Parallel Programming using MPI
Communicators, Topologies & Derived Datatypes

Claudio Gheller, Andreas Jocksch
CSCS

cgheller@cscs.ch, jocksch@cscs.ch

Communicators & Topologies

Groups and Communicators

- A group is an ordered set of processes, each with a unique integer rank. In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a "handle". A group is always associated with a communicator object.

- A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. Like groups, communicators are accessible to the programmer only by "handles". The handle for the communicator that comprises all tasks is MPI_COMM_WORLD.

From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

Groups and Communicators (cont.ed)

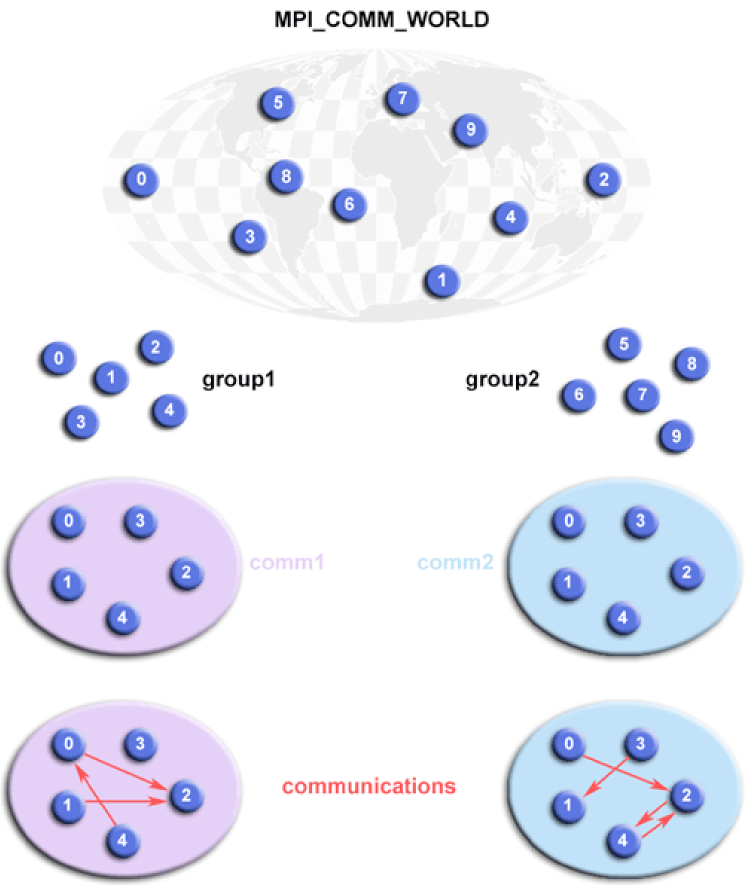
Goals:

- Allow you to organize tasks, based upon function, into task groups.
- Enable Collective Communications operations across a subset of related tasks.
- Provide basis for implementing user defined virtual topologies

Remarks:

Groups/communicators are dynamic - they can be created and destroyed during program execution.

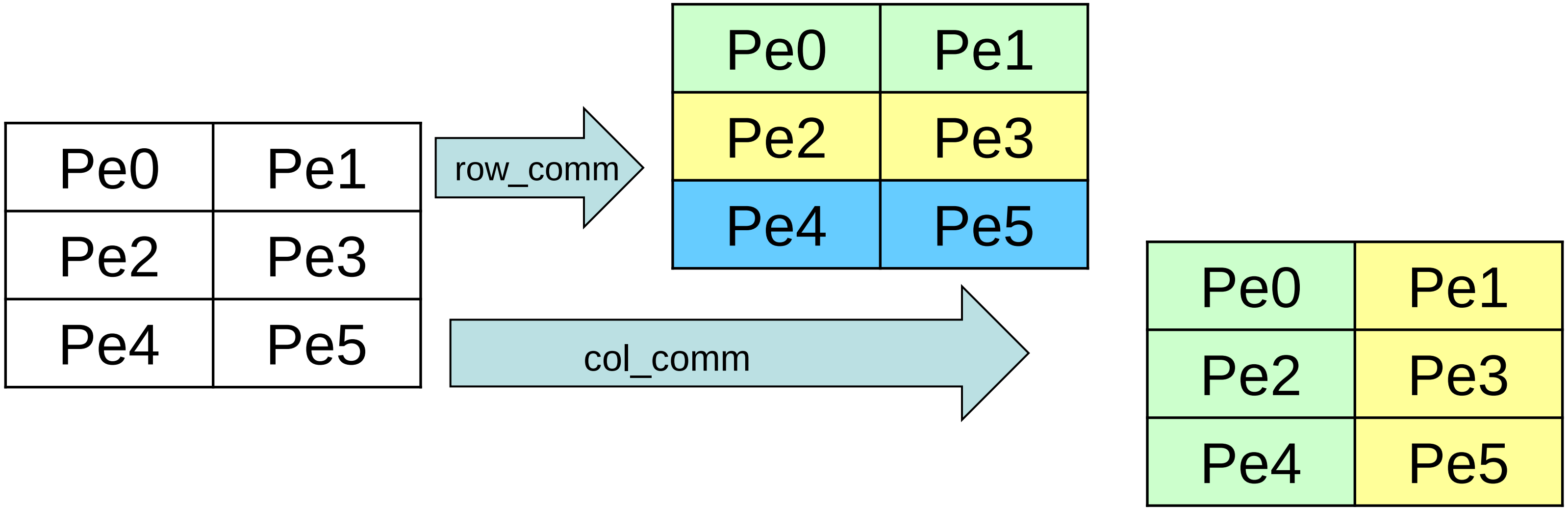
Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.



Defining the new communicator: the general (but convoluted…) approach

```
MPI_group MPI_GROUP_WORLD
MPI_group first_row_group
MPI_Comm first_row_comm
Integer row_size
Parameter(row_size=2)
Integer process_ranks(row_size)

Do i = 1, row_size
   process_ranks(i) = i-1
Enddo

Call MPI_COMM_GROUP(MPI_COMM_WORLD, MPI_GROUP_WORLD, ierr)

Call MPI_GROUP_INCL(MPI_GROUP_WORLD, row_size, process_ranks, first_row_group,
      ierr)

Call MPI_COMM_CREATE(MPI_COMM_WORLD, first_row_group, first_row_comm)
```

MPI_Comm_split: the smart solution



MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *comm_out)

The input variable color identifies the group while the key variable specifies a member of the group

```
! logical 2D topology with nrow=3 rows and mcol=2 columns. 6 processors
     irow = Iam/mcol        !! logical row number
     jcol = mod(Iam, mcol)   !! logical column number
     comm2D = MPI_COMM_WORLD

     call MPI_Comm_split(comm2D, irow, jcol, row_comm, ierr)
     call MPI_Comm_split(comm2D, jcol, irow, col_comm, ierr)
```

Iam	0	1	2	3	4	5
irow	0	0	1	1	2	2
jcol	0	1	0	1	0	1

Topologies

- A virtual topology describes the “connectivity” of MPI processes in a communicator
- The two main types of topologies supported by MPI are Cartesian and Graph.
- MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.
- Virtual topologies are built upon MPI communicators and groups.

Cartesian topology

- each process is “connected” to its neighbors in a virtual grid
- boundaries can be cyclic
- processes are identified by (discrete) Cartesian coordinates **i**; **j**; **k**;

Graph topologies

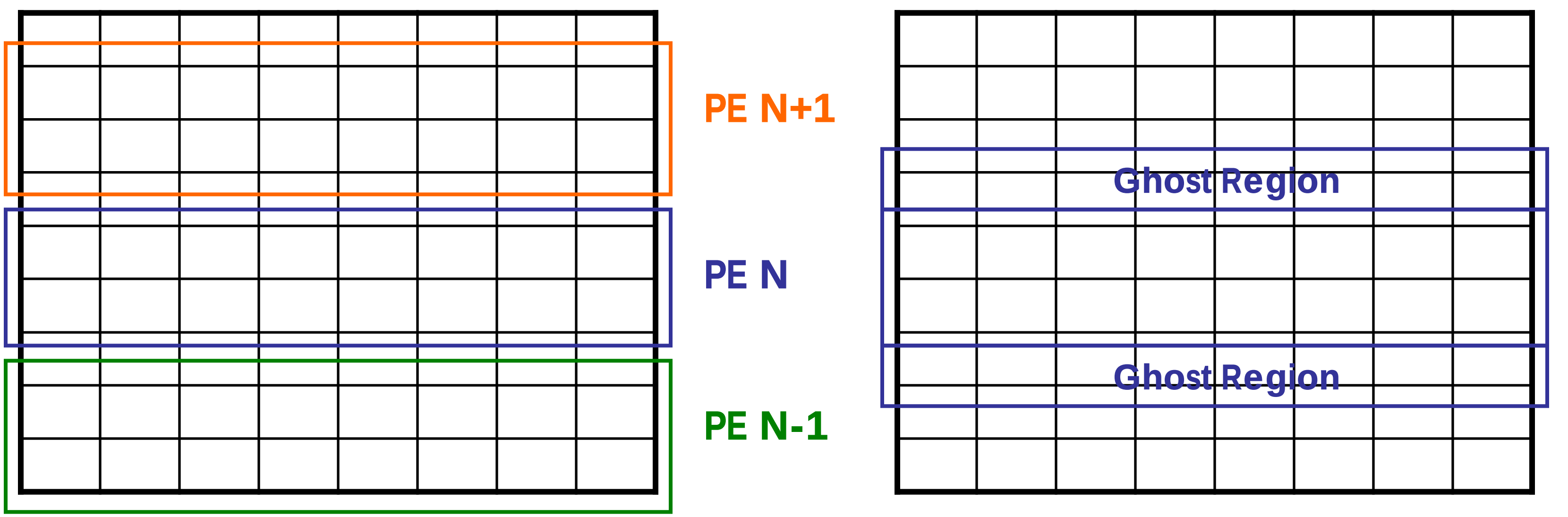
- graphs are used to describe communication patterns
- the most general description of communication patterns

Domain decomposition: planar distribution

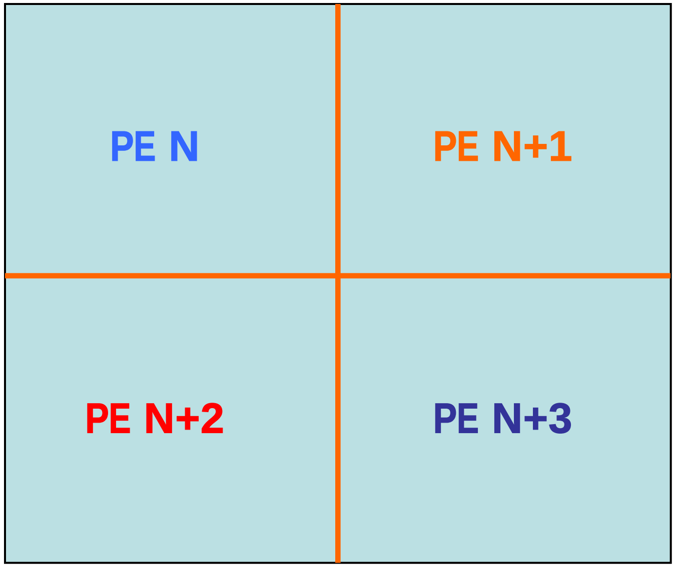
Data are distributed “linearly” between processors

Maps the MPI_COMM_WORLD linear topology

When ghost regions are exchanged, processor N communicates with N-1 and N+1



Domain decomposition: cartesian distribution



This is in general a more effective way of distribute the domain, since:

- It is much more scalable
- Communicated data volume can be smaller (especially when a large number of processors is used)
- It can better map the geometry of the problem and of the algorithm

However, it is more difficult to handle (e.g. who are my neighbors?)

MPI_CART_CREATE

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

[IN comm_old] input communicator (handle)

[IN ndims] number of dimensions of cartesian grid (integer)

[IN dims] integer array of size ndims specifying the number of processes in each dimension

[IN periods] logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension

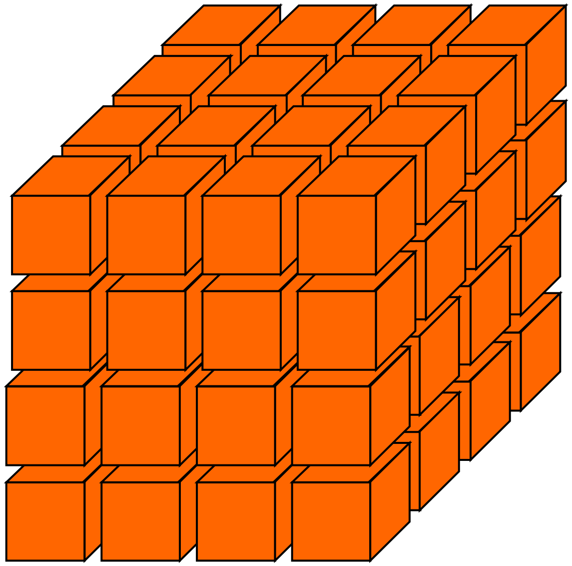
[IN reorder] ranking may be reordered (true) or not (false) (logical)

[OUT comm_cart] communicator with new cartesian topology (handle)

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

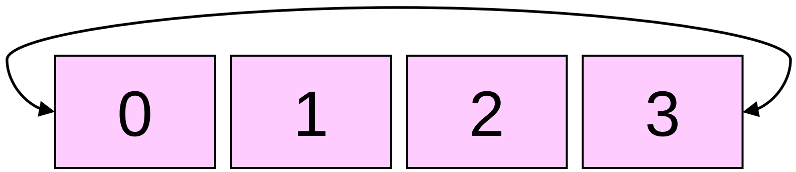
- Process coordinates begin with 0
- Row-major numbering

MPI_CART_CREATE example

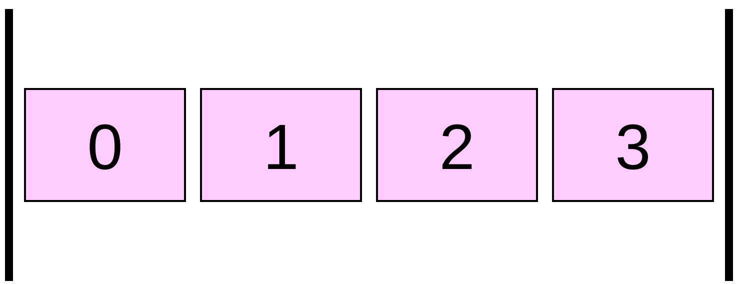


```
integer :: comm_cart
integer :: ierr
integer :: dims(3)
logical :: periods(3)

dims(1) = NprocX
dims(2) = NprocY
dims(3) = NprocZ
periods = .true.

call MPI_CART_CREATE (MPI_COMM_WORLD, 3, dims, periods, &
                      .true. , comm_cart,ierr)
```

Periodic boundaries

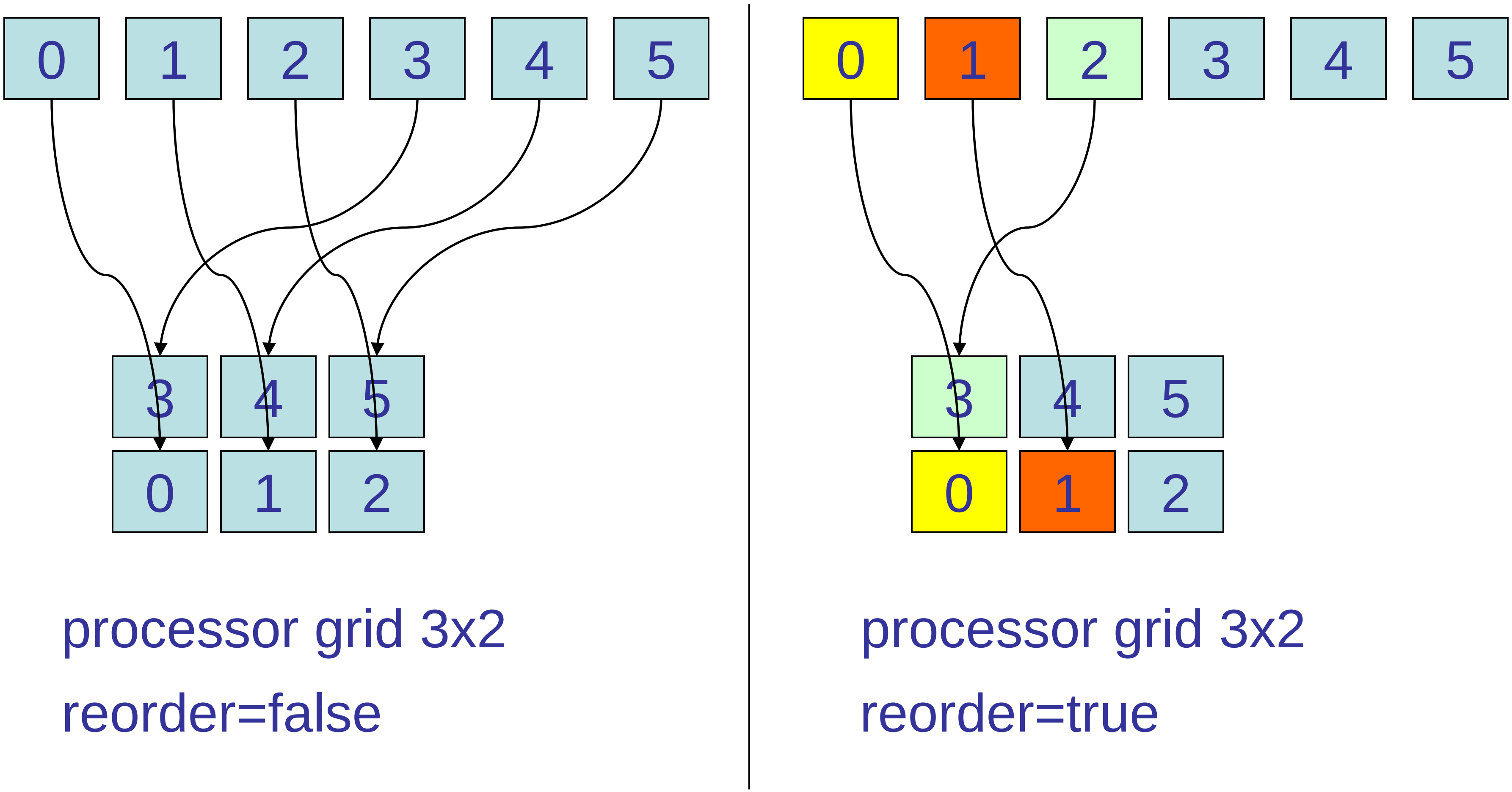


periods(1)=.true.
$\text{left}_0 = 3$
$\text{right}_3 = 0$



$\text{periods(1)=.false.}$
$\text{left}_0 = \text{MPI\_PROC\_NULL}$
$\text{right}_3 = \text{MPI\_PROC\_NULL}$

Reordering



0
1
2
3
4
5
3
4
5
0
1
2
processor grid 3x2
reorder=false
0
1
2
3
4
5
3
4
5
0
1
2
processor grid 3x2
reorder=true

A few functions

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

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

Coordinates to rank

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

Rank to coordinates

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

Finding neighbors:

MPI_CART_SHIFT(comm, direction, disp, rank_1, rank_2)

[IN comm] communicator with cartesian structure (handle)

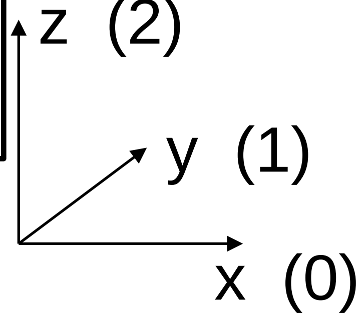
[IN direction] coordinate dimension of shift (integer)

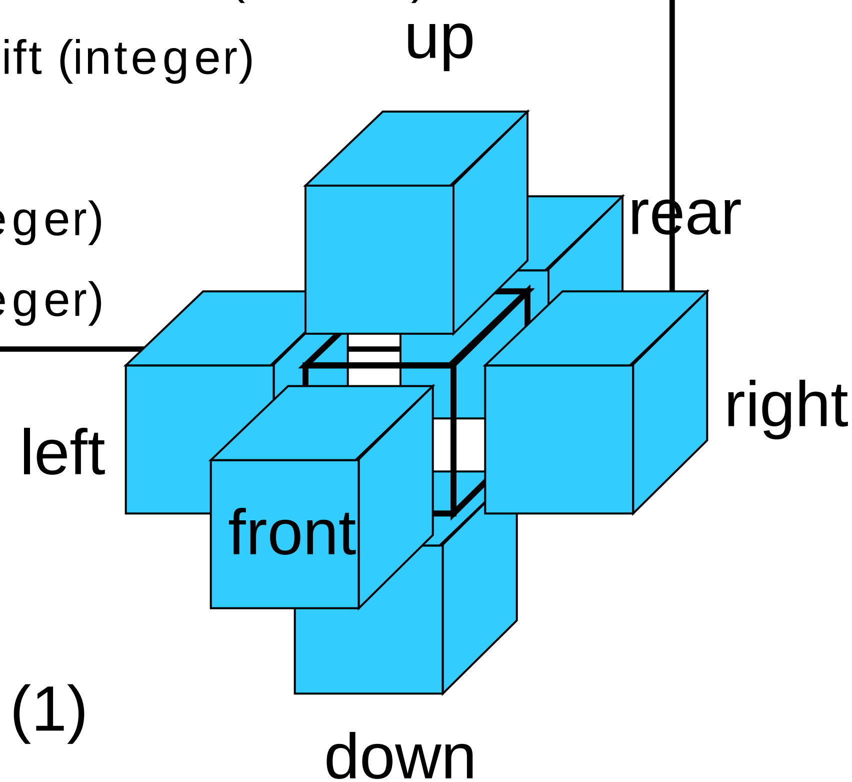
[IN disp] displacement (integer)

[OUT rank_1] rank of nearby process (integer)

[OUT rank_2] rank of nearby process (integer)

call MPI_CART_SHIFT(comm_cart,0, 1,**left,right**,ierr)
call MPI_CART_SHIFT(comm_cart,1, 1,**front,rear**,ierr)
call MPI_CART_SHIFT(comm_cart,2, 1,**down,up**,ierr)





Sub-grids in cartesian topology

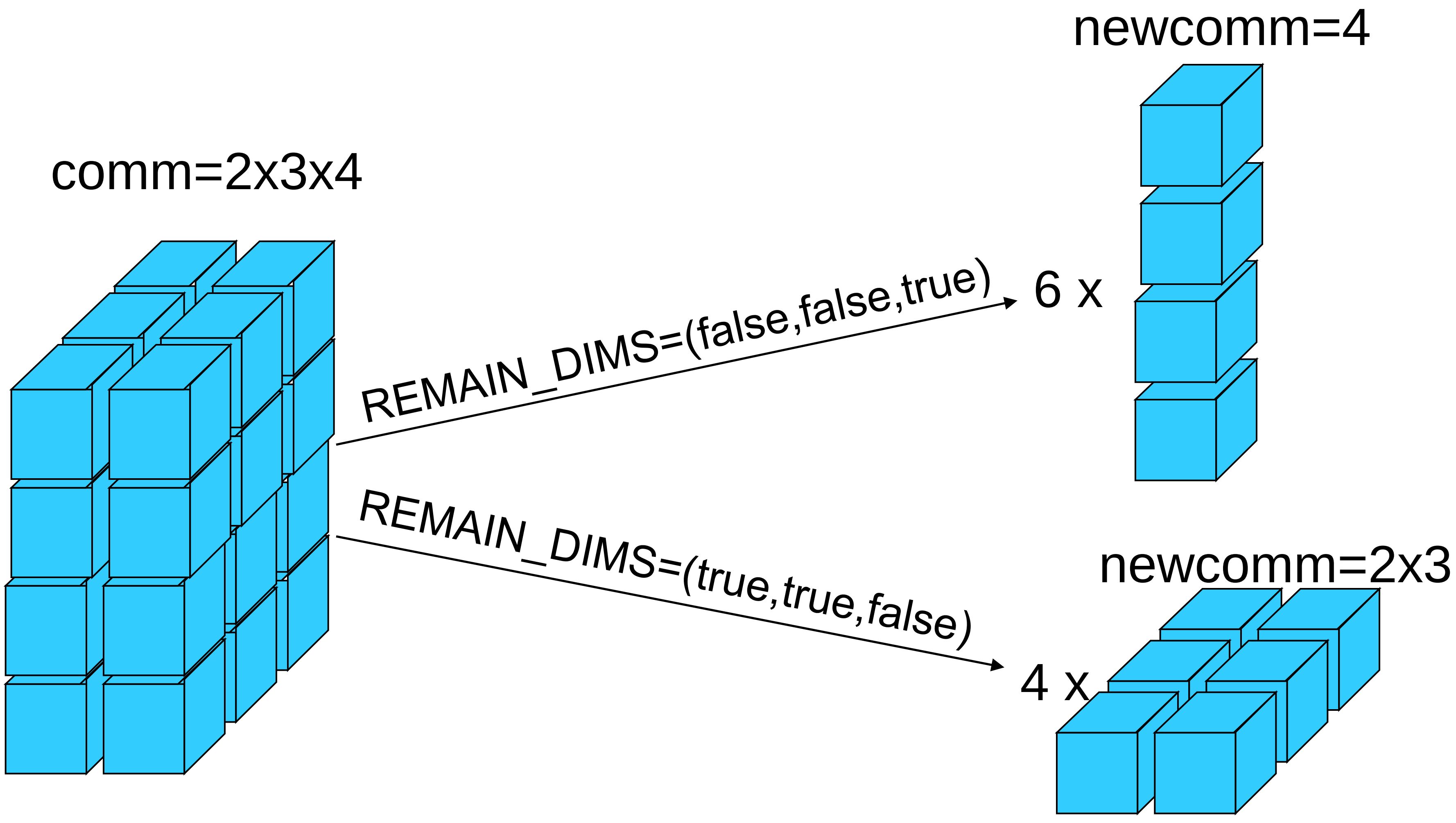
MPI_CART_SUB(comm, remain_dims, newcomm)

[IN comm] communicator with cartesian structure (handle)

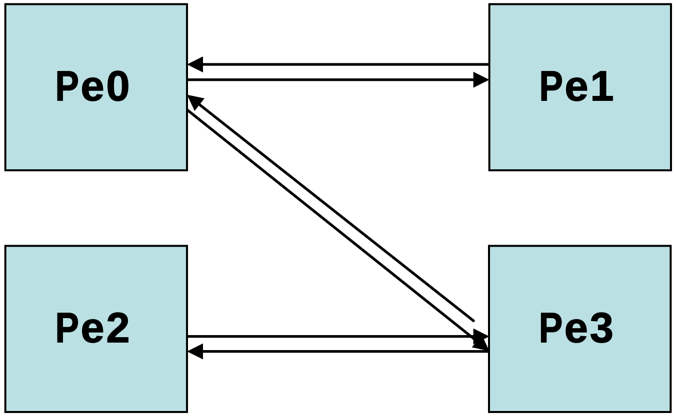
[IN remain_dims] the ith entry of remain_dims specifies whether the ith dimension is kept in the subgrid (true) or is dropped (false) (logical vector)

[OUT newcomm] communicator containing the subgrid that includes the calling process (handle)

Sub-grid examples



Graph topology



Process	Neighbours
0	1,3
1	0
2	3
3	0,2

nnodes = 4

index = 2, 3, 4, 6

edges = 1, 3, 0, 3, 0, 2

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

[IN comm_old] input communicator (handle)

[IN nnodes] number of nodes in graph (integer)

[IN index] array of integers describing node degrees (see below)

[IN edges] array of integers describing graph edges (see below)

[IN reorder] ranking may be reordered (true) or not (false) (logical)

[OUT comm_graph] communicator with graph topology added (handle)

Derived Datatypes

How to Use

MPI derived datatypes (differently from C or Fortran) are created (and destroyed) at **run-time** through calls to MPI library routines.

Implementation steps:

1. Construct the datatype.
2. Allocate the datatype.
3. Use the datatype.
4. Deallocate the datatype.

Construct the Datatype

MPI_Type_contiguous

Produces a new datatype by making count copies of an existing data type.

MPI_Type_vector

MPI_Type_hvector

Similar to contiguous, but allows for regular gaps (stride) in the displacements.

MPI_Type_hvector is identical to MPI _Type_vector except that stride is specified in bytes.

MPI_Type_indexed

MPI_Type_hindexed

An array of displacements of the input data type is provided as the map for the new data type. MPI_Type_hindexed is identical to MPI_Type_indexed except that offsets are specified in bytes.

MPI_Type_struct

The most general of all derived datatypes. The new data type is formed according to completely defined map of the component data types.

Allocate and destroy the Datatype

A constructed datatype must be committed to the system before it can be used in a communication.

* C

```
int MPI_Type_commit (MPI_datatype *datatype)
int MPI_Type_free (MPI_datatype *datatype)
```

* FORTRAN

```
MPI_TYPE_COMMIT (DATATYPE, MPIERROR)
MPI_TYPE_FREE (DATATYPE, MPIERROR)
INTEGER DATATYPE, MPIERROR
```

MPI_TYPE_CONTIGUOUS

C : MPI_Type_contiguous (count, oldtype, *newtype)

Fortran : MPI_TYPE_CONTIGUOUS (count, oldtype, newtype,ierr)

IN count Number of blocks to be added

IN oldtype Datatype of each element

OUT newtype Handle (pointer) for new derived type

OUT ierr reporting the success or failure

REMEMBER: BLOCK=contiguous elements of the same type

MPI_TYPE_CONTIGOUS constructs a typemap consisting of the **replication** of a **datatype** into contiguous locations. newtype is the datatype obtained by concatenating count copies of oldtype.

Example

```
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

9.0	10.0	11.0	12.0

1 element of rowtype

MPI_TYPE_VECTOR

C : MPI_Type_(h)vector (count, blocklength, stride, oldtype, *newtype)

Fortran : MPI_TYPE_(H)VECTOR (count, blocklength, stride, oldtype, newtype, ierr)

IN count: Number of blocks to be added

IN blocklen: Number of elements in block

IN stride: Number of elements (NOT bytes) between start of each block

IN oldtype: Datatype of each element

OUT newtype: Handle (pointer) for new derived type

The Vector constructor is similar to contiguous, but allows for **regular gaps or overlaps (stride)** in the displacements.

Example

count = 4; blocklength = 1; stride = 4;

MPI_Type_vector(count, blocklength, stride, MPI_FLOAT, &columntype);

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);

2.0	6.0	10.0	14.0

1 element of columntype

MPI_TYPE_INDEXED

```
MPI_Type_(h)indexed (int count, int *array_of_blocklengths,
                     int *array_of_displacements,
                     MPI_Datatype oldtype, MPI_datatype *newtype)
```

IN count: Number of blocks and number of elements of following arrays

IN array_of_blocklengths: number of instances of oldtype in each block

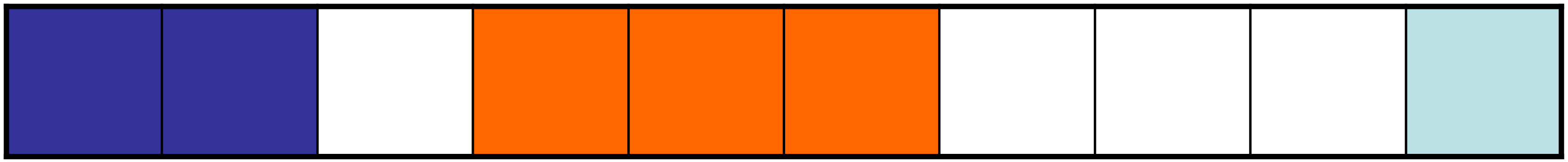
IN array_of_displacements: displacement of each block in units of extent (oldtype)

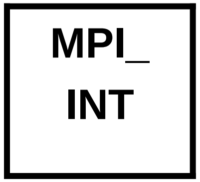
IN oldtype: Datatype of each element (MPI_Datatype)

OUT newtype: Handle (pointer) for new derived type (MPI_Datatype)

Returns a new datatype that represents count blocks. Each block is defined by an entry in array_of_blocklengths and array_of_displacements. Displacements are expressed in units of extent(oldtype).

Example





count = 3;

array_of_blocklengths[0] = 2;

array_of_blocklengths[1] = 3;

array_of_blocklengths[2] = 1;

array_of_displacements[0] = 0;

array_of_displacements[1] = 3;

array_of_displacements[2] = 9;

oldtype = MPI_INT;

MPI_TYPE_STRUCT

int MPI_Type_struct(count, blocklens, indices, old_types, newtype)

IN int count: number of blocks (integer) -- also number of entries in arrays array_of_types , array_of_displacements and array_of_blocklengths

IN int blocklens[ ]: number of elements in each block (array)

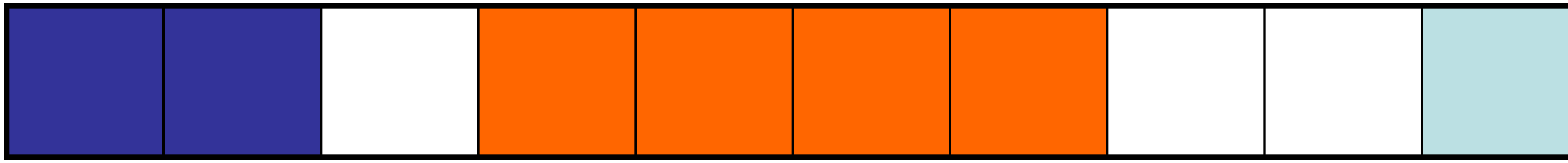
IN MPI_Aint indices[ ]:byte displacement of each block (array)

IN MPI_Datatype old_types[ ]: type of elements in each block (array of handles to datatype objects)

OUT MPI_Datatype *newtype (MPI_Datatype)

This subroutine returns a new datatype that represents count blocks. Each is defined by an entry in array_of_blocklengths, array_of_displacements and array_of_types. Displacements are expressed in bytes (since the type can change!!!)

Example



count = 3;

array_of_blocklengths[0] = 2;

array_of_blocklengths[1] = 2;

array_of_blocklengths[2] = 1;

array_of_displacements[0] = 0; (bytes)

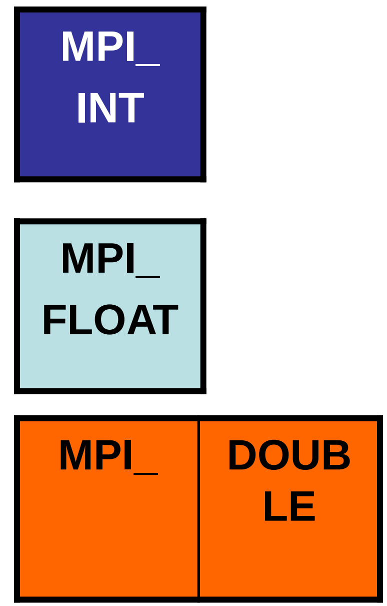
array_of_displacements[1] = 12; (bytes)

array_of_displacements[2] = 36; (bytes)

old_types[0] = MPI_INT;

old_types[1] = MPI_DOUBLE;

old_types[2] = MPI_FLOAT;



Subarrays

The subarray type constructor creates an MPI datatype describing an n-dimensional subarray of an n-dimensional array. The subarray may be placed anywhere within the full array

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

IN **ndims**: number of array dimensions (positive integer)

IN **array_of_sizes**: number of elements of type oldtype in each dimension of the full array (array of positive integers)

IN **array_of_subsizes**: number of elements of type oldtype in each dimension of the subarray (array of positive integers)

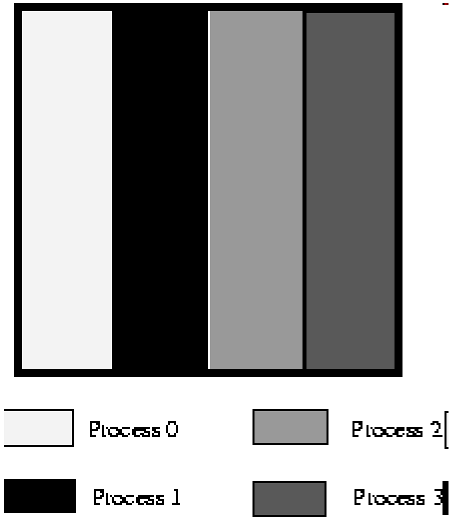
IN **array_of_starts**: starting coordinates of the subarray in each dimension (array of nonnegative integers)

IN **order**: array storage order flag (state)

IN **oldtype**: array element datatype (handle)

OUT **newtype**: new datatype (handle)

Subarrays example



A 100x100 2D array of double precision floating point numbers distributed among 4 processes such that each process has a block of 25 columns.

```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];
MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                         MPI_DOUBLE, &filetype);
```