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

Message Passing Interface Forum

February 21, 2015

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

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal 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 $(1,1)$:	$\operatorname{rank} 3$

7.3 Embedding in MPI

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

39 40 41

42

29 30

31

32 33 34

35

36

37

38

7.4 Overview of the Functions

⁴³ MPI supports three topology types: Cartesian, graph, and distributed graph. The function
 ⁴⁴ MPI_CART_CREATE is used to create Cartesian topologies, the function

⁴⁵ MPI_GRAPH_CREATE is used to create graph topologies, and the functions

⁴⁶ MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE are used to cre-⁴⁷ ate distributed graph topologies. These topology creation functions are collective. As with

48

Unofficial Draft for Comment Only

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 6). In analogy to function MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol.

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

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 6, 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

CHAPTER 7. PROCESS TOPOLOGIES

MPI_INE	GHBOR_ALLTOALLW.	
7.5 To	opology Constructo	rs
7.5.1 C	artesian Constructor	
MPI_CAF	RT_CREATE(comm_old,	, ndims, dims, periods, reorder, comm_cart)
IN	comm_old	input communicator (handle)
IN	ndims	number of dimensions of Cartesian grid (integer)
IN	dims	integer array of size ndims specifying the number of processes in each dimension
IN	periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
IN	reorder	ranking may be reordered (true) or not (false) (logical)
OUT	comm_cart	communicator with new Cartesian topology (handle)
		<pre>iods[], int reorder, MPI_Comm *comm_cart) dims, dims, periods, reorder, comm_cart, ierror)</pre>
		-
	C(MPI_Comm), INTENT(CGER, INTENT(IN) ::	<pre>IN) :: comm_old ndims, dims(ndims)</pre>
		periods(ndims), reorder
	C(MPI_Comm), INTENT(
INTE	GER, OPTIONAL, INTE	NT(OUT) :: ierror
INTE		DIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR) , DIMS(*), COMM_CART, IERROR DRDER
MPI_	CART_CREATE return	as a handle to a new communicator to which the Cartesian
* 00		d. If reorder = false then the rank of each process in the
0	*	hk in the old group. Otherwise, the function may reorder
-		b choose a good embedding of the virtual topology onto otal size of the Cartesian grid is smaller than the size of
	,	me processes are returned MPI_COMM_NULL, in analogy to
		zero then a zero-dimensional Cartesian topology is created.
The call inegative.	is erroneous if it specif	ies a grid that is larger than the group size or if ndims is
negative.		

44

7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE

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

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

	_CREATE(nnodes, ndims, dim		4
	CREATE(Iniodes, Italins, and	5)	5
IN	nnodes	number of nodes in a grid (integer)	6
IN	ndims	number of Cartesian dimensions (integer)	7
INOUT	dims	integer array of size ndims specifying the number of	8 9
		nodes in each dimension	10
			11
int MPI_Di	<pre>ms_create(int nnodes, in</pre>	nt ndims, int dims[])	12
MPI Dims o	create(nnodes, ndims, dir	ns. ierror)	13
	ER, INTENT(IN) :: nnodes	•	14
	ER, INTENT(INOUT) :: dir		15
	ER, OPTIONAL, INTENT(OUT)		16
	,,,,,		17
	CREATE(NNODES, NDIMS, DIN		18
INTEGE	ER NNODES, NDIMS, DIMS(*)), IERROR	19

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

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

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

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

Example 7.1

dims	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

7.5.3 Graph Constructor

 $\mathbf{2}$ 3 4MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph) 5IN 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) (logical) 11 12OUT comm_graph communicator with graph topology added (handle) 13 14 int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[], 15const int edges[], int reorder, MPI_Comm *comm_graph) 16 MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph, 17ierror) 18 TYPE(MPI_Comm), INTENT(IN) :: comm_old 19 INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*) 20LOGICAL, INTENT(IN) :: reorder 21TYPE(MPI_Comm), INTENT(OUT) :: comm_graph 22INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23 24MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH, 25IERROR) 26 INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR 27 LOGICAL REORDER 28MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph 29 topology information is attached. If reorder = false then the rank of each process in the 30 31

new group is identical to its rank in the old group. Otherwise, the function may reorder the
 processes. If the size, nnodes, of the graph is smaller than the size of the group of comm_old,
 then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE
 and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL
 is returned in all processes. The call is erroneous if it specifies a graph that is larger than
 the group size of the input communicator.

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

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

47 Example 7.2

48

46

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

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

Then, the input arguments are:

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

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

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

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

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

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

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

which are the vectors defining the graph structure.

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

Unofficial Draft for Comment Only

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 11 communication 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

22

23

7

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

23 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 38	IN	destweights	weights of the edges out of the calling process (array of non-negative integers)
39 40	IN	info	hints on optimization and interpretation of weights (handle)
41 42	IN	reorder	the ranks may be reordered (true) or not (false) (logical)
43 44 45	OUT	comm_dist_graph	communicator with distributed graph topology (han-dle)
46 47 48	int MPI_I		t(MPI_Comm comm_old, int indegree, 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), outdegree,
        destinations(outdegree)
    INTEGER, INTENT(IN) :: sourceweights(*), destweights(*)
                                                                                 10
    TYPE(MPI_Info), INTENT(IN) ::
                                    info
                                                                                 11
    LOGICAL, INTENT(IN) ::
                            reorder
                                                                                 12
    TYPE(MPI_Comm), INTENT(OUT) ::
                                     comm_dist_graph
                                                                                 13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 14
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
                                                                                 15
             OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
                                                                                 16
             COMM_DIST_GRAPH, IERROR)
                                                                                 17
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
                                                                                 18
        DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
                                                                                 19
    LOGICAL REORDER
                                                                                 20
```

MPI_DIST_GRAPH_CREATE_ADJACENT returns a handle to a new communicator to which the distributed graph topology information is attached. Each process passes all information about 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 46 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

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

40

47

1 2		U 1 <i>i</i>	that MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are they are special values for the total array argument. In
3	-		IPI_WEIGHTS_EMPTY are objects like MPI_BOTTOM (not
4		r initialization or assignm	-
5		· · · · · · · · ·	
6			case of an empty weights array argument passed while
7			ph, one should not pass NULL because the value of equal to NULL. The value of this argument would then
8			IPI_UNWEIGHTED to the implementation. In this case
9 10		0	d be used instead. (<i>End of advice to users.</i>)
11	Adi	vice to implementors. It	is recommended that MPI_UNWEIGHTED not be imple-
12 13		nted as NULL. ($End \ of \ ad$	-
14 15 16			and compatibility, MPI_UNWEIGHTED may still be imple- $\underline{B.2.}$ (End of rationale.)
17	The	meaning of the info and	reorder arguments is defined in the description of the
18	following	9	resider arguments is defined in the description of the
19	8		
20			
21	MPI_DIS	•	m_old, n, sources, degrees, destinations, weights, info,
22		reorder, comm_dist	_graph)
23	IN	comm_old	input communicator (handle)
24 25 26	IN	n	number of source nodes for which this process specifies edges (non-negative integer)
27 28	IN	sources	array containing the n source nodes for which this process specifies edges (array of non-negative integers)
29 30 31	IN	degrees	array specifying the number of destinations for each source node in the source node array (array of non- negative integers)
32 33 34	IN	destinations	destination nodes for the source nodes in the source node array (array of non-negative integers)
35 36	IN	weights	weights for source to destination edges (array of non-negative integers)
37 38	IN	info	hints on optimization and interpretation of weights (handle)
39 40	IN	reorder	the process may be reordered (true) or not (false) (log-ical)
41 42 43	OUT	comm_dist_graph	communicator with distributed graph topology added (handle)
44	int MDT	Digt graph grapt (MD	[_Comm comm_old, int n, const int sources[],
45 46	IIIC MPI_		es[], const int destinations[],
40 47			ts[], MPI_Info info, int reorder,
48		MPI_Comm *comm_	
		-	

MPI_DIST_GRAPH_CREATE returns a handle to a new communicator to which the 16 distributed graph topology information is attached. Concretely, each process calls the con-17structor with a set of directed (source, destination) communication edges as described below. 18 Every process passes an array of n source nodes in the sources array. For each source node, a 19non-negative number of destination nodes is specified in the degrees array. The destination 20nodes are stored in the corresponding consecutive segment of the destinations array. More 21precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the 22 j-th such edge stored in destinations[degrees[0]+...+degrees[i-1]+j]. The weight of this edge 23is stored in weights $[degrees[0]+\ldots+degrees[i-1]+i]$. Both the sources and the destinations 24arrays may contain the same node more than once, and the order in which nodes are listed 25as destinations or sources is not significant. Similarly, different processes may specify edges 26with the same source and destination nodes. Source and destination nodes must be pro-27cess ranks of comm_old. Different processes may specify different numbers of source and 28 destination nodes, as well as different source to destination edges. This allows a fully dis-29 tributed specification of the communication graph. Isolated processes (i.e., processes with 30 no outgoing or incoming edges, that is, processes that do not occur as source or destination 31 node in the graph specification) are allowed. 32

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

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

Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count arguments of later communication calls along specific edges could be used as their edge weights. Multiplicity of edges can likewise indicate more intense communication between pairs of processes. However, the exact meaning of edge weights is not specified by the MPI standard and is left to the implementation. In C or Fortran, an application can supply

1

2

3

4

5

6

7

8

9 10

11

12

13

14

15

33

34

35

36

37

38

39

40

41

the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the 1 same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some but not 2 3 all processes of comm_old. If the graph is weighted but n = 0, then MPI_WEIGHTS_EMPTY or any arbitrary array may be passed to weights. Note that MPI_UNWEIGHTED and 4 MPI_WEIGHTS_EMPTY are not special weight values; rather they are special values for the 5total array argument. In Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects 6 like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4. 7 8 Advice to users. In the case of an empty weights array argument passed while 9 constructing a weighted graph, one should not pass NULL because the value of 10 MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then 11 be indistinguishable from MPI_UNWEIGHTED to the implementation. 12MPI_WEIGHTS_EMPTY should be used instead. (End of advice to users.) 13 14 Advice to implementors. It is recommended that MPI_UNWEIGHTED not be imple-15mented as NULL. (End of advice to implementors.) 16 17*Rationale.* To ensure backward compatibility, MPI_UNWEIGHTED may still be imple-18 mented as NULL. See Annex B.2. (End of rationale.) 19 20The meaning of the weights argument can be influenced by the info argument. Info 21arguments can be used to guide the mapping; possible options include minimizing the 22 maximum number of edges between processes on different SMP nodes, or minimizing the 23 sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it 24is valid for an MPI implementation not to do any reordering. An MPI implementation may 25specify more info key-value pairs. All processes must specify the same set of key-value info 26pairs. 2728Advice to implementors. MPI implementations must document any additionally 29 supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the 30 default creation of the distributed graph topology to the MPI library. 31

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

38 39 40

41

42 43

32

33

34

35

36

37

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

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

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

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

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

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

In both cases above, the application could supply $\mathsf{MPI_UNWEIGHTED}$ 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:

```
/*
```

```
/* 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_GRAF	PHDIMS_GET(comm, nnodes,	nedges)	1
IN	comm	communicator for group with graph structure (handle)	2
OUT	nnodes	number of nodes in graph (integer) (same as number of processes in the group)	3
OUT	nedges	number of edges in graph (integer)	5 6
001	1104805	number of edges in graph (integer)	7
int MPI_G	raphdims_get(MPI_Comm con	nm, int *nnodes, int *nedges)	8
MPT Graph	dims_get(comm, nnodes, no	edges jerror)	9
-	MPI_Comm), INTENT(IN) ::	comm	10 11
INTEG	ER, INTENT(OUT) :: nnode	es, nedges	12
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	13
MPI_GRAPH	DIMS_GET(COMM, NNODES, NI	EDGES, IERROR)	14
INTEG	ER COMM, NNODES, NEDGES,	IERROR	15
Functi	ons MPI_GRAPHDIMS_GET	and MPI_GRAPH_GET retrieve the graph-topology	16 17
		communicator by MPI_GRAPH_CREATE.	18
	· ·	_GRAPHDIMS_GET can be used to dimension the	19
vectors ind	ex and edges correctly for the	following call to MPI_GRAPH_GET.	20
			21
MPI_GRAF	PH_GET(comm, maxindex, max	xedges, index, edges)	22 23
IN	comm	communicator with graph structure (handle)	23 24
IN	maxindex	length of vector index in the calling program	25
		(integer)	26
IN	maxedges	length of vector edges in the calling program	27
	J. J	(integer)	28 29
OUT	index	array of integers containing the graph structure (for	30
		details see the definition of $MPI_GRAPH_CREATE)$	31
OUT	edges	array of integers containing the graph structure	32
			33
int MPI_G		<pre>int maxindex, int maxedges, int index[],</pre>	34 35
	<pre>int edges[])</pre>		36
MPI_Graph	_get(comm, maxindex, maxe	edges, index, edges, ierror)	37
	MPI_Comm), INTENT(IN) ::		38
	ER, INTENT(IN) :: maxing	0	39
	INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges) INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
			41 42
		EDGES, INDEX, EDGES, IERROR)	43
INTEG	ER CUMMI, MAALNUEA, MAXEDO	GES, INDEX(*), EDGES(*), IERROR	44
			45
			46
			47

```
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 (in-
4
                                             teger)
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
                                             length of vectors dims, periods, and
                  maxdims
24
                                             coords in the calling program (integer)
25
26
       OUT
                 dims
                                             number of processes for each Cartesian dimension (ar-
27
                                             ray of integer)
28
       OUT
                  periods
                                             periodicity (true/false) for each Cartesian dimension
29
                                             (array of logical)
30
       OUT
                 coords
                                             coordinates of calling process in Cartesian structure
31
                                             (array of integer)
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 5
OUT	rank	rank of specified process (integer)	6 7
int MPI_C	art_rank(MPI_Comm comm,	const int coords[], int *rank)	8
MPI_Cart_	rank(comm, coords, rank,	ierror)	9 10
	MPI_Comm), INTENT(IN) ::		11
	ER, INTENT(IN) :: coord:	s(*)	12
	ER, INTENT(OUT) :: rank ER, OPTIONAL, INTENT(OUT)) ·· ierror	13
			14 15
	RANK(COMM, COORDS, RANK, ER COMM, COORDS(*), RANK		16
			17
		structure, the function MPI_CART_RANK trans-	18
lates the lo routines.	gical process coordinates to p	process ranks as they are used by the point-to-point	19
	mension i with periods(i) = tr	ue, if the coordinate, coords(i), is out of range, that	20 21
		t is shifted back to the interval	22
		Out-of-range coordinates are erroneous for non-	23
periodic di		in an include the standard standard in the standard standard standard standard standard standard standard stand	24
	is returned in rank.	imensional Cartesian topology, coords is not signif-	25
			26 27
		P	28
	COORDS(comm, rank, maxe	,	29
IN	comm	communicator with Cartesian structure (handle)	30
IN	rank	rank of a process within group of $comm\ (\mathrm{integer})$	31 32
IN	maxdims	length of vector ${\sf coords}$ in the calling program (integer)	32
OUT	coords	integer array (of size ndims) containing the Cartesian coordinates of specified process (array of integers)	34
		coordinates of specified process (array of integers)	35 36
int MPI_C	art_coords(MPI_Comm comm	, int rank, int maxdims, int coords[])	37
MPT Cart	coords(comm, rank, maxdin	ms coords jerror)	38
	MPI_Comm), INTENT(IN) ::		39
INTEG	ER, INTENT(IN) :: rank,	maxdims	40 41
	ER, INTENT(OUT) :: coord		42
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	43
	COORDS(COMM, RANK, MAXDI		44
INTEG	ER COMM, RANK, MAXDIMS, (COORDS(*), IERROR	45
	verse mapping, rank-to-coor COORDS.	dinates translation is provided by	46 47 48
			40

Unofficial Draft for Comment Only

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

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 (han-
24
                                           dle)
25
26
       OUT
                indegree
                                           number of edges into this process (non-negative inte-
27
                                           ger)
28
       OUT
                outdegree
                                           number of edges out of this process (non-negative in-
29
                                           teger)
30
       OUT
                weighted
                                           false if MPI_UNWEIGHTED was supplied during cre-
31
                                           ation, 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, destinations, destweights)	, maxindegree, sources, sourceweights, maxoutdegree,	$\frac{1}{2}$
IN	comm	communicator with distributed graph topology (han- dle)	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 14
OUT	destinations	processes for which the calling process is a source (array of non-negative integers)	15 16
OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)	17 18 19
int MPI_D	010	Comm comm, int maxindegree, int sources[], int maxoutdegree, int destinations[],	20 21 22 23
TYPE(INTEG INTEG d INTEG		<pre>degree, maxoutdegree ces(maxindegree),) destweights(*)</pre>	23 24 25 26 27 28 29 30 31
INTEG	MAXOUTDEGREE, DESTIN	KINDEGREE, SOURCES, SOURCEWEIGHTS, ATIONS, DESTWEIGHTS, IERROR) JRCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, HTS(*), IERROR	32 33 34 35 36
MPI_DIST call to MP tially by p	_GRAPH_NEIGHBORS_COUN PI_DIST_GRAPH_CREATE_AD processes other than the call	of edges into and out of the process returned by IT are the total number of such edges given in the DJACENT or MPI_DIST_GRAPH_CREATE (poten- ing process in the case of the defined edges are all counted and returned by	30 37 38 39 40 41

MPL DIST CRAPH NEICHBORS(comm maxindegree sources sourceweights maxoutdegree

MPI_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

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

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

¹² 7.5.6 Cartesian Shift Coordinates

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 ¹⁵ the 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 ¹⁷ the 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.

```
21
22
```

 $\mathbf{5}$

6

7

8

9

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

			, Talk_Source, Talk_dest)
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)
31 32 33 34 35 36 37 38 39	MPI_Cart_s TYPE(N INTEGH INTEGH	<pre>int *rank_source, in</pre>	isp, rank_source, rank_dest, ierror) comm tion, disp _source, rank_dest
40 41 42 43 44 45 46 47	INTEGE The dir The dimens Depend tion, MPI_C of an end-of	ER COMM, DIRECTION, DISP ection argument indicates the ions are numbered from 0 to ding on the periodicity of the CART_SHIFT provides the ide ff shift, the value MPI_PROC_	ISP, RANK_SOURCE, RANK_DEST, IERROR) , RANK_SOURCE, RANK_DEST, IERROR e coordinate dimension to be traversed by the shift. ndims-1, where ndims is the number of dimensions. c Cartesian group in the specified coordinate direc- ntifiers for a circular or an end-off shift. In the case NULL may be returned in rank_source or rank_dest,
48	mulcating t	nat the source of the destina	tion for the shift is out of range.

 $\mathbf{2}$

 $25 \\ 26$

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.

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

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

7.5.7 Partitioning of Cartesian Structures

MPI_CART_SUB(comm, remain_dims, newcomm)

	· ·	,	00
IN	comm	communicator with Cartesian structure (handle)	31
IN	remain_dims	the i-th entry of remain_dims specifies whether the	32
	_	i-th dimension is kept in the subgrid (true) or is drop-	33
		ped (false) (logical vector)	34
OUT	newcomm	communicator containing the subgrid that includes	35
001	newconnin	the calling process (handle)	36
		the carrier process (nandre)	37
int MDT Ca	art gub (MDI Comm comm co	onst int remain_dims[], MPI_Comm *newcomm)	38
IIIC MFI_Ca	fit_sub(MF1_comm comm, cc	mst int remain_dims[], MF1_Comm *newComm)	39
MPI_Cart_s	sub(comm, remain_dims, ne	ewcomm, ierror)	40 41
TYPE(M	<pre>MPI_Comm), INTENT(IN) ::</pre>	comm	41 42
LOGICA	AL, INTENT(IN) :: remain	n_dims(*)	42
	<pre>IPI_Comm), INTENT(OUT) ::</pre>		43
INTEGE	ER, OPTIONAL, INTENT(OUT)	:: ierror	45
MPI_CART_S	SUB(COMM, REMAIN_DIMS, NE	WCOMM, IERROR)	46
	ER COMM, NEWCOMM, IERROR	•	47
LOGICA	AL REMAIN_DIMS(*)		48

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 5associated with a zero-dimensional Cartesian topology. (This function is closely related to 6 MPI_COMM_SPLIT.) 7 8 Example 7.8 9 Assume that MPI_CART_CREATE(..., comm) has defined a $(2 \times 3 \times 4)$ grid. Let 10 remain_dims = (true, false, true). Then a call to 11 12MPI_CART_SUB(comm, remain_dims, comm_new); 13 will create three communicators each with eight processes in a 2×4 Cartesian topology. 14 If remain_dims = (false, false, true) then the call to $MPI_CART_SUB(comm, remain_dims,$ 15 comm_new) will create six non-overlapping communicators, each with four processes, in a 16 one-dimensional Cartesian topology. 1718 7.5.8 Low-Level Topology Functions 19 20The two additional functions introduced in this section can be used to implement all other 21topology functions. In general they will not be called by the user directly, unless he or she 22 is creating additional virtual topology capability other than that provided by MPI. The two 23 calls are both local. 24 2526MPI_CART_MAP(comm, ndims, dims, periods, newrank) 27IN comm input communicator (handle) 28IN ndims number of dimensions of Cartesian structure (integer) 29 30 IN dims integer array of size ndims specifying the number of 31 processes in each coordinate direction 32 IN periods logical array of size ndims specifying the periodicity 33 specification in each coordinate direction 34 OUT reordered rank of the calling process; newrank 35 MPI_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 41MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror) 42TYPE(MPI_Comm), INTENT(IN) :: comm 43INTEGER, INTENT(IN) :: ndims, dims(ndims) 44 LOGICAL, INTENT(IN) :: periods(ndims) 45INTEGER, INTENT(OUT) :: newrank 46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 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 input communicator (handle) comm 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 be-33 long 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) :: 42 ierror 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 5 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 collective 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(sendbuf, 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 (non-negative integer)	15 16
		- · · ·	17
IN	sendtype	data type of send buffer elements (handle)	
OUT	recybuf	starting address of receive buffer (choice)	18
001	1001001	starting address of receive sunor (choice)	19
IN	recvcount	number of elements received from each neighbor (non-	20
		negative integer)	21
IN	recvtype	data type of receive buffer elements (handle)	22
	reeveype	data type of receive surfer clements (nanale)	23
IN	comm	communicator with topology structure (handle)	24

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);

 $46 \\ 47$

```
int *srcs=(int*)malloc(indegree*sizeof(int));
1
      int *dsts=(int*)malloc(outdegree*sizeof(int));
\mathbf{2}
     MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
3
                                     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.
```

7.6. NEIGHBORHOOD COLLECTIVE COMMUNICATION

The "in place" option is not meaningful for this operation. 1 The vector variant of MPI_NEIGHBOR_ALLGATHER allows one to gather different 2 numbers of elements from each neighbor. 3 4 5MPI_NEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, 6 recvtype, comm) 7 IN sendbuf starting address of send buffer (choice) 8 9 IN sendcount number of elements sent to each neighbor (non-negative 10 integer) 11 IN sendtype data type of send buffer elements (handle) 12 OUT recvbuf starting address of receive buffer (choice) 13 14 IN recvcounts non-negative integer array (of length indegree) con-15taining the number of elements that are received from 16 each neighbor 17IN displs integer array (of length indegree). Entry i specifies the 18 displacement (relative to recvbuf) at which to place the 19 incoming data from neighbor i 20IN recvtype data type of receive buffer elements (handle) 2122 communicator with topology structure (handle) IN comm 23 24int MPI_Neighbor_allgatherv(const void* sendbuf, int sendcount, 25MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], 26const int displs[], MPI_Datatype recvtype, MPI_Comm comm) 27MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 28displs, recvtype, comm, ierror) 29 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 30 TYPE(*), DIMENSION(..) :: recvbuf 31 INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*) 32 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 33 TYPE(MPI_Comm), INTENT(IN) :: comm 34INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3536 MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 37 DISPLS, RECVTYPE, COMM, IERROR) 38 <type> SENDBUF(*), RECVBUF(*) 39 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 40 IERROR 41 42

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); int *srcs=(int*)malloc(indegree*sizeof(int)); 43

44

 $45 \\ 46$

47

```
int *dsts=(int*)malloc(outdegree*sizeof(int));
1
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
^{2}
                                 outdegree,dsts,MPI_UNWEIGHTED);
3
     int k,l;
4
5
     /* assume sendbuf and recvbuf are of type (char*) */
6
     for(k=0; k<outdegree; ++k)</pre>
7
       MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
8
9
     for(l=0; l<indegree; ++1)</pre>
10
       MPI_Irecv(recvbuf+displs[l]*extent(recvtype),recvcounts[l],recvtype,
11
                  srcs[1],...);
12
13
     MPI_Waitall(...);
14
15
```

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

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

All arguments are significant on all processes and the argument comm must have iden-tical values on all processes.

7.6.2 Neighbor Alltoall

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

```
MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                          1
               comm)
                                                                                          2
                                                                                          3
  IN
           sendbuf
                                       starting address of send buffer (choice)
                                                                                          4
  IN
           sendcount
                                       number of elements sent to each neighbor (non-negative
                                                                                          5
                                       integer)
                                                                                          6
  IN
           sendtype
                                       data type of send buffer elements (handle)
                                                                                          7
                                                                                          8
  OUT
           recvbuf
                                       starting address of receive buffer (choice)
                                                                                          9
  IN
           recvcount
                                       number of elements received from each neighbor (non-
                                                                                         10
                                       negative integer)
                                                                                         11
  IN
                                       data type of receive buffer elements (handle)
           recvtype
                                                                                         12
                                                                                         13
  IN
           comm
                                       communicator with topology structure (handle)
                                                                                         14
                                                                                         15
int MPI_Neighbor_alltoall(const void* sendbuf, int sendcount,
                                                                                         16
               MPI_Datatype sendtype, void* recvbuf, int recvcount,
                                                                                         17
               MPI_Datatype recvtype, MPI_Comm comm)
                                                                                         18
MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                         19
               recvtype, comm, ierror)
                                                                                         20
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                         21
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                         22
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                         23
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                         24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                         25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         26
                                                                                         27
MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                         28
               RECVTYPE, COMM, IERROR)
                                                                                         29
     <type> SENDBUF(*), RECVBUF(*)
                                                                                         30
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
                                                                                         31
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                         32
graph communicators as described in Section 7.6. If comm is a distributed graph commu-
                                                                                         33
                                                                                         34
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
and receives from each of its incoming neighbors:
                                                                                         35
                                                                                         36
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
                                                                                         37
int *srcs=(int*)malloc(indegree*sizeof(int));
                                                                                         38
int *dsts=(int*)malloc(outdegree*sizeof(int));
                                                                                         39
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                                                                         40
                            outdegree,dsts,MPI_UNWEIGHTED);
                                                                                         41
int k,l;
                                                                                         42
                                                                                         43
/* assume sendbuf and recvbuf are of type (char*) */
                                                                                         44
for(k=0; k<outdegree; ++k)</pre>
                                                                                         45
  MPI_Isend(sendbuf+k*sendcount*extent(sendtype),sendcount,sendtype,
                                                                                         46
             dsts[k],...);
                                                                                         47
                                                                                         48
```

```
for(l=0; l<indegree; ++1)</pre>
1
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype),recvcount,recvtype,
^{2}
3
                    srcs[1],...);
4
     MPI_Waitall(...);
5
6
          The type signature associated with sendcount, sendtype, at a process must be equal to
7
      the type signature associated with recvcount, recvtype at any other process. This implies
8
      that the amount of data sent must be equal to the amount of data received, pairwise between
9
      every pair of communicating processes. Distinct type maps between sender and receiver are
10
      still allowed.
11
          The "in place" option is not meaningful for this operation.
12
          All arguments are significant on all processes and the argument comm must have iden-
13
      tical values on all processes.
14
          The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different
15
      numbers of elements to and from each neighbor.
16
17
18
      MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
19
                      rdispls, recvtype, comm)
20
        IN
                  sendbuf
                                                starting address of send buffer (choice)
21
22
        IN
                  sendcounts
                                                non-negative integer array (of length outdegree) speci-
23
                                                fying the number of elements to send to each neighbor
24
        IN
                  sdispls
                                                integer array (of length outdegree). Entry j specifies
25
                                                the displacement (relative to sendbuf) from which to
26
                                                send the outgoing data to neighbor j
27
        IN
                  sendtype
                                                data type of send buffer elements (handle)
28
29
        OUT
                  recvbuf
                                                starting address of receive buffer (choice)
30
        IN
                  recvcounts
                                                non-negative integer array (of length indegree) speci-
31
                                                fying the number of elements that are received from
32
                                                each neighbor
33
        IN
                  rdispls
                                                integer array (of length indegree). Entry i specifies the
34
                                                displacement (relative to recvbuf) at which to place the
35
                                                incoming data from neighbor i
36
        IN
                                                data type of receive buffer elements (handle)
                  recvtype
37
38
        IN
                  comm
                                                communicator with topology structure (handle)
39
40
      int MPI_Neighbor_alltoallv(const void* sendbuf, const int sendcounts[],
41
                      const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
42
                      const int recvcounts[], const int rdispls[],
43
                      MPI_Datatype recvtype, MPI_Comm comm)
44
     MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
45
                      recvcounts, rdispls, recvtype, comm, ierror)
46
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
47
          TYPE(*), DIMENSION(..) :: recvbuf
48
```

```
INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
        rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::
                                   comm
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                       ierror
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
             RECVCOUNTS, RDISPLS,
    RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR
                                                                                12
```

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:

<pre>MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted); int *srcs=(int*)malloc(indegree*sizeof(int)); int *dsts=(int*)malloc(outdegree*sizeof(int)); MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED, outdegree,dsts,MPI_UNWEIGHTED);</pre>
int k,l;
<pre>/* assume sendbuf and recvbuf are of type (char*) */ for(k=0; k<outdegree; ++k)="" dsts[k],);<="" mpi_isend(sendbuf+sdispls[k]*extent(sendtype),sendcounts[k],sendtype,="" pre=""></outdegree;></pre>
<pre>for(1=0; l<indegree; ++1)="" mpi_irecv(recvbuf+rdispls[1]*extent(recvtype),recvcounts[1],recvtype,<="" pre=""></indegree;></pre>
srcs[1],);

MPI_Waitall(...);

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

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

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

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

1

2

3

4

56

7

8

9

10

11

13

14

15

32 33

34 35

36

37

38

39

40

41

42

43

44

45

46

47

12	MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)		
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
7 8 9 10	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)
11 12 13	IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)
14 15	OUT	recvbuf	starting address of receive buffer (choice)
16 17 18	IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor
19 20 21 22	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)
23 24 25 26	IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)
20 27 28	IN	comm	communicator with topology structure (handle)
int MPI_Neighbor_alltoallw(const void* sendbuf, const int sendcounts[], const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void* recvbuf, const int recvcounts[], const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm)			
34 35 36 37 38 39 40 41 42 43	 MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, ierror) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*) TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror 		
44 45 46 47 48	<pre>MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,</pre>		

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

int k,l;

```
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
   MPI_Isend(sendbuf+sdispls[k],sendcounts[k], sendtypes[k],dsts[k],...);</pre>
```

```
for(1=0; l<indegree; ++1)
   MPI_Irecv(recvbuf+rdispls[1],recvcounts[1], recvtypes[1],srcs[1],...);</pre>
```

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

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

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

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

7.7 Nonblocking Neighborhood Communication on Process Topologies

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

```
7.7.1
            Nonblocking Neighborhood Gather
1
\mathbf{2}
3
^{4}
     MPI_INEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
5
                    comm, request)
6
       IN
                 sendbuf
                                             starting address of send buffer (choice)
7
       IN
                 sendcount
                                             number of elements sent to each neighbor (non-negative
8
                                             integer)
9
10
       IN
                 sendtype
                                             data type of send buffer elements (handle)
11
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
12
       IN
                                             number of elements received from each neighbor (non-
                 recvcount
13
                                             negative integer)
14
15
       IN
                 recvtype
                                             data type of receive buffer elements (handle)
16
       IN
                                             communicator with topology structure (handle)
                 comm
17
       OUT
                                             communication request (handle)
                 request
18
19
     int MPI_Ineighbor_allgather(const void* sendbuf, int sendcount,
20
21
                    MPI_Datatype sendtype, void* recvbuf, int recvcount,
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
22
23
     MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
24
                    recvtype, comm, request, ierror)
25
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
          INTEGER, INTENT(IN) :: sendcount, recvcount
28
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
          TYPE(MPI_Comm), INTENT(IN) :: comm
30
          TYPE(MPI_Request), INTENT(OUT) :: request
31
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
32
33
     MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
34
                    RECVTYPE, COMM, REQUEST, IERROR)
          <type> SENDBUF(*), RECVBUF(*)
35
36
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
37
         This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHER.
38
39
40
41
42
43
44
45
46
47
48
```

MPI_INEI	GHBOR_ALLGATHERV(sendburrecvtype, comm, request)	ıf, sendcount, sendtype, recvbuf, recvcounts, displs,	1
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	4 5 6
IN	sendtype	data type of send buffer elements (handle)	7
OUT	recvbuf	starting address of receive buffer (choice)	8
IN	recvcounts	non-negative integer array (of length indegree) con- taining the number of elements that are received from each neighbor	9 10 11 12
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	13 14 15
IN	recvtype	data type of receive buffer elements (handle)	16
IN	comm	communicator with topology structure (handle)	17 18
OUT	request	communication request (handle)	19 20
<pre>MPI_Datatype Sendtype, Void* Tecvour, const Tht Tecvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre> MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: sendcount INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*) TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHERV.</type>		21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	
			44 45

```
38
                                                   CHAPTER 7. PROCESS TOPOLOGIES
     7.7.2
            Nonblocking Neighborhood Alltoall
1
\mathbf{2}
3
^{4}
     MPI_INEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
5
                    comm, request)
6
       IN
                 sendbuf
                                             starting address of send buffer (choice)
7
       IN
                 sendcount
                                             number of elements sent to each neighbor (non-negative
8
                                             integer)
9
10
       IN
                 sendtype
                                             data type of send buffer elements (handle)
11
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
12
       IN
                                             number of elements received from each neighbor (non-
                 recvcount
13
                                             negative integer)
14
15
       IN
                 recvtype
                                             data type of receive buffer elements (handle)
16
       IN
                                             communicator with topology structure (handle)
                 comm
17
                                             communication request (handle)
       OUT
                 request
18
19
     int MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount,
20
21
                    MPI_Datatype sendtype, void* recvbuf, int recvcount,
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
22
23
     MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
24
                    recvtype, comm, request, ierror)
25
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
          INTEGER, INTENT(IN) :: sendcount, recvcount
28
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
          TYPE(MPI_Comm), INTENT(IN) :: comm
30
          TYPE(MPI_Request), INTENT(OUT) :: request
31
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
32
33
     MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
34
                    RECVTYPE, COMM, REQUEST, IERROR)
          <type> SENDBUF(*), RECVBUF(*)
35
36
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
37
         This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALL.
38
39
40
41
42
43
44
45
46
47
48
```

	rdispls, recvtype,	comm, request)	2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor	4 5 6
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j	7 8 9
IN	sendtype	data type of send buffer elements (handle)	10
OUT	recvbuf	starting address of receive buffer (choice)	11 12
IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor	13 14 15
IN	rdispls	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	16 17 18 19
IN	recvtype	data type of receive buffer elements (handle)	20
IN	comm	communicator with topology structure (handle)	21
OUT	request	communication request (handle)	22 23
MPI_Inei TYPE TYPE INTE TYPE TYPE TYPE	<pre>const int sdis const int recv MPI_Datatype r ghbor_alltoallv(sen recvcounts, rc (*), DIMENSION(), (*), DIMENSION(), GER, INTENT(IN), AS recvcounts(*), rdis</pre>	YNCHRONOUS :: sendcounts(*), sdispls(*), pls(*) ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: request	24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
<tyr INTE</tyr 	RECVCOUNTS, RE De> SENDBUF(*), REC GER SENDCOUNTS(*), RECVTYPE, COMM, REQ	<pre>SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	39 40 41 42 43 44 45 46

12	MPI_INEI	GHBOR_ALLTOALLW(senc rdispls, recvtypes, cor	lbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, mm, request)
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
7 8 9 10	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)
11 12 13	IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)
14 15	OUT	recvbuf	starting address of receive buffer (choice)
16 17 18	IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor
19 20 21 22 23	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)
24 25 26	IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)
27	IN	comm	communicator with topology structure (handle)
28 29	OUT	request	communication request (handle)
30 31 32 33 34 35	int MPI_	const MPI_Aint sd void* recvbuf, co const MPI_Aint rd	<pre>nst void* sendbuf, const int sendcounts[], displs[], const MPI_Datatype sendtypes[], onst int recvcounts[], displs[], const MPI_Datatype recvtypes[], PI_Request *request)</pre>
35 36	MPT Tnei		
37 38	<pre>MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf</pre>		
39 40			YNCHRONOUS :: recvbuf
40	<pre>INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::</pre>		
42	sdispls(*), rdispls(*)		
43		• -	<pre>(IN), ASYNCHRONOUS :: sendtypes(*),</pre>
44		recvtypes(*)	
45 46		(MPI_Comm), INTENT(IN) (MPI_Request), INTENT(
40		GER, OPTIONAL, INTENT(-
48		, , . (•

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLW.

7.8 An Application Example

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

In each relaxation step each process computes new values for the solution grid function at 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)).

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

```
10
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
                                                                                    11
REAL u(0:101,0:101)
                                                                                    12
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
                                                                                    13
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
                                                                                    14
INTEGER ierr
                                                                                    15
sndbuf(1:100,1) = u( 1,1:100)
                                                                                    16
sndbuf(1:100,2) = u(100,1:100)
                                                                                    17
sndbuf(1:100,3) = u(1:100,
                             1)
                                                                                    18
sndbuf(1:100,4) = u(1:100,100)
                                                                                    19
CALL MPI_NEIGHBOR_ALLTOALL (sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
                                                                                    20
                             comm_cart, ierr)
                                                                                    21
! instead of
                                                                                    22
! DO i=1,num_neigh
                                                                                    23
    CALL MPI_IRECV(rcvbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i-1),&
!
                                                                                    24
!
                    ierr)
    CALL MPI_ISEND(sndbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i ),&
!
                                                                                    26
!
                    ierr)
                                                                                    27
! END DO
                                                                                    28
! CALL MPI_WAITALL (2*num_neigh, rq, statuses, ierr)
                                                                                    29
                                                                                    30
u( 0,1:100) = rcvbuf(1:100,1)
                                                                                    31
u(101,1:100) = rcvbuf(1:100,2)
                                                                                    32
u(1:100, 0) = rcvbuf(1:100,3)
                                                                                    33
u(1:100,101) = rcvbuf(1:100,4)
                                                                                    34
END
                                                                                    35
                                                                                    36
```

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

Unofficial Draft for Comment Only

25

3738

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

Bibliography

		8
		9
[1]	S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole-	10 11
	routed networks for interprocessor communication in multicomputers. In <i>Proceedings of the 1990 Supercomputing Conference</i> , pages 647–656, 1990. 7.1	
[2]	S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication	14
[_]	performance of large multicomputers. In Proceedings of the 1991 International Confer-	15
	ence on Parallel Processing, vol. II (Software), pages II–1 – II–4, 1991. 7.1	
[0]		17
[3]	T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking collectives in quantum	18 19
	mechanical calculations. In Recent Advances in Parallel Virtual Machine and Message	20
	Passing Interface, 15th European PVM/MPI Users' Group Meeting, volume LNCS 5205, pages 55–63. Springer, Sep. 2008. 7.6	20
	pages 55 55. Springer, Sep. 2006. 1.0	22
[4]	T. Hoefler and J. L. Traeff. Sparse collective operations for MPI. In Proceedings of	23
	the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'0 Workshop, May 2009. 7.6	24 25
[5]	O Krämer and H Mühlenhein. Manning strategies in message based multiprocessor	26
[5]	O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. <i>Parallel Computing</i> , 9:213–225, 1989. 7.1	27
	Systems. 1 aramet Compating, 5.216-226, 1565. 1.1	28
		29
		30
		31
		32 33
		34
		35
		36
		37
		38
		39
		40
		41
		42
		43 44
		44 45
		40
		47
		48

Index

CONST:DIMS, 23 EXAMPLES:Neighborhood collective commu-CONST:DIMS(i+1), 23nication, 41 CONST:dims[i], 23 EXAMPLES: Topologies, 41 CONST:DIRECTION = i, 23 EXAMPLES: Virtual topologies, 41 CONST: direction = i, 23MPI_CART_COORDS, 3, 17 CONST: false, 4, 6, 8, 10, 16, 20 MPI_CART_COORDS(comm, rank, maxdims, CONST:MPI_BOTTOM, 10, 12 coords), 17 CONST:MPI_CART, 14 MPI_CART_CREATE, 2-6, 16, 24, 26 CONST:MPI_COMM_NULL, 4, 6 MPI_CART_CREATE(comm, ndims, dims, CONST:MPI_COMM_WORLD, 5 periods, reorder, comm_cart), 25 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_GET, 3, 16 CONST:MPI_PROC_NULL, 22, 26 MPI_CART_GET(comm, maxdims, dims, pe-CONST:MPI_UNDEFINED, 14, 24, 25 riods, coords), 16 CONST:MPI_UNWEIGHTED, 9, 10, 12, 13, MPI_CART_MAP, 3, 25 20. 21 MPI_CART_MAP(comm, ndims, dims, pe-CONST:MPI_WEIGHTS_EMPTY, 9, 10, 12 riods, newrank), 24, 25 CONST:NULL, 10, 12 MPI_CART_RANK, 3, 17 CONST:true, 4, 6, 8, 10, 16, 20 MPI_CART_RANK(comm, coords, rank), 17 EXAMPLES: Cartesian virtual topologies, 41 MPI_CART_SHIFT, 3, 22, 23, 26 EXAMPLES:MPI_CART_COORDS, 23 MPI_CART_SHIFT(comm, direction, disp, EXAMPLES:MPI_CART_GET, 41 rank_source, rank_dest), 22 EXAMPLES:MPI_CART_RANK, 23 MPI_CART_SUB, 3, 24 EXAMPLES:MPI_CART_SHIFT, 23, 41 MPI_CART_SUB(comm, remain_dims, comm_new), EXAMPLES:MPI_CART_SUB, 24 24.25 EXAMPLES:MPI_DIMS_CREATE, 5, 41 MPI_CART_SUB(comm, remain_dims, new-EXAMPLES:MPI_DIST_GRAPH_CREATE, comm), 23 MPI_CARTDIM_GET, 3, 16 12EXAMPLES:MPI_Dist_graph_create, 13 MPI_CARTDIM_GET(comm, ndims), 16 EXAMPLES:MPI_DIST_GRAPH_CREATE_MDJACODNM_CREATE, 3 MPI_COMM_SPLIT, 3, 4, 6, 24 12EXAMPLES:MPI_GRAPH_CREATE, 6, 19 MPI_COMM_SPLIT(comm, color, key, comm_cart), EXAMPLES:MPI_GRAPH_NEIGHBORS, 19 25EXAMPLES:MPI_GRAPH_NEIGHBORS_COMPTCOMM_SPLIT(comm, color, key, comm_graph), 19 26EXAMPLES:MPI_SENDRECV_REPLACE, MPI_COMM_SPLIT(comm, color, key, comm_new), 2325MPI_DIMS_CREATE, 3-5

MPI_DIMS_CREATE(6, 2, dims), 5	MPI_GRAPH_NEIGHBORS_COUNT(comm,	1
MPI_DIMS_CREATE(6, 3, dims), 5	rank, nneighbors), <u>18</u>	2
MPI_DIMS_CREATE(7, 2, dims), 5	MPI_GRAPHDIMS_GET, 3, 15	3
MPI_DIMS_CREATE(7, 3, dims), 5	MPI_GRAPHDIMS_GET(comm, nnodes, nedges	54),
MPI_DIMS_CREATE(nnodes, ndims, dims),	<u>15</u>	5
5	MPI_INEIGHBOR_ALLGATHER, 3	6
MPI_DIST_GRAPH_CREATE, 2, 3, 8, 11,	MPI_INEIGHBOR_ALLGATHER(sendbuf, send	17
13, 21, 22, 26	count, sendtype, recvbuf, recvcount,	8
MPI_DIST_GRAPH_CREATE(comm_old, n,	recvtype, comm, request), $\underline{36}$	9
sources, degrees, destinations, weights	MPI_INEIGHBOR_ALLGATHERV, 3	10
info, reorder, comm_dist_graph), $\underline{10}$	MPI_INEIGHBOR_ALLGATHERV(sendbuf,	11
MPI_DIST_GRAPH_CREATE_ADJACENT	, sendcount, sendtype, recvbuf, recv-	12
2, 3, 8, 9, 13, 21, 26		13
MPI_DIST_GRAPH_CREATE_ADJACENT	(comm_odptiest), <u>37</u>	14
indegree, sources, sourceweights, outde	gwate J_INEIGHBOR_ALLTOALL, 3	15
destinations, destweights, info, reorder	MPI_INEIGHBOR_ALLTOALL(sendbuf, send-	16
$\operatorname{comm_dist_graph}), \underline{8}$	count, sendtype, recvbuf, recvcount,	17
MPI_DIST_GRAPH_NEIGHBOR_COUNT,	recvtype,comm, request), $\underline{38}$	18
22	MPI_INEIGHBOR_ALLTOALLV, 3	19
MPI_DIST_GRAPH_NEIGHBORS, 3, 20, 21,	MPI_INEIGHBOR_ALLTOALLV(sendbuf, sends	20
26	counts, sdispls, sendtype, recvbuf, recv-	21
MPI_DIST_GRAPH_NEIGHBORS(comm, m	axin- counts, rdispls, recvtype, comm, re-	22
degree, sources, sourceweights, max-	quest), $\underline{39}$	23
outdegree, destinations, destweights),	MPI_INEIGHBOR_ALLTOALLW, 4	24
<u>21</u>	MPI_INEIGHBOR_ALLTOALLW(sendbuf, send	2 -5
MPI_DIST_GRAPH_NEIGHBORS_COUNT	, counts, sdispls, sendtypes, recvbuf,	26
3, 20, 21	recvcounts, rdispls, recvtypes, comm,	27
MPI_DIST_GRAPH_NEIGHBORS_COUNT	$(\text{comm, request}), \underline{40}$	28
indegree, outdegree, weighted), $\underline{20}$	MPI_NEIGHBOR_ALLGATHER, 3, 29, 30,	29
MPI_GRAPH_CREATE, 2, 3, 6, 8, 12, 15,	36	30
18, 19, 25, 26	$\label{eq:MPI_NEIGHBOR_ALLGATHER} (sendbuf, sended) \\$	31
MPI_GRAPH_CREATE(comm, nnodes, in-	count, sendtype, recvbuf, recvcount,	32
dex, edges, reorder, comm_graph),	recvtype, comm), $\underline{27}$	33
25	MPI_NEIGHBOR_ALLGATHERV, 3, 37	34
MPI_GRAPH_CREATE(comm_old, nnodes,	$\label{eq:MPI_NEIGHBOR_ALLGATHERV} MPI_NEIGHBOR_ALLGATHERV(sendbuf, \qquad : \\$	35
index, edges, reorder, comm_graph),	sendcount, sendtype, recvbuf, recv-	36
<u>6</u>	counts, displs, recvtype, comm), $\underline{29}$	37
MPI_GRAPH_GET, 3, 15	MPI_NEIGHBOR_ALLTOALL, 3, 32, 38	38
$MPI_GRAPH_GET(comm, maxindex, maxed$	gesePI_NEIGHBOR_ALLTOALL(sendbuf, send-	39
index, edges), $\underline{15}$	count, sendtype, recvbuf, recvcount,	40
MPI_GRAPH_MAP, 3	recvtype,comm), $\underline{31}$	41
MPI_GRAPH_MAP(comm, nnodes, index,	MPI_NEIGHBOR_ALLTOALLV, 3, 39	42
edges, newrank), $\underline{25}$, 26	$MPI_NEIGHBOR_ALLTOALLV (sendbuf, send-display) and the send-display and the sender of the sender o$	43
MPI_GRAPH_NEIGHBORS, 3, 18, 19, 26	counts, sdispls, sendtype, recvbuf, recv-	44
$MPI_GRAPH_NEIGHBORS(comm, rank, matching) \\$	xneigh- counts, rdispls, recvtype, comm), $\underline{32}$	45
bors, neighbors), $\underline{18}$	MPI_NEIGHBOR_ALLTOALLW, 3, 33, 41	46
${\rm MPI_GRAPH_NEIGHBORS_COUNT, 3, 18},$	${\rm MPI_NEIGHBOR_ALLTOALLW} ({\rm sendbuf}, {\rm send-alltoallw}) ({\rm sendbuf}, {\rm sendbuf}, {\rm sendbuf}) ({\rm sendbuf})$	47
19	counts, sdispls, sendtypes, recvbuf,	48

1	recvcounts, rdispls, recvtypes, comm),
2	<u>34</u>
3	MPI_SENDRECV, 22
4	MPI_TOPO_TEST, 3, 14
5	$MPI_TOPO_TEST(comm, status), \underline{14}$
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	
44	
45	
46	
47	

 48