# More Communicators

## Duplicating, Splitting and Slicing

# Overview

- Duplicating communicators
- Splitting communicators
- Deleting communicators
- (Cartesian communicators)

# Copying and duplicating communicators

# Copying communicator handles

- Earlier in the course we had code like (C example):

```
MPI_COMM com_a, com_b;
...
com_b = com_a;
```

- **`com_b`** & **`com_a`**: different names for same communicator
  - messages sent with argument **`com_b`** can be received in **`com_a`**
- Not what you want when e.g. writing a parallel library
  - messages inside the library should not interfere with user code

# Duplicating communicators

- Alternative: Duplicating a communicator
- In C:

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

- In Fortran 90:

```
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
```

- In Python:

```
comm.Dup()
```

- **`comm:`** existing communicator (input)
- **`newcomm:`** new communicator (output)
- newcomm has the same properties (order, topology, etc.) as comm, but messages sent inside comm will not be received in newcomm
- **Rem.:** Fortran 2008: **`type(MPI_Comm)`** for communicators

# Example in C: Copying communicator

Program on P0

```
MPI_Comm com_a, com_b;
...
com_b = com_a;


MPI_Irecv(&a,1,MPI_INT,1
  ,0,com_a,stat);
MPI_Irecv(&b,1,MPI_INT,1
  ,0,com_b,stat);
```

Program on P1

```
MPI_Comm com_a, com_b;
...
com_b = com_a;


MPI_Send(&c,1,MPI_INT,
   0,0,com_b)
```

- The message will be received into **a**
- For MPI **com_a** and **com_b** are the same

# Example in C: Duplicating communicator

Program on P0

```
MPI_Comm com_a, com_b;
...
MPI_Comm_dup(com_a,
  &com_b);


MPI_Irecv(&a,1,MPI_INT,1
  ,0,com_a,stat);
MPI_Irecv(&b,1,MPI_INT,1
  ,0,com_b,stat);
```

Program on P1

```
MPI_Comm com_a, com_b;
...
MPI_Comm_dup(com_a,
&com_b);


MPI_Send(&c,1,MPI_INT,
   0,0,com_b)
```

- The message will be received into **`b`**
- Now **`com_a`** and **`com_b`** are different for MPI

# Example in Python: Copying communicator

Program on P0

```
com_a: MPI.Comm = ...
...
com_b = com_a


req_a =
  com_a.irecv(source=1)
req_b =
  com_b.irecv(source=1)
```

Program on P1

```
com_a: MPI.Comm = ...
...
com_b = com_a;


com_b.send(c,dest=0)
```

- The message will be received into **`req_a`**
- To MPI, **`com_a`** and **`com_b`** are the same

# Example in Python: Duplicate communicator

Program on P0

```
com_a: MPI.Comm = ...
...
com_b = com_a.Dup()

req_a =
 com_a.irecv(source=1)
req_b =
 com_b.irecv(source=1)
```

Program on P1

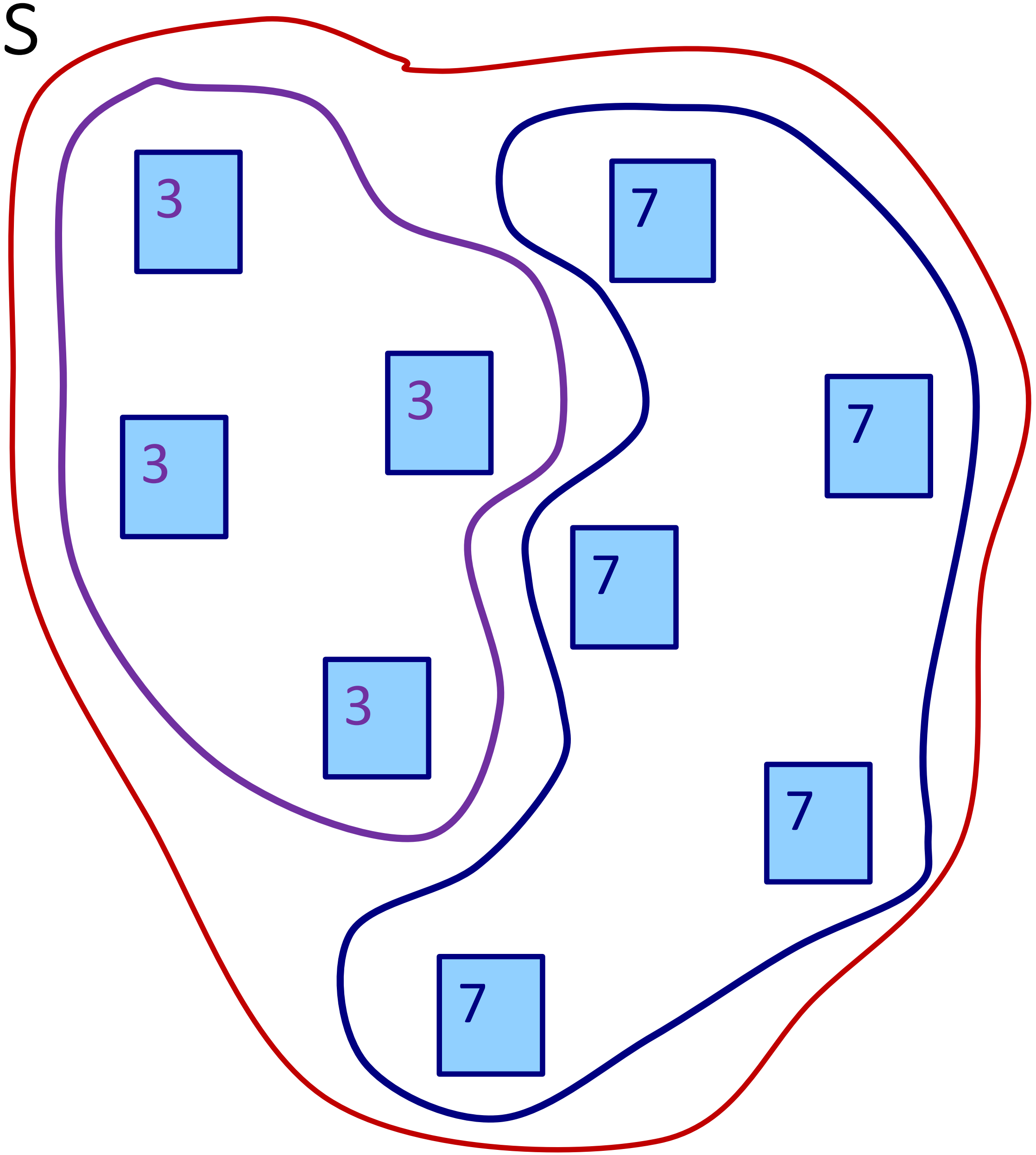
```
com_a: MPI.Comm = ...
...
com_b = com_a;

com_b.send(c,dest=0)
```

- The message will be received into **`req_b`**
- Now **`com_a`** and **`com_b`** are different to MPI

# splitting communicators

# Splitting communicators

- Splitting a communicator
- Define a colour value
- All tasks with the same colour will end up in the same split comm
- Each task knows only about the split comm he is in



# MPI_Comm_split in C

```
int MPI_Comm_split(MPI_Comm comm, int color,
   int key, MPI_Comm *newcomm)
```

**`comm:`** original communicator to be split (input)

**`color:`** ranks with same color value into same newcomm

**`key:`** tasks get newcomm ranks assigned according to key, lowest key first. In case of ties, lower rank in comm comes first

**`newcomm:`** new split communicator (output)

**Remark:** **`newcomm`** is a different comm on tasks with different color

# MPI_Comm_split in Fortran 90

```
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, &
  IERROR)
  INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

**`comm:`** original communicator to be split (input)

**`color:`** ranks with same color value into same newcomm

**`key:`** tasks get newcomm ranks assigned according to key, lowest key first. In case of ties, lower rank in comm comes first

**`newcomm:`** new split communicator (output)

**Remark: `newcomm`** is a different comm on tasks with different color

# split communicator in Python

```
comm.Split(color, key)
```

**`comm:`** original communicator to be split (input)

**`color:`** ranks with same color value into same newcomm

**`key:`** tasks get newcomm ranks assigned according to key, lowest key first. In case of ties, lower rank in comm comes first

New split communicator is returned by `Split`

**Remark: `newcomm`** is a different comm on tasks with different color

# Deleting communicators

- In C

```
int MPI_Comm_free(MPI_Comm *comm)
```

- In Fortran

```
MPI_COMM_FREE(COMM, IERROR)
  INTEGER COMM, IERROR
```

- In Python

```
comm.Free()
```

- This removes the communicator **comm**

**Remark:** Communicator creation and destruction are typically not well optimised. Don't use frequently.

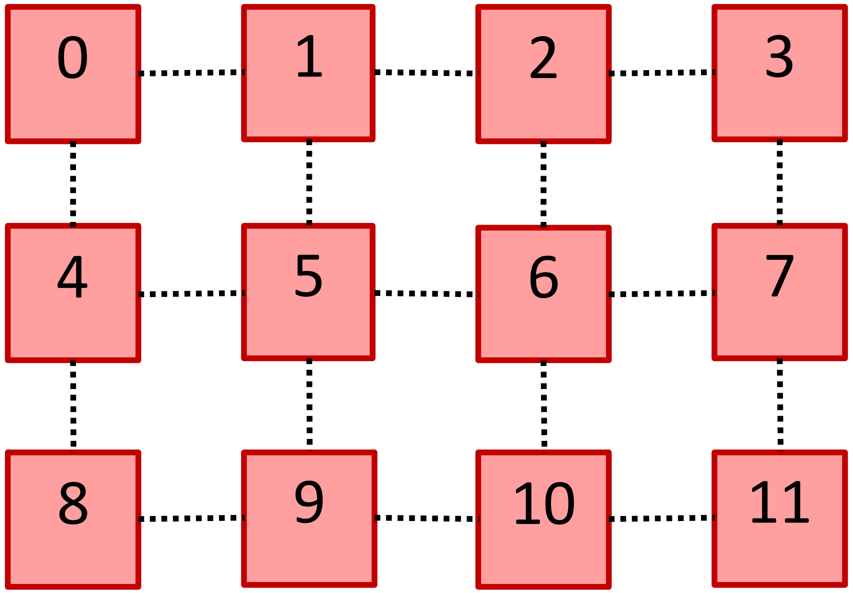
# Cartesian Topologies

# Virtual Topologies

- Communicators can feature virtual topologies
  - Graph (not part of this course)
  - Cartesian structure
- These virtual topologies have **nothing** to do with the network topologies
- Virtual topologies are convenient
- The virtual topology **can** exploit hardware topology to boost performance

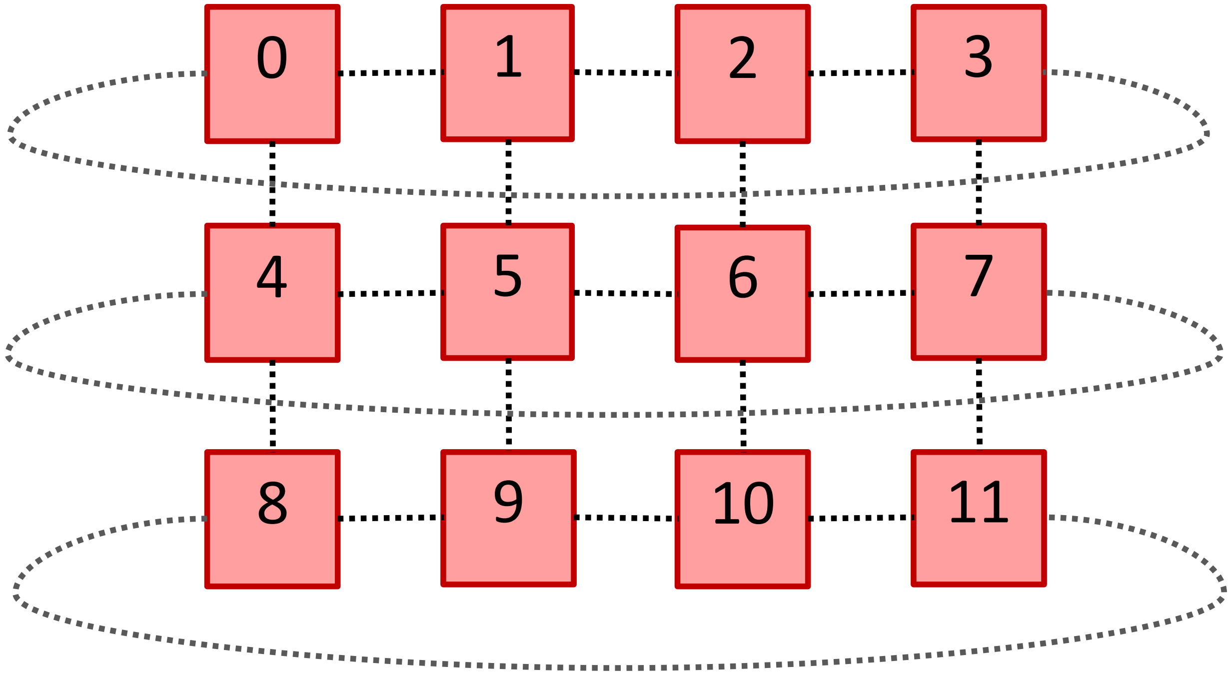
# Cartesian Topologies

- Tasks arranged on an n-dimensional grid
- Each dimension:
  - lower neighbour
  - upper neighbour
- Ranks match onto grid in **row major order**
  - As usual in MPI: The C way
- Example picture:
  - 12 tasks (Rank: 0...11)
  - Dimension: (3,4)
  - Open boundary



# Periodic boundaries

- Alternatively:
  Periodic boundaries
- Can choose separately for each dimension
- Example picture:
  - Open in 1st dimension
  - Periodic in 2nd dimension



# MPI_CART_CREATE in C

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

- `comm_old` input communicator (task to be arranged)
- `ndims` number of dimensions
- `dims` array specifying the extent in each direction
- `periods` logical array, specifying boundary in each dir.
- `reorder` logical, allowing/disallowing reordering
- `comm_cart` new communicator (output) with cart. topology

# MPI_CART_CREATE in Fortran 90

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, &
 PERIODS, REORDER, COMM_CART, IERROR)
```

```
INTEGER COMM_OLD,NDIMS,DIMS(*),COMM_CART,IERROR
LOGICAL PERIODS(*), REORDER
```

- `comm_old` input communicator (task to be arranged)
- `ndims` number of dimensions
- `dims` array specifying the extent in each direction
- `periods` logical array, specifying boundary in each dir.
- `reorder` logical, allowing/disallowing reordering
- `comm_cart` new communicator (output) with cart. topology

# MPI_DIMS_CREATE

- Automatically distribute the available task onto an n-dimensional Cartesian grid
  - Uses all available tasks
  - keeps dimensions as equal as possible
- Gives output suitable for **`MPI_Cart_create`**
- Can restrict dimensions, set dims component ≠ 0

| Tasks | # dims | Dims on input | Dims on output |
|---|---|---|---|
| 6 | 2 | (0,0) | (3,2) |
| 7 | 2 | (0,0) | (7,1) |
| 6 | 3 | (0,3,0) | (2,3,1) |
| 7 | 3 | (0,3,0) | Error, doesn’t divide |

# MPI_DIMS_CREATE: syntax

- In C

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

- In Fortran 90

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

`nnodes`: Number of tasks in cartesian comm

`ndims`: Number of dimensions of cartesian comm

`dims`: Array of size `ndims`, **input**: constraints, **output**: extent

**Remark:** remember to set dims (0 or else) before the call

# MPI_CART_RANK

*The coordinates are know, query the rank of any task*

C:

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

Fortran 90:

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

`comm:` Cartesian communicator

`coords:` Array with the cartesian coordinates (input)

`rank:` Rank of that task (output)

# MPI_CART_COORDS

Enquire the Cartesian coordinates of a given rank (any task)

C:

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

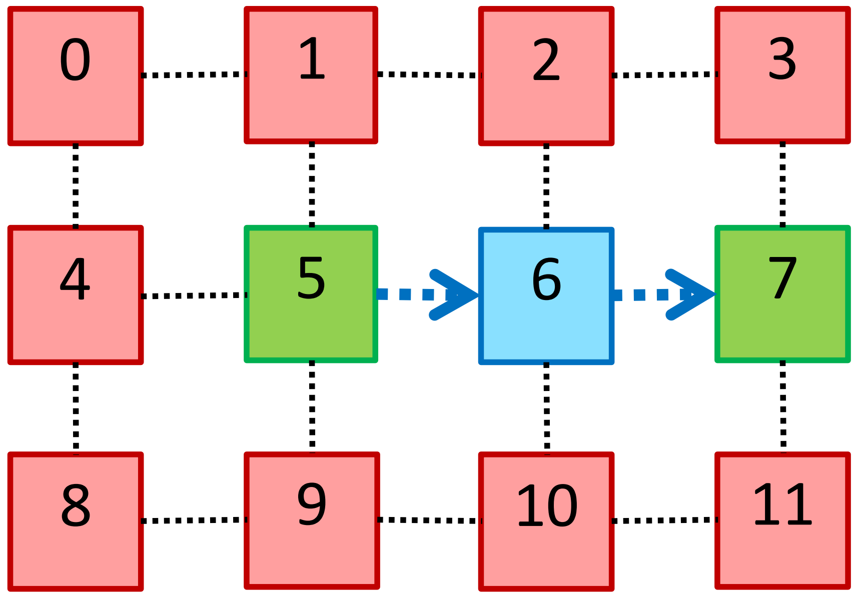
Fortran 90:

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

| | |
|---|---|
| `comm:` | Cartesian communicator |
| `rank:` | Rank of that task (input) |
| `maxdims:` | length of vector coords in calling program |
| `coords:` | Array with the cartesian coordinates (output) |

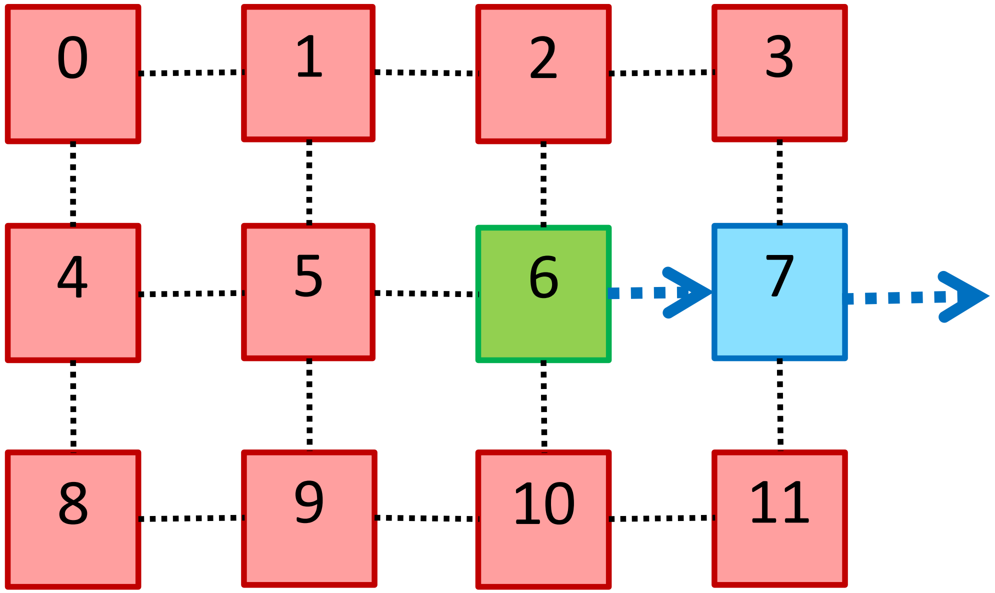
# MPI_CART_SHIFT

- Query function for “neighbours”
- Input
  - direction (from `0` to **`ndim-1`**)
  - shift
- Returns
  - Rank to receive from
  - Rank to send to
- Example:
  - direction: 1,  shift: +1
  - On rank 6 it returns:
    - send to 7
    - receive from 5



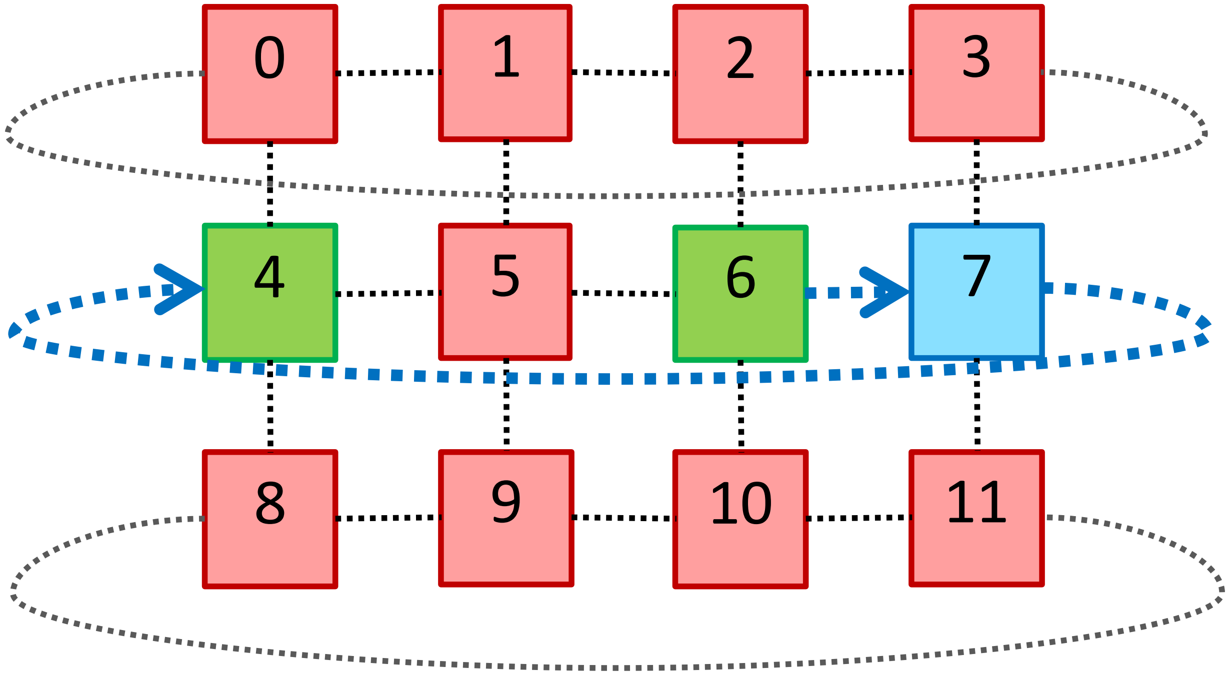
# MPI_CART_SHIFT at the edge

- Non-periodic boundary
- Neighbour beyond the grid:
  **`MPI_PROC_NULL`**
- Comm. with **`MPI_PROC_NULL`**:
  - does nothing
  - returns directly
  - No need for “if-guard”
- Example:
  - direction: 1,  shift: +1
  - On rank 7 it returns:
    - send to **`MPI_PROC_NULL`**
    - receive from 6



# MPI_CART_SHIFT with periodic boundaries

- Periodic boundary
- Works as expected
- Example:
  - direction: 1, shift: +1
  - On rank 7 it returns:
    - send to 4
    - receive from 6



# MPI_CART_SHIFT

In C:

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
 int disp, int *rank_source, int *rank_dest)
```

In Fortran 90:

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE,
 RANK_DEST, IERROR)
```

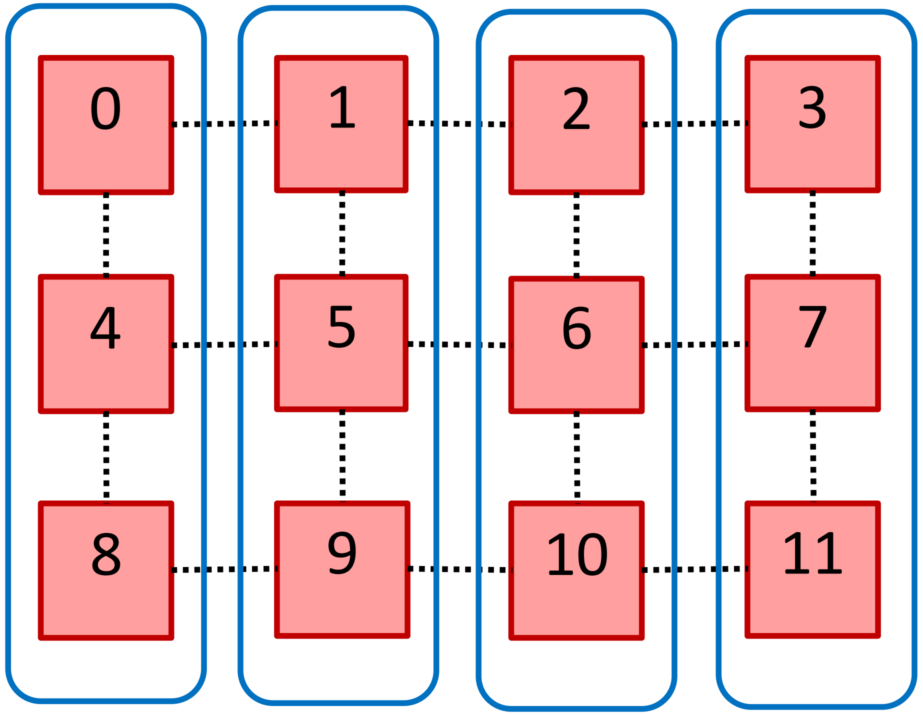
```
INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, &
RANK_DEST, IERROR
```

- `comm:` Communicator
- `direction:` Grid dimension (0 .. n-1)
- `disp:` How far to hop (positive or negative)
- `rank_source:` Rank to expect data from
- `rank_dest:` Rank to send data to

# MPI_CART_SUB

- Create sub-communicators, e.g.:
  - Column of Matrix
  - Slices of a Volume
- Sub-communicator inherits Cartesian topology
- Required for e.g. collectives

- Example:
  - Start: 3 × 4 grid
  - Create: 4 communicators with 1-D topology of length 3



# MPI_CART_SUB syntax

In C:

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,
 MPI_Comm *newcomm)
```

In Fortran 90:

```
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
 INTEGER COMM, NEWCOMM, IERROR
 LOGICAL REMAIN_DIMS(*)
```

- `comm:` Communicator to be split up
- `remain_dims:` Logical array, set ”true” for comp. to remain
- `newcomm:` New split communicator

# Examples for remain_dims

- Splitting a $3 \times 4 \times 5$ Cartesian communicator

| remain_dims | # comm | topology |
|---|---|---|
| (true, false, true) | 4 | $3 \times 5$ |
| (false, true, false) | 15 | 4 |
| (false, true, true) | 3 | $4 \times 5$ |

# Summary

- Copying communicator handles ***vs*** duplicating communicators
- Splitting communicators

*Provided slides for*

- *Creating Cartesian communicators*
  - *Querying properties*
    - *Rank*
    - *Cartesian coordinates*
    - *Neighbours*
- *Cartesian sub-communicators*

# Exercise splitting communicators

- Write an program the creates a base communicator of size **`N`**
- Create split communicators of size **`N/2`**
  - Even ranks should be in one split communicator
  - Odd ranks should be in the other split communicator
- Calculate the average rank number in split and base comm
  - Over the base communicator
  - The relevant split communicator
- Base rank 0 should print the results for the base comm
- The two split ranks 0 should print the results for split comms
- You should implement using only two reductions per task

# Exercise cartesian communicators

- Create a 2D Cartesian communicator
  - open boundaries in the 1st dimension
  - periodic boundaries in the 2nd dimension
  - use **`MPI_Dims_create`**
- Send your rank and coordinates to your four neighbours
- Each processor should print own and neighbours rank and coordinates
- **Additional exercise**: Create sub-communicators and use collective communication to calculate the sum of the ranks in each row