Foundations of High Performance Computing

Lecture 06: Collective MPI operation and virtual topology

2021-2022 Stefano Cozzini

“Foundation of HPC” course

DATA SCIENCE &
SCIENTIFIC COMPUTING

Agenda

- Some remarks on blocking/non blocking operations
- Collective operations
- A primer on Virtual Topologies

Communication mode and MPI routines

• Mode	• **Completion Condition**	• Blocking subroutine	• Non-blocking subroutine
• **Standard send**	• **Message sent (receive state unknown)**	• **MPI_SEND**	• **MPI_ISEND**
• **receive**	• **Completes when a message has arrived**	• **MPI_RECV**	• **MPI_IRECV**
• Synchronous send	• **Only completes when the receive has completed**	• MPI_SSEND	• MPI_ISSEND
• Buffered send	• **Always completes,** • **irrespective of receiver**	• MPI_BSEND	• MPI_IBSEND
• Ready send	• **Always completes, irrespective of whether the receive has completed**	• MPI_RSEND	• MPI_IRSEND

Overview of MPI send modes

- MPI has a number of different "send modes." These represent different choices of buffering (where is the data kept until it is received) and synchronization (when does a send complete).

- MPI_Send
  MPI_Send will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).

- MPI_Bsend
  May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.

- MPI_Ssend
  will not return until matching receive posted

- MPI_Rsend
  May be used ONLY if matching receive already posted. User responsible for writing a correct program

Overview of MPI send modes

- Recommendation:
  - The best performance is likely if you can write your program so that you could use just MPI_Ssend; in that case, an MPI implementation can completely avoid buffering data.
  - Use MPI_Send instead; this allows the MPI implementation the maximum flexibility in choosing how to deliver your data
  - If nonblocking routines are necessary, then try to use MPI_Isend or MPI_Irecv. Use MPI_Bsend only when it is too inconvienent to use MPI_Isend
  - The remaining routines, MPI_Rsend, MPI_Issend, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.

Exercise

- play with different MPI_send call on mpi_pi.c

Collective Operations

- Collective routines provide a higher-level way to organize a parallel program
- Each process executes the same communication operations
- MPI provides a rich set of collective operations…

Collective Operations

- Communications involving group of processes in a communicator.
- Groups and communicators can be constructed “by hand” or using topology routines.
- No non-blocking collective operations.
- Three classes of operations:
  - synchronization,
  - data movement
  - collective computation

MPI_barrier()

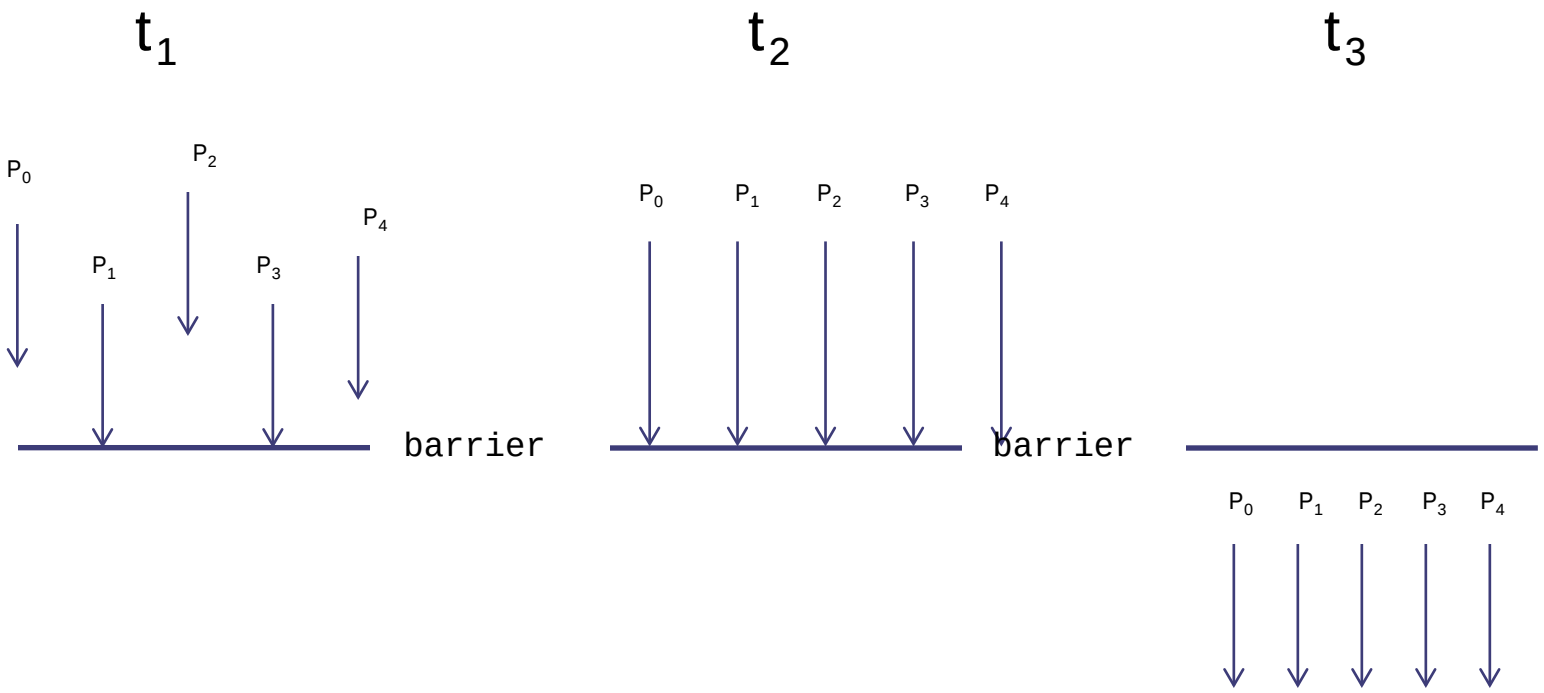
- Stop processes until all processes within a communicator reach the barrier
- Almost never required in a parallel program
  Occasionally useful in measuring performance and load balancing
- Fortran:

```
CALL MPI_BARRIER(comm, ierr)
```

- C:

```
int MPI_Barrier(MPI_Comm comm)
```

MPI_barrier(): graphical view



Broadcast (MPI_bcast)

- One-to-all communication: same data sent from root process to all others in the communicator
- Fortran:

```
INTEGER count, type, root, comm, ierr
CALL MPI_BCAST(buf,count,type,root,comm, ierr)
```

Buf array of type **type**

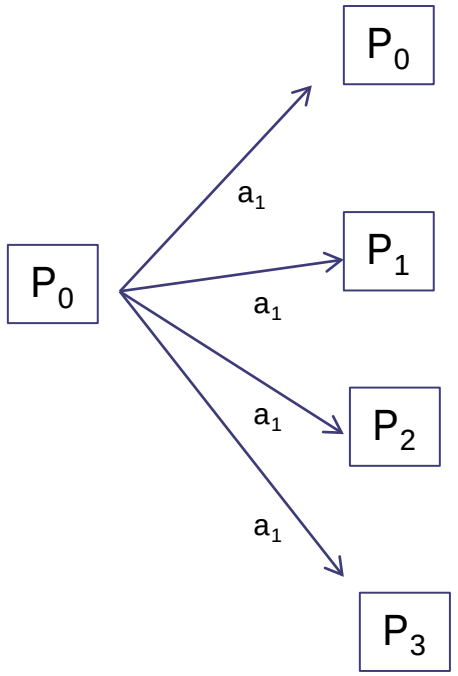
- C:

```
int MPI_Bcast(void *buf,int count,
MPI_Datatype datatype,int root,MPI_Comm comm)
```

- All processes must specify same **root**, **rank** and **comm**

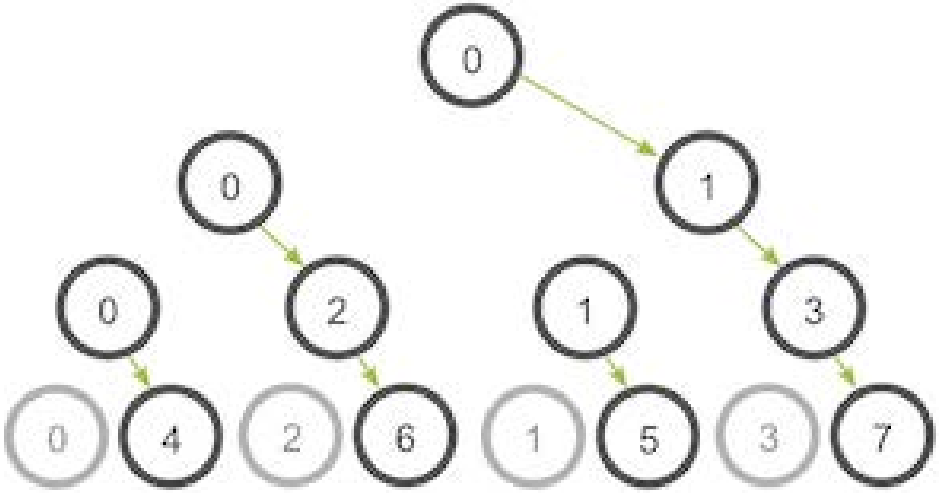
MPI_bcast example

```
PROGRAM broad_cast
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
 IF( myid .EQ. 0 ) THEN
   a(1) = 2.0
   a(2) = 4.0
 END IF
 WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
 CALL MPI_BCAST(a, 2, MPI_REAL, 0, MPI_COMM_WORLD,
ierr)
 WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
 CALL MPI_FINALIZE(ierr)
 END
```



Why to use collective MPI routines ?

- MPI collective operations exploit optimized solutions to realize communication between processors in a group.
- Tricks are played and implemented by MPI implementations.
- Ex: binary tree for broadcast operation:

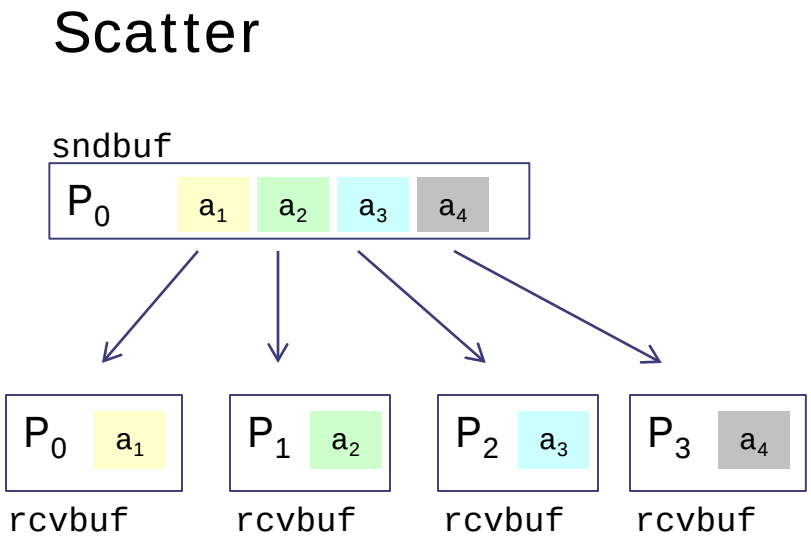


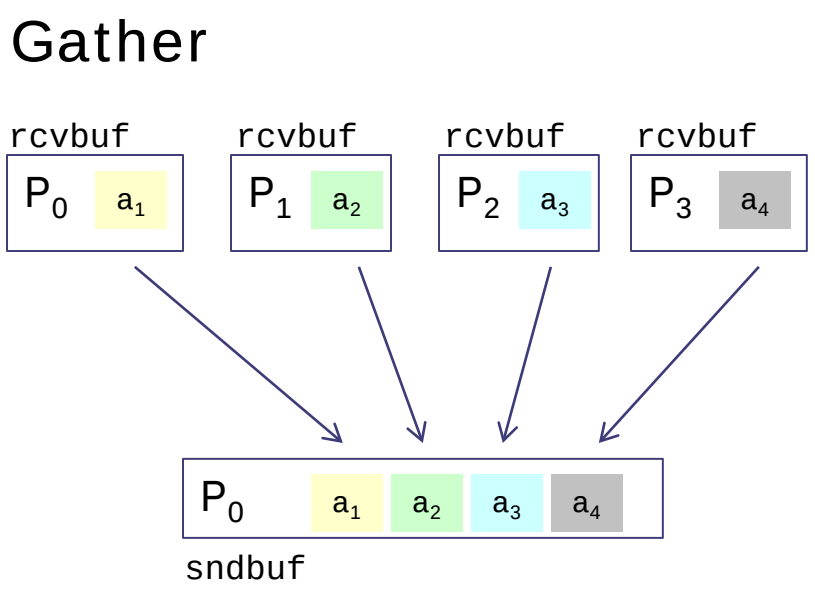
Exercise:

- Compare MPI_BCAST operation and its version developed using MPI_Send and MPI_Receive.

- See **`mpi_bcastcompare.c`** file

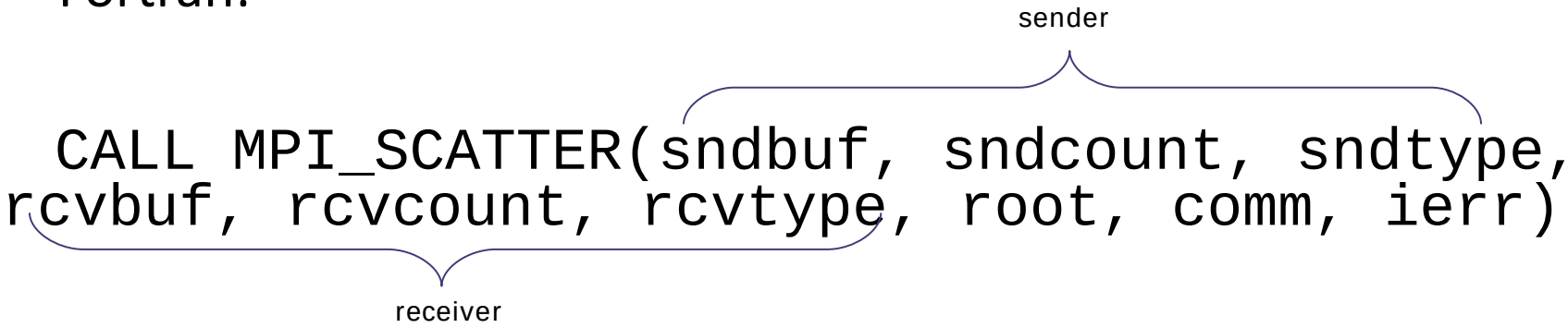
Scatter and Gather operations





MPI_scatter

- One-to-all communication: different data sent from root process to all others in the communicator
- Fortran:



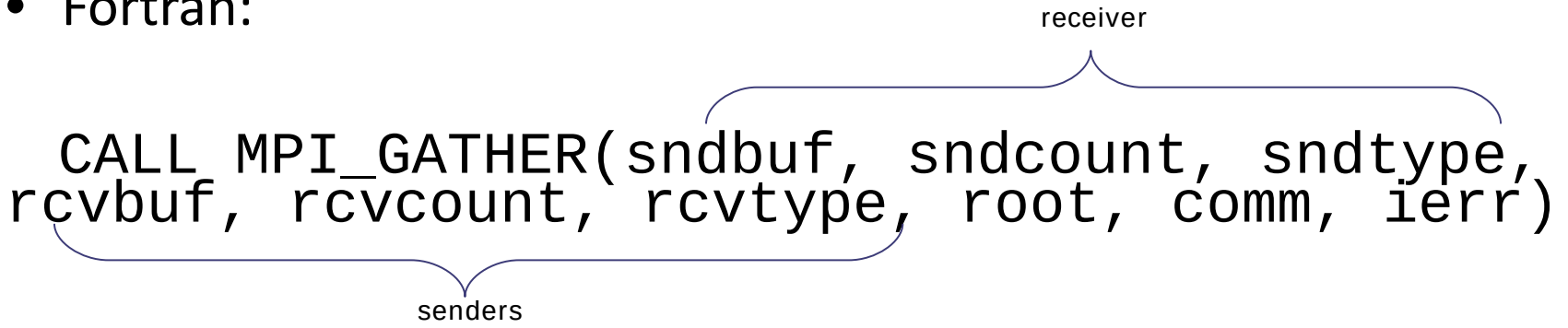
- Arguments definition are like other MPI subroutine
- **`sndcount`** is the number of elements sent to each process, not the size of **`sndbuf`**, that should be **`sndcount`** times the number of process in the communicator
- The sender arguments are significant only at root

Scatter: example

```
 PROGRAM scatter
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, nsnd, I, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(16), B(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 IF( myid .eq. root ) THEN
   DO i = 1, 16
     a(i) = REAL(i)
   END DO
 END IF
 nsnd = 2
 CALL MPI_SCATTER(a, nsnd, MPI_REAL, b, nsnd,
& MPI_REAL, root, MPI_COMM_WORLD, ierr)
 WRITE(6,*) myid, ': b(1)=', b(1), 'b(2)=', b(2)
 CALL MPI_FINALIZE(ierr)
 END
```

MPI_gather

- One-to-all communication: : different data collected by the root process, from all others processes in the communicator. Is the opposite of Scatter
- Fortran:



- Arguments definition are like other MPI subroutine
- **`rcvcount`** is the number of elements collected from each process, not the size of **`rcvbuf`**, that should be **`rcvcount`** times the number of process in the communicator
- The receiver arguments are significant only at root

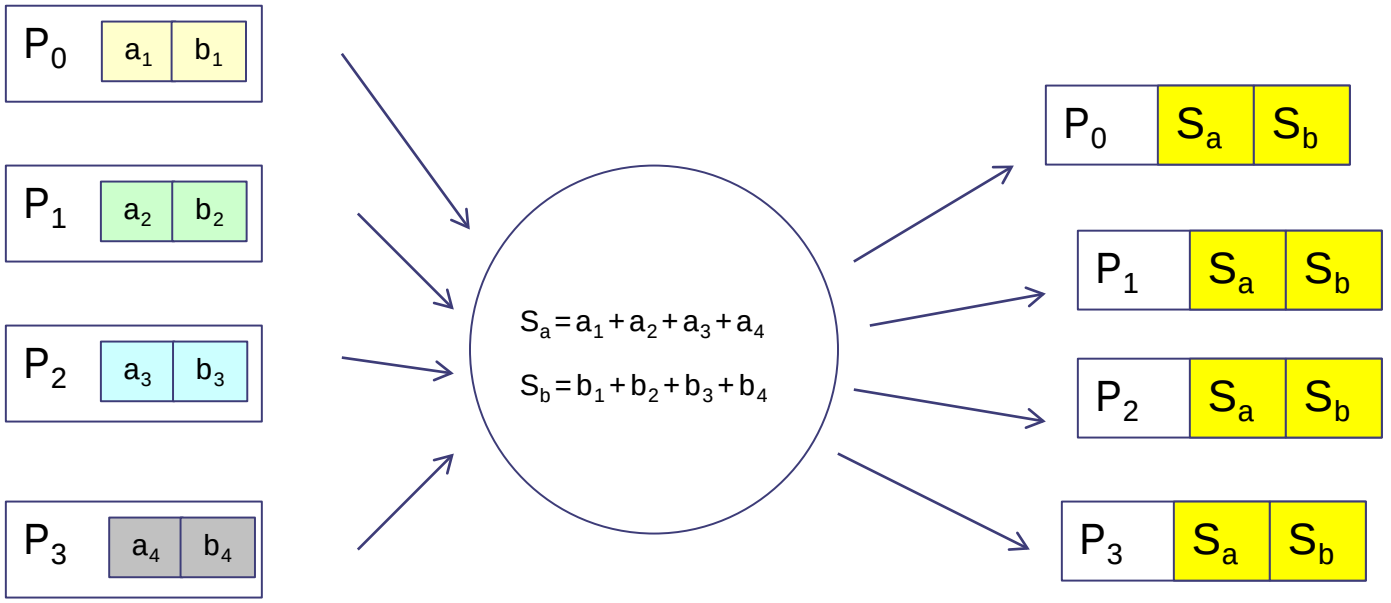
MPI_gather example

```
 PROGRAM gather
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, nsnd, I, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(16), B(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 b(1) = REAL( myid )
 b(2) = REAL( myid )
 nsnd = 2
 CALL MPI_GATHER(b, nsnd, MPI_REAL, a, nsnd,
&  MPI_REAL, root MPI_COMM_WORLD, ierr)
 IF( myid .eq. root ) THEN
   DO i = 1, (nsnd*nproc)
     WRITE(6,*) myid, ': a(i)=', a(i)
   END DO
 END IF
 CALL MPI_FINALIZE(ierr)
 END
```

Reduction

- The reduction operation allows to:
  - Collect data from each process
  - Reduce the data to a single value
  - Store the result on the root processes
  - Store the result on all processes

Reduce: parallel sum



Reduction function works with arrays other operation: product, min, max, and, ….

MPI_reduce/MPI_allreduce

- C:

```
int MPI_Reduce(void * snd_buf, void * rcv_buf, int count,
MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)
```

```
int MPI_Allreduce(void * snd_buf, void * rcv_buf, int
count, MPI_Datatype type, MPI_Op op, MPI_Comm comm)
```

MPI_reduce/MPI_allreduce

- Fortran:

- **MPI_REDUCE(snd_buf,rcv_buf,count,type,op,root,comm,ierr)**

snd_buf	input array of type type containing local values.
rcv_buf	output array of type type containing global results
count	(INTEGER) number of element of snd_buf and rcv_buf
type	(INTEGER) MPI type of snd_buf and rcv_buf
op	(INTEGER) parallel operation to be performed
root	(INTEGER) MPI id of the process storing the result
comm	(INTEGER) communicator of processes involved in the operation
ierr	(INTEGER) output, error code (if ierr=0 no error occours)

- **MPI_ALLREDUCE(snd_buf,rcv_buf,count,type,op,comm,ierr)**
- The argument root is missing, the result is stored to all processes.

Predefined collective operations

MPI op	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Reduce: example

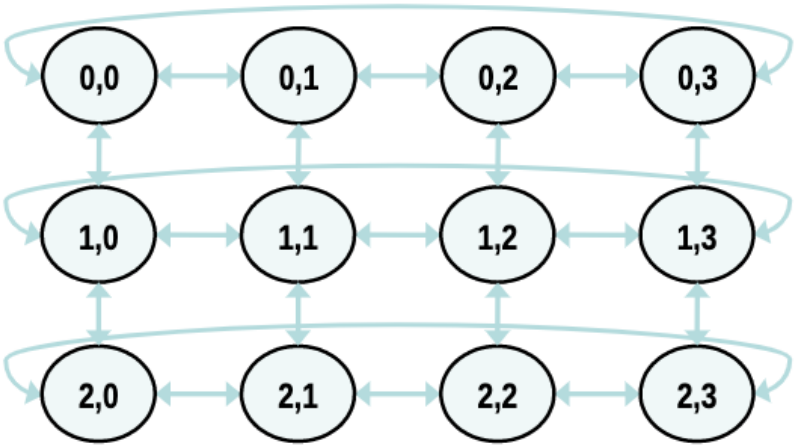
```
 PROGRAM reduce
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2), res(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 a(1) = 2.0
 a(2) = 4.0
 CALL MPI_REDUCE(a, res, 2, MPI_REAL, MPI_SUM, root,
&  MPI_COMM_WORLD, ierr)
 IF( myid .EQ. 0 ) THEN
   WRITE(6,*) myid, ': res(1)=', res(1), 'res(2)=', res(2)
 END IF
 CALL MPI_FINALIZE(ierr)
 END
```

Exercise:

- modify mpi_pi.c to use a reduce operation

Virtual topologies

- MPI provides routines to create new communicators that order the process ranks in a way that may be a better match for the topology of the problem
  - Example: Nearest neighbor exchange in a mesh



Example:

- A 3D box of 40000x40000x2000 DP elements:
  - 1.6 Terabyte of space
- We can partition it on 3D box of processors, assigning each process 10000x10000x1000 elements
- How many processes ? 4x4x2
- Which process is taking care of the of box (1,1,2) ?
- process coordinates: can be handled with virtual Cartesian topologies
- Array decomposition: handled by the application program directly (see next lecture)

Virtual topologies

- Process topologies in MPI allow simple referencing of processes
- Cartesian and graph topologies are supported
- Process topology defines a new communicator
- MPI topologies are virtual
  - no relation to the physical structure of the computer
  - data mapping “more natural” only to the programmer
- Usually, no performance benefits
- But code becomes more readable

Virtual and Physical topology

- A ***virtual topology*** represents the way that MPI processes communicate:

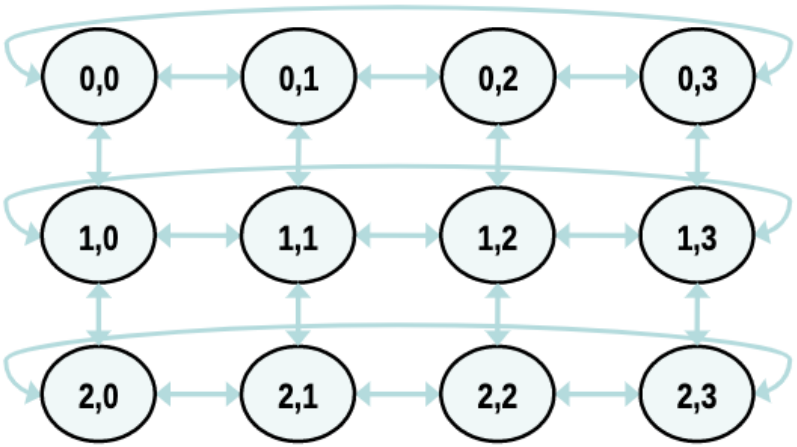
- A ***physical topology*** represents that connections between the cores, chips, and nodes in the hardware
- On modern and complex HPC architecture you may want to use a virtual topology that matches the physical topology.

How to use a Virtual topology ?

- Creating a topology produces a new communicator.
- MPI provides mapping functions:
  - to compute process ranks, based on the topology naming scheme,
  - and vice versa.

Cartesian virtual topologies

- Cartesian Topologies
  - each process is connected to its neighbor in a virtual grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course, communication between any two processes is still allowed.



MPI Cartesian topology

- Create a new virtual topology using

  `MPI_Cart_create`

- Determine “good” sizes of mesh with

  `MPI_Dims_create`

MPI_Cart_create

- **int MPI_Cart_create(**
  MPI_Comm old_communicator,
  int ndim,
  const int* dims,
  const int* qperiods,
  int qreorder,
  MPI_Comm* new_communicator);

- Creates a new communicator **new_communicator** from **old_communicator**, that represents a **ndim** dimensional mesh with sizes **dims**. The mesh is periodic in coordinate direction i if **qperiodic[i]** is true. The ranks in the new communicator are reordered (to better match the physical topology) if **qreorder** is true

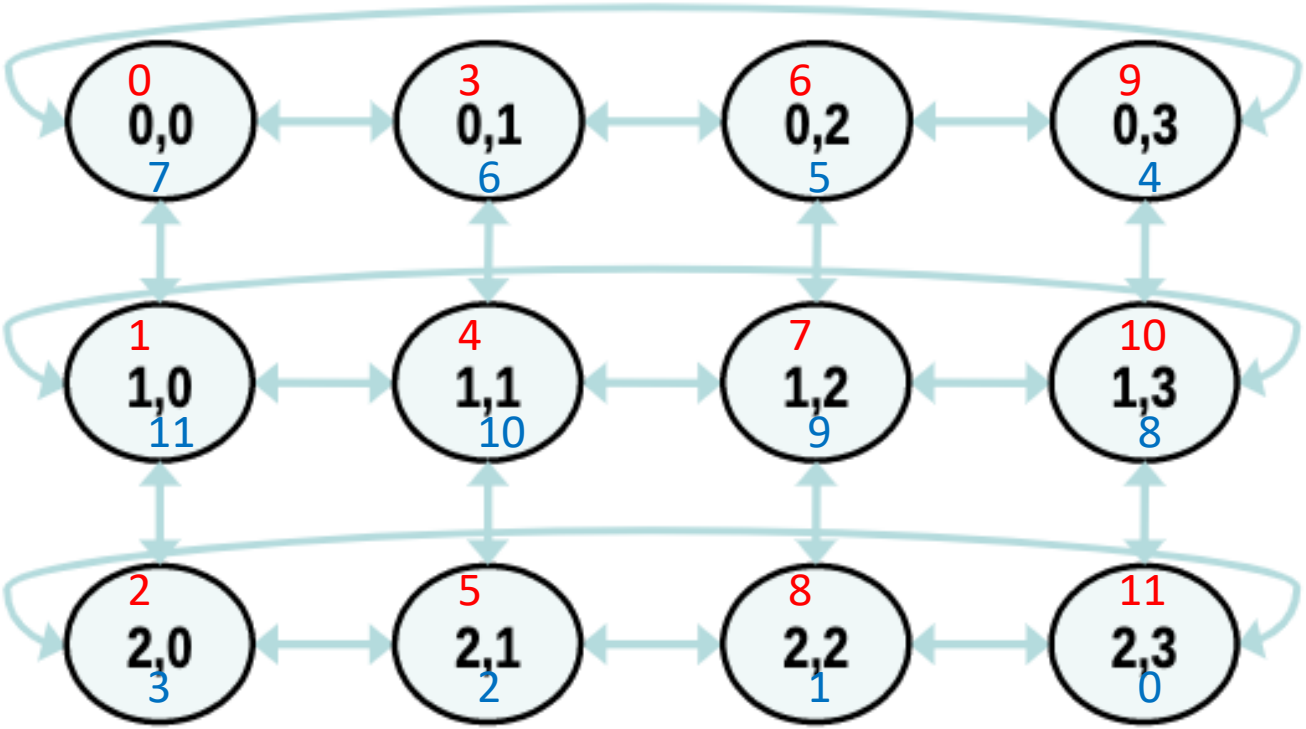
MPI_Dims_create

```
int MPI_Dims_create(int nnodes, int ndim, int dims[])
```

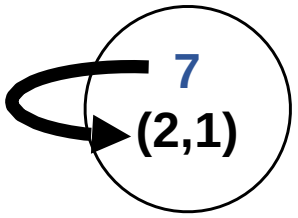
- Fill in the **dims** array such that the product of **dims[i]** for i=0 to **ndim-1** equals **nnodes**.
- Any value of **dims[i]** that is 0 on input will be replaced; values that are > 0 will not be changed

Example: 2D torus

- Ranks and Cartesian process coordinates in comm_cart
- Ranks in comm and comm_cart may differ, if reorder = 1 or .TRUE.
- This reordering can allow MPI to optimize communications



Cartesian mapping functions



Mapping
rank to
process grid coordinates

- Translate a rank to mapping coordinates

```
MPI_Cart_coords(comm,
rank, maxdim,coords)
```

- Arguments:

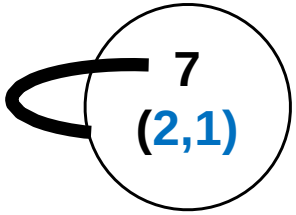
  Comm = coordinator

  Rank : rank to convert

  Maxdim : number of coordinates

  Coords : coordinates in Cartesian topology that corresponds to rank

Cartesian mapping functions (2)



Mapping
grid coordinates to
ranks

- Translate a set of coordinates to rank

```
MPI_Cart_rank(comm, coords, rank)
```

- Arguments:

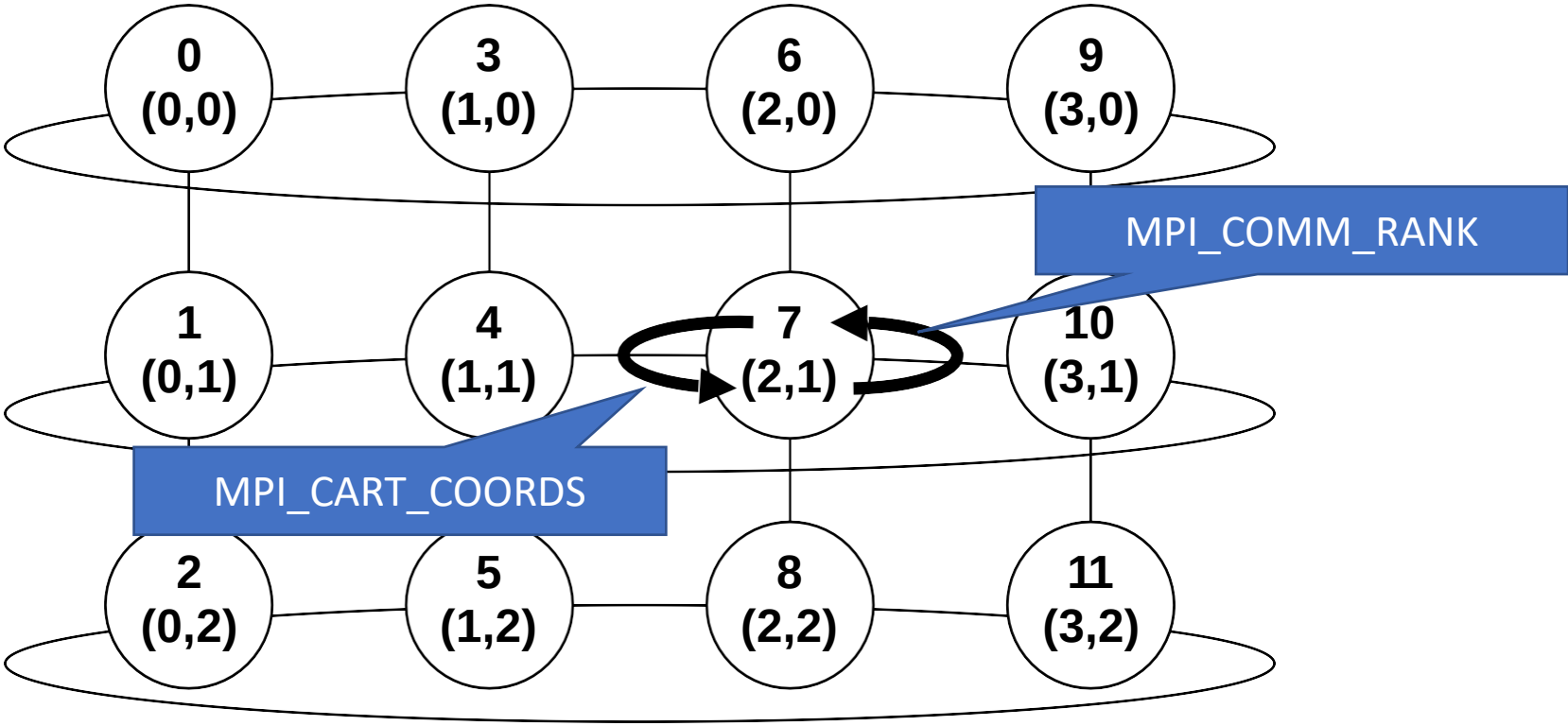
  Comm = coordinator

  Coords : array of coordinates

  Rank : the rank corresponding to coords

Own coordinates

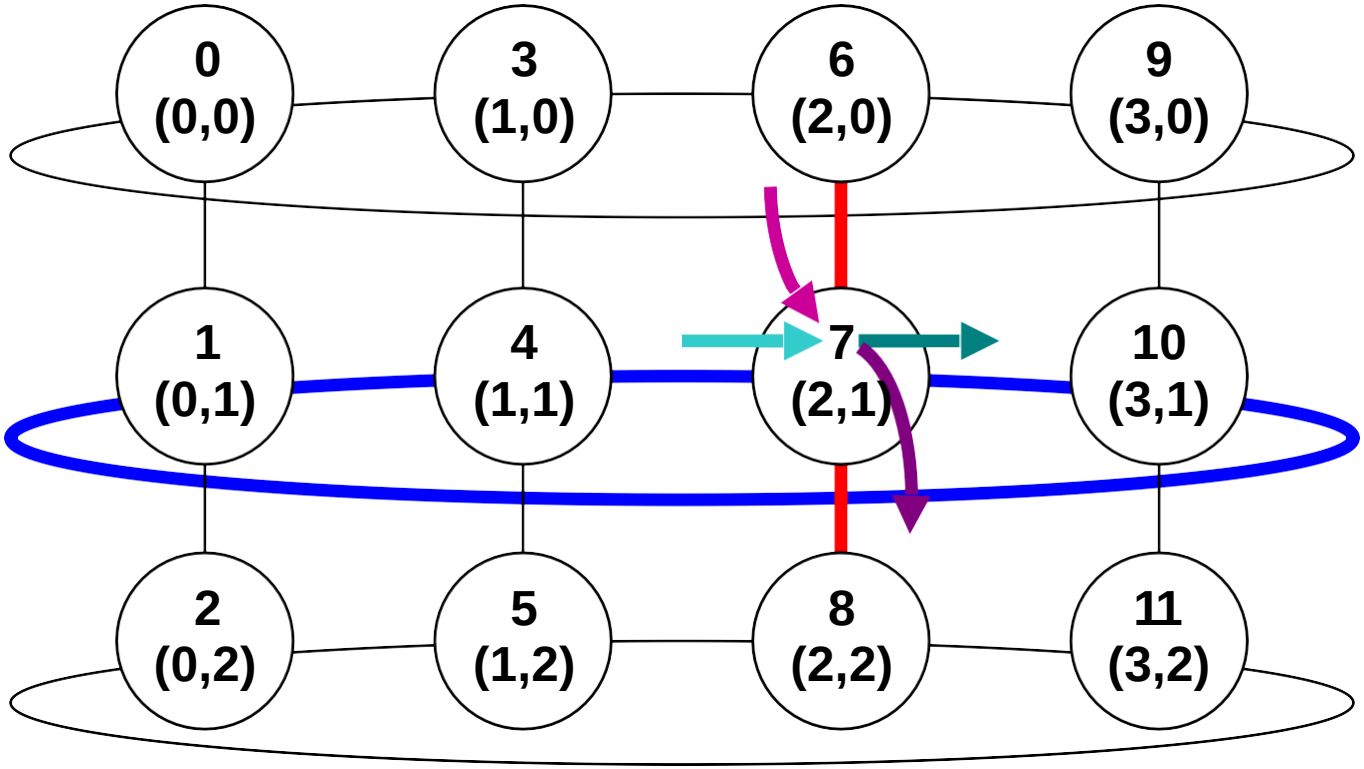
- Each process gets its own coordinates with
  `MPI_Comm_rank(comm_cart, my_rank)`
  `MPI_Cart_coords(comm_cart, my_rank, maxdims, my_coords)`



MPI_Cart_shift

- Does not actually shift data: returns the correct ranks for a shift that can be used in subsequent communication calls
- Arguments:
  - direction (dimension in which the shift should be made)
  - disp (length of the shift in processor coordinates [+ or -])
  - rank_source (where calling process should receive a message from during the shift)
  - rank_dest (where calling process should send a message to during the shift)
  - if we shift off of the topology, MPI_Proc_null (-1) is returned

MPI_Cart_shift example



- **`MPI_Cart_shift( cart, direction, displace, rank_prev, rank_next)`**

MPI_Cart_create Example:2D thorus (1)

```
* @brief Original source code at
https://www.rookiehpc.com/mpi/docs/mpi_cart_create.php
 **/
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    // Size of the default communicator
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Ask MPI to decompose our processes in a 2D cartesian grid for us
    int dims[2] = {0, 0};
    MPI_Dims_create(size, 2, dims);

    // Make both dimensions periodic
    int periods[2] = {true, true};

    // Let MPI assign arbitrary ranks if it deems it necessary
    int reorder = true;

}
```

MPI_Cart_create Example:2D torus (2)

```
* @brief Original source code at
https://www.rookiehpc.com/mpi/docs/mpi_cart_create.php
 **/

 // Create a communicator given the 2D torus topology.
    MPI_Comm new_communicator;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder,
&new_communicator);

    // My rank in the new communicator
    int my_rank;
    MPI_Comm_rank(new_communicator, &my_rank);

    // Get my coordinates in the new communicator
    int my_coords[2];
    MPI_Cart_coords(new_communicator, my_rank, 2, my_coords);

    // Print my location in the 2D torus.
    printf("[MPI process %d] I am located at (%d, %d).\n", my_rank,
my_coords[0],my_coords[1]);

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Exercises

- A. setup a ring
  - 1D periodic array
  - Check the effect of reordering
- B. Exercise sets up 2D domain with
  - periodic and
  - non-periodic boundary condition
  - Play with MPI_cart_shift