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

Message Passing Interface Forum

October 21, 2020

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.

As stated in Chapter 7, 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 [5]. 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 22is 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.

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

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 7.

39 40 41

42

29 30

31

32 33 34

35

36

37

38

7.4 Overview of the Functions

MPI supports three topology types: Cartesian, graph, and distributed graph. The
 function MPI_CART_CREATE is used to create Cartesian topologies, the function
 MPI_GRAPH_CREATE is used to create graph topologies, and the functions
 MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE are used to cre-

ate distributed graph topologies. These topology creation functions are collective. As with

1

 $\mathbf{2}$

3 4 5

other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. For MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical values on all processes of the group of comm_old. When calling MPI_GRAPH_CREATE, each process specifies all nodes and edges in the graph. In contrast, the functions MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE are used to specify the graph in a distributed fashion, whereby each process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 7). In analogy to function MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol.

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

MPI defines functions to query a communicator for topology information. The function MPI_TOPO_TEST is used to query for the type of topology associated with a communicator. Depending on the topology type, different information can be extracted. For a graph topology, the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET return the values that were specified in the call to MPI_GRAPH_CREATE. Additionally, the functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to obtain the neighbors of an arbitrary node in the graph. For a distributed graph topology, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS can be used to obtain the neighbors of the calling process. For a Cartesian topology, the functions MPI_CARTDIM_GET and MPI_CART_GET return the values that were specified in the call to MPI_CART_CREATE. Additionally, the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local.

For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group.

The two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP, are, in general, not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 7, 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, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL,

MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topology associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, 47 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and 48

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

7

8

9

10

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

40

41

42

43

MPI_INEIGHBOR_ALLTOALLW. 2 **Topology Constructors** 7.5 7.5.1 Cartesian Constructor 6 MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) 9 IN comm_old input communicator (handle) 10 11IN ndims number of dimensions of Cartesian grid (integer) 12dims IN integer array of size ndims specifying the number of 13 processes in each dimension 14 IN periods logical array of size ndims specifying whether the grid 15is periodic (true) or not (false) in each dimension 16 17IN reorder ranking may be reordered (true) or not (false) 18 (logical) 19 OUT communicator with new Cartesian topology (handle) comm_cart 2021int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], 22const int periods[], int reorder, MPI_Comm *comm_cart) 23 24MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror) 25TYPE(MPI_Comm), INTENT(IN) :: comm_old 26 INTEGER, INTENT(IN) :: ndims, dims(ndims) 27LOGICAL, INTENT(IN) :: periods(ndims), reorder 28TYPE(MPI_Comm), INTENT(OUT) :: comm_cart 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR) INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR 32LOGICAL PERIODS(*), REORDER 33 34 MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian

35topology information is attached. If reorder = false then the rank of each process in the 36 new group is identical to its rank in the old group. Otherwise, the function may reorder 37 the processes (possibly so as to choose a good embedding of the virtual topology onto 38 the physical machine). If the total size of the Cartesian grid is smaller than the size of 39 the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to 40 MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created. 41 The call is erroneous if it specifies a grid that is larger than the group size or if ndims is 42negative. 43

44 Cartesian Convenience Function: MPI_DIMS_CREATE 7.5.2 45

For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced 46distribution of processes per coordinate direction, depending on the number of processes 47in the group to be balanced and optional constraints that can be specified by the user. 48

1

3

4 $\mathbf{5}$

> 7 8

One use is to partition all the processes (the size of MPI_COMM_WORLD's group) into an *n*-dimensional topology.

MPI_DIMS_CREATE(nnodes, ndims, dims)						
IN nnodes number of nodes in a grid (integer)						
IN	ndims	number of Cartesian dimensions (integer)				
INOUT	integer array of size ndims specifying the number of nodes in each dimension					
int MPI_Di	<pre>int MPI_Dims_create(int nnodes, int ndims, int dims[])</pre>					
MPI_Dims_create(nnodes, ndims, dims, ierror) INTEGER, INTENT(IN) :: nnodes, ndims INTEGER, INTENT(INOUT) :: dims(ndims) INTEGER, OPTIONAL, INTENT(OUT) :: ierror						
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR) INTEGER NNODES, NDIMS, DIMS(*), IERROR						

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

Negative input values of dims[i] are erroneous. An error will occur if nnodes is not a multiple of

$$\prod_{i,dims[i]\neq 0} dims[i].$$

For dims[i] set by the call, dims[i] will be ordered in non-increasing order. Array dims is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is local. If ndims is zero and nnodes is one, MPI_DIMS_CREATE returns MPI_SUCCESS.

Example 7.1

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

6

5

7.5.3 Graph Constructor

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

```
IN
                 comm_old
                                             input communicator (handle)
6
       IN
                 nnodes
                                             number of nodes in graph (integer)
7
8
       IN
                 index
                                             array of integers describing node degrees (see below)
9
       IN
                 edges
                                             array of integers describing graph edges (see below)
10
       IN
                 reorder
                                             ranking may be reordered (true) or not (false)
11
                                             (logical)
12
13
       OUT
                 comm_graph
                                             communicator with graph topology added (handle)
14
15
     int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],
16
                    const int edges[], int reorder, MPI_Comm *comm_graph)
17
     MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
18
                    ierror)
19
          TYPE(MPI_Comm), INTENT(IN) :: comm_old
20
          INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
21
          LOGICAL, INTENT(IN) :: reorder
22
          TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25
     MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
26
                    IERROR)
27
          INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
28
          LOGICAL REORDER
29
```

MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph 30 topology information is attached. If reorder = false then the rank of each process in the 31 new group is identical to its rank in the old group. Otherwise, the function may reorder the 32processes. If the size, nnodes, of the graph is smaller than the size of the group of comm_old, 33 then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE 34 and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL 35 is returned in all processes. The call is erroneous if it specifies a graph that is larger than 36 the group size of the input communicator. 37

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

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

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

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

Then, the input arguments are:

nnodes =	4	10
index =	2, 3, 4, 6	11
edges =	1, 3, 0, 3, 0, 2	12
) -) -) -) -)	13

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),	32
• For a Cartesian topology:	33
1. ndims (number of dimensions),	34 35
2. dims (numbers of processes per coordinate direction),	36
3. periods (periodicity information),	37
4. own_position (own position in grid, could also be computed from rank and dims)	38
• For a graph topology:	39 40
1. index,	41
2. edges,	42
which are the vectors defining the graph structure.	43
	44
For a graph structure the number of nodes is equal to the number of processes in	45
the group. Therefore, the number of nodes does not have to be stored explicitly.	46
An additional zero entry at the start of array index simplifies access to the topology	47
information. (End of advice to implementors.)	48

Unofficial Draft for Comment Only

 $25 \\ 26$

7.5.4 Distributed Graph Constructor 1

 $\mathbf{2}$ MPI_GRAPH_CREATE requires that each process passes the full (global) communication 3 graph to the call. This limits the scalability of this constructor. With the distributed graph 4 interface, the communication graph is specified in a fully distributed fashion. Each process 5specifies only the part of the communication graph of which it is aware. Typically, this 6 could be the set of processes from which the process will eventually receive or get data. or the set of processes to which the process will send or put data, or some combination of 8 such edges. Two different interfaces can be used to create a distributed graph topology. 9 MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph communicator with 10 each process specifying each of its incoming and outgoing (adjacent) edges in the logical 11communication graph and thus requires minimal communication during creation. 12

MPI_DIST_GRAPH_CREATE provides full flexibility such that any process can indicate that 13 communication will occur between any pair of processes in the graph. 14

To provide better possibilities for optimization by the MPI library, the distributed 15graph constructors permit weighted communication edges and take an info argument that 16 can further influence process reordering or other optimizations performed by the MPI library. 17For example, hints can be provided on how edge weights are to be interpreted, the quality 18 of the reordering, and/or the time permitted for the MPI library to process the graph. 19

2021

2223

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, source	weights,
outdegree, destinations, destweights, info, reorder, comm_dist_g	raph)

24	IN	comm_old	input communicator (handle)
25 26	IN	indegree	size of sources and source weights 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 28	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 (handle)
46 47	int MPI_	Dist_graph_create_adja const int source	acent(MPI_Comm comm_old, int indegree, s[], const int sourceweights[], int outdegree,

```
const int destinations[], const int destweights[],
             MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
             outdegree, destinations, destweights, info, reorder,
             comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),
              outdegree, destinations(outdegree), destweights(*)
    TYPE(MPI_Info), INTENT(IN) :: info
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
             OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
             COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
              DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH,
              IERROR
    LOGICAL REORDER
```

MPI_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 its incoming and outgoing edges 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 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.

38 Weights are specified as non-negative integers and can be used to influence the process 39 remapping strategy and other internal MPI optimizations. For instance, approximate count arguments of later communication calls along specific edges could be used as their edge 41 weights. Multiplicity of edges can likewise indicate more intense communication between 42 pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 43standard and is left to the implementation. In C or Fortran, an application can supply 44 the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have 45the same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some but not all processes of comm_old. If the graph is weighted but indegree or outdegree is zero, then MPI_WEIGHTS_EMPTY or any arbitrary array may be passed to sourceweights

1

2 3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

2021

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

40

46

47

1	or destweig	hts respectively. Note that M	API LINWEIGHTED and MPI WEIGHTS EMPTY are				
2	not special weight values; rather they are special values for the total array argument. In						
3	Fortran, MPI UNWEIGHTED and MPI WEIGHTS EMPTY are objects like MPI BOTTOM (not						
4	usable for initialization or assignment). See Section 2.5.4.						
5							
6	Advice to users. In the case of an empty weights array argument passed while						
7	constructing a weighted graph, one should not pass NULL because the value of						
8	MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then						
9 10	be indistinguishable from MPI_UNWEIGHTED to the implementation. In this case MPI_WEIGHTS_EMPTY_should be used instead (End of advice to users)						
11	_	_					
12	Advic	the to implementors. It is rec	ommended that MPI_UNWEIGHTED not be imple-				
13	mente	ed as NULL. (End of advice to	implementors.)				
14 15	Ratio	nale. To ensure backward co	mpatibility, $MPI_UNWEIGHTED$ may still be imple-				
16	mente	ed as NULL. See Annex B.4.	(End of rationale.)				
17							
18	f-lli	leaning of the info and reorde	er arguments is defined in the description of the				
19	following re	butine.					
20							
21 22	MPI_DIST_	.GRAPH_CREATE(comm_old, reorder, comm_dist_graph	n, sources, degrees, destinations, weights, info,				
23	INI		;,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,				
24	IIN	comm_old	input communicator (nandie)				
25 26	IN	n	number of source nodes for which this process specifies edges (non-negative integer)				
27	IN	sources	array containing the n source nodes for which this				
28			process specifies edges (array of non-negative				
29			integers)				
30	IN	degrees	array specifying the number of destinations for each				
31		degrees	source node in the source node array (array of				
32			non-negative integers)				
33							
34	IN	destinations	destination nodes for the source nodes in the source				
35			node array (array of non-negative integers)				
36	IN	weights	weights for source to destination edges (array of				
31			non-negative integers)				
39	IN	info	hints on optimization and interpretation of weights				
40			(handle)				
41	IN	reorder	the ranks may be reordered (true) or not (false)				
42			(logical)				
43	OUT	comm_dist_graph	communicator with distributed graph topology				
44			added (handle)				
45							
46	· ·						

```
const int weights[], MPI_Info info, int reorder,
             MPI_Comm *comm_dist_graph)
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
              info, reorder, comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*),
              weights(*)
    TYPE(MPI_Info), INTENT(IN) :: info
    LOGICAL, INTENT(IN) :: reorder
                                                                                  10
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                  11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
                                                                                  13
              INFO, REORDER, COMM_DIST_GRAPH, IERROR)
                                                                                  14
    INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
                                                                                  15
              WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
                                                                                  16
    LOGICAL REORDER
                                                                                  17
```

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

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

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

Weights are specified as non-negative integers and can be used to influence the process 45remapping strategy and other internal MPI optimizations. For instance, approximate count 46 arguments of later communication calls along specific edges could be used as their edge weights. Multiplicity of edges can likewise indicate more intense communication between

Unofficial Draft for Comment Only

1

2 3

5

6

7

8

9

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

47

pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 1 standard and is left to the implementation. In C or Fortran, an application can supply 2 3 the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some but not 4 all processes of comm_old. If the graph is weighted but n = 0, then MPI_WEIGHTS_EMPTY 5or any arbitrary array may be passed to weights. Note that MPI_UNWEIGHTED and 6 MPI_WEIGHTS_EMPTY are not special weight values; rather they are special values for the 7 total array argument. In Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects 8 like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4. 9 10 In the case of an empty weights array argument passed while Advice to users. 11 constructing a weighted graph, one should not pass NULL because the value of 12MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then 13 be indistinguishable from MPI_UNWEIGHTED to the implementation. 14 MPI_WEIGHTS_EMPTY should be used instead. (End of advice to users.) 1516 Advice to implementors. It is recommended that MPI_UNWEIGHTED not be imple-17mented as NULL. (End of advice to implementors.) 18 19 *Rationale.* To ensure backward compatibility, MPI_UNWEIGHTED may still be imple-20mented as NULL. See Annex B.4. (End of rationale.) 21The meaning of the weights argument can be influenced by the info argument. Info 22arguments can be used to guide the mapping; possible options include minimizing the 23 maximum number of edges between processes on different SMP nodes, or minimizing the 24sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it 25is valid for an MPI implementation not to do any reordering. An MPI implementation may 26 specify more info key-value pairs. All processes must specify the same set of key-value info 27 pairs. 2829 Advice to implementors. MPI implementations must document any additionally 30 supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the 31 default creation of the distributed graph topology to the MPI library. 32 An implementation does not explicitly need to construct the topology from its dis-33 tributed parts. However, all processes can construct the full topology from the dis-34

tributed 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*.)

39 40 41

42

43

35

36

37

38

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:

44	process	neighbors
45	0	1, 3
46	1	0
47	2	3
48	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 $\mathsf{MPI_UNWEIGHTED}$ instead of explicitly providing identical weights.

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

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

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

```
/*
```

MPI_Comm comm_dist_graph;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

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

 $\mathbf{2}$

```
1
     /* get my communication partners along x dimension */
2
     destinations[0] = P*y+(x+1)%P; weights[0] = 2;
3
     destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
4
5
     /* get my communication partners along y dimension */
6
     destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
7
     destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;
8
9
     /* get my communication partners along diagonals */
10
     destinations[4] = P*((y+1))(Q)+(x+1)(P); weights[4] = 1;
11
     destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
12
     destinations[6] = P*((y+1))(Q)+(P+x-1)(P); weights[6] = 1;
13
     destinations[7] = P*((Q+y-1))(Q)+(P+x-1)(P); weights[7] = 1;
14
15
     sources[0] = rank;
16
     degrees [0] = 8;
17
     MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
18
                             weights, MPI_INFO_NULL, 1, &comm_dist_graph);
19
20
21
     7.5.5 Topology Inquiry Functions
22
     If a topology has been defined with one of the above functions, then the topology information
23
     can be looked up using inquiry functions. They all are local calls.
24
25
26
     MPI_TOPO_TEST(comm, status)
27
       IN
                                           communicator (handle)
                comm
28
29
       OUT
                status
                                           topology type of communicator comm (state)
30
31
     int MPI_Topo_test(MPI_Comm comm, int *status)
32
     MPI_Topo_test(comm, status, ierror)
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         INTEGER, INTENT(OUT) :: status
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_TOPO_TEST(COMM, STATUS, IERROR)
38
         INTEGER COMM, STATUS, IERROR
39
         The function MPI_TOPO_TEST returns the type of topology that is assigned to a
40
     communicator.
41
         The output value status is one of the following:
42
43
       MPI_GRAPH
                                            graph topology
44
                                           Cartesian topology
       MPI_CART
45
       MPI_DIST_GRAPH
                                           distributed graph topology
46
       MPI_UNDEFINED
                                           no topology
47
48
```

MPI_GRA	PHDIMS_GE1(com	im, nnodes, nedges)	1
IN	comm	communicator for group with graph structure (handle)	2 3
OUT	nnodes	number of nodes in graph (same as number of processes in the group) (integer)	4 5 6
OUT	nedges	number of edges in graph (integer)	7 8
int MPI_	Graphdims_get(MP	I_Comm comm, int *nnodes, int *nedges)	9
MPI_Grap	hdims_get(comm,	nnodes, nedges, ierror)	10 11
TYPE	(MPI_Comm), INTE	NT(IN) :: comm	12
INTE	GER, INTENT(OUT)	:: nnodes, nedges	13
INTE	GER, OPTIONAL, I	NTENT(OUT) :: ierror	14
MPT GRAP	HDIMS GET(COMM.	NNODES, NEDGES, TEBROR)	15
INTE	GER COMM, NNODES	, NEDGES, IERROR	16
T			17
Funct	tions MPI_GRAPHL	DIMS_GET and MPI_GRAPH_GET retrieve the graph-topology	18
The	In that was associated associated as the second sec	ad by MPL CRAPHDIMS CET can be used to dimension the	19
vectors in	dex and edges corre	betty for the following call to MPL $GRAPH$ GET	20
VCCUOIS III	dex and edges corre		21
			22
MPI_GRA	PH_GET(comm, ma	axindex, maxedges, index, edges)	24
IN	comm	communicator with graph structure (handle)	25
IN	maxindex	length of vector index in the calling program (integer)	26
IN	maxedges	length of vector edges in the calling program (integer)	27 28
OUT	index	array of integers containing the graph structure (for details see the definition of MPI_GRAPH_CREATE)	29 30
OUT	edges	array of integers containing the graph structure	31 32
int MPT	Graph get(MPI Co	mm comm, int maxindex, int maxedges, int index[].	33
	int edges[])	34
	0		35
MP1_Grap	h_get(comm, maxi	ndex, maxedges, index, edges, ierror)	36
TYPE	(MPI_Comm), INTE	NT(IN) :: comm	37
	GER, INIENI(IN) GER INTENT(OUT)	:: maximuex, maxedges :: index(maximdex) edges(maxedges)	38
INTE TNTE	GER, INTENI(UOI) GER OPTIONAL T	NTENT(OUT) ·· jerror	39
11110			40
MPI_GRAP	H_GET(COMM, MAXI	NDEX, MAXEDGES, INDEX, EDGES, IERROR)	42
INTE	GER COMM, MAXIND	EX, MAXEDGES, INDEX(*), EDGES(*), IERROR	43
			44
			45
			46
			47

MDI CONDUDING CET(andra nadrac)

```
MPI_CARTDIM_GET(comm, ndims)
1
\mathbf{2}
       IN
                 comm
                                             communicator with Cartesian structure (handle)
3
       OUT
                 ndims
                                             number of dimensions of the Cartesian structure
4
                                             (integer)
5
6
     int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
7
8
     MPI_Cartdim_get(comm, ndims, ierror)
9
          TYPE(MPI_Comm), INTENT(IN) :: comm
10
          INTEGER, INTENT(OUT) :: ndims
11
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
13
          INTEGER COMM, NDIMS, IERROR
14
15
          The functions MPI_CARTDIM_GET and MPI_CART_GET return the Cartesian topol-
16
     ogy information that was associated with a communicator by MPI_CART_CREATE. If comm
17
     is associated with a zero-dimensional Cartesian topology, MPI_CARTDIM_GET returns
18
     ndims = 0 and MPI_CART_GET will keep all output arguments unchanged.
19
20
21
     MPI_CART_GET(comm, maxdims, dims, periods, coords)
22
       IN
                 comm
                                             communicator with Cartesian structure (handle)
23
       IN
                 maxdims
                                             length of vectors dims, periods, and coords in the
24
                                             calling program (integer)
25
26
       OUT
                 dims
                                             number of processes for each Cartesian dimension
27
                                             (array of integers)
28
       OUT
                 periods
                                             periodicity (true/false) for each Cartesian dimension
29
                                             (array of logicals)
30
       OUT
                 coords
                                             coordinates of calling process in Cartesian structure
31
                                             (array of integers)
32
33
34
     int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
                     int coords[])
35
36
     MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
37
          TYPE(MPI_Comm), INTENT(IN) :: comm
38
          INTEGER, INTENT(IN) :: maxdims
39
          INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
40
          LOGICAL, INTENT(OUT) :: periods(maxdims)
41
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
          INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
44
45
          LOGICAL PERIODS(*)
46
47
48
```

MPI_CART	_RANK(comm, coords, rank)		1
IN	comm	communicator with Cartesian structure (handle)	2
IN	coords	integer array (of size ndims) specifying the Cartesian coordinates of a process	3 4
OUT	rank	rank of specified process (integer)	5
001	Idlik	Tank of specified process (integer)	7
int MPI_C	art_rank(MPI_Comm comm, c	const int coords[], int *rank)	8
MPI_Cart_	rank(comm, coords, rank,	ierror)	9 10
TYPE(MPI_Comm), INTENT(IN) ::	comm	11
INTEG	ER, INTENT(IN) :: coords(*)	12
INTEG	ER, INTENT(OUT) :: rank		13
INTEG.	ER, UPIIUNAL, INIENI(UUI)	:: lerror	14
MPI_CART_	RANK (COMM, COORDS, RANK,	IERROR)	15
INTEG	ER COMM, COORDS(*), RANK,	IERROR	16
For a	communicator with an associ	ated Cartesian topology, the function	18
MPI_CART	_RANK translates the logical	process coordinates to process ranks as they are	19
used by the	e point-to-point routines.		20
For di	mension i with periods(i) = tru	e, if the coordinate, coords(i), is out of range, that	21
1s, coords(1)	$0 < 0$ or coords(1) \geq dims(1), it	Out of range coordinates are erroneous for non	22
$0 \leq coords$	mensions	Out-of-range coordinates are erroneous for non-	23
If com	m is associated with a zero-di	mensional Cartesian topology, coords is not signif-	24 25
icant and 0	is returned in rank.	1 007 0	26
			27
MPI CART	COORDS(comm. rank. maxd	ims. coords)	28
INI		communicator with Cartesian structure (handle)	29
			30
IN	rank	rank of a process within group of comm (integer)	32
IN	maxdims	length of vector coords in the calling program	33
		(integer)	34
OUT	coords	integer array (of size maxdims) containing the	35
		Cartesian coordinates of specified process (array of	36
		integers)	37
int MDT C	ant accords (MDT Comm comm	int work int mondime int counde[])	38
int MPI_C	art_coords(MP1_Comm comm,	int rank, int maxdims, int coords[])	39 40
MPI_Cart_	coords(comm, rank, maxdim	s, coords, ierror)	40
TYPE()	MPI_Comm), INTENT(IN) ::	comm	42
INTEG	ER, INTENT(IN) :: rank, m	axdims	43
INTEG.	ER, INIENI(UUI) :: COOTAS FR APTIANNI INTENT(AUTA)	· jerror	44
111 I II (J.			45
MPI_CART_			
TNITTO	COURDS (COMM, RANK, MAXDIM	S, COORDS, IERROR)	46

```
The inverse mapping, rank-to-coordinates translation is provided by
1
     MPI_CART_COORDS.
2
3
         If comm is associated with a zero-dimensional Cartesian topology,
     coords will be unchanged.
4
5
6
     MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)
7
8
       IN
                 comm
                                            communicator with graph topology (handle)
9
       IN
                 rank
                                            rank of process in group of comm (integer)
10
       OUT
                 nneighbors
                                            number of neighbors of specified process (integer)
11
12
     int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
13
14
     MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
15
          TYPE(MPI_Comm), INTENT(IN) :: comm
16
          INTEGER, INTENT(IN) :: rank
17
          INTEGER, INTENT(OUT) :: nneighbors
18
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
     MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
20
          INTEGER COMM, RANK, NNEIGHBORS, IERROR
21
22
23
24
     MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)
25
       IN
                comm
                                            communicator with graph topology (handle)
26
       IN
                 rank
                                            rank of process in group of comm (integer)
27
28
                maxneighbors
       IN
                                            size of array neighbors (integer)
29
       OUT
                 neighbors
                                            ranks of processes that are neighbors to specified
30
                                            process (array of integers)
31
32
     int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
33
                    int neighbors[])
34
35
     MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
36
          TYPE(MPI_Comm), INTENT(IN) :: comm
37
          INTEGER, INTENT(IN) :: rank, maxneighbors
38
          INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
39
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
41
          INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
42
43
          MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency
44
     information for a graph topology. The returned count and array of neighbors for the queried
45
     rank will both include all neighbors and reflect the same edge ordering as was specified by
46
     the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT
47
     and MPI_GRAPH_NEIGHBORS will return values based on the original index and edges array
48
```

passed to MPI_GRAPH_CREATE (for the purpose of this example, we assume that index[-1] is zero):

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

Example 7.5 Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix (note that some neighbors are listed multiple times):

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

Thus, the input arguments to MPI_GRAPH_CREATE are:

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

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

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

Example 7.6 Suppose that comm is a communicator with a shuffle-exchange topology. The group has 2^n members. Each process is labeled by a_1, \ldots, a_n with $a_i \in \{0, 1\}$, and has three neighbors: exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ ($\bar{a} = 1 - a$), shuffle $(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$, and unshuffle $(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is illustrated below for n = 3.

1	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

Unofficial Draft for Comment Only

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

```
4
     ! assume: each process has stored a real number A.
\mathbf{5}
     ! extract neighborhood information
6
     CALL MPI_COMM_RANK(comm, myrank, ierr)
7
     CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
8
     ! perform exchange permutation
9
     CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0, &
10
                                 neighbors(1), 0, comm, status, ierr)
11
     ! perform shuffle permutation
12
     CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0, &
13
                                 neighbors(3), 0, comm, status, ierr)
14
     ! perform unshuffle permutation
15
     CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0, &
16
                                 neighbors(2), 0, comm, status, ierr)
17
18
         MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS pro-
19
     vide adjacency information for a distributed graph topology.
20
21
22
     MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, indegree, outdegree, weighted)
23
       IN
                comm
                                           communicator with distributed graph topology
24
                                           (handle)
25
26
       OUT
                indegree
                                           number of edges into this process (non-negative
27
                                           integer)
28
       OUT
                outdegree
                                           number of edges out of this process (non-negative
29
                                           integer)
30
       OUT
                weighted
                                           false if MPI_UNWEIGHTED was supplied during
31
                                           creation, true otherwise (logical)
32
33
     int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
34
                    int *outdegree, int *weighted)
35
36
     MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         INTEGER, INTENT(OUT) :: indegree, outdegree
39
         LOGICAL, INTENT(OUT) :: weighted
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
         INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
43
         LOGICAL WEIGHTED
44
45
46
47
48
```

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

IN	comm	communicator with distributed graph topology (handle)	3 4
IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)	5 6 7
OUT	sources	processes for which the calling process is a destination (array of non-negative integers)	8 9
OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)	10 11
IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)	12 13
OUT	destinations	processes for which the calling process is a source (array of non-negative integers)	14 15 16
OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)	17 18 19
int MPI_D	<pre>ist_graph_neighbors(MPI_C int sourceweights[], int destweights[])</pre>	<pre>omm comm, int maxindegree, int sources[], int maxoutdegree, int destinations[],</pre>	20 21 22 23
MPI_Dist_, TYPE() INTEG INTEG INTEG INTEG	<pre>graph_neighbors(comm, max maxoutdegree, desting MPI_Comm), INTENT(IN) :: ER, INTENT(IN) :: maxinde ER, INTENT(OUT) :: source destinations(maxoutd ER :: sourceweights(*), d ER, OPTIONAL, INTENT(OUT)</pre>	<pre>indegree, sources, sourceweights, ations, destweights, ierror) comm gree, maxoutdegree s(maxindegree), egree) estweights(*) :: ierror</pre>	24 25 26 27 28 29 30 31
MPI_DIST_0	GRAPH_NEIGHBORS(COMM, MAX MAXOUTDEGREE, DESTINA ER COMM, MAXINDEGREE, SOU DESTINATIONS(*), DES	INDEGREE, SOURCES, SOURCEWEIGHTS, ATIONS, DESTWEIGHTS, IERROR) RCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, TWEIGHTS(*), IERROR	32 33 34 35
These MPI_DIST_	calls are local. The number _GRAPH_NEIGHBORS_COUN	of edges into and out of the process returned by T are the total number of such edges given in the	30 37 38

call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (poten-39 tially by processes other than the calling process in the case of 40MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by 41 MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for 42sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied during the con-43struction of the graph then no weight information is returned in that array or those arrays. 44If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then for 45each rank in comm, the order of the values in sources and destinations is identical to the in-46put that was used by the process with the same rank in comm_old in the creation call. If the 47communicator was created with MPI_DIST_GRAPH_CREATE then the only requirement on 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

the order of values in sources and destinations is that two calls to the routine with same in-1 put argument comm will return the same sequence of edges. If maxindegree or maxoutdegree 2 is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT, then 3 only the first part of the full list is returned. 4

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 12

13 If the process topology is a Cartesian structure, an MPI_SENDRECV operation may be used 14 along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes 15the rank of a source process for the receive, and the rank of a destination process for the 16 send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides 17the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. 18 The user specifies the coordinate direction and the size of the step (positive or negative). 19 The function is local. 20

```
21
22
```

48

 $\mathbf{5}$

6

7

8

9

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

0.2			
23 24	IN	comm	communicator with Cartesian structure (handle)
25	IN	direction	coordinate dimension of shift (integer)
26 27	IN	disp	displacement (> 0: upwards shift, < 0: downwards shift) (integer)
28 29	OUT	rank_source	rank of source process (integer)
30	OUT	rank_dest	rank of destination process (integer)
32 33 34 35 36 37 38	int MPI_C MPI_Cart_ TYPE(INTEG INTEG	<pre>art_shift(MPI_Comm comm,</pre>	<pre>int direction, int disp, t *rank_dest) isp, rank_source, rank_dest, ierror) comm ion, disp source, rank_dest</pre>
39 40	INTEG	ER, OPTIONAL, INTENT(OUT) SHIFT(COMM, DIRECTION, DI) :: ierror ISP, RANK_SOURCE, RANK_DEST, IERROR)
41 42 43	The d	irection argument indicates the	e coordinate dimension to be traversed by the shift.
44 45	The dimen Deper	sions are numbered from 0 to ading on the periodicity of the CART_SHIFT provides the ide	ndims-1, where ndims is the number of dimensions. e Cartesian group in the specified coordinate direc- entifiers for a circular or an end-off shift. In the case
46 47 48	of an end-of indicating	off shift, the value MPI_PROC_ that the source or the destina	NULL may be returned in rank_source or rank_dest, ation for the shift is out of range.

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

Example 7.7 The communicator, **comm**, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of **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 column) by **i** steps.

```
11
! find process rank
                                                                                          12
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                          13
! find Cartesian coordinates
                                                                                          14
CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
                                                                                          15
! compute shift source and destination
                                                                                          16
CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
                                                                                          17
! skew array
                                                                                          18
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
                                                                                          19
                             status, ierr)
                                                                                          20
                                                                                          21
     Advice to users. In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1)
                                                                                          22
     nodes, where DIMS is the array that was used to create the grid. In C, the dimension
                                                                                          23
     indicated by direction = i is the dimension specified by dims[i]. (End of advice
                                                                                          24
     to users.)
                                                                                          25
                                                                                          26
7.5.7 Partitioning of Cartesian Structures
                                                                                          27
                                                                                          28
                                                                                          29
MPI_CART_SUB(comm, remain_dims, newcomm)
                                                                                          30
  IN
            comm
                                        communicator with Cartesian structure (handle)
                                                                                          31
                                                                                          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
                                                                                          34
                                        dropped (false) (array of logicals)
                                                                                          35
  OUT
           newcomm
                                        communicator containing the subgrid that includes
                                                                                          36
                                        the calling process (handle)
                                                                                          37
                                                                                          38
int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)
                                                                                          39
                                                                                          40
MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
                                                                                          41
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          42
    LOGICAL, INTENT(IN) :: remain_dims(*)
                                                                                          43
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                          44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          45
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
                                                                                          46
    INTEGER COMM, NEWCOMM, IERROR
                                                                                          47
    LOGICAL REMAIN_DIMS(*)
                                                                                          48
```

1

2

3

4 5

6

7

8

If a Cartesian topology has been created with MPI_CART_CREATE, the function 1 MPI_CART_SUB can be used to partition the communicator group into subgroups that 2 3 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 4 comm is already associated with a zero-dimensional Cartesian topology then newcomm is $\mathbf{5}$ associated with a zero-dimensional Cartesian topology. (This function is closely related to 6 MPI_COMM_SPLIT.) 7 8 **Example 7.8** Assume that MPI_CART_CREATE(..., comm) has defined a $(2 \times 3 \times 4)$ grid. 9 Let remain_dims = (true, false, true). Then a call to 10 11 MPI_CART_SUB(comm, remain_dims, comm_new); 1213 will create three communicators each with eight processes in a 2×4 Cartesian topology. If remain_dims = (false, false, true) then the call to $MPI_CART_SUB(comm, remain_dims,$ 14comm_new) will create six non-overlapping communicators, each with four processes, in a 1516one-dimensional Cartesian topology. 1718 7.5.8 Low-Level Topology Functions 19 The two additional functions introduced in this section can be used to implement all other 20topology functions. In general they will not be called by the user directly, unless he or she 21is creating additional virtual topology capability other than that provided by MPI. The two 22calls are both local. 23 2425MPI_CART_MAP(comm, ndims, dims, periods, newrank) 26IN comm input communicator (handle) 2728IN ndims number of dimensions of Cartesian structure (integer) 29 IN dims integer array of size ndims specifying the number of 30 processes in each coordinate direction 31 periods IN logical array of size ndims specifying the periodicity 32specification in each coordinate direction 33 34OUT reordered rank of the calling process; newrank 35MPI_UNDEFINED if calling process does not belong 36 to grid (integer) 37 38 int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[], 39 const int periods[], int *newrank) 40 MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror) 41 TYPE(MPI_Comm), INTENT(IN) :: comm 42INTEGER, INTENT(IN) :: ndims, dims(ndims) 43LOGICAL, INTENT(IN) :: periods(ndims) 44 INTEGER, INTENT(OUT) :: newrank 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46 47MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR) 48

INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR 1 LOGICAL PERIODS(*) 2 3 MPI_CART_MAP computes an "optimal" placement for the calling process on the phys-4 ical machine. A possible implementation of this function is to always return the rank of the 5calling process, that is, not to perform any reordering. 6 7 Advice to implementors. The function MPI_CART_CREATE(comm, ndims, dims, 8 periods, reorder, comm_cart), with reorder = true can be implemented by calling 9 MPI_CART_MAP(comm, ndims, dims, periods, newrank), then calling 10 MPI_COMM_SPLIT(comm, color, key, comm_cart), with color = 0 if newrank \neq 11 MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank. If ndims 12 is zero then a zero-dimensional Cartesian topology is created. 13 The function MPI_CART_SUB(comm, remain_dims, comm_new) can be implemented 14 by a call to MPI_COMM_SPLIT(comm, color, key, comm_new), using a single number 15 encoding of the lost dimensions as color and a single number encoding of the preserved 16 dimensions as key. 17All other Cartesian topology functions can be implemented locally, using the topology 18 information that is cached with the communicator. (End of advice to implementors.) 19 20The corresponding function for graph structures is as follows. 2122 23 MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank) 24IN comm input communicator (handle) 2526IN nnodes number of graph nodes (integer) 27IN index integer array specifying the graph structure, see 28MPI_GRAPH_CREATE 29 IN edges integer array specifying the graph structure 30 31 OUT newrank reordered rank of the calling process; 32 MPI_UNDEFINED if the calling process does not 33 belong to graph (integer) 34 35int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[], 36 const int edges[], int *newrank) 37 MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) 38 TYPE(MPI_Comm), INTENT(IN) :: comm 39 INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*) 40 INTEGER, INTENT(OUT) :: newrank 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42 43MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) 44 INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR 4546 Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, 47

edges, reorder, comm_graph), with reorder = true can be implemented by calling

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_graph), with color = 0 if newrank \neq

MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank.

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

7.6 Neighborhood Collective Communication on Process Topologies

MPI process topologies specify a communication graph, but they implement no communication function themselves. Many applications require sparse nearest neighbor communications that can be expressed as graph topologies. We now describe several collective operations that perform communication along the edges of a process topology. All of these functions are collective; i.e., they must be called by all processes in the specified communicator. See Section 6 for an overview of other dense (global) collective communication operations and the semantics of collective operations.

¹⁶ If the graph was created with MPI_DIST_GRAPH_CREATE_ADJACENT with sources ¹⁷ and destinations containing 0, ..., n-1, where n is the number of processes in the group ¹⁹ of comm_old (i.e., the graph is fully connected and also includes an edge from each node ²⁰ to itself), then the sparse neighborhood communication routine performs the same data ²¹ exchange as the corresponding dense (fully-connected) collective operation. In the case of a ²² Cartesian communicator, only nearest neighbor communication is provided, corresponding ²³ to rank_source and rank_dest 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 [4]. This functionality can significantly simplify the implementation of neighbor exchanges [3]. (End of rationale.)

For a distributed graph topology, created with MPI_DIST_GRAPH_CREATE, the se-31 quence of neighbors in the send and receive buffers at each process is defined as the sequence 32returned by MPI_DIST_GRAPH_NEIGHBORS for destinations and sources, respectively. For 33 a general graph topology, created with MPI_GRAPH_CREATE, the use of neighborhood col-34lective communication is restricted to adjacency matrices, where the number of edges be-35 tween any two processes is defined to be the same for both processes (i.e., with a symmetric 36 adjacency matrix). In this case, the order of neighbors in the send and receive buffers is 37 defined as the sequence of neighbors as returned by MPI_GRAPH_NEIGHBORS. Note that 38 general graph topologies should generally be replaced by the distributed graph topologies. 39

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 collec tive communication behaves like a point-to-point communication with MPI_PROC_NULL in

1

2

3 4

5

6 7

8 9

10

11

12

13

14

15

24

25

26

27

28

29

this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

7.6.1 Neighborhood Gather

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

MPI_NEIGHBOR_ALLGATHER(sendb	uf, sendcount	, sendtype,	recvbuf,	recvcount,	recvtype,
comm)					

			13
IN	sendbuf	starting address of send buffer (choice)	14
IN	sendcount	number of elements sent to each neighbor	15
		(non-negative integer)	16
IN	sendtype	data type of send buffer elements (handle)	17
	recybuf	starting address of receive buffer (choice)	18
001	TCCVDUT	starting address of receive build (choice)	19
IN	recvcount	number of elements received from each neighbor	20
		(non-negative integer)	21
IN	recytype	data type of receive buffer elements (handle)	22
			23
IN	comm	communicator with topology structure (handle)	24

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

```
MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(..) :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
```

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. 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);

1

2 3

4 5

6

7

8

25

26

27

28 29

30

31

32

33

34

35

36

37

38

39

40

41 42

43

44

45

```
int *srcs=(int*)malloc(indegree*sizeof(int));
1
      int *dsts=(int*)malloc(outdegree*sizeof(int));
\mathbf{2}
3
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                    outdegree, dsts, MPI_UNWEIGHTED);
4
      int k,l;
\mathbf{5}
6
      /* assume sendbuf and recvbuf are of type (char*) */
7
      for(k=0; k<outdegree; ++k)</pre>
8
        MPI_Isend(sendbuf, sendcount, sendtype,dsts[k],...);
9
10
      for(l=0; l<indegree; ++1)</pre>
11
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
12
                    srcs[1],...);
13
14
      MPI_Waitall(...);
15
16
          Figure 7.1 shows the neighborhood gather communication of one process with outgoing
17
      neighbors d_0 \ldots d_3 and incoming neighbors s_0 \ldots s_5. The process will send its sendbuf to
18
      all four destinations (outgoing neighbors) and it will receive the contribution from all six
19
      sources (incoming neighbors) into separate locations of its receive buffer.
20
21
22
                                                              d_2, s_4
23
                                            s_0
24
25
                               d_1
                                                          S_1
26
27
28
                                                                  S_3
                                           s_2
29
                                                      d_{3}, s_{5}
30
31
                       sendbuf
32
33
34
                                           s_1
                                                   s_2
                                                           s_3
                                                                   s_4
                                                                           s_5
                                    s_0
35
                       recvbuf
36
37
                      Figure 7.1: Neighborhood gather communication example.
38
39
          All arguments are significant on all processes and the argument comm must have iden-
40
      tical values on all processes.
41
          The type signature associated with sendcount, sendtype, at a process must be equal to
42
      the type signature associated with recvcount, recvtype at all other processes. This implies
43
      that the amount of data sent must be equal to the amount of data received, pairwise between
44
      every pair of communicating processes. Distinct type maps between sender and receiver are
45
      still allowed.
46
                         For optimization reasons, the same type signature is required indepen-
47
            Rationale.
```

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

Example 7.9 On a Cartesian virtual grid, the buffer usage in a given direction d with dims[d] == 3 and 1, respectively during creation of the communicator is described in the following figure.



The figure may apply to any (or multiple) directions in the Cartesian topology. The grey buffers are required in all cases but are only accessed if during creation of the communicator, periods[d] was defined as 1 (in C) or .TRUE. (in Fortran).

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

MPI_	_NEIGHBOR_ALLGATHERV(recvtype, comm)	endbuf, sendcount, sendtype, recvbuf, recvcounts, displs,	16 17
IN	sendbuf	starting address of send buffer (choice)	18 19
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	20 21
IN	sendtype	data type of send buffer elements (handle)	22
OU	JT recvbuf	starting address of receive buffer (choice)	23 24
IN	recvcounts	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor	25 26 27
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	28 29 30
IN	recvtype	data type of receive buffer elements (handle)	31
IN	comm	communicator with topology structure (handle)	33
int	MPI_Neighbor_allgatherv(MPI_Datatype se const int displ	const void *sendbuf, int sendcount, ndtype, void *recvbuf, const int recvcounts[], s[], MPI_Datatype recvtype, MPI_Comm comm)	34 35 36 37
MPI_	<pre>Neighbor_allgatherv(send</pre>	<pre>buf, sendcount, sendtype, recvbuf, recvcounts, we, comm, ierror) INTENT(IN) :: sendbuf endcount, recvcounts(*), displs(*) NT(IN) :: sendtype, recvtype : recvbuf N) :: comm T(OUT) :: ierror</pre>	38 39 40 41 42 43 44 45 46
MPI_	NEIGHBOR_ALLGATHERV(SEND DISPLS, RECVTYF	BUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, E, COMM, IERROR)	47 48

Unofficial Draft for Comment Only

 $1 \\ 2$

 $\overline{7}$

```
<type> SENDBUF(*), RECVBUF(*)
1
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
\mathbf{2}
3
                     IERROR
4
         This function supports Cartesian communicators, graph communicators, and distributed
5
     graph communicators as described in Section 7.6. If comm is a distributed graph commu-
6
     nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
7
     and receives from each of its incoming neighbors:
8
9
     MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
10
     int *srcs=(int*)malloc(indegree*sizeof(int));
11
     int *dsts=(int*)malloc(outdegree*sizeof(int));
12
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
13
                                 outdegree, dsts, MPI_UNWEIGHTED);
14
     int k,l;
15
16
     /* assume sendbuf and recvbuf are of type (char*) */
17
     for(k=0; k<outdegree; ++k)</pre>
18
       MPI_Isend(sendbuf, sendcount, sendtype, dsts[k],...);
19
20
     for(l=0; l<indegree; ++l)</pre>
21
       MPI_Irecv(recvbuf+displs[1]*extent(recvtype), recvcounts[1], recvtype,
22
                  srcs[1],...);
23
24
     MPI_Waitall(...);
25
```

26The type signature associated with sendcount, sendtype, at process j must be equal 27to the type signature associated with recvcounts[l], recvtype at any other process with 28srcs[I] = = j. This implies that the amount of data sent must be equal to the amount of 29 data received, pairwise between every pair of communicating processes. Distinct type maps 30 between sender and receiver are still allowed. The data received from the l-th neighbor is 31 placed into recvbuf beginning at offset displs[I] elements (in terms of the recvtype). 32

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)38 exists in the topology graph or Cartesian topology. Similarly, each process i sends data 39 items to all processes j where an edge (i, j) exists. This call is more general than 40

MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor. 41 The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in 42 the receive buffer is received from the *l*-th neighbor. 43

44

33

34

35 36

- 45
- 46
- 47
- 48

MPI_NEIG	HBOR_ALLTOALL(sendbuf, se comm)	endcount, sendtype, recvbuf, recvcount, recvtype,	1 2		
IN	sendbuf	starting address of send buffer (choice)	3		
IN	sendcount	number of elements sent to each neighbor	4 5		
		(non-negative integer)	6		
IN	sendtype	data type of send buffer elements (handle)	7		
OUT	recvbuf	starting address of receive buffer (choice)	8		
IN	recvcount	number of elements received from each neighbor (non-negative integer)	9 10 11		
IN	recvtype	data type of receive buffer elements (handle)	12		
IN	comm	communicator with topology structure (handle)	13 14		
int MPT N	Jeighbor alltoall(const vo	oid *sendbuf int sendcount	15		
1110 111 1_1	MPI_Datatype sendtyp	e, void *recvbuf, int recvcount,	16		
	MPI_Datatype recvtyp	e, MPI_Comm comm)	17		
MPI_Neigh	nbor_alltoall(sendbuf, sen	ndcount, sendtype, recybuf, recycount,	19		
- 0	recvtype, comm, ierr	or)	20		
TYPE	(*), DIMENSION(), INTEN	Γ(IN) :: sendbuf	21		
INTE	GER, INTENT(IN) :: sendcou	int, recvcount	22		
TYPE	(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	23		
TVDE	TYPE(*), DIMENSION() :: recvbuf 2 TYPE(MDI Comm) INTENT(IN) :: comm 2				
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24					
MFI_NEIGHDUR_ALLIUALL(SENDBUF, SENDCUUNI, SENDIYPE, RECVBUF, RECVCUUNI, RECVTVPE COMM TERROR)					
RECVIIPE, CUMM, IERRUR)			29		
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR			30		
			31		
I nis i graph.com	unction supports Cartesian con municators as described in Se	nmunicators, graph communicators, and distributed	32		
nicator th	e outcome is as if each proces	s executed sends to each of its outgoing neighbors	34		
and receiv	es from each of its incoming n	eighbors:	35		
	0	0	36		
MPI_Dist_	_graph_neighbors_count(con	nm, &indegree, &outdegree, &weighted);	37		
int *srcs	s=(int*)malloc(indegree*s:	<pre>izeof(int));</pre>	38		
int *dsts	s=(int*)malloc(outdegree*:	sizeof(int));	39		
MPI_DISt_	_graph_neighbors(comm, ind	dere MPI UNWEIGHTED).	40		
int k.l:	oubucgio	s, abos, in i_onwlitenilb),	42		
,			43		
/* assume	e sendbuf and recvbuf are	of type (char*) */	44		
<pre>for(k=0;</pre>	k <outdegree; ++k)<="" td=""><td></td><td>45</td></outdegree;>		45		
MPI_Ise	end(sendbuf+k*sendcount*e	xtent(sendtype), sendcount, sendtype,	46		
	dsts[k],);		47		
			48		

```
1 for(1=0; l<indegree; ++1)
2 MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
3 srcs[1],...);</pre>
```

```
MPI_Waitall(...);
```

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

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

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

Example 7.10 For a halo communication on a Cartesian grid, the buffer usage in a given direction d with dims[d]==3 and 1, respectively during creation of the communicator is described in the following figure.



The figure may apply to any (or multiple) directions in the Cartesian topology. The grey buffers are required in all cases but are only accessed if during creation of the communicator, periods[d] was defined as 1 (in C) or .TRUE. (in Fortran).

If each array element of sendbuf and recvbuf are described by sendcount, sendtype and recvbuf, recvtype, then after MPI_NEIGHBOR_ALLTOALL on a Cartesian communicator returned, the content of the recvbuf is as if the following code is executed:

```
38
     MPI_Cartdim_get(comm, &ndims);
39
     for( /*direction*/ d=0; d < ndims; d++) {</pre>
40
      MPI_Cart_shift(comm, /*direction*/ d, /*disp*/ 1, &rank_source, &rank_dest);
41
      MPI_Sendrecv(sendbuf[d*2+0],sendcount,sendtype,rank_source,/*sendtag*/d*2,
42
                    recvbuf[d*2+1],recvcount,recvtype,rank_dest, /*recvtag*/ d*2,
43
                    comm,&status);/*communication in direction of displacment -1*/
44
      MPI_Sendrecv(sendbuf[d*2+1],sendcount,sendtype,rank_dest, /*sendtag*/ d*2+1,
45
                    recvbuf[d*2+0],recvcount,recvtype,rank_source,/*recvtag*/d*2+1,
46
                    comm,&status);/*communication in direction of displacment +1*/
47
     }
48
```

Advice to implementors. For a Cartesian topology, if the virtual grid in a direction d is periodic and dims[d] is equal to 0 or 1, then rank_source and rank_dest are identical, but still all ndims send and ndims receive operations use different buffers. If in this case, the two send and receive operations per direction or of all directions are internally parallelized, then the several send and receive operations for the same sender-receiver process pair must be initiated in the same sequence on sender and receiver side or they shall be distinguished by different tags. The code above shows a valid sequence of operations and tags. (*End of advice to implementors.*)

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

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

)	15
IN	sendbuf	starting address of send buffer (choice)	16
IN	sendcounts	non-negative integer array (of length outdegree)	17
		specifying the number of elements to send to each	18
		neighbor	19
IN	sdispls	integer array (of length outdegree). Entry j specifies	20
		the displacement (relative to $sendbuf)$ from which to	21
		send the outgoing data to neighbor j	22
IN	sendtype	data type of send buffer elements (handle)	23 24
Ol	JT recvbuf	starting address of receive buffer (choice)	25
IN	recvcounts	non-negative integer array (of length indegree)	26
		specifying the number of elements that are received	27
		from each neighbor	28
IN	rdispls	integer array (of length indegree). Entry i specifies	29
	·	the displacement (relative to recvbuf) at which to	30
		place the incoming data from neighbor i	32
IN	recvtype	data type of receive buffer elements (handle)	33
IN	comm	communicator with topology structure (handle)	34
			35
int	MPI_Neighbor_alltoallv(co	onst void *sendbuf, const int sendcounts[],	36
	const int sdisp	<pre>ls[], MPI_Datatype sendtype, void *recvbuf,</pre>	37
	const int recvo	<pre>ounts[], const int rdispls[],</pre>	38
	MPI_Datatype re	cvtype, MPI_Comm comm)	39
мрт	Neighbor alltoally(sendbu	if sendcounts sdispls sendtype recybuf.	40
	recvcounts. rdi	spls. recvtvpe. comm. ierror)	42
	TYPE(*), DIMENSION(),]	INTENT(IN) :: sendbuf	43
	INTEGER, INTENT(IN) :: se	endcounts(*), sdispls(*), recvcounts(*),	44
	rdispls(*)	-	45
	TYPE(MPI_Datatype), INTEN	<pre>NT(IN) :: sendtype, recvtype</pre>	46
	<pre>TYPE(*), DIMENSION() ::</pre>	: recvbuf	47
	TYPE(MPI_Comm), INTENT(IN	V) :: comm	48

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$

8 9

10

11 12 13

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
1
\mathbf{2}
     MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
3
                     RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
4
          <type> SENDBUF(*), RECVBUF(*)
5
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
6
                      RECVTYPE, COMM, IERROR
7
          This function supports Cartesian communicators, graph communicators, and distributed
8
     graph communicators as described in Section 7.6. If comm is a distributed graph commu-
9
     nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
10
11
     and receives from each of its incoming neighbors:
12
     MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
13
     int *srcs=(int*)malloc(indegree*sizeof(int));
14
     int *dsts=(int*)malloc(outdegree*sizeof(int));
15
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
16
                                  outdegree, dsts, MPI_UNWEIGHTED);
17
     int k,l;
18
19
     /* assume sendbuf and recvbuf are of type (char*) */
20
     for(k=0; k<outdegree; ++k)</pre>
21
        MPI_Isend(sendbuf+sdispls[k]*extent(sendtype), sendcounts[k], sendtype,
22
                   dsts[k],...);
23
24
     for(l=0; l<indegree; ++1)</pre>
25
       MPI_Irecv(recvbuf+rdispls[1]*extent(recvtype), recvcounts[1], recvtype,
26
                   srcs[1],...);
27
28
     MPI_Waitall(...);
29
30
          The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process
31
     i must be equal to the type signature associated with recvcounts[I], recvtype with srcs[I]==i
32
     at process j. This implies that the amount of data sent must be equal to the amount of
33
     data received, pairwise between every pair of communicating processes. Distinct type maps
34
     between sender and receiver are still allowed. The data in the sendbuf beginning at offset
35
     sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data
36
     received from the I-th incoming neighbor is placed into recvbuf beginning at offset rdispls[I]
37
     elements (in terms of the recvtype).
38
          The "in place" option is not meaningful for this operation.
39
          All arguments are significant on all processes and the argument comm must have iden-
40
     tical values on all processes.
41
          MPI_NEIGHBOR_ALLTOALLW allows one to send and receive with different datatypes
42
     to and from each neighbor.
43
44
45
46
47
48
```

MPI_NE	IGHBOR_ALLTOALLW(s rdispls, recytypes	endbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, . comm)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length outdegree)	4
IIN	Senacounts	specifying the number of elements to send to each	5
		neighbor	6
IN	sdispls	integer array (of length outdegree) Entry i specifies	8
	5015015	the displacement in bytes (relative to sendbuf) from	9
		which to take the outgoing data destined for	10
		neighbor j (array of integers)	11
IN	sendtypes	array of datatypes (of length outdegree). Entry j	12
		specifies the type of data to send to neighbor j (array	13
		of handles)	14
OUT	recvbuf	starting address of receive buffer (choice)	15 16
IN	recvcounts	non-negative integer array (of length indegree)	17
		specifying the number of elements that are received	18
		from each neighbor	19
IN	rdispls	integer array (of length indegree). Entry i specifies	20
		the displacement in bytes (relative to $recvbuf)$ at	21
		which to place the incoming data from neighbor i	22
		(array of integers)	24
IN	recvtypes	array of datatypes (of length indegree). Entry i	25
		specifies the type of data received from neighbor $($	26
		(array of handles)	27
IN	comm	communicator with topology structure (handle)	28 29
int MPI	Neighbor alltoallw(const void *sendbuf, const int sendcounts□.	30
	const MPI_Aint	sdispls[], const MPI_Datatype sendtypes[],	31
	void *recvbuf,	<pre>const int recvcounts[],</pre>	32
	const MPI_Aint	rdispls[], const MPI_Datatype recvtypes[],	33
	MPI_Comm comm)		34
MPI_Nei	.ghbor_alltoallw(send	buf, sendcounts, sdispls, sendtypes, recvbuf,	35
	recvcounts, rd	lispls, recvtypes, comm, ierror)	37
TYF	<pre>PE(*), DIMENSION(),</pre>	INTENT(IN) :: sendbuf	38
INT	EGER, INTENT(IN) ::	<pre>sendcounts(*), recvcounts(*)</pre>	39
INT	EGER(KIND=MPI_ADDRES	S_KIND), INTENT(IN) :: sdispls(*), rdispls(*)	40
TYF	<pre>'E(MPI_Datatype), INT DE(*) DIMENSION()</pre>	<pre>LNI(IN) :: senatypes(*), recvtypes(*) recvbuf</pre>	41
TYF	PE(MPI Comm). INTENT(IN) :: comm	42
INT	EGER, OPTIONAL, INTE	NT(OUT) :: ierror	43 44
мрт мет	רעם אונייטאנייזע מבאיש. י		45
11F T _ 11G T	RECVCOINTS RE	DOF, SEMPCOUNTS, SPISELS, SEMPTIFES, RECVBUF,	46
<tv< td=""><td>pe> SENDBUF(*). RECV</td><td>BUF(*)</td><td>47</td></tv<>	pe> SENDBUF(*). RECV	BUF(*)	47
- 5	· · · · · · · · · · · · · · · · · · ·		19

MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,

INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 1 IERROR 2 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 3 4 This function supports Cartesian communicators, graph communicators, and distributed 5graph communicators as described in Section 7.6. If comm is a distributed graph commu-6 nicator, the outcome is as if each process executed sends to each of its outgoing neighbors 7 and receives from each of its incoming neighbors: 8 9 MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted); 10 int *srcs=(int*)malloc(indegree*sizeof(int)); 11int *dsts=(int*)malloc(outdegree*sizeof(int)); 12MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED, 13 outdegree, dsts, MPI_UNWEIGHTED); 14 int k,l; 15 16 /* assume sendbuf and recvbuf are of type (char*) */ 17for(k=0; k<outdegree; ++k)</pre> 18 MPI_Isend(sendbuf+sdispls[k], sendcounts[k], sendtypes[k], dsts[k],...); 19 20for(l=0; l<indegree; ++1)</pre> 21MPI_Irecv(recvbuf+rdispls[1], recvcounts[1], recvtypes[1], srcs[1],...); 2223MPI_Waitall(...); 2425The type signature associated with sendcounts[k], sendtypes[k] with dsts[k] == j at pro-26cess i must be equal to the type signature associated with recvcounts[1], recvtypes[1] with 27

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

30 31

32

33 34

35 36

37

38

28

29

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

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

7.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 6.12.

7.7. NONBLOCKING NEIGHBORHOOD COMMUNICATION

7.7.1 Nonblocking Neighborhood Gather

			2
			3
MPI_INI	COMM request)	sendbut, sendcount, sendtype, recvbut, recvcount, recvtype,	5
IN	condbuf	starting address of send buffer (shoise)	6
	Senabal	starting address of send burler (choice)	7
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	8 9
IN	sendtype	data type of send buffer elements (handle)	1
Ουτ	recvbuf	starting address of receive buffer (choice)	1
IN	recvcount	number of elements received from each neighbor (non-negative integer)	1
IN	recvtype	data type of receive buffer elements (handle)	1
IN	comm	communicator with topology structure (handle)	1
OUT	request	communication request (handle)	1
int MPI	_Ineighbor_allgather	c(const void *sendbuf, int sendcount,	2
int MPI		const void *senabul, int senacount,	2
	MPI Datatype :	recutupe, MPI Comm comm MPI Request *request)	2
	HFI_Datatype .	recycype, Mri_comm comm, Mri_Kequest *request)	2
MPI_Ine	ighbor_allgather(ser	ndbuf, sendcount, sendtype, recvbuf, recvcount,	2
	recvtype, com	m, request, ierror)	2
TYP	<pre>PE(*), DIMENSION(),</pre>	, INTENT(IN), ASYNCHRUNUUS :: sendbuf	2
TN.I	EGER, INTENT(IN) ::	sendcount, recvcount	2
TYP	PE(MPI_Datatype), INT	TENT(IN) :: sendtype, recvtype	2
TYP	<pre>PE(*), DIMENSION(),</pre>	, ASYNCHRUNUUS :: recvbuf	2
TYP	PE(MPI_Comm), INTENT((IN) :: comm	3
TYP	E(MPI_Request), INTE	SNT(UUT) :: request	3
T N.I	EGER, OPTIONAL, INTE	ENT(UUT) :: ierror	3
MPI_INE	IGHBOR_ALLGATHER(SEN	IDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	3
_	RECVTYPE, COM	M, REQUEST, IERROR)	3
<ty< td=""><td>pe> SENDBUF(*), RECV</td><td>/BUF(*)</td><td>3</td></ty<>	pe> SENDBUF(*), RECV	/BUF(*)	3
INT	EGER SENDCOUNT, SEND	TYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	3
ΤЪ;	a coll storts o populació	ng variant of MDL NEICHROP ALL CATHER	3
1 111	s can starts a nondiocki	ILE VALIAILE OF MITT_INCIDENT ALLOATHEN.	3
			3
			4
			4

1 2	MPI_INEI	GHBOR_ALLGATHERV(sendburner) recvtype, comm, request)	ıf, 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) containing 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 29 30 31 32 33 34 35	<pre>MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],</pre>			
36 37 38 39 40 41 42 43 44 45 46 47 48	<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,</pre>			

7.7. NONBLOCKING NEIGHBORHOOD COMMUNICATION

7.7.2 Nonblocking Neighborhood Alltoall

			2
			3
MPI_INE	EIGHBOR_ALLTOALL(se	endbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,	4 5
	comm, request)		6
IN	sendbut	starting address of send buffer (choice)	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
Ουτ	request	communication request (handle)	17
			19
int MPI	_Ineighbor_alltoall(const void *sendbuf, int sendcount,	20
	MPI_Datatype a	sendtype, void *recvbuf, int recvcount,	21
	MPI_Datatype 1	recvtype, MPI_Comm comm, MPI_Request *request)	22
MPI_Ine	ighbor_alltoall(send	lbuf, sendcount, sendtype, recvbuf, recvcount,	23
	recvtype, comm	n, request, ierror)	24
TYP	E(*), DIMENSION(),	INTENT(IN), ASYNCHRONOUS :: sendbuf	-0 26
INT	EGER, INTENT(IN) ::	sendcount, recvcount	27
TYP	E(MPI_Datatype), INT	ENT(IN) :: sendtype, recvtype	28
TYP	E(*), DIMENSION(),	ASYNCHRONOUS :: recvbuf	29
TYP	E(MPI_Comm), INTENT(IN) :: comm	30
TYP	E(MPI_Request), INTE	ENT(OUT) :: request	31
INT	EGER, OPTIONAL, INTE	ENT(OUT) :: ierror	32
MDT TNF	TCHBOR ALLTOALL (SENC	NEIF SENDCOINT SENDTVDE BECVRIJE BECVCOINT	33
III I_INL		M REGUEST IERROR)	34
<+v	SENDBUE(*) BECV	RIF(*)	35
INT	EGER SENDCOUNT, SEND	DITYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	36
			37
Thi	s call starts a nonblockii	ng variant of MPI_NEIGHBOK_ALLIOALL.	38
			39
			40
			41
			42

```
MPI_INEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
1
                     rdispls, recvtype, comm, request)
^{2}
3
       IN
                 sendbuf
                                              starting address of send buffer (choice)
4
       IN
                 sendcounts
                                              non-negative integer array (of length outdegree)
5
                                              specifying the number of elements to send to each
6
                                              neighbor
7
       IN
                 sdispls
                                              integer array (of length outdegree). Entry j specifies
8
                                              the displacement (relative to sendbuf) from which
9
                                              send the outgoing data to neighbor j
10
11
       IN
                 sendtype
                                              data type of send buffer elements (handle)
12
       OUT
                 recvbuf
                                              starting address of receive buffer (choice)
13
       IN
                 recycounts
                                              non-negative integer array (of length indegree)
14
                                              specifying the number of elements that are received
15
16
                                              from each neighbor
17
       IN
                 rdispls
                                              integer array (of length indegree). Entry i specifies
18
                                              the displacement (relative to recvbuf) at which to
19
                                              place the incoming data from neighbor i
20
       IN
                                              data type of receive buffer elements (handle)
                 recvtype
21
       IN
                                              communicator with topology structure (handle)
22
                 comm
23
       OUT
                                              communication request (handle)
                 request
24
25
     int MPI_Ineighbor_alltoallv(const void *sendbuf, const int sendcounts[],
26
                     const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
27
                     const int recvcounts[], const int rdispls[],
28
                     MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
29
     MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
30
                     recvcounts, rdispls, recvtype, comm, request, ierror)
31
32
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
          INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
33
34
                      recvcounts(*), rdispls(*)
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
35
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
36
          TYPE(MPI_Comm), INTENT(IN) :: comm
37
          TYPE(MPI_Request), INTENT(OUT) :: request
38
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
41
                     RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
42
          <type> SENDBUF(*), RECVBUF(*)
43
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
44
                      RECVTYPE, COMM, REQUEST, IERROR
45
46
          This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.
47
48
```

MPI_INE	IGHBOR_ALLTOALLW(se rdispls, recvtypes,	endbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, comm, request)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor	4 5 6 7
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)	8 9 10 11
IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)	12 13 14
OUT	recvbuf	starting address of receive buffer (choice)	15 16
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor	17 18 19
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)	20 21 22 23
IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)	24 25 26 27
IN	comm	communicator with topology structure (handle)	28
OUT	request	communication request (handle)	29 30
int MPI.	_Ineighbor_alltoallw(const MPI_Aint void *recvbuf, const MPI_Aint MPI_Comm comm,	<pre>const void *sendbuf, const int sendcounts[], sdispls[], const MPI_Datatype sendtypes[], const int recvcounts[], rdispls[], const MPI_Datatype recvtypes[], MPI_Request *request)</pre>	31 32 33 34 35
MPI Ine:	ighbor alltoallw(send	buf, sendcounts, sdispls, sendtypes, recybuf,	30 37
_	recvcounts, rdi	spls, recvtypes, comm, request, ierror)	38
TYPI	E(*), DIMENSION(),	INTENT(IN), ASYNCHRONOUS :: sendbuf	39
INT	EGER, INTENT(IN), ASY	NCHRONOUS :: sendcounts(*), recvcounts(*)	40
TNU	GER(KIND=MP1_ADDRESS	_KIND), INTENT(IN), ASYNCHRUNUUS :: sdispis(*),	41
ТҮРІ	E(MPI Datatvne). INTE	NT(IN), ASYNCHRONOUS :: sendtypes(*).	42 43
	recvtypes(*)	· · · · · · · · · · · · · · · · · · ·	44
TYPI	E(*), DIMENSION(),	ASYNCHRONOUS :: recvbuf	45
TYPI	E(MPI_Comm), INTENT(I	N) :: comm	46
TYPI	E(MPI_Request), INTEN	T(OUT) :: request	47
INTI	EGER, OPTIONAL, INTEN	T(OUT) :: ierror	48

Unofficial Draft for Comment Only

1	MPI_INEI	GHBOR_ALLTOALLW(S RECVCOUNTS.	ENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
3	<typ< td=""><td><pre>> SENDBUF(*), RE</pre></td><td>CVBUF(*)</td></typ<>	<pre>> SENDBUF(*), RE</pre>	CVBUF(*)
4	INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,		
5	тыт	REQUEST, IE	
6 7	INTE	GER(KIND=MP1_ADDR	ESS_KIND) SDISPLS(*), RDISPLS(*)
8 9	This	call starts a nonbloc	king variant of MPI_NEIGHBOR_ALLTOALLW.
10	7.8 Pe	ersistent Neighbor	rhood Communication on Process Topologies
12 13 14 15	Persisten mance be similar to	t variants of the neig enefits for programs o persistent collective	ghborhood collective operations can offer significant perfor- with repetitive communication patterns. The semantics are operations as described in Section 6.13 .
16 17	7.8.1 P	ersistent Neighborho	od Gather
18 19 20	MPI_NEI	GHBOR_ALLGATHEF recvtype, comr	R_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, n, info, request)
21 22	IN	sendbuf	starting address of send buffer (choice)
23 24	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
25	IN	sendtype	data type of send buffer elements (handle)
26 27	OUT	recvbuf	starting address of receive buffer (choice)
28 29	IN	recvcount	number of elements received from each neighbor (non-negative integer)
30	IN	recvtype	data type of receive buffer elements (handle)
31	IN	comm	communicator with topology structure (handle)
33	IN	info	info argument (handle)
34 35	OUT	request	communication request (handle)
36 37 38 39 40	int MPI_	Neighbor_allgathe MPI_Datatype MPI_Datatype MPI_Request	<pre>r_init(const void *sendbuf, int sendcount, e sendtype, void *recvbuf, int recvcount, e recvtype, MPI_Comm comm, MPI_Info info, *request)</pre>
41	MPI_Neig	ghbor_allgather_in	it(sendbuf, sendcount, sendtype, recvbuf,
42	ጥህጉቦ	recvcount, r	recvtype, comm, info, request, ierror)
43 44	ТУРЕ ТМТЕ	CGER. INTENT(IN) ·), INIENI(IN), ASINCHKUNUUS :: sendbui : sendcount. recycount
45	TYPE	E(MPI_Datatype), I	NTENT(IN) :: sendtype, recvtype
46	TYPE	E(*), DIMENSION(), ASYNCHRONOUS :: recvbuf
47	TYPE	E(MPI_Comm), INTEN	T(IN) :: comm
48	TYPE	E(MPI Info), INTEN	T(IN) :: info

Unofficial Draft for Comment Only

TYPI INTI	E(MPI_Request), INTEN EGER, OPTIONAL, INTEN	T(OUT) :: request T(OUT) :: ierror	1 2
MDT NET	THROP ALLCATHER INTTO	SENDRIF SENDCOINT SENDTVDE RECVRIE	3
	RECVCOUNT. RECV	TYPE, COMM, INFO, REQUEST, TERBOR)	4
<tyr< td=""><td><pre>De> SENDBUF(*), RECVBU</pre></td><td>UF(*)</td><td>5</td></tyr<>	<pre>De> SENDBUF(*), RECVBU</pre>	UF(*)	5
INTI	EGER SENDCOUNT, SENDT	YPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	6
	IERROR		7
Crea	tes a persistent collective	e communication request for the neighborhood allgather	9
operation	l.	· ····································	10
-			11
		NIT (conduct condemns you but you counts	12
	displs. recvtype. co	mm, info, request)	13
IN	sendbuf	starting address of send buffer (choice)	14 15
IN	sendcount	number of elements sent to each neighbor	16
IIN	Schucount	(non-negative integer)	17
IN	sendtype	data type of send buffer elements (handle)	18
	recybuf	starting address of receive buffer (choice)	20
	recybar	starting address of receive build (choice)	21
IIN	recvcounts	containing the number of elements that are received	22
		from each neighbor	23
IN	displs	integer array (of length indegree). Entry i specifies	24
IIN		the displacement (relative to recybuf) at which to	25
		place the incoming data from neighbor i	20 27
IN	recvtype	data type of receive buffer elements (handle)	28
IN	comm	communicator with topology structure (handle)	29
IN	info	info argument (handle)	30
	nino resultati		31
001	request	communication request (nandle)	32
int MPT	Neighbor allgathery	init(const woid *sendbuf int sendcount	34
1110 III 1 <u>.</u>	_Meignboi_aligatheiv_ MPI Datatype se	ndtype, void *recybuf, const int recycounts[].	35
	const int displ	s[], MPI_Datatype recvtype, MPI_Comm comm,	36
	MPI_Info info,	MPI_Request *request)	37
MPT Neig	whor allgathery init	(sendbuf, sendcount, sendtype, recybuf,	38
111 1_11018	recvcounts, dis	pls, recvtype, comm, info, request, ierror)	39 40
TYPI	E(*), DIMENSION(), I	INTENT(IN), ASYNCHRONOUS :: sendbuf	40
INTE	EGER, INTENT(IN) :: se	endcount, displs(*)	42
TYPI	E(MPI_Datatype), INTE	NT(IN) :: sendtype, recvtype	43
TYPI	E(*), DIMENSION(), A	ASYNCHRONOUS :: recvbuf	44
באעד נסעיד	LGER, INIENT(IN), ASYL E(MDT Comm) INTENT(II	NUHKUNUUS :: recvcounts(*)	45
ТҮРІ	E(MPI Info). INTENT(II	N) :: info	46
TYPI	E(MPI_Request), INTEN	T(OUT) :: request	47 48

Unofficial Draft for Comment Only

1	INTE	EGER, OPTIONAL, INT	ENT(OUT) :: ierror		
2 3 4 5 6 7	<pre>MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, INFO, REQUEST, IERROR</type></pre>				
8 9	Crea operation	tes a persistent collect	vive communication request for the neighborhood all gathered		
10 11 12 13	7.8.2 P	ersistent Neighborhoo	d Alltoall		
14 15	MPI_NEI	GHBOR_ALLTOALL_I recvtype, comm	NIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, , info, request)		
16 17	IN	sendbuf	starting address of send buffer (choice)		
18 19	IN	sendcount	number of elements sent to each neighbor (non-negative integer)		
20	IN	sendtype	data type of send buffer elements (handle)		
21 22	OUT	recvbuf	starting address of receive buffer (choice)		
23 24	IN	recvcount	number of elements received from each neighbor (non-negative integer)		
25	IN	recvtype	data type of receive buffer elements (handle)		
26 27	IN	comm	communicator with topology structure (handle)		
28	IN	info	info argument (handle)		
29 30	OUT	request	communication request (handle)		
31 32 33 34 35	int MPI_	Neighbor_alltoall_ MPI_Datatype MPI_Datatype MPI_Request *	<pre>init(const void *sendbuf, int sendcount, sendtype, void *recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Info info, request)</pre>		
36	MPI_Neig	hbor_alltoall_init	(sendbuf, sendcount, sendtype, recvbuf,		
37 38	TYPE	recvcount, re E(*), DIMENSION()	ecvtype, comm, info, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf		
39 40	INTE	EGER, INTENT(IN) :: E(MRI Deteture) IN	sendcount, recvcount		
41	TYPE	E(*), DIMENSION()	ASYNCHRONOUS :: recvbuf		
42	TYPE	E(MPI_Comm), INTENT	(IN) :: comm		
43	TYPE	E(MPI_Info), INTENT	(IN) :: info		
44 45	ТҮРЕ	(MPI_Request), INT	ENT(UUT) :: request		
46		UTILUNAL, INI	ENI(UUI) IEIIUI		
47	MPI_NEIC	HBOR_ALLTOALL_INIT	(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,		
48		RECVCUUNT, RE	SCVIYPE, CUMM, INFU, REQUEST, IERROR)		

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

Creates a persistent collective communication request for the neighborhood all toall operation.

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

IN	sendbuf	starting address of send buffer (choice)	10		
181			11		
IIN	senacounts	specifying the number of elements to send to each	12		
		neighbor	13		
INI	odianla	internet (af langth antilarman) Entry i an aifea	14		
IIN	saispis	the displacement (relative to sendbuf) from which	16		
		send the outgoing data to neighbor i	17		
INI	condtuno	data time of good buffer elements (hendle)	18		
IIN	sendtype	data type of send burier elements (nandle)	19		
OUT	recvbuf	starting address of receive buffer (choice)	20		
IN	recvcounts	non-negative integer array (of length indegree)	21		
		specifying the number of elements that are received	22		
		from each neighbor	23		
IN	rdispls	integer array (of length indegree). Entry i specifies	24		
		the displacement (relative to $recvbuf)$ at which to	25 26		
		place the incoming data from neighbor i	20		
IN	recvtype	data type of receive buffer elements (handle)	28		
IN	comm	communicator with topology structure (handle)	29		
IN	info	info argument (handle)	30		
			31		
001	request	communication request (handle)	32		
			33		
int MPI_	Neighbor_alltoallv_init	(const void *sendbuf,	34		
	Const int sendcour	uts[], const int sdispis[],	35		
	const int rdispla	MPI Datatupe recutupe MPI Comm comm	37		
	MPI Info info. MPI	Bequest *request)	38		
			39		
MPI_Neig	hbor_alltoallv_init(sen	dbuf, sendcounts, sdispls, sendtype,	40		
	recvbuf, recvcount	s, rdispls, recvtype, comm, info, request,	41		
TVDE	lerror)	ENT(IN) ACVNCUDONOUS condbuf	42		
1 I P E T N T E	CFR INTENSION(), INI	ENT(IN), ASINGHRONOUS .: SEHADAL RONOHS ·· sendcounts(*) sdispls(*)	43		
T 14 T T	recycounts(*) rd	<pre>ispls(*)</pre>	44		
TYPE	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype				

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf

TYPE(MPI_Comm), INTENT(IN) :: comm

 $\mathbf{2}$

```
TYPE(MPI_Info), INTENT(IN) :: info
1
           TYPE(MPI_Request), INTENT(OUT) :: request
2
3
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
      MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
5
                      RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,
6
                      IERROR)
7
           <type> SENDBUF(*), RECVBUF(*)
8
           INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
9
                       RECVTYPE, COMM, INFO, REQUEST, IERROR
10
11
           Creates a persistent collective communication request for the neighborhood alloally
12
      operation.
13
14
      MPI_NEIGHBOR_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
15
                      recvcounts, rdispls, recvtypes, comm, info, request)
16
17
        IN
                  sendbuf
                                                 starting address of send buffer (choice)
18
        IN
                  sendcounts
                                                 non-negative integer array (of length outdegree)
19
                                                 specifying the number of elements to send to each
20
                                                 neighbor
21
        IN
                  sdispls
                                                 integer array (of length outdegree). Entry j specifies
22
                                                 the displacement in bytes (relative to sendbuf) from
23
                                                 which to take the outgoing data destined for
24
                                                 neighbor j (array of integers)
25
        IN
26
                  sendtypes
                                                 array of datatypes (of length outdegree). Entry j
27
                                                 specifies the type of data to send to neighbor j (array
28
                                                 of handles)
29
        OUT
                  recvbuf
                                                 starting address of receive buffer (choice)
30
        IN
                   recvcounts
                                                 non-negative integer array (of length indegree)
31
                                                 specifying the number of elements that are received
32
                                                 from each neighbor
33
34
        IN
                   rdispls
                                                 integer array (of length indegree). Entry i specifies
35
                                                 the displacement in bytes (relative to recvbuf) at
36
                                                 which to place the incoming data from neighbor i
37
                                                 (array of integers)
38
        IN
                   recvtypes
                                                 array of datatypes (of length indegree). Entry i
39
                                                 specifies the type of data received from neighbor i
40
                                                 (array of handles)
41
        IN
                  comm
                                                 communicator with topology structure (handle)
42
        IN
                  info
43
                                                 info argument (handle)
44
        OUT
                   request
                                                 communication request (handle)
45
46
      int MPI_Neighbor_alltoallw_init(const void *sendbuf,
47
                      const int sendcounts[], const MPI_Aint sdispls[],
48
```

```
const MPI_Datatype sendtypes[], void *recvbuf,
                                                                                   1
              const int recvcounts[], const MPI_Aint rdispls[],
                                                                                   2
              const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
                                                                                   3
              MPI_Request *request)
                                                                                   4
                                                                                   5
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
                                                                                   6
              recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
                                                                                   7
              ierror)
                                                                                   8
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   9
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                   10
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                   11
              rdispls(*)
                                                                                   12
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                   13
              recvtypes(*)
                                                                                   14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   16
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   17
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   19
MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
                                                                                   20
              RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
                                                                                   21
              IERROR)
                                                                                   22
    <type> SENDBUF(*), RECVBUF(*)
                                                                                   23
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
                                                                                   24
              INFO, REQUEST, IERROR
                                                                                   25
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
                                                                                   26
```

Creates a persistent collective communication request for the neighborhood alltoallw operation.

7.9 An Application Example

Example 7.11 The example in Figures 7.2-7.5 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)).

27

28

29 30

31 32 33

34

35

36

37

38

39

40

41

42

43

```
INTEGER ndims, num_neigh
1
     LOGICAL reorder
2
     PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
3
     INTEGER comm, comm_size, comm_cart, dims(ndims), ierr
4
     INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
5
     LOGICAL periods(ndims)
6
     REAL u(0:101,0:101), f(0:101,0:101)
7
     DATA dims / ndims * 0 /
8
     comm = MPI_COMM_WORLD
9
     CALL MPI_COMM_SIZE(comm, comm_size, ierr)
10
         Set process grid size and periodicity
     !
11
     CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
12
     periods(1) = .TRUE.
13
     periods(2) = .TRUE.
14
         Create a grid structure in WORLD group and inquire about own position
     1
15
     CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
16
                           comm_cart, ierr)
17
     CALL MPI_CART_GET(comm_cart, ndims, dims, periods, own_coords, ierr)
18
     i = own_coords(1)
19
     j = own_coords(2)
20
     ! Look up the ranks for the neighbors. Own process coordinates are (i,j).
21
     ! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
22
     CALL MPI_CART_SHIFT(comm_cart, 0,1, neigh_rank(1), neigh_rank(2), ierr)
23
     CALL MPI_CART_SHIFT(comm_cart, 1,1, neigh_rank(3), neigh_rank(4), ierr)
24
     ! Initialize the grid functions and start the iteration
25
     CALL init(u, f)
26
     DO it=1,100
27
        CALL relax(u, f)
28
            Exchange data with neighbor processes
     !
29
        CALL exchange(u, comm_cart, neigh_rank, num_neigh)
30
     END DO
31
     CALL output(u)
32
33
34
        Figure 7.2: Set-up of process structure for two-dimensional parallel Poisson solver.
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
                                                                                    1
REAL u(0:101,0:101)
                                                                                    2
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
                                                                                    3
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
                                                                                    4
INTEGER ierr
                                                                                    5
sndbuf(1:100,1) = u(1,1:100)
                                                                                    6
sndbuf(1:100,2) = u(100,1:100)
                                                                                    7
sndbuf(1:100,3) = u(1:100, 1)
                                                                                    8
sndbuf(1:100,4) = u(1:100,100)
CALL MPI_NEIGHBOR_ALLTOALL(sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
                                                                                   10
                            comm_cart, ierr)
                                                                                   11
! instead of
                                                                                   12
! CALL MPI_IRECV(rcvbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(1), ierr)
                                                                                   13
! CALL MPI_ISEND(sndbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(2), ierr)
                                                                                   14
    Always pairing a receive from rank_source with a send to rank_dest
1
                                                                                   15
!
    of the same direction in MPI_CART_SHIFT!
                                                                                   16
! CALL MPI_IRECV(rcvbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(3), ierr)
                                                                                   17
! CALL MPI_ISEND(sndbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(4), ierr)
                                                                                   18
! CALL MPI_IRECV(rcvbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(5), ierr)
                                                                                   19
! CALL MPI_ISEND(sndbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(6), ierr)
                                                                                   20
! CALL MPI_IRECV(rcvbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(7), ierr)
                                                                                   21
! CALL MPI_ISEND(sndbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(8), ierr)
                                                                                   22
    Of course, one can first start all four IRECV and then all four ISEND,
!
                                                                                   23
!
    Or vice versa, but both in the sequence shown above. Otherwise, the
                                                                                   24
    matching would be wrong for 2 or only 1 processes in a direction.
!
                                                                                   25
! CALL MPI_WAITALL(2*num_neigh, rq, statuses, ierr)
                                                                                   26
u(0,1:100) = rcvbuf(1:100,1)
                                                                                   27
u(101,1:100) = rcvbuf(1:100,2)
                                                                                   28
u(1:100, 0) = rcvbuf(1:100,3)
                                                                                   29
u(1:100,101) = rcvbuf(1:100,4)
                                                                                   30
END
                                                                                   31
                                                                                   32
                                                                                   33
Figure 7.3: Communication routine with local data copying and sparse neighborhood all-
                                                                                   34
to-all.
                                                                                   35
                                                                                   36
                                                                                   37
                                                                                   38
                                                                                   39
                                                                                   40
```

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
1
     IMPLICIT NONE
2
     USE MPI
3
     REAL u(0:101,0:101)
4
     INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
5
     INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
     INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
6
     INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
7
     INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
8
     INTEGER type_vec, ierr
9
     ! The following initialization need to be done only once
10
     ! before the first call of exchange.
11
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
12
     CALL MPI_TYPE_VECTOR(100, 1, 102, MPI_REAL, type_vec, ierr)
13
     CALL MPI_TYPE_COMMIT(type_vec, ierr)
     sndtypes(1:2) = type_vec
14
     sndcounts(1:2) = 1
15
     sndtypes(3:4) = MPI_REAL
16
     sndcounts(3:4) = 100
17
     rcvtypes = sndtypes
18
     rcvcounts = sndcounts
19
     sdispls(1) = (1 + 1*102) * sizeofreal ! first element of u( 1 , 1:100)
                                                                             , 1:100)
20
     sdispls(2) = (100 + 1*102) * sizeofreal ! first element of u(100
21
     sdispls(3) = (1 + 1*102) * sizeofreal ! first element of u( 1:100, 1
                                                                                     )
     sdispls(4) = (1 + 100*102) * sizeofreal ! first element of u( 1:100,100
                                                                                     )
22
                                                                             , 1:100)
     rdispls(1) = (0 + 1*102) * size of real ! first element of u(0)
23
                                                                             , 1:100)
     rdispls(2) = (101 + 1*102) * size of real ! first element of u(101)
24
     rdispls(3) = (1 + 0*102) * sizeofreal ! first element of u( 1:100, 0
                                                                                     )
25
     rdispls(4) = (1 + 101*102) * sizeofreal ! first element of u( 1:100,101
                                                                                     )
26
     ! the following communication has to be done in each call of exchange
27
     CALL MPI_NEIGHBOR_ALLTOALLW(u, sndcounts, sdispls, sndtypes, &
                                  u, rcvcounts, rdispls, rcvtypes, &
28
                                  comm_cart, ierr)
29
     ! The following finalizing need to be done only once
30
     ! after the last call of exchange.
31
     CALL MPI_TYPE_FREE(type_vec, ierr)
32
     END
33
34
     Figure 7.4: Communication routine with sparse neighborhood all-to-all-w and without local
35
     data copying.
36
37
38
39
40
41
42
43
44
45
46
47
48
```

```
INTEGER ndims, num_neigh
                                                                                    1
LOGICAL reorder
                                                                                    2
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
                                                                                    3
INTEGER comm, comm_size, comm_cart, dims(ndims), it, ierr
                                                                                    4
LOGICAL periods(ndims)
                                                                                    5
REAL u(0:101,0:101), f(0:101,0:101)
                                                                                    6
DATA dims / ndims * 0 /
                                                                                    7
INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
                                                                                    8
INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
                                                                                    9
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
                                                                                    10
INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
                                                                                    11
INTEGER type_vec, request, status
                                                                                    12
comm = MPI_COMM_WORLD
                                                                                    13
CALL MPI_COMM_SIZE(comm, comm_size, ierr)
                                                                                    14
    Set process grid size and periodicity
1
                                                                                    15
CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
                                                                                    16
periods(1) = .TRUE.
                                                                                    17
periods(2) = .TRUE.
                                                                                    18
    Create a grid structure in WORLD group
1
                                                                                    19
CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
                                                                                    20
                      comm_cart, ierr)
                                                                                   21
! Create datatypes for the neighborhood communication
                                                                                   22
                                                                                   23
! Insert code from example in Figure 7.4 to create and initialize
                                                                                   24
! sndcounts, sdispls, sndtypes, rcvcounts, rdispls, and rcvtypes
                                                                                    25
                                                                                   26
! Initialize the neighborhood all-to-all-w operation
                                                                                    27
CALL MPI_NEIGHBOR_ALLTOALLW_INIT(u, sndcounts, sdispls, sndtypes, &
                                                                                    28
                                   u, rcvcounts, rdispls, rcvtypes, &
                                                                                   29
                                   comm_cart, info, request, ierr)
                                                                                    30
! Initialize the grid functions and start the iteration
                                                                                   31
CALL init(u, f)
                                                                                    32
DO it=1,100
                                                                                    33
       Start data exchange with neighbor processes
1
                                                                                    34
   CALL MPI_START(request, ierr)
                                                                                   35
!
       Compute inner cells
                                                                                   36
   CALL relax_inner (u, f)
                                                                                   37
ļ
       Check on completion of neighbor exchange
                                                                                    38
   CALL MPI_WAIT(request, status, ierr)
                                                                                    39
1
       Compute edge cells
                                                                                    40
   CALL relax_edges(u, f)
                                                                                    41
END DO
                                                                                    42
CALL output(u)
                                                                                    43
CALL MPI_REQUEST_FREE(request, ierr)
                                                                                    44
CALL MPI_TYPE_FREE(type_vec, ierr)
                                                                                    45
                                                                                    46
```

Figure 7.5: Two-dimensional parallel Poisson solver with persistent sparse neighborhood ⁴⁷ all-to-all-w and without local data copying. ⁴⁸

Bibliography

- S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormholerouted 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] 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
- [4] T. Hoefler and J. L. Träff. Sparse collective operations for MPI. In Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop, May 2009. 7.6
 - [5] O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. *Parallel Computing*, 9:213–225, 1989. 7.1

Index

CONST:MPI_BOTTOM, 10, 12 MPI_CART_COORDS, 3, 18 MPI_CART_COORDS(comm, rank, maxdims, CONST:MPI_CART, 14 CONST:MPI_COMM_NULL, 4, 6 coords), 17 CONST:MPI_COMM_WORLD, 5 MPI_CART_CREATE, 2-6, 16, 24, 26 CONST:MPI_DIST_GRAPH, 14 MPI_CART_CREATE(comm_old, ndims, dims, CONST:MPI_GRAPH, 14 periods, reorder, comm_cart), 4 CONST:MPI_INFO_NULL, 12 MPI_CART_CREATE(comm, ndims, dims, CONST:MPI_PROC_NULL, 22, 26 periods, reorder, comm_cart), 25 CONST:MPI_SUCCESS, 5 MPI_CART_GET, 3, 16 MPI_CART_GET(comm, maxdims, dims, periods, CONST:MPI_UNDEFINED, 14, 24, 25 CONST:MPI_UNWEIGHTED, 9, 10, 12, 13, coords), 16 MPI_CART_MAP, 3, 25 20, 21 MPI_CART_MAP(comm, ndims, dims, periods, CONST:MPI_WEIGHTS_EMPTY, 9, 10, 12 newrank), $\underline{24}$ EXAMPLES: Cartesian virtual topologies, 47 MPI_CART_MAP(comm, ndims, dims, pe-EXAMPLES: MPI_CART_COORDS, 23 riods, newrank), 25 EXAMPLES:MPI_CART_GET, 47 MPI_CART_RANK, 3, 17 EXAMPLES:MPI_CART_RANK, 23 MPI_CART_RANK(comm, coords, rank), 17 EXAMPLES:MPI_CART_SHIFT, 23, 47 MPI_CART_SHIFT, 3, 22, 23, 26 EXAMPLES:MPI_CART_SUB, 24 MPI_CART_SHIFT(comm, direction, disp, EXAMPLES: MPI_DIMS_CREATE, 5, 47 rank_source, rank_dest), 22 EXAMPLES:MPI_DIST_GRAPH_CREATE, MPI_CART_SUB, 3, 24 12MPI_CART_SUB(comm, remain_dims, newcomm), EXAMPLES:MPI_Dist_graph_create, 13 EXAMPLES:MPI_DIST_GRAPH_CREATE_MPI_ACARF_SUB(comm, remain_dims, comm_new), 1224, 25 EXAMPLES:MPI_GRAPH_CREATE, 7, 19 MPI_CARTDIM_GET, 3, 16 $EXAMPLES: MPI_GRAPH_NEIGHBORS, 19MPI_CARTDIM_GET(comm, ndims), 16MPI_CARTDIM_GET(comm, ndims$ $EXAMPLES: MPI_GRAPH_NEIGHBORS_CQHPYTCOMM_CREATE, 3$ 19MPI_COMM_SPLIT, 3, 4, 6, 24 $EXAMPLES: MPI_NEIGHBOR_ALLGATHERMPI_COMM_SPLIT(comm, color, key, comm_cart), \\$ 2925 $EXAMPLES: MPI_NEIGHBOR_ALLTOALL, MPI_COMM_SPLIT(comm, color, key, comm_graph), \\$ 32 26 $EXAMPLES: MPI_SENDRECV_REPLACE, \ MPI_COMM_SPLIT(comm, color, key, comm_new), \\$ 2325EXAMPLES:Neighborhood collective commu- MPI_DIMS_CREATE, 3-5nication, 47 MPI_DIMS_CREATE(6, 2, dims), 5 EXAMPLES: Topologies, 47 MPI_DIMS_CREATE(6, 3, dims), 5 EXAMPLES: Virtual topologies, 47 MPI_DIMS_CREATE(7, 2, dims), 5

1	$MPI_DIMS_CREATE(7, 3, dims), 5$	<u>15</u>
2	MPI_DIMS_CREATE(nnodes, ndims, dims),	MPI_INEIGHBOR_ALLGATHER, 3
3	<u>5</u>	MPI_INEIGHBOR_ALLGATHER(sendbuf, sendcount,
4	MPI_DIST_GRAPH_CREATE, 2, 3, 8, 11,	sendtype, recvbuf, recvcount, recvtype,
5	13, 21, 22, 26	comm, request), 37
6	MPI_DIST_GRAPH_CREATE(comm_old, n,	MPI_INEIGHBOR_ALLGATHERV, 3
7	sources, degrees, destinations, weights	MPI_INEIGHBOR_ALLGATHERV(sendbuf,
8	info, reorder, comm_dist_graph), $\underline{10}$	sendcount, sendtype, recvbuf, recvcounts,
9	MPI_DIST_GRAPH_CREATE_ADJACENT	, displs, recvtype, comm, request), <u>38</u>
10	2, 3, 8, 9, 13, 21, 26	MPI_INEIGHBOR_ALLTOALL, 3
11	MPI_DIST_GRAPH_CREATE_ADJACENT	(MARIGHBOR_ALLTOALL(sendbuf, sendcount,
12	indegree, sources, sourceweights, outde	egree, sendtype, recvbuf, recvcount, recvtype,
13	destinations, destweights, info, reorder	$c, ext{ comm, request}, \underline{39}$
14	$\operatorname{comm_dist_graph}), \underline{8}$	MPI_INEIGHBOR_ALLTOALLV, 3
15	MPI_DIST_GRAPH_NEIGHBORS, 3, 20, 21,	$\label{eq:mpi_integration} MPI_INEIGHBOR_ALLTOALLV (sendbuf, sendcounts, \\$
16	26	sdispls, sendtype, recvbuf, recvcounts,
17	MPI_DIST_GRAPH_NEIGHBORS(comm, m	axindegr ed ispls, recvtype, comm, request), <u>40</u>
18	sources, sourceweights, maxoutdegree,	MPI_INEIGHBOR_ALLTOALLW, 4
19	destinations, destweights), $\underline{21}$	${\it MPI_INEIGHBOR_ALLTOALLW} ({\it sendbuf}, {\it sendcounts},$
20	MPI_DIST_GRAPH_NEIGHBORS_COUNT	, sdispls, sendtypes, recvbuf, recvcounts,
21	3, 20–22	rdispls, recvtypes, comm, request),
22	MPI_DIST_GRAPH_NEIGHBORS_COUNT	$(\text{comm}, \ \underline{41})$
23	indegree, outdegree, weighted), $\underline{20}$	MPI_NEIGHBOR_ALLGATHER, 3, 29, 30,
24	MPI_GRAPH_CREATE, 2, 3, 6, 8, 12, 15,	37
25	18, 19, 25, 26	MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount,
26	MPI_GRAPH_CREATE(comm_old, nnodes,	sendtype, recvbuf, recvcount, recvtype,
27	$index, edges, reorder, comm_graph),$	comm), 27
28	<u>6</u>	MPI_NEIGHBOR_ALLGATHER_INIT(sendbuf,
29	MPI_GRAPH_CREATE(comm, nnodes, in-	sendcount, sendtype, recvbuf, recvcount,
30	dex, edges, reorder, comm_graph),	recvtype, comm, info, request), $\underline{42}$
31	25	MPI_NEIGHBOR_ALLGATHERV, 3, 38
32	MPI_GRAPH_GET, 3, 15	MPI_NEIGHBOR_ALLGATHERV(sendbuf,
33	MPI_GRAPH_GET(comm, maxindex, maxed)	ges, sendcount, sendtype, recvbuf, recvcounts,
34	index, edges), $\frac{15}{15}$	displs, recvtype, comm), $\frac{29}{29}$
35	MPI_GRAPH_MAP, 3	MPI_NEIGHBOR_ALLGATHERV_INIT(sendbuf,
36	MPI_GRAPH_MAP(comm, nnodes, index,	sendcount, sendtype, recvbuf, recvcounts,
37	edges, newrank), $\underline{25}$	displs, recvtype, comm, info, request),
38	MPI_GRAPH_MAP(comm, nnodes, index,	$\frac{43}{1000000000000000000000000000000000000$
39	edges, newrank), 26	MPI_NEIGHBOR_ALLTOALL, 3, 32, 33, 39
40	MPI_GRAPH_NEIGHBORS, 3, 18, 19, 26	MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount,
41	MPI_GRAPH_NEIGHBORS(comm, rank, ma	xneighboushdtype, recvbuf, recvcount, recvtype,
42	neighbors), $\underline{18}$	$\frac{\text{comm}}{31}$
43	MPI_GRAPH_NEIGHBORS_COUNT, 3, 18,	MPI_NEIGHBOR_ALLTOALL_INIT(sendbuf,
44	19 MDI ODADII NEIQUDODO COUNTU	senacount, senatype, recvour, recvcount,
45	MFI_GRAFH_NEIGHBURS_COUNT(comm	, recvtype, comm, inio, request), <u>44</u>
46	rank, integration $\frac{18}{15}$	MELNEIGHDOR_ALLTOALLV, 5, 40
47	MPL_GRAPHDIM5_GE1, 3, 15 MDL_CDADIIDIMS_CET(wiri_weiGHBOK_ALLIOALLV (sendbui, sendcounts,
48	MF1_GKAFHDIM5_GE1 (comm, nnodes, ned	ges), saispis, senatype, recvour, recvcounts,

rdispls, recvtype, comm), $\underline{33}$	1
MPI_NEIGHBOR_ALLTOALLV_INIT(sendbuf,	2
sendcounts, sdispls, sendtype, recvbuf,	3
recvcounts, rdispls, recvtype, comm,	4
info, request), $\underline{45}$	5
MPI_NEIGHBOR_ALLTOALLW, 3, 34, 42	6
MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts,	7
sdispls, sendtypes, recvbuf, recvcounts,	8
rdispls, recvtypes, comm), <u>35</u>	9
MPI_NEIGHBOR_ALLTOALLW_INIT(sendbuf,	10
sendcounts, sdispls, sendtypes, recvbuf,	11
recvcounts, rdispls, recvtypes, comm,	12
info, request), $\underline{46}$	13
MPI_SENDRECV, 22	14
MPI_TOPO_TEST, 3, 14	15
MPI_TOPO_TEST(comm, status), $\underline{14}$	16
TERM·Cartesian	17
topology 2 4	18
TEBM: collective communication	19
neighborhood 26	20
TEBM:distributed graph	21
topology 2 8	22
TERM:graph	23
topology 2 6	24
TERM:neighborhood collective communica-	25
tion. 26	26
nonblocking. 37	27
TERM: persistent communication requests	28
collective persistent, 42	29
TERM:topologies. 1	30
TERM:topology	31
Cartesian, 2, 4	32
distributed graph, 2, 8	33
graph, 2, 6	25
virtual, 2	36
TERM: virtual topology, 2	27
	38
	30
	40
	40
	41
	43
	10

 $\frac{47}{48}$