Parallel programming / computation

Sultan ALPAR
s.alpar@iitu.edu.kz

IITU

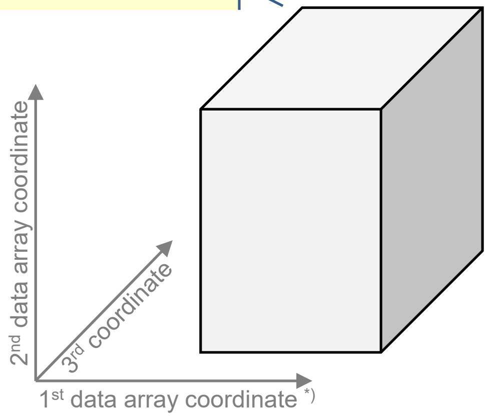
Lecture 7

Virtual Topologies
A multi-dimensional process naming scheme

Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)

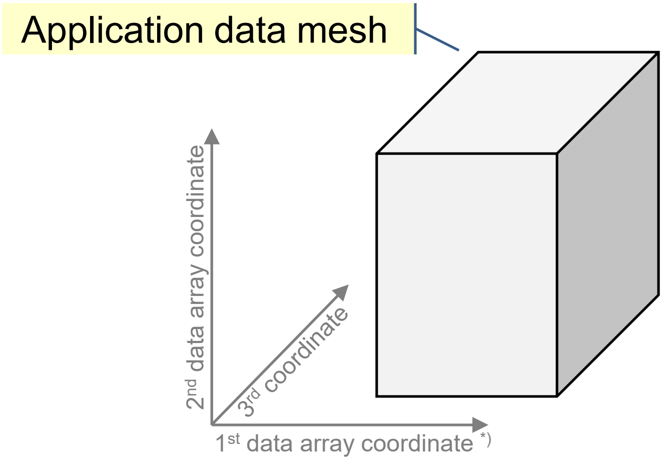
Application data mesh



*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)

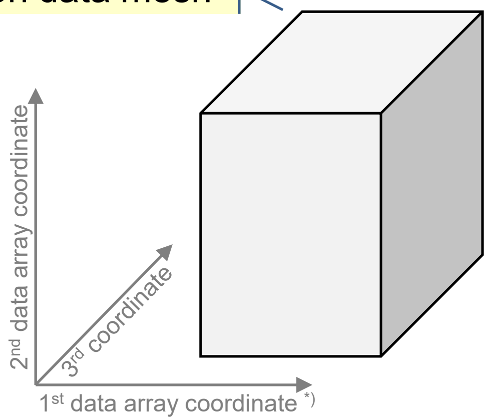


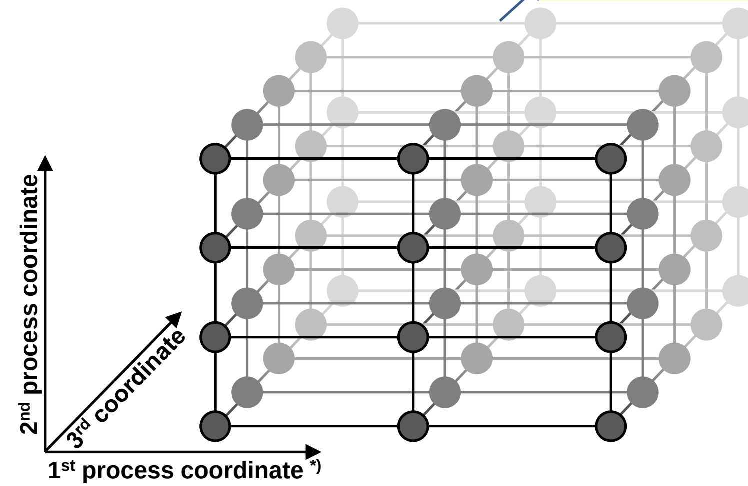
- on **3 x 4 x 5 = 60 processes**
- process coordinates **0..2, 0..3, 0..4**

*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)





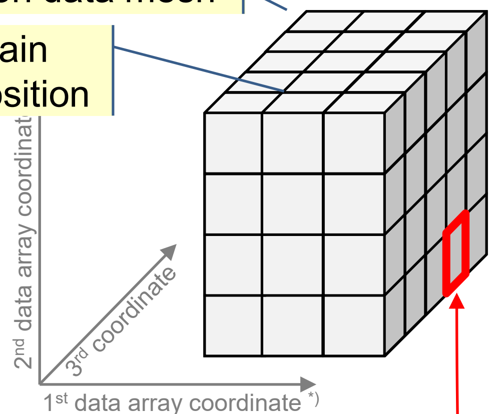
- on **3 x 4 x 5 = 60 processes**
- process coordinates **0..2, 0..3, 0..4**

*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

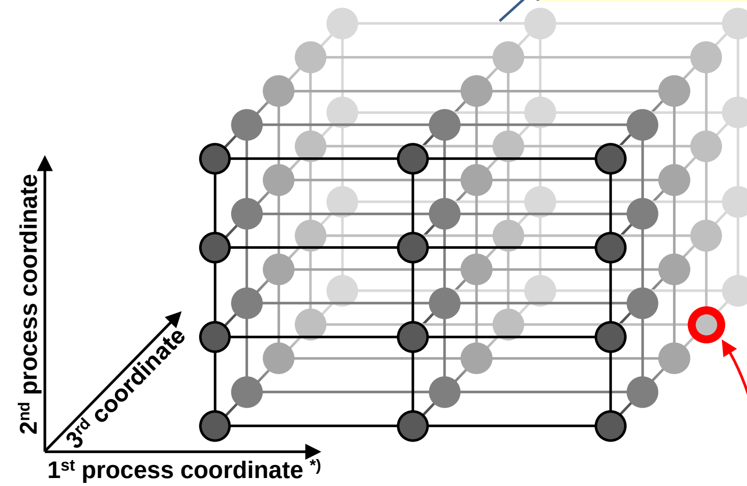
Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)

Application data mesh with domain decomposition



Virtual process grid

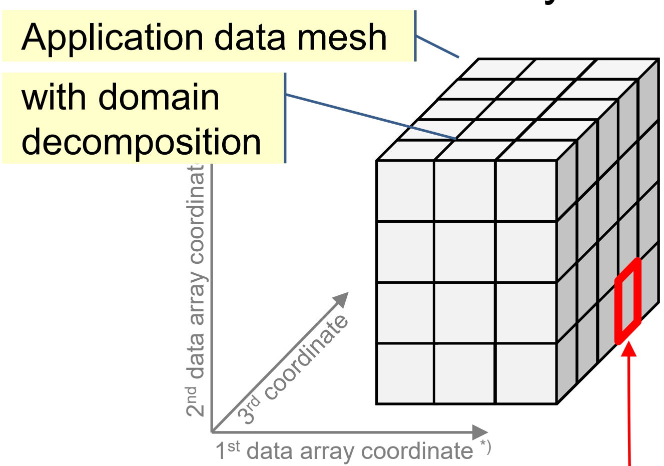


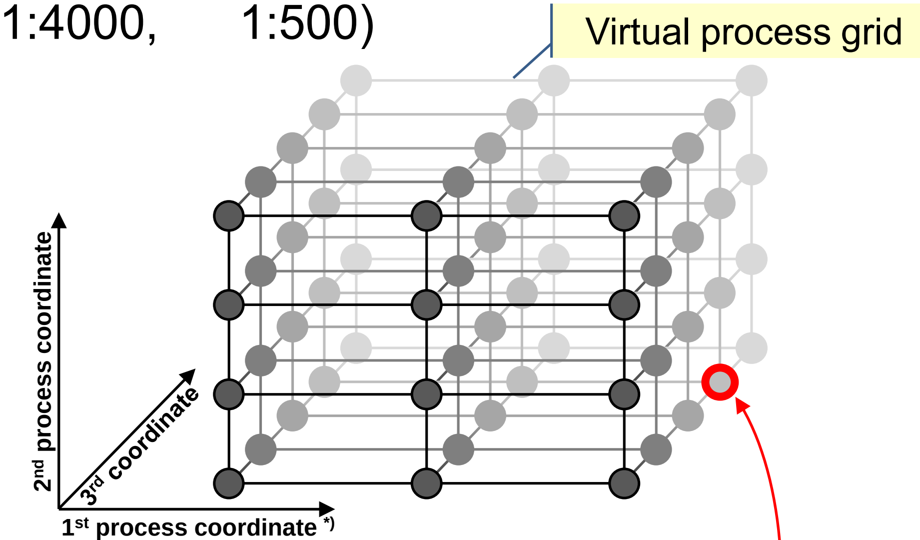
- on **3 x 4 x 5 = 60 processes**
- process coordinates **0..2, 0..3, 0..4**
- example:
  on process **ic_0=2, ic_1=0, ic_2=3 (rank=43)**
  decomposition, e.g., A(2001:3000, 1:1000, 301:400)

*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)



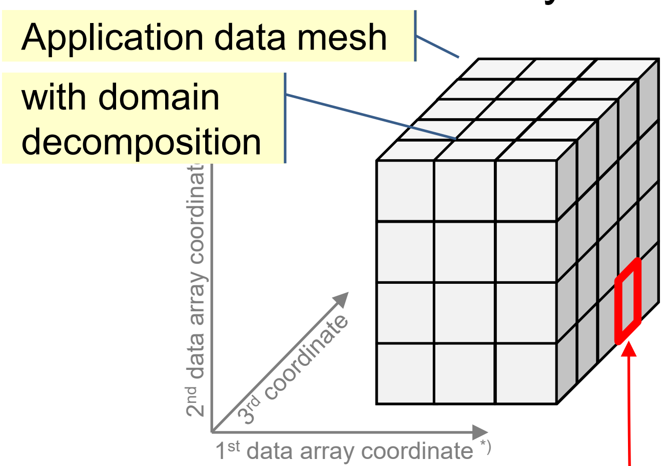


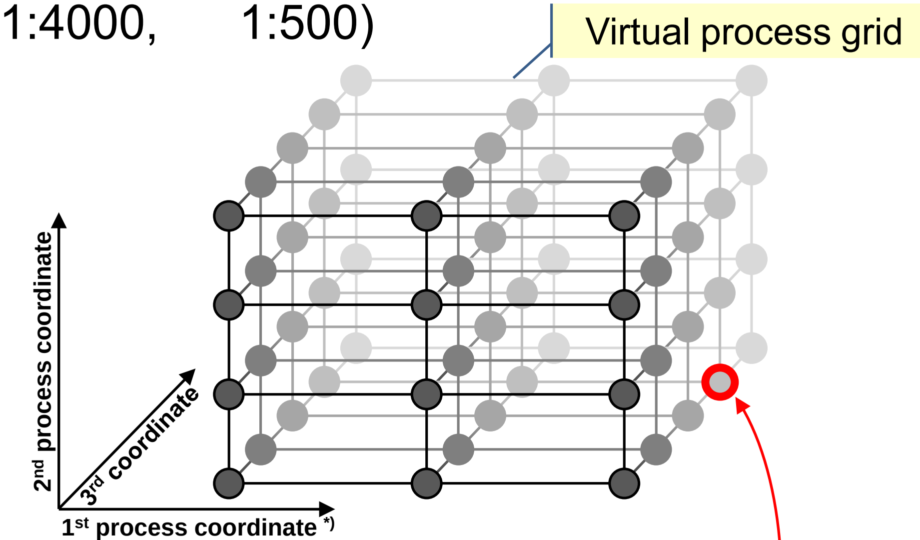
- on **3 x 4 x 5 = 60 processes**
- process coordinates **0..2, 0..3, 0..4**
- example:
  on process **ic_0=2, ic_1=0, ic_2=3 (rank=43)**
  decomposition, e.g., A(2001:3000, 1:1000, 301:400)
- **process coordinates:** handled with **virtual Cartesian topologies**

*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)





- on **3 x 4 x 5 = 60 processes**
- process coordinates **0..2, 0..3, 0..4**
- example:
  on process **ic_0=2, ic_1=0, ic_2=3 (rank=43)**
  decomposition, e.g., A(2001:3000, 1:1000, 301:400)
- **process coordinates:** handled with **virtual Cartesian topologies**
- array decomposition: handled by the application program directly

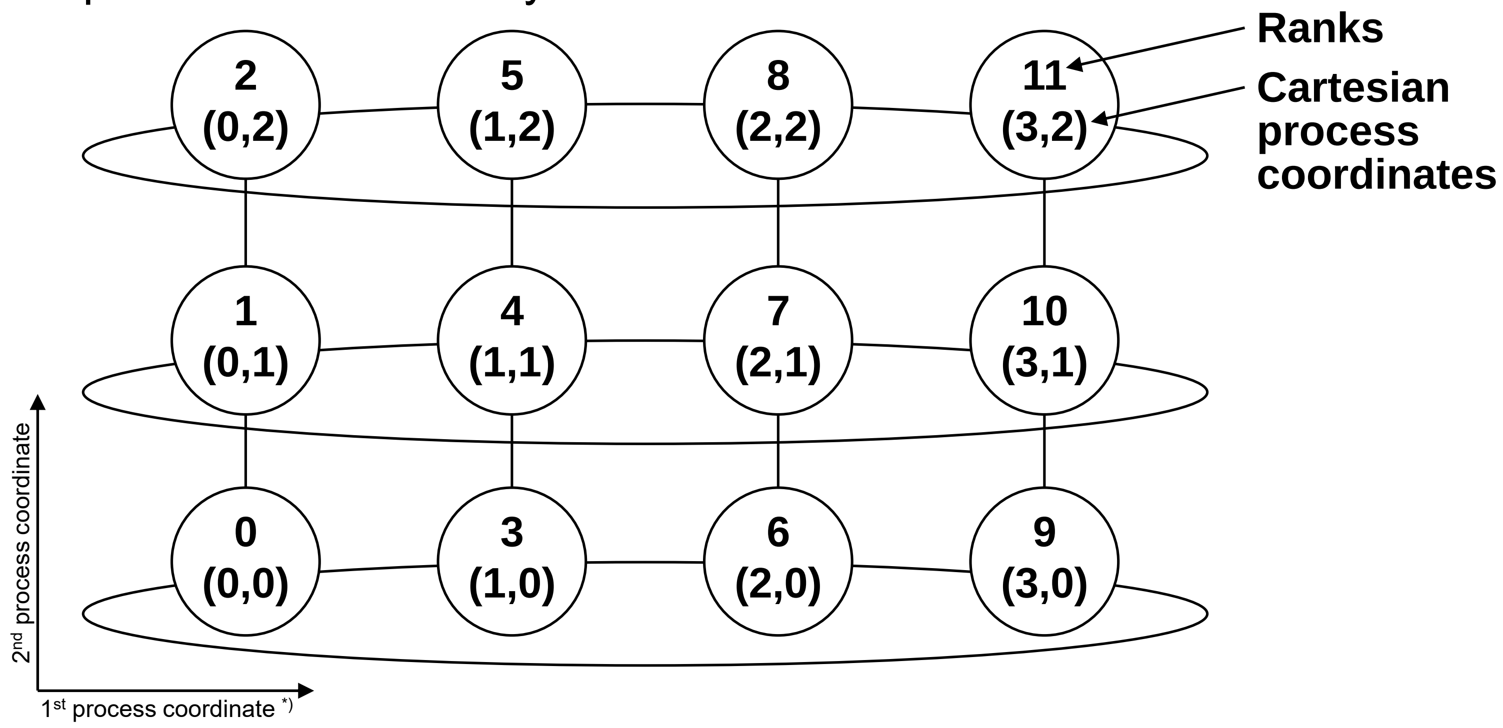
*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

Virtual Topologies

- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimize communications → see course Chapter 9-(3)

How to use a Virtual Topology

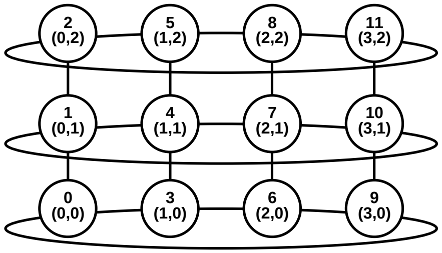
- Creating a topology produces a new communicator.
- MPI provides mapping functions:
  - to compute process ranks, based on the topology naming scheme,
  - and vice versa.
- Example: 2-dimensional cylinder



*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

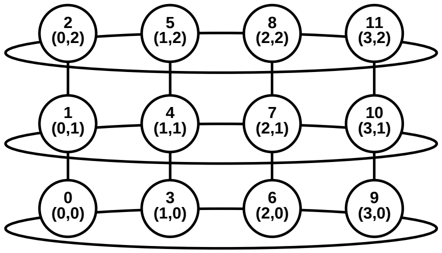
Topology Types

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



Topology Types

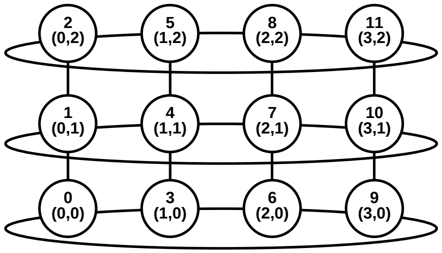
- Cartesian Topologies
  - each process is *connected* to its neighbor in a virtual process grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course, communication between any two processes is still allowed.



- Graph Topologies
  - general graphs,

Topology Types

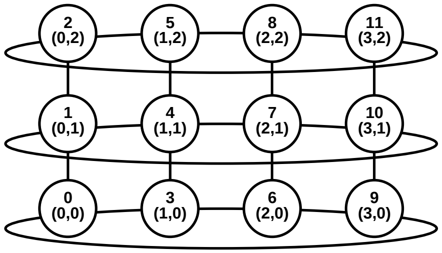
- Cartesian Topologies
  - each process is *connected* to its neighbor in a virtual process grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course, communication between any two processes is still allowed.



- Graph Topologies
  - general graphs,
  - two interfaces:
    - **MPI_Graph_create** (since MPI-1)
    - **MPI_Dist_graph_create_adjacent &**
      MPI_Dist_graph_create (new scalable interface since MPI-2.2)

Topology Types

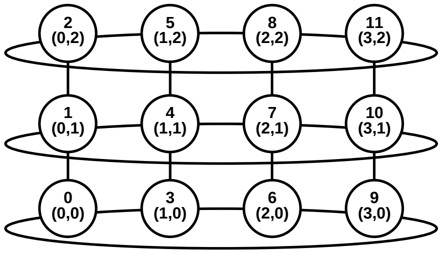
- Cartesian Topologies
  - each process is *connected* to its neighbor in a virtual process grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course, communication between any two processes is still allowed.



- Graph Topologies
  - general graphs,
  - two interfaces:
    - **MPI_Graph_create** (since MPI-1)
    - **MPI_Dist_graph_create_adjacent &**
      MPI_Dist_graph_create (new scalable interface since MPI-2.2)
  - not covered here.

Topology Types

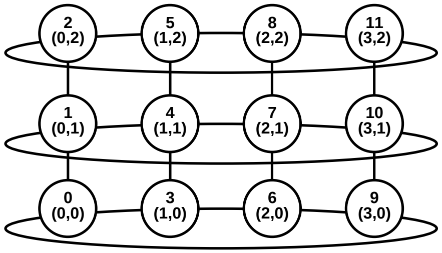
- Cartesian Topologies
  - each process is *connected* to its neighbor in a virtual process grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course, communication between any two processes is still allowed.



- Graph Topologies
  - general graphs,
  - two interfaces:
    - **MPI_Graph_create** (since MPI-1)
    - **MPI_Dist_graph_create_adjacent &**
      MPI_Dist_graph_create (new scalable interface since MPI-2.2)
  - not covered here.

Topology Types

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



- Graph Topologies
  - general graphs,
  - two interfaces:
    - **MPI_Graph_create** (since MPI-1)
    - **MPI_Dist_graph_create_adjacent &**
      MPI_Dist_graph_create (new scalable interface since MPI-2.2)
  - not covered here.

    See also T. Hoefler and M. Snir. 2011. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*. ACM, 75–85.

Creating a Cartesian Virtual Topology

C

Fortran

Python

- C/C++: int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)
- Fortran: MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, *comm_cart*, *ierror*)

mpi_f08:

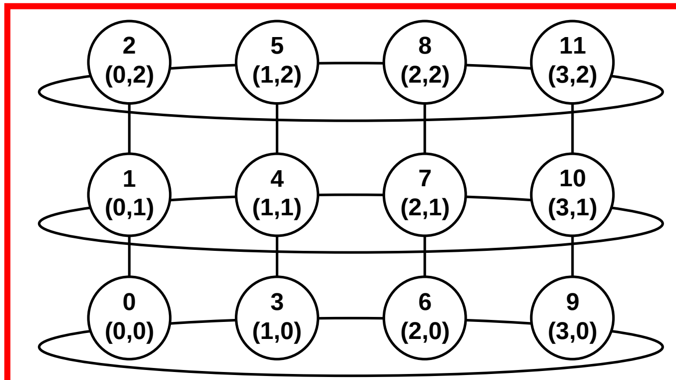
```
TYPE(MPI_Comm)          :: comm_old, comm_cart
INTEGER                 :: ndims, dims(*)
LOGICAL                 :: periods(*), reorder
INTEGER, OPTIONAL       :: ierror
```

mpi & mpif.h: INTEGER comm_old, ndims, dims(*), comm_cart, ierror ; LOGICAL periods(*), reorder

- Python: *comm_cart* = comm_old.Create_cart(dims, periods, reorder)

see mpi4py.MPI.Intracomm — MPI for Python 3.1.1 documentation

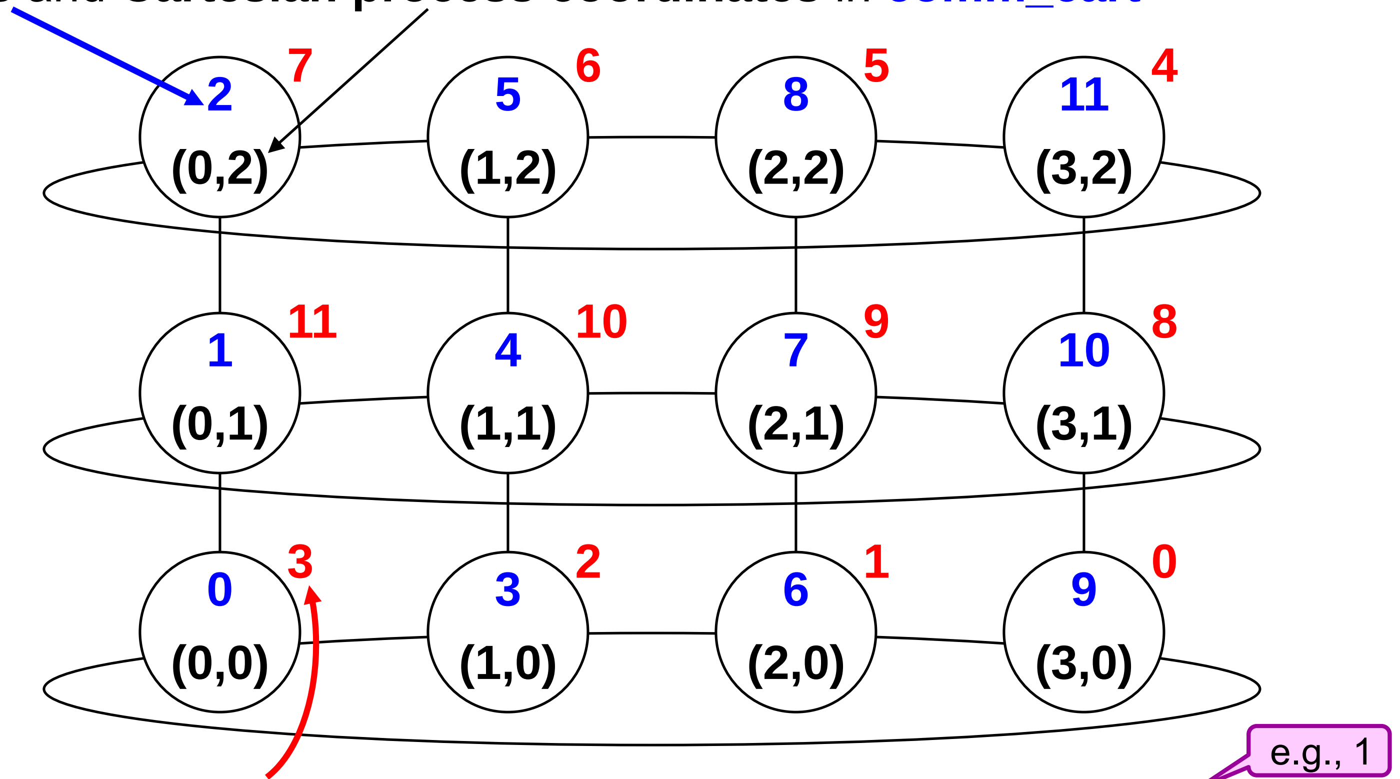
comm_old	**=**	**MPI_COMM_WORLD**		
ndims	**=**	**2**		
dims	**= (**	**4,**	**3**	**)**
periods	**= (**	**1,**	**0**	**)** (in C)
periods	**= (**	**.true.,**	**.false.**	**)** (in Fortran)
reorder	**=**	***see next slide***		



e.g., size==12 factorized with MPI_Dims_create(), see later the slide „**Typical usage of MPI_Cart_create & MPI_Dims_create**" and the advanced exercise 1b

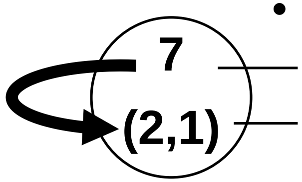
Example – A 2-dimensional Cylinder

- **Ranks** and **Cartesian process coordinates** in **comm_cart**



- Ranks in **comm** and **comm_cart** may differ if **reorder == non-zero** or **.TRUE.**
- This reordering can allow MPI to optimize communications

Cartesian Mapping Functions



- Mapping ranks to virtual process grid coordinates

C

- C/C++: int MPI_Cart_coords(MPI_Comm comm_cart, int rank,
  int maxdims, int **coords*)

Fortran

- Fortran: MPI_CART_COORDS(comm_cart, rank, maxdims, *coords*, *ierror)*

  mpi_f08:
  ```
  TYPE(MPI_Comm)          :: comm_cart
  INTEGER                 :: rank, maxdims, coords(*)
  INTEGER, OPTIONAL       :: ierror
  ```
  mpi & mpif.h:
  ```
  INTEGER comm_cart, rank, maxdims, coords(*), ierror
  ```

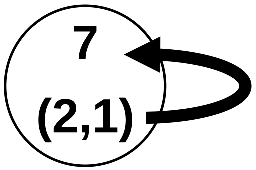
Python

- Python: *coords* = comm_cart.Get_coords(rank)

  see mpi4py.MPI.Cartcomm — MPI for Python 3.1.1 documentation

Cartesian Mapping Functions

- Mapping virtual process grid coordinates to ranks



C

- C/C++: int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int **rank*)

Fortran

- Fortran: MPI_CART_RANK(comm_cart, coords, *rank*, *ierror*)

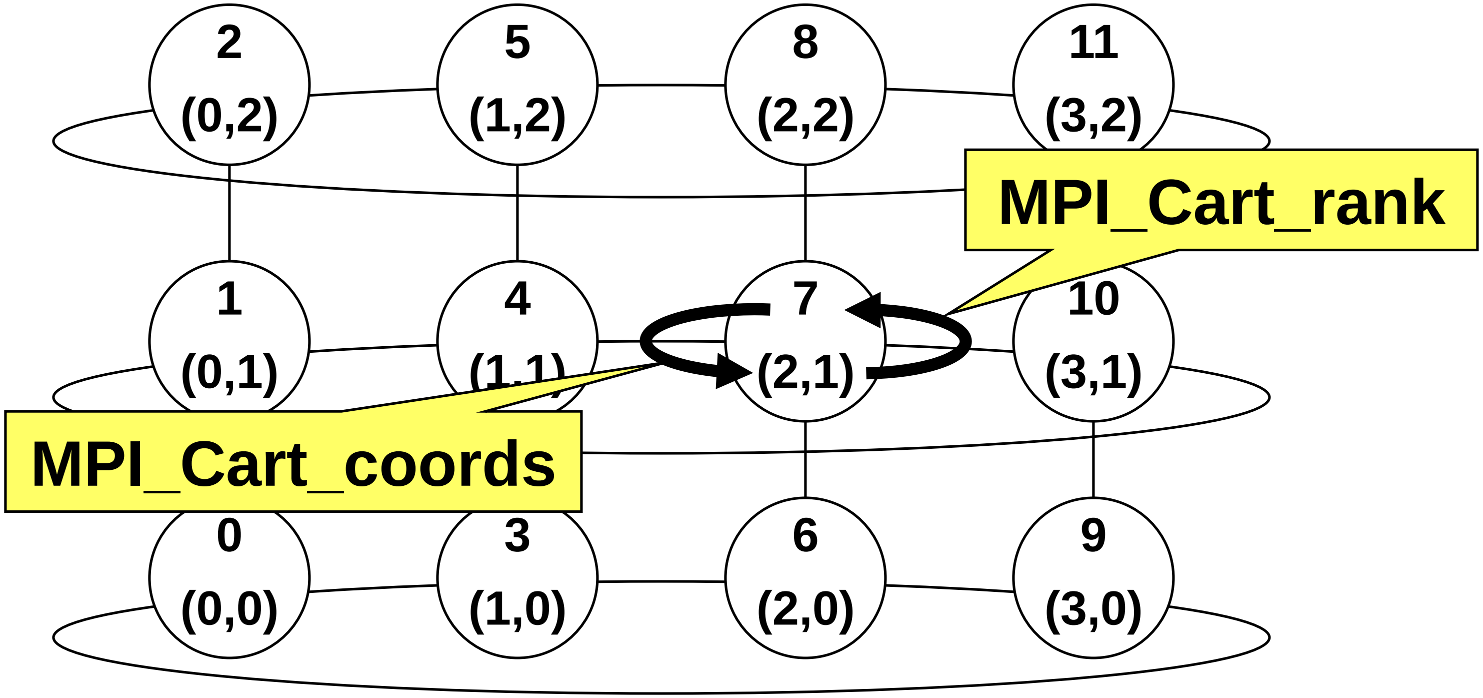
mpi_f08:	TYPE(MPI_Comm)	:: comm_cart
	INTEGER	:: coords(*), rank
	INTEGER, OPTIONAL	:: ierror
mpi & mpif.h:	INTEGER comm_cart, coords(*), rank, ierror	

Python

- Python: *rank* = comm_cart.Get_cart_rank(coords)

see mpi4py.MPI.Cartcomm — MPI for Python 3.1.1 documentation

Own coordinates



- Each process gets its own coordinates with (example in **Fortran**)

CALL MPI_Comm_rank(comm_cart, ***my_rank***, *ierror*)

CALL MPI_Cart_coords(comm_cart, **my_rank**, maxdims, ***my_coords***, *ierror*)

Typical usage of MPI_Cart_create & MPI_Dims_create

```
#define ndims 3
int i, nnodes, world_myrank, cart_myrank, dims[ndims], periods[ndims], my_coords[ndims];  MPI_Comm  comm_cart;
MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &world_myrank);
for (i=0; i<ndims; i++) { dims[i]=0; periods[i]=…; }
MPI_Dims_create(numprocs, ndims, dims); // computes factorization of numprocs
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods,1, &comm_cart);
MPI_Comm_rank(comm_cart, &cart_myrank);
MPI_Cart_coords(comm_cart, cart_myrank, ndims, my_coords)
```

with reorder

From now on, all communication should be based on **comm_cart + cart_myrank + my_coords** and **one can setup the sub-domains & read in the application data**

C

- C/C++: int MPI_Dims_create(int nnodes, int ndims, int **dims*)

Fortran

- Fortran: MPI_DIMS_CREATE(nnodes, ndims, *dims*, *IERROR*)

mpi_f08: INTEGER :: nnodes, ndims, dims(*)
INTEGER, OPTIONAL :: ierror

mpi & mpif.h: INTEGER nnodes, ndims, dims(*), ierror

Array *dims* must be **initialized** with **zeros**
(other possibilities, see MPI standard)

Python

- Python: *dims_out* = MPI.Compute_dims(nnodes, dims)

See mpi4py.MPI.Compute_dims — MPI for Python 3.1.1 documentation

Exercise 1 — One-dimensional ring topology

Exercise 1

In MPI/tasks/…

- Use a one-dimensional virtual Cartesian topology in the pass-around-the-ring program:

  Add a call to **MPI_Cart_create**, of course with reorder == non-zero (e.g., 1) or .TRUE. or True

- Use C C/Ch9/cart-create-skel.c or Fortran F_30/Ch9/cart-create-skel_30.f90

  or Python PY/Ch9/cart-create-skel.py

- **Caution**: Do only the prepared **one-dimensional virtual Cartesian topology**

Exercise 1 — One-dimensional ring topology

- Use a one-dimensional virtual Cartesian topology in the pass-around-the-ring program:

  Add a call to **MPI_Cart_create**, of course with reorder == non-zero (e.g., 1) or .TRUE. or True

In MPI/tasks/…

- Use C C/Ch9/cart-create-skel.c or Fortran F_30/Ch9/cart-create-skel_30.f90

  or Python PY/Ch9/cart-create-skel.py

- **Caution**: Do only the prepared **one-dimensional virtual Cartesian topology**

- Hints:
  - After calling MPI_Cart_create,
    - there should be no further usage of MPI_COMM_WORLD, and
    - the my_rank must be recomputed on the base of comm_cart.

Exercise 1 — One-dimensional ring topology

Exercise 1

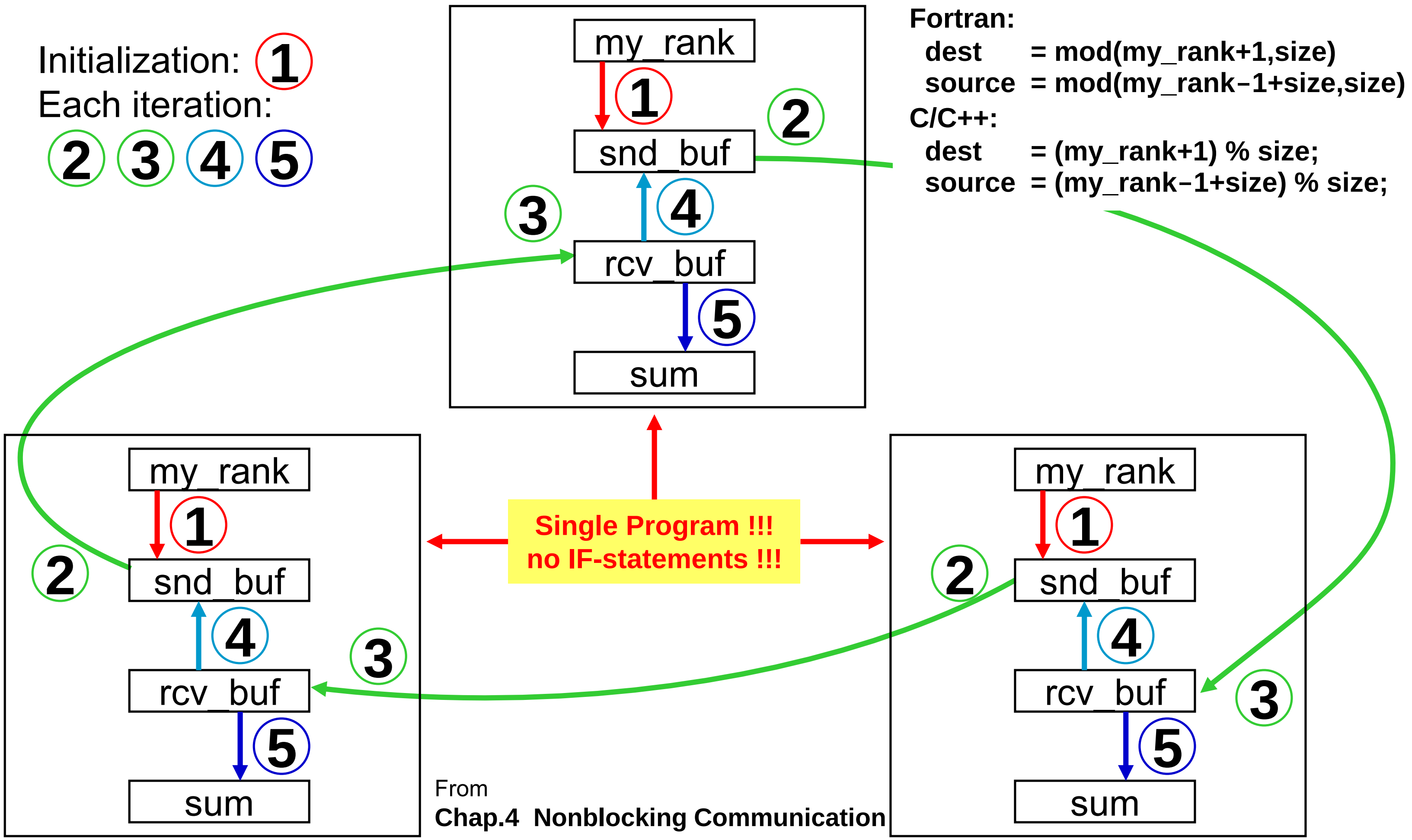
- Use a one-dimensional virtual Cartesian topology in the pass-around-the-ring program:
  Add a call to **MPI_Cart_create**, of course with reorder == non-zero (e.g., 1) or .TRUE. or True

In MPI/tasks/…

- Use C C/Ch9/cart-create-skel.c or Fortran F_30/Ch9/cart-create-skel_30.f90
  or Python PY/Ch9/cart-create-skel.py
- **Caution**: Do only the prepared **one-dimensional virtual Cartesian topology**
- Hints:
  - After calling MPI_Cart_create,
    - there should be no further usage of MPI_COMM_WORLD, and
    - the my_rank must be recomputed on the base of comm_cart.
  - Only **one**-dimensional:
    - → coordinates are not necessary, because coord==rank

    In this exercise not relevant, because the skeleton already uses arrays:
    - → In C: `dims` and `period` as normal variables, i.e., no arrays, but call by reference with &dims, …
    - → In Fortran: `dims` and `period` must be arrays (i.e., with only 1 element, e.g., `(/.TRUE./)`)

Slide from Chap. 4 — Rotating information around a ring



Slide from Chap. 4 — Rotating information around a ring

Fortran:

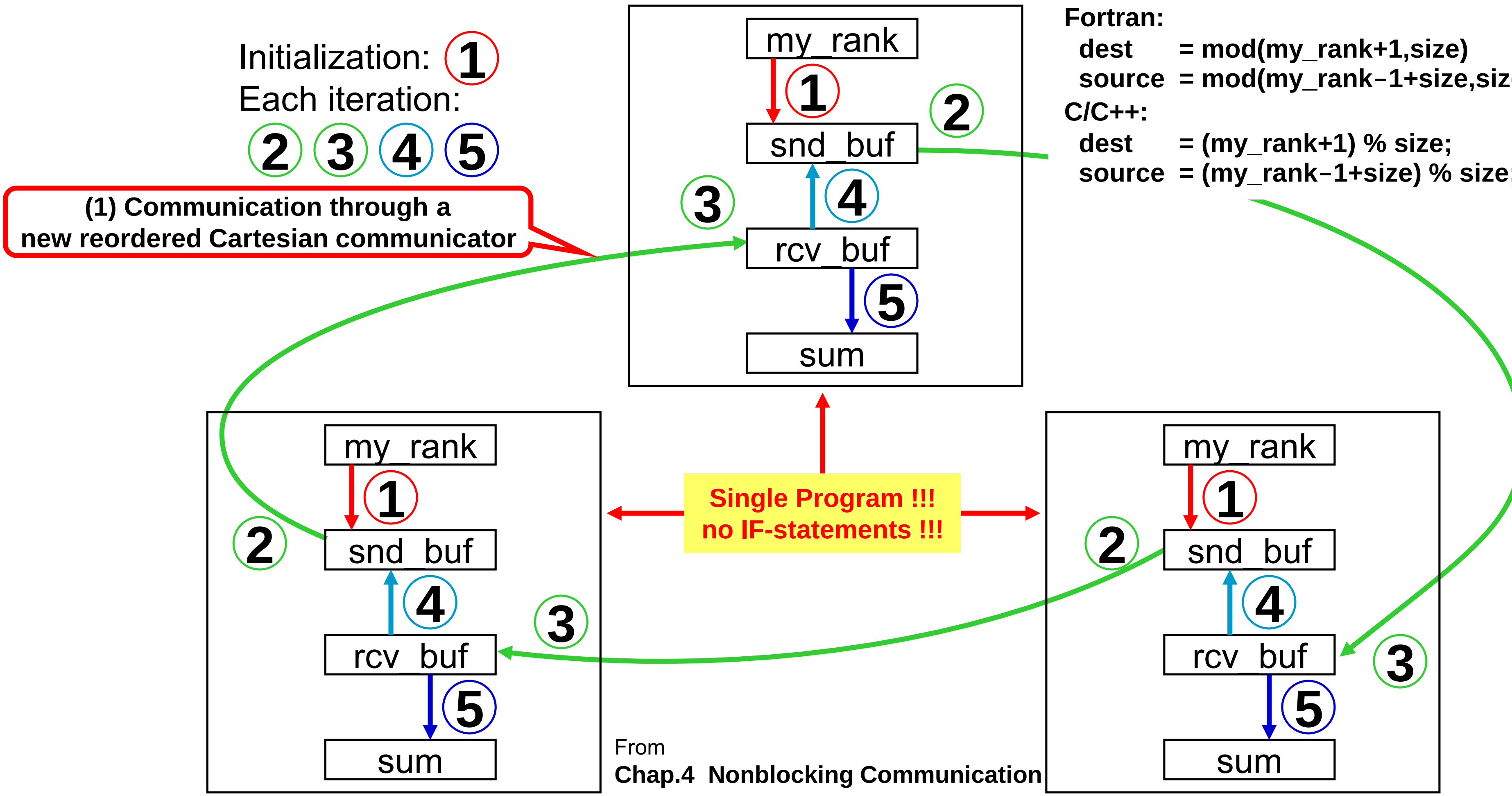
dest = mod(my_rank+1,size)

source = mod(my_rank−1+size,size)

C/C++:

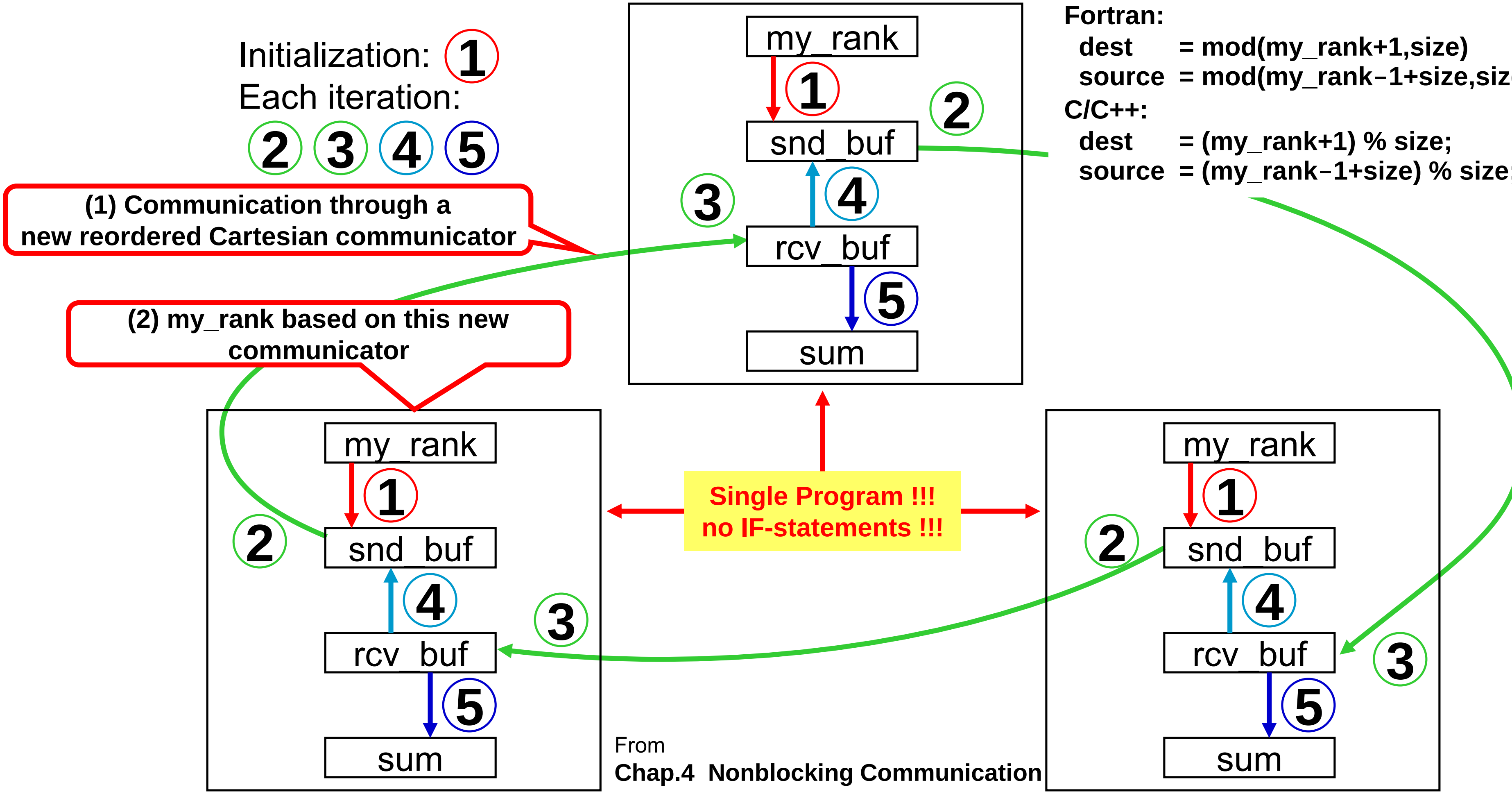
dest = (my_rank+1) % size;

source = (my_rank−1+size) % size;



From
Chap.4 Nonblocking Communication

Slide from Chap. 4 — Rotating information around a ring



Fortran:

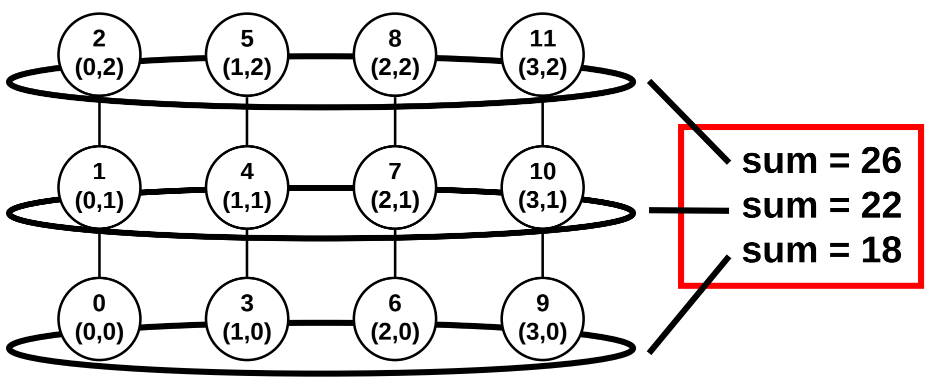
dest = mod(my_rank+1,size)
source = mod(my_rank–1+size,size)

C/C++:

dest = (my_rank+1) % size;
source = (my_rank–1+size) % size;

Advanced Exercise 1b — Two-dimensional topology

- Task: Rewrite the exercise in two dimensions, as a cylinder.
  - Each row of the cylinder, i.e. each ring, should compute its own separate sum of the original ranks in the two dimensional comm_cart.
  - Compute the two dimensional factorization with MPI_Dims_create().
  - Array *dims* must be **initialized** with **(0,0) !**
  - Execute the ring algorithm in direction 0, i.e., communicating only to its left and right neighbors.
  - Calculate the neighbor ranks `left` and `right` using MPI_Cart_rank().
- Use **C** C/Ch9/cylinder-skel.c or **Fortran** F_30/Ch9/cylinder-skel_30.f90 or **Python** PY/Ch9/cylinder-skel.py
- Run with `mpirun -np 12 ./a.out | sed -e 's/PE//' | sort`



Cartesian Mapping Functions

- Computing ranks of neighboring processes

C

- C/C++: int MPI_Cart_shift(MPI_Comm comm_cart, int direction, int disp, int *rank_source, int *rank_dest)

Fortran

- Fortran: MPI_CART_SHIFT(comm_cart, direction, disp, rank_source, rank_dest, ierror)

mpi_f08: TYPE(MPI_Comm) :: comm_cart
INTEGER :: direction, disp, rank_source, rank_dest
INTEGER, OPTIONAL :: ierror

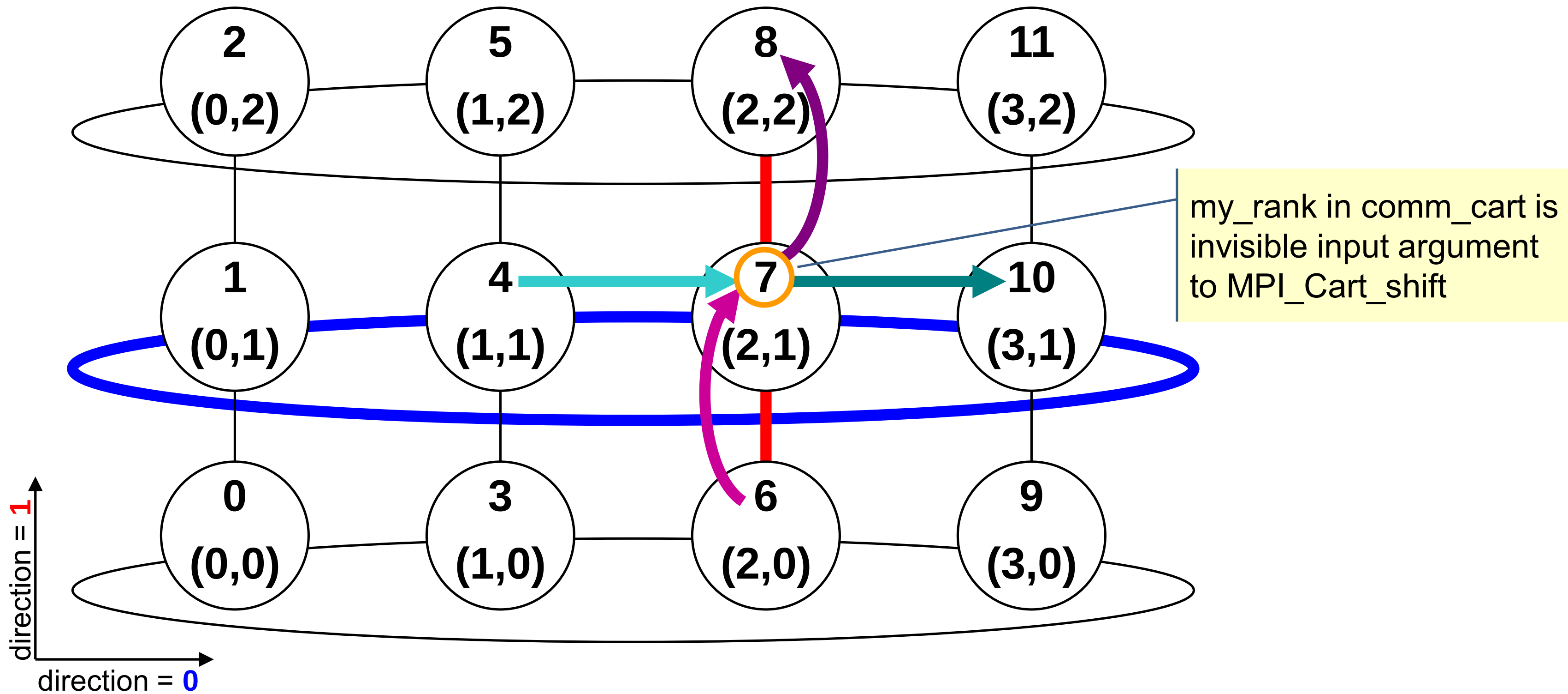
mpi & mpif.h: INTEGER comm_cart, direction, disp, rank_source, rank_dest, ierror

Python

- Python: (rank_source, rank_dest) = comm_cart.Shift(direction, disp)

- Returns MPI_PROC_NULL if there is no neighbor.
- MPI_PROC_NULL can be used as source or destination rank in each communication ➔ Then, this communication will be a no-operation!

MPI_Cart_shift – Example



CALL MPI_Cart_shift (comm_cart, direction, disp, *rank_source*, *rank_dest*, *ierror*)

	direction	disp	rank_source	rank_dest
example on	0 or	+1	4	10
process rank=7	1	+1	6	8

Cartesian Partitioning

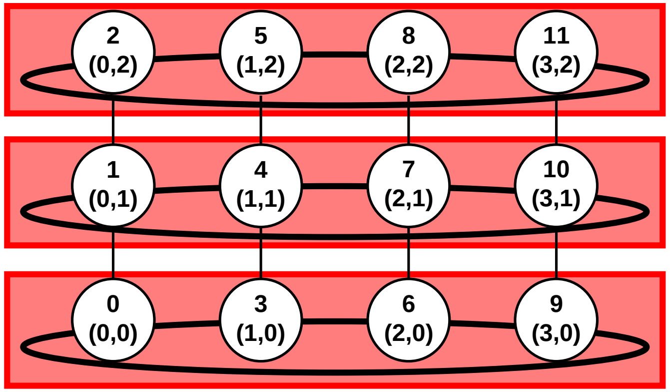
- Cut a virtual process grid up into *slices*.
- A new communicator is produced for each slice.
- Each slice can then perform its own collective communications.

C

- C/C++: int MPI_Cart_sub(MPI_Comm comm_cart, int *remain_dims,
  MPI_Comm **comm_slice*)

Fortran

- Fortran: MPI_CART_SUB(comm_cart, remain_dims, *comm_slice*, *ierror*)



mpi_f08:
TYPE(MPI_Comm) :: comm_cart
LOGICAL :: remain_dims(*)
TYPE(MPI_Comm) :: comm_slice
INTEGER, OPTIONAL :: ierror

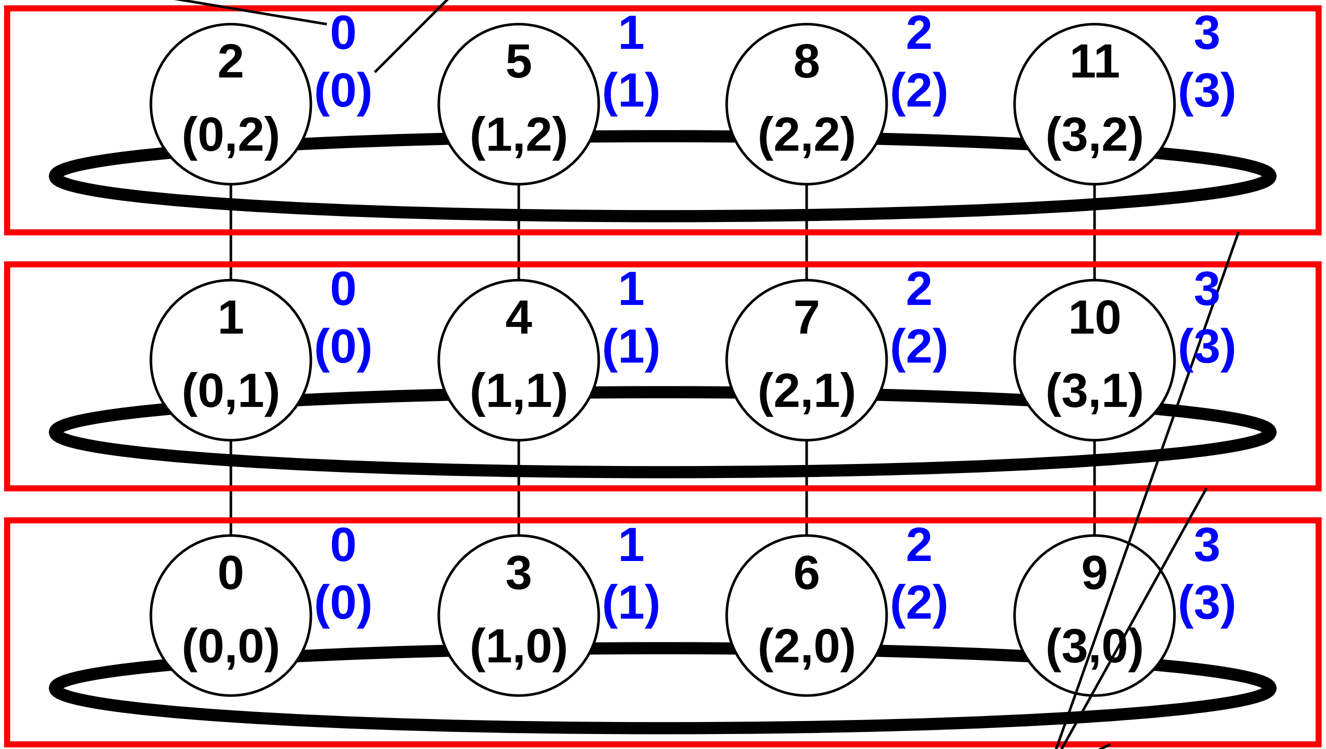
mpi & mpif.h: INTEGER comm_cart, comm_slice, ierror
LOGICAL remain_dims(*)

Python

- Python: *comm_slice* = comm_cart.Sub(remain_dims)

MPI_Cart_sub – Example

- **Ranks** and **Cartesian process coordinates** in **comm_slice**



- CALL MPI_Cart_sub(comm_cart, remain_dims, ***comm_slice***, *ierror*)

(true, false)

Each process gets only its own sub-communicator

Four slides with **general remarks** before next exercise

Multidimensional domain decomposition

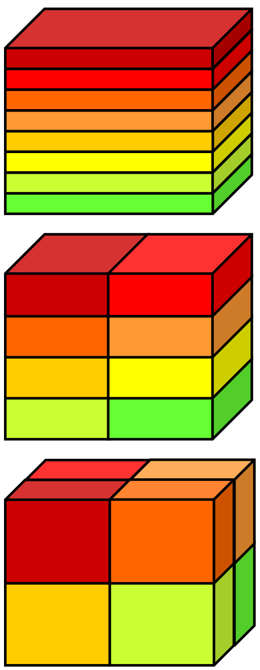
- Applications with 3 dimensions
  - each sub-domain (computed by one MPI process) should
  - have the same size → optimal load balance
  - minimal surface → minimal communication
  - Usually optimum with **3-dim. domain decomposition** & **cubic** sub-domains[1)]
- Same rule for 2 dimensional application → 2-D domain decomposition & quadratic[1)] sub-domains

[1)] "cubic" and "quadratic" may be qualified due to different communication bandwidth in each direction caused by sending (fast) non-strided or (slow) strided data

Four slides with **general remarks** before next exercise

Multidimensional domain decomposition

- Applications with 3 dimensions
  - each sub-domain (computed by one MPI process) should
  - have the same size → optimal load balance
  - minimal surface → minimal communication
  - Usually optimum with **3-dim. domain decomposition** & **cubic** sub-domains[1)]
- Same rule for 2 dimensional application → 2-D domain decomposition & quadratic[1)] sub-domains



Splitting in

- one dimension:
  communication
  $= n^2 * 2 * w * 1 / p^0$
- two dimensions:
  communication
  $= n^2 * 2 * w * 2 / p^{1/2}$
- three dimensions:
  communication
  $= n^2 * 2 * w * 3 / p^{2/3}$

optimal for p≥12

w = width of halo
n^3 = size of matrix
p = number of processes
cyclic boundary
—> two neighbors
in each direction

[1)] "cubic" and "quadratic" may be qualified due to different communication bandwidth in each direction caused by sending (fast) non-strided or (slow) strided data

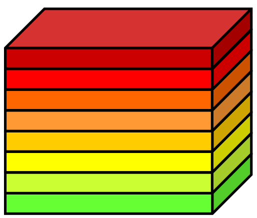
Four slides with **general remarks** before next exercise

Multidimensional domain decomposition

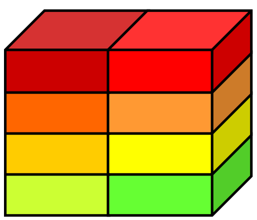
- Applications with 3 dimensions
  - each sub-domain (computed by one MPI process) should
  - have the same size → optimal load balance
  - minimal surface → minimal communication
  - Usually optimum with **3-dim. domain decomposition** & **cubic** sub-domains[1)]
- Same rule for 2 dimensional application → 2-D domain decomposition & quadratic[1)] sub-domains

Exception: The total domain has extremely different dimensions, e.g., weather/climate: **40,000 km x 40,000 km x 15 km** (→ **only 2-dim** domain decomp.)

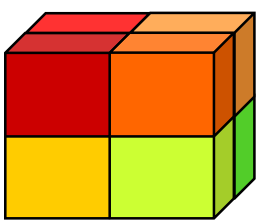
Splitting in



- one dimension: communication $= n^2 * 2 * w * 1 / p^0$



- two dimensions: communication $= n^2 * 2 * w * 2 / p^{1/2}$



- three dimensions: communication $= n^2 * 2 * w * 3 / p^{2/3}$

optimal for p≥12

w = width of halo
n^3 = size of matrix
p = number of processes
cyclic boundary
—> two neighbors in each direction

1) "cubic" and "quadratic" may be qualified due to different communication bandwidth in each direction caused by sending (fast) non-strided or (slow) strided data

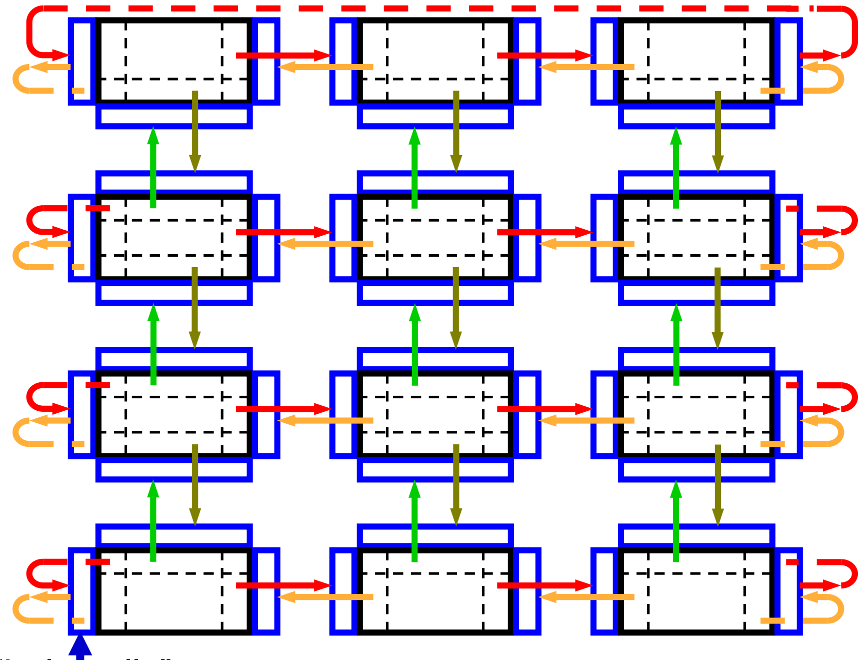
General rule:
Symmetric vs. asymmetric manager/worker parallelization

- We know this example already from course chapter 1

One iteration in the

- **serial** code:
  - x_{new} = function(x_{old})
  - x_{old} = x_{new}
- **parallel** code:
  - Update halo
    [=Communication, e.g., with 4 x MPI_Sendrecv]
  - x_{new} = function(x_{old})
  - x_{old} = x_{new}

Examples with 12 sub-domains and horizontally cyclic boundary conditions
→ communication around rings

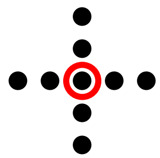


- General rule:
  - Always try to implement such a **symmetric parallelization design**
  - **Avoid** (asymmetric) **manager-worker**[1)]-paradigm
    → the manager always tend to **limit the scaling** to a larger number of processes

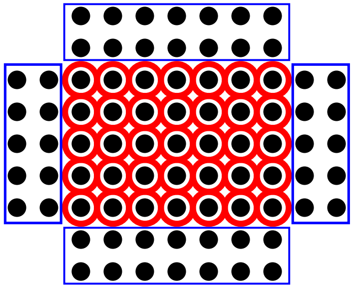
[1)] The outdated wording “*master/slave*” should be avoided

Halo

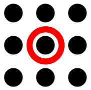
- Stencil:
  - To calculate a new data mesh point (○), old data from the stencil mesh points (•) are needed
    - **E.g., 9 point stencil**

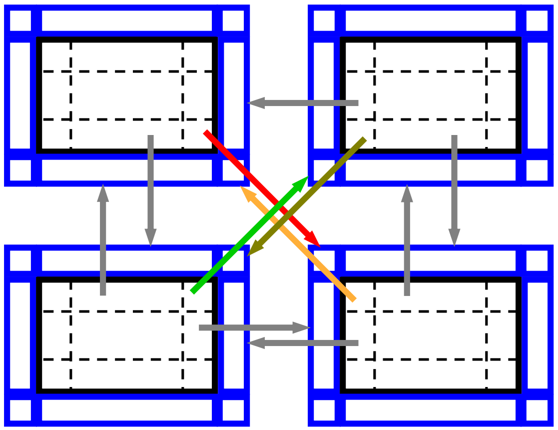


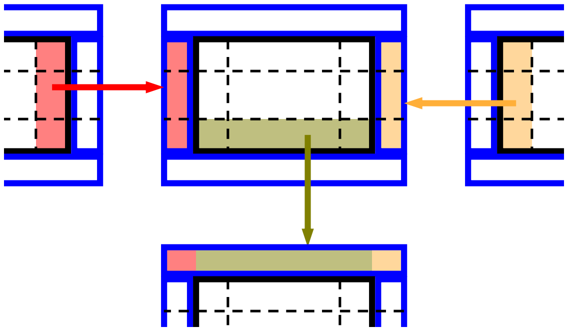
- Halo
  - To calculate the new data mesh points of a sub-domain, additional mesh points from other sub-domains are needed.
  - They are stored in halos (ghost cells, shadows)
  - Halo depends on form of stencil



Diagonals Problem

- Stencil with diagonal point, e.g., 
  - i.e., halos include corners →→→ substitute small corner messages:
    - **one may use 2-phase-protocol:**
    - **normal horizontal halo communication**
    - **include corner into vertical exchange**





Chris Ding and Yun He: A ghost cell expansion method for reducing communications in solving PDE problems. Proc. SC2001. DOI:10.1145/582034.582084

Exercise 2 — One-dimensional ring topology

- Use a one-dimensional in the pass-around-the-ring program:
  Add a call to **MPI_Cart_shift** to calculate left and right
- Use **C** C/Ch9/cart-shift-skel.c or **Fortran** F_30/Ch9/cart-shift-skel_30.f90
  or **Python** PY/Ch9/cart-shift-skel.py

Exercise 2 — One-dimensional ring topology

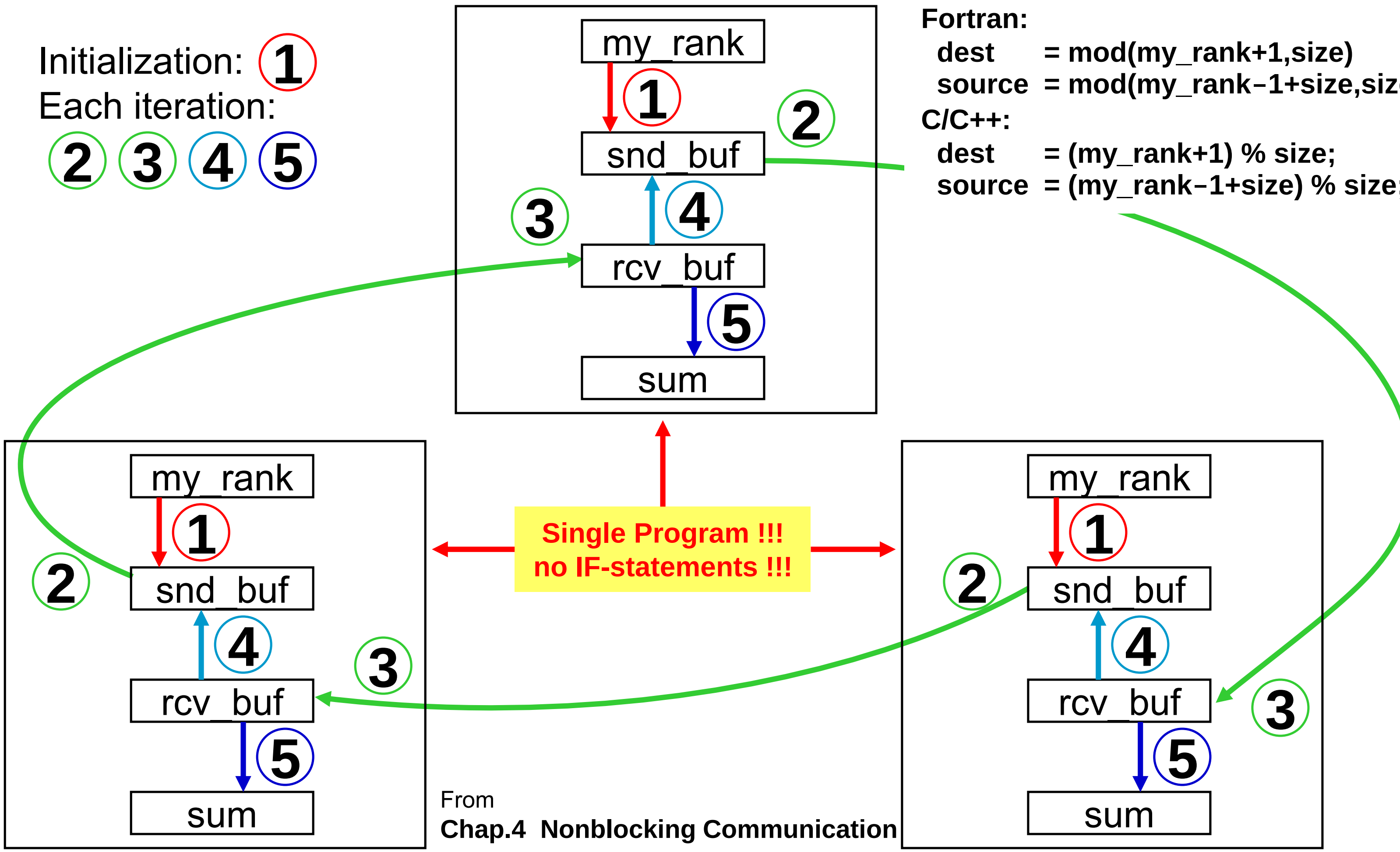
- Use a one-dimensional in the pass-around-the-ring program:
  Add a call to **MPI_Cart_shift** to calculate left and right
- Use **C** C/Ch9/cart-shift-skel.c or **Fortran** F_30/Ch9/cart-shift-skel_30.f90
  or **Python** PY/Ch9/cart-shift-skel.py
- Goal:

Exercise 2 — One-dimensional ring topology

- Use a one-dimensional in the pass-around-the-ring program:
  Add a call to **MPI_Cart_shift** to calculate left and right
- Use **C** C/Ch9/cart-shift-skel.c or **Fortran** F_30/Ch9/cart-shift-skel_30.f90
  or **Python** PY/Ch9/cart-shift-skel.py
- Goal:
  - the cryptic way to compute the neighbor ranks should be substituted by one call to MPI_Cart_shift, that should be before starting the loop.

Slide from Chap. 4 — Rotating information around a ring

Initialization: ①
Each iteration: ② ③ ④ ⑤




```
Fortran:
  dest     = mod(my_rank+1,size)
  source   = mod(my_rank–1+size,size)
C/C++:
  dest     = (my_rank+1) % size;
  source   = (my_rank–1+size) % size;
```

Slide from Chap. 4 — Rotating information around a ring

Initialization: ①
Each iteration: ② ③ ④ ⑤

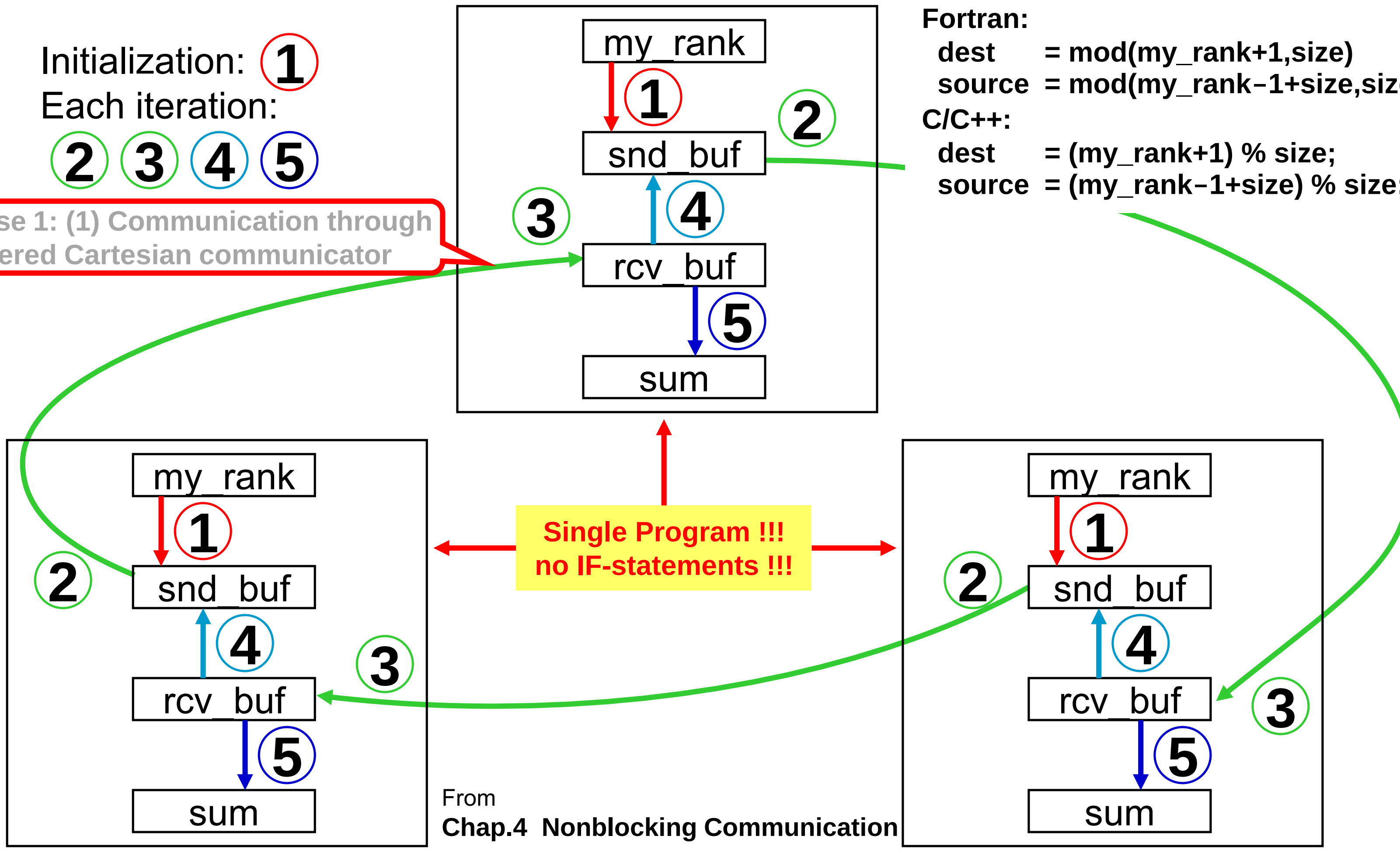
Done in Exercise 1: (1) Communication through a new reordered Cartesian communicator

Fortran:

```
dest    = mod(my_rank+1,size)
source  = mod(my_rank−1+size,size)
```

C/C++:

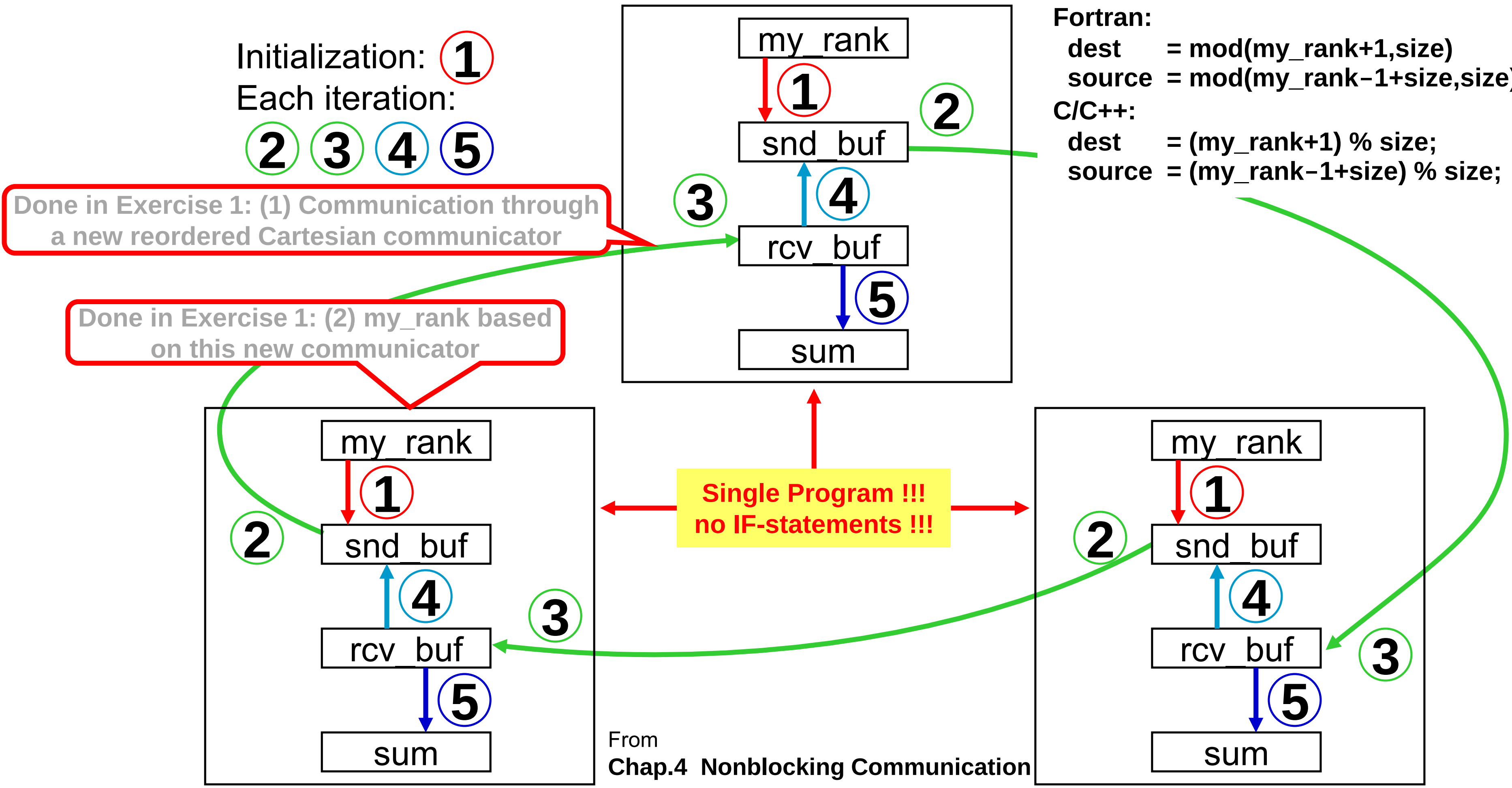
```
dest    = (my_rank+1) % size;
source  = (my_rank−1+size) % size;
```



From
Chap.4 Nonblocking Communication

Slide from Chap. 4 — Rotating information around a ring

Initialization: 1
Each iteration: 2 3 4 5

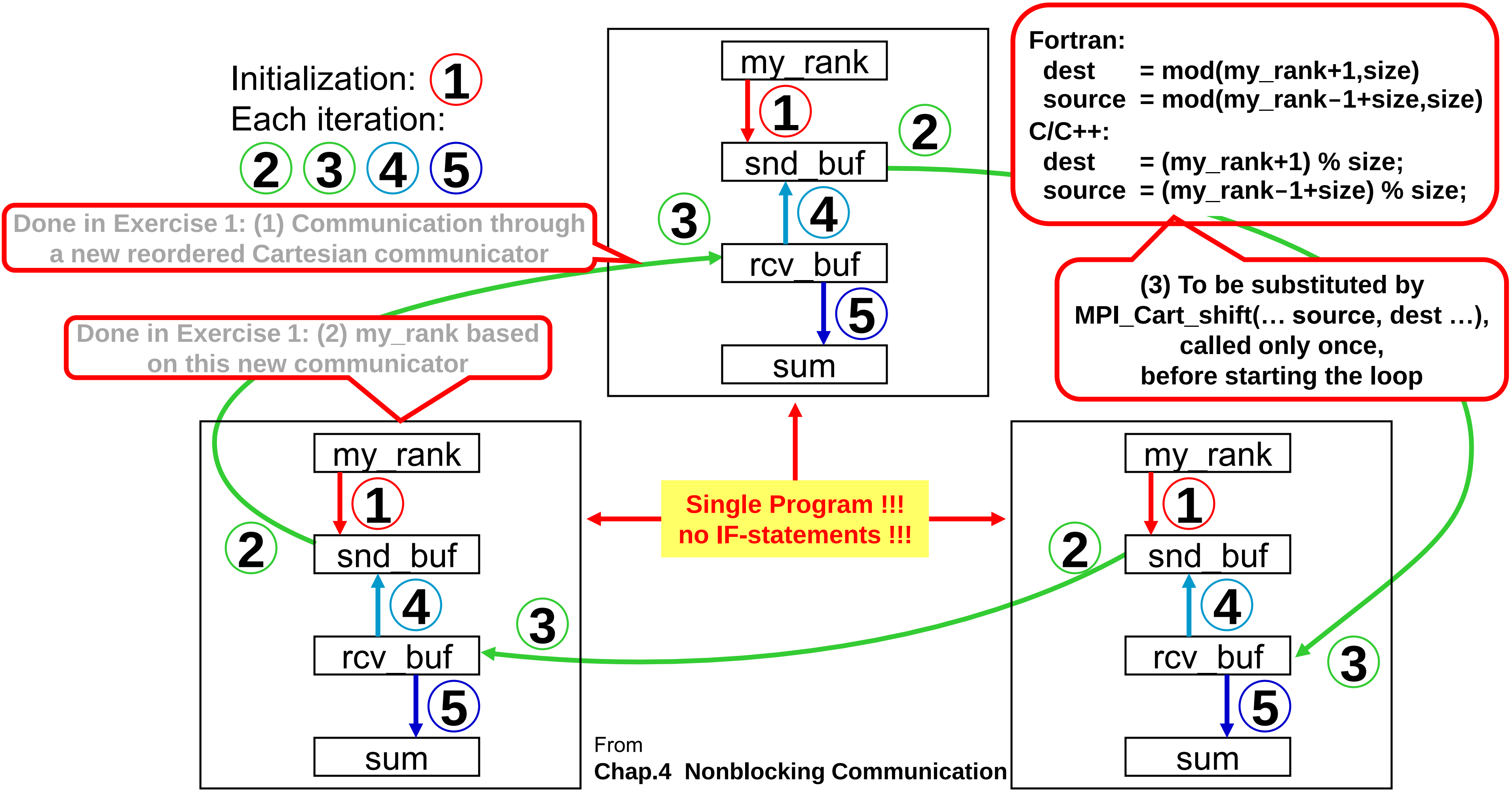


Fortran:
dest = mod(my_rank+1,size)
source = mod(my_rank−1+size,size)

C/C++:
dest = (my_rank+1) % size;
source = (my_rank−1+size) % size;

From
Chap.4 Nonblocking Communication

Slide from Chap. 4 — Rotating information around a ring



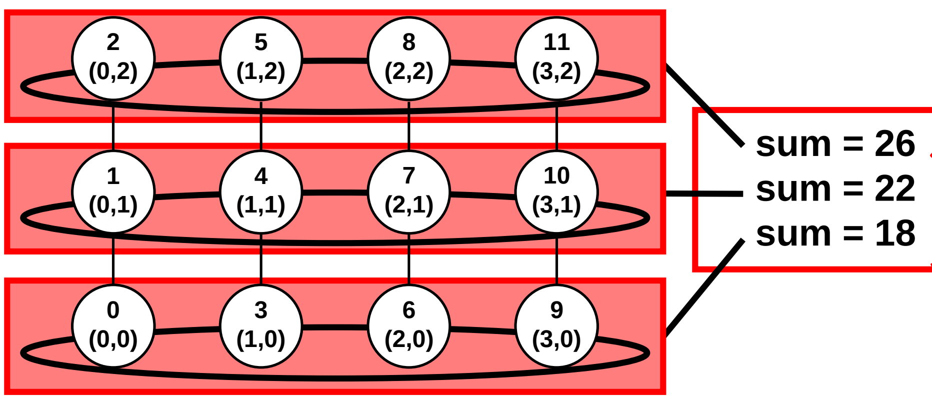
From
Chap.4 Nonblocking Communication

Exercise 3+4 (advanced) — Two-dimensional topology

C
Fortran
Python

- **Exercise 3**: Rewrite the exercise in two dimensions, as a cylinder.
  - Each row of the cylinder, i.e. each ring, should compute its own separate sum of the original ranks in the two dimensional comm_cart.
  - Task: substitute 2x MPI_Cart_rank by 1x MPI_Cart_shift
  - **Use** (your) solution of Ch.9-(1) Advanced exercise 1b:
    - Your modified **C, F_30, PY/Ch9/cylinder-skel.c, _30.f90, .py**

- **Exercise 4:** Use MPI_Cart_sub to create the one-dimensional slice communicators
  - Results are the same



Summing up the myrank of the 2-dimensional Cartesian topology:
Advanced Exercise 4a:
Ring-communication in the comm_slice, and using the ring with myrank, left, right and size of the comm_slice.

Additional Advanced Exercise 4b:
Using MPI_Allreduce within the comm_slice instead of the ring communication algorithm.