D R A F T
Document for a Standard Message-Passing Interface

Message Passing Interface Forum

March 20, 2012
This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

Chapter 7

Process Topologies

7.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter 6, a process group in MPI is a collection of `n` processes. Each process in the group is assigned a rank between `0` and `n-1`. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the "virtual topology."

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal [only]with machine-independent mapping and communication on virtual process topologies. ticket258. ticket258.

> *Rationale.* Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [6]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [1, 2].

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and notational power in message-passing programming. (*End of rationale.*)

7.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping.

Specifying the virtual topology in terms of a graph is sufficient for all applications. However, in many applications the graph structure is regular, and the detailed set-up of the graph would be inconvenient for the user and might be less efficient at run time. A large fraction of all parallel applications use process topologies like rings, two- or higher-dimensional grids, or tori. These structures are completely defined by the number of dimensions and the numbers of processes in each coordinate direction. Also, the mapping of grids and tori ticket0. is generally an easier problem [then]than that of general graphs. Thus, it is desirable to address these cases explicitly.

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a (2×2) grid is as follows.

coord (0,0):	rank 0
coord (0,1):	rank 1
coord (1,0):	rank 2
coord (1,1):	rank 3

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.

7.4 Overview of the Functions

The functions MPI_GRAPH_CREATE, MPI_DIST_GRAPH_CREATE_ADJACENT, MPI_DIST_GRAPH_CREATE and MPI_CART_CREATE are used to create general (graph) virtual topologies and Cartesian topologies, respectively. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. For MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical values on all processes of the group of comm_old. For MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE the input communication graph is distributed across the calling processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 6). In analogy to function MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an n-dimensional hypercube is an n-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

Rationale. Similar functions are contained in EXPRESS [3] and PARMACS. (*End of rationale.*)

The function MPI_TOPO_TEST can be used to inquire about the topology associated with a communicator. The topological information can be extracted from the communicator using the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET, for general graphs, and MPI_CARTDIM_GET and MPI_CART_GET, for Cartesian topologies. Several additional functions are provided to manipulate Cartesian topologies: the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa; the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). The function MPI_CART_SHIFT provides the information needed to communicate with neighbors in a Cartesian dimension. The two functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to extract the neighbors of a node in a graph. For distributed graphs, the functions MPI_DIST_NEIGHBORS_COUNT and MPI_DIST_NEIGHBORS can be used to extract the neighbors of the calling node. The function MPI_CART_SUB is collective over the input communicator's group; all other functions are local.

Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation. ticket258.

The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topology associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW.

7.5 Topology Constructors

7.5.1 Cartesian Constructor

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

IN	comm_old	input communicator (handle)
IN	ndims	number of dimensions of Cartesian grid (integer)
IN	dims	integer array of size ndims specifying the number of processes in each dimension
IN	periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
IN	reorder	ranking may be reordered (true) or not (false) (logical)
OUT	comm_cart	communicator with new Cartesian topology (handle)

ticket140.
ticket140.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int *dims, const
              int *periods, int reorder, MPI_Comm *comm_cart)
```

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

{MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[], const bool periods[], bool reorder) const *(binding deprecated, see Section 15.2)* }

MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the Cartesian grid is smaller than the size of the group of [comm]comm_old, then some processes are returned MPI_COMM_NULL, in analogy to MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.

ticket0.

7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE

For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of MPI_COMM_WORLD's group) into an n-dimensional topology.

MPI_DIMS_CREATE(nnodes, ndims, dims)

IN	nnodes	number of nodes in a grid (integer)
IN	ndims	number of Cartesian dimensions (integer)
INOUT	dims	integer array of size ndims specifying the number of nodes in each dimension

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

{`void MPI::Compute_dims(int nnodes, int ndims, int dims[])` *(binding deprecated, see Section 15.2)* }

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions and a total of nnodes nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array `dims`. If `dims[i]` is set to a positive number, the routine will not modify the number of nodes in dimension `i`; only those entries where `dims[i] = 0` are modified by the call.

Negative input values of `dims[i]` are erroneous. An error will occur if `nnodes` is not a multiple of $\prod_{i, dims[i] \neq 0} dims[i]$.

For `dims[i]` set by the call, `dims[i]` will be ordered in non-increasing order. Array `dims` is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is local.

Example 7.1

`dims` before call	function call	`dims` on return
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

7.5.3 General (Graph) Constructor

MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)

IN	comm_old	input communicator (handle)
IN	nnodes	number of nodes in graph (integer)
IN	index	array of integers describing node degrees (see below)
IN	edges	array of integers describing graph edges (see below)
IN	reorder	ranking may be reordered (true) or not (false) (logical)
OUT	comm_graph	communicator with graph topology added (handle)

ticket140. ticket140.

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int *index, const
              int *edges, int reorder, MPI_Comm *comm_graph)
```

```
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
              IERROR)
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER
```

{MPI::Graphcomm MPI::Intracomm::Create_graph(int nnodes, const int index[], const int edges[], bool reorder) const *(binding deprecated, see Section 15.2)* }

MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes. If the size, nnodes, of the graph is smaller than the size of the group of ticket0. [comm]comm_old, then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL is returned in all processes. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the following simple example.

Example 7.2

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

Then, the input arguments are:

nnodes = 4
index = 2, 3, 4, 6
edges = 1, 3, 0, 3, 0, 2

Thus, in C, `index[0]` is the degree of node zero, and `index[i] - index[i-1]` is the degree of node `i, i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges[j]`, for $0 \leq j \leq \mathtt{index}[0] - 1$ and the list of neighbors of node `i`, $i > 0$, is stored in `edges[j]`, $\mathtt{index}[i-1] \leq j \leq \mathtt{index}[i] - 1$.

In Fortran, `index(1)` is the degree of node zero, and `index(i+1) - index(i)` is the degree of node `i, i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges(j)`, for $1 \leq j \leq \mathtt{index}(1)$ and the list of neighbors of node `i`, $i > 0$, is stored in `edges(j)`, $\mathtt{index}(i) + 1 \leq j \leq \mathtt{index}(i + 1)$.

A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.

Advice to users. Performance implications of using multiple edges or a non-symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (*End of advice to users.*)

Advice to implementors. The following topology information is likely to be stored with a communicator:

- Type of topology (Cartesian/graph),
- For a Cartesian topology:
  1. `ndims` (number of dimensions),
  2. `dims` (numbers of processes per coordinate direction),
  3. `periods` (periodicity information),
  4. `own_position` (own position in grid, could also be computed from rank and dims)
- For a graph topology:
  1. `index`,
  2. `edges`,

  which are the vectors defining the graph structure.

For a graph structure the number of nodes is equal to the number of processes in the group. Therefore, the number of nodes does not have to be stored explicitly. An additional zero entry at the start of array `index` simplifies access to the topology information. (*End of advice to implementors.*)

7.5.4 Distributed (Graph) Constructor

The general graph constructor assumes that each process passes the full (global) communication graph to the call. This limits the scalability of this constructor. With the distributed graph interface, the communication graph is specified in a fully distributed fashion. Each process specifies only the part of the communication graph of which it is aware. Typically, this could be the set of processes from which the process will eventually receive or get data, or the set of processes to which the process will send or put data, or some combination of such edges. Two different interfaces can be used to create a distributed graph topology. MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph communicator with each process specifying ticket0. [all]each of its incoming and outgoing (adjacent) edges in the logical communication graph and thus requires minimal communication during creation. MPI_DIST_GRAPH_CREATE provides full flexibility, and processes can indicate that communication will occur between other pairs of processes.

To provide better possibilities for optimization by the MPI library, the distributed graph constructors permit weighted communication edges and take an info argument that can further influence process reordering or other optimizations performed by the MPI library. For example, hints can be provided on how edge weights are to be interpreted, the quality of the reordering, and/or the time permitted for the MPI library to process the graph.

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm_dist_graph)

IN	comm_old	input communicator (handle)
IN	indegree	size of sources and sourceweights arrays (non-negative integer)
IN	sources	ranks of processes for which the calling process is a destination (array of non-negative integers)
IN	sourceweights	weights of the edges into the calling process (array of non-negative integers)
IN	outdegree	size of destinations and destweights arrays (non-negative integer)
IN	destinations	ranks of processes for which the calling process is a source (array of non-negative integers)
IN	destweights	weights of the edges out of the calling process (array of non-negative integers)
IN	info	hints on optimization and interpretation of weights (handle)
IN	reorder	the ranks may be reordered (true) or not (false) (logical)
OUT	comm_dist_graph	communicator with distributed graph topology (handle)

ticket140.
ticket140.
ticket140.

```
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree, const
              int sources[], const int sourceweights[], int outdegree, const
```

```
            int destinations[], const int destweights[], MPI_Info info,
            int reorder, MPI_Comm *comm_dist_graph)
```
ticket140.

```
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
            OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
            COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
        DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
```

{`MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create_adjacent(int indegree, const int sources[], const int sourceweights[], int outdegree, const int destinations[], const int destweights[], const MPI::Info& info, bool reorder) const`*(binding deprecated, see Section 15.2)* }

{`MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create_adjacent(int indegree, const int sources[], int outdegree, const int destinations[], const MPI::Info& info, bool reorder) const`*(binding deprecated, see Section 15.2)* }

MPI_DIST_GRAPH_CREATE_ADJACENT returns a handle to a new communicator to which the distributed graph topology information is attached. Each process passes all information about the edges to its neighbors in the virtual distributed graph topology. The calling processes must ensure that each edge of the graph is described in the source and in the destination process with the same weights. If there are multiple edges for a given (source,dest) pair, then the sequence of the weights of these edges does not matter. The complete communication topology is the combination of all edges shown in the sources arrays of all processes in comm_old, which must be identical to the combination of all edges shown in the destinations arrays. Source and destination ranks must be process ranks of comm_old. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that have specified indegree and outdegree as zero and that thus do not occur as source or destination rank in the graph specification) are allowed.

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_DIST_GRAPH_CREATE_ADJACENT is collective.

Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count arguments of later communication calls along specific edges could be used as their edge weights. Multiplicity of edges can likewise indicate more intense communication between pairs of processes. However, the exact meaning of edge weights is not specified by the MPI standard and is left to the implementation. In C or Fortran, an application can supply the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the same (effectively no) weight. In C++, this constant does not exist and the weight arguments may be omitted from the argument list. It is erroneous to supply MPI_UNWEIGHTED, or in C++ omit the weight arrays, for some but not all processes of comm_old. Note that

MPI_UNWEIGHTED is not a special weight value; rather it is a special value for the total array argument. In C, one would expect it to be NULL. In Fortran, MPI_UNWEIGHTED is an object like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

The meaning of the info and reorder arguments is defined in the description of the following routine.

MPI_DIST_GRAPH_CREATE(comm_old, n, sources, degrees, destinations, weights, info, reorder, comm_dist_graph)

IN	comm_old	input communicator (handle)
IN	n	number of source nodes for which this process specifies edges (non-negative integer)
IN	sources	array containing the n source nodes for which this process specifies edges (array of non-negative integers)
IN	degrees	array specifying the number of destinations for each source node in the source node array (array of non-negative integers)
IN	destinations	destination nodes for the source nodes in the source node array (array of non-negative integers)
IN	weights	weights for source to destination edges (array of non-negative integers)
IN	info	hints on optimization and interpretation of weights (handle)
IN	reorder	the process may be reordered (true) or not (false) (logical)
OUT	comm_dist_graph	communicator with distributed graph topology added (handle)

ticket140.
ticket140.
ticket140.
ticket140.

```
int MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],
              const int degrees[], const int destinations[], const
              int weights[], MPI_Info info, int reorder,
              MPI_Comm *comm_dist_graph)
```

```
MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
              INFO, REORDER, COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
    WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
```

```
{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,
              const int sources[], const int degrees[], const int
              destinations[], const int weights[], const MPI::Info& info,
              bool reorder) const(binding deprecated, see Section 15.2) }
```

```
{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,
              const int sources[], const int degrees[],
```

```
const int destinations[], const MPI::Info& info, bool reorder)
const
```
(binding deprecated, see Section 15.2) }

MPI_DIST_GRAPH_CREATE returns a handle to a new communicator to which the distributed graph topology information is attached. Concretely, each process calls the constructor with a set of directed (source,destination) communication edges as described below. Every process passes an array of n source nodes in the sources array. For each source node, a non-negative number of destination nodes is specified in the degrees array. The destination nodes are stored in the corresponding consecutive segment of the destinations array. More precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the j-th such edge stored in destinations[degrees[0]+...+degrees[i-1]+j]. The weight of this edge is stored in weights[degrees[0]+...+degrees[i-1]+j]. Both the sources and the destinations arrays may contain the same node more than once, and the order in which nodes are listed as destinations or sources is not significant. Similarly, different processes may specify edges with the same source and destination nodes. Source and destination nodes must be process ranks of comm_old. Different processes may specify different numbers of source and destination nodes, as well as different source to destination edges. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that do not occur as source or destination node in the graph specification) are allowed.

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_Dist_graph_create is collective.

If reorder = false, all processes will have the same rank in comm_dist_graph as in comm_old. If reorder = true then the MPI library is free to remap to other processes (of comm_old) in order to improve communication on the edges of the communication graph. The weight associated with each edge is a hint to the MPI library about the amount or intensity of communication on that edge, and may be used to compute a "best" reordering.

Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count arguments of later communication calls along specific edges could be used as their edge weights. Multiplicity of edges can likewise indicate more intense communication between pairs of processes. However, the exact meaning of edge weights is not specified by the MPI standard and is left to the implementation. In C or Fortran, an application can supply the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the same (effectively no) weight. In C++, this constant does not exist and the weights argument may be omitted from the argument list. It is erroneous to supply MPI_UNWEIGHTED, or in C++ omit the weight arrays, for some but not all processes of comm_old. Note that MPI_UNWEIGHTED is not a special weight value; rather it is a special value for the total array argument. In C, one would expect it to be NULL. In Fortran, MPI_UNWEIGHTED is an object like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4

The meaning of the weights argument can be influenced by the info argument. Info arguments can be used to guide the mapping; possible options include minimizing the maximum number of edges between processes on different SMP nodes, or minimizing the sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it is valid for an MPI implementation not to do any reordering. An MPI implementation may specify more info key-value pairs. All processes must specify the same set of key-value info pairs.

Advice to implementors. MPI implementations must document any additionally supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.

An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI_GRAPH_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors.*)

Example 7.3 As for Example 7.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

With MPI_DIST_GRAPH_CREATE, this graph could be constructed in many different ways. One way would be that each process specifies its outgoing edges. The arguments per process would be:

process	n	sources	degrees	destinations	weights
0	1	0	2	1,3	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

Another way would be to pass the whole graph on process 0, which could be done with the following arguments per process:

process	n	sources	degrees	destinations	weights
0	4	0,1,2,3	2,1,1,2	1,3,0,3,0,2	1,1,1,1,1,1
1	0	-	-	-	-
2	0	-	-	-	-
3	0	-	-	-	

In both cases above, the application could supply MPI_UNWEIGHTED instead of explicitly providing identical weights.

MPI_DIST_GRAPH_CREATE_ADJACENT could be used to specify this graph using the following arguments:

process	indegree	sources	sourceweights	outdegree	destinations	destweights
0	2	1,3	1,1	2	1,3	1,1
1	1	0	1	1	0	1
2	1	3	1	1	3	1
3	2	0,2	1,1	2	0,2	1,1

Example 7.4 A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be [modelled]modeled with Cartesian topologies, but can easily be captured with MPI_DIST_GRAPH_CREATE as shown in the following code. In this example, the communication along the dimensions is twice as heavy as the communication along the diagonals: ticket0.

```
/*
Input:     dimensions P, Q
Condition: number of processes equal to P*Q; otherwise only
           ranks smaller than P*Q participate
*/
int rank, x, y;
int sources[1], degrees[1];
int destinations[8], weights[8];

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* get x and y dimension */
y=rank/P; x=rank%P;

/* get my communication partners along x dimension */
destinations[0] = P*y+(x+1)%P; weights[0] = 2;
destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;

/* get my communication partners along y dimension */
destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;

/* get my communication partners along diagonals */
destinations[4] = P*((y+1)%Q)+(x+1)%P; weights[4] = 1;
destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
destinations[6] = P*((y+1)%Q)+(P+x-1)%P; weights[6] = 1;
destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;

sources[0] = rank;
degrees[0] = 8;
MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
                      weights, MPI_INFO_NULL, 1, &comm_dist_graph);
```

7.5.5 Topology Inquiry Functions

If a topology has been defined with one of the above functions, then the topology information can be looked up using inquiry functions. They all are local calls.

MPI_TOPO_TEST(comm, status)

IN	comm	communicator (handle)
OUT	status	topology type of communicator comm (state)

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

```
MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR
```

{`int MPI::Comm::Get_topology() const`*(binding deprecated, see Section 15.2)* }

The function MPI_TOPO_TEST returns the type of topology that is assigned to a communicator.

The output value `status` is one of the following:

MPI_GRAPH	graph topology
MPI_CART	Cartesian topology
MPI_DIST_GRAPH	distributed graph topology
MPI_UNDEFINED	no topology

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

IN	comm	communicator for group with graph structure (handle)
OUT	nnodes	number of nodes in graph (integer) (same as number of processes in the group)
OUT	nedges	number of edges in graph (integer)

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
```

```
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR
```

{`void MPI::Graphcomm::Get_dims(int nnodes[], int nedges[]) const`*(binding deprecated, see Section 15.2)* }

Functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topology information that was associated with a communicator by MPI_GRAPH_CREATE.

The information provided by MPI_GRAPHDIMS_GET can be used to dimension the vectors `index` and `edges` correctly for the following call to MPI_GRAPH_GET.

MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)

IN	comm	communicator with graph structure (handle)
IN	maxindex	length of vector index in the calling program (integer)
IN	maxedges	length of vector edges in the calling program (integer)
OUT	index	array of integers containing the graph structure (for details see the definition of MPI_GRAPH_CREATE)
OUT	edges	array of integers containing the graph structure

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
             int *edges)
```

```
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
```

{void MPI::Graphcomm::Get_topo(int maxindex, int maxedges, int index[],
int edges[]) const *(binding deprecated, see Section 15.2)* }

MPI_CARTDIM_GET(comm, ndims)

IN	comm	communicator with Cartesian structure (handle)
OUT	ndims	number of dimensions of the Cartesian structure (integer)

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

```
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
```

{int MPI::Cartcomm::Get_dim() const *(binding deprecated, see Section 15.2)* }

The functions MPI_CARTDIM_GET and MPI_CART_GET return the Cartesian topology information that was associated with a communicator by MPI_CART_CREATE. If comm is associated with a zero-dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and MPI_CART_GET will keep all output arguments unchanged.

MPI_CART_GET(comm, maxdims, dims, periods, coords)

IN	comm	communicator with Cartesian structure (handle)
IN	maxdims	length of vectors `dims`, `periods`, and `coords` in the calling program (integer)
OUT	dims	number of processes for each Cartesian dimension (array of integer)
OUT	periods	periodicity (true/false) for each Cartesian dimension (array of logical)
OUT	coords	coordinates of calling process in Cartesian structure (array of integer)

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
              int *coords)
```

```
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)
```

{`void MPI::Cartcomm::Get_topo(int maxdims, int dims[], bool periods[], int coords[]) const`*(binding deprecated, see Section 15.2)* }

MPI_CART_RANK(comm, coords, rank)

IN	comm	communicator with Cartesian structure (handle)
IN	coords	integer array (of size `ndims`) specifying the Cartesian coordinates of a process
OUT	rank	rank of specified process (integer)

ticket140.

```
int MPI_Cart_rank(MPI_Comm comm, const int *coords, int *rank)
```

```
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER COMM, COORDS(*), RANK, IERROR
```

{`int MPI::Cartcomm::Get_cart_rank(const int coords[]) const`*(binding deprecated, see Section 15.2)* }

For a process group with Cartesian structure, the function MPI_CART_RANK translates the logical process coordinates to process ranks as they are used by the point-to-point routines.

For dimension `i` with `periods(i) = true`, if the coordinate, `coords(i)`, is out of range, that is, `coords(i)` $<$ `0` or `coords(i)` $\geq$ `dims(i)`, it is shifted back to the interval `0` $\leq$ `coords(i)` $<$ `dims(i)` automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

If comm is associated with a zero-dimensional Cartesian topology, coords is not significant and 0 is returned in rank.

MPI_CART_COORDS(comm, rank, maxdims, coords)

IN	comm	communicator with Cartesian structure (handle)
IN	rank	rank of a process within group of comm (integer)
IN	maxdims	length of vector coords in the calling program (integer)
OUT	coords	integer array (of size ndims) containing the Cartesian coordinates of specified process (array of integers)

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

{`void MPI::Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const` *(binding deprecated, see Section 15.2)* }

The inverse mapping, rank-to-coordinates translation is provided by MPI_CART_COORDS.

If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.

MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

IN	comm	communicator with graph topology (handle)
IN	rank	rank of process in group of comm (integer)
OUT	nneighbors	number of neighbors of specified process (integer)

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
```

```
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
```

{`int MPI::Graphcomm::Get_neighbors_count(int rank) const` *(binding deprecated, see Section 15.2)* }

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

IN	comm	communicator with graph topology (handle)
IN	rank	rank of process in group of comm (integer)
IN	maxneighbors	size of array neighbors (integer)
OUT	neighbors	ranks of processes that are neighbors to specified process (array of integer)

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
             int *neighbors)
```

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
```

{void MPI::Graphcomm::Get_neighbors(int rank, int maxneighbors, int neighbors[]) const *(binding deprecated, see Section 15.2)* }

MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency information for a general graph topology. The returned count and array of neighbors for the queried rank will both include *all* neighbors and reflect the same edge ordering as was specified by the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS will return values based on the original `index` and `edges` array passed to MPI_GRAPH_CREATE (assuming that `index[-1]` effectively equals zero):

ticket0.

- The [count]number of neighbors (nneighbors) returned from MPI_GRAPH_NEIGHBORS_COUNT will be (`index[rank] - index[rank-1]`).
- The `neighbors` array returned from MPI_GRAPH_NEIGHBORS will be `edges[index[rank-1]]` through `edges[index[rank]-1]`.

Example 7.5

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix (note that some neighbors are listed multiple times):

process	neighbors
0	1, 1, 3
1	0, 0
2	3
3	0, 2, 2

Thus, the input arguments to MPI_GRAPH_CREATE are:

nnodes = 4
index = 3, 5, 6, 9
edges = 1, 1, 3, 0, 0, 3, 0, 2, 2

Therefore, calling MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS for each of the 4 processes will return:

Input rank	Count	Neighbors
0	3	1, 1, 3
1	2	0, 0
2	1	3
3	3	0, 2, 2

Example 7.6

Suppose that comm is a communicator with a shuffle-exchange topology. The group has 2^n members. Each process is labeled by $a_1, \ldots, a_n$ with $a_i \in \{0, 1\}$, and has three neighbors: exchange$(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ $(\bar{a} = 1 - a)$, shuffle$(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$, and unshuffle$(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is illustrated below for $n = 3$.

node		**exchange** neighbors(1)	**shuffle** neighbors(2)	**unshuffle** neighbors(3)
0	(000)	1	0	0
1	(001)	0	2	4
2	(010)	3	4	1
3	(011)	2	6	5
4	(100)	5	1	2
5	(101)	4	3	6
6	(110)	7	5	3
7	(111)	6	7	7

Suppose that the communicator comm has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
C  assume: each process has stored a real number A.
C  extract neighborhood information
      CALL MPI_COMM_RANK(comm, myrank, ierr)
      CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
C  perform exchange permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0,
     +     neighbors(1), 0, comm, status, ierr)
C  perform shuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0,
     +     neighbors(3), 0, comm, status, ierr)
C  perform unshuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0,
     +     neighbors(2), 0, comm, status, ierr)
```

MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS provide adjacency information for a distributed graph topology.

MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, indegree, outdegree, weighted)

IN	comm	communicator with distributed graph topology (handle)
OUT	indegree	number of edges into this process (non-negative integer)
OUT	outdegree	number of edges out of this process (non-negative integer)
OUT	weighted	false if MPI_UNWEIGHTED was supplied during creation, true otherwise (logical)

```
int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
              int *outdegree, int *weighted)
```

```
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
    INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
    LOGICAL WEIGHTED
```

{void MPI::Distgraphcomm::Get_dist_neighbors_count(int rank,
int indegree[], int outdegree[], bool& weighted) const *(binding deprecated, see Section 15.2)* }

MPI_DIST_GRAPH_NEIGHBORS(comm, maxindegree, sources, sourceweights, maxoutdegree, destinations, destweights)

IN	comm	communicator with distributed graph topology (handle)
IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)
OUT	sources	processes for which the calling process is a destination (array of non-negative integers)
OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)
IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)
OUT	destinations	processes for which the calling process is a source (array of non-negative integers)
OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)

```
int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
              int sourceweights[], int maxoutdegree, int destinations[],
              int destweights[])
```

```
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
              MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
```

```
    INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,
        DESTINATIONS(*), DESTWEIGHTS(*), IERROR
```

{void MPI::Distgraphcomm::Get_dist_neighbors(int maxindegree, int sources[], int sourceweights[], int maxoutdegree, int destinations[], int destweights[]) *(binding deprecated, see Section 15.2)* }

These calls are local. The number of edges into and out of the process returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (potentially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays. [The]If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then ticket258. for each rank in comm, the order of the values in sources and destinations is identical to the input that was used by the process with the same rank in comm_old in the creation call. If the communicator was created with MPI_DIST_GRAPH_CREATE then the only requirement on the order of values in sources and destinations is that two calls to the routine with same input argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBOR_COUNT, then only the first part of the full list is returned. [Note, that the order of returned edges does need ticket258. not to be identical to the order that was provided in the creation of comm for the case that MPI_DIST_GRAPH_CREATE_ADJACENT was used.]

> *Advice to implementors.* Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI_DIST_GRAPH_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (*End of advice to implementors.*)

7.5.6 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation is likely to be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

IN	comm	communicator with Cartesian structure (handle)
IN	direction	coordinate dimension of shift (integer)
IN	disp	displacement (> 0: upwards shift, < 0: downwards shift) (integer)
OUT	rank_source	rank of source process (integer)
OUT	rank_dest	rank of destination process (integer)

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
              int *rank_source, int *rank_dest)
```

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
```

{`void MPI::Cartcomm::Shift(int direction, int disp, int& rank_source, int& rank_dest) const`*(binding deprecated, see Section 15.2)* }

The direction argument indicates the coordinate dimension to be traversed by the shift. The dimensions are numbered from 0 to `ndims-1`, where `ndims` is the number of dimensions.

Depending on the periodicity of the Cartesian group in the specified coordinate direction, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.

It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.

Example 7.7

The communicator, comm, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of `REAL`s is stored one element per process, in variable `A`. One wishes to skew this array, by shifting column `i` (vertically, i.e., along the column) by `i` steps.

```
....
C find process rank
      CALL MPI_COMM_RANK(comm, rank, ierr)
C find Cartesian coordinates
      CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
      CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
C skew array
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
     +                          status, ierr)
```

Advice to users. In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1) nodes, where DIMS is the array that was used to create the grid. In C, the dimension indicated by direction = i is the dimension specified by dims[i]. (*End of advice to users.*)

7.5.7 Partitioning of Cartesian [s]Structures ticket0.

MPI_CART_SUB(comm, remain_dims, newcomm)

IN	comm	communicator with Cartesian structure (handle)
IN	remain_dims	the i-th entry of remain_dims specifies whether the i-th dimension is kept in the subgrid (true) or is dropped (false) (logical vector)
OUT	newcomm	communicator containing the subgrid that includes the calling process (handle)

```
int MPI_Cart_sub(MPI_Comm comm, const int *remain_dims, MPI_Comm *newcomm)
```
ticket140.

```
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)
```

{MPI::Cartcomm MPI::Cartcomm::Sub(const bool remain_dims[]) const *(binding deprecated, see Section 15.2)* }

If a Cartesian topology has been created with MPI_CART_CREATE, the function MPI_CART_SUB can be used to partition the communicator group into subgroups that form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid Cartesian topology. If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology. (This function is closely related to MPI_COMM_SPLIT.)

Example 7.8

Assume that MPI_CART_CREATE(..., comm) has defined a $(2 \times 3 \times 4)$ grid. Let `remain_dims = (true, false, true)`. Then a call to,

```
MPI_CART_SUB(comm, remain_dims, comm_new),
```

will create three communicators each with eight processes in a 2×4 Cartesian topology. If `remain_dims = (false, false, true)` then the call to MPI_CART_SUB(comm, remain_dims, comm_new) will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.

7.5.8 Low-Level Topology Functions

The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, unless he or she is creating additional virtual topology capability other than that provided by MPI.

MPI_CART_MAP(comm, ndims, dims, periods, newrank)

IN	comm	input communicator (handle)
IN	ndims	number of dimensions of Cartesian structure (integer)
IN	dims	integer array of size `ndims` specifying the number of processes in each coordinate direction
IN	periods	logical array of size `ndims` specifying the periodicity specification in each coordinate direction
OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)

ticket140. ticket140.

```
int MPI_Cart_map(MPI_Comm comm, int ndims, const int *dims, const
              int *periods, int *newrank)
```

```
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
    LOGICAL PERIODS(*)
```

{`int MPI::Cartcomm::Map(int ndims, const int dims[], const bool periods[]) const` *(binding deprecated, see Section 15.2)* }

MPI_CART_MAP computes an "optimal" placement for the calling process on the physical machine. A possible implementation of this function is to always return the rank of the calling process, that is, not to perform any reordering.

Advice to implementors. The function MPI_CART_CREATE(comm, ndims, dims, periods, reorder, comm_cart), with `reorder = true` can be implemented by calling MPI_CART_MAP(comm, ndims, dims, periods, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_cart), with `color = 0` if `newrank` $\neq$ `MPI_UNDEFINED`, `color = MPI_UNDEFINED` otherwise, and `key = newrank`.

The function MPI_CART_SUB(comm, remain_dims, comm_new) can be implemented by a call to MPI_COMM_SPLIT(comm, color, key, comm_new), using a single number encoding of the lost dimensions as `color` and a single number encoding of the preserved dimensions as `key`.

All other Cartesian topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

The corresponding new function for general graph structures is as follows.

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)

IN	comm	input communicator (handle)
IN	nnodes	number of graph nodes (integer)
IN	index	integer array specifying the graph structure, see MPI_GRAPH_CREATE
IN	edges	integer array specifying the graph structure
OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if the calling process does not belong to graph (integer)

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, const int *index, const
              int *edges, int *newrank)
```

ticket140.
ticket140.

```
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
```

{int MPI::Graphcomm::Map(int nnodes, const int index[], const int edges[]) const *(binding deprecated, see Section 15.2)* }

Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm_graph), with `reorder = true` can be implemented by calling MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_graph), with `color = 0` if `newrank` $\neq$ `MPI_UNDEFINED`, `color = MPI_UNDEFINED` otherwise, and `key = newrank`.

All other graph topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

ticket258.

7.6 Neighborhood Collective Communication on Process Topologies

MPI process topologies specify a communication graph, but they implement no communication function themselves. Many applications require sparse nearest neighbor communications that can be expressed as graph topologies. We now describe several collective operations that perform communication along the edges of a process topology. All these functions are collective; i.e., they must be called by all processes in the specified communicator. See Section 5 on page 1 for an overview of other dense (global) collective communication operations and the semantics of collective operations.

If the graph was created with MPI_DIST_GRAPH_CREATE_ADJACENT with sources and destinations containing 0, ..., n-1, where n is the number of processes in the group of comm_old (i.e., the graph is fully connected and includes also an edge from each node to itself), then the sparse neighborhood communication routine performs the same data exchange as the corresponding dense (fully-connected) collective operation. In the case of a Cartesian communicator, only nearest neighbor communication is provided, corresponding to rank_source and rank_dist in MPI_CART_SHIFT with input disp=1.

Rationale. Neighborhood collective communications enable communication on a process topology. This high-level specification of data exchange among neighboring processes enables optimizations in the MPI library because the communication pattern is known statically (the topology). Thus, the implementation can compute optimized message schedules during creation of the topology [5]. This functionality can significantly simplify the implementation of neighbor exchanges [4]. (*End of rationale.*)

For a distributed graph topology, created with MPI_DIST_GRAPH_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined as the sequence returned by MPI_DIST_GRAPH_NEIGHBORS for destinations and sources, respectively. For a general graph topology, created with MPI_GRAPH_CREATE, the order of neighbors in the send and receive buffers is defined as the sequence of neighbors as returned by MPI_GRAPH_NEIGHBORS. Note that general graph topologies should generally be replaced by the distributed graph topologies.

For a Cartesian topology, created with MPI_CART_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined by order of the dimensions, first the neighbor in the negative direction and then in the positive direction with displacement 1. The numbers of sources and destinations in the communication routines are `2*ndims` with `ndims` defined in MPI_CART_CREATE. If a neighbor does not exist, i.e., at the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., `periods[...]==false`), then this neighbor is defined to be MPI_PROC_NULL.

If a neighbor in any of the functions is MPI_PROC_NULL, then the neighborhood collective communication behaves like a point-to-point communication with MPI_PROC_NULL in this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

7.6.1 Neighborhood Gather

In this function, each process i gathers data items from each process j if an edge (j, i) exists in the topology graph, and each process i sends the same data items to all processes j where an edge (i, j) exists. The send buffer is sent to each neighboring process and the l-th block in the receive buffer is received from the l-th neighbor.

MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcount	number of elements received from each neighbor (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)

ticket140.

```
int MPI_Neighbor_allgather(const void* sendbuf, int sendcount, MPI_Datatype
```

```
             sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
             MPI_Comm comm)
```

```
MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6 on page 25. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
                         outdegree,dsts,MPI_UNWEIGHTED);
int k,l;

for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+l*recvcount*extent(recvtype),recvcount,recvtype,
            srcs[l],...);

MPI_Waitall(...);
```

Figure 7.6.1 shows the neighborhood gather communication of one process with outgoing neighbors $d_0 \dots d_3$ and incoming neighbors $s_0 \dots s_5$. The process will send its sendbuf to all four destinations (outgoing neighbors) and it will receive the contribution from all six sources (incoming neighbors) into separate locations of its receive buffer.

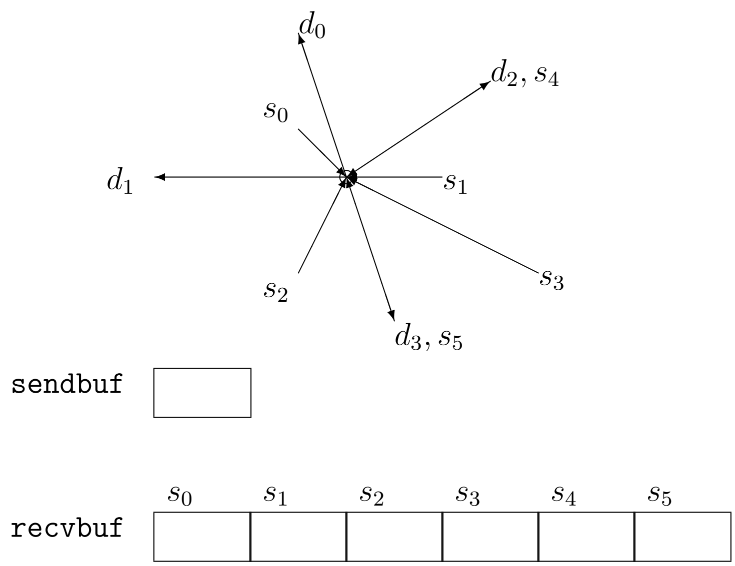
All arguments are significant on all processes and the argument comm must have identical values on all processes.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at all other processes. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

> *Rationale.* For optimization reasons, the same type signature is required independently of whether the topology graph is connected or not. (*End of rationale.*)

The "in place" option is not meaningful for this operation.

The vector variant of MPI_NEIGHBOR_ALLGATHER allows one to gather different numbers of elements from each neighbor.



MPI_NEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor
IN	displs	integer array (of length indegree). Entry `i` specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor `i`
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)

ticket140.
ticket140.
ticket140.

```
int MPI_Neighbor_allgatherv(const void* sendbuf, int sendcount,
              MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
              const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
              DISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6 on page 25. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
                         outdegree,dsts,MPI_UNWEIGHTED);
int k,l;

for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+displs[l]*extent(recvtype),recvcounts[l],recvtype,
            srcs[l],...);

MPI_Waitall(...);
```

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[l], recvtype at any other process with srcs[l]==j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed. The data received from the l-th neighbor is placed into recvbuf beginning at offset displs[l] elements (in terms of the recvtype).

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have identical values on all processes.

7.6.2 Neighbor Alltoall

In this function, each process i receives data items from each process j if an edge (j, i) exists in the topology graph or Cartesian topology. Similarly, each process i sends data items to all processes j where an edge (i, j) exists. This call is more general than MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor. The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in the receive buffer is received from the l-th neighbor.

MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcount	number of elements received from each neighbor (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)

ticket140.

```
int MPI_Neighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype
              sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)
```

```
MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
              RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6 on page 25. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
                         outdegree,dsts,MPI_UNWEIGHTED);
int k,l;

for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf+k*sendcount*extent(sendtype),sendcount,sendtype,
            dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+l*recvcount*extent(recvtype),recvcount,recvtype,
            srcs[l],...);

MPI_Waitall(...);
```

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have identical values on all processes.

The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different numbers of elements to and from each neighbor.

MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
IN	sdispls	integer array (of length outdegree). Entry `j` specifies the displacement (relative to sendbuf) from which to send the outgoing data to neighbor `j`
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that can are received from each neighbor
IN	rdispls	integer array (of length indegree). Entry `i` specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor `i`
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)

```
int MPI_Neighbor_alltoallv(const void* sendbuf, const int sendcounts[],
             const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
             const int recvcounts[], const  int rdispls[], MPI_Datatype
             recvtype, MPI_Comm comm)
```

ticket140.
ticket140.
ticket140.
ticket140.
ticket140.

```
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6 on page 25. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
```

```
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
                         outdegree,dsts,MPI_UNWEIGHTED);
int k,l;

for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf+sdispls[k]*extent(sendtype),sendcounts[k],sendtype,
            dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+rdispls[l]*extent(recvtype),recvcounts[l],recvtype,
            srcs[l],...);

MPI_Waitall(...);
```

The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process i must be equal to the type signature associated with recvcounts[l], recvtype with srcs[l]==i at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed. The data in the sendbuf beginning at offset sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data received from the l-th incoming neighbor is placed into recvbuf beginning at offset rdispls[l] elements (in terms of the recvtype).

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have identical values on all processes.

MPI_NEIGHBOR_ALLTOALLW allows one to send and receive with different datatypes to and from each neighbor.

MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
IN	sdispls	integer array (of length outdegree). Entry `j` specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor `j` (array of integers)
IN	sendtypes	array of datatypes (of length outdegree). Entry `j` specifies the type of data to send to neighbor `j` (array of handles)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that can are received from each neighbor
IN	rdispls	integer array (of length indegree). Entry `i` specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor `i` (array of integers)
IN	recvtypes	array of datatypes (of length indegree). Entry `i` specifies the type of data received from neighbor `i` (array of handles)
IN	comm	communicator with topology structure (handle)

```
int MPI_Neighbor_alltoallw(const void* sendbuf, const int sendcounts[],
             const int sdispls[], const MPI_Datatype sendtypes[], void*
             recvbuf, const int recvcounts[], const int rdispls[], const
             MPI_Datatype recvtypes[], MPI_Comm comm)
```

ticket140. ticket140. ticket140. ticket140. ticket140. ticket140. ticket140.

```
MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
    RDISPLS(*), RECVTYPES(*), COMM, IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6 on page 25. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
                         outdegree,dsts,MPI_UNWEIGHTED);
```

```
int k,l;

for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf+sdispls[k],sendcounts[k], sendtypes[k],dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+rdispls[l],recvcounts[l], recvtypes[l],srcs[l],...);

MPI_Waitall(...);
```

The type signature associated with sendcounts[k], sendtypes[k] with dsts[k]==j at process i must be equal to the type signature associated with recvcounts[l], recvtypes[l] with srcs[l]==i at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have identical values on all processes.

7.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 5.12.

7.7.1 Nonblocking Neighborhood Gather

MPI_INEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcount	number of elements received from each neighbor (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
OUT	request	communication request (handle)

ticket140.

```
int MPI_Ineighbor_allgather(const void* sendbuf, int sendcount,
              MPI_Datatype sendtype, void* recvbuf, int recvcount,
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

```
MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHER.

MPI_INEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
OUT	request	communication request (handle)

```
int MPI_Ineighbor_allgatherv(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
             MPI_Request *request)
```

ticket140.
ticket140.
ticket140.

```
MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
             DISPLS, RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHERV.

7.7.2 Nonblocking Neighborhood Alltoall

MPI_INEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcount	number of elements received from each neighbor (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
OUT	request	communication request (handle)

ticket140.

```
int MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype
              sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm, MPI_Request *request)
```

```
MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
              RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALL.

MPI_INEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
IN	sdispls	integer array (of length outdegree). Entry `j` specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor `j`
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that can are received from each neighbor
IN	rdispls	integer array (of length indegree). Entry `i` specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor `i`
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
OUT	request	communication request (handle)

```
int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[],
              const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
              const int recvcounts[], const int rdispls[], MPI_Datatype
              recvtype, MPI_Comm comm, MPI_Request *request)
```

ticket140.
ticket140.
ticket140.
ticket140.
ticket140.

```
MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
              RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.

MPI_INEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)
IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that can are received from each neighbor
IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)
IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)
IN	comm	communicator with topology structure (handle)
OUT	request	communication request (handle)

```
int MPI_Ineighbor_alltoallw(const void* sendbuf, const int sendcounts[],
              const int sdispls[], const MPI_Datatype sendtypes[], void*
              recvbuf, const int recvcounts[], const int rdispls[], const
              MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)
```

```
MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
              RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
    RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLW.

7.8 An Application Example

Example 7.9 []The example in [Figure 7.1]Figures 7.2-7.4 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine `relax`.

In each relaxation step each process computes new values for the solution grid function at [all]the points `u(1:100,1:100)` owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the [exchange subroutine might contain a call like MPI_SEND(...,neigh_rank(1),...) to send updated values to the left-hand neighbor `(i-1,j)`.]newly calculated values in `u(1,1:100)` must be sent into the halo cells `u(101,1:100)` of the left-hand neighbor with coordinates `(own_coord(1)-1,own_coord(2))` ticket258. ticket258. ticket258. ticket258.

[ ] ticket258. ticket258. ticket258.

```
      integer ndims, num_neigh
      logical reorder
      parameter (ndims=2, num_neigh=4, reorder=.true.)
      integer comm, comm_cart, dims(ndims), neigh_def(ndims), ierr
      integer neigh_rank(num_neigh), own_position(ndims), i, j
      logical periods(ndims)
      real*8 u(0:101,0:101), f(0:101,0:101)
      data dims / ndims * 0 /
      comm = MPI_COMM_WORLD
C     Set process grid size and periodicity
      call MPI_DIMS_CREATE(comm, ndims, dims,ierr)
      periods(1) = .TRUE.
      periods(2) = .TRUE.
C     Create a grid structure in WORLD group and inquire about own position
      call MPI_CART_CREATE (comm, ndims, dims, periods, reorder, comm_cart,ierr)
      call MPI_CART_GET (comm_cart, ndims, dims, periods, own_position,ierr)
C     Look up the ranks for the neighbors.  Own process coordinates are (i,j).
C     Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1)
      i = own_position(1)
      j = own_position(2)
      neigh_def(1) = i-1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(1),ierr)
      neigh_def(1) = i+1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2),ierr)
      neigh_def(1) = i
      neigh_def(2) = j-1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3),ierr)
      neigh_def(1) = i
      neigh_def(2) = j+1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4),ierr)
C     Initialize the grid functions and start the iteration
      call init (u, f)
      do 10 it=1,100
        call relax (u, f)
C     Exchange data with neighbor processes
        call exchange (u, comm_cart, neigh_rank, num_neigh)
10    continue
      call output (u)
      end
```

Figure 7.1: Set-up of process structure for two-dimensional parallel Poisson solver.

```
INTEGER ndims, num_neigh
LOGICAL reorder
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
INTEGER comm, comm_cart, dims(ndims), ierr
INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
LOGICAL periods(ndims)
REAL u(0:101,0:101), f(0:101,0:101)
DATA dims / ndims * 0 /
comm = MPI_COMM_WORLD
!   Set process grid size and periodicity
CALL MPI_DIMS_CREATE(comm, ndims, dims,ierr)
periods(1) = .TRUE.
periods(2) = .TRUE.
!   Create a grid structure in WORLD group and inquire about own position
CALL MPI_CART_CREATE (comm, ndims, dims, periods, reorder, &
                  comm_cart,ierr)
CALL MPI_CART_GET (comm_cart, ndims, dims, periods, own_coords,ierr)
i = own_coords(1)
j = own_coords(2)
!   Look up the ranks for the neighbors.  Own process coordinates are (i,j).
!   Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
CALL MPI_CART_SHIFT (comm_cart, 0,1,  neigh_rank(1),neigh_rank(2), ierr)
CALL MPI_CART_SHIFT (comm_cart, 1,1,  neigh_rank(3),neigh_rank(4), ierr)
!   Initialize the grid functions and start the iteration
CALL init (u, f)
DO it=1,100
   CALL relax (u, f)
!      Exchange data with neighbor processes
   CALL exchange (u, comm_cart, neigh_rank, num_neigh)
END DO
CALL output (u)
```

Figure 7.2: Set-up of process structure for two-dimensional parallel Poisson solver.

```
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
REAL u(0:101,0:101)
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
INTEGER ierr
sndbuf(1:100,1) = u(  1,1:100)
sndbuf(1:100,2) = u(100,1:100)
sndbuf(1:100,3) = u(1:100,  1)
sndbuf(1:100,4) = u(1:100,100)
CALL MPI_NEIGHBOR_ALLTOALL (sndbuf, 100, MPI_REAL,  rcvbuf, 100, MPI_REAL, &
                            comm_cart, ierr)
! instead of
! DO i=1,num_neigh
!   CALL MPI_IRECV(rcvbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i-1),ierr)
!   CALL MPI_ISEND(sndbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i  ),ierr)
! END DO
! CALL MPI_WAITALL (2*num_neigh, rq, statuses, ierr)

u(  0,1:100) = rcvbuf(1:100,1)
u(101,1:100) = rcvbuf(1:100,2)
u(1:100,  0) = rcvbuf(1:100,3)
u(1:100,101) = rcvbuf(1:100,4)
END
```

Figure 7.3: Communication routine with local data copying and sparse neighborhood all-to-all.

```
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
USE MPI
REAL u(0:101,0:101)
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
INTEGER sndcounts(num_neigh), sdispls(num_neigh), sndtypes(num_neigh)
INTEGER rcvcounts(num_neigh), rdispls(num_neigh), rcvtypes(num_neigh)
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
INTEGER type_vec, i, ierr
!   The following initialization need to be done only once
!   before the first call of exchange.
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
CALL MPI_TYPE_VECTOR (100, 1, 102, MPI_REAL, type_vec, ierr)
CALL MPI_TYPE_COMMIT (type_vec, ierr)
sndtypes(1) = type_vec
sndtypes(2) = type_vec
sndtypes(3) = MPI_REAL
sndtypes(4) = MPI_REAL
DO i=1,num_neigh
   sndcounts(i) = 100
   rcvcounts(i) = 100
   rcvtypes(i)  = sndtypes(i)
END DO
sdispls(1) = (  1 +   1*102) * sizeofreal   ! first element of u(  1,1:100)
sdispls(2) = (100 +   1*102) * sizeofreal   ! first element of u(100,1:100)
sdispls(3) = (  1 +   1*102) * sizeofreal   ! first element of u(1:100,  1)
sdispls(4) = (  1 + 100*102) * sizeofreal   ! first element of u(1:100,100)
rdispls(1) = (  0 +   1*102) * sizeofreal   ! first element of u(  0,1:100)
rdispls(2) = (101 +   1*102) * sizeofreal   ! first element of u(101,1:100)
rdispls(3) = (  1 +   0*102) * sizeofreal   ! first element of u(1:100,  0)
rdispls(4) = (  1 + 101*102) * sizeofreal   ! first element of u(1:100,101)

! the following communication has to be done in each call of exchange
CALL MPI_NEIGHBOR_ALLTOALLW (u, sndcounts, sdispls, sndtypes, &
                           u, rcvcounts, rdispls, rcvtypes, comm_cart, ierr)

!   The following finalizing need to be done only once
!   after the last call of exchange.
CALL MPI_TYPE_FREE (type_vec, ierr)
END
```

Figure 7.4: Communication routine with sparse neighborhood all-to-all-w and without local data copying.

Bibliography

[1] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole-routed networks for interprocessor communication in multicomputers. In *Proceedings of the 1990 Supercomputing Conference*, pages 647–656, 1990. 7.1

[2] S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication performance of large multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing, vol. II (Software)*, pages II–1 – II–4, 1991. 7.1

[3] Parasoft Corporation. Express version 1.0: A communication environment for parallel computers, 1988. 7.4

[4] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse Non-Blocking Collectives in Quantum Mechanical Calculations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 55–63. Springer, Sep. 2008. 7.6

[5] T. Hoefler and J. L. Traeff. Sparse Collective Operations for MPI. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop*, May 2009. 7.6

[6] O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. *Parallel Computing*, 9:213–225, 1989. 7.1

Index