Parallel programming / computation

Sultan ALPAR

s.alpar@iitu.edu.kz

IITU

Lecture 6

Error Handling
Groups & Communicators, Environment management

Error Handling → “assembler for parallel computing”

2-level-concept with **error codes** and **error classes**, see MPI-3.1/MPI-4.0 Sect. 8.3-5/9.3-5

Error Handling → “assembler for parallel computing”

2-level-concept with **error codes** and **error classes**, see MPI-3.1/MPI-4.0 Sect. 8.3-5/9.3-5

Most important aspects:

- The communication should be reliable (same rule as for processor and memory)

Error Handling → “assembler for parallel computing”

2-level-concept with **error codes** and **error classes**, see MPI-3.1/MPI-4.0 Sect. 8.3-5/9.3-5

Most important aspects:

- The communication should be reliable (same rule as for processor and memory)
- If the MPI program is erroneous → no warranties:
  - by default: abort, if error detected by MPI library
    otherwise, unpredictable behavior

i.e., error handler MPI_ERRORS_ARE_FATAL is the default

Error Handling → "assembler for parallel computing"

2-level-concept with **error codes** and **error classes**, see MPI-3.1/MPI-4.0 Sect. 8.3-5/9.3-5

Most important aspects:

- The communication should be reliable (same rule as for processor and memory)
- If the MPI program is erroneous → no warranties:
  - by default: abort, if error detected by MPI library
    otherwise, unpredictable behavior

    i.e., error handler MPI_ERRORS_ARE_FATAL is the default
  - C/C++: MPI_Comm_set_errhandler (comm, MPI_ERRORS_RETURN);
    Fortran: call MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN, ierr)
    Python: comm.Set_errhandler(MPI.ERRORS_RETURN)

    Newly added in MPI-4.0
  - directly after MPI_INIT with both comm = MPI_COMM_WORLD and MPI_COMM_SELF, then
    - **ierror returned by each MPI routine (except MPI window and MPI file routines)**
    - **undefined state after an erroneous MPI call has occurred**
      (only MPI_Abort(…) should be still callable)

Error Handling → "assembler for parallel computing"

2-level-concept with **error codes** and **error classes**, see MPI-3.1/MPI-4.0 Sect. 8.3-5/9.3-5

Most important aspects:

- The communication should be reliable (same rule as for processor and memory)
- If the MPI program is erroneous → no warranties:
  - by default: abort, if error detected by MPI library
    otherwise, unpredictable behavior

    i.e., error handler MPI_ERRORS_ARE_FATAL is the default
  - C/C++: MPI_Comm_set_errhandler (comm, MPI_ERRORS_RETURN);
    Fortran: call MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN, ierr)
    Python: comm.Set_errhandler(MPI.ERRORS_RETURN)

    Newly added in MPI-4.0
  - directly after MPI_INIT with both comm = MPI_COMM_WORLD and MPI_COMM_SELF, then
    - **ierror returned by each MPI routine (except MPI window and MPI file routines)**
    - **undefined state after an erroneous MPI call has occurred**
      (only MPI_Abort(…) should be still callable)
  - Exception: MPI-I/O has default MPI_ERRORS_RETURN
    - Default can be changed through MPI_FILE_NULL:
    - MPI_File_set_errhandler (MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL)
      Python: MPI.FILE_NULL.Set_errhandler(MPI.ERRORS_ARE_FATAL)
    - See MPI-3.1 Sect. 13.7, page 555 / MPI-4.0 Sect. 14.7, page 719, and course Chapter 7

Error Handling → "assembler for parallel computing"

2-level-concept with **error codes** and **error classes**, see MPI-3.1/MPI-4.0 Sect. 8.3-5/9.3-5

Most important aspects:

- The communication should be reliable (same rule as for processor and memory)
- If the MPI program is erroneous → no warranties:
  - by default: abort, if error detected by MPI library
    otherwise, unpredictable behavior

    i.e., error handler MPI_ERRORS_ARE_FATAL is the default
  - C/C++: MPI_Comm_set_errhandler (comm, MPI_ERRORS_RETURN);
    Fortran: call MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN, ierr)
    Python: comm.Set_errhandler(MPI.ERRORS_RETURN)

    Newly added in MPI-4.0
  - directly after MPI_INIT with both comm = MPI_COMM_WORLD and MPI_COMM_SELF, then
    - **ierror returned by each MPI routine (except MPI window and MPI file routines)**
    - **undefined state after an erroneous MPI call has occurred**
      (only MPI_Abort(…) should be still callable)
  - Exception: MPI-I/O has default MPI_ERRORS_RETURN
    - Default can be changed through MPI_FILE_NULL:
    - MPI_File_set_errhandler (MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL)
      Python: MPI.FILE_NULL.Set_errhandler(MPI.ERRORS_ARE_FATAL)
    - See MPI-3.1 Sect. 13.7, page 555 / MPI-4.0 Sect. 14.7, page 719, and course Chapter 7
  - MPI_ERRORS_ARE_FATAL aborts the process and all connected processes
  - MPI_ERRORS_ABORT aborts only all processes of the related communicator

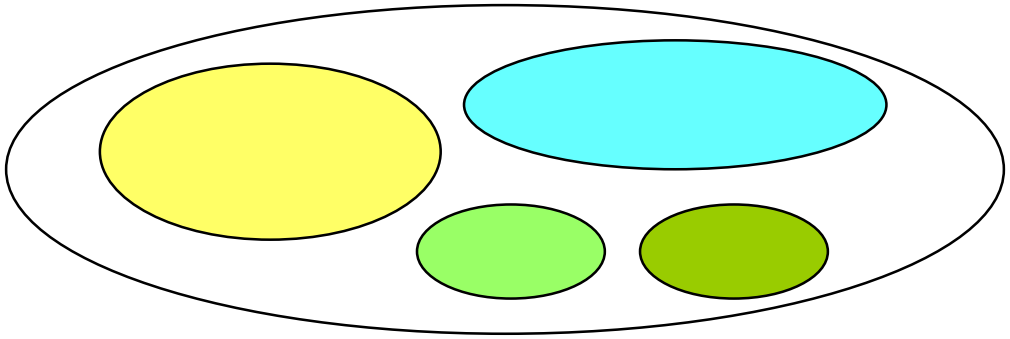
    New in MPI-4.0

Goals

Support for libraries or application sub-spaces

- Safe communication context spaces
  - **e.g., for subsets of processes,**
  - **or duplicated communicators for independent software layers (middle-ware)**
- Collective operations (→course Chapter 6) on a subset of processes

A library should always use a duplicate of MPI_COMM_WORLD, and never MPI_COMM_WORLD itself.

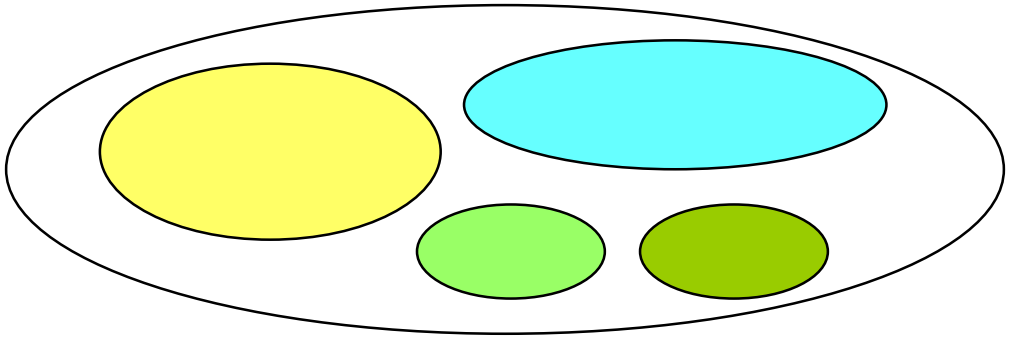


Goals

Support for libraries or application sub-spaces

- Safe communication context spaces
  - **e.g., for subsets of processes,**
  - **or duplicated communicators for independent software layers (middle-ware)**
- Collective operations (→course Chapter 6) on a subset of processes
- Re-numbering of the ranks of communicators

A library should always use a duplicate of MPI_COMM_WORLD, and never MPI_COMM_WORLD itself.

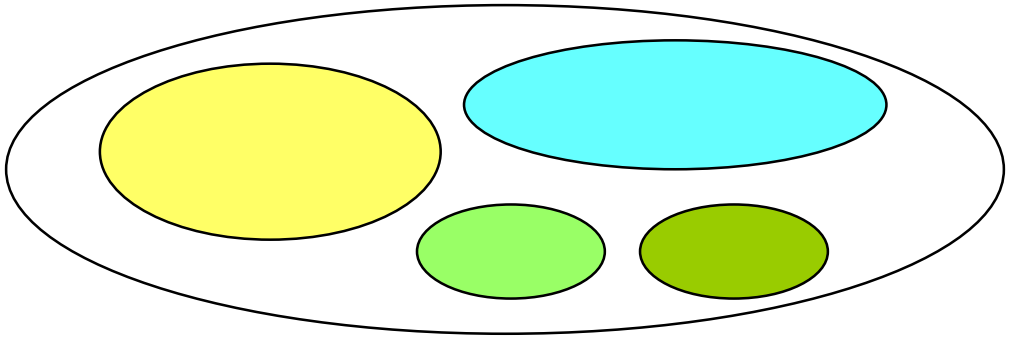


Goals

Support for libraries or application sub-spaces

- Safe communication context spaces
  - **e.g., for subsets of processes,**
  - **or duplicated communicators for independent software layers (middle-ware)**
- Collective operations (→course Chapter 6) on a subset of processes
- Re-numbering of the ranks of communicators
- Inter-communicators

A library should always use a duplicate of MPI_COMM_WORLD, and never MPI_COMM_WORLD itself.

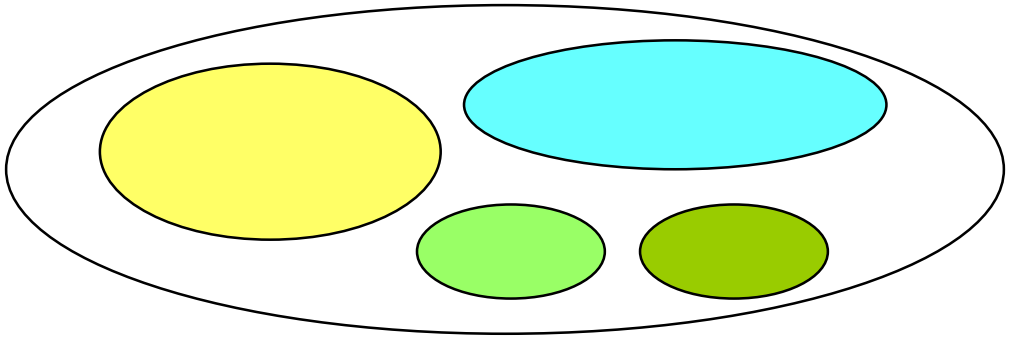


Goals

Support for libraries or application sub-spaces

- Safe communication context spaces
  - **e.g., for subsets of processes,**
  - **or duplicated communicators for independent software layers (middle-ware)**
- Collective operations (→course Chapter 6) on a subset of processes
- Re-numbering of the ranks of communicators
- Inter-communicators
- Info handles

A library should always use a duplicate of MPI_COMM_WORLD, and never MPI_COMM_WORLD itself.



Goals

Support for libraries or application sub-spaces

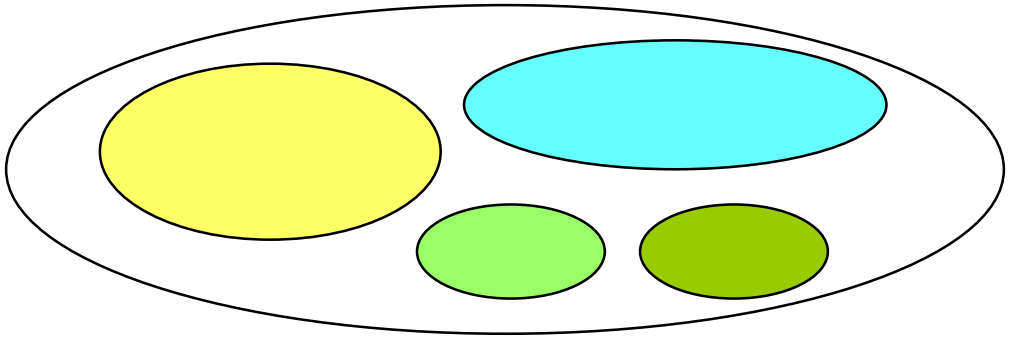
- Safe communication context spaces
  - **e.g., for subsets of processes,**
  - **or duplicated communicators for independent software layers (middle-ware)**
- Collective operations (→course Chapter 6) on a subset of processes

A library should always use a duplicate of MPI_COMM_WORLD, and never MPI_COMM_WORLD itself.

- Re-numbering of the ranks of communicators
- Inter-communicators
- Info handles
- Naming of context spaces
- Add additional user-defined attributes to a communication context
- Inquiry methods
- *World Model* and *Sessions Model*

New in MPI-4.0

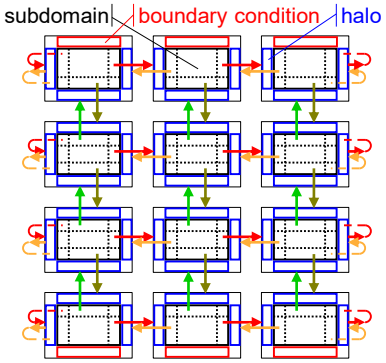
→ Section (2) of this course chapter



Why do we need additional (sub-) communicators?

Coupled applications, e.g.

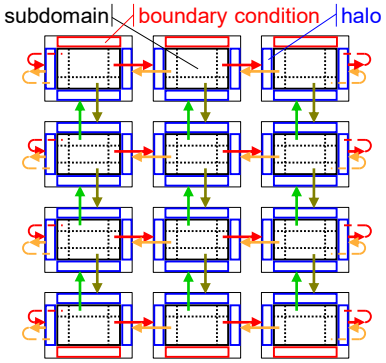
- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)



Why do we need additional (sub-) communicators?

Coupled applications, e.g.

- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)

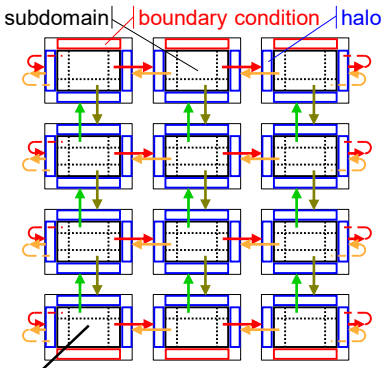


Major design decision

Why do we need additional (sub-) communicators?

Coupled applications, e.g.

- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)



Major design decision

- Each MPI process runs all codes (e.g. m-AIA)
  - Within each simulation step (e.g. time-step):
    - **code A, code B, …**
    - **Same / different data grid for all / each code**
    - **data-grids distributed over all MPI processes**
    - **Data exchange of the coupling may be within each process, e.g., code A accesses data of B**

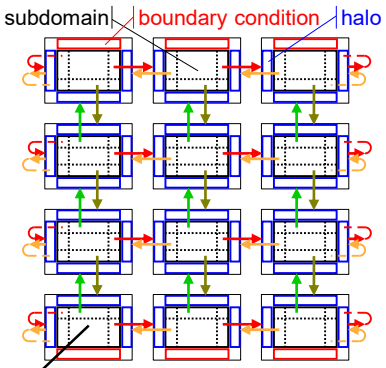
```
do it=1, itmax
  call A
  call B
  call AB_exchange
end do
```

```
subroutine A
  halo-exchange for A
  one time-step for A
end
```

Why do we need additional (sub-) communicators?

Coupled applications, e.g.

- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)



Major design decision

- Each MPI process runs all codes (e.g. m-AIA)
  - Within each simulation step (e.g. time-step):
    - **code A, code B, …**
    - **Same / different data grid for all / each code**
    - **data-grids distributed over all MPI processes**
    - **Data exchange of the coupling may be within each process, e.g., code A accesses data of B**

```
do it=1, itmax
  call A
  call B
  call AB_exchange
end do
```

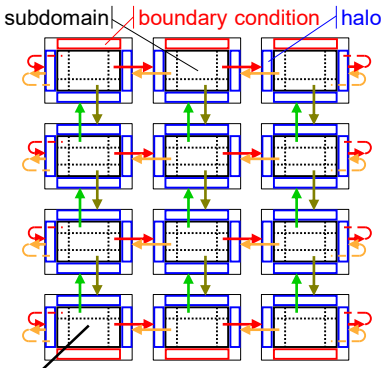
```
subroutine A
  halo-exchange for A
  one time-step for A
end
```

e.g. using a duplicate of MPI_COMM_WORLD

Why do we need additional (sub-) communicators?

Coupled applications, e.g.

- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)



Major design decision

- Each MPI process runs all codes (e.g. m-AIA)
  - Within each simulation step (e.g. time-step):
    - **code A, code B, …**
    - **Same / different data grid for all / each code**
    - **data-grids distributed over all MPI processes**
    - **Data exchange of the coupling may be within each process, e.g., code A accesses data of B**

```
do it=1, itmax
  call A
  call B
  call AB_exchange
end do
```

```
subroutine A
  halo-exchange for A
  one time-step for A
end
```

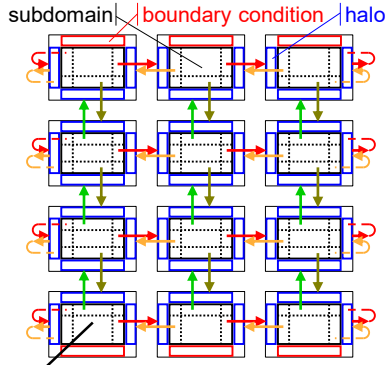
e.g. using a duplicate of MPI_COMM_WORLD

- Each MPI process is dedicated to a specific code (e.g. ICON, APES, FLOWer)

Why do we need additional (sub-) communicators?

Coupled applications, e.g.

- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)



Major design decision

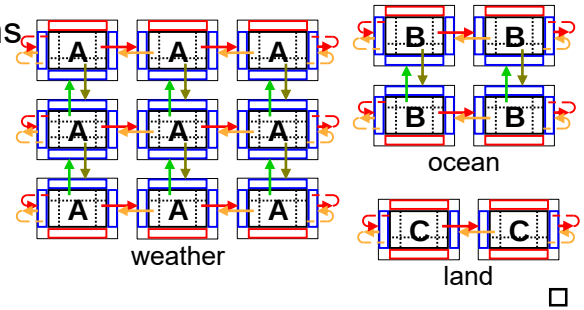
- Each MPI process runs all codes (e.g. m-AIA)
  - Within each simulation step (e.g. time-step):
    - **code A, code B, …**
    - **Same / different data grid for all / each code**
    - **data-grids distributed over all MPI processes**
    - **Data exchange of the coupling may be within each process, e.g., code A accesses data of B**

```
do it=1, itmax
  call A
  call B
  call AB_exchange
end do
```

```
subroutine A
  halo-exchange for A
  one time-step for A
end
```

e.g. using a duplicate of MPI_COMM_WORLD

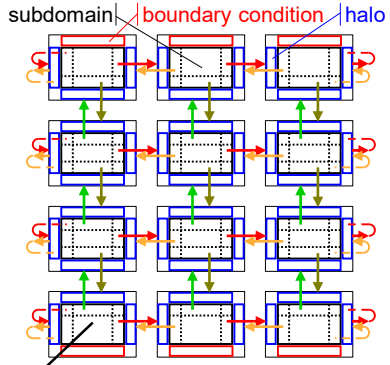
- Each MPI process is dedicated to a specific code (e.g. ICON, APES, FLOWer)
  - *a* % of all processes **(of each node)** simulate code A on subdomains of the simulation domain A (e.g. ocean flow on a ocean data grid)
  - *b* % for code B, …



Why do we need additional (sub-) communicators?

Coupled applications, e.g.

- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)



Major design decision

- Each MPI process runs all codes (e.g. m-AIA)
  - Within each simulation step (e.g. time-step):
    - **code A, code B, …**
    - **Same / different data grid for all / each code**
    - **data-grids distributed over all MPI processes**
    - **Data exchange of the coupling may be within each process, e.g., code A accesses data of B**

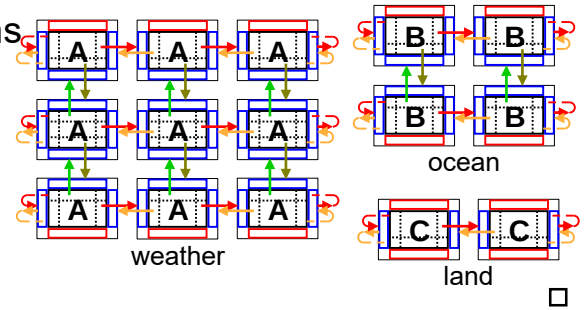
```
do it=1, itmax
  call A
  call B
  call AB_exchange
end do
```

```
subroutine A
  halo-exchange for A
  one time-step for A
end
```

e.g. using a duplicate of MPI_COMM_WORLD

- Each MPI process is dedicated to a specific code (e.g. ICON, APES, FLOWer)
  - *a* % of all processes **(of each node)** simulate code A on subdomains of the simulation domain A (e.g. ocean flow on a ocean data grid)
  - *b* % for code B, …

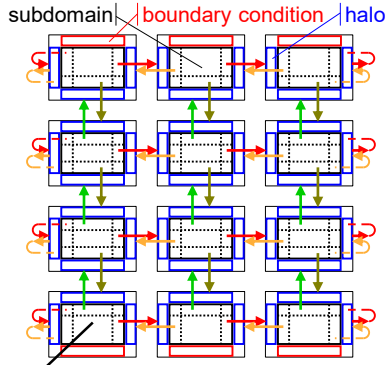
using a sub-communicator



Why do we need additional (sub-) communicators?

Coupled applications, e.g.

- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)



Major design decision

- Each MPI process runs all codes (e.g. m-AIA)
  - Within each simulation step (e.g. time-step):
    - **code A, code B, …**
    - **Same / different data grid for all / each code**
    - **data-grids distributed over all MPI processes**
    - **Data exchange of the coupling may be within each process, e.g., code A accesses data of B**

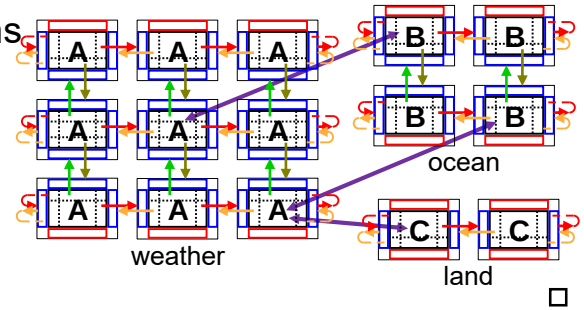
```
do it=1, itmax
  call A
  call B
  call AB_exchange
end do
```

```
subroutine A
  halo-exchange for A
  one time-step for A
end
```

e.g. using a duplicate of MPI_COMM_WORLD

- Each MPI process is dedicated to a specific code (e.g. ICON, APES, FLOWer)
  - *a* % of all processes **(of each node)** simulate code A on subdomains of the simulation domain A (e.g. ocean flow on a ocean data grid)
  - *b* % for code B, …
  - Additional messages for the coupling

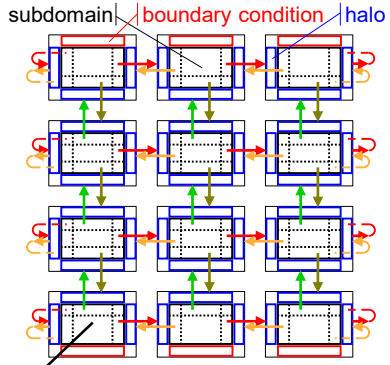
using a sub-communicator



Why do we need additional (sub-) communicators?

Coupled applications, e.g.

- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)



Major design decision

- Each MPI process runs all codes (e.g. m-AIA)
  - Within each simulation step (e.g. time-step):
    - **code A, code B, …**
    - **Same / different data grid for all / each code**
    - **data-grids distributed over all MPI processes**
    - **Data exchange of the coupling may be within each process, e.g., code A accesses data of B**

```
do it=1, itmax
  call A
  call B
  call AB_exchange
end do
```

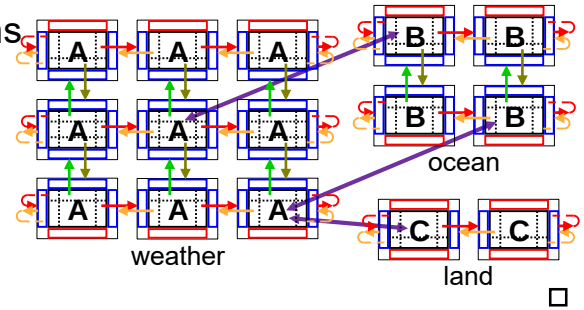
```
subroutine A
  halo-exchange for A
  one time-step for A
end
```

e.g. using a duplicate of MPI_COMM_WORLD

- Each MPI process is dedicated to a specific code (e.g. ICON, APES, FLOWer)
  - *a* % of all processes **(of each node)** simulate code A on subdomains of the simulation domain A (e.g. ocean flow on a ocean data grid)
  - *b* % for code B, …
  - Additional messages for the coupling

using a sub-communicator

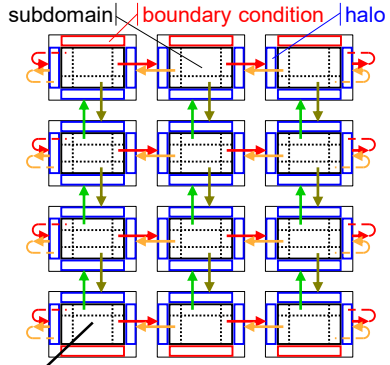
using inter-communi-cators (in ICON)



Why do we need additional (sub-) communicators?

Coupled applications, e.g.

- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)



Major design decision

- Each MPI process runs all codes (e.g. m-AIA)
  - Within each simulation step (e.g. time-step):
    - **code A, code B, …**
    - **Same / different data grid for all / each code**
    - **data-grids distributed over all MPI processes**
    - **Data exchange of the coupling may be within each process, e.g., code A accesses data of B**

```
do it=1, itmax
  call A
  call B
  call AB_exchange
end do
```

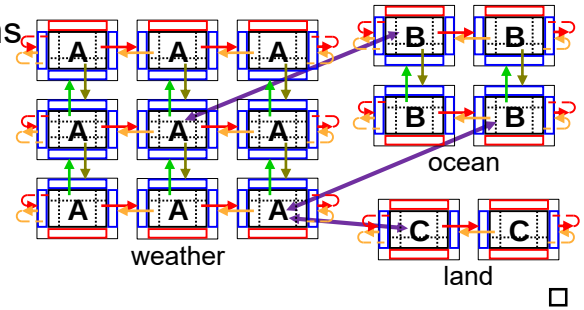
```
subroutine A
  halo-exchange for A
  one time-step for A
end
```

e.g. using a duplicate of MPI_COMM_WORLD

- Each MPI process is dedicated to a specific code (e.g. ICON, APES, FLOWer)
  - *a* % of all processes **(of each node)** simulate code A on subdomains of the simulation domain A (e.g. ocean flow on a ocean data grid)
  - *b* % for code B, …
  - Additional messages for the coupling
  - Additional service processes, e.g., for asynchronous parallel I/O

using a sub-communicator

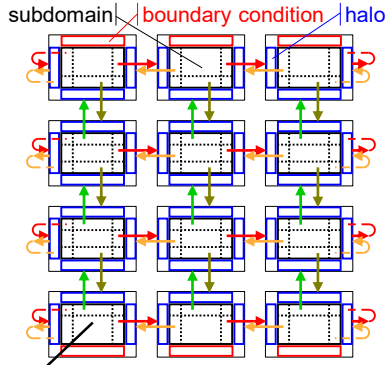
using inter-communi-cators (in ICON)



Why do we need additional (sub-) communicators?

Coupled applications, e.g.

- Computational fluid dynamics (CFD) & structural mechanics
  - e.g., simulating rotators with FLOWer (DLR)
- Weather / climate: ocean & atmosphere & land surface
  - e.g., ICON (Blue Marble, DWD, Max Planck Institute for Meteorolgy MPI-MET, DKRZ)
- Multi-physics code m-AIA (Institute of Aerodynamics (AIA), RWTH Aachen)
- Adaptable Poly-Engineering Simulator (www.APES-suite.org, DLR)



Major design decision

- Each MPI process runs all codes (e.g. m-AIA)
  - Within each simulation step (e.g. time-step):
    - **code A, code B, …**
    - **Same / different data grid for all / each code**
    - **data-grids distributed over all MPI processes**
    - **Data exchange of the coupling may be within each process, e.g., code A accesses data of B**

```
do it=1, itmax
  call A
  call B
  call AB_exchange
end do
```

```
subroutine A
  halo-exchange for A
  one time-step for A
end
```

e.g. using a duplicate of MPI_COMM_WORLD

- Each MPI process is dedicated to a specific code (e.g. ICON, APES, FLOWer)

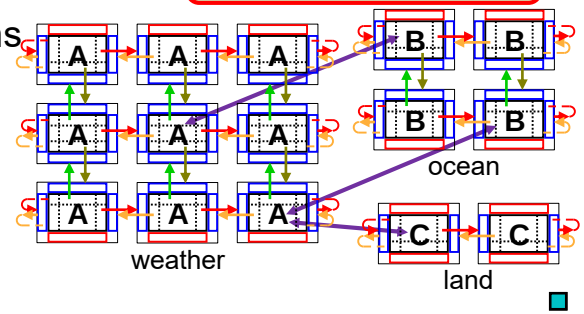
Enables a larger number of processes

using a sub-communicator

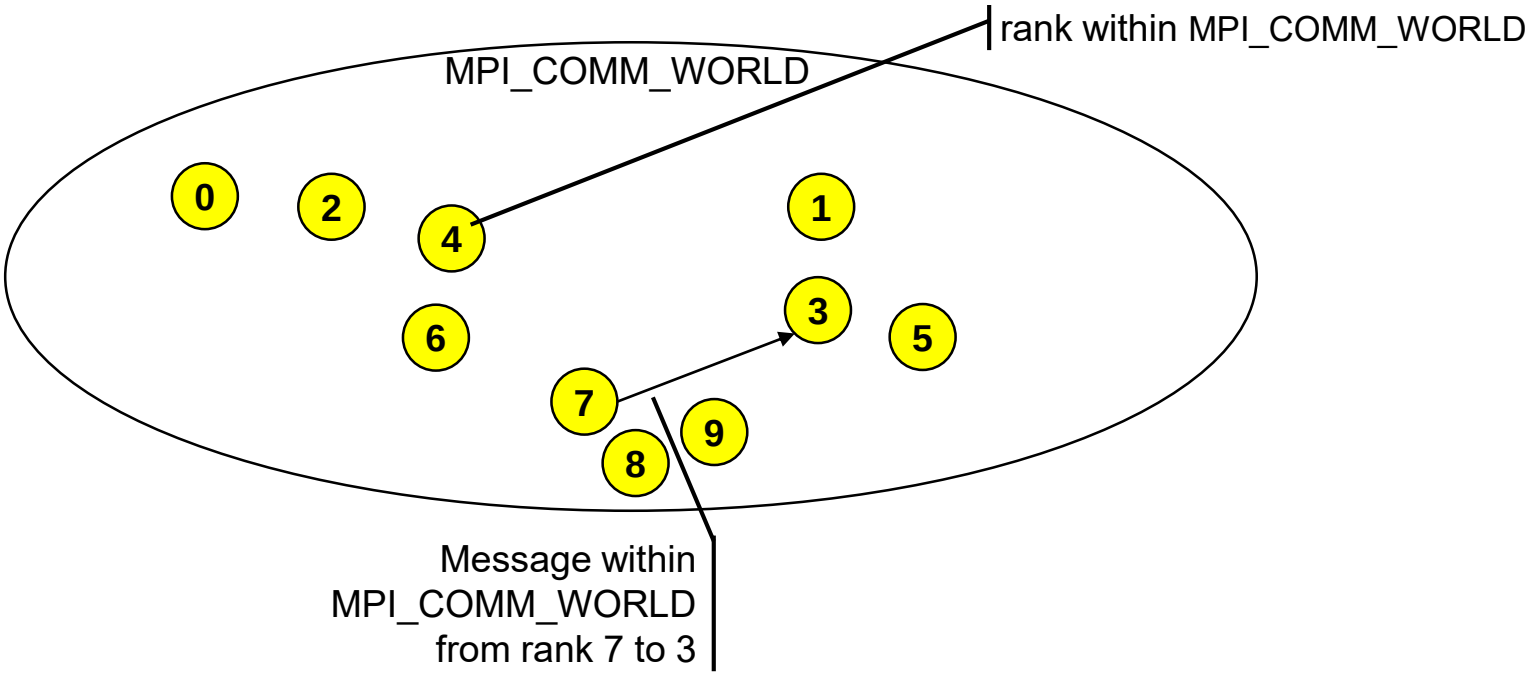
- *a* % of all processes **(of each node)** simulate code A on subdomains of the simulation domain A (e.g. ocean flow on a ocean data grid)
- *b* % for code B, …
- Additional messages for the coupling

using inter-communi-cators (in ICON)

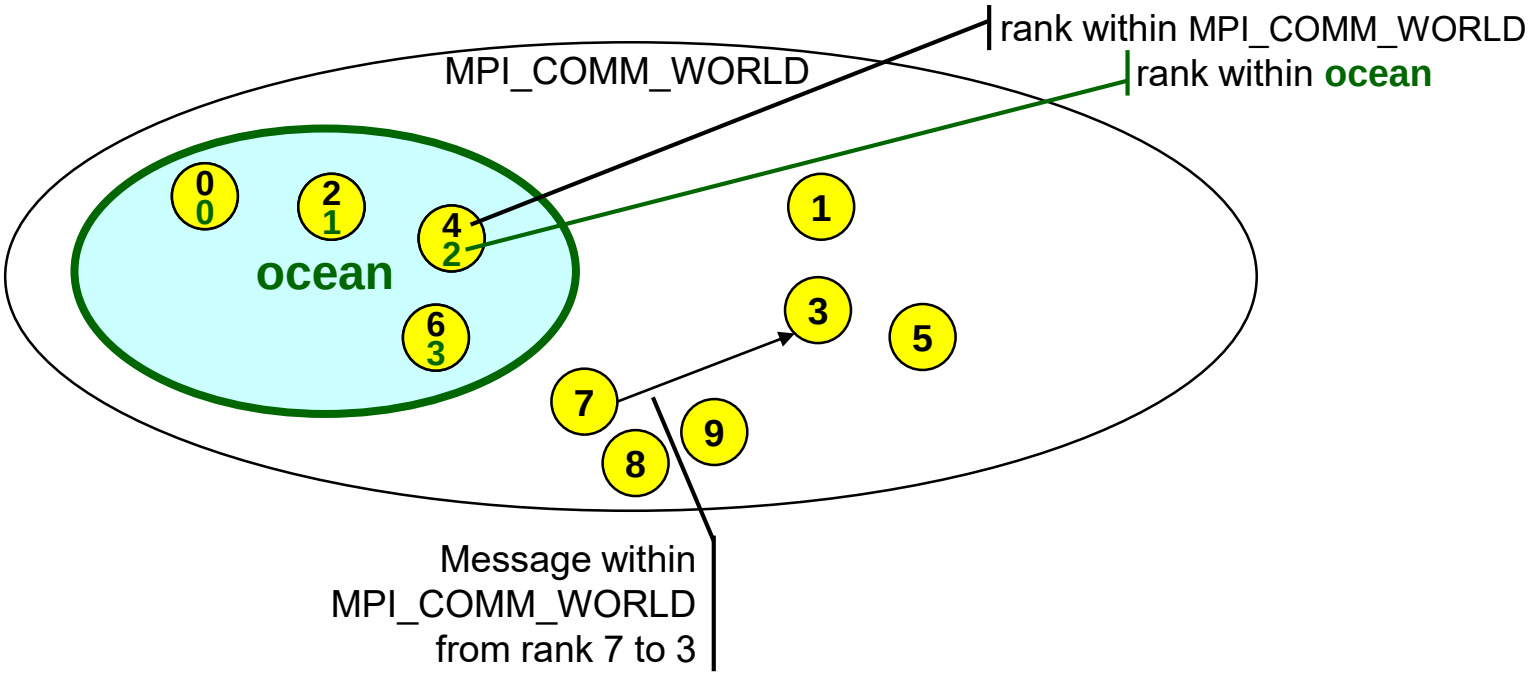
- Additional service processes, e.g., for asynchronous parallel I/O



Methods – e.g., for coupled applications

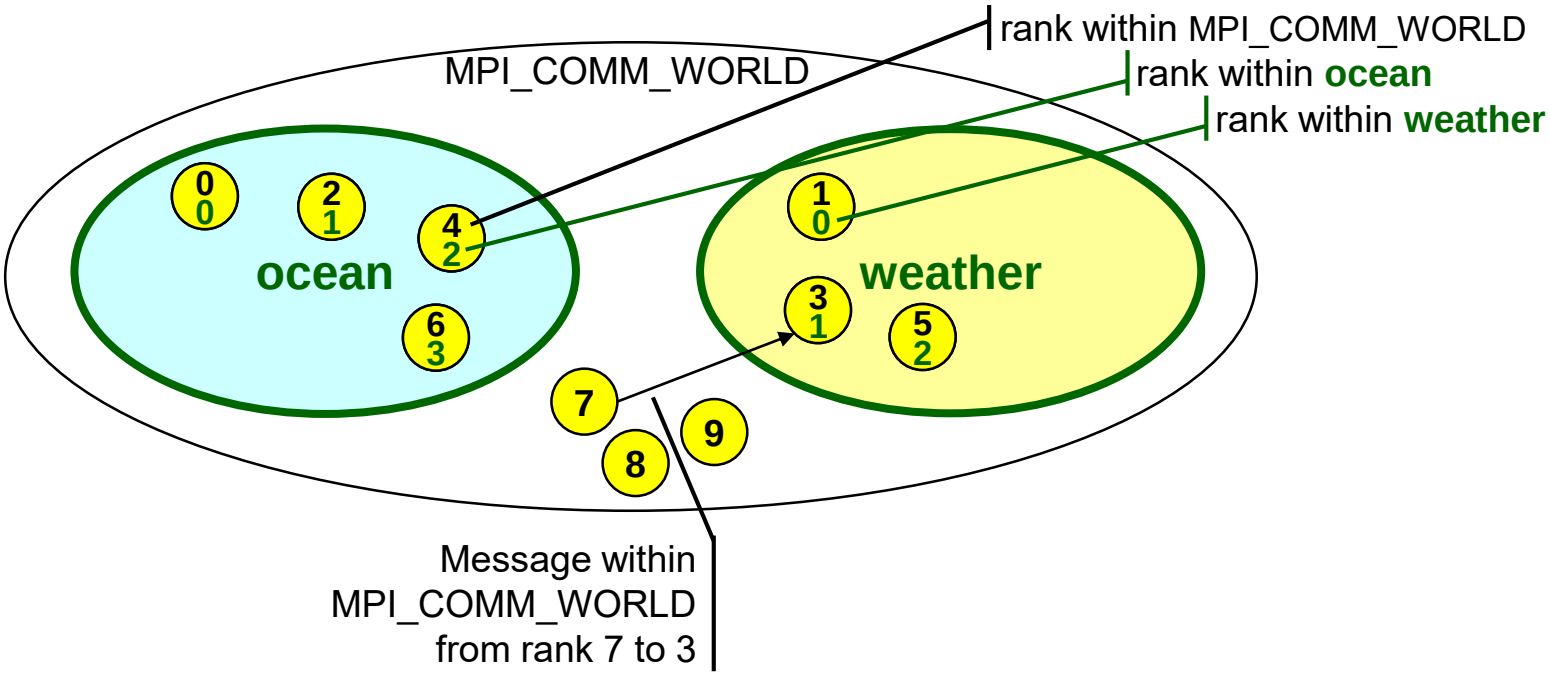


Methods – e.g., for coupled applications



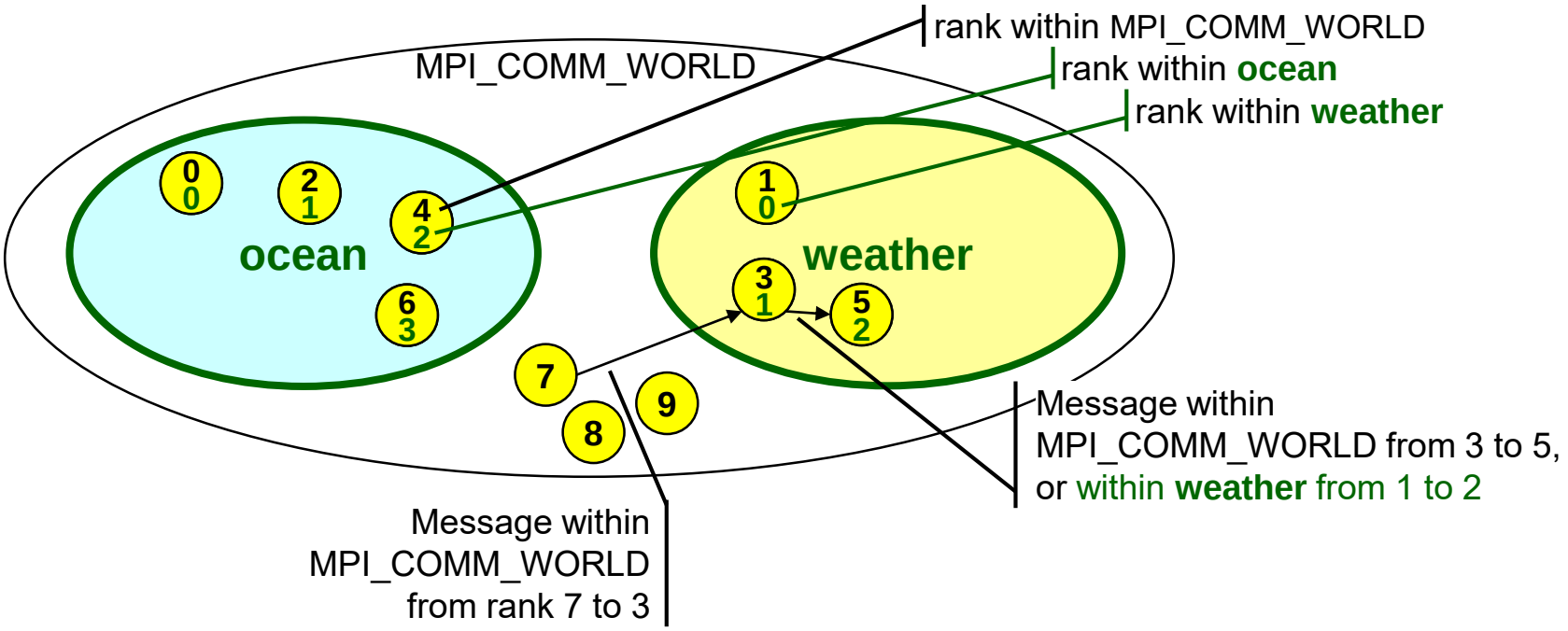
- **Sub-communicators:** Collectively defined communication sub-spaces

Methods – e.g., for coupled applications



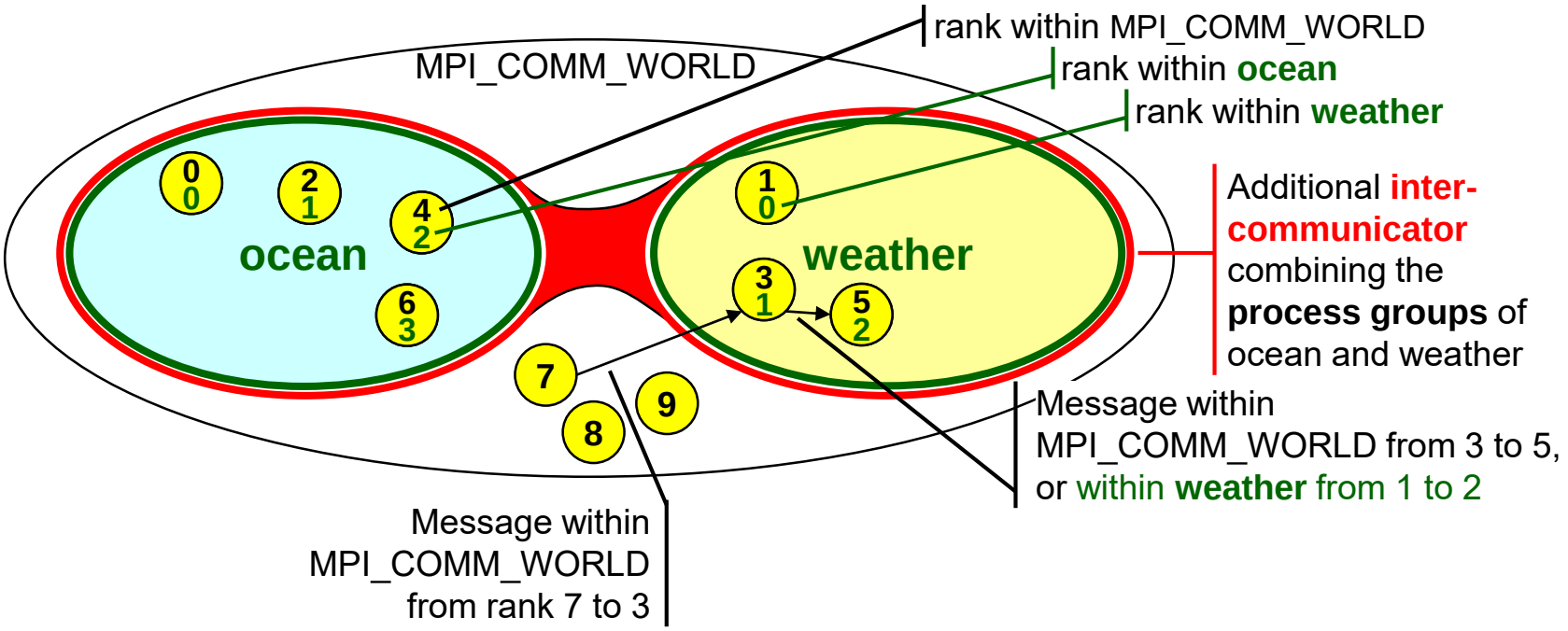
- **Sub-communicators:** Collectively defined communication sub-spaces

Methods – e.g., for coupled applications



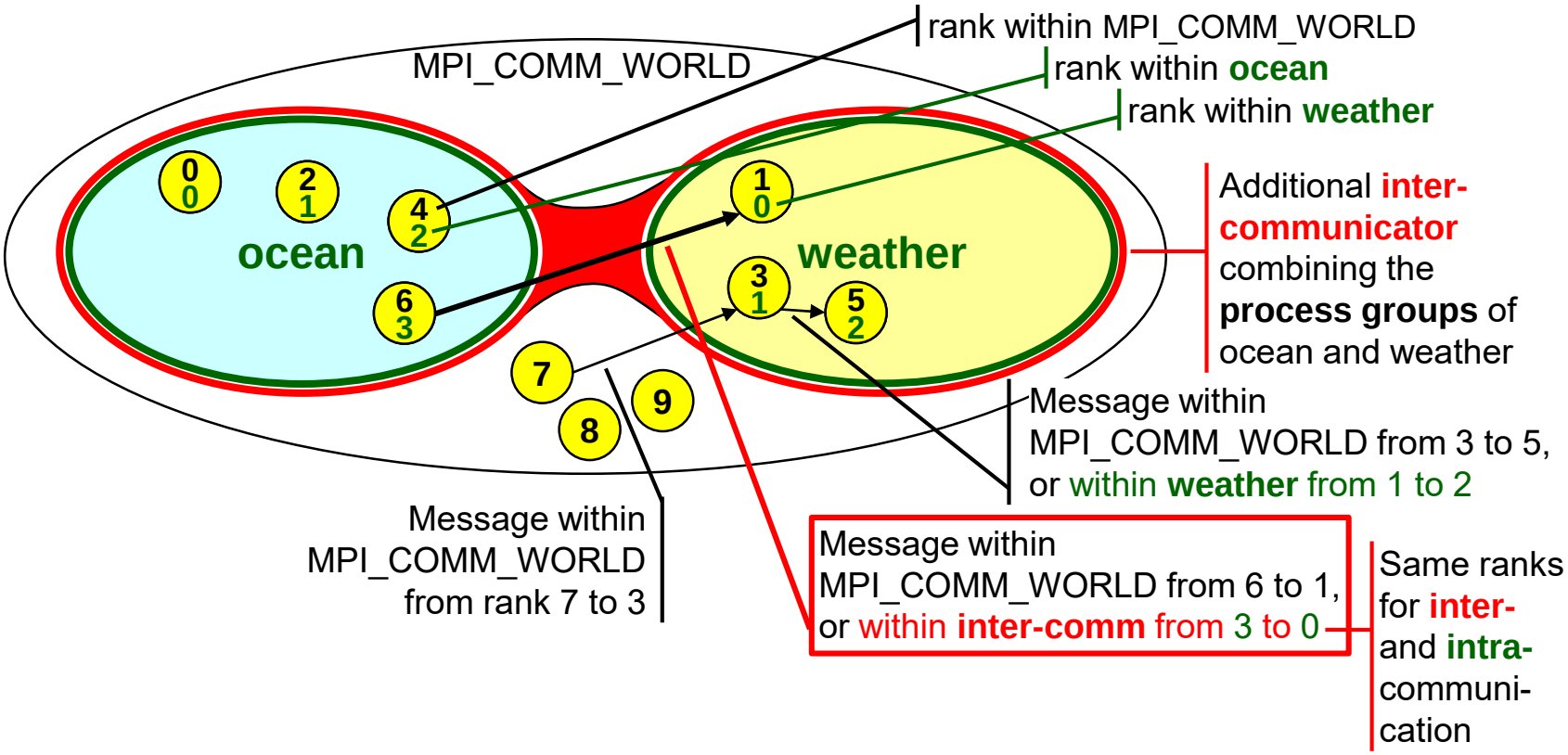
- **Sub-communicators:** Collectively defined communication sub-spaces

Methods – e.g., for coupled applications



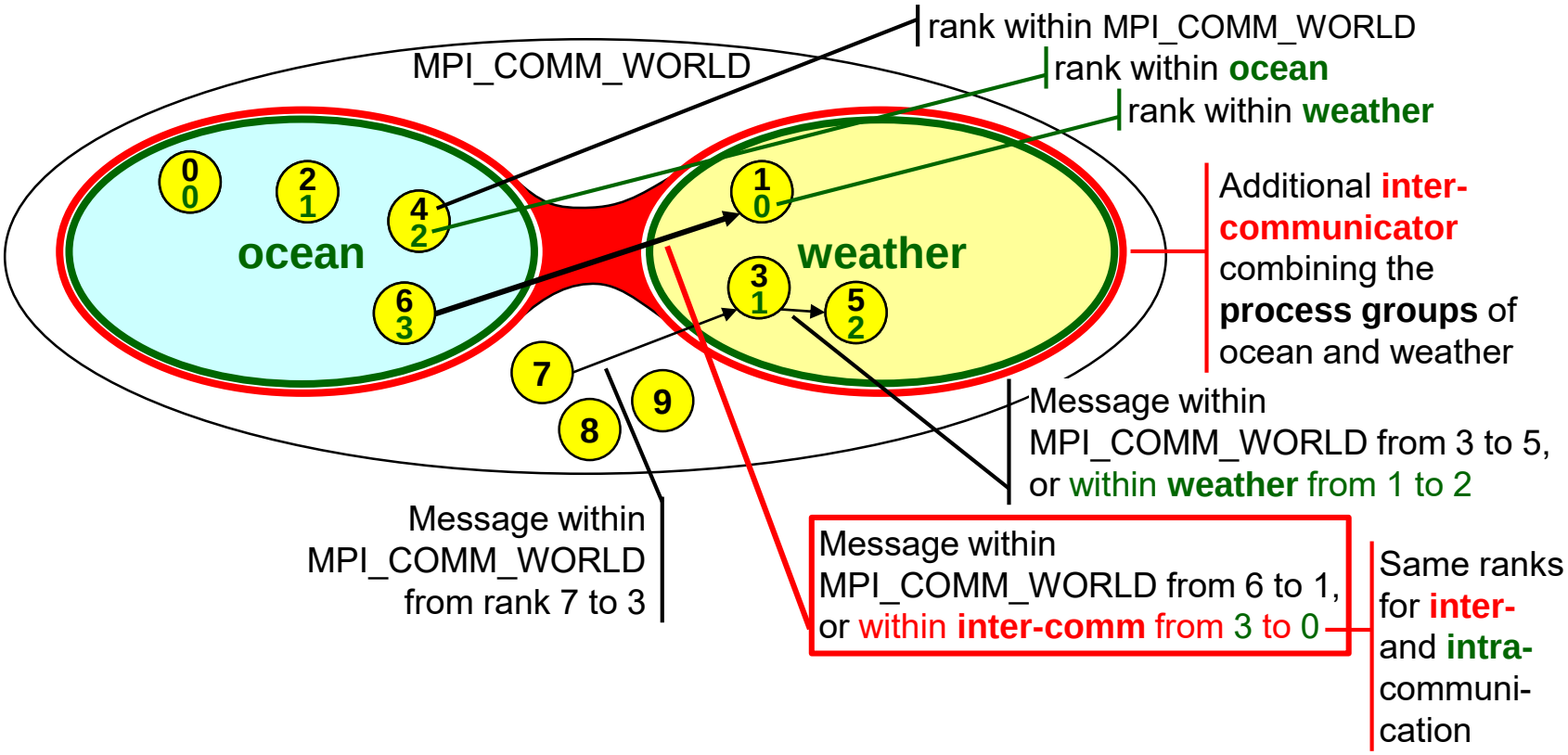
- **Sub-communicators:** Collectively defined communication sub-spaces
- **Intra-** and **inter-communicators**

Methods – e.g., for coupled applications



- **Sub-communicators:** Collectively defined communication sub-spaces
- **Intra-** and **inter-communicators**

Methods – e.g., for coupled applications



- **Sub-communicators:** Collectively defined communication sub-spaces
- **Intra-** and **inter-communicators**

Perfect for any communication between processes of the two groups (ocean and weather)

Sub-groups and sub-communicators (1)

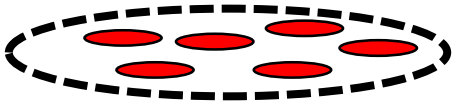
Several ways to establish sub-communicators

Two levels:

- Group  of processes 

Sub-groups and sub-communicators (1)

Several ways to establish sub-communicators

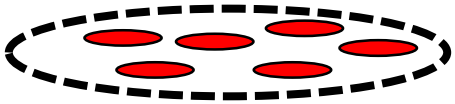


Two levels:

- Group  of processes 
  - **Without the ability to communicate**
  - **Local routines to build group & sub-sets**
  - **Same ranks as in related communicator**

Sub-groups and sub-communicators (1)

Several ways to establish sub-communicators

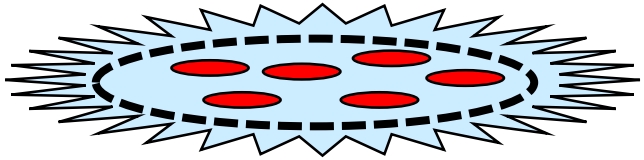


Two levels:

- Group  of processes 
  - **Without the ability to communicate**
  - **Local routines to build group & sub-sets**
  - **Same ranks as in related communicator**
- Communicators
  - **Group of processes with additional ability to communicate**

Sub-groups and sub-communicators (1)

Several ways to establish sub-communicators

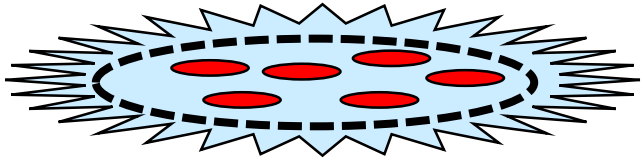


Two levels:

- Group  of processes 
  - **Without the ability to communicate**
  - **Local routines to build group & sub-sets**
  - **Same ranks as in related communicator**
- Communicators 
  - **Group of processes with additional ability to communicate**

Sub-groups and sub-communicators (1)

Several ways to establish sub-communicators



Two levels:

- Group  of processes 
  - **Without the ability to communicate**
  - **Local routines to build group & sub-sets**
  - **Same ranks as in related communicator**
- Communicators 
  - **Group of processes with additional ability to communicate**

Scalability problems when handling many processes in each process

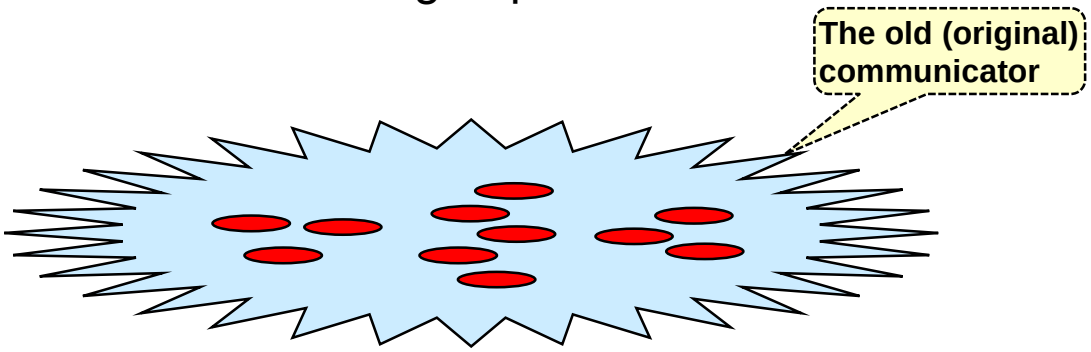
Sub-groups and sub-communicators (2)

Sub-groups and sub-communicators (2)

- New sub-communicators via sub-groups

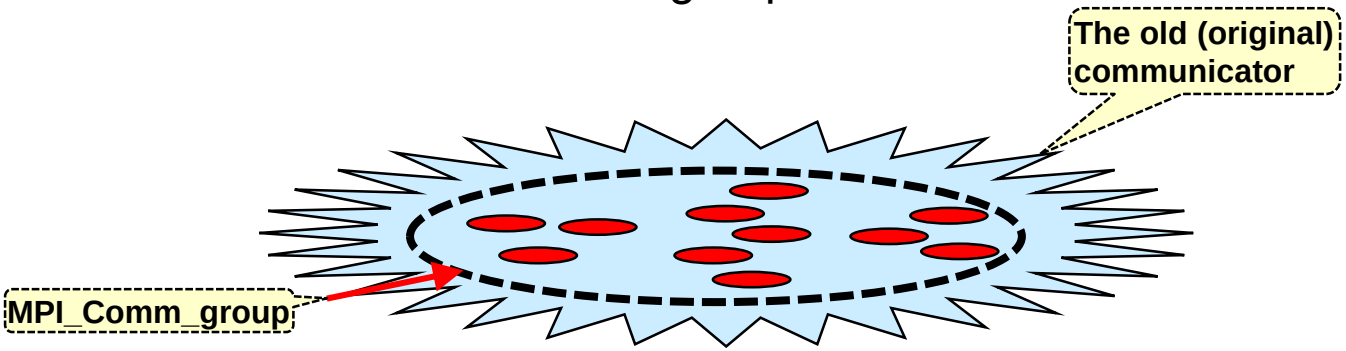
Sub-groups and sub-communicators (2)

- New sub-communicators via sub-groups



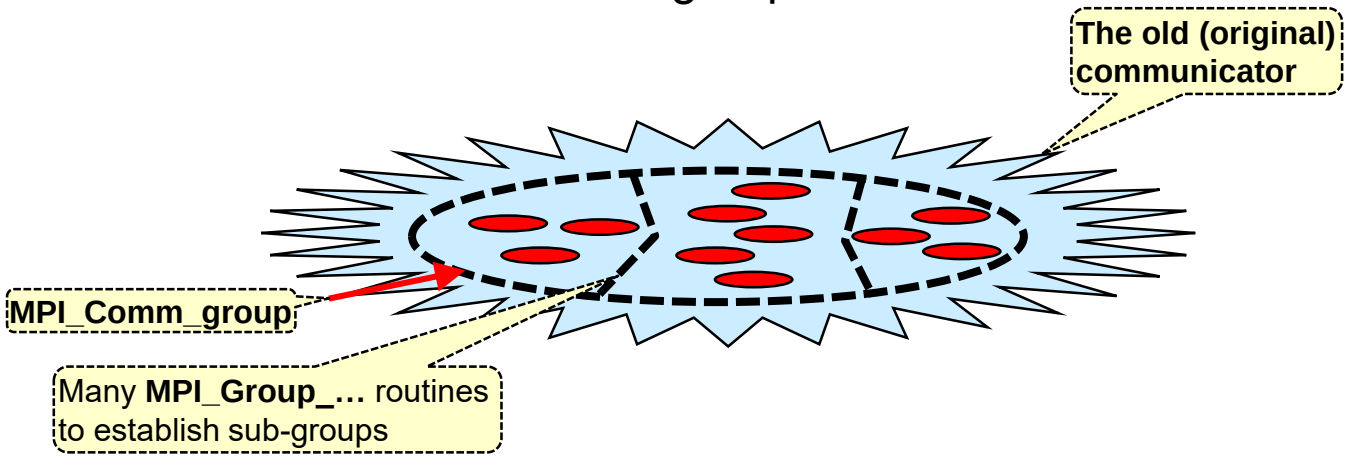
Sub-groups and sub-communicators (2)

- New sub-communicators via sub-groups



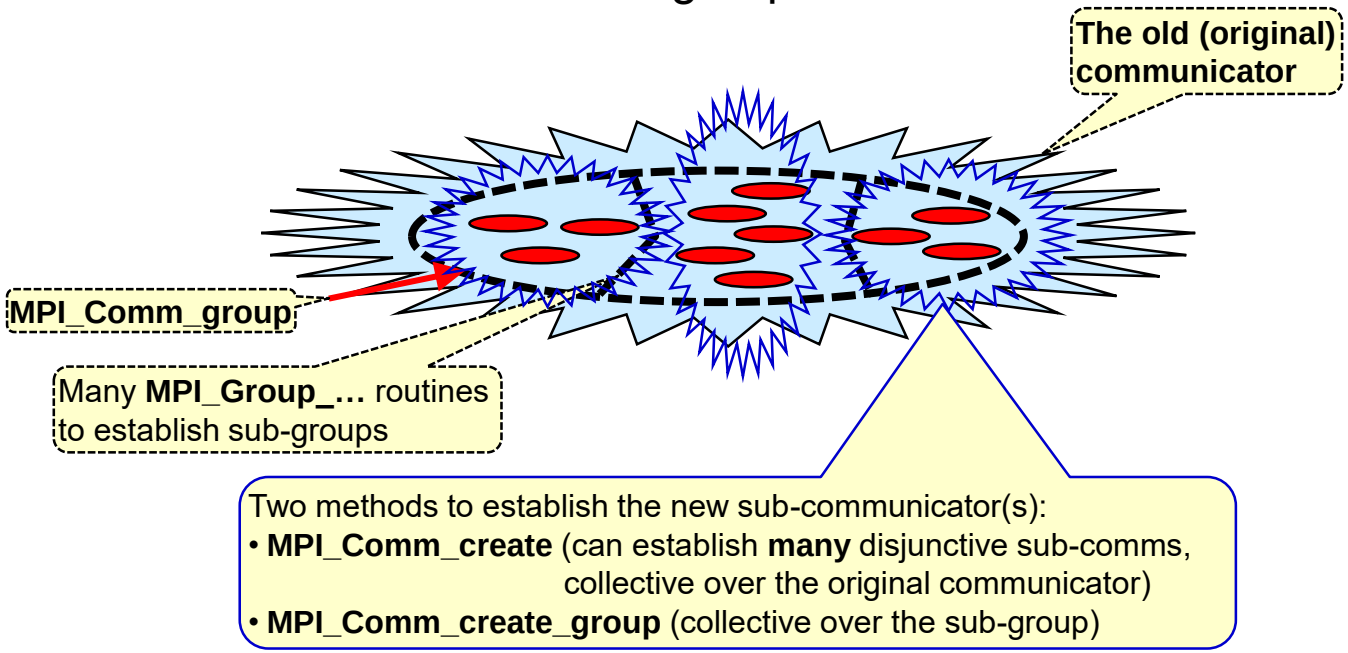
Sub-groups and sub-communicators (2)

- New sub-communicators via sub-groups



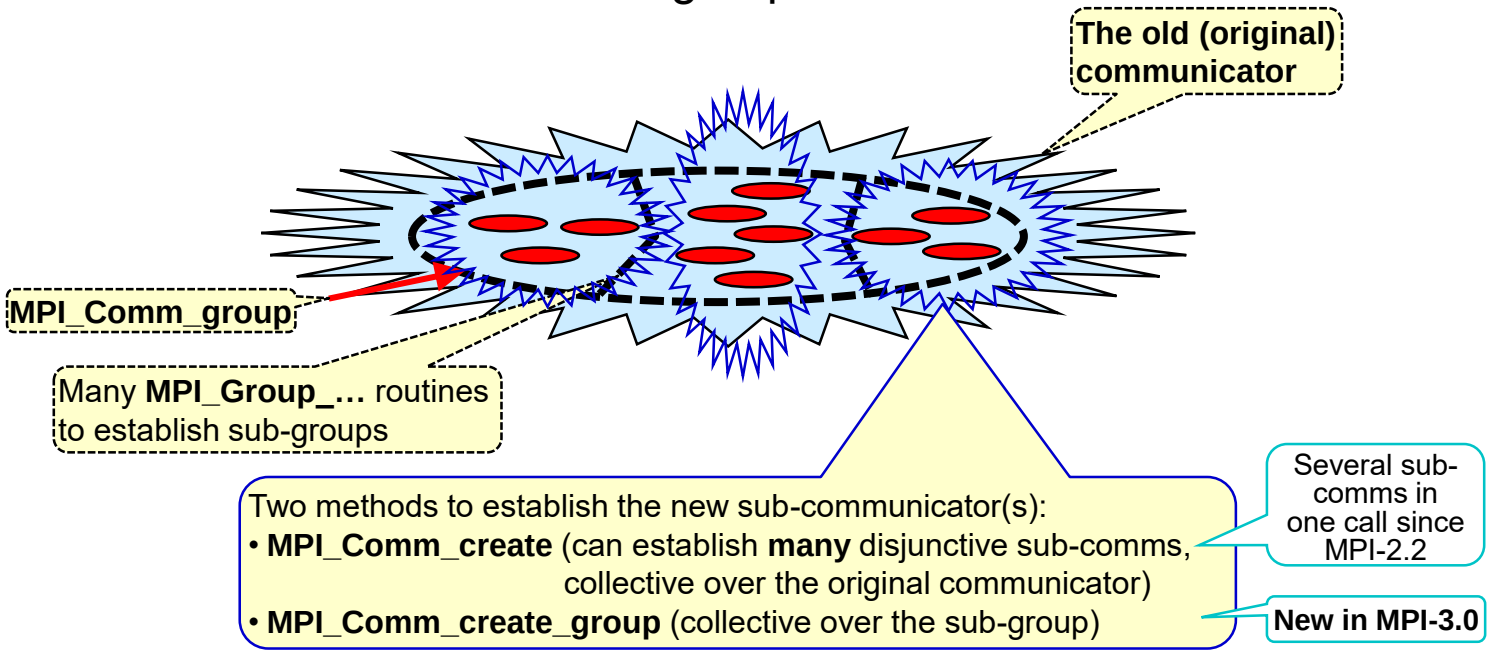
Sub-groups and sub-communicators (2)

- New sub-communicators via sub-groups



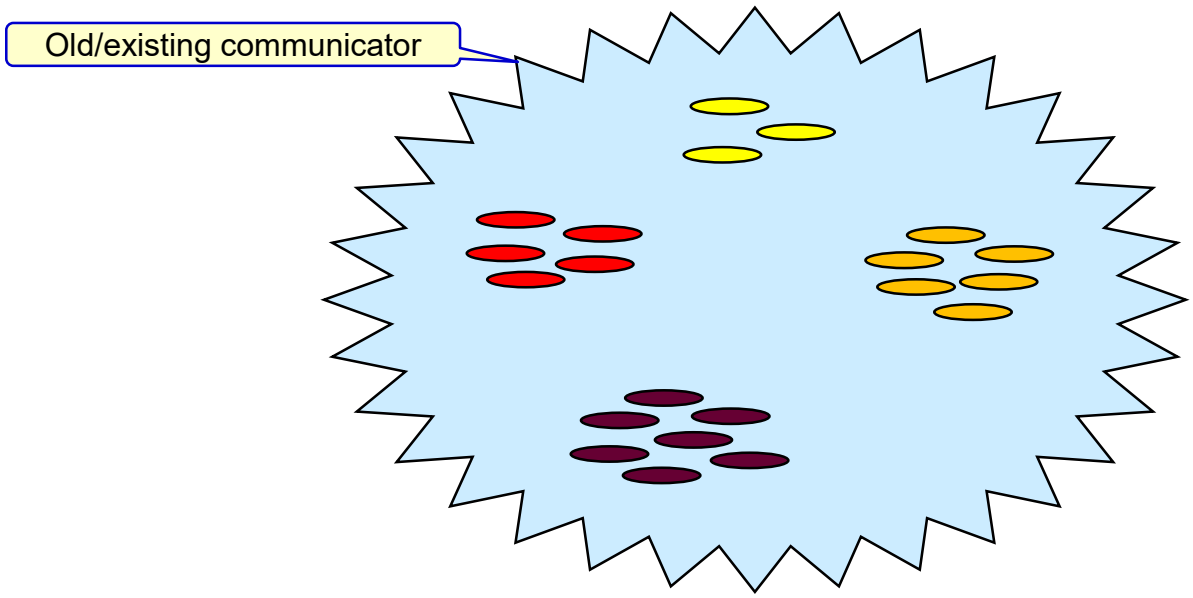
Sub-groups and sub-communicators (2)

- New sub-communicators via sub-groups



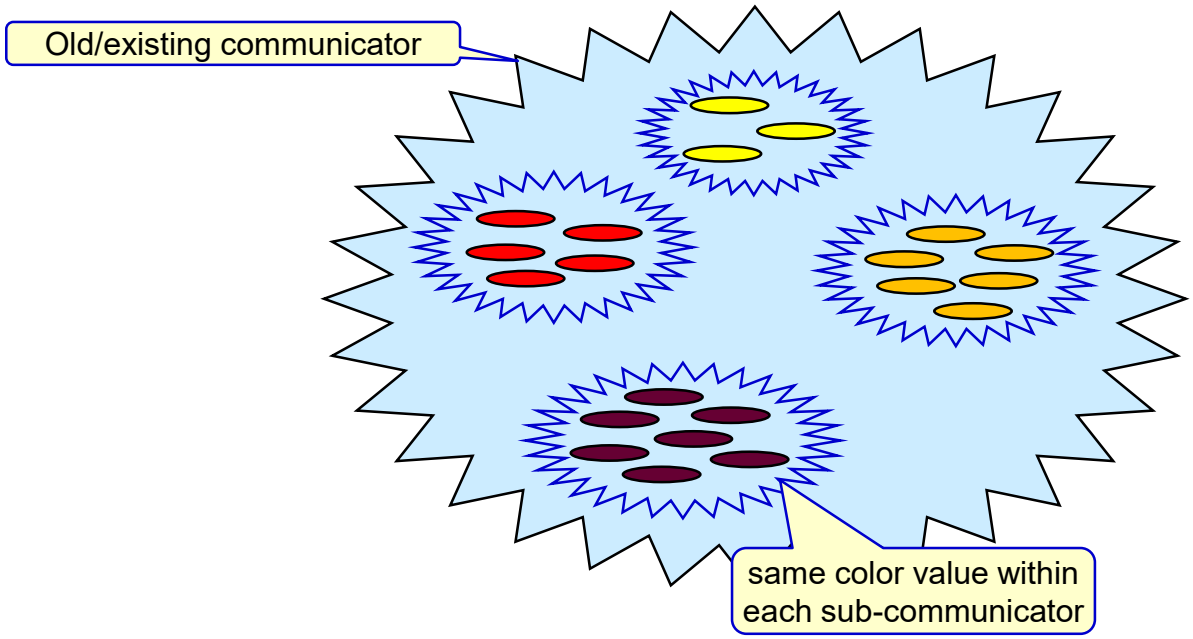
Sub-groups and sub-communicators (3)

- New sub-communicators via MPI_Comm_split
  - Each process must specify a color



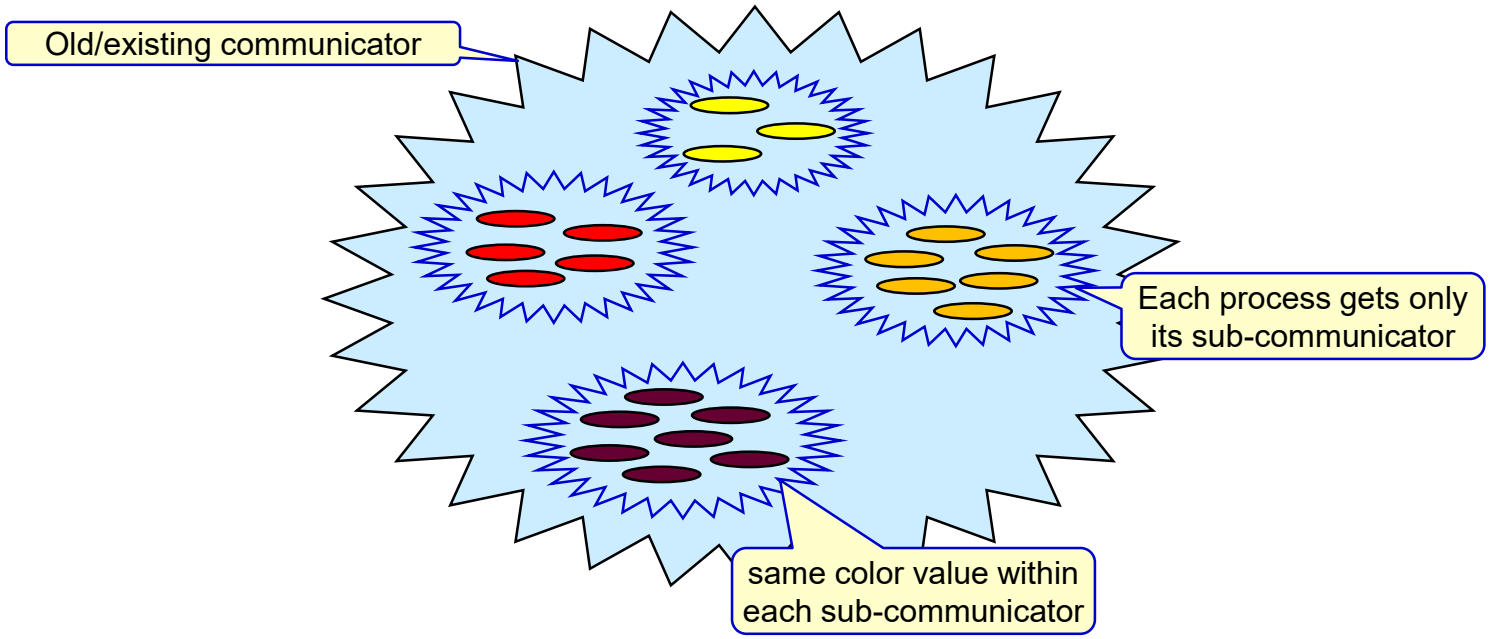
Sub-groups and sub-communicators (3)

- New sub-communicators via MPI_Comm_split
  - Each process must specify a color
  - Processes with same color are put together in new sub-communicators



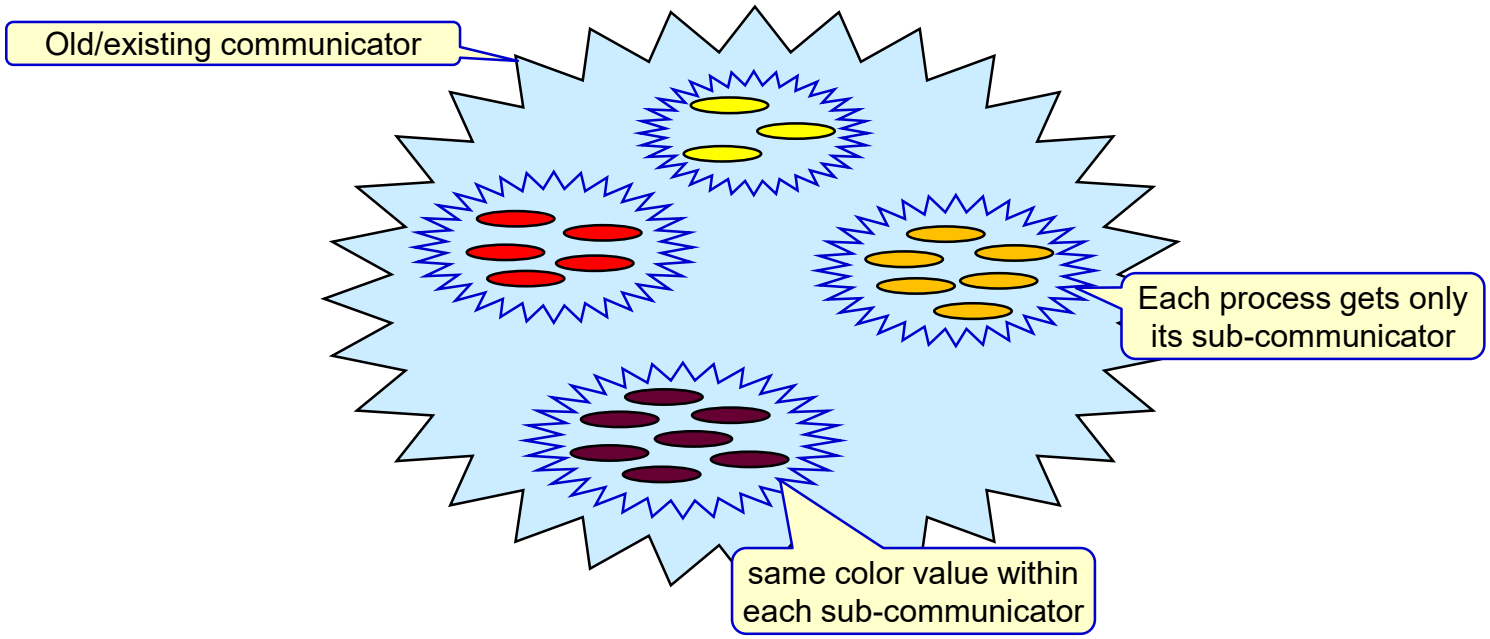
Sub-groups and sub-communicators (3)

- New sub-communicators via MPI_Comm_split
  - Each process must specify a color
  - Processes with same color are put together in new sub-communicators



Sub-groups and sub-communicators (3)

- New sub-communicators via MPI_Comm_split & MPI_Comm_split_type (New in MPI-3.0)
  - Each process must specify a color
  - Processes with same color are put together in new sub-communicators



Example: MPI_Comm_split()

Creation is **collective** in the **old** communicator.

All processes with same color are grouped into separate sub-communicators

C

- C/C++: int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm **newcomm*)

Fortran

- Fortran: MPI_COMM_SPLIT (comm, color, key, *newcomm*, *ierror*)

  mpi_f08:
  ```
  TYPE(MPI_Comm) :: comm, newcomm
  INTEGER :: color, key;
  INTEGER, OPTIONAL :: ierror
  ```

  mpi & mpif.h:
  ```
  INTEGER comm, color, key, newcomm, ierror
  ```

Python

- Python: *newcomm* = comm.Split(color=0, key=0)

Each process gets only its own sub-communicator

Example: MPI_Comm_split()

All processes with same color are grouped into separate sub-communicators

Creation is **collective** in the **old** communicator.

C

- C/C++: int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm **newcomm*)

Fortran

- Fortran: MPI_COMM_SPLIT (comm, color, key, *newcomm*, *ierror*)

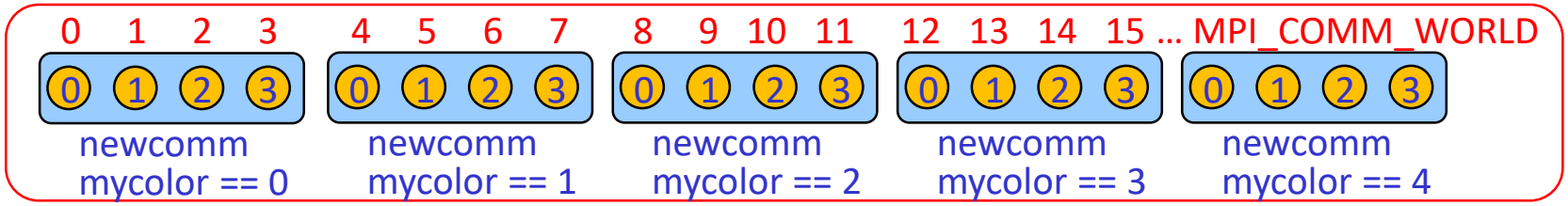
  mpi_f08:
  ```
  TYPE(MPI_Comm) :: comm, newcomm
  INTEGER :: color, key;
  INTEGER, OPTIONAL :: ierror
  ```

  mpi & mpif.h:
  ```
  INTEGER comm, color, key, newcomm, ierror
  ```

Each process gets only its own sub-communicator

Python

- Python: *newcomm* = comm.Split(color=0, key=0)



Example: MPI_Comm_split()

Creation is **collective** in the **old** communicator.

All processes with same color are grouped into separate sub-communicators

C

- C/C++: int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm*)

Fortran

- Fortran: MPI_COMM_SPLIT (comm, color, key, *newcomm*, *ierror*)

  mpi_f08:
  ```
  TYPE(MPI_Comm) :: comm, newcomm
  INTEGER :: color, key;
  INTEGER, OPTIONAL :: ierror
  ```

  mpi & mpif.h: INTEGER comm, color, key, newcomm, ierror

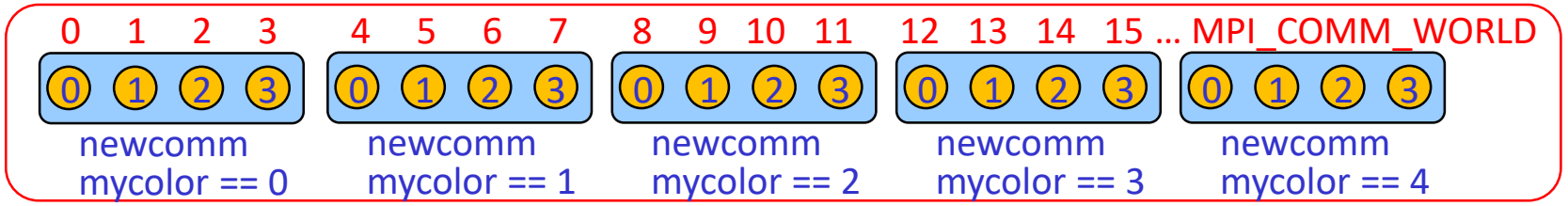
Python

- Python: *newcomm* = comm.Split(color=0, key=0)

Each process gets only its own sub-communicator

Example:

```
int my_rank, mycolor, key, my_newrank;
MPI_Comm newcomm;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
mycolor = my_rank/4;
key = 0;
MPI_Comm_split(MPI_COMM_WORLD, mycolor, key, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```



Example: MPI_Comm_split()

Creation is **collective** in the **old** communicator.

All processes with same color are grouped into separate sub-communicators

C

- C/C++: int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm*)

Fortran

- Fortran: MPI_COMM_SPLIT (comm, color, key, *newcomm*, *ierror*)

  mpi_f08:
  ```
  TYPE(MPI_Comm) :: comm, newcomm
  INTEGER :: color, key;
  INTEGER, OPTIONAL :: ierror
  ```

  mpi & mpif.h: INTEGER comm, color, key, newcomm, ierror

Each process gets only its own sub-communicator

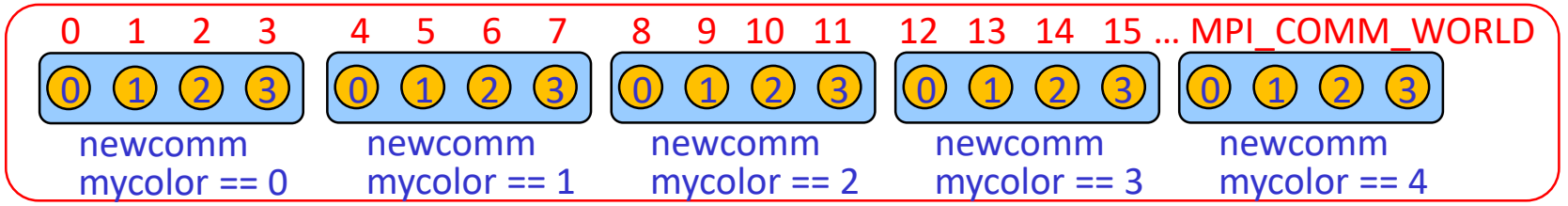
Python

- Python: *newcomm* = comm.Split(color=0, key=0)

Example:

```
int my_rank, mycolor, key, my_newrank;
MPI_Comm newcomm;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
mycolor = my_rank/4;
key = 0;
MPI_Comm_split(MPI_COMM_WORLD, mycolor, key, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

Always 4 process get same color → grouped in an own newcomm



Example: MPI_Comm_split()

Creation is **collective** in the **old** communicator.

All processes with same color are grouped into separate sub-communicators

C

- C/C++: int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm*)

Fortran

- Fortran: MPI_COMM_SPLIT (comm, color, key, *newcomm*, *ierror*)

  mpi_f08:
  ```
  TYPE(MPI_Comm) :: comm, newcomm
  INTEGER :: color, key;
  INTEGER, OPTIONAL :: ierror
  ```

  mpi & mpif.h: INTEGER comm, color, key, newcomm, ierror

Python

- Python: *newcomm* = comm.Split(color=0, key=0)

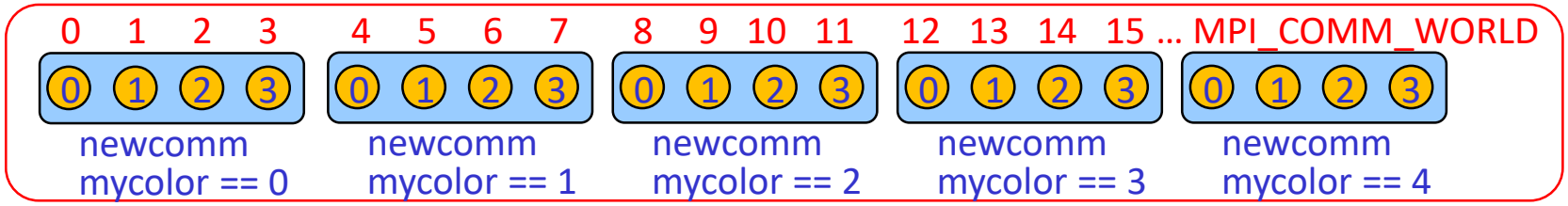
Each process gets only its own sub-communicator

Example:

```
int my_rank, mycolor, key, my_newrank;
MPI_Comm newcomm;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
mycolor = my_rank/4;
key = 0;
MPI_Comm_split(MPI_COMM_WORLD, mycolor, key, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

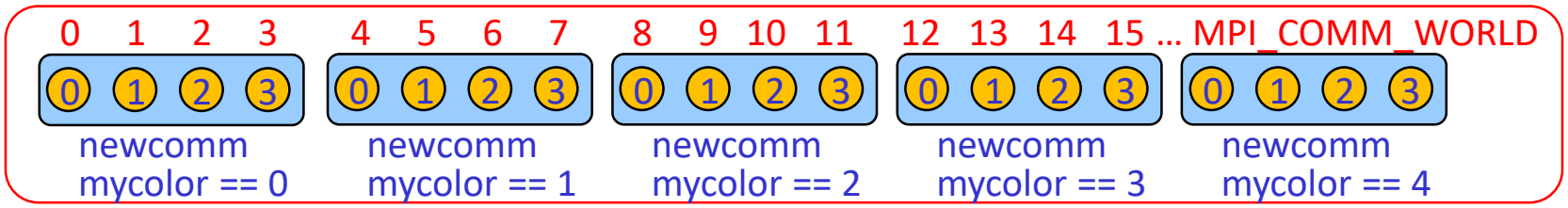
Always 4 process get same color → grouped in an own newcomm

key==0 → ranking in newcomm is sorted as in old comm
key ≠ 0 → ranking in newcomm is sorted according key values



Example: MPI_Group_range_incl() + MPI_Comm_create()

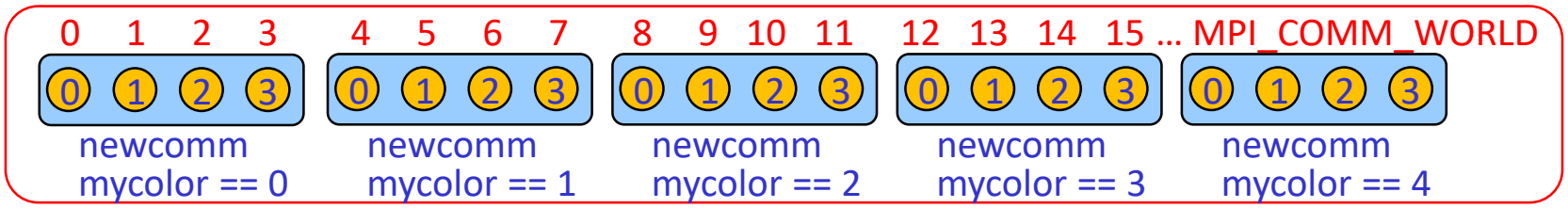
```
int my_rank, mycolor, my_newrank, ranges[1][3];
MPI_Group world_group, sub_group;
MPI_Comm newcomm;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_group(MPI_COMM_WORLD, &world_group);

mycolor = my_rank/4;
/* first rank of my range:*/ ranges[0][0] = mycolor*4;
/* last  rank of my range:*/ ranges[0][1] = mycolor*4 + (4-1);
/* stride of ranks:       */ ranges[0][2] = 1;
MPI_Group_range_incl ( world_group, 1, ranges, &sub_group);

MPI_Comm_create(MPI_COMM_WORLD, sub_group, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```



Example: MPI_Group_range_incl() + MPI_Comm_create()

```
int my_rank, mycolor, my_newrank, ranges[1][3];
MPI_Group world_group, sub_group;
MPI_Comm newcomm;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_group(MPI_COMM_WORLD, &world_group);

mycolor = my_rank/4;
/* first rank of my range:*/ ranges[0][0] = mycolor*4;
/* last  rank of my range:*/ ranges[0][1] = mycolor*4 + (4-1);
/* stride of ranks:       */ ranges[0][2] = 1;
MPI_Group_range_incl ( world_group, 1, ranges, &sub_group);

MPI_Comm_create(MPI_COMM_WORLD, sub_group, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

Group of the processes in MPI_COMM_WORLD.
Group and sub-group creation is **local** (non-collective).

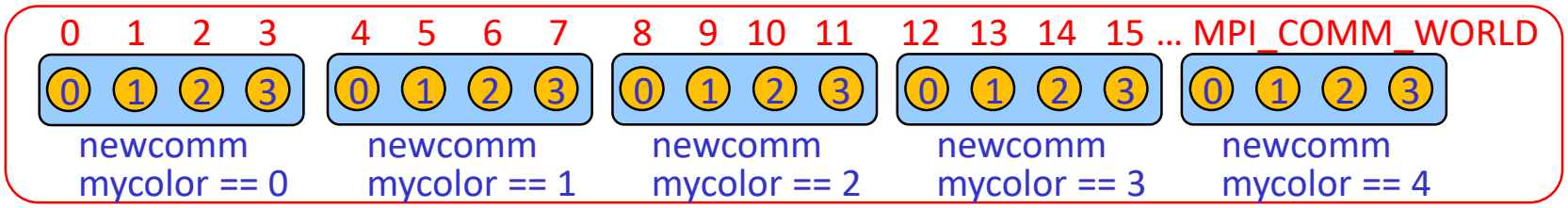


Example: MPI_Group_range_incl() + MPI_Comm_create()

```
int my_rank, mycolor, my_newrank, ranges[1][3];
MPI_Group world_group, sub_group;
MPI_Comm newcomm;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_group(MPI_COMM_WORLD, &world_group)

mycolor = my_rank/4;
/* first rank of my range:*/ ranges[0][0] = mycolor*4;
/* last  rank of my range:*/ ranges[0][1] = mycolor*4 + (4-1);
/* stride of ranks:       */ ranges[0][2] = 1;
MPI_Group_range_incl ( world_group, 1, ranges, &sub_group);

MPI_Comm_create(MPI_COMM_WORLD, sub_group, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

Group of the processes in MPI_COMM_WORLD.
Group and sub-group creation is **local** (non-collective).

Always 4 process get same color
→ **grouped in an own sub_group**
→ grouped in an own newcomm



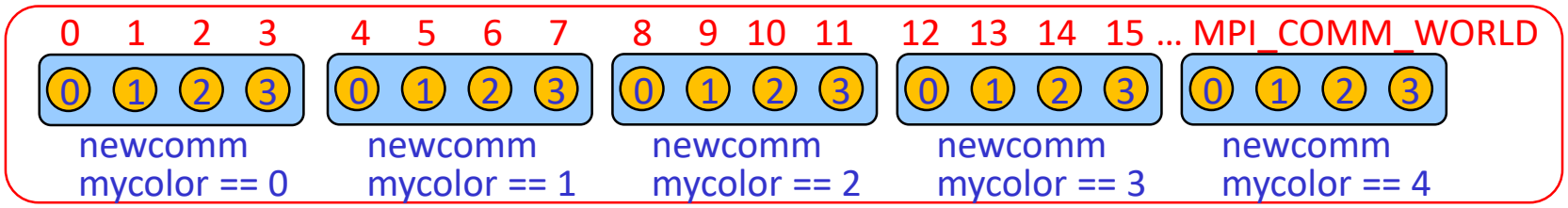
Example: MPI_Group_range_incl() + MPI_Comm_create()

```
int my_rank, mycolor, my_newrank, ranges[1][3];
MPI_Group world_group, sub_group;
MPI_Comm newcomm;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_group(MPI_COMM_WORLD, &world_group)

mycolor = my_rank/4;
/* first rank of my range:*/ ranges[0][0] = mycolor*4;
/* last  rank of my range:*/ ranges[0][1] = mycolor*4 + (4-1);
/* stride of ranks:       */ ranges[0][2] = 1;
MPI_Group_range_incl ( world_group, 1, ranges, &sub_group);

MPI_Comm_create(MPI_COMM_WORLD, sub_group, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

Only one range

Group of the processes in MPI_COMM_WORLD.
Group and sub-group creation is **local** (non-collective).

Always 4 process get same color
→ **grouped in an own sub_group**
→ grouped in an own newcomm



Example: MPI_Group_range_incl() + MPI_Comm_create()

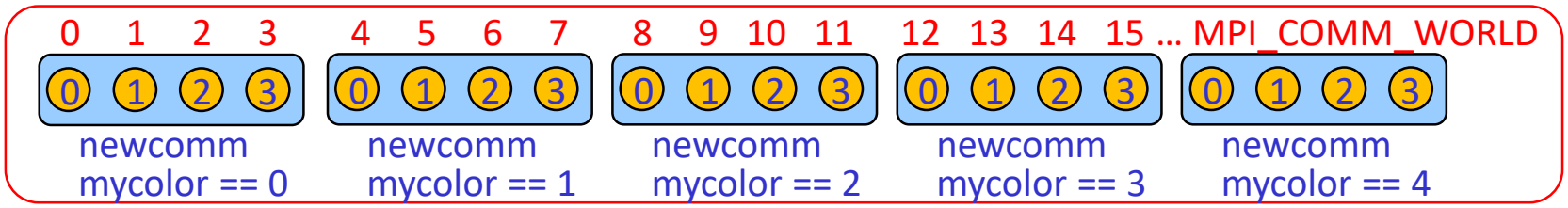
```
int my_rank, mycolor, my_newrank, ranges[1][3];
MPI_Group world_group, sub_group;
MPI_Comm newcomm;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_group(MPI_COMM_WORLD, &world_group)

mycolor = my_rank/4;
/* first rank of my range:*/ ranges[0][0] = mycolor*4;
/* last  rank of my range:*/ ranges[0][1] = mycolor*4 + (4-1);
/* stride of ranks:       */ ranges[0][2] = 1;
MPI_Group_range_incl ( world_group, 1, ranges, &sub_group);

MPI_Comm_create(MPI_COMM_WORLD, sub_group, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

Only one range

Three values per range:
[0]: first rank
[1]: last rank
[2]: stride

Group of the processes in MPI_COMM_WORLD.
Group and sub-group creation is **local** (non-collective).

Always 4 process get same color
→ **grouped in an own sub_group**
→ grouped in an own newcomm



Example: MPI_Group_range_incl() + MPI_Comm_create()

```
int my_rank, mycolor, my_newrank, ranges[1][3];
MPI_Group world_group, sub_group;
MPI_Comm newcomm;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_group(MPI_COMM_WORLD, &world_group)

mycolor = my_rank/4;
/* first rank of my range:*/ ranges[0][0] = mycolor*4;
/* last  rank of my range:*/ ranges[0][1] = mycolor*4 + (4-1);
/* stride of ranks:       */ ranges[0][2] = 1;
MPI_Group_range_incl ( world_group, 1, ranges, &sub_group);

MPI_Comm_create(MPI_COMM_WORLD, sub_group, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

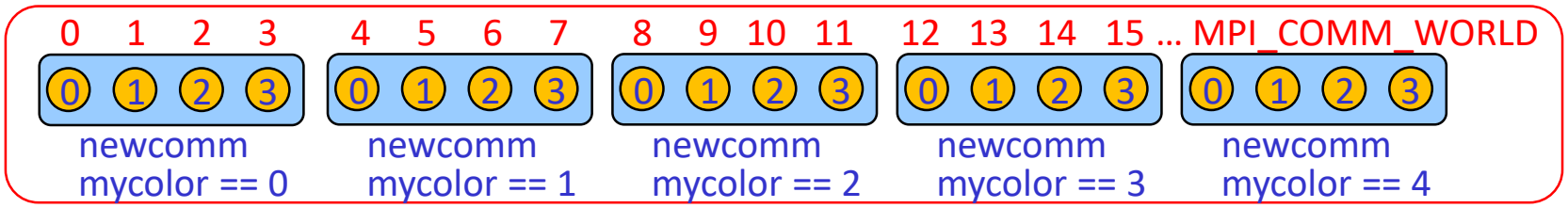
Only one range

Three values per range:
[0]: first rank
[1]: last rank
[2]: stride

Group of the processes in MPI_COMM_WORLD.
Group and sub-group creation is **local** (non-collective).

Always 4 process get same color
→ **grouped in an own sub_group**
→ grouped in an own newcomm

Must be restricted to < **num_procs**



Example: MPI_Group_range_incl() + MPI_Comm_create()

```
int my_rank, mycolor, my_newrank, ranges[1][3];
MPI_Group world_group, sub_group;
MPI_Comm newcomm;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_group(MPI_COMM_WORLD, &world_group)

mycolor = my_rank/4;
/* first rank of my range:*/ ranges[0][0] = mycolor*4;
/* last  rank of my range:*/ ranges[0][1] = mycolor*4 + (4-1);
/* stride of ranks:       */ ranges[0][2] = 1;
MPI_Group_range_incl ( world_group, 1, ranges, &sub_group);

MPI_Comm_create(MPI_COMM_WORLD, sub_group, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

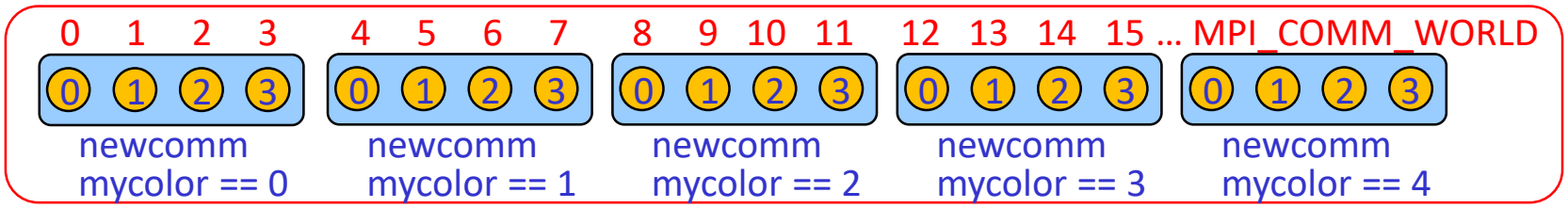
Only one range

Three values per range:
[0]: first rank
[1]: last rank
[2]: stride

Group of the processes in MPI_COMM_WORLD.
Group and sub-group creation is **local** (non-collective).

Always 4 process get same color
→ **grouped in an own sub_group**
→ grouped in an own newcomm

Must be restricted to < **num_procs**



Example: MPI_Group_range_incl() + MPI_Comm_create()

```
int my_rank, mycolor, my_newrank, ranges[1][3];
MPI_Group world_group, sub_group;
MPI_Comm newcomm;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_group(MPI_COMM_WORLD, &world_group)

mycolor = my_rank/4;
/* first rank of my range:*/ ranges[0][0] = mycolor*4;
/* last  rank of my range:*/ ranges[0][1] = mycolor*4 + (4-1);
/* stride of ranks:       */ ranges[0][2] = 1;
MPI_Group_range_incl ( world_group, 1, ranges, &sub_group);

MPI_Comm_create(MPI_COMM_WORLD, sub_group, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

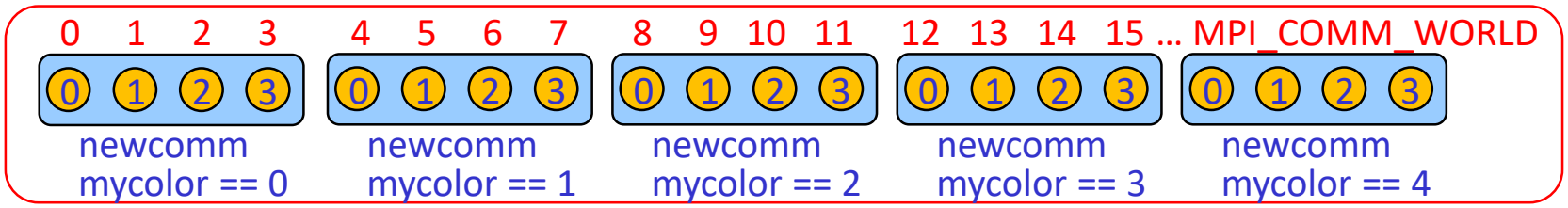
Only one range

Three values per range:
[0]: first rank
[1]: last rank
[2]: stride

Group of the processes in MPI_COMM_WORLD.
Group and sub-group creation is **local** (non-collective).

Always 4 process get same color
→ **grouped in an own sub_group**
→ grouped in an own newcomm

Must be restricted to < **num_procs**



Example: MPI_Group_range_incl() + MPI_Comm_create()

```
int my_rank, mycolor, my_newrank, ranges[1][3];
MPI_Group world_group, sub_group;
MPI_Comm newcomm;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_group(MPI_COMM_WORLD, &world_group)

mycolor = my_rank/4;
/* first rank of my range:*/ ranges[0][0] = mycolor*4;
/* last  rank of my range:*/ ranges[0][1] = mycolor*4 + (4-1);
/* stride of ranks:       */ ranges[0][2] = 1;
MPI_Group_range_incl ( world_group, 1, ranges, &sub_group);

MPI_Comm_create(MPI_COMM_WORLD, sub_group, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

Only one range

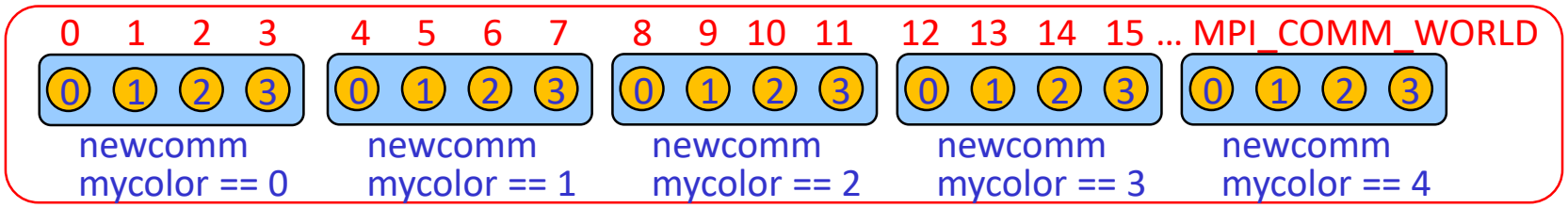
Three values per range:
[0]: first rank
[1]: last rank
[2]: stride

Group of the processes in MPI_COMM_WORLD.
Group and sub-group creation is **local** (non-collective).

Always 4 process get same color
→ **grouped in an own sub_group**
→ grouped in an own newcomm

Must be restricted to < **num_procs**

(Sub-)communicator creation is **collective**.

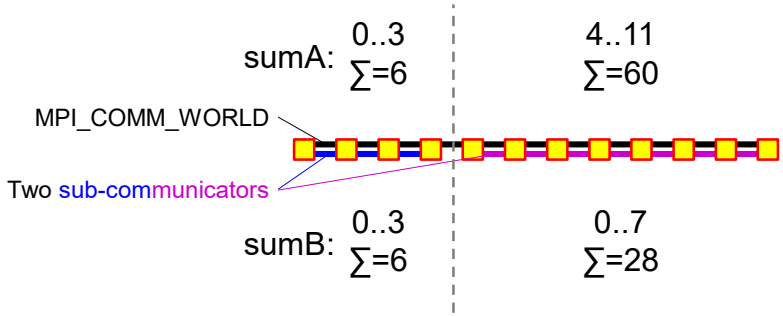


Exercises 1+2

Exercise 1 — Two independent sub-communicators

In MPI/tasks/…

- Use C C/Ch8/comm-split-skel.c or Fortran F_30/Ch8/comm-split-skel_30.f90 or Python PY/Ch8/comm-split-skel.py
- Modify the *allreduce* program:
  - Split the communicator into 1/3 and 2/3, e.g., with color = (rank > $\left\lfloor \frac{\text{size}-1}{3} \right\rfloor$) as input for **MPI_Comm_split**
  - Calculate **sumA** and **sumB** over all processes within each **sub-communicator**
  - sumA: **ranks in MPI_COMM_WORLD** (but summed up only within each sub-communicator)
    - E.g., with 12 processes → split into 4 & 8 with world ranks 0..3 & 4..11 and sums 6 & 60 → sumA
  - sumB: **ranks in new sub-communicators** (and summed up only within each sub-comm.)
    - E.g., with 12 processes → split into 4 & 8 with sub-comm ranks 0..3 & 0..7 and sums 6 & 28 → sumB
  - Use `mpirun … | sort +2n -3`



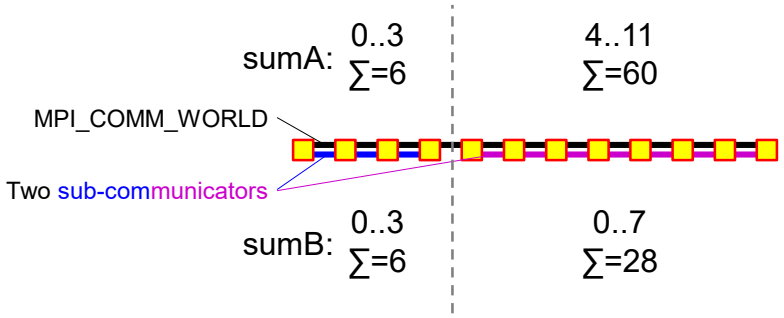

```
Expected results with 12 processes:
PE world:  0, color=0 sub:  0, SumA=  6, SumB=  6 in sub_comm
PE world:  1, color=0 sub:  1, SumA=  6, SumB=  6 in sub_comm
PE world:  2, color=0 sub:  2, SumA=  6, SumB=  6 in sub_comm
PE world:  3, color=0 sub:  3, SumA=  6, SumB=  6 in sub_comm
PE world:  4, color=1 sub:  0, SumA= 60, SumB= 28 in sub_comm
PE world:  5, color=1 sub:  1, SumA= 60, SumB= 28 in sub_comm
PE world:  6, color=1 sub:  2, SumA= 60, SumB= 28 in sub_comm
PE world:  7, color=1 sub:  3, SumA= 60, SumB= 28 in sub_comm
PE world:  8, color=1 sub:  4, SumA= 60, SumB= 28 in sub_comm
PE world:  9, color=1 sub:  5, SumA= 60, SumB= 28 in sub_comm
PE world: 10, color=1 sub:  6, SumA= 60, SumB= 28 in sub_comm
PE world: 11, color=1 sub:  7, SumA= 60, SumB= 28 in sub_comm
```

Exercise 2 (advanced) — MPI_Comm_create

- Use C C/Ch8/comm-create-skel.c or Fortran F_30/Ch8/comm-create-skel_30.f90 or Python PY/Ch8/comm-create-skel.py

Python
```
group = comm.Get_group()
sub_group = group.Range_incl(ranges)
sub_comm = comm.Create(sub_group)
```

- Same as Exercise 1, but with **MPI_Comm_group()**, **MPI_Group_range_incl()**, and **MPI_Comm_create()**
  - instead of MPI_Comm_split()
  - Two different ranges for color 0 and 1 !!!
  - Same results in sumA/B as in Exercise 1
- Same details as in Exercise 1:
  - Split the communicator into 1/3 and 2/3, e.g., with color = (rank > $\left\lfloor \frac{\text{size}-1}{3} \right\rfloor$)
  - Calculate **sumA** and **sumB** over all processes within each **sub-communicator**
  - sumA: **ranks in MPI_COMM_WORLD** (but summed up only within each sub-communicator)
  - sumB: **ranks in new sub-communicators** (and summed up only within each sub-comm.)
  - Use `mpirun … | sort +2n -3`



Expected results with 12 processes:

```
PE world:  0, color=0 sub:  0, SumA=  6, SumB=  6 in sub_comm
PE world:  1, color=0 sub:  1, SumA=  6, SumB=  6 in sub_comm
PE world:  2, color=0 sub:  2, SumA=  6, SumB=  6 in sub_comm
PE world:  3, color=0 sub:  3, SumA=  6, SumB=  6 in sub_comm
PE world:  4, color=1 sub:  0, SumA= 60, SumB= 28 in sub_comm
PE world:  5, color=1 sub:  1, SumA= 60, SumB= 28 in sub_comm
PE world:  6, color=1 sub:  2, SumA= 60, SumB= 28 in sub_comm
PE world:  7, color=1 sub:  3, SumA= 60, SumB= 28 in sub_comm
PE world:  8, color=1 sub:  4, SumA= 60, SumB= 28 in sub_comm
PE world:  9, color=1 sub:  5, SumA= 60, SumB= 28 in sub_comm
PE world: 10, color=1 sub:  6, SumA= 60, SumB= 28 in sub_comm
PE world: 11, color=1 sub:  7, SumA= 60, SumB= 28 in sub_comm
```

Quiz on Chapter 8-(1) – Groups & Communicators

A. Why should you use
   - A duplicate of MPI_COMM_WORLD?
   - Sub-communicators?
   - Inter-communicators?

B. Which is the easiest way to build a set of disjoint subcommunicators?

   _____

C. What are the major differences between
   - a group of processes referenced by a group handle, and
   - a communicator referenced by a communicator handle?

D. Can you produce with one call to MPI_Comm_create [single choice question]
   - Only one subcommunicator?
   - One or more disjoint subcommunicators?
   - One or more overlapping subcommunicators?

E. If you split a communicator in five subcommunicators,
   must you then use an array for 5 handles as *newcomm* output argument
   instead of only a single *newcomm* handle variable?

Changing (= reordering / = re-numbering) ranks of a communicator

- Same rank-mapping provided by all processes:
  - communicator → MPI_Comm_group() → group handle
  - group handle → MPI_Group_incl(mapping_array) → reordered group
  - Communicator + reordered group → MPI_Comm_create() → reordered comm.

- Each process provides its new rank:
  - MPI_Comm_split (comm_old, /*color=*/ 0, /*key=*/ new_rank, &comm_new);

see *Advice to implementors* of MPI_CART_MAP, MPI-4.0, Sec. 8.5.8, page 415, lines 33-38

Inter-communicator – combines a local and a remote communicator

Perfect for any communication between the processes of two groups (e.g., ocean and weather)

The tag must be the same on all processes.
Creating several inter-comms, see example in the MPI standard.
Since MPI-3.0: no conflict with pt-to-pt message tags.

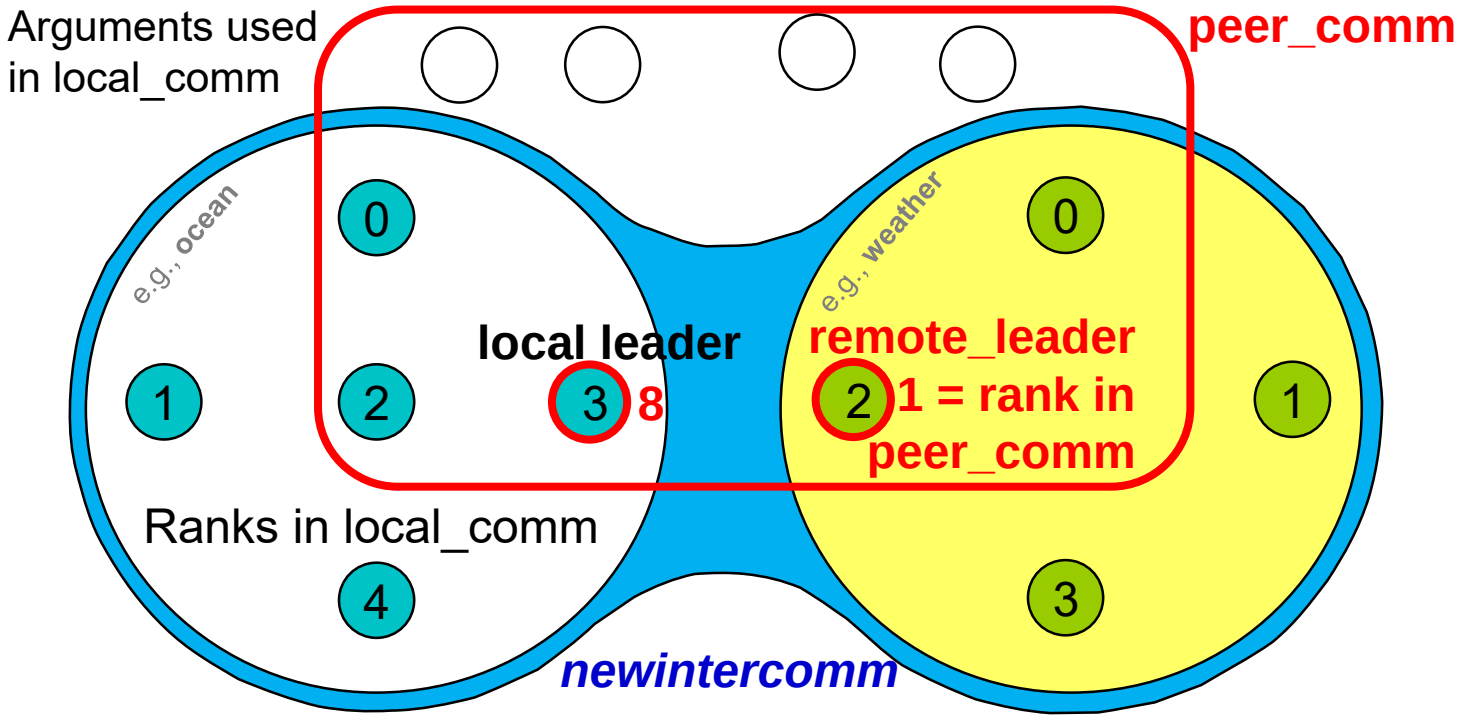
MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag, *newintercomm*)

PY: *newintercomm* = local_comm.Create_intercomm(local_leader, peer_comm, remote_leader, tag)

In left processes: 3
In right processes: 2

Significant only in left proc. 3 and right process 2

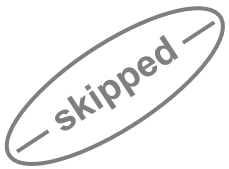
1
8



The arguments in the remote communicator are defined in the same way, but local and remote role is interchanged.

Inter-communicator – Accessors

- Which routines can be applied for inter-communicator handles?
  - **MPI_Comm_Size**, **MPI_Comm_rank**
    return same result as if applied to the local group
    → **used in next Exercise 3**
  - MPI_Comm_inter_test(comm, flag)
    returns true in flag if comm is an inter-communicator
  - MPI_Comm_remote_size(inter_comm, size)
    returns the size of the remote group
  - MPI_Comm_group
    returns the local group
  - MPI_Comm_remote_group(inter_comm, group)
    return the remote group
  - MPI_Comm_compare, see MPI-3.1 Chap. 6.6.1 or MPI-4.0 Chap. 7.6.1

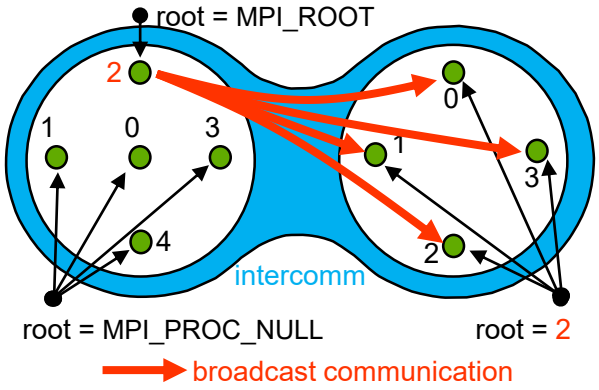


Collective Operations for Intercommunicators

- Most collective operations are extended by an additional functionality for intercommunicators, e.g.,

Since MPI-2.0

  - Bcast on a *parents-children* intercommunicator: Sends data
    - from one *parent* process
    - to all *children*

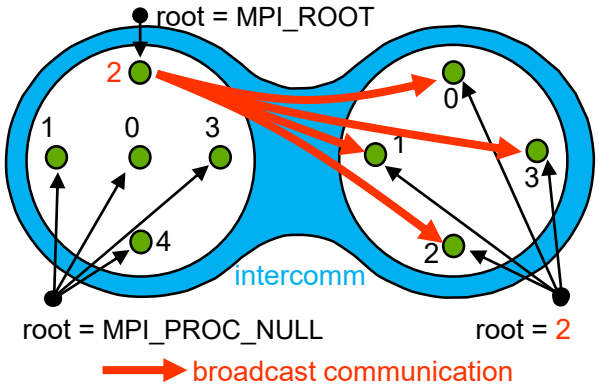


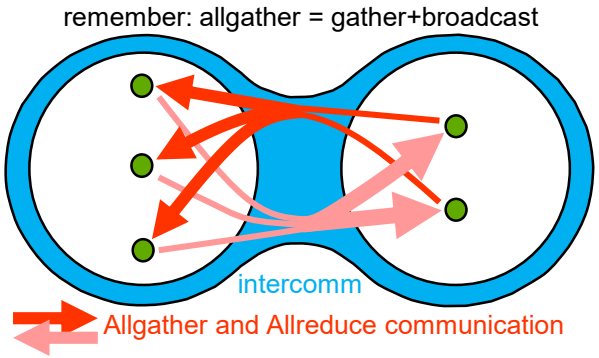
Collective Operations for Intercommunicators

- Most collective operations are extended by an additional functionality for intercommunicators, e.g.,

Since MPI-2.0

  - Bcast on a *parents-children* intercommunicator: Sends data
    - from one *parent* process
    - to all *children*
  - MPI_Allgather and MPI_Allreduce:
    - collects on each group
    - and sends it to the other group





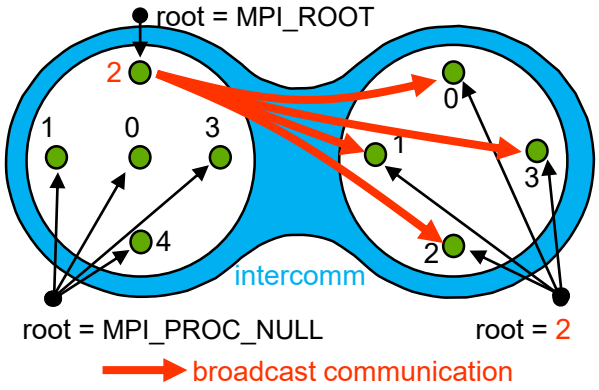
Collective Operations for Intercommunicators

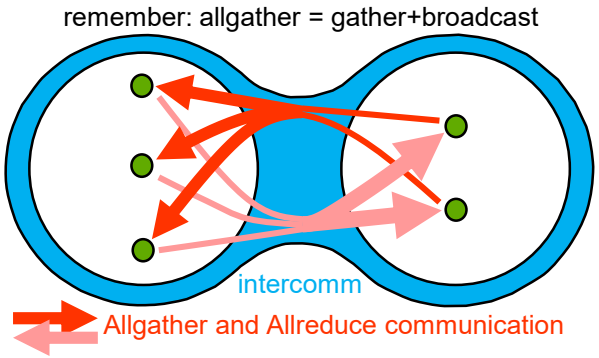
- Most collective operations are extended by an additional functionality for intercommunicators, e.g.,

Since MPI-2.0

  - Bcast on a *parents-children* intercommunicator: Sends data
    - from one *parent* process
    - to all *children*
  - MPI_Allgather and MPI_Allreduce:
    - collects on each group
    - and sends it to the other group

- Intercommunicators do not apply in
  - MPI_(I)(Ex)Scan,
  - MPI_(I)Neighbor_allgather(v)
  - MPI_(I)Neighbor_alltoall(v,w)





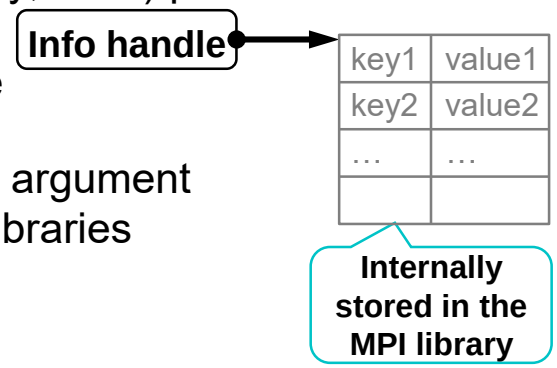
MPI_Info Object

A general service for many MPI procedures

MPI_Info Object

A general service for many MPI procedures

- An **`MPI_Info`** is an opaque object that consists of a set of (key,value) pairs
  - Both key and value are **strings**
  - A **key** should have a **unique** name within one info handle
  - Several keys are reserved by standard / implementation
  - Portable programs may use **`MPI_INFO_NULL`** as the info argument
  - Vendor keys are also portable, may be ignored by other libraries
  - Several sets of vendor-specific keys may be used

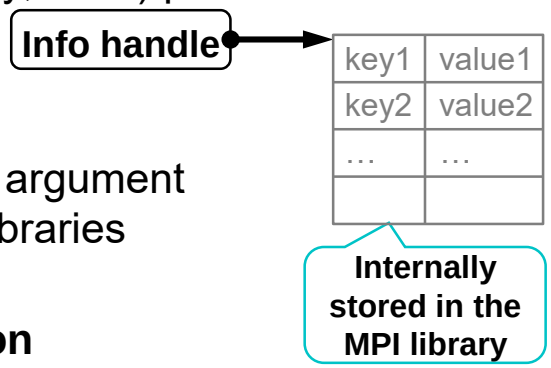


MPI_Info Object

A general service for many MPI procedures

- An **`MPI_Info`** is an opaque object that consists of a set of (key,value) pairs
  - Both key and value are **strings**
  - A **key** should have a **unique** name within one info handle
  - Several keys are reserved by standard / implementation
  - Portable programs may use **`MPI_INFO_NULL`** as the info argument
  - Vendor keys are also portable, may be ignored by other libraries
  - Several sets of vendor-specific keys may be used
- Allows applications to **pass environment-specific information**
- Allow applications to **provide assertions** regarding their usage of MPI objects and operations →to improve performance or resource utilization

New in MPI-4.0



MPI_Info Object

A general service for many MPI procedures

- An **`MPI_Info`** is an opaque object that consists of a set of (key,value) pairs
  - Both key and value are **strings**
  - A **key** should have a **unique** name within one info handle
  - Several keys are reserved by standard / implementation
  - Portable programs may use **`MPI_INFO_NULL`** as the info argument
  - Vendor keys are also portable, may be ignored by other libraries
  - Several sets of vendor-specific keys may be used
- Allows applications to **pass environment-specific information**
- Allow applications to **provide assertions** regarding their usage of MPI objects and operations →to improve performance or resource utilization
- Several functions provided to manipulate the info objects

Info handle →

key1	value1
key2	value2
…	…

Internally stored in the MPI library

New in MPI-4.0

Adds 1 new entry, or modifies the value if key already exists

Example:

```
MPI_Info  info_noncontig;
MPI_Info_create (&info_noncontig);
MPI_Info_set (info_noncontig,
              "alloc_shared_noncontig", "true");
MPI_Win_allocate_shared (…,  info_noncontig, …);
```

Creates the list with 0 entries

MPI_Info Object

A general service for many MPI procedures

- An **`MPI_Info`** is an opaque object that consists of a set of (key,value) pairs
  - Both key and value are **strings**
  - A **key** should have a **unique** name within one info handle
  - Several keys are reserved by standard / implementation
  - Portable programs may use **`MPI_INFO_NULL`** as the info argument
  - Vendor keys are also portable, may be ignored by other libraries
  - Several sets of vendor-specific keys may be used

Info handle →

key1	value1
key2	value2
…	…

Internally stored in the MPI library

- Allows applications to **pass environment-specific information**
- Allow applications to **provide assertions** regarding their usage of MPI objects and operations →to improve performance or resource utilization

New in MPI-4.0

- Several functions provided to manipulate the info objects
- Used in:
  - *Process Creation*,
  - *Window Creation*,
  - *MPI-I/O,*
  - *MPI_Comm_(i)dup_with_info,*
  - *MPI_INFO_ENV*

New in MPI-4.0

Adds 1 new entry, or modifies the value if key already exists

Example:

```
MPI_Info  info_noncontig;
MPI_Info_create (&info_noncontig);
MPI_Info_set (info_noncontig,
              "alloc_shared_noncontig", "true");
MPI_Win_allocate_shared (…,  info_noncontig, …);
```

Creates the list with 0 entries

MPI_Info Object

A general service for many MPI procedures

- An **`MPI_Info`** is an opaque object that consists of a set of (key,value) pairs
  - Both key and value are **strings**
  - A **key** should have a **unique** name within one info handle
  - Several keys are reserved by standard / implementation
  - Portable programs may use **`MPI_INFO_NULL`** as the info argument
  - Vendor keys are also portable, may be ignored by other libraries
  - Several sets of vendor-specific keys may be used

Info handle →

key1	value1
key2	value2
…	…

Internally stored in the MPI library

- Allows applications to **pass environment-specific information**
- Allow applications to **provide assertions** regarding their usage of MPI objects and operations →to improve performance or resource utilization

New in MPI-4.0

- Several functions provided to manipulate the info objects
- Used in:
  - *Process Creation*,
  - *Window Creation*,
  - *MPI-I/O,*
  - *MPI_Comm_(i)dup_with_info,*
  - *MPI_INFO_ENV*

New in MPI-4.0

Adds 1 new entry, or modifies the value if key already exists

Example:

```
MPI_Info  info_noncontig;
MPI_Info_create (&info_noncontig);
MPI_Info_set (info_noncontig,
              "alloc_shared_noncontig", "true");
MPI_Win_allocate_shared (…,  info_noncontig, …);
```

Creates the list with 0 entries

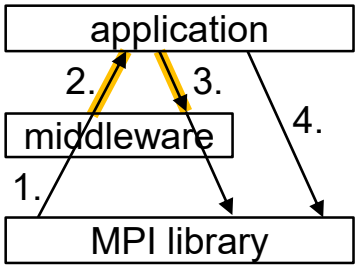
- The key/value list returned by MPI_**Comm|File|Win_get**_info in the handle may differ from a those set by the application during Comm|File|Win creation or stored with MPI_Comm|File|Win_**set**_info: The MPI library may or may not set or recognize some (system specific) hints.

New in MPI-4.0: Use MPI_Info_get_string instead of deprecated MPI_Info_get_valuelen and MPI_Info_get.

An interesting method
for middle-ware developers

Naming & attribute caching

Problem:

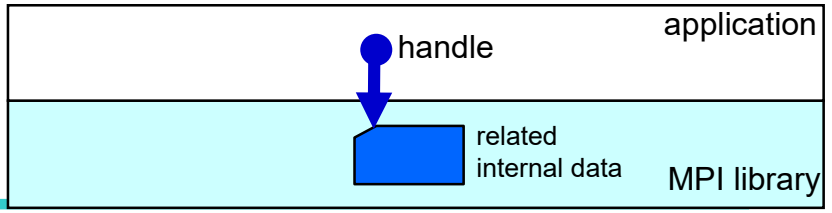


1. The MPI library provides communicators for a middleware, and
2. the middleware hands it over to the application,
3. which gives it back to the middleware, or
4. the MPI library,

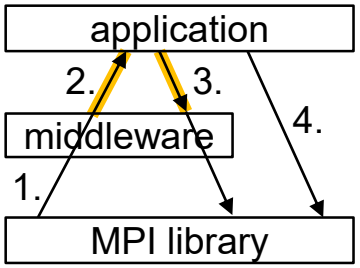
and the middleware wants to remember middleware-specific data — with such a communicator handle

An interesting method
for middle-ware developers

Naming & attribute caching



Problem:

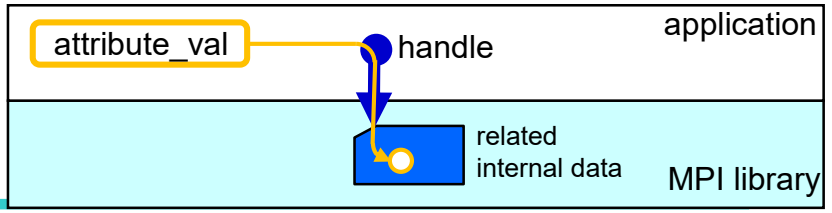


1. The MPI library provides communicators for a middleware, and
2. the middleware hands it over to the application,
3. which gives it back to the middleware, or
4. the MPI library,

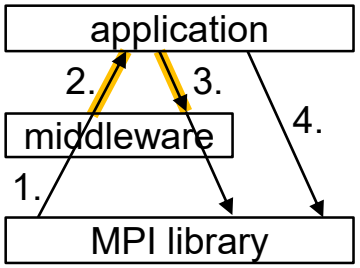
and the middleware wants to remember middleware-specific data — with such a communicator handle

Naming & attribute caching

An interesting method
for middle-ware developers



Problem:

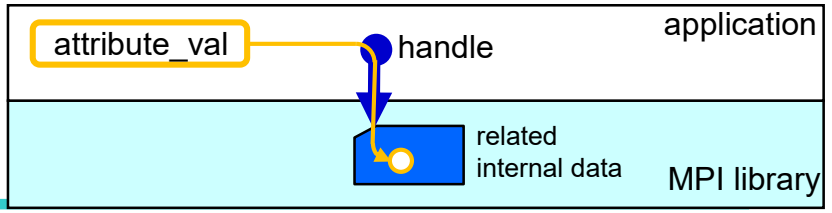


1. The MPI library provides communicators for a middleware, and
2. the middleware hands it over to the application,
3. which gives it back to the middleware, or
4. the MPI library,

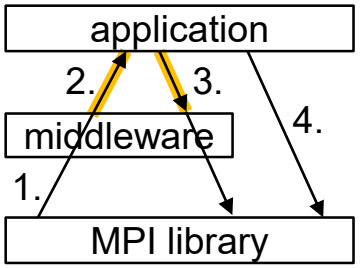
and the middleware wants to remember middleware-specific data — with such a communicator handle

Naming & attribute caching

An interesting method
for middle-ware developers



Problem:



1. The MPI library provides communicators for a middleware, and
2. the middleware hands it over to the application,
3. which gives it back to the middleware, or
4. the MPI library,

and the middleware wants to remember middleware-specific data — with such a communicator handle

Caching attributes on handles in two steps:

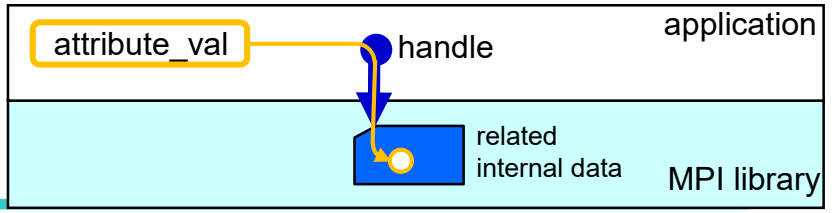
- 1st step – generating a keyval:
  - MPI_Comm_create_keyval (comm_copy_attr_fn, comm_delete_attr_fn, ***comm_keyval***, extra_state)
- 2nd step – storing & retrieving an attribute on/from a communicator handle:
  - MPI_Comm_set_attr (comm, comm_keyval, **attribute_val**)
  - MPI_Comm_get_attr (comm, comm_keyval, ***attribute_val***, *flag*)
- Other routines:
  - MPI_Comm_delete_attr & MPI_Comm_free_keyval

Other objects: Same method for **datatypes** and **windows**

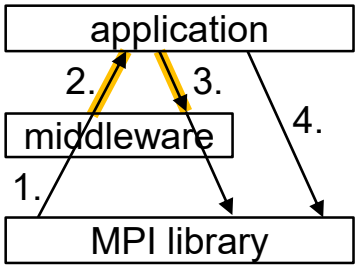
Examples: See MPI-3.1/MPI-4.0 Sect. 17.2.7/19.3.7 *Attributes*

Naming & attribute caching

An interesting method
for middle-ware developers



Problem:



1. The MPI library provides communicators for a middleware, and
2. the middleware hands it over to the application,
3. which gives it back to the middleware, or
4. the MPI library,

and the middleware wants to remember middleware-specific data — with such a communicator handle

Caching attributes on handles in two steps:

- 1st step – generating a keyval:
  - MPI_Comm_create_keyval (comm_copy_attr_fn, comm_delete_attr_fn, ***comm_keyval***, extra_state)
- 2nd step – storing & retrieving an attribute on/from a communicator handle:
  - MPI_Comm_set_attr (comm, comm_keyval, **attribute_val**)
  - MPI_Comm_get_attr (comm, comm_keyval, ***attribute_val***, *flag*)
- Other routines:
  - MPI_Comm_delete_attr & MPI_Comm_free_keyval

Other objects: Same method for **datatypes** and **windows**

Examples: See MPI-3.1/MPI-4.0 Sect. 17.2.7/19.3.7 *Attributes*

Name an object:

- MPI_Comm_set_name(comm, comm_name), MPI_Comm_get_name(…)

Environment inquiry – implementation information (1)

Version of MPI

- Compile time information, e.g.,
  - integer MPI_VERSION=3, MPI_SUBVERSION=1
  - Valid pairs: (4,1), (4,0), (3,1), (3,0), (2,2), (2,1), (2,0), and (1,2).
- Runtime information
  - MPI_Get_version(*version, subversion*)
  - Can be called before MPI_Init and after MPI_Finalize

Environment inquiry – implementation information (1)

Version of MPI

- Compile time information, e.g.,
  - integer MPI_VERSION=3, MPI_SUBVERSION=1
  - Valid pairs: (4,1), (4,0), (3,1), (3,0), (2,2), (2,1), (2,0), and (1,2).
- Runtime information
  - MPI_Get_version(*version, subversion*)
  - Can be called before MPI_Init and after MPI_Finalize

New in MPI-3.0

Inquire start environment

- Predefined info object **MPI_INFO_ENV** (in the World Model)
  or info handle created with **MPI_Info_create_env** (in the Sessions Model)
  holds arguments from
  - mpiexec, or
  - MPI_COMM_SPAWN

New in MPI-4.0

see a few slides later

Environment inquiry – implementation information (1)

Version of MPI

- Compile time information, e.g.,
  - integer MPI_VERSION=3, MPI_SUBVERSION=1
  - Valid pairs: (4,1), (4,0), (3,1), (3,0), (2,2), (2,1), (2,0), and (1,2).
- Runtime information
  - MPI_Get_version(*version, subversion*)
  - Can be called before MPI_Init and after MPI_Finalize

New in MPI-3.0

Inquire start environment

- Predefined info object **MPI_INFO_ENV** (in the World Model)
  or info handle created with **MPI_Info_create_env** (in the Sessions Model)
  holds arguments from
  - mpiexec, or
  - MPI_COMM_SPAWN

New in MPI-4.0

see a few slides later

Inquire processor name

- MPI_Get_processor_name(*name*, *resultlen*)

Caution: several MPI ranks may return the same name, e.g., the node name

Environment inquiry – implementation information (2)

C

Fortran

Python

Environmental inquiries

- C: MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, *&p*, *&flag*)
  - Will return in *p* a pointer to an int containing the *attribute_val*
- Fortran: MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, *attribute_val*, *flag*, *ierror*)
- Python: *attribute_val* = MPI.COMM_WORLD.Get_attr(keyval)

C: **pointer** based attributes
Fortran: **integer(kind=MPI_ADDRESS_KIND)** based attributes

- with keyval =
  - **MPI_TAG_UB** (Python: MPI.TAG_UB)
    → returns upper bound for tag values in *attribute_val*
    → must be at least 32767
  - **MPI_HOST** (deprecated in MPI-4.1)
    → returns host-rank (if exists) or MPI_PROC_NULL (if there is no host)
  - **MPI_IO**
    → returns MPI_ANY_SOURCE in *attribute_val* (if every process can provide I/O)
  - **MPI_WTIME_IS_GLOBAL**
    → returns 1 in *attribute_val* (if clocks are synchronized), otherwise, 0

Examples: see MPI-3.1, Sect. 17.2.7, page 664, line 43 – page 665, line 13 or
MPI-4.0, Sect. 19.3.7, page 852, line 29-47

Since MPI-1

New in MPI-4.0

World Model and *Sessions Model*

Since MPI-2.0: duplicates **with** associated key values, topology and **info hints**.
Since MPI-4.0: Now **without info hints**

- ***The World Model***
  - MPI_COMM_WORLD can be used between MPI_Init and MPI_Finalize
  - Exactly one call to MPI_Init and MPI_Finalize
  - Problem, if several independent software layers want to use MPI:
    - **Each layer can duplicate MPI_COMM_WORLD using MPI_COMM_DUP()**
    - **But there is no rule on which layer calls MPI_Init and which one MPI_Finalize**
- ***The Sessions Model***
  - Each independent software layer **xxx** can initialize and finalize MPI, e.g., as follows:
    - **As part of layer_xxx_init**
      - MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL, &session);
      - MPI_Group_from_session_pset(session, "mpi://WORLD", &xxx_world_group);
      - MPI_Comm_create_from_group(xxx_world_group, "stringtag_xxx", MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL, &xxx_world_comm);
      - MPI_Group_free(&xxx_world_group);
    - **As part of layer_xxx_finalize**
      - MPI_Comm_free(&xxx_world_comm);
      - MPI_Session_finalize(&session);
  - **Caution:** MPI objects derived from different MPI Session handles shall **not** be intermixed with each other in a single MPI procedure call.
- An MPI application may use the World Model (not more than once) together with the Sessions Model (with several overlapping or non-overlapping sessions)

Conclusions of this course chapter

- Sub-communicators
  - Scalability problems
    - methods with local data with O(#MPI_COMM_WORLD) are not scalable
    - e.g., MPI_Comm_group(MPI_COMM_WORLD, group)
  - Sub-communicator splitting is a scalable interface
    - This does not guarantee that an MPI implementation is scalable
  - Inter-communicators
    - mainly used in coupled applications
    - Also used for MPI_Comm_spawn
      (See course Chapter 16 *Process creation and management*)
- Info Object → used in several interfaces → MPI_INFO_NULL is always a choice
- Object naming & attribute caching – useful only for libraries between MPI and appl.
- Environment inquiry → small functionality, MPI_INFO_ENV new in MPI-3.0
- The Sessions Model → a method to init/finalize MPI within independent application components / software layers

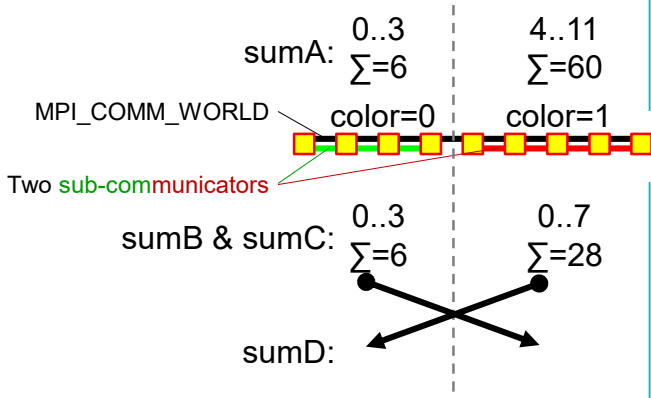
New in MPI-4.0

Exercise 3

Exercise 3 — Create an inter-communicator

In MPI/tasks/…

- Use C C/Ch8/intercomm-skel.c or Fortran F_30/Ch8/intercomm-skel_30.f90 or Python PY/Ch8/intercomm-skel.py
- Same details as in Exercise 1:
  - Split the communicator into 1/3 and 2/3, e.g., with color = (rank > $\lfloor \frac{\text{size}-1}{3} \rfloor$)
  - Calculate **sumA** and **sumB** over all processes within each **sub-communicator**
  - sumA: **ranks in MPI_COMM_WORLD** (but summed up only within each sub-communicator)
  - sumB: **ranks in new sub-communicators** (and summed up only within each sub-comm.)
  - Use `mpirun … | sort +2n -3`
- And additionally:
  - **Create an inter-communicator from the two sub-communicators**
  - **Choose rank 0 as local leader in both sub_comm**
  - **sumC**: **ranks in inter_comm** summed up over the sub_comm
  - **sumD**: **ranks in inter_comm** summed up over the **inter_comm**



Expected results with 12 processes:

```
PE world:  0, color=0 sub:  0 inter:  0 SumA=  6, SumB=  6, SumC=  6, SumD= 28
PE world:  1, color=0 sub:  1 inter:  1 SumA=  6, SumB=  6, SumC=  6, SumD= 28
PE world:  2, color=0 sub:  2 inter:  2 SumA=  6, SumB=  6, SumC=  6, SumD= 28
PE world:  3, color=0 sub:  3 inter:  3 SumA=  6, SumB=  6, SumC=  6, SumD= 28
PE world:  4, color=1 sub:  0 inter:  0 SumA= 60, SumB= 28, SumC= 28, SumD=  6
PE world:  5, color=1 sub:  1 inter:  1 SumA= 60, SumB= 28, SumC= 28, SumD=  6
PE world:  6, color=1 sub:  2 inter:  2 SumA= 60, SumB= 28, SumC= 28, SumD=  6
PE world:  7, color=1 sub:  3 inter:  3 SumA= 60, SumB= 28, SumC= 28, SumD=  6
PE world:  8, color=1 sub:  4 inter:  4 SumA= 60, SumB= 28, SumC= 28, SumD=  6
PE world:  9, color=1 sub:  5 inter:  5 SumA= 60, SumB= 28, SumC= 28, SumD=  6
PE world: 10, color=1 sub:  6 inter:  6 SumA= 60, SumB= 28, SumC= 28, SumD=  6
PE world: 11, color=1 sub:  7 inter:  7 SumA= 60, SumB= 28, SumC= 28, SumD=  6
```

Advanced Exercise 4 — MPI_TAG_UB

- Use C C/Ch8/tag-ub-skel.c or Fortran F_30/Ch8/tag-ub-skel_30.f90 or Python PY/Ch8/tag-ub-skel.py

- Goal: Inquiry the upper bound for MPI tag arguments
- Hint:
  - See the reference to the MPI standard on the previous slide on *"Environment inquiry – implementation information (2)"*

- Expected results (with `mpirun -np 1 ./a.out`)
  - `PE 0, tag_ub=2147483647, flag=1` (=2^{31}-1) with OpenMPI 2.1.6.0
  - `PE 0, tag_ub=268435455, flag=1` (=2^{28}-1) with mpich 3.2.1
  - `PE 0, tag_ub=32767, flag=1` (=2^{15}-1) at least required

Quiz on Chapter 8-(2) – Groups & Communicators

A. Where do you use inter-communicators?

   _____

B. Which data can be stored in an info handle?

   _____

C. Which rules apply to such content?

   1. _____
   2. _____
   3. _____