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

Message Passing Interface Forum

December 9, 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 intercommunicators. 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.

 $46 \\ 47$

As stated in Chapter ??, 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 with machine-independent mapping and communication on virtual process topologies.

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

 $\overline{7}$ The communication pattern of a set of processes can be represented by a graph. The 8 nodes represent processes, and the edges connect processes that communicate with each 9 other. MPI provides message-passing between any pair of processes in a group. There 10 is no requirement for opening a channel explicitly. Therefore, a "missing link" in the 11user-defined process graph does not prevent the corresponding processes from exchanging 12messages. It means rather that this connection is neglected in the virtual topology. This 13 strategy implies that the topology gives no convenient way of naming this pathway of 14 communication. Another possible consequence is that an automatic mapping tool (if one 15exists for the runtime environment) will not take account of this edge when mapping.

16Specifying the virtual topology in terms of a graph is sufficient for all applications. 17However, in many applications the graph structure is regular, and the detailed set-up of the 18 graph would be inconvenient for the user and might be less efficient at run time. A large frac-19 tion of all parallel applications use process topologies like rings, two- or higher-dimensional 20grids, or tori. These structures are completely defined by the number of dimensions and 21the numbers of processes in each coordinate direction. Also, the mapping of grids and tori 22 is generally an easier problem than that of general graphs. Thus, it is desirable to address 23these cases explicitly.

²⁴ Process coordinates in a Cartesian structure begin their numbering at 0. Row-major ²⁵ numbering is always used for the processes in a Cartesian structure. This means that, for ²⁶ example, the relation between group rank and coordinates for four processes in a (2×2) ²⁷ grid is as follows.

28 29

30

- coord (0,0): coord (0,1): coord (1,0):
- 31 32
- 33 34

35

7.3 Embedding in MPI

coord (1,1):

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 ??.

40 41

42

7.4 Overview of the Functions

rank 0

rank 1

rank 2

rank 3

⁴³ 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.

1

2 3

 $\frac{4}{5}$

The topology creation functions take as input an existing communicator 1 comm_old, which defines the set of processes on which the topology is to be mapped. For 2 MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical 3 values on all processes of the group of comm_old. For MPI_DIST_GRAPH_CREATE_ADJACENT 4 and MPI_DIST_GRAPH_CREATE the input communication graph is distributed across the 5calling processes. Therefore the processes provide different values for the arguments spec-6 ifying the graph. However, all processes must give the same value for reorder and the $\overline{7}$ info argument. In all cases, a new communicator comm_topol is created that carries the 8 topological structure as cached information (see Chapter ??). In analogy to function 9 MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol. 10

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 viceversa; 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 ??, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation.

The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, 40 MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, 41 [and MPI_NEIGHBOR_ALLTOALLW]MPI_NEIGHBOR_ALLTOALLW, MPI_NEIGHBOR_REDUCE, and MPI_NEIGHBOR_REDUCEV communicate with the near-43 est neighbors on the topology associated with the communicator. The nonblocking vari-44 ants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, 45 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, [and $_{46}$ ticketXXX MPI_INEIGHBOR_ALLTOALLW]MPI_INEIGHBOR_ALLTOALLW, MPI_INEIGHBOR_REDUCE, 47 and MPI_INEIGHBOR_REDUCEV. 48

11

12

13

14

15

16 17

18

19 20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

 $_{42}$ ticketXXX.

```
Topology Constructors
               7.5
          1
          \mathbf{2}
               7.5.1 Cartesian Constructor
          3
          4
          \mathbf{5}
               MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
          6
          7
                                                        input communicator (handle)
                 IN
                            comm_old
          8
                 IN
                            ndims
                                                         number of dimensions of Cartesian grid (integer)
          9
                 IN
                            dims
                                                        integer array of size ndims specifying the number of
         10
         11
                                                         processes in each dimension
         12
                 IN
                            periods
                                                        logical array of size ndims specifying whether the grid
         13
                                                        is periodic (true) or not (false) in each dimension
         14
                                                        ranking may be reordered (true) or not (false) (logical)
                 IN
                            reorder
         15
         16
                 OUT
                            comm_cart
                                                        communicator with new Cartesian topology (handle)
         17
         18
               int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
         19
                               int reorder, MPI_Comm *comm_cart)
         20
               MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
         21
                    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
         22
                    LOGICAL PERIODS(*), REORDER
         23
ticket150.
         24
               {MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[],
         25
ticket150.
                               const bool periods[], bool reorder) const (binding deprecated, see
         26
                               Section ??) }
         27
                    MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian
         28
               topology information is attached. If reorder = false then the rank of each process in the
         29
               new group is identical to its rank in the old group. Otherwise, the function may reorder
         30
               the processes (possibly so as to choose a good embedding of the virtual topology onto
         31
               the physical machine). If the total size of the Cartesian grid is smaller than the size of
         32
               the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to
         33
               MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created.
         34
               The call is erroneous if it specifies a grid that is larger than the group size or if ndims is
         35
               negative.
         36
         37
                      Cartesian Convenience Function: MPI_DIMS_CREATE
               7.5.2
         38
         39
               For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced
         40
               distribution of processes per coordinate direction, depending on the number of processes
         41
               in the group to be balanced and optional constraints that can be specified by the user.
         42
               One use is to partition all the processes (the size of MPI_COMM_WORLD's group) into an
         43
               n-dimensional topology.
         44
         45
         46
         47
         48
```

MPI_DIMS_CREATE(nnodes, ndims, dims)					
IN	nnodes	number of nodes in a grid (integer)	2		
IN	ndims	number of Cartesian dimensions (integer)	3		
			4		
INOUT	dims	integer array of size ndims specifying the number of	5		
		nodes in each dimension	6 7		
			8		
<pre>int MPI_Dims_create(int nnodes, int ndims, int *dims)</pre>					
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)					
INTEG	ER NNODES, NDIMS, DIMS(*)), IERROR	11 ticket 150.		
{void MPI	::Compute_dims(int nnodes	s, int ndims, int dims[]) <i>(binding</i>	12 ticket 150.		
C .	deprecated, see Section		13		
T 1			14		
	e e	to describe a Cartesian grid with ndims dimensions	15		
and a total	of nnodes nodes. The dimens	sions are set to be as close to each other as possible,	16		

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 dims[i].

 $i, dims[i] \neq 0$

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.

	dims	function call	dims
	before call		on return
Example 71	(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
Example 7.1	(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)

			= (
	5 6	IN	comm_old	input communicator (handle)			
	7	IN	nnodes	number of nodes in graph (integer)			
	8	IN	index	array of integers describing node degrees (see below)			
	9 10	IN	edges	array of integers describing graph edges (see below)			
	11	IN	reorder	ranking may be reordered (true) or not (false) (logical)			
	12	OUT	comm_graph	communicator with graph topology added (handle)			
13	13						
	14	int MPT G	raph create(MPI Comm comm	n_old, int nnodes, int *index, int *edges,			
	15	Inc mi_di	int reorder, MPI_Com	6			
	16		Int reorder, hri_com	m *comm_graph)			
	17	MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,					
	18		IERROR)				
	19	INTEG	ER COMM_OLD, NNODES, INDE	EX(*), EDGES(*), COMM_GRAPH, IERROR			
ti aleat 150	20	LOGICAL REORDER					
ticket150	• 21	(107 a					
	22	{MP1::Grap		<pre>reate_graph(int nnodes, const int index[],</pre>			
ticket150	23		6	ool reorder) const <i>(binding deprecated, see</i>			
	24		Section $\ref{eq:section}$ }				
	25	MPI_G	RAPH_CREATE returns a ha	andle to a new communicator to which the graph			
	26			order = false then the rank of each process in the			
	27	new group i	is identical to its rank in the o	old group. Otherwise, the function may reorder the			
	28	processes. I	If the size, $nnodes$, of the grap	h is smaller than the size of the group of comm_old,			

then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE 29 and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL 30 is returned in all processes. The call is erroneous if it specifies a graph that is larger than 31 the group size of the input communicator. 32

The three parameters nnodes, index and edges define the graph structure. nnodes is 33 the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The 34i-th entry of array index stores the total number of neighbors of the first i graph nodes. 35The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations 36 in array edges. The array edges is a flattened representation of the edge lists. The total 37 number of entries in index is nodes and the total number of entries in edges is equal to the 38 number of graph edges. 39

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

4142

40

Example 7.2 Assume there are four processes 0, 1, 2, 3 with the following adjacency 43matrix: 44

- 45
- 46
- 47
- 48

 $\mathbf{1}$ $\mathbf{2}$ 3 4

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

Then, the input arguments are:

nnodes = 4index = 2, 3, 4, 6edges = 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 $0 \le j \le index[0] - 1$ and the list of neighbors of node i, i > 0, is stored in edges[j], index[i - 1] $\le j \le index[i] - 1$.

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 $1 \le j \le index(1)$ and the list of neighbors of node i, i > 0, is stored in edges(j), index(i) + $1 \le j \le index(i+1)$.

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:

index,
 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.*)

¹ 7.5.4 Distributed (Graph) Constructor

 $\mathbf{2}$ The general graph constructor assumes that each process passes the full (global) communi-3 cation graph to the call. This limits the scalability of this constructor. With the distributed 4 graph interface, the communication graph is specified in a fully distributed fashion. Each $\mathbf{5}$ process specifies only the part of the communication graph of which it is aware. Typically, 6 this could be the set of processes from which the process will eventually receive or get 7 data, or the set of processes to which the process will send or put data, or some combi-8 nation of such edges. Two different interfaces can be used to create a distributed graph 9 topology. MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph commu-10 nicator with each process specifying each of its incoming and outgoing (adjacent) edges 11in the logical communication graph and thus requires minimal communication during cre-12ation. MPI_DIST_GRAPH_CREATE provides full flexibility, and processes can indicate that 13 communication will occur between other pairs of processes. 14

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.

20

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

23 24	IN	comm_old	input communicator (handle)
25 26	IN	indegree	size of $sources$ and $sourceweights$ arrays (non-negative integer)
27 28	IN	sources	ranks of processes for which the calling process is a destination (array of non-negative integers)
29 30 31	IN	sourceweights	weights of the edges into the calling process (array of non-negative integers)
32 33	IN	outdegree	size of destinations and destweights arrays (non-negative integer)
34 35	IN	destinations	ranks of processes for which the calling process is a source (array of non-negative integers)
36 37 38	IN	destweights	weights of the edges out of the calling process (array of non-negative integers)
39 40	IN	info	hints on optimization and interpretation of weights (handle)
41 42	IN	reorder	the ranks may be reordered (true) or not (false) (logical)
43 44 45	OUT	comm_dist_graph	communicator with distributed graph topology (han-dle)
46 47 48	int MPI_I		t(MPI_Comm comm_old, int indegree, ourceweights[], int outdegree,

<pre>int destinations[], int destweights[], MPI_Info info,</pre>	1					
<pre>int reorder, MPI_Comm *comm_dist_graph)</pre>	2					
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,	3					
OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,	4					
COMM_DIST_GRAPH, IERROR)	5					
INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,	6					
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[],						
<pre>int outdegree, const int destinations[],</pre>	12					
const int destweights[], const MPI::Info& info, bool reorder)	13					
<pre>const (binding deprecated, see Section ??) }</pre>	¹⁴ ticket150. 15 ticket150.					
MDT. Distance hours						
{MPI::Distgraphcomm						
<pre>MPI::Intracomm::Dist_graph_create_adjacent(int indegree, const int sources[], int outdegree, const int destinations[],</pre>						
const MPI::Info& info, bool reorder) const (binding deprecated,	¹⁸ ticket150.					
see Section ??) }	19					
	20					

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 38 remapping strategy and other internal MPI optimizations. For instance, approximate count 39 arguments of later communication calls along specific edges could be used as their edge 40 weights. Multiplicity of edges can likewise indicate more intense communication between 41 pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 42standard and is left to the implementation. In C or Fortran, an application can supply 43the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the 44 same (effectively no) weight. In C++, this constant does not exist and the weight arguments 45may be omitted from the argument list. It is erroneous to supply MPI_UNWEIGHTED, or 46 47in C++ omit the weight arrays, for some but not all processes of comm_old. Note that

9

20

21

22

23

24

25

26

27

28

29

30

31

32

33 34

35 36

37

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

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
         10
                                                        input communicator (handle)
         11
                 IN
                                                        number of source nodes for which this process specifies
                           n
         12
                                                        edges (non-negative integer)
         13
                 IN
                                                        array containing the n source nodes for which this pro-
                           sources
         14
                                                        cess specifies edges (array of non-negative integers)
         15
                 IN
                           degrees
                                                        array specifying the number of destinations for each
         16
                                                        source node in the source node array (array of non-
         17
                                                        negative integers)
         18
         19
                 IN
                           destinations
                                                        destination nodes for the source nodes in the source
         20
                                                        node array (array of non-negative integers)
         21
                           weights
                 IN
                                                        weights for source to destination edges (array of non-
         22
                                                        negative integers)
         23
                 IN
                           info
                                                        hints on optimization and interpretation of weights
         24
                                                         (handle)
         25
                 IN
                           reorder
                                                        the process may be reordered (true) or not (false) (log-
         26
                                                        ical)
         27
         28
                 OUT
                           comm_dist_graph
                                                        communicator with distributed graph topology added
         29
                                                         (handle)
         30
         31
               int MPI_Dist_graph_create(MPI_Comm comm_old, int n, int sources[],
         32
                               int degrees[], int destinations[], int weights[],
         33
                               MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
         34
               MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
         35
                               INFO, REORDER, COMM_DIST_GRAPH, IERROR)
         36
                    INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
         37
                    WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
         38
                    LOGICAL REORDER
         39
ticket150.
         40
               {MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,
         41
                               const int sources[], const int degrees[], const int
         42
                               destinations[], const int weights[], const MPI::Info& info,
         43
ticket150.
                               bool reorder) const (binding deprecated, see Section ??) }
         44
ticket150.
               {MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,
         45
                               const int sources[], const int degrees[],
         46
                               const int destinations[], const MPI::Info& info, bool reorder)
         47
                               const (binding deprecated, see Section ??) }
ticket150. 48
```

6 7

8

MPI_DIST_GRAPH_CREATE returns a handle to a new communicator to which the 1 distributed graph topology information is attached. Concretely, each process calls the con- $\mathbf{2}$ structor with a set of directed (source, destination) communication edges as described below. 3 Every process passes an array of n source nodes in the sources array. For each source node, a 4non-negative number of destination nodes is specified in the degrees array. The destination 5nodes are stored in the corresponding consecutive segment of the destinations array. More 6 precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the j-th $\overline{7}$ such edge stored in destinations[degrees[0]+...+degrees[i-1]+j]. The weight of this edge is 8 stored in weights [degrees[0]+...+degrees[i-1]+i]. Both the sources and the destinations arrays 9 may contain the same node more than once, and the order in which nodes are listed as 10 destinations or sources is not significant. Similarly, different processes may specify edges 11 with the same source and destination nodes. Source and destination nodes must be pro-12cess ranks of comm_old. Different processes may specify different numbers of source and 13 destination nodes, as well as different source to destination edges. This allows a fully dis-14 tributed specification of the communication graph. Isolated processes (i.e., processes with 15no outgoing or incoming edges, that is, processes that do not occur as source or destination 16 node in the graph specification) are allowed. 17

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 27 remapping strategy and other internal MPI optimizations. For instance, approximate count 28arguments of later communication calls along specific edges could be used as their edge 29 weights. Multiplicity of edges can likewise indicate more intense communication between 30 pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 31 standard and is left to the implementation. In C or Fortran, an application can supply 32 the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the 33 same (effectively no) weight. In C++, this constant does not exist and the weights argument 34 may be omitted from the argument list. It is erroneous to supply MPI_UNWEIGHTED, or 35 in C++ omit the weight arrays, for some but not all processes of comm_old. Note that 36 MPI_UNWEIGHTED is not a special weight value; rather it is a special value for the total 37 array argument. In C, one would expect it to be NULL. In Fortran, MPI_UNWEIGHTED is 38 an object like MPI_BOTTOM (not usable for initialization or assignment). See Section ?? 39

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

18

19

20

21

22

23

24

25

26

40

41

42

43

44

45

 $46 \\ 47$

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	$1,\!3,\!0,\!3,\!0,\!2$	$1,\!1,\!1,\!1,\!1,\!1,\!1$
1	0	-	-	-	-
2	0	-	-	-	-
3	0	-	-	-	

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

 $\mathsf{MPI}_\mathsf{DIST}_\mathsf{GRAPH}_\mathsf{CREATE}_\mathsf{ADJACENT}$ 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

 $\mathbf{5}$

Example 7.4 A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be 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:

```
/*
                                                                                     7
Input:
           dimensions P, Q
                                                                                     8
Condition: number of processes equal to P*Q; otherwise only
                                                                                     9
           ranks smaller than P*Q participate
                                                                                     10
*/
                                                                                     11
int rank, x, y;
                                                                                     12
int sources[1], degrees[1];
                                                                                     13
int destinations[8], weights[8];
                                                                                     14
                                                                                     15
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                                                                                     16
                                                                                     17
/* get x and y dimension */
                                                                                     18
y=rank/P; x=rank%P;
                                                                                     19
                                                                                     20
/* get my communication partners along x dimension */
                                                                                     21
destinations[0] = P*y+(x+1)%P; weights[0] = 2;
                                                                                    22
destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
                                                                                    23
                                                                                    24
/* get my communication partners along y dimension */
                                                                                     25
destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
                                                                                     26
destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;
                                                                                     27
                                                                                     28
/* get my communication partners along diagonals */
                                                                                     29
destinations[4] = P*((y+1))(Q)+(x+1)(P); weights[4] = 1;
                                                                                     30
destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
                                                                                    31
destinations[6] = P*((y+1)%Q)+(P+x-1)%P; weights[6] = 1;
                                                                                     32
destinations[7] = P*((Q+y-1))(Q)+(P+x-1)(P); weights[7] = 1;
                                                                                     33
                                                                                     34
sources[0] = rank;
                                                                                    35
degrees[0] = 8;
                                                                                    36
MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
                                                                                    37
                       weights, MPI_INFO_NULL, 1, comm_dist_graph)
                                                                                    38
                                                                                     39
```

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.

1

 $\mathbf{2}$

3

4

5 6

40 41

42

```
MPI_TOPO_TEST(comm, status)
         1
         \mathbf{2}
                 IN
                                                       communicator (handle)
                           comm
         3
                 OUT
                                                        topology type of communicator comm (state)
                           status
          4
         \mathbf{5}
               int MPI_Topo_test(MPI_Comm comm, int *status)
         6
         \overline{7}
               MPI_TOPO_TEST(COMM, STATUS, IERROR)
          8
                   INTEGER COMM, STATUS, IERROR
ticket150. 9
ticket150. 10
               {int MPI::Comm::Get_topology() const (binding deprecated, see Section ??) }
         11
                   The function MPI_TOPO_TEST returns the type of topology that is assigned to a
         12
               communicator.
         13
                   The output value status is one of the following:
         14
         15
                 MPI_GRAPH
                                                          graph topology
         16
                 MPI CART
                                                          Cartesian topology
         17
                                                          distributed graph topology
                 MPI_DIST_GRAPH
         18
                                                          no topology
                 MPI_UNDEFINED
         19
         20
         21
               MPI_GRAPHDIMS_GET(comm, nnodes, nedges)
         22
                 IN
                                                       communicator for group with graph structure (handle)
                           comm
         23
                 OUT
         24
                            nnodes
                                                        number of nodes in graph (integer) (same as number
                                                        of processes in the group)
         25
         26
                 OUT
                            nedges
                                                       number of edges in graph (integer)
         27
         28
               int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
         29
         30
               MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
         31
                   INTEGER COMM, NNODES, NEDGES, IERROR
ticket150. 32
ticket150. 33
               {void MPI::Graphcomm::Get_dims(int nnodes[], int nedges[]) const (binding
                               deprecated, see Section ??) }
         34
         35
                   Functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topology
         36
               information that was associated with a communicator by MPI_GRAPH_CREATE.
         37
                   The information provided by MPI_GRAPHDIMS_GET can be used to dimension the
         38
               vectors index and edges correctly for the following call to MPI_GRAPH_GET.
         39
         40
         41
         42
         43
         44
         45
         46
         47
         48
```

MPI_GRAP	H_GET(comm, maxindex, max	xedges, index, edges)	1		
IN	comm	communicator with graph structure (handle)	2		
IN	maxindex	length of vector index in the calling program	3		
		(integer)	4 5		
IN	maxedges	length of vector edges in the calling program	6		
		(integer)	7		
OUT	index	array of integers containing the graph structure (for	8		
		details see the definition of MPI_GRAPH_CREATE)	9		
OUT	edges	array of integers containing the graph structure	10		
001	cages	array of moegers containing the graph structure	11		
int MPT G	raph get(MPT Comm comm.	int maxindex, int maxedges, int *index,	12 13		
1110 111 1_0	int *edges)	maninuon, ino manougos, ino vinaon,	13		
	<u> </u>		15		
		EDGES, INDEX, EDGES, IERROR)	16		
INTEG	ER COMM, MAXINDEX, MAXEDO	GES, INDEX(*), EDGES(*), IERROR	17 ticket 150.		
{void MPI	::Graphcomm::Get_topo(int	<pre>maxindex, int maxedges, int index[],</pre>	18		
	<pre>int edges[]) const (</pre>	binding deprecated, see Section ??) }	19 ticket 150 .		
			20		
			21		
MPI_CARTDIM_GET(comm, ndims)					
IN	comm	communicator with Cartesian structure (handle)	23		
OUT	ndims		24		
001	nums	number of dimensions of the Cartesian structure (in- teger)	25		
		leger)	26		
int MDT C	artdim_get(MPI_Comm comm	int undima)	27 28		
IIIC MFI_Co	arturm_get(Mr1_Comm Comm)	, IIIC #IIGIIIIS)	29		
MPI_CARTD	IM_GET(COMM, NDIMS, IERRO	JR)	30		
INTEG	ER COMM, NDIMS, IERROR		31 ticket 150.		
{int MPT:	:Cartcomm::Get dim() cons	st (binding deprecated, see Section ??) }	32 ticket 150.		
C C			33		
		and MPI_CART_GET return the Cartesian topol-	34		
00		a communicator by MPI_CART_CREATE. If comm	35		
		Cartesian topology, MPI_CARTDIM_GET returns	36		
ndims=0 ar	Id MIPI_CART_GET will keep	all output arguments unchanged.	37		
			38		
			39		
			40		
			41		
			42		
			43		
			44		
45					

MDI	CRADH	CET	(comm	maxindax	maxedges,	indox	odaoc)	
	GIVAL II	GLI		maximuex,	maxeuges.	muex,	euges	

```
MPI_CART_GET(comm, maxdims, dims, periods, coords)
          1
          \mathbf{2}
                 IN
                            comm
                                                         communicator with Cartesian structure (handle)
          3
                 IN
                            maxdims
                                                         length of vectors dims, periods, and coords in the
          4
                                                         calling program (integer)
          5
                 OUT
                            dims
                                                         number of processes for each Cartesian dimension (ar-
          6
          7
                                                         ray of integer)
          8
                 OUT
                            periods
                                                         periodicity (true/false) for each Cartesian dimension
          9
                                                         (array of logical)
          10
                 OUT
                            coords
                                                         coordinates of calling process in Cartesian structure
          11
                                                         (array of integer)
          12
         13
               int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
         14
                                int *coords)
         15
         16
               MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
         17
                    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
          18
                    LOGICAL PERIODS(*)
ticket150. 19
               {void MPI::Cartcomm::Get_topo(int maxdims, int dims[], bool periods[],
         20
                               int coords[]) const (binding deprecated, see Section ??) }
ticket150. 21
         22
         23
         24
               MPI_CART_RANK(comm, coords, rank)
         25
                 IN
                                                         communicator with Cartesian structure (handle)
                            comm
         26
                 IN
                            coords
                                                         integer array (of size ndims) specifying the Cartesian
         27
                                                         coordinates of a process
         28
         29
                 OUT
                                                         rank of specified process (integer)
                            rank
         30
         31
               int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
         32
               MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
         33
                    INTEGER COMM, COORDS(*), RANK, IERROR
          34
ticket150.
ticket150.
               {int MPI::Cartcomm::Get_cart_rank(const int coords[]) const (binding
          36
                               deprecated, see Section ??) }
         37
                    For a process group with Cartesian structure, the function MPI_CART_RANK trans-
         38
               lates the logical process coordinates to process ranks as they are used by the point-to-point
         39
               routines.
          40
                    For dimension i with periods(i) = true, if the coordinate, coords(i), is out of
         41
               range, that is, coords(i) < 0 or coords(i) \ge dims(i), it is shifted back to the interval
         42
               0 \leq \text{coords}(i) < \text{dims}(i) automatically. Out-of-range coordinates are erroneous for
         43
               non-periodic dimensions.
         44
                    If comm is associated with a zero-dimensional Cartesian topology, coords is not signif-
         45
               icant and 0 is returned in rank.
         46
         47
         48
```

MPI_CAR	MPI_CART_COORDS(comm, rank, maxdims, coords)				
IN	comm	communicator with Cartesian structure (handle)	2		
IN	rank	rank of a process within group of comm (integer)	3		
IN	maxdims	(- ,	4		
IN	maxums	length of vector coords in the calling program (integer)	5 6		
OUT	coords	integer array (of size ndims) containing the Cartesian	7		
001	coords	coordinates of specified process (array of integers)	8		
		coordinates of specifical process (array of integers)	9		
int MPI (art coords(MPI Comm comm	, int rank, int maxdims, int *coords)	10		
_			11		
	COORDS (COMM, RANK, MAXDI		12		
INTEG	ER COMM, RANK, MAXDIMS,	CUURDS(*), IERRUR	$^{13}_{14}$ ticket 150.		
{void MPI		nt rank, int maxdims, int coords[]) const	15		
	(binding deprecated, see	e Section ??) }	$\frac{10}{16}$ ticket 150.		
The i	nverse mapping, rank-to-coo	rdinates translation is provided by	17		
	LCOORDS.		18		
If con	nm is associated with a zero	o-dimensional Cartesian topology,	19		
coords will	be unchanged.		20		
			21		
MPI_GRA	PH_NEIGHBORS_COUNT(cor	nm, rank, nneighbors)	22 23		
IN	comm	communicator with graph topology (handle)	23		
			25		
IN	rank	rank of process in group of comm (integer)	26		
OUT	nneighbors	number of neighbors of specified process (integer)	27		
			28		
int MPI_C	raph_neighbors_count(MPI	_Comm comm, int rank, int *nneighbors)	29		
MPI_GRAPH	_NEIGHBORS_COUNT(COMM, R	ANK, NNEIGHBORS, IERROR)	30 31		
INTEG	ER COMM, RANK, NNEIGHBOR	S, IERROR			
{int MPT:	:Graphcomm::Get_neighbor	s_count(int rank) const (binding deprecated,	32 ticket150. 33 ticket150.		
(1110 111 11	see Section ??) }		34		
			35		
			36		
MPI_GRA	PH_NEIGHBORS(comm, rank,	maxneighbors, neighbors)	37		
IN	comm	communicator with graph topology (handle)	38 39		
IN	rank	rank of process in group of comm (integer)	40		
IN	maxneighbors	size of array neighbors (integer)	41		
	0		42		
OUT	neighbors	ranks of processes that are neighbors to specified pro- cess (array of integer)	43		
int MDT (ranh neighbors(MDT Comm	comm, int rank, int maxneighbors,	45		
IN0 IN I_C	int *neighbors)	somm, int fair, int mannerghours,	46 47		
			48		

1	MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
2	INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
ticket150. $\frac{1}{3}$	INTEGER COMM, RAWK, MAXNETGIDDUGS, NEIGIDDUGS(*), IERROR
4	<pre>{void MPI::Graphcomm::Get_neighbors(int rank, int maxneighbors, int</pre>
ticket 150. $\frac{1}{5}$	<pre>neighbors[]) const (binding deprecated, see Section ??) }</pre>
6	MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency
7	information for a general graph topology. The returned count and array of neighbors for
8	the queried rank will both include all neighbors and reflect the same edge ordering as
9	was specified by the original call to MPI_GRAPH_CREATE. Specifically,
10	MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS will return values based
11	on the original index and edges array passed to MPI_GRAPH_CREATE (assuming that
12	index[-1] effectively equals zero):
13	
14	$\bullet~$ The number of neighbors (nneighbors) returned from MPI_GRAPH_NEIGHBORS_COUNT
15	will be $(index[rank] - index[rank-1])$.
16	• The neighbors array returned from MPI_GRAPH_NEIGHBORS will be
17	• The heighbors array returned from MFI_GKAFT_NEIGHBORS will be edges[index[rank-1]] through edges[index[rank]-1].
18	edges[Index[Iank-I]] through edges[Index[Iank]-I].
19	Example 7.5 Assume there are four processes 0, 1, 2, 3 with the following adjacency
20	matrix (note that some neighbors are listed multiple times):
21	· · · · · · · · · · · · · · · · · · ·
22	process neighbors
23	0 1, 1, 3
24	1 0, 0
25	2 3
26	3 0, 2, 2
27	Thus, the input arguments to MPI_GRAPH_CREATE are:
28	Thus, the input arguments to with Edit at the energy and are.
29 30	nnodes = 4
31	index = 3, 5, 6, 9
32	edges = 1, 1, 3, 0, 0, 3, 0, 2, 2
33	Therefore, calling MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS
34	for each of the 4 processes will return:
35	
36	Input rank Count Neighbors
37	0 3 1, 1, 3
38	1 2 0, 0
39	2 1 3
40	$3 \qquad 3 \qquad 0, 2, 2$
41	
42	Example 7.6 Suppose that comm is a communicator with a shuffle-exchange topology. The
43	group has 2^n members. Each process is labeled by a_1, \ldots, a_n with $a_i \in \{0, 1\}$, and has
44	three neighbors: exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ $(\bar{a} = 1 - a)$, shuffle $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$
45	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
46	illustrated below for $n = 3$.
47	
48	

node		exchange	shuffle	unshuffle
		neighbors(1)	neighbors(2)	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 (hap	
	IIN	comm	communicator with distributed graph topology (han-	35
			dle)	36
	OUT	indegree	number of edges into this process (non-negative inte-	37
			$\operatorname{ger})$	38
	OUT	outdegree	number of edges out of this process (non-negative in-	39
	001	outdegree		40
			teger)	41
	OUT	weighted	false if MPI_UNWEIGHTED was supplied during cre-	42
			ation, true otherwise (logical)	43
				44
i	nt MPI_Di	<pre>ist_graph_neighbors_count</pre>	(MPI_Comm comm, int *indegree,	45
		int *outdegree, int *	weighted)	46

MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)

¹ ticket150.		GER COMM, INDEGREE, C CAL WEIGHTED	DUTDEGREE, IERROR	
ticket150. 4 5 6 7	{void MP		<pre>s_dist_neighbors_count(int rank, int outdegree[], bool& weighted) const (binding etion ??) }</pre>	
8 9 10		_GRAPH_NEIGHBORS(ns, destweights)	comm, maxindegree, sources, sourceweights, maxoutdegree,	
11 12	IN	comm	communicator with distributed graph topology (han-dle)	
13 14 15	IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)	
16 17	OUT	sources	processes for which the calling process is a destination (array of non-negative integers)	
18 19	OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)	
20 21 22	IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)	
23 24	OUT	destinations	processes for which the calling process is a source (array of non-negative integers)	
25 26	OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)	
27 28 29 30	<pre>int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>			
31 32 33 34 ticket150. ³⁵	MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR) INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR			
36 37 ticket150. 38 39 40	<pre>{void MPI::Distgraphcomm::Get_dist_neighbors(int maxindegree,</pre>			
40 41 42 43 44 45 46 47 48	MPI_DIST call to MI tially by MPI_DIST MPI_DIST sourceweig	GRAPH_NEIGHBORS_ PI_DIST_GRAPH_CREAT processes other than th GRAPH_CREATE). Mu GRAPH_NEIGHBORS i ghts or destweights or bot	umber of edges into and out of the process returned by COUNT are the total number of such edges given in the TE_ADJACENT or MPI_DIST_GRAPH_CREATE (poten- ne calling process in the case of altiply defined edges are all counted and returned by an some order. If MPI_UNWEIGHTED is supplied for th, or if MPI_UNWEIGHTED was supplied during the con- ght information is returned in that array or those arrays.	

If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then 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.

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.

			27		
IN	comm	communicator with Cartesian structure (handle)	28		
IN	direction	coordinate dimension of shift (integer)	29		
IN	disp	displacement (> 0: upwards shift, < 0 : downwards	30		
		shift) (integer)	31		
OUT	rank_source	rank of source process (integer)	32		
001	Tank_Source	Tank of source process (integer)	33		
OUT	rank_dest	rank of destination process (integer)	34		
			35		
int MPI_Ca	art_shift(MPI_Comm comm,	int direction, int disp,	36		
	int *rank_source, int *rank_dest)				
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)					
		, RANK_SOURCE, RANK_DEST, IERROR	39		
TNIEGE	IR COMM, DIRECTION, DISP,	, RANK_SOURCE, RANK_DESI, IERROR	40 ticket 150.		
{void MPI:	:Cartcomm::Shift(int dir	rection, int disp, int& rank_source,	41		
	int& rank_dest) cons	t (binding deprecated, see Section ??) }	42 ticket 150.		
The dir	eaction argument indicates the	e coordinate dimension to be traversed by the shift.	43		
	0	· · · · · · · · · · · · · · · · · · ·	44		
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 direc-					
-			46		
	ANT_SHIFT provides the ide	entifiers for a circular or an end-off shift. In the case	47		
48					

 $\mathbf{2}$

 $\overline{7}$

of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest, 1 indicating that the source or the destination for the shift is out of range. 2

It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or 3 greater than or equal to the number of dimensions in the Cartesian communicator. This 4 implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with $\mathbf{5}$ a zero-dimensional Cartesian topology. 6

Example 7.7 The communicator, comm, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of REALs is stored one element per process, in variable A. One wishes to skew this array, by shifting column i (vertically, i.e., along the 10 column) by i steps. 11

```
12
               . . . .
         13
               C find process rank
         14
                      CALL MPI_COMM_RANK(comm, rank, ierr)
         15
               C find Cartesian coordinates
         16
                      CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
         17
               C compute shift source and destination
         18
                      CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
         19
               C skew array
         20
                      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
         21
                     +
                                                    status, ierr)
         22
                    Advice to users. In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1)
         23
                    nodes, where DIMS is the array that was used to create the grid. In C, the dimension
         24
                    indicated by direction = i is the dimension specified by dims[i]. (End of advice to users.)
         25
         26
         27
               7.5.7
                      Partitioning of Cartesian Structures
         28
         29
         30
               MPI_CART_SUB(comm, remain_dims, newcomm)
         31
                 IN
                                                       communicator with Cartesian structure (handle)
                           comm
         32
                 IN
                           remain_dims
                                                       the i-th entry of remain_dims specifies whether the
         33
                                                       i-th dimension is kept in the subgrid (true) or is drop-
         34
                                                       ped (false) (logical vector)
         35
         36
                 OUT
                           newcomm
                                                       communicator containing the subgrid that includes
         37
                                                       the calling process (handle)
         38
         39
               int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
         40
               MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
         41
                   INTEGER COMM, NEWCOMM, IERROR
         42
                   LOGICAL REMAIN_DIMS(*)
         43
ticket150.
         44
ticket150.
               {MPI::Cartcomm MPI::Cartcomm::Sub(const bool remain_dims[]) const (binding
         45
                              deprecated, see Section ??) }
         46
                   If a Cartesian topology has been created with MPI_CART_CREATE, the function
         47
               MPI_CART_SUB can be used to partition the communicator group into subgroups that
```

22

7

8

9

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.)

23

1

 $\mathbf{2}$

3

4

5 6

7

8 9

10 11

12

13

14

15 16

17

18

19

20 21 22

47 48

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) 23 IN comm input communicator (handle) 2425IN ndims number of dimensions of Cartesian structure (integer) 26IN dims integer array of size ndims specifying the number of 27 processes in each coordinate direction 28IN periods logical array of size ndims specifying the periodicity 29 specification in each coordinate direction 30 31 OUT newrank reordered rank of the calling process; 32 MPI_UNDEFINED if calling process does not belong 33 to grid (integer) 34 35int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods, 36 int *newrank) 37 MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR) 38 INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR 39 LOGICAL PERIODS(*) ticket150. 41 {int MPI::Cartcomm::Map(int ndims, const int dims[], const bool periods[]) 42const (binding deprecated, see Section ??) } ticket150. 43 MPI_CART_MAP computes an "optimal" placement for the calling process on the phys-44 ical machine. A possible implementation of this function is to always return the rank of the 45calling process, that is, not to perform any reordering. 46

1 2 3 4 5 6 7 8 9 10 11 12	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. (<i>End of advice to implementors.</i>)		
13	The o	corresponding new fund	ction for general graph structures is as follows.
14 15 16	MPI_GRA	.PH_MAP(comm, nnod	es, index, edges, newrank)
10	IN	comm	input communicator (handle)
18	IN	nnodes	number of graph nodes (integer)
19	IN	index	integer array specifying the graph structure, see
20	IN	mucx	MPI_GRAPH_CREATE
21 22	IN	edges	integer array specifying the graph structure
23	OUT	newrank	reordered rank of the calling process;
24			MPI_UNDEFINED if the calling process does not be-
25 26			long to graph (integer)
20	int MDT	Crearly man (MDI Comm	comm int modes int window int wedges
28 29	IIIC MPI_	int *newrank)	<pre>comm, int nnodes, int *index, int *edges,</pre>
30	MPI_GRAP	H_MAP(COMM, NNODES,	INDEX, EDGES, NEWRANK, IERROR)
ticket 150. 31	INTE	GER COMM, NNODES, I	NDEX(*), EDGES(*), NEWRANK, IERROR
32	{int MPI	::Graphcomm::Map(in	t nnodes, const int index[], const int edges[])
ticket150. $\frac{^{33}}{_{34}}$		const (binding	deprecated, see Section ??) }
35	Adv	ice to implementors.	The function MPI_GRAPH_CREATE(comm, nnodes, index,
36 37	-		n), with reorder = true can be implemented by calling
38		GRAPH_MAP(comm,	• ,, ·
39		•	, color, key, comm_graph), with color = 0 if newrank \neq MPI_UNDEFINED otherwise, and key = newrank.
40		,	functions can be implemented locally, using the topology
41 42		· · · ·	with the communicator. (<i>End of advice to implementors.</i>)
43			
44	7.6 Ne	eighborhood Collec	tive Communication on Process Topologies
45	MDT	···· ··· ··· ··· ···	
46 47			a communication graph, but they implement no commu- Iany applications require sparse nearest neighbor commu-
48			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 ?? on page ?? 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

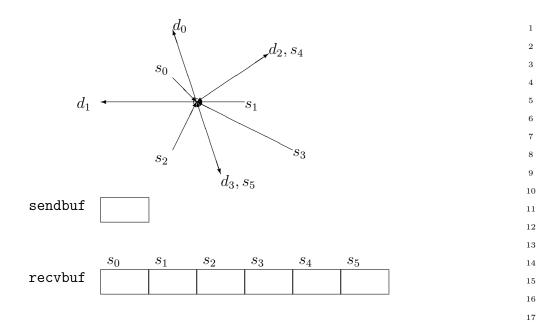
In this function, each process i gathers data items from each process j if an edge (j, i) exists in the topology graph, and each process i sends the same data items to all processes j where an edge (i, j) exists. The send buffer is sent to each neighboring process and the l-th block in the receive buffer is received from the l-th neighbor. $\mathbf{2}$

 $\mathbf{5}$

 $\overline{7}$

```
MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
1
      comm)
\mathbf{2}
3
       IN
                 sendbuf
                                              starting address of send buffer (choice)
4
                 sendcount
       IN
                                               number of elements sent to each neighbor (non-negative
5
                                              integer)
6
       IN
                 sendtype
                                              data type of send buffer elements (handle)
7
8
                                              starting address of receive buffer (choice)
        OUT
                 recvbuf
9
       IN
                                               number of elements received from each neighbor (non-
                  recvcount
10
                                               negative integer)
11
       IN
                                               data type of receive buffer elements (handle)
                  recvtype
12
13
       IN
                 comm
                                               communicator with topology structure (handle)
14
15
      int MPI_Neighbor_allgather(void* sendbuf, int sendcount, MPI_Datatype
16
                     sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
17
                     MPI_Comm comm)
18
      MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
19
                     RECVTYPE, COMM, IERROR)
20
          <type> SENDBUF(*), RECVBUF(*)
21
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
22
23
          This function supports Cartesian communicators, graph communicators, and distributed
24
     graph communicators as described in Section 7.6 on page 24. If comm is a distributed graph
25
     communicator, the outcome is as if each process executed sends to each of its outgoing
26
      neighbors and receives from each of its incoming neighbors:
27
28
     MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
29
     int *srcs=(int*)malloc(indegree*sizeof(int));
30
      int *dsts=(int*)malloc(outdegree*sizeof(int));
31
     MPI_Dist_graph_neighbors(comm,indegree,srcs,outdegree,dsts,MPI_UNWEIGHTED);
32
      int k,l;
33
34
      for(k=0; k<outdegree; ++k)</pre>
35
        MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
36
37
      for(l=0; l<indegree; ++1)</pre>
38
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
39
                    srcs[1],...);
40
41
     MPI_Waitall(...)
42
          Figure 7.6.1 shows the neighborhood gather communication of one process with out-
43
     going neighbors d_0 \ldots d_3 and incoming neighbors s_0 \ldots s_5. The process will send its sendbuf
44
      to all four destinations (outgoing neighbors) and it will receive the contribution from all six
45
      sources (incoming neighbors) into separate locations of its receive buffer.
46
          All arguments are significant on all processes and the argument
47
```

⁴⁸ 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.

 $45 \\ 46$

```
MPI_NEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
1
      recvtype, comm)
\mathbf{2}
3
       IN
                 sendbuf
                                              starting address of send buffer (choice)
4
       IN
                 sendcount
                                               number of elements sent to each neighbor (non-negative
5
                                              integer)
6
       IN
                 sendtype
                                               data type of send buffer elements (handle)
7
8
        OUT
                  recvbuf
                                              starting address of receive buffer (choice)
9
       IN
                  recvcounts
                                              non-negative integer array (of length indegree) con-
10
                                               taining the number of elements that are received from
11
                                               each neighbor
12
       IN
                 displs
                                              integer array (of length indegree). Entry i specifies
13
                                               the displacement (relative to recvbuf) at which to place
14
                                               the incoming data from neighbor i
15
16
       IN
                  recvtype
                                               data type of receive buffer elements (handle)
17
       IN
                 comm
                                              communicator with topology structure (handle)
18
19
      int MPI_Neighbor_allgatherv(void* sendbuf, int sendcount, MPI_Datatype
20
                     sendtype, void* recvbuf, int recvcounts[], int displs[],
21
                     MPI_Datatype recvtype, MPI_Comm comm)
22
23
     MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
24
                     DISPLS, RECVTYPE, COMM, IERROR)
25
          <type> SENDBUF(*), RECVBUF(*)
26
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
27
          IERROR
28
          This function supports Cartesian communicators, graph communicators, and distributed
29
      graph communicators as described in Section 7.6 on page 24. If comm is a distributed graph
30
      communicator, the outcome is as if each process executed sends to each of its outgoing
31
     neighbors and receives from each of its incoming neighbors:
32
33
     MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
34
      int *srcs=(int*)malloc(indegree*sizeof(int));
35
      int *dsts=(int*)malloc(outdegree*sizeof(int));
36
     MPI_Dist_graph_neighbors(comm,indegree,srcs,outdegree,dsts,MPI_UNWEIGHTED);
37
      int k,l;
38
39
     for(k=0; k<outdegree; ++k)</pre>
40
        MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
41
42
      for(l=0; l<indegree; ++l)</pre>
43
        MPI_Irecv(recvbuf+displs[1]*extent(recvtype), recvcounts[1], recvtype,
44
                   srcs[1],...);
45
46
     MPI_Waitall(...)
47
48
```

 $\mathbf{2}$

 $46 \\ 47$

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[1], recvtype at any other process with srcs[1]==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 1-th neighbor is placed into recvbuf beginning at offset displs[1] 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)

			22
IN	sendbuf	starting address of send buffer (choice)	23
IN	sendcount	number of elements sent to each neighbor (non-negative	24
		integer)	25
IN	sendtype	data type of send buffer elements (handle)	26
0.U.T			27
OUT	recvbuf	starting address of receive buffer (choice)	28
IN	recvcount	number of elements received from each neighbor (non-	29
		negative integer)	30
IN	recvtype	data type of receive buffer elements (handle)	31
			32
IN	comm	communicator with topology structure (handle)	33

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 24. 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));
1
      int *dsts=(int*)malloc(outdegree*sizeof(int));
\mathbf{2}
     MPI_Dist_graph_neighbors(comm, indegree, srcs, outdegree, dsts, MPI_UNWEIGHTED);
3
      int k,l;
4
\mathbf{5}
      for(k=0; k<outdegree; ++k)</pre>
6
        MPI_Isend(sendbuf+k*sendcount*extent(sendtype),sendcount,sendtype,
\overline{7}
                    dsts[k],...);
8
9
      for(l=0; l<indegree; ++1)</pre>
10
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
11
                    srcs[1],...);
12
13
     MPI_Waitall(...)
14
15
          The type signature associated with sendcount, sendtype, at a process must be equal to
16
      the type signature associated with recvcount, recvtype at any other process. This implies
17
      that the amount of data sent must be equal to the amount of data received, pairwise between
18
      every pair of communicating processes. Distinct type maps between sender and receiver are
19
      still allowed.
20
          The "in place" option is not meaningful for this operation.
21
          All arguments are significant on all processes and the argument
22
      comm must have identical values on all processes.
23
          The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different
24
      numbers of elements to and from each neighbor.
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
^{41}
42
43
44
45
46
47
48
```

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

1

2

30

31 32

33 34

35 36

48

•	51 7		
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor	4 5 6
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	7 8 9
IN	sendtype	data type of send buffer elements (handle)	10
OUT	recvbuf	starting address of receive buffer (choice)	11 12
IN	recvcounts	non-negative integer array (of length indegree) spec- ifying the number of elements that can are received from each neighbor	12 13 14 15
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	16 17 18
IN	recvtype	data type of receive buffer elements (handle)	19 20
IN	comm	communicator with topology structure (handle)	20 21 22
int MPI_No	MPI_Datatype sendtype	<pre>sendbuf, int sendcounts[], int sdispls[], e, void* recvbuf, int recvcounts[], int ype recvtype, MPI_Comm comm)</pre>	23 24 25
<type< td=""><td>RECVCOUNTS, RDISPLS, > SENDBUF(*), RECVBUF(*)</td><td>ENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVTYPE, COMM, IERROR)</td><td>26 27 28 29</td></type<>	RECVCOUNTS, RDISPLS, > SENDBUF(*), RECVBUF(*)	ENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVTYPE, COMM, IERROR)	26 27 28 29

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 24. 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);
                                                                                     37
int *srcs=(int*)malloc(indegree*sizeof(int));
                                                                                     38
int *dsts=(int*)malloc(outdegree*sizeof(int));
                                                                                     39
MPI_Dist_graph_neighbors(comm,indegree,srcs,outdegree,dsts,MPI_UNWEIGHTED);
                                                                                     40
int k,l;
                                                                                     41
                                                                                     42
for(k=0; k<outdegree; ++k)</pre>
                                                                                     43
  MPI_Isend(sendbuf+sdispls[k]*extent(sendtype),sendcounts[k],sendtype,
                                                                                     44
            dsts[k],...);
                                                                                     45
                                                                                     46
for(l=0; l<indegree; ++1)</pre>
                                                                                     47
```

MPI_Irecv(recvbuf+rdispls[1]*extent(recvtype),recvcounts[1],recvtype,

1		<pre>srcs[1],);</pre>				
2 3	MPI_Waitall()					
4		The type signature associated with sendcounts[k], sendtype with dsts[k]==j at pro-				
5			e signature associated with recvcounts[1], recvtype with			
6 7			nplies that the amount of data sent must be equal to the			
8			between every pair of communicating processes. Distinct			
9			ceiver are still allowed. The data in the sendbuf beginning			
10		-	erms of the sendtype) is sent to the k-th outgoing neighbor. noming neighbor is placed into recvbuf beginning at offset			
11		elements (in terms of t				
12			meaningful for this operation.			
13			at on all processes and the argument			
14 15		ust have identical values	•			
16			W allows one to send and receive with different datatypes			
17	to and fr	om each neighbor.				
18						
19	MPI_NEI	GHBOR_ALLTOALLW(se	endbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,			
20	rdispls, re	ecvtypes, comm)				
21 22	IN	sendbuf	starting address of send buffer (choice)			
23	IN	sendcounts	non-negative integer array (of length outdegree) speci-			
24			fying the number of elements to send to each neighbor			
25	IN	sdispls	integer array (of length outdegree). Entry j specifies			
26			the displacement in bytes (relative to $sendbuf)$ from			
27			which to take the outgoing data destined for neighbor			
28 29			j (array of integers)			
30	IN	sendtypes	array of datatypes (of length outdegree). Entry j spec-			
31			ifies the type of data to send to neighbor j (array of			
32			handles)			
33	OUT	recvbuf	starting address of receive buffer (choice)			
34	IN	recvcounts	non-negative integer array (of length indegree) spec-			
35			ifying the number of elements that can are received			
$\frac{36}{37}$			from each neighbor			
38	IN	rdispls	integer array (of length indegree). Entry i specifies			
39			the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor <i>i</i> (array of			
40			integers)			
41	IN	recvtypes	array of datatypes (of length indegree). Entry i spec-			
42	11 N	icertypes	ifies the type of data received from neighbor i (array			
43			of handles)			
44	IN	comm	communicator with topology structure (handle)			
45 46						
47	int MPI	_Neighbor_alltoallw(v	<pre>void* sendbuf, int sendcounts[], int sdispls[],</pre>			
48		•	endtypes[], void* recvbuf, int recvcounts[], int			
		• -				

rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)	1
MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,	2
RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)	3
<type> SENDBUF(*), RECVBUF(*)</type>	4
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),	5
RDISPLS(*), RECVTYPES(*), COMM, IERROR	6
	7
This function supports Cartesian communicators, graph communicators, and distributed	8
graph communicators as described in Section 7.6 on page 24. If comm is a distributed graph	9
communicator, the outcome is as if each process executed sends to each of its outgoing	10
neighbors and receives from each of its incoming neighbors:	11
	12
<pre>MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);</pre>	13
<pre>int *srcs=(int*)malloc(indegree*sizeof(int));</pre>	14
<pre>int *dsts=(int*)malloc(outdegree*sizeof(int));</pre>	15
<pre>MPI_Dist_graph_neighbors(comm,indegree,srcs,outdegree,dsts,MPI_UNWEIGHTED);</pre>	16
int k,l;	17
	18
<pre>for(k=0; k<outdegree; ++k)<="" pre=""></outdegree;></pre>	19
<pre>MPI_Isend(sendbuf+sdispls[k],sendcounts[k], sendtypes[k],dsts[k],);</pre>	20
	21
<pre>for(1=0; 1<indegree; ++1)<="" pre=""></indegree;></pre>	22
<pre>MPI_Irecv(recvbuf+rdispls[l],recvcounts[l], recvtypes[l],srcs[l],);</pre>	23
	24
MPI_Waitall()	25
	26
The type signature associated with sendcounts[k], sendtypes[k] with dsts[k]==j at	27
process i must be equal to the type signature associated with recvcounts[1], recvtypes[1]	28
with srcs[1]==i at process j. This implies that the amount of data sent must be equal	29
to the amount of data received, pairwise between every pair of communicating processes.	30
Distinct type maps between sender and receiver are still allowed.	31
The "in place" option is not meaningful for this operation.	32
All arguments are significant on all processes and the argument	33
comm must have identical values on all processes.	$_{34}$ ticketXXX.
	35
7.6.3 Neighborhood Reduction Operations	36
In some applications, each process might require the sum of a value of all its neighbors.	37
For this, MPI offers the neighborhood reduction call. MPI_Neighbor_reduce acts like an	38
MPI_Reduce with one communicator per process in which the owning process is rank 0	39
and all other processes are the incoming neighbors of rank 0 (in the order returned by the	40
neighborhood query function). Similar restrictions as for MPI_Reduce apply.	41
nongenoor nood query runction, ominar restrictions as for wir i_neduce appry.	42
	43

MPI_NEIGHBOR_REDUCE(sendbuf, recvbuf, count, datatype, op, comm) 1 $\mathbf{2}$ IN sendbuf starting address of send buffer (choice) 3 OUT recvbuf starting address of receive buffer (choice) 4 IN number of elements in all buffers (non-negative inte-5count 6 ger) 7 IN datatype data type of all buffer elements (handle) 8 IN reduce operation (handle) op 9 IN communicator with topology structure (handle) 10 comm 11 12int MPI_Neighbor_reduce(void* sendbuf, void* recvbuf, int count, 13MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) 14 MPI_NEIGHBOR_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 15 <type> SENDBUF(*), RECVBUF(*) 16INTEGER COUNT, DATATYPE, RECVCOUNT, OP, COMM, IERROR 1718 Processes might contribute to multiple different reductions in different neighborhoods. 19If not all reduction neighborhoods have the same number of elements, the vector variant 20MPI_NEIGHBOR_REDUCEV can be used to specify the correct size for each neighborhood. 2122MPI_NEIGHBOR_REDUCEV(sendbuf, sendcounts, displs, datatype, recvbuf, recvcount, op, 23 comm) 2425IN sendbuf starting address of send buffer (choice) 26 IN sendcounts non-negative integer array (of length outdegree) speci-27fying the number of elements to send to each processor 28IN displs integer array (of length outdegree). Entry j specifies 29 the displacement (relative to sendbuf) from which to 30 take the outgoing data destined for process j 31 32 IN datatype data type of send buffer elements (handle) 33 OUT recvbuf starting address of receive buffer (choice) 34 IN number of elements received from any process (nonrecvcount 35 negative integer) 36 37 IN reduce operation (handle) op 38 IN communicator with topology structure (handle) comm 39 40 int MPI_Neighbor_reducev(void* sendbuf, int sendcounts, int displs[], 41 MPI_Datatype datatype, void* recvbuf, int recvcount, MPI_Op 42op, MPI_Comm comm) 4344 MPI_NEIGHBOR_REDUCEV(SENDBUF, SENDCOUNTS, DISPLS, DATATYPE, RECVBUF, 45 RECVCOUNT, OP, COMM, IERROR) 46 <type> SENDBUF(*), RECVBUF(*) 47INTEGER SENDCOUNTS(*), DISPLS(*), DATATYPE, RECVCOUNT, OP, COMM, IERROR 48

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 **??**.

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)	12
		Ŭ ()	13
IN	sendcount	number of elements sent to each neighbor (non-negative	14
		integer)	15
IN	sendtype	data type of send buffer elements (handle)	16
OUT	recvbuf	starting address of receive buffer (choice)	17
001		starting address of receive starter (choree)	18
IN	recvcount	number of elements received from each neighbor (non-	19
		negative integer)	20
IN	recvtype	data type of receive buffer elements (handle)	21
IN	comm	communicator with topology structure (handle)	22
IIN	comm	communicator with topology structure (nandle)	23
OUT	request	communication request (handle)	24

int MPI_Ineighbor_allgather(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.

 $1 \\ 2$

12		GHBOR_ALLGATHERV(sendbu omm, request)	f, sendcount, sendtype, recvbuf, recvcounts, displs,
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
7	IN	sendtype	data type of send buffer elements (handle)
8	OUT	recvbuf	starting address of receive buffer (choice)
9 10 11 12	IN	recvcounts	non-negative integer array (of length indegree) con- taining the number of elements that are received from each neighbor
13 14 15	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
16	IN	recvtype	data type of receive buffer elements (handle)
17 18	IN	comm	communicator with topology structure (handle)
19	OUT	request	communication request (handle)
22 23 24 25 26 27 28	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 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.</type>		
29 30 31			
32			
33 34			
34 35			
36			
37			
38 39			
40			
41			
42			
43 44			
44			
46			
47			
48			

7.7. NONBLOCKING NEIGHBORHOOD COMMUNICATION ON PROCESS TOPOLOGIES37

7.7.2	Nonblocking Neighborh	nood Alltoall	1	
			2 3	
MPI_INE	EIGHBOR_ALLTOALL(s	sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,	4	
request)	X		5	
IN	sendbuf	starting address of send buffer (choice)	6 7	
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	8 9	
IN	sendtype	data type of send buffer elements (handle)	10	
OUT	recvbuf	starting address of receive buffer (choice)	11	
IN	recvcount	number of elements received from each neighbor (non-negative integer)	12 13 14	
IN	recvtype	data type of receive buffer elements (handle)	15	
IN	comm	communicator with topology structure (handle)	16	
OUT	request	communication request (handle)	17 18	
			19	
int MPI	U U	(void* sendbuf, int sendcount, MPI_Datatype	20	
	V 1	<pre>id* recvbuf, int recvcount, MPI_Datatype recvtype, n, MPI_Request *request)</pre>	21 22	
			22	
MPI_INE		IDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	24	
<t.< td=""><td colspan="4">RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*)</type></td></t.<>	RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>			
•		IDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	26 27	
Thi	s call starts a nonblock	ing variant of MPI_NEIGHBOR_ALLTOALL.	21	
1 111			29	
			30	
			31	
			32 33	
			34	
			35	
			36	
			37 38	
			39	
			40	
			41	
			42	
			43 44	
			45	

12		GHBOR_ALLTOALLV(sendbuf, vtype, comm, request)	sendcounts, sdispls, sendtype, recvbuf, recvcounts,
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
7 8 9	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j
10	IN	sendtype	data type of send buffer elements (handle)
11 12	OUT	recvbuf	starting address of receive buffer (choice)
13 14 15	IN	recvcounts	non-negative integer array (of length indegree) spec- ifying the number of elements that can are received from each neighbor
16 17 18 19	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
20	IN	recvtype	data type of receive buffer elements (handle)
21	IN	comm	communicator with topology structure (handle)
22 23	OUT	request	communication request (handle)
24 25 26 27 28	MPI_Datatype sendtype, void* recvbuf, int recvcounts[], int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)		
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	 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</type> This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV. 		
46 47 48			

MPI_INEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, 1 rdispls, recvtypes, comm, request) 2				
IN	sendbuf	starting address of send buffer (choice)	3	
IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor	4 5 6	
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)	7 8 9 10	
IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)	11 12 13	
OUT	recvbuf	starting address of receive buffer (choice)	14 15	
IN	recvcounts	non-negative integer array (of length indegree) spec- ifying the number of elements that can are received from each neighbor	16 17 18	
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)	19 20 21 22	
IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)	23 24 25 26	
IN	comm	communicator with topology structure (handle)	27	
OUT	request	communication request (handle)	28 29	
<pre>int MPI_Ineighbor_alltoallw(void* sendbuf, int sendcounts[], int sdispls[],</pre>			29 30 31 32 33 34	
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 $\mathsf{MPI_NEIGHBOR_ALLTOALLW}.$

 $^{40}_{41}$ ticketXXX.

1 2	7.7.3 N	lonblocking Neighbor	hood Reductions
3 4 5	MPI_INE comm, re	`	endbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, op,
6	IN	sendbuf	starting address of send buffer (choice)
7 8 9	IN	sendcount	number of elements sent to each process (non-negative integer)
10	IN	sendtype	data type of send buffer elements (handle)
11 12	OUT	recvbuf	starting address of receive buffer (choice)
12 13 14	IN	recvcount	number of elements received from any process (non-negative integer)
15	IN	recvtype	data type of receive buffer elements (handle)
16	IN	ор	reduce operation (handle)
17 18	IN	comm	communicator with topology structure (handle)
19	OUT	request	communication request (handle)
22 23 24		sendtype, vo MPI_Op op, M	<pre>void* sendbuf, int sendcount, MPI_Datatype bid* recvbuf, int recvcount, MPI_Datatype recvtype, IPI_Comm comm, MPI_Request *request)</pre>
25	MPI_INE:		BUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
26 27	<tv< td=""><td>pe> SENDBUF(*), R</td><td></td></tv<>	pe> SENDBUF(*), R	
28 29	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, OP, COMM, REQUEST, IERROR		
30 31	This	call starts a nonbloc	king variant of MPI_NEIGHBOR_REDUCE.
32			
33 34			
35			
36			
37			
38 39			
40			
41			
42			
43 44			
44 45			
46			
47			
48			

MPI_INEIGHBOR_REDUCEV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recv-1 type, op, comm) $\mathbf{2}$ 3 IN sendbuf starting address of send buffer (choice) 4 IN sendcounts non-negative integer array (of length outdegree) speci-5fying the number of elements to send to each processor 6 IN displs integer array (of length outdegree). Entry j specifies 7 the displacement (relative to sendbuf) from which to 8 take the outgoing data destined for process j 9 10 IN sendtype data type of send buffer elements (handle) 11 OUT recvbuf starting address of receive buffer (choice) 12 IN recvcount number of elements received from any process (non-13 negative integer) 14 15data type of receive buffer elements (handle) IN recvtype 16 IN reduce operation (handle) op 17IN comm communicator with topology structure (handle) 18 19 OUT request communication request (handle) 2021int MPI_Ineighbor_reducev(void* sendbuf, int *sendcounts, int *displs, 22 MPI_Datatype sendtype, void* recvbuf, int recvcount, 23 MPI_Datatype recvtype, MPI_Op op, MPI_Comm comm, 24MPI_Request *request) 25MPI_INEIGHBOR_REDUCEV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, 26RECVCOUNT, RECVTYPE, OP, COMM, REQUEST, IERROR) 27<type> SENDBUF(*), RECVBUF(*) 28INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, OP, 29 COMM, REQUEST, IERROR 30 31

This call starts a nonblocking variant of MPI_NEIGHBOR_REDUCEV.

7.8 An Application Example

Example 7.9 The example in Figures 7.1-7.3 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 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 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))

47 48

32 33

34 35

36

37

38

39

40

41

42

43

44

45

```
1
2
3
4
5
6
7
     INTEGER ndims, num_neigh
8
     LOGICAL reorder
9
     PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
10
     INTEGER comm, comm_cart, dims(ndims), ierr
11
     INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j
12
     LOGICAL periods(ndims)
13
     REAL u(0:101,0:101), f(0:101,0:101)
14
     DATA dims / ndims * 0 /
15
     comm = MPI_COMM_WORLD
16
         Set process grid size and periodicity
     !
17
     CALL MPI_DIMS_CREATE(comm, ndims, dims,ierr)
18
     periods(1) = .TRUE.
19
     periods(2) = .TRUE.
20
         Create a grid structure in WORLD group and inquire about own position
21
     1
     CALL MPI_CART_CREATE (comm, ndims, dims, periods, reorder,
22
                        comm_cart,ierr)
23
     CALL MPI_CART_GET (comm_cart, ndims, dims, periods, own_coords,ierr)
24
     i = own_coords(1)
25
     j = own_coords(2)
26
         Look up the ranks for the neighbors. Own process coordinates are (i,j).
27
     1
         Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
     1
28
     CALL MPI_CART_SHIFT (comm_cart, 0,1, neigh_rank(1),neigh_rank(2), ierr)
29
     CALL MPI_CART_SHIFT (comm_cart, 1,1, neigh_rank(3), neigh_rank(4), ierr)
30
     !
         Initialize the grid functions and start the iteration
31
     CALL init (u, f)
32
     DO it=1,100
33
        CALL relax (u, f)
34
            Exchange data with neighbor processes
     !
35
        CALL exchange (u, comm_cart, neigh_rank, num_neigh)
36
     END DO
37
     CALL output (u)
38
39
40
        Figure 7.1: Set-up of process structure for two-dimensional parallel Poisson solver.
41
42
43
44
45
46
47
48
```

```
7
                                                                                    8
                                                                                    9
                                                                                    10
                                                                                    11
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
                                                                                    12
REAL u(0:101,0:101)
                                                                                    13
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
                                                                                    14
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
                                                                                    15
INTEGER ierr
                                                                                    16
sndbuf(1:100,1) = u( 1,1:100)
                                                                                    17
sndbuf(1:100,2) = u(100,1:100)
                                                                                    18
sndbuf(1:100,3) = u(1:100, 1)
                                                                                    19
sndbuf(1:100,4) = u(1:100,100)
                                                                                    20
CALL MPI_NEIGHBOR_ALLTOALL (sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
                                                                                    21
                           comm_cart, ierr)
                                                                                    22
! instead of
                                                                                    23
! DO i=1,num_neigh
                                                                                    24
    CALL MPI_IRECV(rcvbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i-1),ierr)
!
                                                                                    25
    CALL MPI_ISEND(sndbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i ),ierr)
!
                                                                                    26
! END DO
                                                                                    27
! CALL MPI_WAITALL (2*num_neigh, rq, statuses, ierr)
                                                                                    28
                                                                                    29
u( 0,1:100) = rcvbuf(1:100,1)
                                                                                    30
u(101,1:100) = rcvbuf(1:100,2)
                                                                                    31
u(1:100, 0) = rcvbuf(1:100,3)
                                                                                    32
u(1:100,101) = rcvbuf(1:100,4)
                                                                                    33
END
                                                                                    34
                                                                                    35
```

Figure 7.2: Communication routine with local data copying and sparse neighborhood all-to-all.

36

```
2
3
     SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
4
     REAL u(0:101,0:101)
5
     INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
6
     INTEGER sndcounts(num_neigh), sdispls(num_neigh), sndtypes(num_neigh)
7
     INTEGER rcvcounts(num_neigh), rdispls(num_neigh), rcvtypes(num_neigh)
8
     INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
9
     INTEGER type_vec, i, ierr
10
         The following initialization need to be done only once
11
     !
         before the first call of exchange.
12
     CALL MPI_TYPE_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
13
     CALL MPI_TYPE_VECTOR (100, 1, 102, MPI_REAL, type_vec, ierr)
14
     CALL MPI_TYPE_COMMIT (type_vec, ierr)
15
     sndtypes(1) = type_vec
16
     sndtypes(2) = type_vec
17
     sndtypes(3) = MPI_REAL
18
     sndtypes(4) = MPI_REAL
19
    DO i=1,num_neigh
20
        sndcounts(i) = 100
21
        rcvcounts(i) = 100
22
        rcvtypes(i) = sndtypes(i)
23
    END DO
24
     sdispls(1) = ( 1 + 1*102) * sizeofreal
                                                  ! first element of u( 1,1:100)
25
     sdispls(2) = (100 + 1*102) * sizeofreal
                                                  ! first element of u(100,1:100)
26
     sdispls(3) = ( 1 + 1*102) * sizeofreal
                                                  ! first element of u(1:100, 1)
27
     sdispls(4) = ( 1 + 100*102) * sizeofreal ! first element of u(1:100,100)
28
     rdispls(1) = ( 0 + 1*102) * sizeofreal
                                                  ! first element of u( 0,1:100)
29
    rdispls(2) = (101 + 1*102) * sizeofreal ! first element of u(101,1:100)
30
     rdispls(3) = (1 + 0*102) * size of real ! first element of u(1:100, 0)
31
     rdispls(4) = ( 1 + 101*102) * sizeofreal ! first element of u(1:100,101)
32
33
34
     ! the following communication has to be done in each call of exchange
     CALL MPI_NEIGHBOR_ALLTOALLW (u, sndcounts, sdispls, sndtypes,
35
                                u, rcvcounts, rdispls, rcvtypes, comm_cart, ierr)
36
37
     !
         The following finalizing need to be done only once
38
         after the last call of exchange.
     !
39
     CALL MPI_TYPE_FREE (type_vec, ierr)
40
     END
41
42
43
     Figure 7.3: Communication routine with sparse neighborhood all-to-all-w and without local
44
     data copying.
45
46
47
48
```

Bibliography

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

[1] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole-

- [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

CONST:DIMS, 22 CONST:DIMS(i+1), 22CONST:dims[i], 22 CONST:DIRECTION = i, 22CONST: direction = i, 22CONST: false, 4, 6, 8, 10, 16, 19 CONST:MPI::Cartcomm, 4 CONST:MPI::Graphcomm, 6 CONST:MPI_BOTTOM, 10, 11 CONST:MPI_CART, 14 CONST:MPI_COMM_NULL, 4, 6 CONST:MPI_COMM_WORLD, 4 CONST:MPI_DIST_GRAPH, 14 CONST:MPI_GRAPH, 14 CONST:MPI_INFO_NULL, 12 CONST:MPI_PROC_NULL, 22, 25 CONST:MPI_UNDEFINED, 14, 23, 24 CONST:MPI_UNWEIGHTED, 9-12, 19, 20 CONST:NULL, 10, 11 CONST:true, 4, 6, 8, 10, 16, 19

EXAMPLES: Cartesian virtual topologies, 41 MPI_CART_SHIFT, 3, 21, 22, 25 EXAMPLES:MPI_CART_COORDS, 22 MPI_CART_SHIFT(comm, direction, disp, EXAMPLES:MPI_CART_CREATE, 41 EXAMPLES:MPI_CART_GET, 41 MPI_CART_SUB, 3, 22 EXAMPLES:MPI_CART_RANK, 22 MPI_CART_SUB(comm, remain_dims, comm_new), EXAMPLES:MPI_CART_SHIFT, 22, 41 EXAMPLES:MPI_CART_SUB, 23 MPI_CART_SUB(comm, remain_dims, new-EXAMPLES:MPI_DIMS_CREATE, 5, 41 EXAMPLES:MPI_DIST_GRAPH_CREATE, MPI_CARTDIM_GET, 3, 15 MPI_CARTDIM_GET(comm, ndims), 15 12EXAMPLES:MPI_Dist_graph_create, 12 12

22

MPI_COMM_CREATE, 3 EXAMPLES: MPI_DIST_GRAPH_CREATE_MDJACODNM_SPLIT, 3, 4, 6, 23 MPI_COMM_SPLIT(comm, color, key, comm_cart), EXAMPLES:MPI_GRAPH_CREATE, 6, 18 24EXAMPLES:MPI_GRAPH_NEIGHBORS, 18MPI_COMM_SPLIT(comm, color, key, comm_graph),

EXAMPLES: MPI_GRAPH_NEIGHBORS_COUNT, 24 MPI_COMM_SPLIT(comm, color, key, comm_new), 18 EXAMPLES:MPI_SENDRECV_REPLACE, 24

MPI_DIMS_CREATE, 3–5

EXAMPLES:Neighborhood collective commu-

MPI_CART_COORDS(comm, rank, maxdims,

MPI_CART_CREATE, 2-6, 15, 22, 23, 25

MPI_CART_CREATE(comm, ndims, dims,

periods, reorder, comm_cart), 24

MPI_CART_CREATE(comm_old, ndims, dims,

periods, reorder, comm_cart), 4

MPI_CART_GET(comm, maxdims, dims, pe-

MPI_CART_MAP(comm, ndims, dims, pe-

MPI_CART_RANK(comm, coords, rank), 16

rank_source, rank_dest), 21

riods, newrank), 23, 24

nication, 41

EXAMPLES: Topologies, 41

MPI_CART_COORDS, 3, 17

coords), 17

MPI_CART_GET, 3, 15

MPI_CART_MAP, 3, 23

MPI_CART_RANK, 3, 16

23.24

 comm), 22

riods, coords), 16

EXAMPLES: Virtual topologies, 41

MPI_DIMS_CREATE(6, 2, dims), 5 MPI_GRAPH_NEIGHBORS(comm, rank, maxneigh-MPI_DIMS_CREATE(6, 3, dims), 5 bors, neighbors), 17 MPI_DIMS_CREATE(7, 2, dims), 5 MPI_GRAPH_NEIGHBORS_COUNT, 3, 18 3 MPI_DIMS_CREATE(7, 3, dims), 5 MPI_GRAPH_NEIGHBORS_COUNT(comm, 4 MPI_DIMS_CREATE(nnodes, ndims, dims), rank, nneighbors), 17 MPI_GRAPHDIMS_GET, 3, 14 5 MPI_DIST_GRAPH_CREATE, 2, 3, 8, 11-MPI_GRAPHDIMS_GET(comm, nnodes, nedges). 13, 20, 21, 25 14 MPI_Dist_graph_create, 11 MPI_INEIGHBOR_ALLGATHER, 3 MPI_DIST_GRAPH_CREATE(comm_old, n, MPI_INEIGHBOR_ALLGATHER(sendbuf, send+ sources, degrees, destinations, weights, count, sendtype, recybuf, recvcount, 11 info, reorder, comm_dist_graph), 10 recvtype, comm, request), 35 12MPI_DIST_GRAPH_CREATE_ADJACENT, MPI_INEIGHBOR_ALLGATHERV, 3 132, 3, 8, 9, 12, 20, 21, 25 MPI_INEIGHBOR_ALLGATHERV(sendbuf, 14 MPI_DIST_GRAPH_CREATE_ADJACENT(comm_oldndcount, sendtype, recvbuf, recv-15indegree, sources, sourceweights, outcounts, displs, recytype, comm, re-16 degree, destinations, destweights, info, quest), 3617reorder, comm_dist_graph), 8 MPI_INEIGHBOR_ALLTOALL, 3 18 MPI_DIST_GRAPH_NEIGHBOR_COUNT, MPI_INEIGHBOR_ALLTOALL(sendbuf, send- 19 count, sendtype, recvbuf, recvcount, 2120MPI_DIST_GRAPH_NEIGHBORS, 19, 20, recvtype, comm, request), 37 21MPI_INEIGHBOR_ALLTOALLV, 3 2522MPI_DIST_GRAPH_NEIGHBORS(comm, maximI_INEIGHBOR_ALLTOALLV(sendbuf, send23) counts, sdispls, sendtype, recvbuf, recv- 24 degree, sources, sourceweights, maxoutdegree, destinations, destweights), counts, rdispls, recvtype, comm, re-2520quest), 3826MPI_DIST_GRAPH_NEIGHBORS_COUNT, MPI_INEIGHBOR_ALLTOALLW, 3 27 19.20 MPI_INEIGHBOR_ALLTOALLW(sendbuf, sends MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, counts, sdispls, sendtypes, recvbuf, 29 indegree, outdegree, weighted), 19 recvcounts, rdispls, recvtypes, comm, 30 MPI_DIST_NEIGHBORS, 3 request), 39 31 MPI_DIST_NEIGHBORS_COUNT, 3 MPI_INEIGHBOR_REDUCE, 3 32 MPI_GRAPH_CREATE, 2, 3, 6, 12, 14, 15, MPI_INEIGHBOR_REDUCE(sendbuf, send-33 18, 24, 25 count, sendtype, recvbuf, recvcount, 34 MPI_GRAPH_CREATE(comm, nnodes, inrecvtype, op, comm, request), 40 35dex, edges, reorder, comm_graph), MPI_INEIGHBOR_REDUCEV, 3 36 24MPI_INEIGHBOR_REDUCEV(sendbuf, send-37 MPI_GRAPH_CREATE(comm_old, nnodes, counts, displs, sendtype, recvbuf, recv-38 index, edges, reorder, comm_graph), count, recvtype, op, comm), 4139 MPI_NEIGHBOR_ALLGATHER, 3, 27, 29, 6 40MPI_GRAPH_GET, 3, 14 3541 MPI_GRAPH_GET(comm, maxindex, maxedgesPI_NEIGHBOR_ALLGATHER(sendbuf, send42 index, edges), 15 count, sendtype, recvbuf, recvcount, 43MPI_GRAPH_MAP, 3 recvtype, comm), 26 44 MPI_GRAPH_MAP(comm, nnodes, index, MPI_NEIGHBOR_ALLGATHERV, 3, 36 45edges, newrank), 24, 24 MPI_NEIGHBOR_ALLGATHERV(sendbuf, 46MPI_GRAPH_NEIGHBORS, 3, 18, 25 sendcount, sendtype, recvbuf, recv-47counts, displs, recvtype, comm), 28 48

1	MPI_NEIGHBOR_ALLTOALL, 3, 30, 37
2	MPI_NEIGHBOR_ALLTOALL(sendbuf, send-
3	count, sendtype, recvbuf, recvcount,
4	recvtype, comm), $\underline{29}$
5	MPI_NEIGHBOR_ALLTOALLV, 3, 38
6	$\label{eq:MPI_NEIGHBOR_ALLTOALLV} (sendbuf, send-$
7	counts, sdispls, sendtype, recvbuf, recv-
8	counts, rdispls, recvtype, comm), $\underline{31}$
9	MPI_NEIGHBOR_ALLTOALLW, 3, 32, 39
10	MPI_NEIGHBOR_ALLTOALLW(sendbuf, send-
11	counts, sdispls, sendtypes, recvbuf,
12	recvcounts, rdispls, recvtypes, comm),
13	$\frac{32}{3}$
14	MPI_NEIGHBOR_REDUCE, 3, 40
15	MPI_Neighbor_reduce, 33
16	MPI_NEIGHBOR_REDUCE(sendbuf, recvbuf,
17	count, datatype, op, comm), $\underline{34}$
18	MPI_NEIGHBOR_REDUCEV, 3, 34, 41 MPI_NEIGHBOR_REDUCEV(sendbuf, send-
19	counts, displs, datatype, recvbuf, recv-
20 21	count, op, comm), 34
22	MPI_Reduce, 33
23	MPI_SENDRECV, 21
24	MPI_TOPO_TEST, 3, 14
25	MPI_TOPO_TEST(comm, status), <u>14</u>
26	(), <u>, , ,</u>
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40 41	
41	
43	
43	
45	
46	
47	
48	