggestions for current MPI versions.

2. Communicators and Topological Structure

A communicator in MPI represents an ordered set of processes that can communicate with each other, in particular collectively. Standard, *intra-communicators* are complete, "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, fully connected communicators with additional topological information define neighborhoods for so-called neighbor collective communication through special, dedicated interfaces. A *virtual process topology* describes a typically sparse graph of preferred communication neighbors. Cartesian topologies defined via `MPI_Cart_create` are (almost) regular (with corner cases, depending on whether the Cartesian grid is periodic in some dimensions), whereas distributed graph topologies defined via `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent` can describe arbitrary process communication graphs over the set of processes in the communicator. Distributed graph topologies are directed and can be weighted and even have edges occurring multiple times; in contrast, Cartesian topologies are bi-directed and unweighted.

We suggest to unify terminology and concepts, and to speak of the *topological structure* of the processes in a communicator and of topological structure being carried and implied by communicators. An MPI intra-communicator with no associated virtual topology has the topological structure of an *unweighted, complete, fully connected graph*, where any process can communicate with any other process, and collectives have their well-known and well-defined semantics [3, Chapter 5]. An MPI inter-communicator has the topological structure of an *unweighted, bi-partite graph* and provides the point-to-point and collective functionality and semantics associated with inter-communicators. The processes are partitioned into two groups and communication, both point-to-point and collective, is between processes in different groups; one-sided communication and I/O are not defined on inter-communicators. An intra-communicator with an associated MPI virtual topology has the topological structure of a *weighted, directed graph*, and in our proposal restricts collective communication to the process neighborhoods (adjacent process neighbors). The neighborhood of a process in a communicator with directed graph topology consists of the processes that are adjacent to a process in the directed graph. The restriction of communication to follow the topological structure is new in our proposal, and allow us to eliminate the neighborhood collective interfaces with no loss of functionality or expressivity. We define the *topological structure* that is always associated with a communicator as a possibly weighted, possibly directed (multi-)graph $G = (P, E)$ over a set of processes P with directed, weighted communication edges $(u, v) \in E$ between processes $u, v \in P$ that are permitted to communicate; the weight associated with edge (u, v) can reflect the cost or other property of communication between the two processes. The topological structure graph does not have to be represented explicitly, and should certainly not have to be stored in full at any one process. Current MPI query functionality that makes it possible for a process to look up its adjacent neighbors suggest that neighborhoods (and nothing more) be stored with the processes.

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

Point-to-point communication on inter-communicators is restricted by the MPI interface design. 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 needed, the communicator for the local set 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 collective `MPI_Comm_create` functionality. This is tedious and constraining (and collective). It would be more effective and efficient to provide functionality to get the communicator for the local group of processes directly and non-collectively.

On current MPI communicators with an associated virtual topology, as can be created with either `MPI_Cart_create`, or `MPI_Dist_graph_create_adjacent` and `MPI_Dist_graph_create`, all processes can communicate with each other, although communication between processes that are neighbors in the virtual topology has a vaguely defined, special status as preferred communication. We suggest to allow point-to-point (and by implication one-sided) communication only between processes that are neighbors in the topology, and disallow direct communication between non-neighbors. A high-quality MPI library implementation could easily check for correct usage, which would impose a useful discipline for application programmers and help in debugging for unintended communication.

We propose to let the distributed graph communicator con-

[1] `https://github.com/parlab-tuwien/mpi-ortholib`

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

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

structors MPI_Dist_graph_create_adjacent and MPI_Dist_graph_create be the only means of associating a weighted (or MPI_UNWEIGHTED), directed graph topology with a communicator. The MPI_Cart_create operation and the related Cartesian functionality will thus be purely (naming) convenience functions (that could easily be implemented on top of MPI without having to be part of the standard), and MPI_Cart_create should not carry any reordering imperative. We discuss the Cartesian naming functionality in detail in Section 3.

Let us define the *base communicator* of a communicator as follows. The base communicator of a fully connected intra-communicator is (a duplicate of) 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 of the inter--communicator). The base communicator of a directed graph topology communicator is the fully connected intra-communicator that can be constructed from the group of processes in the graph topology. In all three cases, the base communicator 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 the processes mapped in that order. The call should preferably have local (non-collective) semantics; in current MPI it must unfortunately be implemented collectively with MPI_Comm_group and MPI_Comm_create (see our mpiortholib.c library).

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 terms and thus 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 intended for inter-communicators could be deprecated, and instead the query functionality MPI_Topo_test should return the new value MPI_INTER when the the communicator of the calling process is an inter-communicator topology. In line with our further suggestions in Section 3, the value MPI_CART should be deprecated, since Cartesian communicators are not carriers of topological information. They only carry naming (geometric) information. The value MPI_GRAPH designating a non-distributed, replicated graph should be deprecated for scalability reasons [6, 15].

Our crucial suggestion now is to require that the *semantics of the collective operations and interfaces* (and perhaps also other operations, like point-to-point communication on inter--communicators) be determined by the *topological structure* of the communicators, and to have only one set of interfaces for expressing collective communication patterns. The MPI collective interfaces are suggestive of certain communication patterns (one-to-many, one-to-all, all-to-one, all-to-all, etc.), and can be given concrete semantics depending on the topological process structure. In addition, we suggest that all other communication be restricted by the *topological structure* in the same manner (that is, to adjacent neighbors), extending the restriction that already applies with inter-communicators to all intra-communicators.

On fully connected (intra-communicator) and on bi-partite (inter-communicator) topologies, the semantics of the collectives are already defined in the MPI standard [3, Chapter 5]. 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, MPI_Alltoall etc., with the actual communication pattern and semantics determined by the directed graph topology of the communicator as now defined in MPI for the neighborhood collectives [3, Section 7.6]. This is possible by the observation that the function signatures of the neighborhood collectives and the corresponding intra- and inter-communicator collectives are the same (there is one curious exception to this observation: With MPI_Alltoallw displacements are integers, whereas with MPI_Neighbor_alltoallw they are MPI_Aint address sized integers; probably the latter is the intention, and the former a slip; in MPI 4.0 this has been rectified with the MPI_Alltoallw_c function).

We propose to define only a single set of collective interfaces for MPI, namely the 17 operations currently defined for fully connected intra-communicators. Many of these already have meaning for inter-communicators with bi-partite process topological structure. On directed graph communicators, as defined for instance by MPI_Dist_graph_create_adjacent, the meaning of the MPI_Allgather and MPI_Alltoall and related interfaces will be as now defined for MPI_Neighbor_allgather and MPI_Neighbor_alltoall [3, Section 7.6]. It is important to note that the way both MPI_Neighbor_allgather and MPI_Neighbor_alltoall are defined, in the limit where the virtual graph topology is the complete, fully connected graph spanning all processes in the communicator they will have exactly the same effect as their counterparts MPI_Allgather and MPI_Alltoall.

There is currently no collective operations with directed graph neighborhood semantics defined for the barrier, broadcast, rooted gather/scatter, and reduction and scan patterns. However, the existing collective interfaces give rise to new, possibly applica-

tion relevant collective communication operations on communicators with directed graph topological structure. Adhering to the in-the-limit principle mentioned above, possible interpretations of the collective interfaces are as follows:

- `MPI_Barrier` could be defined to allow a calling process to return and continue as soon as all adjacent, incoming processes have called the `MPI_Barrier` operation. Extending this transitively would be ill-defined and very hard to implement because of potential cyclic dependencies. Alternatively, `MPI_Barrier` could be left undefined for directed graph topologies.
- `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 in the directed graph topology. Nothing is done for processes not in the neighborhood of the root.
- The `MPI_Gather`, `MPI_Gatherv`, `MPI_Scatter`, `MPI_Scatterv` operations also take place only in the neighborhood of the root. The `MPI_Gather` and `MPI_Gatherv` operations gather data blocks to the root from the adjacent neighbors via incoming edges in the directed graph. The `MPI_Scatter` and `MPI_Scatterv` operations scatter data blocks from the root to adjacent processes via outgoing edges in the directed graph. The order of the data on the root is determined by the order of the neighbors. All processes must give the same root argument.
- `MPI_Reduce` similarly to gather/scatter, performs reduction in the neighborhood of the root, but only for commutative operators (alternatively the order of the neighbors determines the order of the reductions, but this is a strong requirement that could potentially limit the set of possible algorithms). Each process in the neighborhood of the root, possibly including the root itself, contributes a block of the same number of elements with the reduced result stored at the root process.
- `MPI_Allreduce` collectively performs reductions as described for `MPI_Reduce` in the neighborhoods of all processes. We give an application in Section 3.4.
- `MPI_Reduce_scatter_block` and `MPI_Reduce_scatter` are undefined for directed graph topologies (as for bi-partite topologies).
- `MPI_Scan`, `MPI_Exscan` are undefined for directed graph topologies (as for bi-partite topologies).

All these operations are defined as collective over all processes in the communicator in the sense that they must be called by all processes with consistent arguments, but do not necessarily imply that processes synchronize in any way. For the rooted collectives, processes that are not in the neighborhood of the root might be able to complete the operations immediately (local completion semantics); but more advanced algorithms could benefit from the collective semantics and use such processes as intermediate or helper processes.

In our template `mpiortholib.c` library, we present straightforward, point-to-point based implementations of the collectives interfaces with neighborhood topology semantics. Such an implementation for `MPI_Gather` is shown in Listing 2.

Our suggestions now save 10 concrete interfaces in MPI 3.1 [3] for the blocking and non-blocking neighborhood collectives plus the `MPI_Comm_test_inter` test function, and 5 more persistent variants in MPI 4.0. Scope for potentially useful new functionality (especially `MPI_Allreduce`) is introduced at no interface burden.

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 communicators for the two use cases. The `Comm_base` call (cf. Listing 1) extracts a fully connected base communicator from a communicator with directed graph or bi-partite topology. Also, unrestricted point-to-point and one-sided communication between any two processes may be allowed only on the fully connected (base) communicator.

In stencil applications, the structure of the data for the 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 [16]. In such cases, the `MPI_Neighbor_alltoallw` interface (with our proposal, the operation will be `MPI_Alltoallw`) is most useful, since each neighbor can be given a different datatype (row or column type, for instance, see the example in Section 3.3). Also for such applications on sparse, directed neighborhoods, an `MPI_Allgatherw` interface with different datatypes could be relevant, but this interface is, possibly for scalability reasons, 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 [15].

2.2. Advice to Implementers

A potential, small performance drawback of the suggestion is that the decoding of the communicator topology type may be on the critical path for all communication operations, as is the case in the straight-forward gather implementation shown in Listing 2. Explicit interfaces for the different types of topologies do not have this drawback, because the decoding has essentially been delegated to the application programmer. However, some MPI library implementations already pay the small decoding price for inter-communicators that need to be checked for in almost all (point-to-point and collective) communication operations. An alternative, used by other MPI library implementations, would be to do the decoding once 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, at the cost of dereferencing a function pointer at each collective operation call. Which alternative to chose is an issue for MPI library implementers.

Listing 2: Implementation of the interface for `MPI_Gather` covering both fully connected, bi-partite and directed graph topologies. A straight-forward gather algorithm is implemented for the directed graph case.

```
int Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
           void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
{
  int commtype;

  MPI_Topo_test(comm,&commtype);

  if (commtype==MPI_UNDEFINED) {
    // standard collective semantics for inter- and intra-communicators
    return
      MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm);
  } else if (commtype==MPI_DIST_GRAPH) {
    // neighborhood collective semantics
    int rank;
    int s, t; // number of sources and targets
    int weighted;
    int i;

    MPI_Dist_graph_neighbors_count(comm,&s,&t,&weighted);

    int source[s];
    int target[t];

    MPI_Dist_graph_neighbors(comm,s,source,MPI_UNWEIGHTED,t,target,MPI_UNWEIGHTED);

    MPI_Comm_rank(comm,&rank);
    if (rank==root) {
      MPI_Aint lb, extent;

      MPI_Type_get_extent(recvtype,&lb,&extent);

      for (i=0; i<s; i++) {
        if (source[i]==root) {
          // local copy
          MPI_Sendrecv(sendbuf,sendcount,sendtype,0,GATHER_TAG,
                       (char*)recvbuf+i*recvcount*extent,recvcount,recvtype,0,GATHER_TAG,
                       MPI_COMM_SELF,MPI_STATUS_IGNORE);
        } else {
          MPI_Recv((char*)recvbuf+i*recvcount*extent,recvcount,recvtype,source[i],GATHER_TAG,
                   comm,MPI_STATUS_IGNORE);
        }
      }
    } else {
      for (i=0; i<t; i++) {
        if (target[i]==root) {
          MPI_Send(sendbuf,sendcount,sendtype,root,GATHER_TAG,comm);
        }
      }
    }
  }

  return MPI_SUCCESS;
}
```

2.3. The Promise of Persistence

Persistence for collective communication operations are included in the MPI 4.0 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 [8]). Since persistent versions are added for all collectives including the neighborhood collectives, our proposal would again reduce the actual number of interfaces by five. With 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.

2.4. Communicator Splitting and Topological Structure

Since communicators are the (sole) carriers of topological structure, all communicator managing operations have to define what happens to the topological structure and how structure is inherited to new communicators. For intra- and inter--communicators (fully connected and bi-partite topologies) this is well-defined in the MPI standard for operations like `MPI_Comm_split`, `MPI_Comm_split_type`, `MPI_Comm_create`, and `MPI_Comm_create_group`. For communicators with directed graph topological structure, we suggest these operations to work as subgraph constructors. A new subcommunicator created by either of these functions from an old communicator will thus again be a communicator with a directed graph topology, and there will be a directed communication edge between two processes if there was an edge between the processes in the old communicator. This conflicts with the current MPI standard: The MPI 3.1 standard states that virtual topology information is inherited with `MPI_Comm_dup` [3, Chapter 6.4.2], but takes no explicit stance for `MPI_Comm_split` etc., except that no cached information is propagated; in MPI 4.0 it is explicit that no virtual topology information is supposed to be propagated to the new communicators.

2.5. One-sided Communication

Communication windows are created over intra-communicators (and not defined for inter-communicators), and it is tempting to give meaning to `MPI_Win_create` also for directed graph communicators. It would be natural to have the same convention as for point-to-point communication, and restrict remote memory communication operations to processes that are neighbors in the topology of the communicator. Also passive synchronization could be restricted to work only among neighboring, adjacent processes, and allow an origin process to lock a target process only if there is a directed edge from origin to target.

On a fully connected topology, the collective `MPI_Win_fence` synchronization operation opens a next epoch with access and exposure for all processes in the window communicator. On a directed graph topology, a natural restriction of `MPI_Win_fence`, still being collective, would be for each process to grant exposure to its incoming neighboring processes and give the process access to its outgoing neighbors in the topology. The `MPI_Win_fence` operation would thus work like a combination of an `MPI_Win_post` for the (group of) incoming neighbors and an `MPI_Win_start` for the (group of) outgoing neighbors. With this convention, `MPI_Win_fence` would include the general active target synchronization mechanism [3, Section 11.5.2] as a special case, except possibly for the `MPI_Win_test` operation. These four (five) synchronization calls could be deprecated.

In relatively static applications using the dedicated `MPI_Win_start`/`MPI_Win_post` synchronization mechanisms where access and exposure groups do not change (often), this might indeed be attractive. In dynamic applications where groups change often, there might be considerable efficiency tradeoffs. With our proposal, new windows with new topologies would have to be created often by collective operations, and this could indeed be significantly more expensive than repeatedly creating the new groups by purely process-local operations.

3. Cartesian Process Naming

Through the Cartesian process topologies, the current MPI standard [3, Chapter 7] 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 and molecular dynamics codes, e.g., GROMACS, NAMD, LAMMPS (see [9] for a selection of codes). 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 predefined, fixed and 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 rank reordering for Cartesian communicators, as is possible with the current MPI interface.

In the terminology of Section 2, the Cartesian communicator creating function `MPI_Cart_create` imposes a bi-directed graph topology structure on the created communicator, consisting of the set of bi-directed neighborhoods described above. 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 rank 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 rank 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 [2] 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, see e.g., [17]. 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 often (mostly) symmetric, meaning that communication edges between neighboring processes are bi-directed.

The neighborhood implicitly defined by `MPI_Cart_create` is the $2d$ so-called von Neumann neighborhood only (see, e.g., [18] for terminology). 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 [3, Section 7.6, page 314]. The order of the neighbors determines the order of the data blocks for the neighborhood collectives `MPI_Neighbor_allgather` and `MPI_Neighbor_alltoall`. 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 for lists of neighbors.

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 [6, 7, 19] 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.

We make the implicit concept of a *process naming* as a one to one correspondence between MPI process ranks in a communicator and elements in a set of the same size, e.g., set of coordinates in a d-dimensional grid, explicit. A naming function is a process-local operation that makes it possible to map between process ranks and names, e.g., coordinates, by associating the necessary map with the communicator. A naming function for Cartesian names can look as shown in Listing 3. Naming schemes are not intended to be inherited by communicator managing functions that create new communicators. Therefore, all naming functions take explicit rank or name arguments, instead of implicitly relying on the rank or name of the calling process.

The naming function `Cart_name` like `MPI_Cart_create` takes a number of dimensions d, the order (size) of each dimension, the periodicity of each, and attaches this information to the communicator `comm`. In addition, it takes a `dimorder`

Listing 3: Cartesian naming functionality for associating a d-dimensional grid with a communicator and providing for one to one mapping between ranks and coordinate vectors. The local query functionality is also shown.

```
int Cart_name(MPI_Comm comm, int d, int dimorder,
              int order[], int period[],
              int *size);

int Cart_test(MPI_Comm cart,
              int *flag, int *d, int *size);
int Cart_get(MPI_Comm cart, int *dimorder,
             int maxd, int order[], int period[]);
```

argument that tells whether processes are ordered row wise (C-like) or column-wise (FORTRAN-like) in the d-dimensional grid. The argument values could be `ROWMAJOR` or `COLMAJOR`, or `MPI_ORDER_C` or `MPI_ORDER_FORTRAN`, as already used for the subarray data type constructors [3, Section 4.1.3 and 4.1.4]. It could furthermore even be allowed to let the `dimorder` (pointer) argument be a permutation array (of size d) giving the order of the dimensions, in the same way that `MPI_Dist_graph_create` allows either the special value `MPI_UNWEIGHTED` or a list of weights as arguments. The function returns the total number of processes that can fit in the grid of $\texttt{size} = \Pi_{i=0}^{d-1}\texttt{order}[i]$ processes. Processes with a rank larger than this `size` will not have a name, and the query functions will return `MPI_PROC_NULL` for such processes. Note that it would be possible (and perhaps convenient for some applications) to allow the `size` of the grid to be larger than the number of processes in the communicators; for some grid points there would be no process. Applying `Cart_name` to an already named communicator may either be disallowed, or simply overwrite the name. In contrast to `MPI_Cart_create`, `Cart_name` does not create a new communicator and therefore cannot affect the mapping of the MPI processes to processors. No reorder argument is therefore necessary anymore. Since `Cart_name` does not create a new communicator object, there is no corresponding free function or destructor. Instead of attaching the naming scheme to the communicator, a new MPI object for naming schemes with specific support for translating between ranks and names could be introduced. We do not do so here.

The `Cart_name` 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. We have defined the naming function as having process-local completion semantics, and thus the function cannot guarantee that all processes in the communicator use the same naming scheme. Doing so is the intended usage, however. Alternatively, and perhaps better, Cartesian naming should be defined or thought of as a (non-synchronizing) collective operation with the requirement that all processes call with the same argument values. If `Cart_name` is called with a mesh/torus smaller than the size of the communicator, some larger ranked processes will remain unnamed. Alternatively, it could be required that the size of the grid must match exactly the size of the communicator.

Two query functions for testing and getting back the size and order of the grid, intended to support MPI library building,

Listing 4: Cartesian naming functionality for mapping between ranks and d-dimensional coordinates. Both absolute and relative mapping functions are defined.

```
int Cart_rank(MPI_Comm cart, int coordinates[],
              int *rank);
int Cart_coordinates(MPI_Comm cart, int rank,
                     int coordinates[]);

int Cart_relative_rank(MPI_Comm cart,
                       int source, int relative[],
                       int *dest);
int Cart_relative_coordinates(MPI_Comm cart,
                              int source,
                              int dest,
                              int relative[]);
int Cart_relative_shift(MPI_Comm cart, int rank,
                        int relative[],
                        int *inrank,
                        int *outrank);
int Cart_allranks(MPI_Comm cart, int n,
                  int coordinates[], int ranks[]);
int Cart_allranks_relative(MPI_Comm cart,
                           int source,
                           int n, int relatives[],
                           int ranks[]);
```

are likewise suggested as also shown in Listing 3.

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. Such an implementation is shown in our `mpiortholib.c` library. 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 can be defined, and likewise implemented on top of MPI. The same applies to the current Cartesian support functions in MPI [3], 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 proposed Cartesian naming scheme makes it possible to map between process ranks in the named communicator and coordinates in the d-dimensional grid. Coordinates are represented as small d-element arrays, and parsed in the `dimorder` (row or column major) defined for the naming. The corresponding functions are shown in Listing 4, and come in both absolute and relative coordinate versions. A relative offset is a d-dimensional vector that is added to an absolute position given as a source rank to get a new absolute position vector. A generalized shift operation `Cart_relative_shift` that shifts along a relative offset vector (and not only along a principal axis, as does `MPI_Cart_shift`) from a given source rank and returns the ranks of incoming and outgoing processes is also defined and could be helpful as well. The function `Cart_allranks` takes a flattened list of absolute coordinates and computes the corresponding list of ranks. This function will be most helpful for computing neighborhoods as input to the MPI functions that create communicators with a directed graph topological structure. The counterpart function `Cart_allranks_relative` takes a list of relative coordinates and computes the list of ranks from the given source rank. The mapping functions are all local and non-collective, and again can easily be implemented on top of current MPI as shown in our `mpiortholib.c` library.

Listing 5: Functionality for using Cartesian naming to guide creation of (unnamed) subcommunicators.

```
Cart_create_sub(MPI_Comm cart,
  int dimension[], MPI_Comm *subcart);
```

Listing 6: Cartesian support functionality for generating relative neighborhoods.

```
int Cart_neighbors_count(MPI_Comm cart, int rank,
                         int metric,
                         int shadow, int depth,
                         int *size);
int Cart_neighbors(MPI_Comm cart, int rank,
                   int metric,
                   int shadow, int depth,
                   int maxsize, int neighbors[]);
```

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 can describe its neighborhood, for instance in a stencil application, by listing explicitly the d-dimensional vector offsets of its neighbors. The Cartesian `Cart_allranks` function translates a coordinate list into a list of ranks which can be used directly 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). The `Cart_allranks_relative` function instead takes a list of relative coordinates, as for instance computed by the stencil generating functions of Listing 6, and a given rank, and computes a list of ranks of neighboring processes.

In MPI a sometimes convenient functionality for creating Cartesian subcommunicators is defined, namely `MPI_Cart_sub`. We propose and implement similar functionality that uses the Cartesian naming associated with a communicator to guide the splitting of the communicator into smaller communicators. This is shown in Listing 5. The idea is to give a Boolean d-element vector that tells which coordinate dimensions are to be excluded from the d-dimensional grid. Since naming schemes are not intended to be inherited when splitting a named communicator, the resulting `subcart` communicators will be unnamed. However, the relative order of the processes will be retained. For this function to work, it is a precondition that all calling processes call with communicators with the same naming scheme. Assume that some dimensions are excluded from a d-dimensionally named communicator, resulting in sets of processes corresponding to a d'-dimensional grid. If a d'-dimensional naming is imposed on these communicators (with the same order for the non-excluded dimensions), then a process rank with d coordinates in the original communicator will have the same coordinates for the remaining d' dimensions in the subcommunicator to which it belongs.

For use in stencil applications, we propose additional functionality that can generate specific process neighborhoods. This is shown in Listing 6. The `Cart_neighbors` function takes as input a metric or distance measure to be used in the neighborhood generation, and returns a list of relative coordinates corresponding to stencil offsets with at most the distance given by the `depth` parameter and at least the distance given by the `shadow` from the given rank. The periodicity of the Cartesian naming can be taken into account in this way. The size in number of processes of such a neighborhood can be computed by the `Cart_neighbors_count` function. Possible distance measures could be Manhattan distance in d dimensions, indicated by `MANHATTAN_DISTANCE`, or Chebychev distance in d dimensions, indicated by `CHEBYCHEV_DISTANCE`. Using Manhattan distance with depth one would give rise to the standard 5-point stencil in two dimensions (D2Q5), in general would produce von Neumann neighborhoods of the given depth. Using instead Chebychev distance, the standard D2Q9 and D3Q27 stencils in two and three dimensions can be generated, in general Moore neighborhoods of the given depth. An example is shown in Listing 7.

The generated list of relative coordinates of the neighborhood can be used as input to the `Cart_allranks_relative` function that translates the list into a list of ranks which in turn can be used as input to the `MPI_Dist_graph_create_adjacent` communicator creation function.

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 used in the collectives, see [3, 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.

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. This eliminates the need for any special, weighted Cartesian communicator creation functions as were suggested for instance in [19].

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`) can access the Cartesian naming of the calling communicator. 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 [12]. Also in this case, special collective algorithms for structured stencil communication can be used and preselected by the MPI library [8]. This is the main reason why it is proposed to have the `Cart_name` function as an interface in the MPI standard. 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`. Note that since the proposed Cartesian naming scheme for translating between ranks and coordinates is oblivious to the calling process' rank, it is not required to carry the Cartesian naming scheme over to the created communicator with directed graph topology. Naming schemes are, as explained, not intended to be inherited by communicator creating and managing functions.

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. 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, namely the case where all processes have the same set of relative, outgoing neighbors [7, 8] (by symmetry, all processes will also have the same set of relative, incoming neighbors). 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 [8].

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 a communicator with Cartesian naming (in which case 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 on periodic grids, `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent` do the following. Let t the maximal number of processes in any process neighborhood in the Cartesianly named communicator.

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. Process 0 broadcasts its target neighbors (ranks).
5. All processes translate their own list of target neighbors and the target neighbors of the root to lists of relative coordinates. Both lists are sorted lexicographically. Each process compares the two lists of relative coordinates. If identical for all processes, the set of neighborhoods is Cartesian.

As can be seen, the analysis to determine whether the underlying communication pattern is Cartesian (stencil) takes only a few broadcast and allreduce operations of size at most t integers (ranks). With a little care, the analysis can be extended to non-periodic grids as well. A concrete implementation as a check predicate can be seen in our `mpiortholib.c` library.

3.3. Example Application Code: Simple Stencil

In Figure 1b, we show an iterated 9-point stencil code with convergence check implemented using our proposal. The stencil communication itself 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.

In contrast, 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 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 rank reordering. With current MPI, it is indeed a detour; a 5-point stencil code could be written without the extra call to `MPI_Dist_graph_create_adjacent` by relying on the implicitly defined neighborhood [17]; but any other stencil would need to take the detour and define the neighborhood explicitly.

Both places for the reordering 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 [8]. 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.

Listing 7: An implementation of Conway's Game of Life using `MPI_Allreduce` on a directed graph topology and implicit generation of the 9-point neighborhoods.

```
int p, pp;
MPI_Comm_size(comm,&p);

int d = 2; // number of dimensions
MPI_Dims_create(p,d,order);
period[0] = 0; period[1] = 0;
Cart_name(comm,d,ROWMAJOR,order,period,&pp);

int r;
MPI_Comm_rank(comm,&r);

int t;
Cart_neighbors_count(comm,r,
                     CHEBYCHEV_DISTANCE,1,1,&t);

int target[d*t];
Cart_neighbors(comm,r,
               CHEBYCHEV_DISTANCE,1,1,t,target);

int ranks[t];
Cart_allranks(comm,r,t,target,ranks);

reorder = 1; // reorder?
MPI_Dist_graph_create_adjacent(comm,
  t,ranks,MPI_UNWEIGHTED,
  t,ranks,MPI_UNWEIGHTED,
  MPI_INFO_NULL,reorder,&cartcomm);

short state, live;

short forever = 1;
while (forever) {
  MPI_Allreduce(&state,&live,1,MPI_SHORT,
                MPI_SUM,cartcomm);
  if (live<2||live>3) state = 0;
  else if (live==3)   state = 1;
}
```

3.4. Example Application Code: Game of Life

The 9-point stencil is also used in Conway's Game of Life cellular automaton [18]. Each cell in the 2-dimensional grid has an alive/dead state, and the state for the next generation is computed from the number of alive, neighboring cells, including the cell itself. Since the sum function is commutative, the order of the neighbors does not matter, so for the sum computation our new, sparse `MPI_Allreduce` collective can be used. The neighborhoods are generated automatically by the `Cart_neighbors` interface function, and since the calling process itself is not to be included in the neighborhood, both `shadow` and `depth` parameters are set to 1. This code is shown in Listing 7.

3.5. 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.

```
MPI_Comm_size(comm,&p);
d = 2; // number of dimensions
MPI_Dims_create(p,d,order);
period[0] = 0; period[1] = 0;
reorder = 1; // reorder here?
MPI_Cart_create(comm,d,order,period,reorder,
                &cart);

double matrix[n+2][n+2];
int t = 8;
int sources[t], destinations[t];
int target[] = { 0,1, 0,-1, -1,0,   1,0,
                -1,1, 1,1,   1,-1, -1,-1};

int vector[d];
for (i=0; i<8; i++) {
  MPI_Cart_coords(cart,rank,2,vector);
  vector[0] += target[2*i];
  vector[1] += target[2*i+1];
  MPI_Cart_rank(cart,vector,&destinations[i]);
  MPI_Cart_coords(cart,rank,2,vector);
  vector[0] -= target[2*i];
  vector[1] -= target[2*i+1];
  MPI_Cart_rank(cart,vector,&sources[i]);
}
reorder = 1; // or reorder here?
MPI_Dist_graph_create_adjacent(cart,
  t,sources,MPI_UNWEIGHTED,
  t,destinations,MPI_UNWEIGHTED,
  MPI_INFO_NULL,reorder,&cartcomm);

// datatypes for matrix borders
MPI_Datatype ROW, COL, COR;

senddisp[0] = 1*(n+2)+n;   sendtype[0] = COL;
recvdisp[0] = 1*(n+2)+n+1; recvtype[0] = COL;
// ...
senddisp[2] = 1*(n+2)+1;   sendtype[2] = ROW;
recvdisp[2] = 1;           recvtype[2] = ROW;
// ...
senddisp[4] = n*(n+2)+1;   sendtype[4] = COR;
recvdisp[4] = (n+1)*(n+2); recvtype[4] = COR;
// ...
// counts and byte offsets
for (i=0; i<t; i++) {
  sendcount[i] = 1; recvcount[i] = 1;
  senddisp[i] *= sizeof(double);
  recvdisp[i] *= sizeof(double);
}

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

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

  int local = 1; // local convergence check
  MPI_Allreduce(&local,&iterate,1,MPI_SHORT,
                MPI_LAND,cartcomm);
}
```

(a) With MPI as is (current).

```
MPI_Comm_size(comm,&p);
d = 2; // number of dimensions
MPI_Dims_create(p,d,order);
period[0] = 0; period[1] = 0;
Cart_name(comm,d,ROWMAJOR,order,period,&pp);

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

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

int r;
MPI_Comm_rank(comm,&r);
Cart_allranks_relative(comm,r,t,target,ranks);

reorder = 1; // reorder?
MPI_Dist_graph_create_adjacent(comm,
  t,ranks,MPI_UNWEIGHTED,
  t,ranks,MPI_UNWEIGHTED,
  MPI_INFO_NULL,reorder,&cartcomm);

Comm_base(cartcomm,&basecomm);

// datatypes for matrix borders
MPI_Datatype ROW, COL, COR;

senddisp[0] = 1*(n+2)+n;   sendtype[0] = COL;
recvdisp[0] = 1*(n+2)+n+1; recvtype[0] = COL;
// ...
senddisp[2] = 1*(n+2)+1;   sendtype[2] = ROW;
recvdisp[2] = 1;           recvtype[2] = ROW;
// ...
senddisp[4] = n*(n+2)+1;   sendtype[4] = COR;
recvdisp[4] = (n+1)*(n+2); recvtype[4] = COR;
// ...
// counts and byte offsets
for (i=0; i<t; i++) {
  sendcount[i] = 1; recvcount[i] = 0;
  senddisp[i] *= sizeof(double);
  recvdisp[i] *= sizeof(double);
}

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

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

  int local = 1; // local convergence check
  MPI_Allreduce(&local,&iterate,1,MPI_SHORT,
                MPI_LAND,basecomm);
}
```

(b) With the proposed, single interface set.

Figure 1: (a) The 9-point stencil code example with current MPI using both the neighbor and the "normal" collective interfaces. Note that there are two places where process rank reordering can be suggested to the MPI library. (b) The 9-point stencil code example with the unified, single interface as proposed here 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. The local matrices all have order n.

A very high-quality implementation of the `MPI_Alltoallw` collective on directed 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 [8], 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). Persistent collectives as defined in MPI 4.0 provide handles for possibly doing such advanced analysis and optimization.

4. Summary of a Concrete Proposal

This paper presented a proposal for exploiting the orthogonality of concepts in MPI to unify different collective patterns under a single set of interfaces, leading to cleaner support for 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 one essential additional function, our suggestion can remove 10 concrete interface functions from MPI 3.1, and 15 from MPI 4.0.

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] J. L. Träff, S. Hunold, G. Mercier, D. J. Holmes, Collectives and communicators: A case for orthogonality (or: How to get rid of MPI neighbor and enhance Cartesian collectives), in: 27th European MPI Users' Group Meeting (EuroMPI/USA), ACM, 2020, pp. 1:1–1:8.

[2] MPI Forum, MPI: A Message-Passing Interface Standard. Version 3.0, `www.mpi-forum.org` (September 21st 2012).

[3] MPI Forum, MPI: A Message-Passing Interface Standard. Version 3.1, `www.mpi-forum.org` (June 4th 2015).

[4] M. Espinoza-Andaluz, A. Moyón, M. Andersson, A comparative study between D2Q9 and D2Q5 lattice boltzmann scheme for mass transport phenomena in porous media, Computers & Mathematics with Applications 78 (9) (2019) 2886–2896.

[5] K. Suga, Y. Kuwata, K. Takashima, R. Chikasue, A D3Q27 multiple-relaxation-time lattice boltzmann method for turbulent flows, Computers & Mathematics with Applications 69 (6) (2015) 518–529.

[6] J. L. Träff, SMP-aware message passing programming, in: Eighth International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS), 17th International Parallel and Distributed Processing Symposium (IPDPS), 2003, pp. 56–65.

[7] J. L. Träff, F. D. Lübbe, A. Rougier, S. Hunold, Isomorphic, sparse MPI-like collective communication operations for parallel stencil computations, in: 22nd European MPI Users' Group Meeting (EuroMPI), ACM, 2015, pp. 10:1–10:10.

[8] J. L. Träff, S. Hunold, Cartesian collective communication, in: 48th International Conference on Parallel Processing (ICPP), 2019, pp. 48:1–48:11.

[9] W. D. Gropp, Using node and socket information to implement MPI cartesian topologies, Parallel Computing 85 (2019) 98–108.

[10] T. Hatazaki, Rank reordering strategy for MPI topology creation functions, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface. 5th European PVM/MPI User's Group Meeting, Vol. 1497 of Lecture Notes in Computer Science, Springer, 1998, pp. 188–195.

[11] E. Jeannot, G. Mercier, F. Tessier, Process placement in multicore clusters: Algorithmic issues and practical techniques, IEEE Transactions on Parallel and Distributed Systems 25 (4) (2014) 993–1002.

[12] K. von Kirchbach, M. Lehr, S. Hunold, C. Schulz, J. L. Träff, Efficient process-to-node mapping algorithms for stencil computations, in: IEEE International Conference on Cluster Computing (CLUSTER), IEEE Computer Society, 2020, pp. 1–11.

[13] J. L. Träff, Implementing the MPI process topology mechanism, in: ACM/IEEE Supercomputing, ACM, 2002, pp. 40:1–40:14.

[14] H. Yu, I.-H. Chung, J. E. Moreira, Topology mapping for Blue Gene/L supercomputer, in: ACM/IEEE Supercomputing, 2006, p. 116.

[15] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, J. L. Träff, MPI on millions of cores, Parallel Processing Letters 21 (1) (2011) 45–60.

[16] T. Hoefler, S. Gottlieb, 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, Vol. 6305 of Lecture Notes in Computer Science, Springer, 2010, pp. 132–141.

[17] W. Gropp, T. Hoefler, R. Thakur, E. Lusk, Using Advanced MPI, MIT Press, 2014.

[18] T. Toffoli, N. Margolus, Cellular Automata Machines: A New Environment for Modeling, MIT Press, 1987.

[19] C. Niethammer, R. Rabenseifner, An MPI interface for application and hardware aware Cartesian topology optimization, in: Proceedings of the 26th European MPI Users' Group Meeting (EuroMPI), 2019, pp. 6:1–6:8.