24th Summer School on PARALLEL COMPUTING

MPI Derived Data Types

Massimiliano Guarrasi, Andrew Emerson

Isabella Baccarelli

SuperComputing Applications and Innovation Department

CINECA

Data Types

- MPI predefines (for portability reason) its primitive data types

C Data Types		Fortran Data Types
MPI_CHAR	MPI_C_COMPLEX	MPI_CHARACTER
MPI_WCHAR	MPI_C_FLOAT_COMPLEX	MPI_INTEGER
MPI_SHORT	MPI_C_DOUBLE_COMPLEX	MPI_INTEGER1
MPI_INT	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER2
MPI_LONG	MPI_C_BOOL	MPI_INTEGER4
MPI_LONG_LONG_INT	MPI_LOGICAL	MPI_REAL
MPI_LONG_LONG	MPI_C_LONG_DOUBLE_COMPLEX	MPI_REAL2
MPI_SIGNED_CHAR	MPI_INT8_T	MPI_REAL4
MPI_UNSIGNED_CHAR	MPI_INT16_T	MPI_REAL8
MPI_UNSIGNED_SHORT	MPI_INT32_T	MPI_DOUBLE_PRECISION
MPI_UNSIGNED_LONG	MPI_INT64_T	MPI_COMPLEX
MPI_UNSIGNED	MPI_UINT8_T	MPI_DOUBLE_COMPLEX
MPI_FLOAT	MPI_UINT16_T	MPI_LOGICAL
MPI_DOUBLE	MPI_UINT32_T	MPI_BYTE
MPI_LONG_DOUBLE	MPI_UINT64_T	MPI_PACKED
	MPI_BYTE	
	MPI_PACKED	

Derived Data Types

- MPI also provides facilities for you to define your own data structures based upon sequences of the MPI primitive data types. Such user defined structures are called derived data types.
- Primitive data types are contiguous. Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.
- MPI provides several methods for constructing derived data types:
  - Contiguous
  - Vector
  - Indexed
  - Struct

Derived Data Types

- What are they?
  - Data types built from the basic MPI datatypes. Formally, the MPI Standard defines a general datatype as an object that specifies two things:
    - a sequence of basic datatypes
    - a sequence of integer (byte) displacements
  - An easy way to represent such an object is as a sequence of pairs of basic datatypes and displacements. MPI calls this sequence a typemap.

typemap = {(type 0, displ 0), … (type n-1, displ n-1)}

  - But for most situations you do not need to worry about the typemap.

Derived Data Types

- Why use them?
  - Sometimes more convenient and efficient. For example, you may need to send messages that contain
    1. non-contiguous data of a single type (e.g. a sub-block of a matrix)
    2. contiguous data of mixed types (e.g., an integer count, followed by a sequence of real numbers)
    3. non-contiguous data of mixed types.
- As well as improving program readability and portability they may improve performance.

How to use

1. Construct the datatype using a template or *constructor*.
2. Allocate the datatype.
3. Use the datatype.
4. Deallocate the datatype.

You must construct and allocate a datatype before using it. You are not required to use it or deallocate it, but it is recommended (there may be a limit).

Datatype constructors

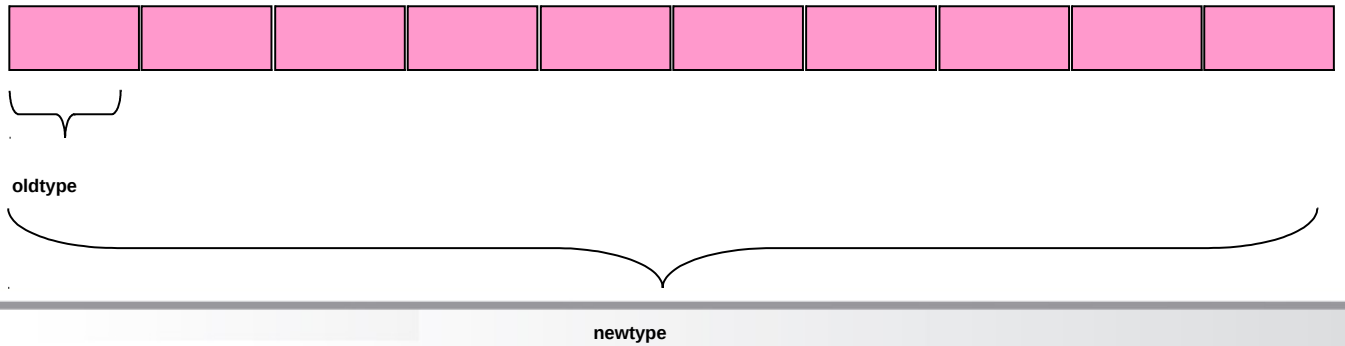
- MPI_Type_contiguous
  - Simplest constructor. Makes count copies of an existing datatype
- MPI_Type_vector, MPI_Type_hvector
  - Like contiguous, but allows for regular gaps (stride) in the displacements. For MPI_Type_hvector the stride is specified in bytes.
- MPI_Type_indexed, MPI_Type_hindexed
  - 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 byte
- 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

Datatype constructors (1): MPI_Type_contiguous

The simplest constructor. Produces a new data type by making count copies of an existing data type.

```
MPI_Type_contiguous (count,oldtype,&newtype)
MPI_TYPE_CONTIGUOUS (count,oldtype,newtype,ierr)

IN count: replication count (non-negative integer)
IN oldtype: old datatype (handle)
OUT newtype: new datatype (handle)
```

- MPI_TYPE_CONTIGUOUS 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 1 – A rowtype

Contiguous Derived datatype construction

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]

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(int argc, char *argv[])  {
int numtasks, rank;
   float a[SIZE][SIZE] =
     {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};

   MPI_Datatype rowtype;   // required variable

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

   // create contiguous derived data type
   MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
```

Allocating/deallocating datatypes

- 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)
```

Example 1 - A rowtype (C)

```
MPI_Datatype rowtype;   // required variable

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

   // create contiguous derived data type
   MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
   MPI_Type_commit(&rowtype);
```

Using datatypes

```
MPI_Type_contiguous(count, oldtype, &newtype);
MPI_Type_commit (&newtype);

MPI_Send(buffer, 1, newtype, dest, tag, comm);
```

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

Example 1 – A rowtype

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] =
  {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};
float b[SIZE];

MPI_Status stat;
MPI_Datatype rowtype;   // required variable

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

// create contiguous derived data type
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

if (numtasks == SIZE) {
   // task 0 sends one element of rowtype to all tasks
   if (rank == 0) {
      for (i=0; i<numtasks; i++)
        MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
      }

   // all tasks receive rowtype data from task 0
   MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
          rank,b[0],b[1],b[2],b[3]);
   }
else
   printf("Must specify %d processors. Terminating.\n",SIZE);

// free datatype when done using it
MPI_Type_free(&rowtype);
MPI_Finalize();
}
```

Example 1 – contiguous derived data type in Fortran: A columntype

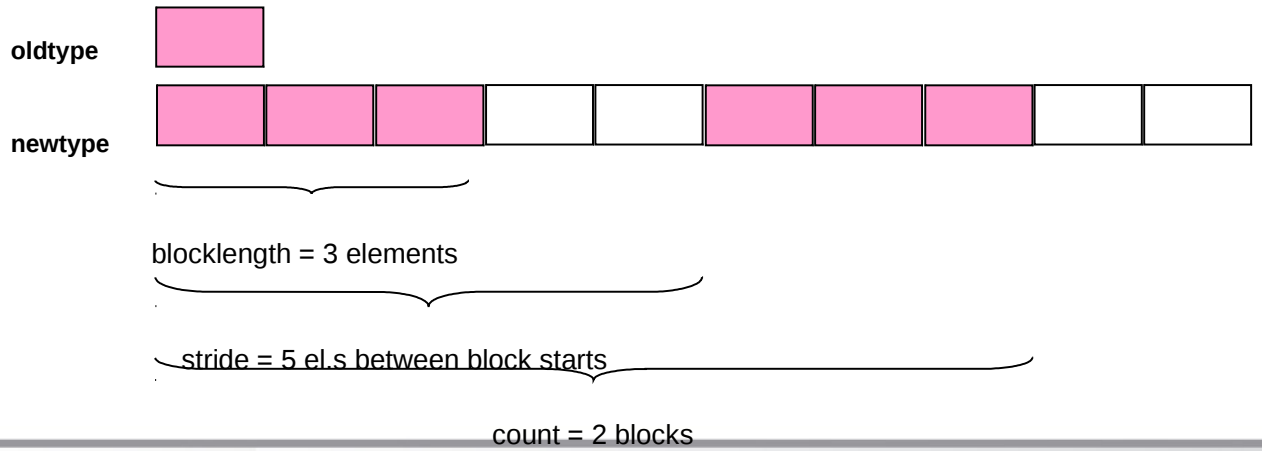
```
program contiguous
 include 'mpif.h'

 integer SIZE
 parameter(SIZE=4)
 integer numtasks, rank, source, dest, tag, i,  ierr
 real*4 a(0:SIZE-1,0:SIZE-1), b(0:SIZE-1)
 integer stat(MPI_STATUS_SIZE)
 integer columntype   ! required variable
 tag = 1

! Fortran stores this array in column major order
 data a  /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 /

 call MPI_INIT(ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

! create contiguous derived data type
 call MPI_TYPE_CONTIGUOUS(SIZE, MPI_REAL, columntype, ierr)
 call MPI_TYPE_COMMIT(columntype, ierr)

if (numtasks .eq. SIZE) then
! task 0 sends one element of columntype to all tasks
 if (rank .eq. 0) then
  do i=0, numtasks-1
   call MPI_SEND(a(0,i), 1, columntype, i,   &
                 tag,MPI_COMM_WORLD,ierr)
  end do
 endif

! all tasks receive columntype data from task 0
  source = 0
  call MPI_RECV(b, SIZE, MPI_REAL, source, tag, MPI_COMM_WORLD, &
                stat, ierr)
  print *, 'rank= ',rank,' b= ',b
else
  print *, 'Must specify',SIZE,' processors.  Terminating.'
endif

    ! free datatype when done using it
     call MPI_TYPE_FREE(columntype, ierr)
     call MPI_FINALIZE(ierr)

     end
```

DT constructors (2): MPI_Type_vector

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_vector (count,blocklength,stride, oldtype,&newtype)
MPI_TYPE_VECTOR (count,blocklength, stride,oldtype,newtype,ierr)

IN count: replication count (non-negative integer)
IN blocklen: Number of elements in each block (non-negative integer)
IN stride: Number of elements (NOT bytes) between start of each block (integer)
IN oldtype: old datatype (handle)
OUT newtype: new datatype (handle)
```

Consists of a number of elements of the same datatype repeated with a certain stride



Allocating/deallocating and using datatypes (again)

Allocate and deallocate

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

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

- C

```
MPI_Type_vector(count, blocklength, stride, oldtype, &newtype);
MPI_Type_commit (&newtype);
MPI_Send(buffer, 1, newtype, dest, tag, comm);
```

Example 2 - columntype

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

C: Vector derived data type example

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] =
     {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};
float b[SIZE];

MPI_Status stat;
MPI_Datatype columntype;   // required variable

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

// create vector derived data type
 MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
 MPI_Type_commit(&columntype);
```

```
if (numtasks == SIZE) {
// task 0 sends one element of columntype to all tasks
  if (rank == 0) {
   for (i=0; i<numtasks; i++)
    MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
   }

// all tasks receive columntype data from task 0
 MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
 printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
        rank,b[0],b[1],b[2],b[3]);
 }
 else
   printf("Must specify %d processors. Terminating.\n",SIZE);

 // free datatype when done using it
 MPI_Type_free(&columntype);
 MPI_Finalize();
 }
```

Fortran: Vector derived data type example

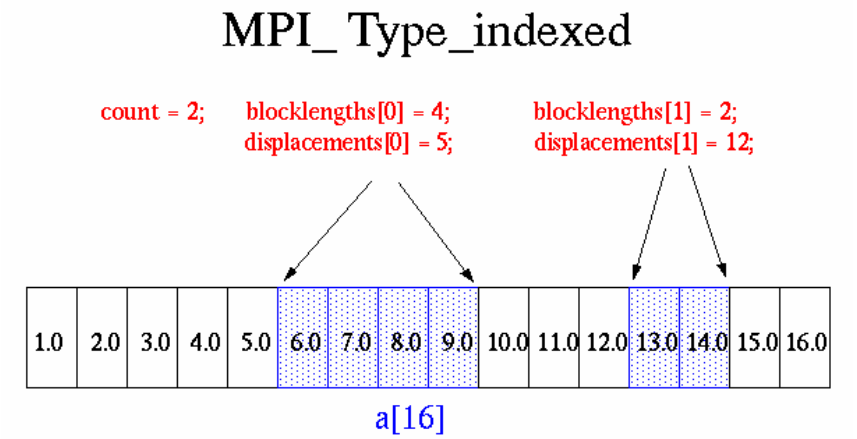
```
program vector
include 'mpif.h'

integer SIZE
parameter(SIZE=4)
integer numtasks, rank, source, dest, tag, i,  ierr
real*4 a(0:SIZE-1,0:SIZE-1), b(0:SIZE-1)
integer stat(MPI_STATUS_SIZE)
integer rowtype   ! required variable
tag = 1

! Fortran stores this array in column major order
data a  /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 /

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

! create vector derived data type
call MPI_TYPE_VECTOR(SIZE, 1, SIZE, MPI_REAL, rowtype, ierr)
call MPI_TYPE_COMMIT(rowtype, ierr)
```

```
if (numtasks .eq. SIZE) then
! task 0 sends one element of rowtype to all tasks
  if (rank .eq. 0) then
   do i=0, numtasks-1
   call MPI_SEND(a(i,0), 1, rowtype, i, tag, MPI_COMM_WORLD, ierr)
   end do
  endif

!all tasks receive rowtype data from task 0
  source = 0
  call MPI_RECV(b, SIZE, MPI_REAL, source, tag, MPI_COMM_WORLD, &
                stat, ierr)
  print *, 'rank= ',rank,' b= ',b
else
  print *, 'Must specify',SIZE,' processors.  Terminating.'
endif

! free datatype when done using it
  call MPI_TYPE_FREE(rowtype, ierr)
  call MPI_FINALIZE(ierr)

end
```

Datatype constructors

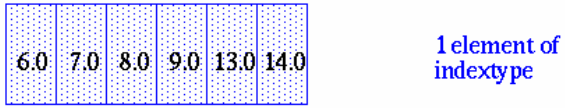
- MPI_Type_contiguous
  - Simplest constructor. Makes count copies of an existing datatype
- MPI_Type_vector, MPI_Type_hvector
  - Like contiguous, but allows for regular gaps (stride) in the displacements. For MPI_Type_hvector the stride is specified in bytes.
- MPI_Type_indexed, MPI_Type_hindexed
  - 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 byte
- 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

DT constructors (3):



MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);

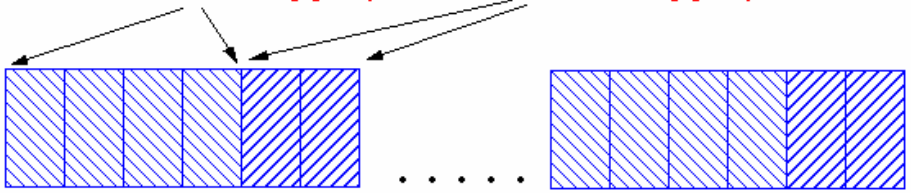
MPI_Send(&a, 1, indextype, dest, tag, comm);



DT constructors (4)

MPI_ Type_struct

```
typedef struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2; oldtypes[0] = MPI_FLOAT;   oldtypes[1] = MPI_INT
           offsets[0] = 0;            offsets[1] = 4 * extent;
           blockcounts[0] = 4;        blockcounts[1] = 2;
```



particles[NELEM]

```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);

MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

Other tools

- MPI_GET_COUNT,MPI_GET_ELEMENTS
  - Routines which return the number of "copies" of type datatype and the number of basic elements (often used after a MPI_RECV).

```
int MPI_Get_count( const MPI_Status *status, MPI_Datatype datatype, int *count )
int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,  int
    *count)
```

- MPI_TYPE_GET_EXTENT (*Advanced*)
  - Returns the lower bound and extent of a datatype (i.e. upper bound + padding to align the datatype). Useful for creating new datatypes with **`MPI_TYPE_CREATE_RESIZED,`** `for example.`

Derived Datatype Summary

- Provide a portable and elegant way of communicating non-contiguous or mixed types in a message.
- By optimising how data is stored, should improve efficiency during MPI send and receive (perhaps avoiding buffering).
- Derived datatypes are built from basic MPI datatypes, according to a template. Can be used for many variables of the same form.
- Remember to commit the datatypes before using them.

24th Summer School on PARALLEL COMPUTING

MPI Virtual Topologies

Massimiliano Guarrasi, Andrew Emerson, Giusy Muscianisi, Luca Ferraro

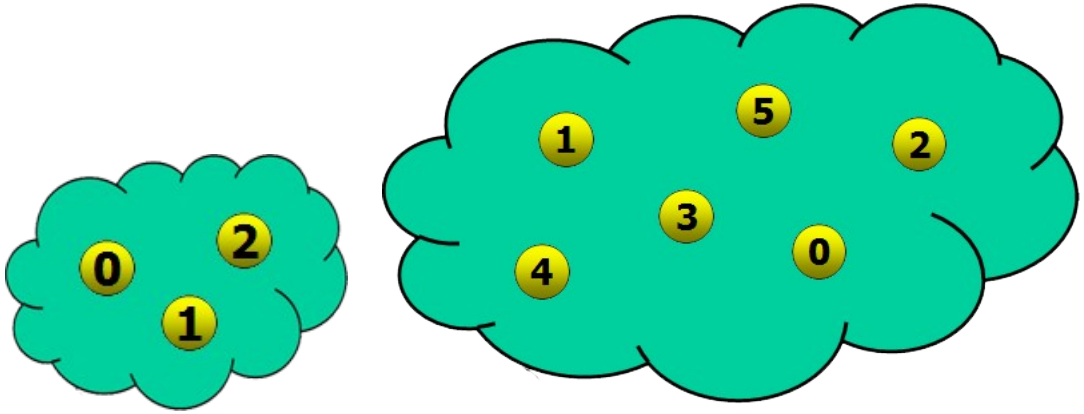
Isabella Baccarelli

SuperComputing Applications and Innovation Department

CINECA

Beyond MPI_Comm_World: Communicators

- A communicator defines the communication world of a group of processes



MPI_Comm_World

- In addition to **MPI_Comm_World**, in MPI programs other communicators can be defined for particular needs, such as:
  - Using collective functions only inside a sub-set of the default communicator (MPI_COMM_WORLD) processes (i.e., creating separate communication spaces)
  - **Using an identification pattern of processes more convenient for a specific communication pattern (virtual topologies)**

Communicator's components

- ***GROUP* of processes** : an *ordered* set of processes
  - The group is used to identify the processes which will be able to communicate the one with the other (or in a collective way)
  - Each process belonging to a group is assigned a ordered index (*rank*), used to univocally identify that process

- ***Context***: a system-defined object used to uniquely identify the communicator.
  - Two different communicators have two different contexts, even though they have the same underlying group of processes
  - Used by the communicator to deal with the send/receive of message
  - e.g. it contains the information on the status of a message to be received sent with a `MPI_Isend`

- ***Attributes***: additional (optional) characterization of the communicator:
  - **The communicator topology**
  - The *rank* of the process in charge of executing I/O operations

Virtual topologies of processes

- Definition of a new identifying addressing scheme of processes more *convenient* to a communication pattern:
  - It makes writing a program easier
  - It may help MPI communication optimization
- Creating a virtual topology of processes in MPI requires the definition of a new comunicator with specific *attributes*

Virtual topologies: Outline

- Virtual topology: definition
- MPI supported topologies:
  - Cartesian
    - How to create cartesian communicators
    - Cartesian mapping function
    - Cartesian partitioning
  - (Graph)

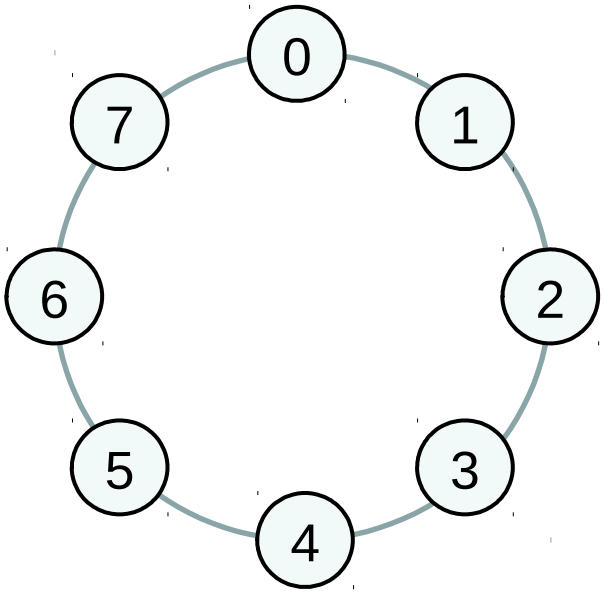
Virtual Topology

- **Topology**:
  - extra, optional attribute that can be given to an intra-communicator; topologies cannot be added to inter-communicators.
  - 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.
- **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).

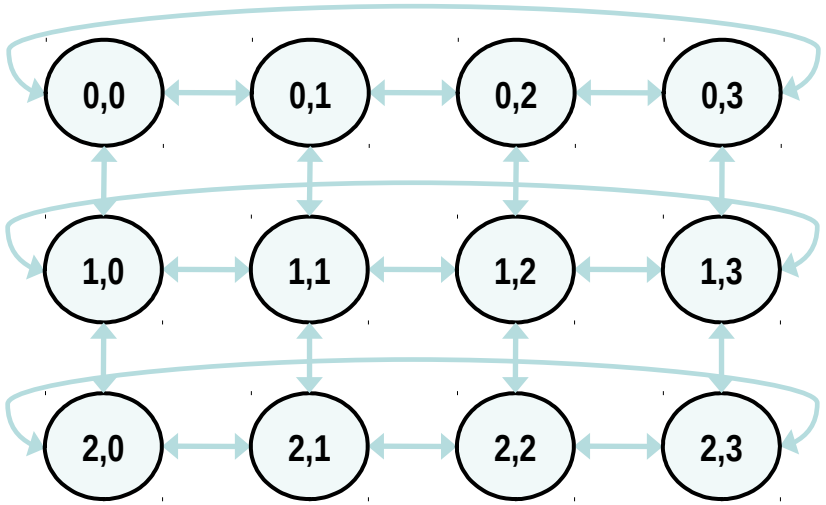
Virtual Topology

- **Virtual topology:**
  - logical process arrangement in topological patterns such as 2D or 3D grid; more generally, the logical process arrangement is described by a graph.
- **Virtual process topology .vs. topology of the underlying, physical hardware:**
  - 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.
  - the description of the virtual topology depends only on the application, and is machine-independent.

Virtual Topology – Examples



RING

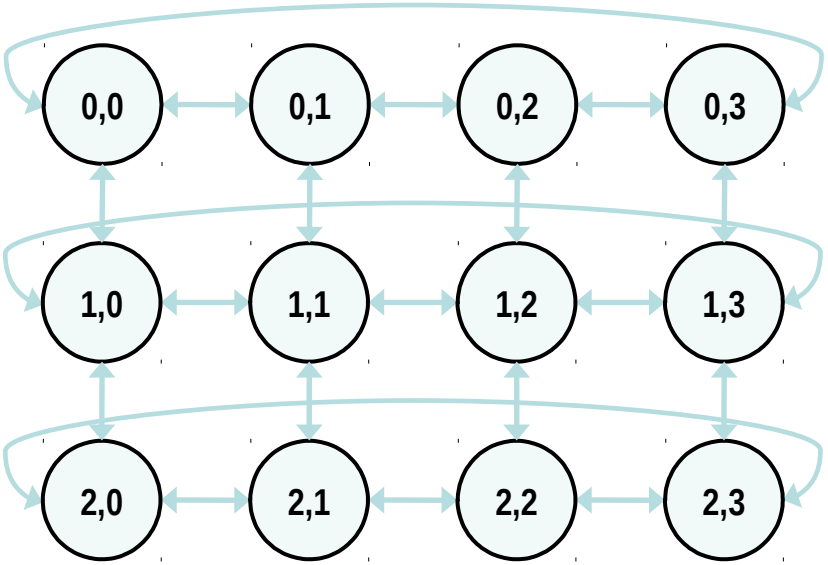


2D-GRID WITH PERIODIC BOUNDARY CONDITIONS

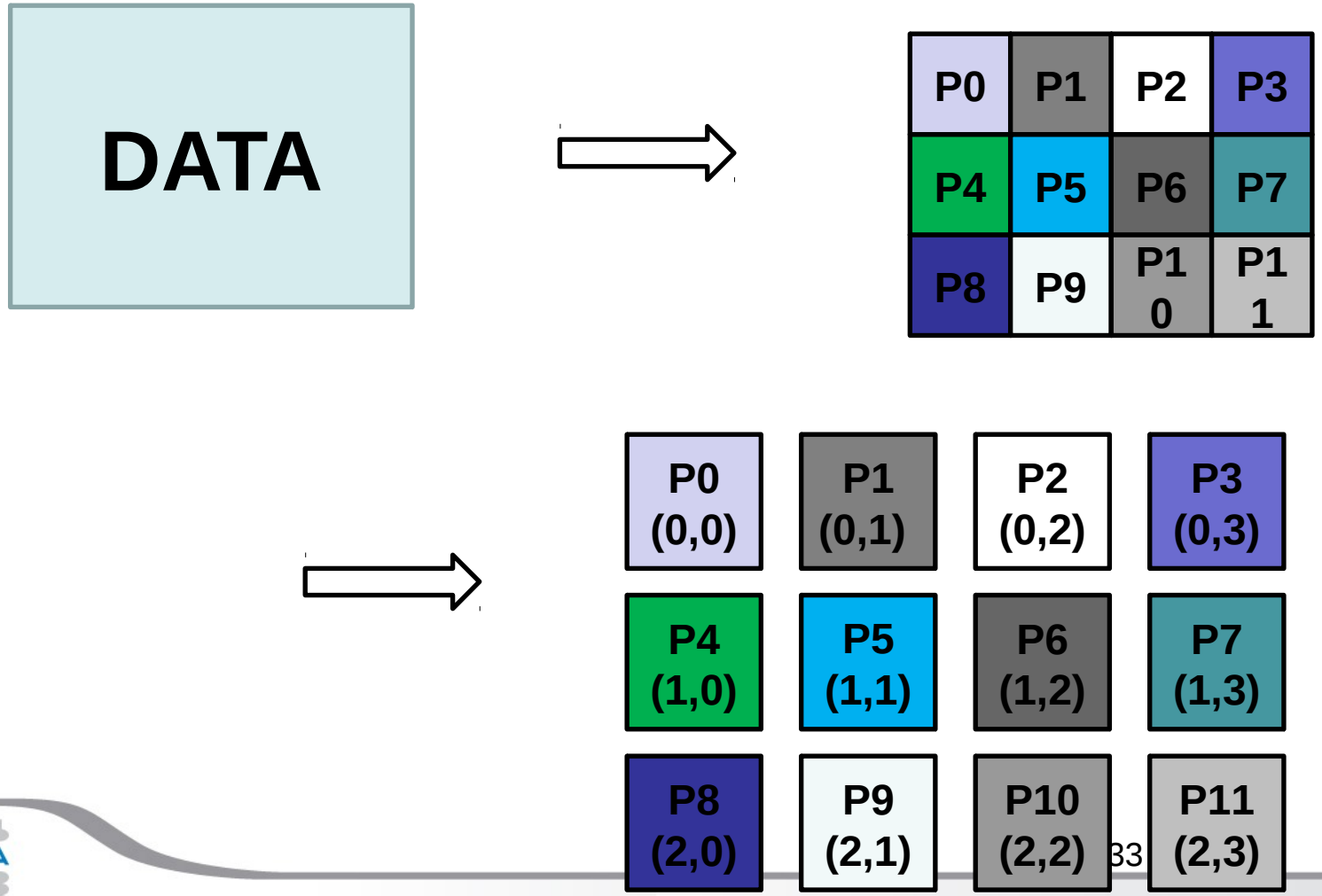
Cartesian Topology

A grid of processes is easily described with a cartesian topology:

- each process can be identified by cartesian coordinates
- periodicity can be selected for each direction
- communications are performed along grid dimensions only



Example: 2D Domain decomposition



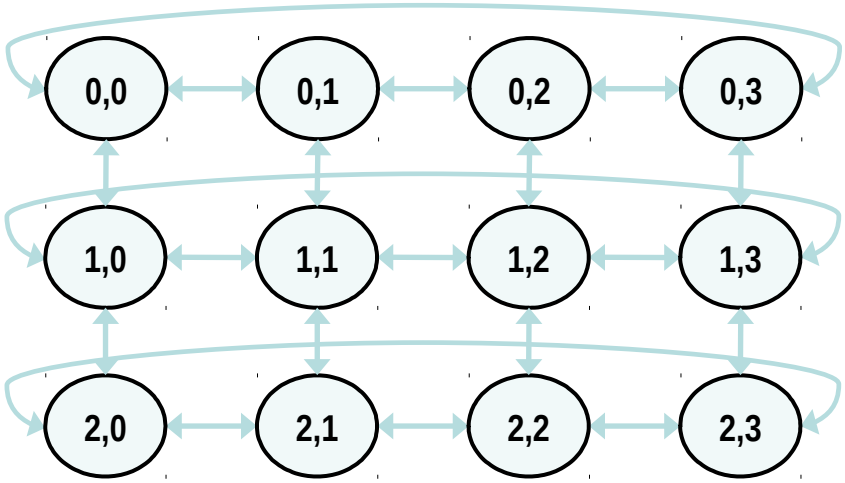
Cartesian Constructor

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

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)
    OUT comm_cart: communicator with new Cartesian topology (handle)
```

- Returns a handle to a new communicator to which the Cartesian topology information is attached.
- Reorder:
  - False: the rank of each process in the new group is identical to its rank in the old group.
  - True: the processes may be reordered, possibly so as to choose a good embedding of the virtual topology onto physical machine (well, is that actually implemented?)

How to create a Cartesian Topology

```
#include <mpi.h>

int main(int argc, char *argv[])
{

  MPI_Comm cart_comm;
  int dim[] = {3, 4};
  int period[] = {0, 1};
  int reorder = 0;

  MPI_Init(&argc, &argv);

  MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder,
   &cart_comm);
  ...
}
```



Cartesian Topology Utilities

- MPI_Dims_Create:
  - compute optimal balanced distribution of processes per coordinate direction with respect to:
    - a given dimensionality
    - the number of processes in a group
    - optional constraints
- MPI_Cart_coords:
  - given a rank, returns process's coordinates
- MPI_Cart_rank:
  - given process's coordinates, returns the rank
- MPI_Cart_shift:
  - get source and destination rank ids in SendRecv operations

Binding of MPI_Dims_create

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

MPI_DIMS_CREATE(nnodes, ndims, dims)

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

- Help user to 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)
- if `dims[i]` is set to a positive number, the routine will not modify the number of nodes in that i dimension
- negative value of `dims[i]` are erroneous

IN / OUT of “dims”

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

MPI_DIMS_CREATE(nnodes, ndims, dims)

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

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

Using MPI_Dims_create

```
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

int dim[3];
dim[0] = 0; // let MPI arrange
dim[1] = 0; // let MPI arrange
dim[2] = 3; // I want exactly 3 planes

MPI_Dims_create(nprocs, 3, dim);

if (dim[0]*dim[1]*dim[2] < nprocs) {
  fprintf(stderr, "WARNING: some processes are not in use!\n"
}

int period[] = {1, 1, 0};
int reorder = 0;

MPI_Cart_create(MPI_COMM_WORLD, 3, dim, period, reorder, &cube_comm);

  ...
```

Coordinate -> Rank: `MPI_Cart_rank`

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

MPI_CART_RANK(comm, coords, rank)

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

- translation of the logical process coordinates to process ranks as they are used by the point-to-point routines
- if `dimension i` is periodic, when i-th coordinate is out of range, it is shifted back to the interval `0<coords(i)<dims(i)` automatically
- out-of-range coordinates are erroneous for non-periodic dimensions

Mapping: old and new ranks

```
// buffer to collect MPI_COMM_WORLD rank ids in new cartesian rank sorting
int *world_ranks = (int *) malloc (nprocs*sizeof(int));

int oldrank;
MPI_Comm_rank(MPI_COMM_WORLD, &oldrank);

MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, 1, &cart_comm);

// indexing sorting is now performed on rank id of cart_comm communicator
MPI_Gather(&oldrank, 1, MPI_INT, world_ranks, 1, MPI_INT, 0, cart_comm);

if (oldrank == 0) {
  for (int i=0; i<dim[0]; i++) {
    for (int j=0; j<dim[1]; j++) {
     int new_rank;
     int coords[2]; coords[0]=i; coords[1]=j;
     MPI_Cart_rank(cart_comm, coords, &new_rank);

     printf("([%d, %d]) ", new_rank, world_ranks[new_rank]);
    }; printf("\n");
  }
}
```

Rank -> Coordinate: MPI_Cart_coords

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

MPI_CART_COORDS(comm, rank, maxdims, coords)

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

- For each MPI process in Cartesian communicator, the coordinate within the cartesian topology are returned

Usage of MPI_Cart_coords

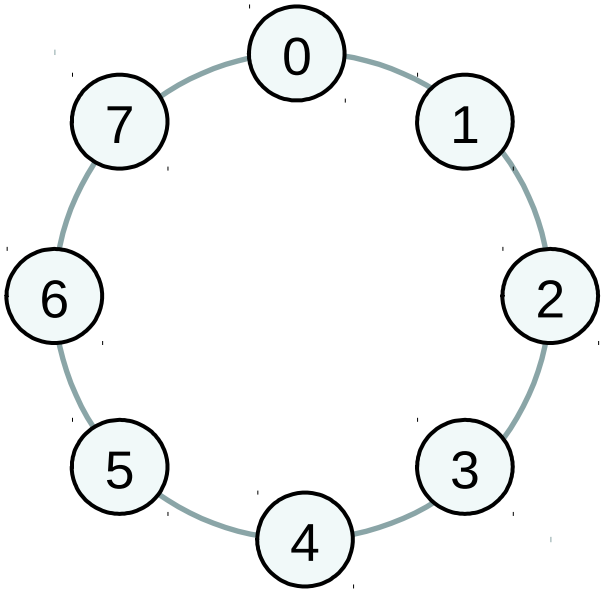
```
. . .
ndim = (int*)calloc(dim,sizeof(int));
ndim[0] = row; ndim[1] = col;

period = (int*)calloc(dim,sizeof(int));
period[0] = period[1] = 0;

reorder = 0;

// 2D grid creation
MPI_Cart_Create(MPI_COMM_WORLD,dim,ndim,period,reorder, &comm_grid);
MPI_Comm_rank(comm_grid,&menum_grid);

// Coordinate of each mpi rank within the cartesian communicator
MPI_Cart_coords(comm_grid,menum,dim,coordinate);

printf("Procs %d coordinates in 2D grid (%d,%d)
   \n",menum,*coordinate,*(coordinate+1));
. . .
}
```

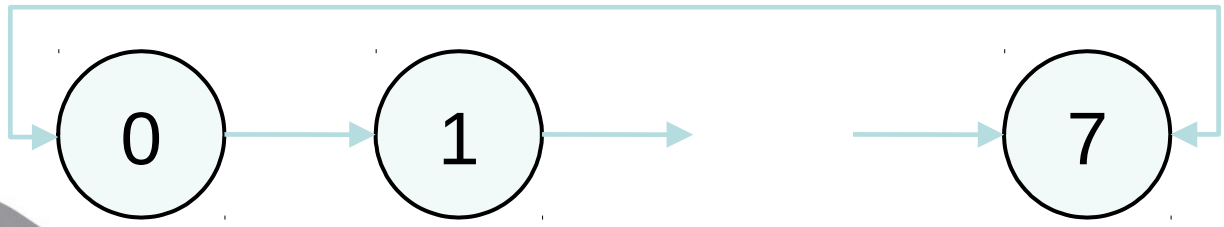
Circular Shift: a 1D Cartesian Topology

Circular shift is another typical MPI communication pattern:

- each process communicates only with its neighbours along one direction
- periodic boundary conditions can be set for letting first and last processes partecipate in the communication

such a pattern is nothing more than a 1D cartesian grid topology with optional periodicity





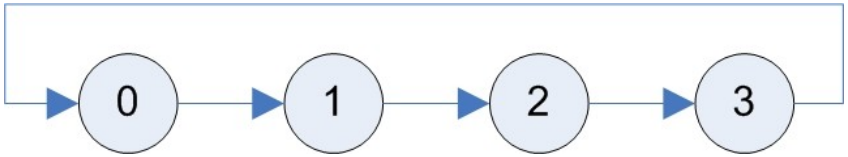
Sendrecv with Cartesian Topologies: MPI_Cart_shift

```
int MPI_Cart_shift( MPI_Comm comm, int direction, int displ, int *source,
int *dest );

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

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

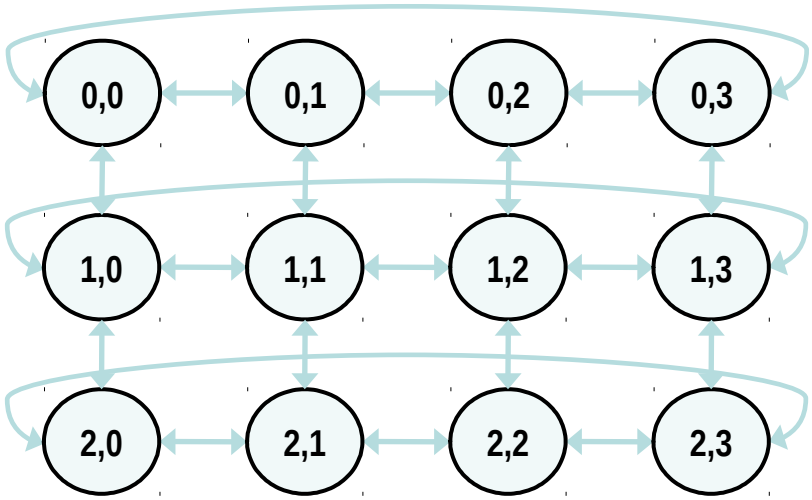
- Depending on the periodicity of the Cartesian group in the specified coordinate direction, MPI_CART_SHIFT provides the identifiers for a circular or an end-o shift.
- In the case of an end-o shift, the value **MPI_PROC_NULL** may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.
- provides the calling process the ranks of source and destination processes for an MPI_SENDRECV with respect to a specified coordinate direction and step size of the shift

Sendrecv with 1D Cartesian Topologies

```
...
int dim[1],period[1];
dim[0] = nprocs;
period[0] = 1;
MPI_Comm ring_comm;

MPI_Cart_create(MPI_COMM_WORLD, 1, dim, period, 0, &ring_comm);

int source, dest;
MPI_Cart_shift(ring_comm, 0, 1, &source, &dest);

MPI_Sendrecv(right_bounday, n, MPI_INT, dest, rtag,
             left_boundary, n, MPI_INT, source, ltag,
             ring_comm, &status);
...
```



Sendrecv with 2D Cartesian Topologies




```
...

int dim[] = {4, 3};
int period[] = {1, 0};
MPI_Comm grid_comm;

MPI_Cart_create(MPI_COMM_WORLD, 2,
   dim, period, 0, &grid_comm);

int source, dest;
for (int dimension = 0; dimension < 2; dimension++) {
  for (int versus = -1; versus < 2; versus+=2;) {
    MPI_Cart_shift(ring_comm, dimension, versus, &source, &dest);
    MPI_Sendrecv(buffer, n, MPI_INT, source, stag,
             buffer, n, MPI_INT, dest, dtag,
             grid_comm, &status);
  }
}
```

News from MPI-3.x

MPI-3.0 introduces more functionalities for topologies:

- neighbor collective communications
  - enables optimizations in the MPI library because the communication pattern is known statically
  - the implementation can compute optimized message schedules during creation of the topology

  MPI_NEIGHBOR_ALL(GATHER[V] | TOALL[V])
- non-blocking collective communications:
  - semantic similar to non-blocking point-to-point

  MPI_INEIGHBOR_ALL(GATHER[V] | TOALL[V])

Lab session 4

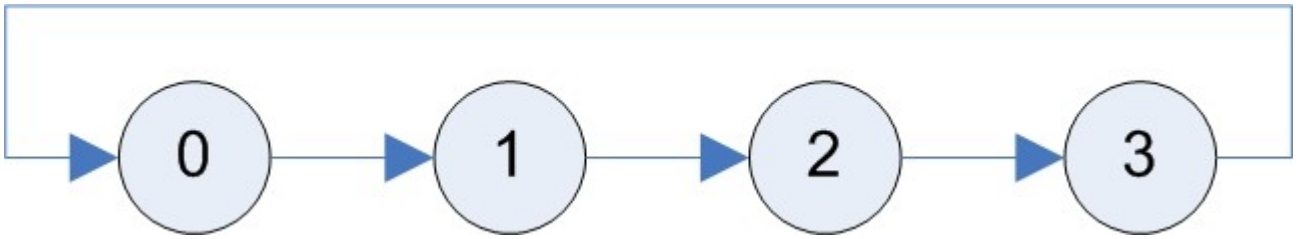
Virtual topologies

- *Circular shift* with 1D cartesian topology (Exercise 15)

- Arithmetic average on first neighbours in a 2D cartesian topology (Exercise 16)

Circular shift with 1D cartesian topology

Exercise 15



- Modify the code of the Circular Shift with MPI_Sendrecv (Exercise 7), using MPI functions for *virtual topologies* to determine the arguments of the MPI_Sendrecv function
- Note: use MPI_Cart_shift to determine the sender/receiver processes for MPI_Sendrecv

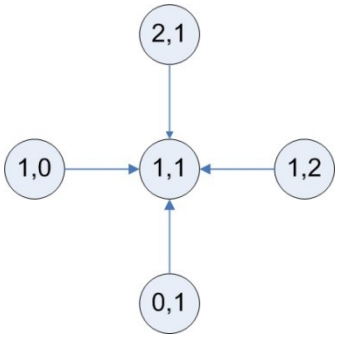
Arithmetic average on first neighbours in 2D topology

Exercise 16

The processes are distributed on a rectangular grid

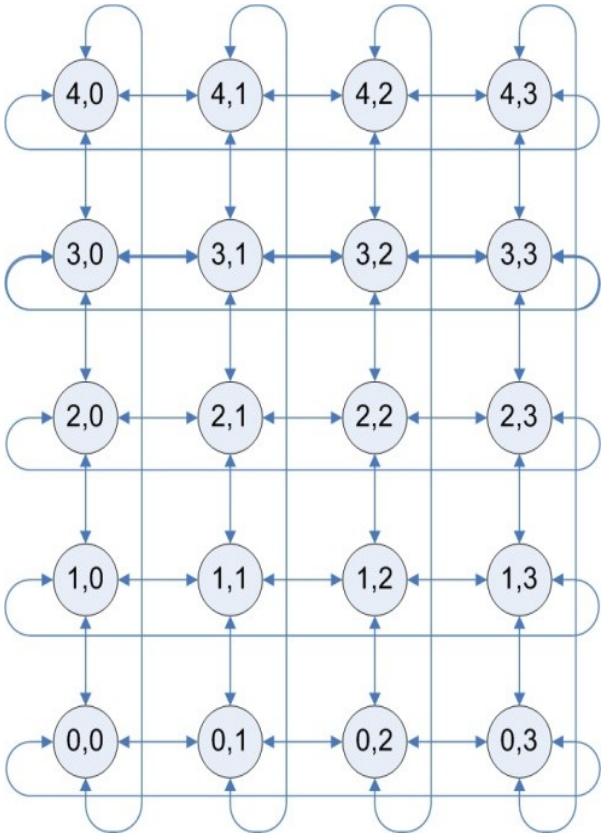
Each process:

- Initializes an integer variable A to the value of its *rank*
- calculates the average of A on first neighbours



Rank 0 process:

- Gathers the results from all other processes
- Writes the results as a table structured according to the coordinates of the processes



Arithmetic average on first neighbours in 2D topology

Exercise 16