Parallel programming / computation

Sultan ALPAR
s.alpar@iitu.edu.kz

IITU

Lecture 5
Collective Communication

Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronization.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.
  - Neighbor communication in a virtual process grid

New in MPI-3.0

Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronization.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.
  - Neighbor communication in a virtual process grid

New in MPI-3.0

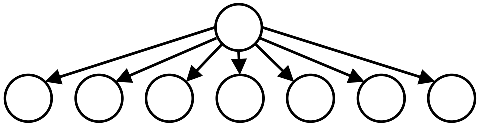
Should be faster than any programming with point-to-point messages!

Internally: tree-based algorithms

E.g., broadcast

Internally: tree-based algorithms

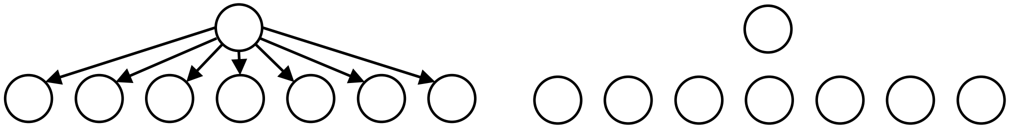
E.g., broadcast



Sequential algorithm
O(# processes)

Internally: tree-based algorithms

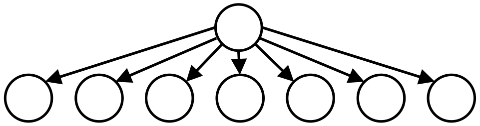
E.g., broadcast



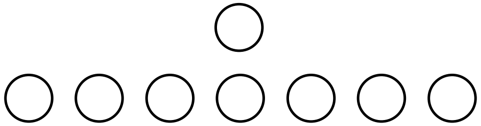
Sequential algorithm
O(# processes)

Internally: tree-based algorithms

E.g., broadcast



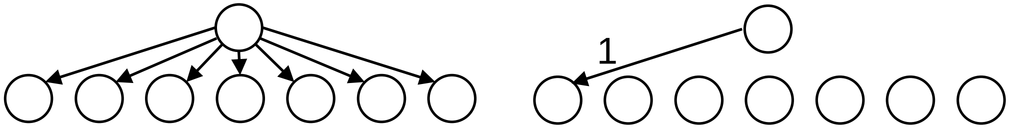
Sequential algorithm
O(# processes)



Tree based algorithm

Internally: tree-based algorithms

E.g., broadcast

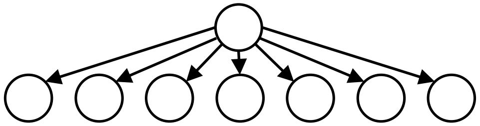


Sequential algorithm
O(# processes)

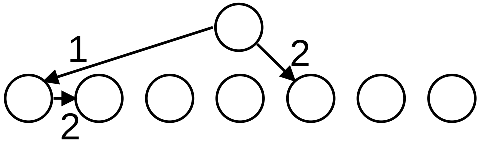
Tree based algorithm

Internally: tree-based algorithms

E.g., broadcast



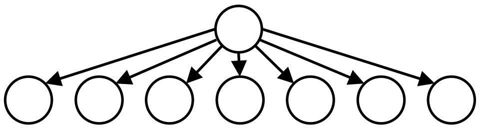
Sequential algorithm
O(# processes)



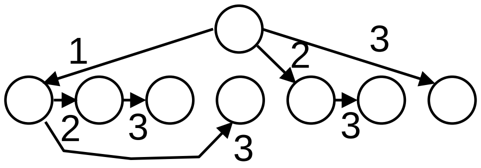
Tree based algorithm

Internally: tree-based algorithms

E.g., broadcast



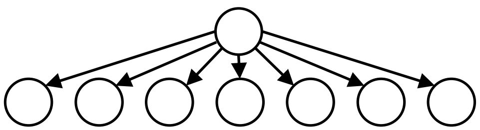
Sequential algorithm
O(# processes)



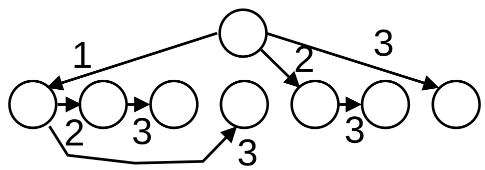
Tree based algorithm

Internally: tree-based algorithms

E.g., broadcast



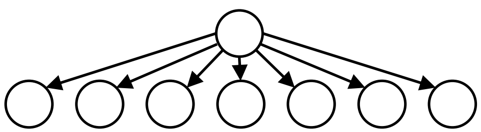
Sequential algorithm
O(# processes)



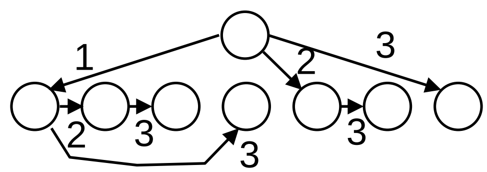
Tree based algorithm
$O(\log_2(\# \text{ processes}))$

Internally: tree-based algorithms

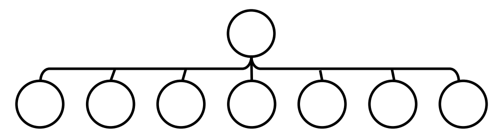
E.g., broadcast



Sequential algorithm
O(# processes)



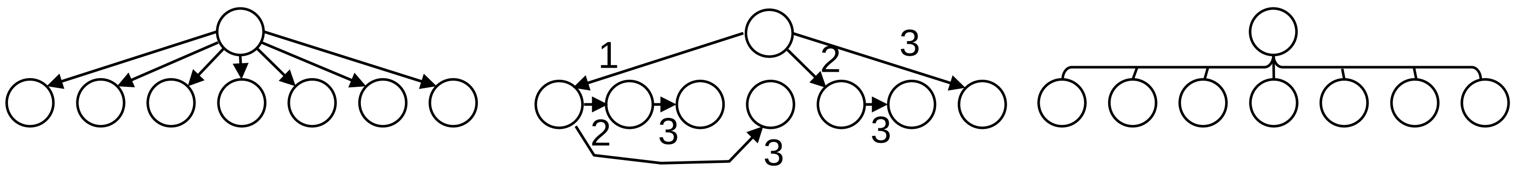
Tree based algorithm
O($\log_2$(# processes))



Hardware-broadcast
O(1)

Internally: tree-based algorithms

E.g., broadcast



Sequential algorithm
O(# processes)

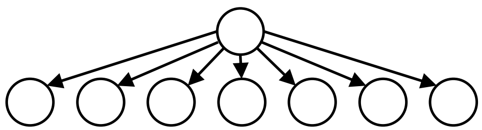
Tree based algorithm
O($\log_2$(# processes))

Hardware-broadcast
O(1)

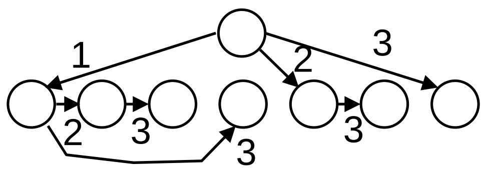
And **optimized** algorithms on **clusters of SMP nodes** are even **more complicated!**

Internally: tree-based algorithms

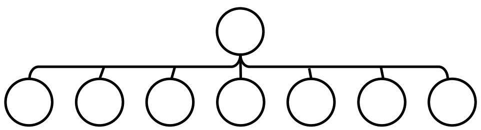
E.g., broadcast



Sequential algorithm
O(# processes)



Tree based algorithm
O($\log_2$(# processes))



Hardware-broadcast
O(1)

And **optimized** algorithms on **clusters of SMP nodes** are even **more complicated!**

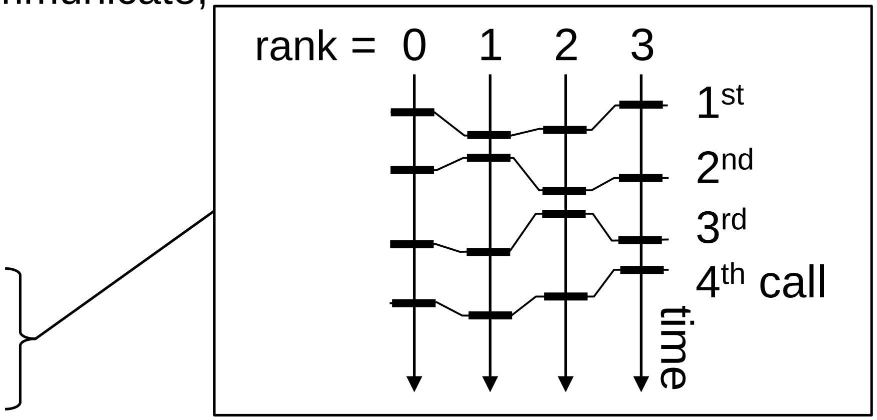
You need not to care about !
It is the job of the MPI lib !!!

Characteristics of Collective Communication

- Collective action over a communicator.
- All process of the communicator must communicate, i.e., must call the collective routine.
- Synchronization may or may not occur, therefore all processes must be able to start the collective routine.

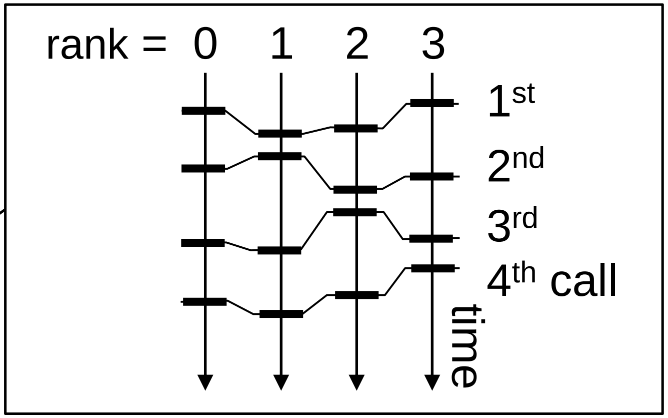
Characteristics of Collective Communication

- Collective action over a communicator.
- All process of the communicator must communicate, i.e., must call the collective routine.
- Synchronization may or may not occur, therefore all processes must be able to start the collective routine.
- On a given communicator, the n-th collective call must match on all processes of the communicator.



Characteristics of Collective Communication

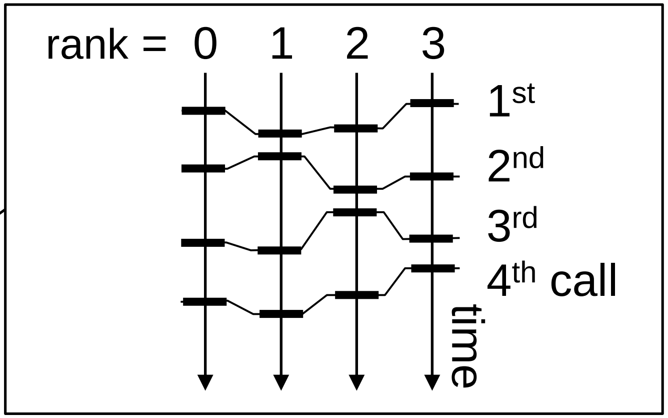
- Collective action over a communicator.
- All process of the communicator must communicate, i.e., must call the collective routine.
- Synchronization may or may not occur, therefore all processes must be able to start the collective routine.
- On a given communicator, the n-th collective call must match on all processes of the communicator.



- In MPI-1.0 – MPI-2.2, all collective operations are blocking. Nonblocking versions since MPI-3.0.
- No tags.

Characteristics of Collective Communication

- Collective action over a communicator.
- All process of the communicator must communicate, i.e., must call the collective routine.
- Synchronization may or may not occur, therefore all processes must be able to start the collective routine.
- On a given communicator, the n-th collective call must match on all processes of the communicator.



- In MPI-1.0 – MPI-2.2, all collective operations are blocking. Nonblocking versions since MPI-3.0.
- No tags.

very important

- For each message, the amount of data sent must exactly match the amount of data specified by the receiver

  → It is forbidden to provide receive buffer count arguments that are too long (and also too short, of course)

Exception with Python (mpi4py): if a buffer argument represents #processes of messages (e.g. snd_buf in comm.Scatter) and the argument count is to be derived from the buffer argument (i.e. is not explicitly defined in the argument list), then this count argument is derived from the inferred number of elements of the buffer divided by the size of the communicator.

e.g., when passing snd_buf, or (snd_buf, datatype).

Barrier Synchronization

C

Fortran

Python

- C/C++: int MPI_Barrier(MPI_Comm comm)
- Fortran: MPI_BARRIER(comm, *ierror*)
  mpi_f08: TYPE(MPI_Comm) :: comm ; INTEGER, OPTIONAL :: ierror
  mpi & mpif.h: INTEGER comm, ierror
- Python: comm.**B**arrier() or comm.**b**arrier()

- MPI_Barrier is normally never needed:
  - all synchronization is done automatically by the data communication:
    - **a process cannot continue before it has the data that it needs.**

Barrier Synchronization

C | Fortran | Python

- C/C++: int MPI_Barrier(MPI_Comm comm)
- Fortran: MPI_BARRIER(comm, *ierror*)

  mpi_f08: TYPE(MPI_Comm) :: comm ; INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: INTEGER comm, ierror
- Python: comm.**B**arrier() or comm.**b**arrier()

- MPI_Barrier is normally never needed:
  - all synchronization is done automatically by the data communication:
    - **a process cannot continue before it has the data that it needs.**
  - if used for debugging:
    - **please guarantee, that it is removed in production.**

Barrier Synchronization

C

Fortran

Python

- C/C++: int MPI_Barrier(MPI_Comm comm)
- Fortran: MPI_BARRIER(comm, *ierror*)
  mpi_f08: TYPE(MPI_Comm) :: comm ; INTEGER, OPTIONAL :: ierror
  mpi & mpif.h: INTEGER comm, ierror
- Python: comm.Barrier() or comm.barrier()

- MPI_Barrier is normally never needed:
  - all synchronization is done automatically by the data communication:
    - **a process cannot continue before it has the data that it needs.**
  - if used for debugging:
    - **please guarantee, that it is removed in production.**
  - for profiling: to separate time measurement of
    - **Load imbalance of computation [MPI_Wtime(); MPI_Barrier(); MPI_Wtime()]**
    - **communication epochs [MPI_Wtime(); MPI_Allreduce(); …; MPI_Wtime()]**

Barrier Synchronization

C | Fortran | Python

- C/C++: int MPI_Barrier(MPI_Comm comm)
- Fortran: MPI_BARRIER(comm, *ierror*)
  mpi_f08: TYPE(MPI_Comm) :: comm ; INTEGER, OPTIONAL :: ierror
  mpi & mpif.h: INTEGER comm, ierror
- Python: comm.**B**arrier() or comm.**b**arrier()

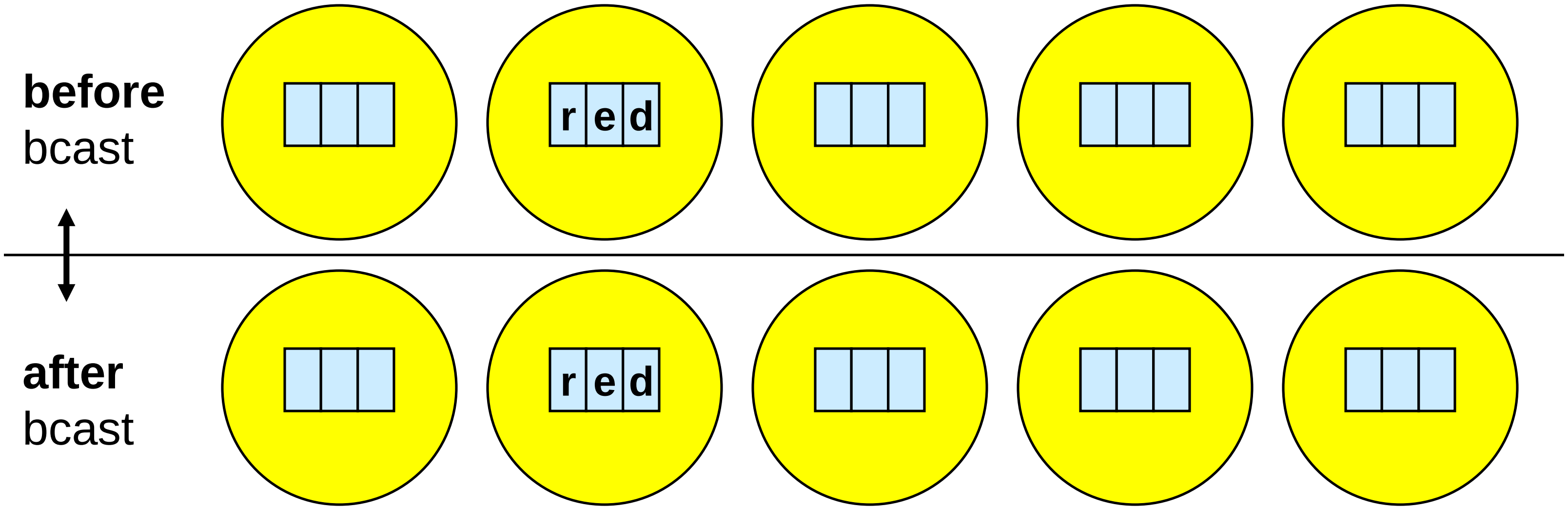
- MPI_Barrier is normally never needed:
  – all synchronization is done automatically by the data communication:
    - **a process cannot continue before it has the data that it needs.**
  – if used for debugging:
    - **please guarantee, that it is removed in production.**
  – for profiling: to separate time measurement of
    - **Load imbalance of computation [MPI_Wtime(); MPI_Barrier(); MPI_Wtime()]**
    - **communication epochs [MPI_Wtime(); MPI_Allreduce(); …; MPI_Wtime()]**
  – ~~if used for synchronizing external *communication* (e.g. I/O):~~
    - ~~**exchanging tokens may be more efficient and scalable than a barrier on MPI_COMM_WORLD,**~~
    - ~~**see also advanced exercise of this course chapter.**~~

Broadcast

- C/C++: int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- Fortran: MPI_BCAST(buf, count, datatype, root, comm, *ierror*)

  mpi_f08:
  TYPE(*), DIMENSION(..) :: buf
  TYPE(MPI_Datatype) :: datatype; TYPE(MPI_Comm) :: comm
  INTEGER :: count, root; INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: <type> buf(*); INTEGER count, datatype, root, comm, ierror
- Python: comm.**B**cast(buf, int root=0) or comm.**b**cast(obj, int root=0)



e.g., root=1

- **rank of the sending process (i.e., root process)**
- **must be given identically by all processes**

Example:
MPI_Bcast(buf, 3, MPI_CHAR, 1, MPI_COMM_WORLD);

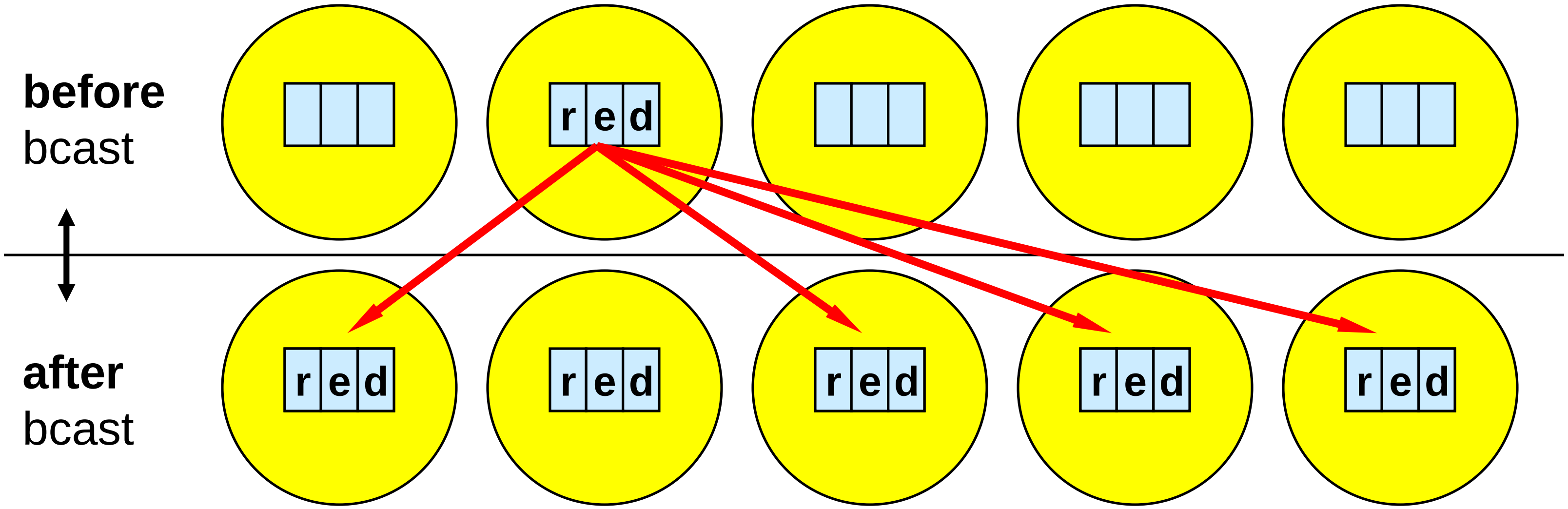
Broadcast

C | Fortran | Python

- C/C++: int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- Fortran: MPI_BCAST(buf, count, datatype, root, comm, *ierror*)

  mpi_f08: TYPE(*), DIMENSION(..) :: buf
  TYPE(MPI_Datatype) :: datatype; TYPE(MPI_Comm) :: comm
  INTEGER :: count, root; INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: <type> buf(*); INTEGER count, datatype, root, comm, ierror
- Python: comm.**B**cast(buf, int root=0) or comm.**b**cast(obj, int root=0)



e.g., root=1

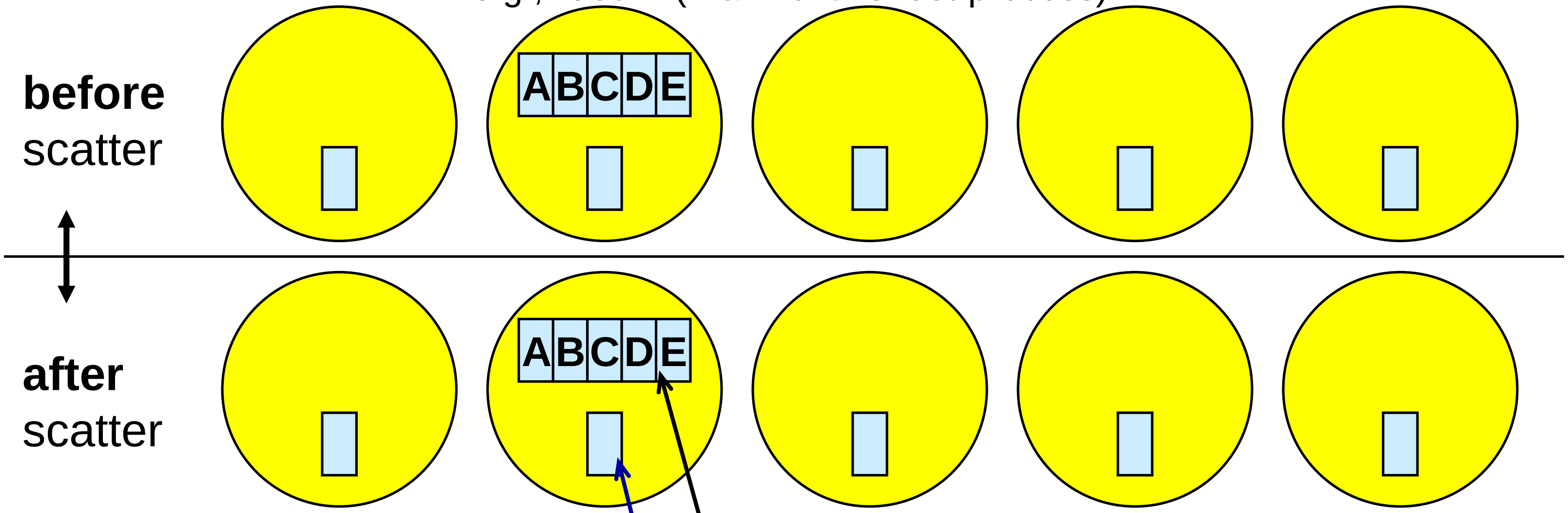
- **rank of the sending process (i.e., root process)**
- **must be given identically by all processes**

Example:

MPI_Bcast(buf, 3, MPI_CHAR, 1, MPI_COMM_WORLD);

Scatter

e.g., **root=1** (=rank of this *root* process)



C

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount,  MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

Fortran

```
MPI_SCATTER(sendbuf, sendcount, sendtype,
            recvbuf, recvcount,  recvtype,  root, comm, ierror)
```

mpi_f08: TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: sendcount, recvcount, root;
TYPE(MPI_Datatype) :: sendtype, recvtype; TYPE(MPI_Comm) :: comm; INTEGER, OPTIONAL :: ierror

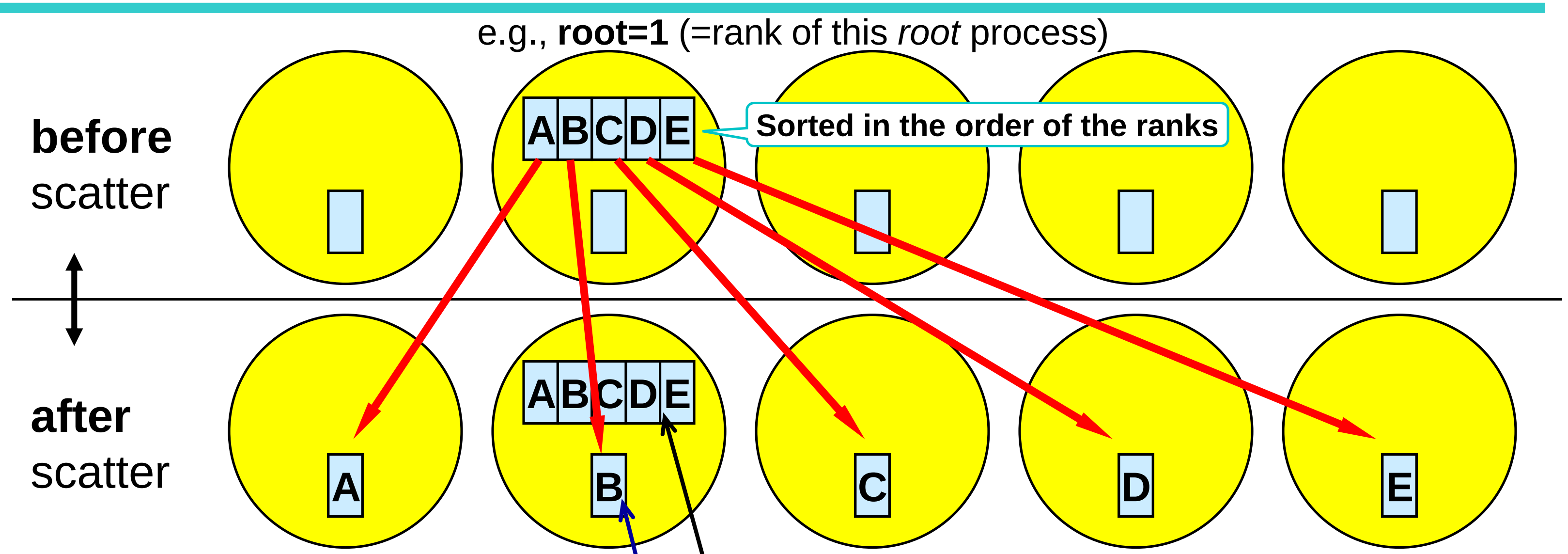
mpi & mpif.h: <type> sendbuf(*), recvbuf(*); INTEGER sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Python

comm.Scatter(sendbuf *or* None, recvbuf, int root=0)
recvobj = comm.scatter(sendobj *or* None, int root=0)

See, e.g., [Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)]

Scatter



C int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void ****recvbuf***, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)

Fortran MPI_SCATTER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm, ***ierror***)

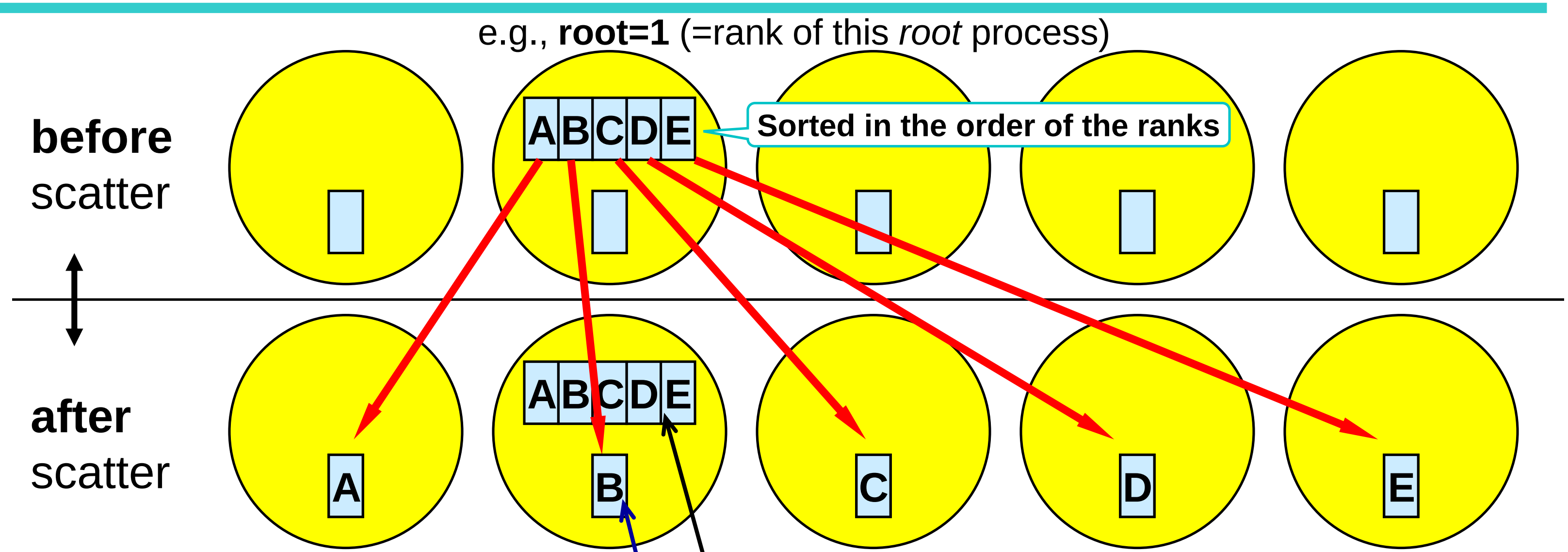
mpi_f08: TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: sendcount, recvcount, root;
TYPE(MPI_Datatype) :: sendtype, recvtype; TYPE(MPI_Comm) :: comm; INTEGER, OPTIONAL :: ierror

mpi & mpif.h: <type> sendbuf(*), recvbuf(*); INTEGER sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Python comm.**S**catter(sendbuf *or* None, recvbuf, int root=0)
recvobj = comm.**s**catter(sendobj *or* None, int root=0)

See, e.g., Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)

Scatter



C

int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void ****recvbuf***, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)

Fortran

MPI_SCATTER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm, ***ierror***)

mpi_f08: TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: sendcount, recvcount, root;
TYPE(MPI_Datatype) :: sendtype, recvtype; TYPE(MPI_Comm) :: comm; INTEGER, OPTIONAL :: ierror

mpi & mpif.h: <type> sendbuf(*), recvbuf(*); INTEGER sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Python

comm.Scatter(sendbuf *or* None, recvbuf, int root=0)
recvobj = comm.scatter(sendobj *or* None, int root=0)

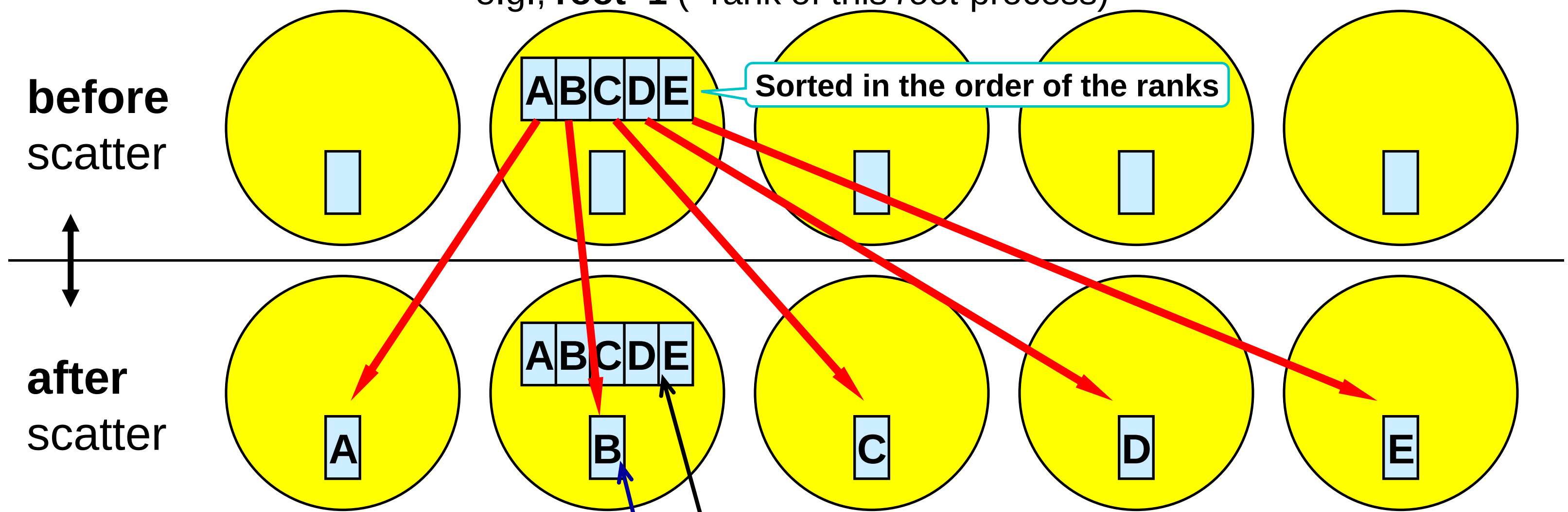
See, e.g., [Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)]

Example: MPI_Scatter(sbuf, 1, MPI_CHAR, *rbuf*, 1, MPI_CHAR, 1, MPI_COMM_WORLD);

Completely ignored at all processes except *root*

Scatter

e.g., **root=1** (=rank of this *root* process)



C

int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void ****recvbuf***, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)

Fortran

MPI_SCATTER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm, ***ierror***)

mpi_f08: TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: sendcount, recvcount, root;
TYPE(MPI_Datatype) :: sendtype, recvtype; TYPE(MPI_Comm) :: comm; INTEGER, OPTIONAL :: ierror

mpi & mpif.h: <type> sendbuf(*), recvbuf(*); INTEGER sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Python

comm.Scatter(sendbuf *or* None, recvbuf, int root=0)
recvobj = comm.scatter(sendobj *or* None, int root=0)

See, e.g., Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)

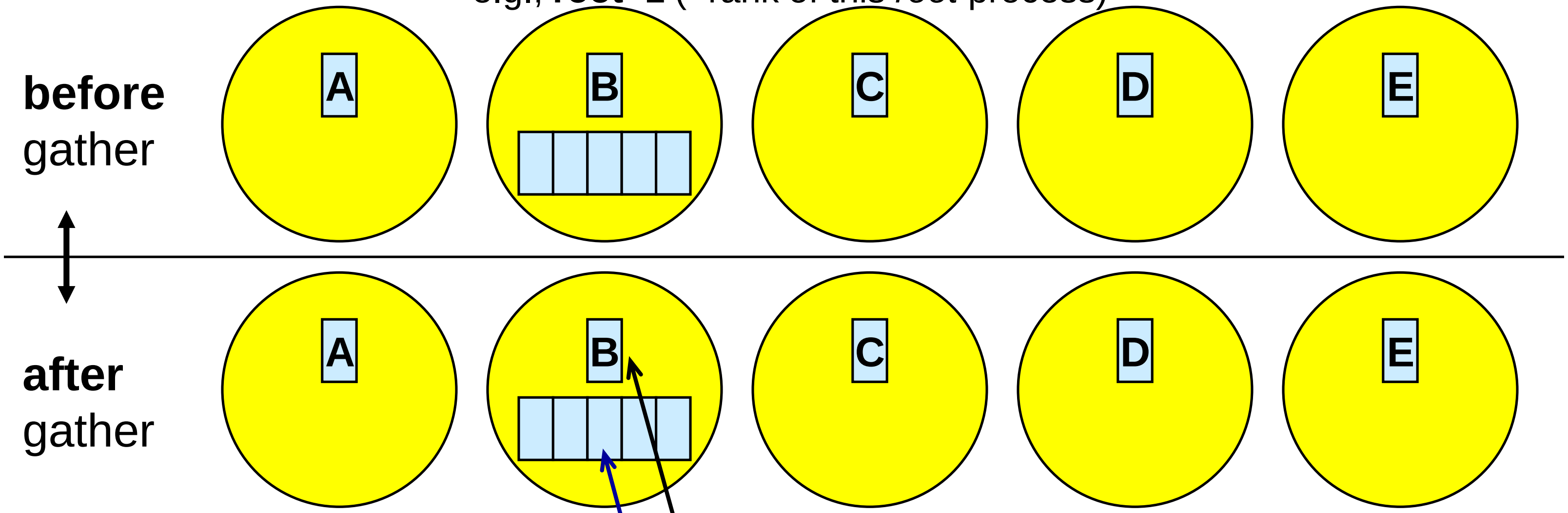
sendcount describes only one message

Example: MPI_Scatter(sbuf, 1, MPI_CHAR, *rbuf*, 1, MPI_CHAR, 1, MPI_COMM_WORLD);

Completely ignored at all processes except *root*

Gather

e.g., **root=1** (=rank of this *root* process)



C

int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void ****recvbuf***, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)

Fortran

MPI_GATHER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm, ***ierror***)

mpi_f08: TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: sendcount, recvcount, root;
TYPE(MPI_Datatype) :: sendtype, recvtype; TYPE(MPI_Comm) :: comm; INTEGER, OPTIONAL :: ierror

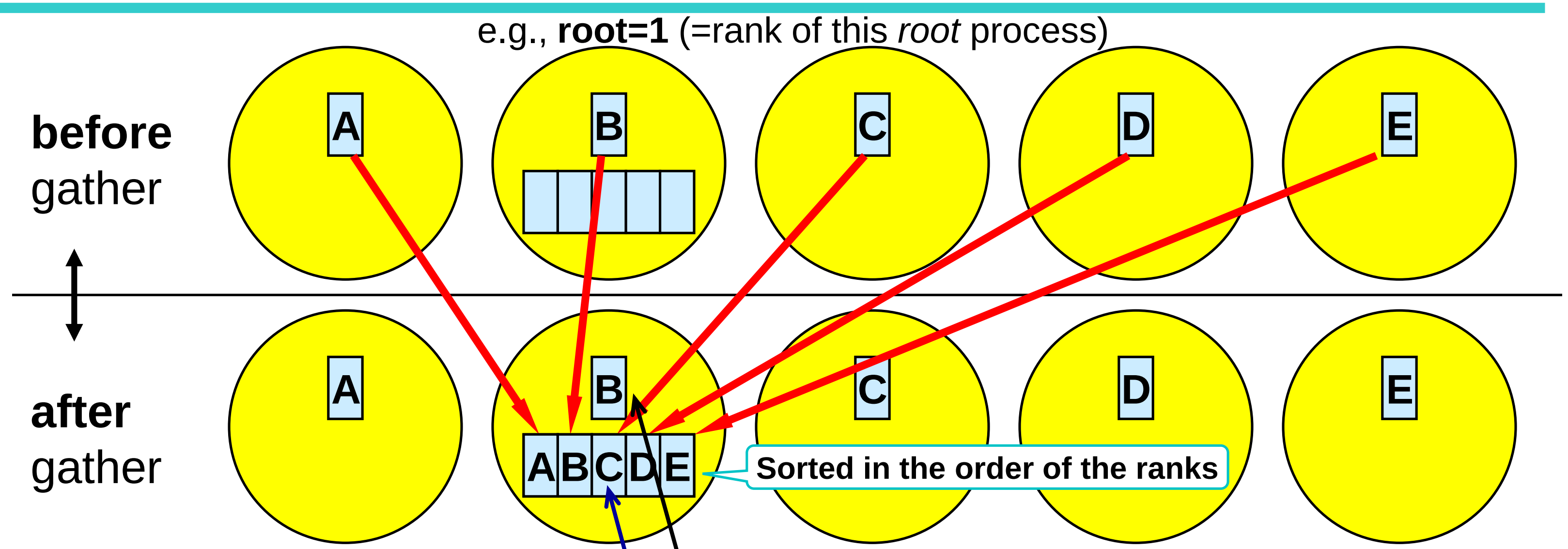
mpi & mpif.h: <type> sendbuf(*), recvbuf(*); INTEGER sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Python

comm.**G**ather(sendbuf, recvbuf *or* None, int root=0)
recvobj = comm.**g**ather(sendobj, int root=0)

See, e.g., Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)

Gather



C
```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount,  MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

Fortran
```
MPI_GATHER(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype, root, comm, ierror)
```

mpi_f08: TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: sendcount, recvcount, root;
TYPE(MPI_Datatype) :: sendtype, recvtype; TYPE(MPI_Comm) :: comm; INTEGER, OPTIONAL :: ierror

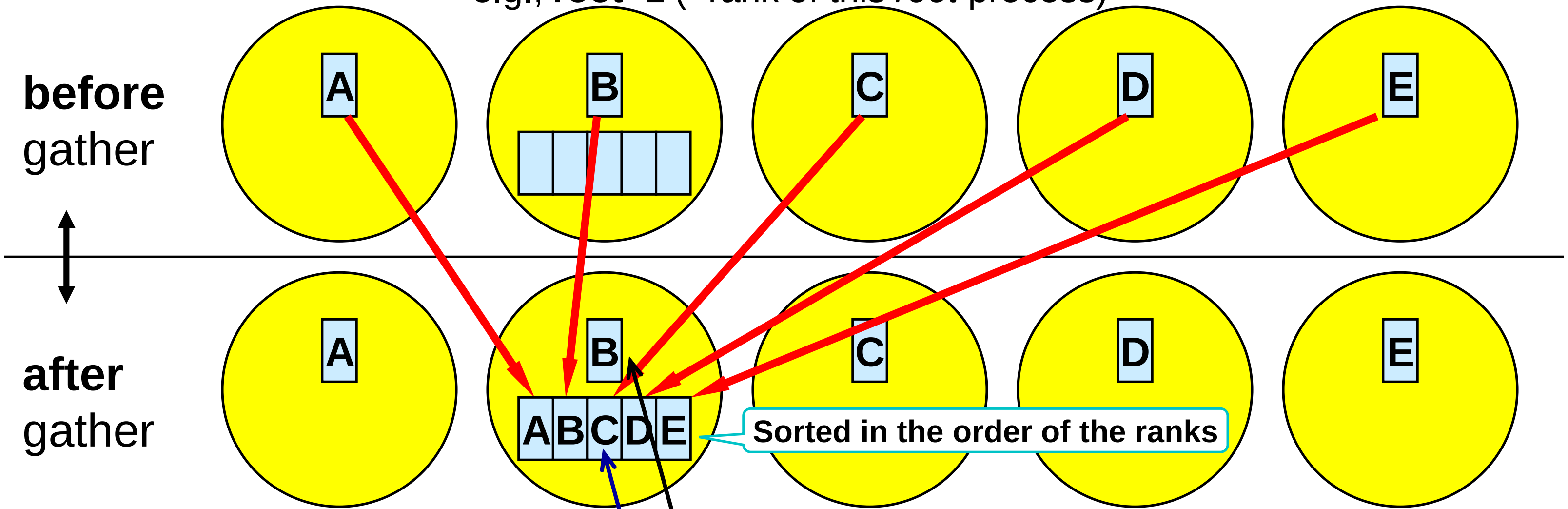
mpi & mpif.h: <type> sendbuf(*), recvbuf(*); INTEGER sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Python
comm.Gather(sendbuf, recvbuf *or* None, int root=0)
recvobj = comm.gather(sendobj, int root=0)

See, e.g., [Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)]

Gather

e.g., **root=1** (=rank of this *root* process)



C

int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void ****recvbuf***, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)

Fortran

MPI_GATHER(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm, ***ierror***)

mpi_f08: TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: sendcount, recvcount, root;
TYPE(MPI_Datatype) :: sendtype, recvtype; TYPE(MPI_Comm) :: comm; INTEGER, OPTIONAL :: ierror

mpi & mpif.h: <type> sendbuf(*), recvbuf(*); INTEGER sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Python

comm.**G**ather(sendbuf, recvbuf *or* None, int root=0)
recvobj = comm.**g**ather(sendobj, int root=0)

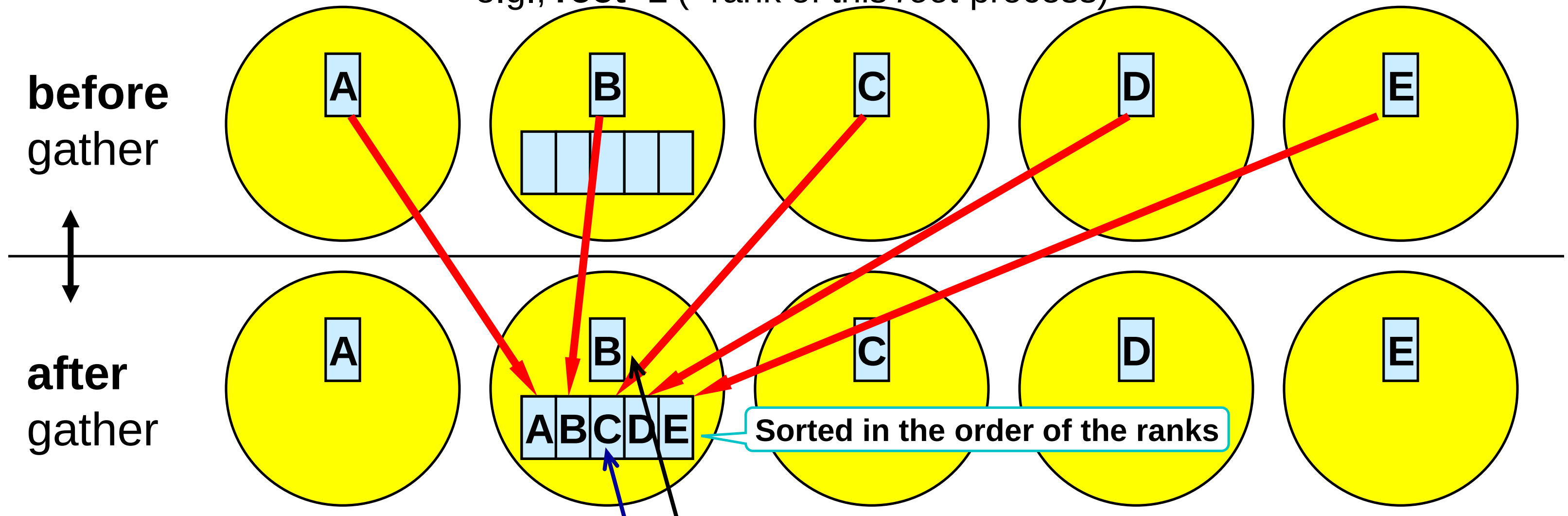
See, e.g., [Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)]

CALL MPI_Gather(sbuf, 1, MPI_CHARACTER, *rbuf*, 1, MPI_CHARACTER, 1, MPI_COMM_WORLD, ierror);

Completely ignored at all processes except *root*

Gather

e.g., **root=1** (=rank of this *root* process)



C

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount,  MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

Fortran

```
MPI_GATHER(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype, root, comm, ierror)
```

mpi_f08: TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: sendcount, recvcount, root;
TYPE(MPI_Datatype) :: sendtype, recvtype; TYPE(MPI_Comm) :: comm; INTEGER, OPTIONAL :: ierror

mpi & mpif.h: <type> sendbuf(*), recvbuf(*); INTEGER sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Python

comm.Gather(sendbuf, recvbuf *or* None, int root=0)
recvobj = comm.gather(sendobj, int root=0)

See, e.g., Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)

CALL MPI_Gather(sbuf, 1, MPI_CHARACTER, *rbuf*, 1, MPI_CHARACTER, 1, MPI_COMM_WORLD, ierror);

recvcount describes only one message

Completely ignored at all processes except *root*

Exercise 1 — Gather

In MPI/tasks/…

- Use C C/Ch6/gather-skel.c or Fortran F_30/Ch6/gather-skel_30.f90 or Python PY/Ch6/gather-skel.py (hint: use Gather, i.e. with numPy buffers)
- The skeleton is based on our first example in course Chapter 1.
- Differences:
  - This skeleton first gathers the data into an array at process 0
  - And then, process 0 prints the array.
- In this exercise, you should substitute the point-to-point communication by one call to MPI_Gather
- Hint for Python
  - The `result_array` (used in MPI_Gather) needs to be declared on all processes. Therefore add "**`else: result_array = None`**"

```
if (my_rank == 0):
   result_array = np.empty(num_procs, dtype=np.double)
else:
  result_array = None
```

Advanced Exercise 1b — Barrier / profiling

Advanced Exercise 1b

- Based on C/Ch6/solutions/pi.c → pi-mpi.c → pi-mpi-inbalance.c
  (pi-mpi.c: balanced; pi-mpi-inbalance.c: inbalanced)
- Use **C** C/Ch6/pi-mpi-inbalance-profiling-skel.c
  or **Python** PY/Ch6/pi-mpi-inbalance-profiling-skel.py
  or **Fortran** (my apologies, Fortran does not yet exists, but this shouldn't be a problem)
- This program has several parts:
  - Perfect work-distribution for n=10,000,000 intervals.
  - If 3 or more processes:
    Introducing an inbalance: The last 2 processes get double and zero intervals.
  - Calculation of π with a distributed integral → partial sums in p_sum.
  - Global reduction of all p_sum into one global sum.
  - Time measurements for all parts
- Your task, see "`// EXERCISE`" in the skeleton:
  - Add MPI_Barrier wherever useful, and especially to measure idle time due to the bad load balance.
  - Substitute all `wt?` by `wt1` .. `wt4` as needed
  - Compile and run it with 2 processes
    → expected result 99,9% parallel efficiency
  - Run with more than 3 processes
    → about 50% parallel efficiency and 50% in idle time

Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- d_0 **o** d_1 **o** d_2 **o** d_3 **o** … **o** d_{s-2} **o** d_{s-1}
  - d_i = data in process rank i
    - **single variable, or**
    - **vector**
  - **o** = associative operation
  - Example:
    - **global sum or product**
    - **global maximum or minimum**
    - **global user-defined operation**

Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- d_0 **o** d_1 **o** d_2 **o** d_3 **o** … **o** d_{s-2} **o** d_{s-1}
  - d_i = data in process rank i
    - **single variable, or**
    - **vector**
  - **o** = associative operation
  - Example:
    - **global sum or product**
    - **global maximum or minimum**
    - **global user-defined operation**
- floating point rounding may depend on usage of associative law:
  - [(d_0 **o** d_1) **o** (d_2 **o** d_3)] **o** [… **o** (d_{s-2} **o** d_{s-1})]
  - (((((d_0 **o** d_1) **o** d_2) **o** d_3) **o** …) **o** d_{s-2}) **o** d_{s-1})
  - May be even worse through partial sums in each process:

$$\sum_{i=0}^{n-1} x_i \rightarrow [[[(\sum_{i=0}^{n/s-1} x_i \;\mathbf{o}\; \sum_{i=n/s}^{2n/s-1} x_i) \;\mathbf{o}\; (\ldots \;\mathbf{o}\; \ldots)] \;\mathbf{o}\; [\ldots \;\mathbf{o}\; (\ldots \;\mathbf{o}\; \ldots)]]]$$

E.g., with n=10^8 rounding errors may modify last 3 or 4 digits!

Example of Global Reduction

- Global integer sum.
- Sum of all inbuf values should be returned in *resultbuf.*
- C/C++: root=0;

  MPI_Reduce(&inbuf, &*resultbuf*, 1, MPI_INT,
  MPI_SUM, root, MPI_COMM_WORLD);
- Fortran: root=0

  CALL MPI_REDUCE(inbuf, *resultbuf*, 1, MPI_INTEGER,
  MPI_SUM, root, MPI_COMM_WORLD, *IERROR*)
- Python: comm_world = MPI.COMM_WORLD

  snd_buf = np.array(value, dtype=np.intc)
  resultbuf = np.empty((), dtype=np.intc)
  comm_world.Reduce(snd_buf, *resultbuf*, op=MPI.SUM)

  `op=MPI.SUM` and `root=0` are defaults
- The result is only placed in *resultbuf* at the root process.

C

Fortran

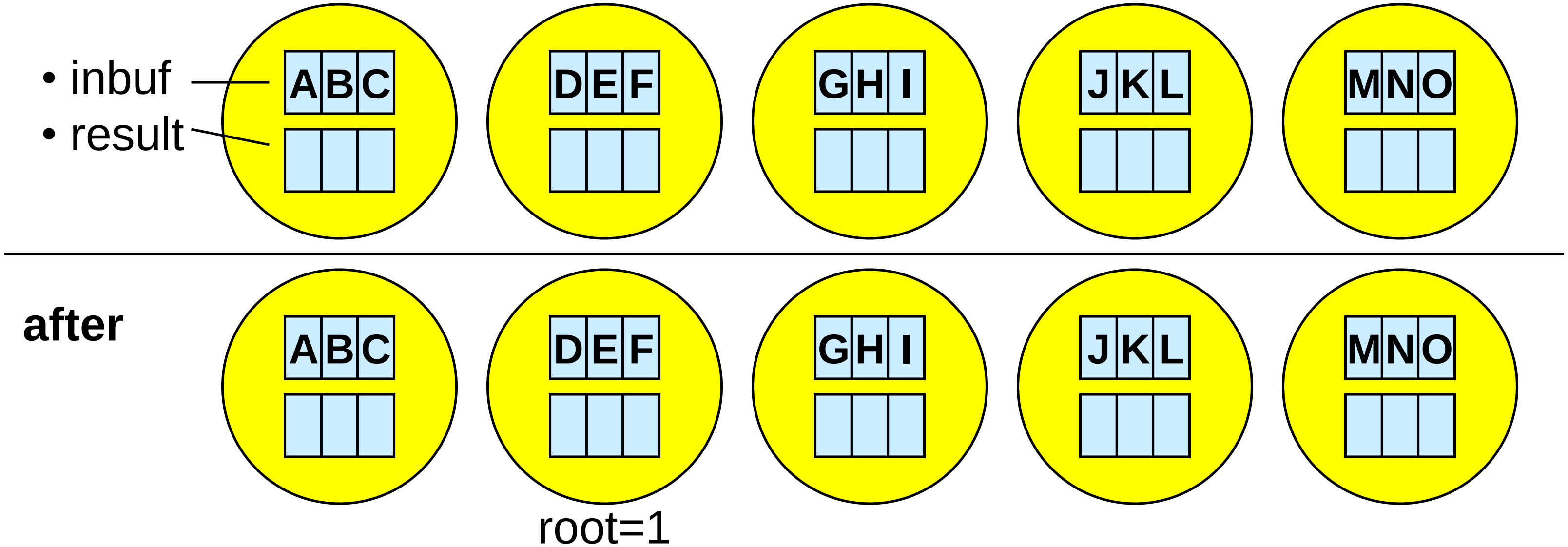
Python

Predefined Reduction Operation Handles

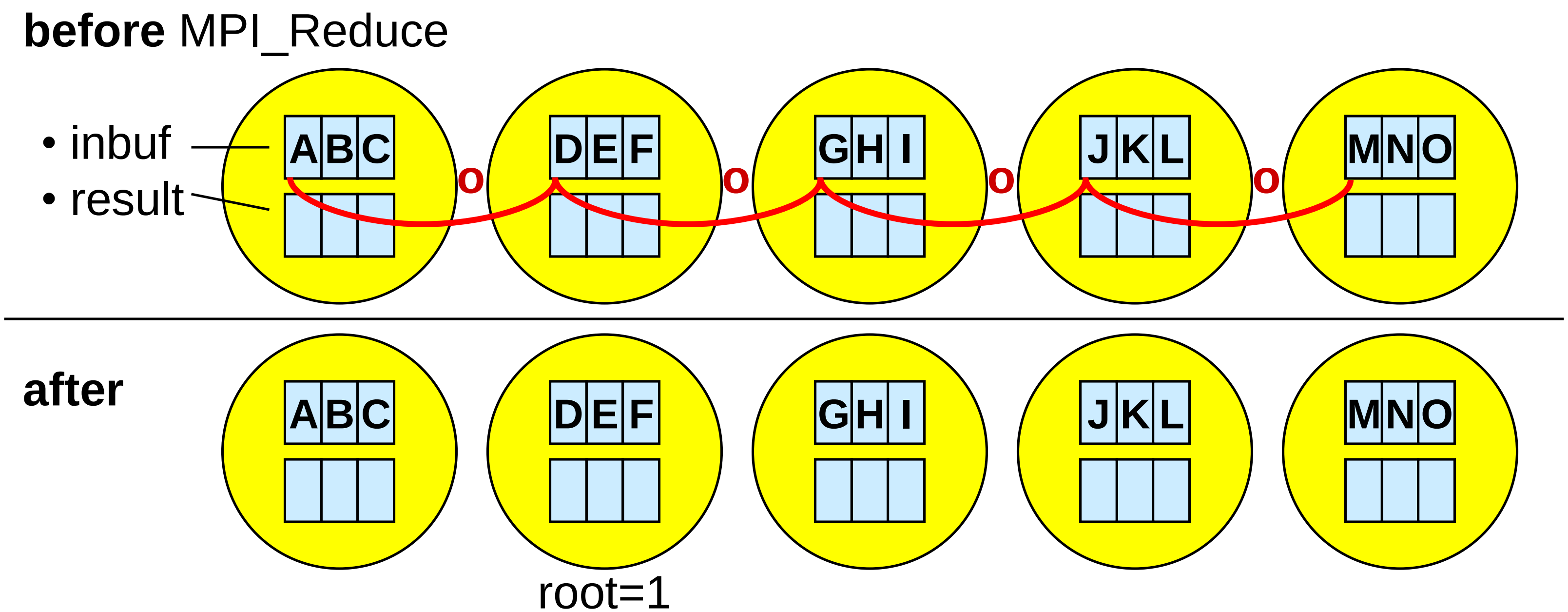
Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

MPI_Reduce

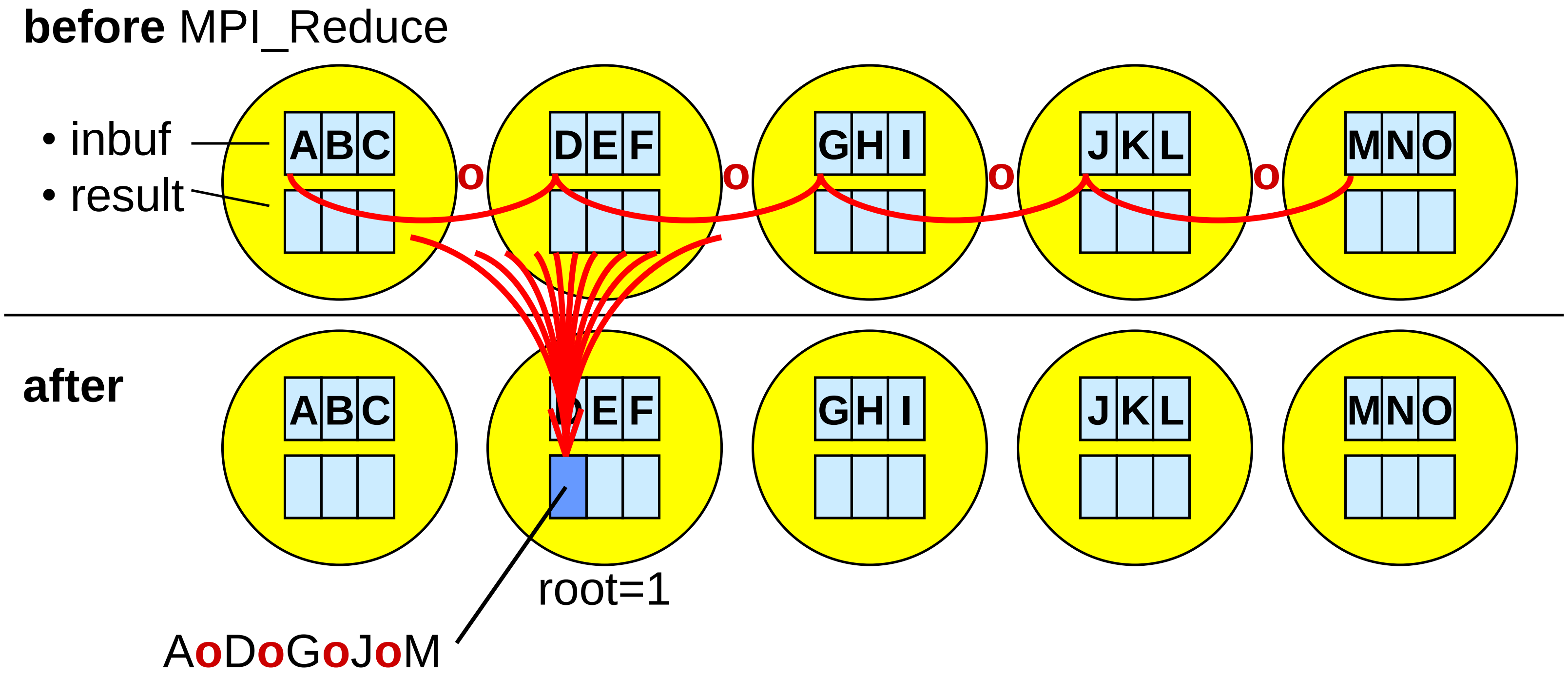
before MPI_Reduce



MPI_Reduce

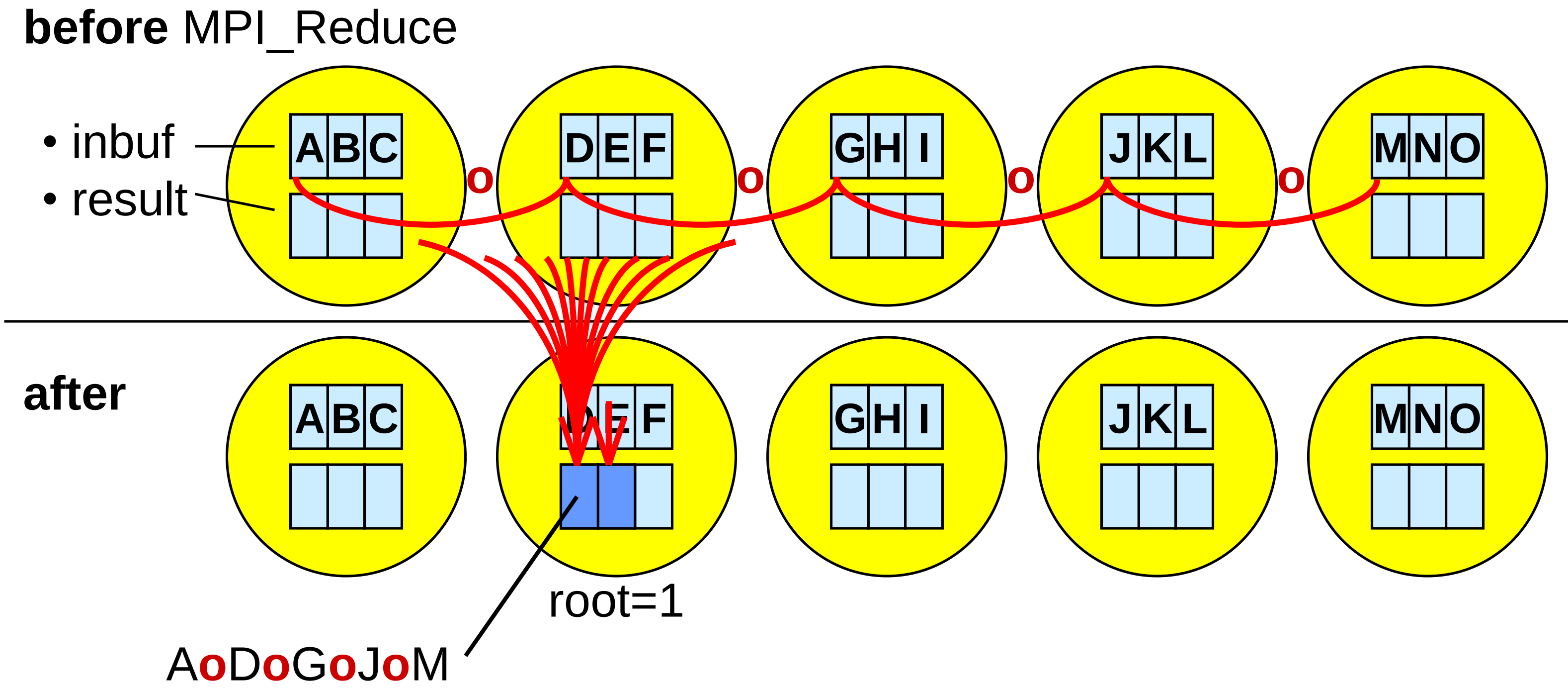

before MPI_Reduce
• inbuf
• result
A B C
D E F
G H I
J K L
M N O
o
o
o
o
after
A B C
D E F
G H I
J K L
M N O
root=1

MPI_Reduce



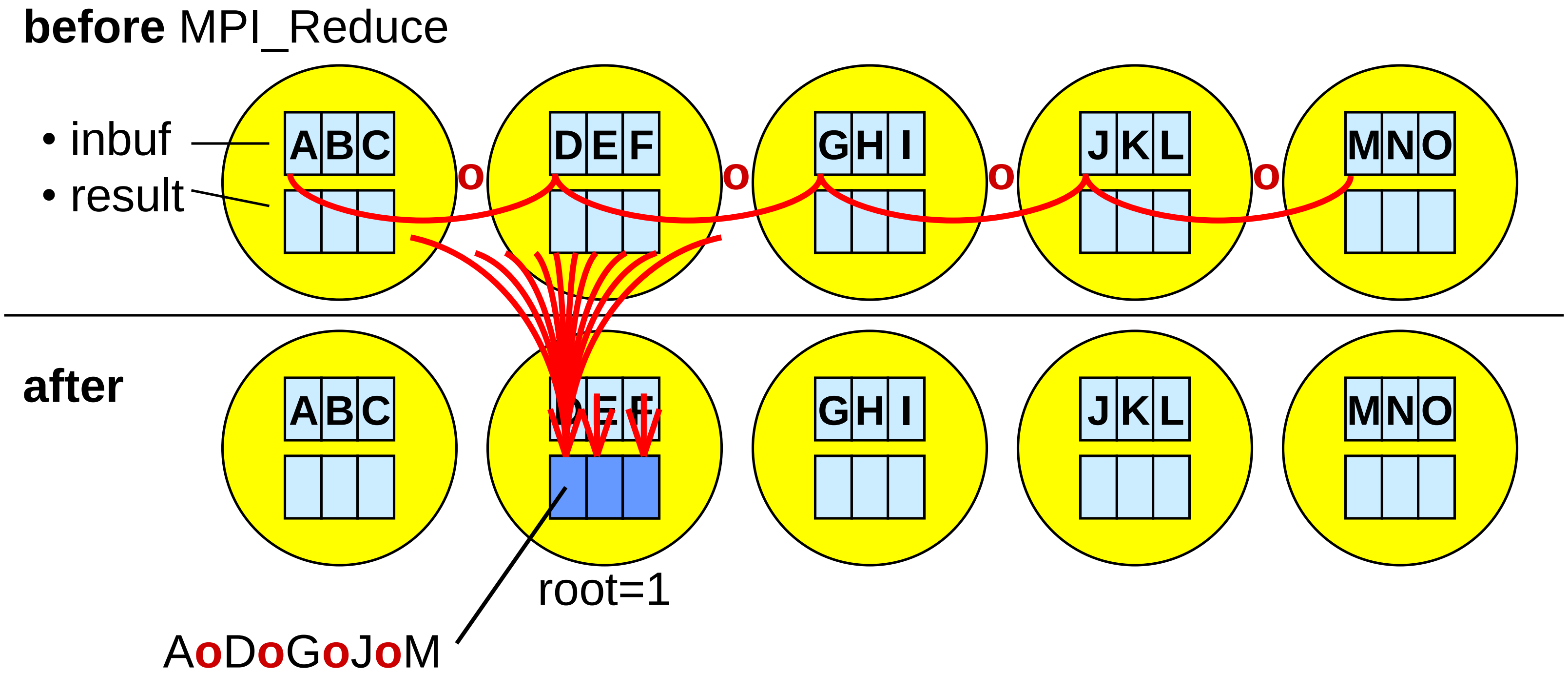
before MPI_Reduce
• inbuf
• result
A B C
D E F
G H I
J K L
M N O
o
o
o
o
after
A B C
D E F
G H I
J K L
M N O
root=1
AoDoGoJoM

MPI_Reduce



before MPI_Reduce
• inbuf
• result
A B C
D E F
G H I
J K L
M N O
o
o
o
o
after
A B C
D E F
G H I
J K L
M N O
root=1
AoDoGoJoM

MPI_Reduce



before MPI_Reduce
• inbuf
• result
A B C
D E F
G H I
J K L
M N O
o
o
o
o
after
A B C
D E F
G H I
J K L
M N O
root=1
AoDoGoJoM

User-Defined Reduction Operations

- Operator handles
  - predefined – see table above
  - user-defined
- User-defined operation □:
  - associative
  - user-defined function must perform the operation vector_A □ vector_B
  - syntax of the user-defined function → MPI standard
- Registering a user-defined reduction function:

C
  - C/C++: MPI_Op_create(MPI_User_function *func, int commute, MPI_Op **op*)

Fortran
  - Fortran: MPI_OP_CREATE(FUNC, COMMUTE, *OP*, *IERROR*)

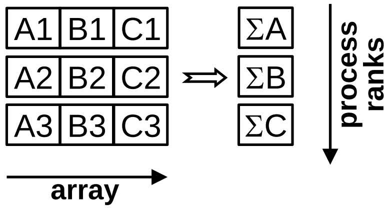
Python
  - Python: *op* = MPI.Op.Create(func, commute=True *or* False)
- COMMUTE tells the MPI library whether FUNC is commutative.

Variants of Reduction Operations

- MPI_Allreduce
  - no root,
  - returns the result in all processes

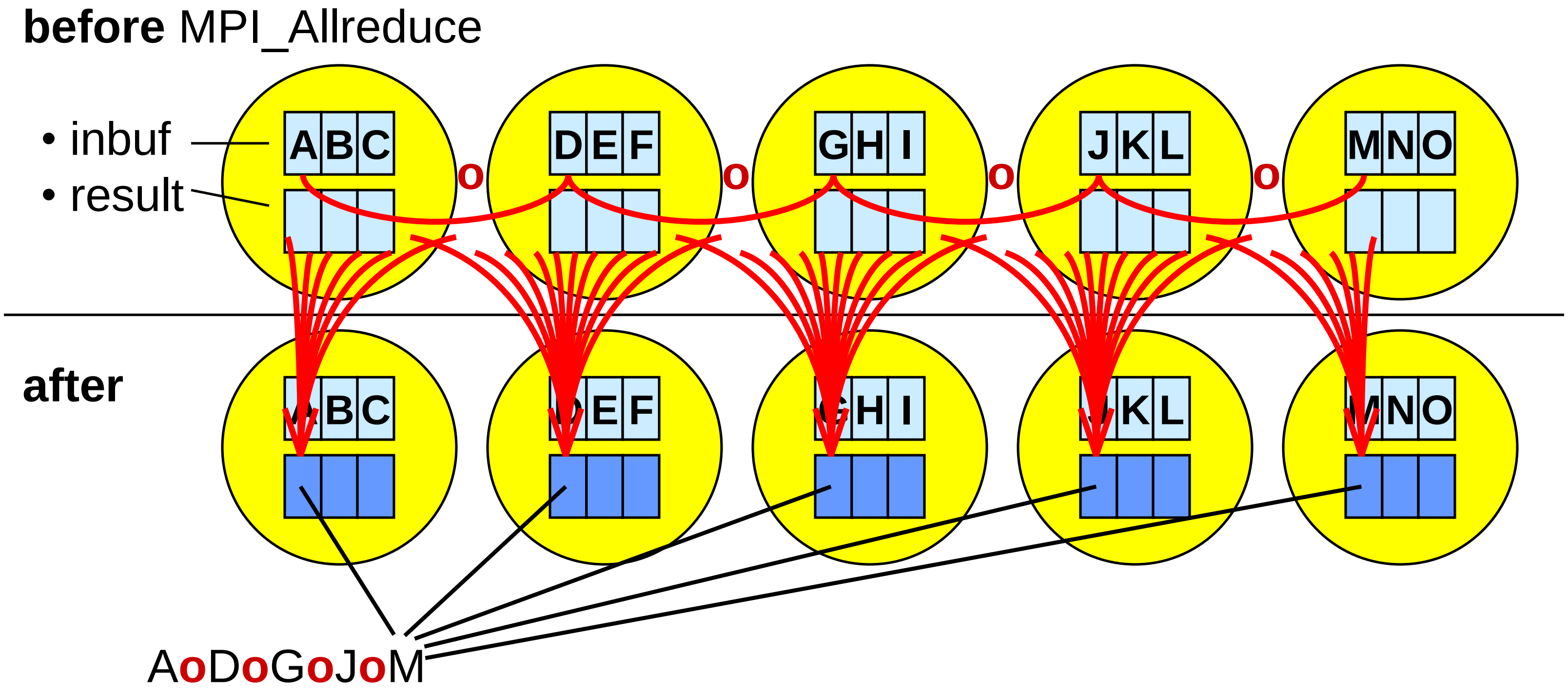
New in MPI-2.2

- MPI_Reduce_scatter_block and MPI_Reduce_scatter
  - result vector of the reduction operation is scattered to the processes into the real result buffers



- MPI_Scan
  - prefix reduction
  - result at process with rank i :=
    reduction of inbuf-values from rank 0 to rank i
- MPI_Exscan
  - result at process with rank i :=
    reduction of inbuf-values from rank 0 to rank **i-1**

MPI_Allreduce



skipped

Interface of MPI_Allreduce

Language independent specification (LIS)

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

Additional **MPI_Count** version since MPI-4.0: MPI_Allreduce_c

C — C/C++ binding

```
int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Fortran — **mpi_f08** Module Fortran binding

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

Overloaded with **INTEGER(KIND=MPI_COUNT_KIND)** version since MPI-4.0

mpi module + mpif.h Fortran binding

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

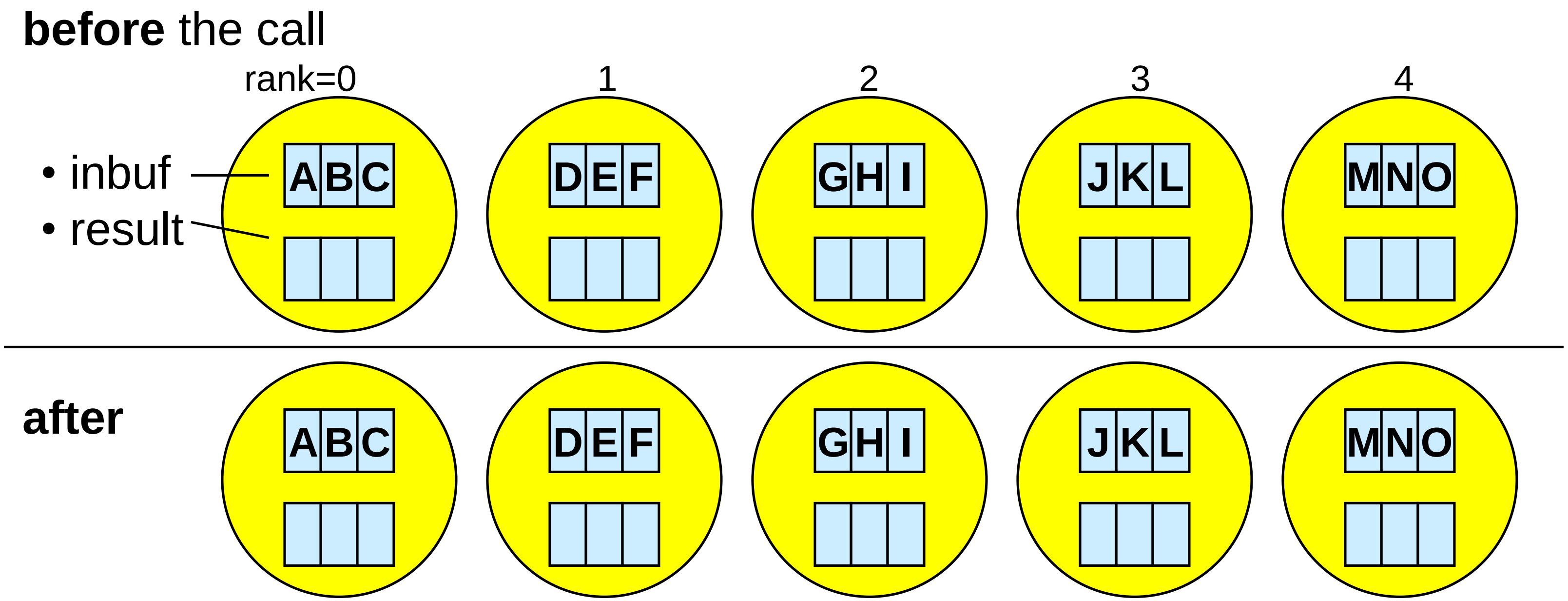
Python

Python: *win* = comm.Allreduce(sendbuf, recvbuf, op)

op=MPI.SUM is the default

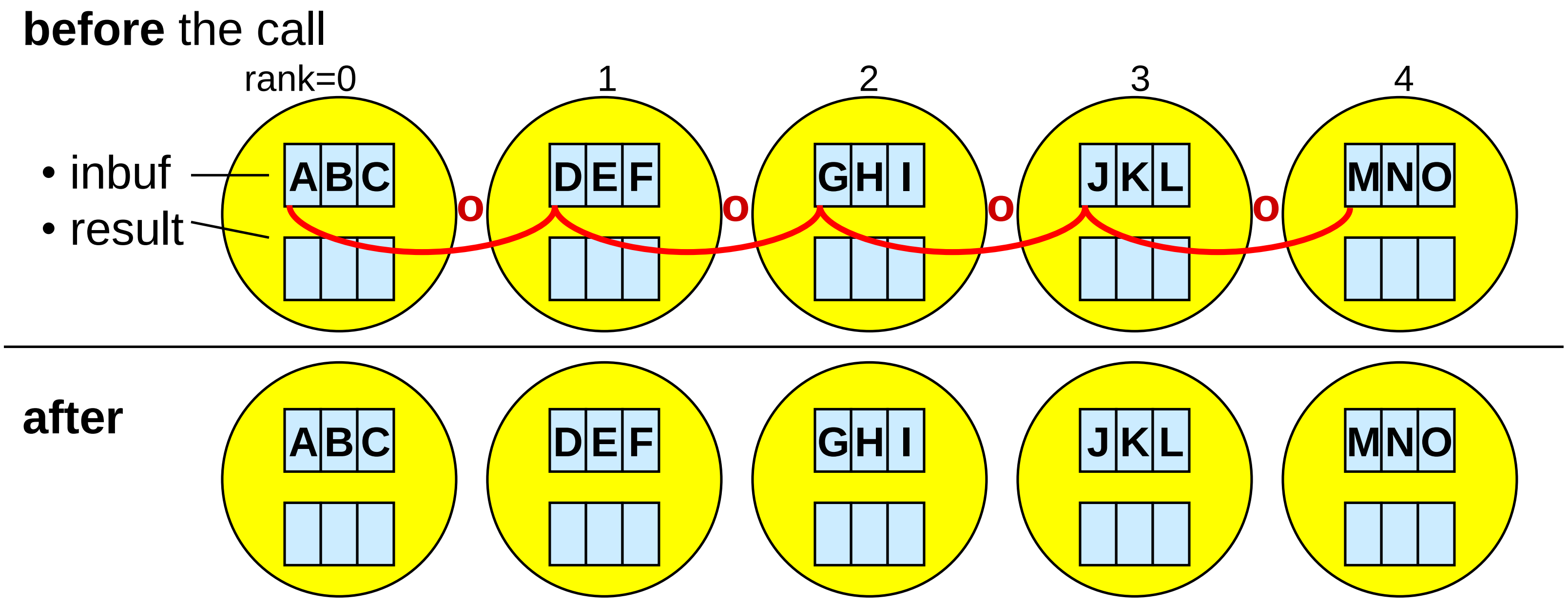
numpy arrays, e.g., **sendbuf,(recvbuf,1,MPI.INT)**

MPI_Scan and MPI_Exscan



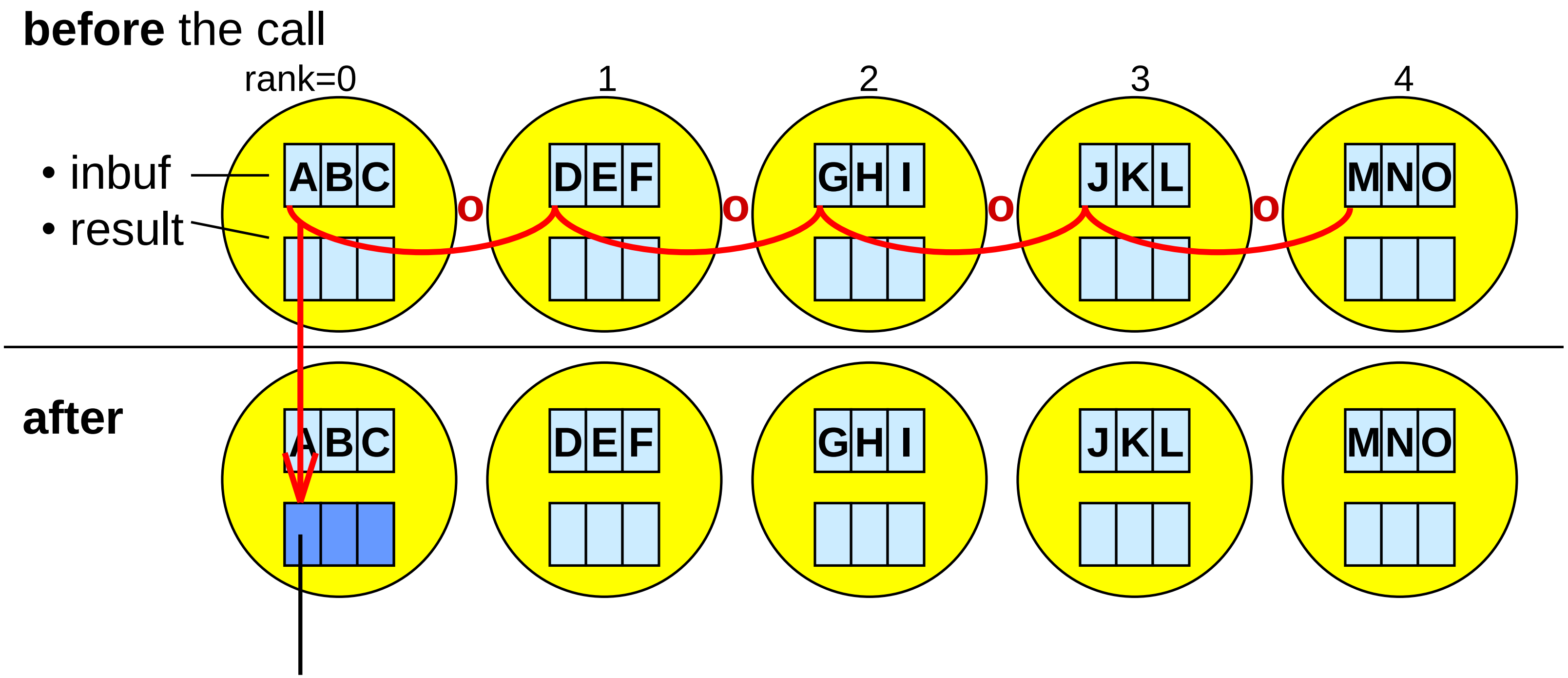
MPI_Scan:

MPI_Scan and MPI_Exscan



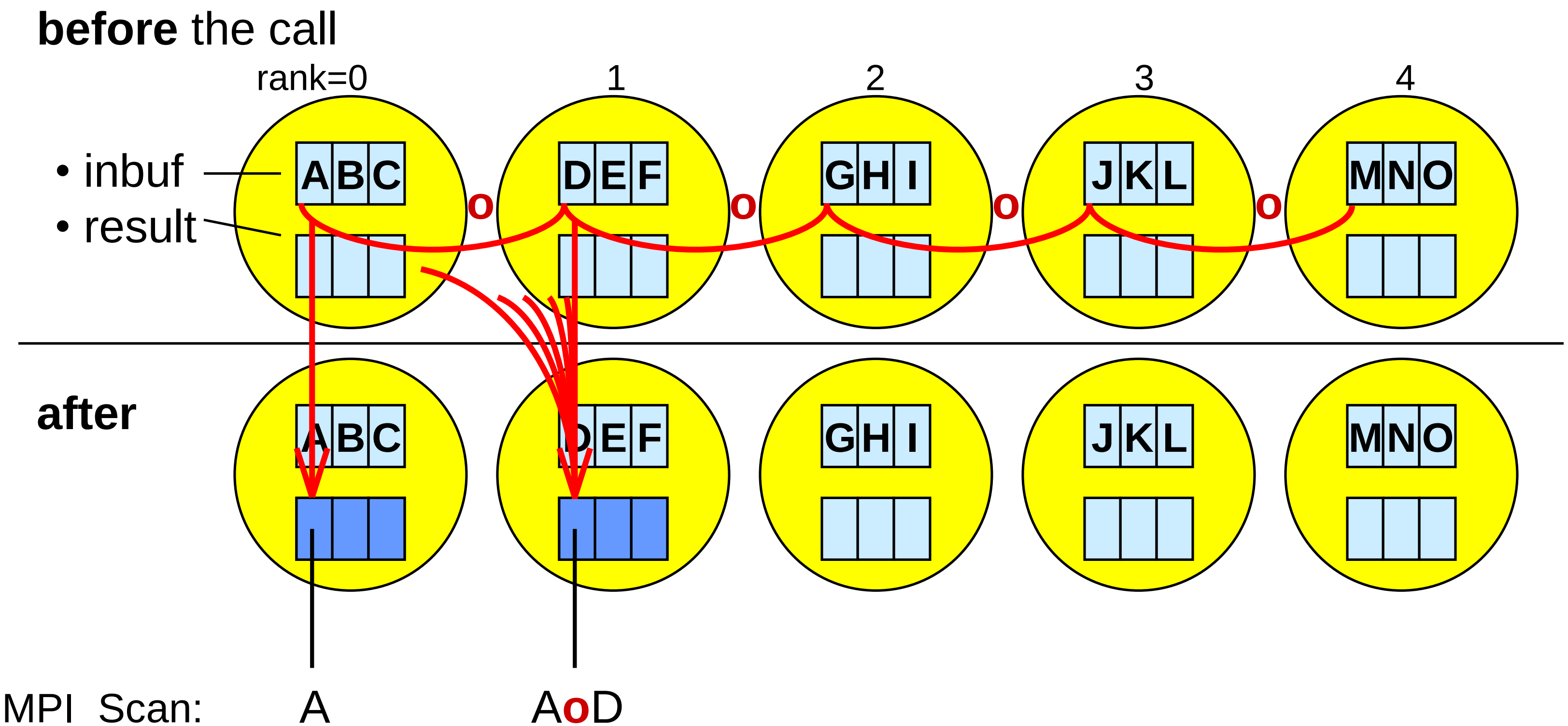
MPI_Scan:

MPI_Scan and MPI_Exscan

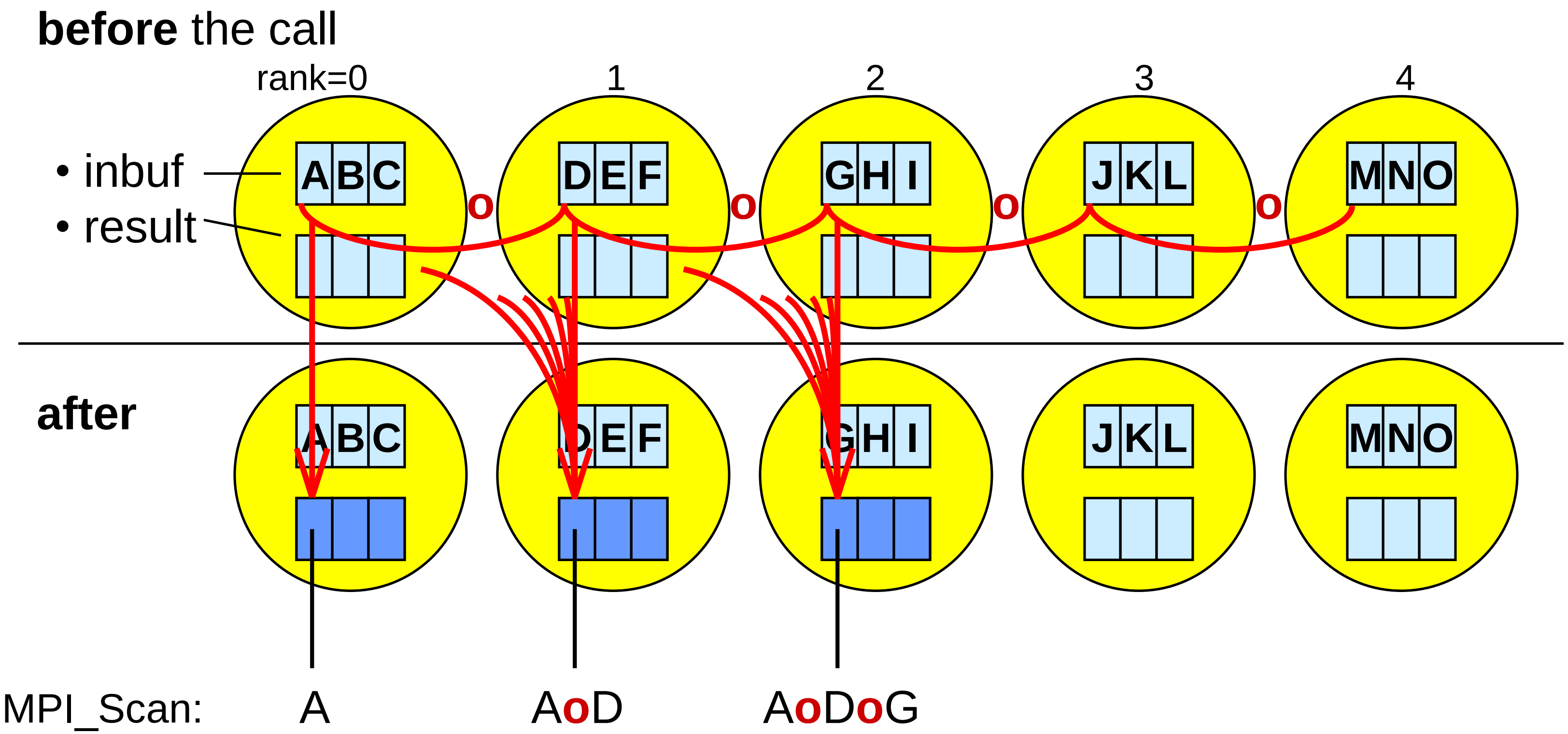


MPI_Scan: A

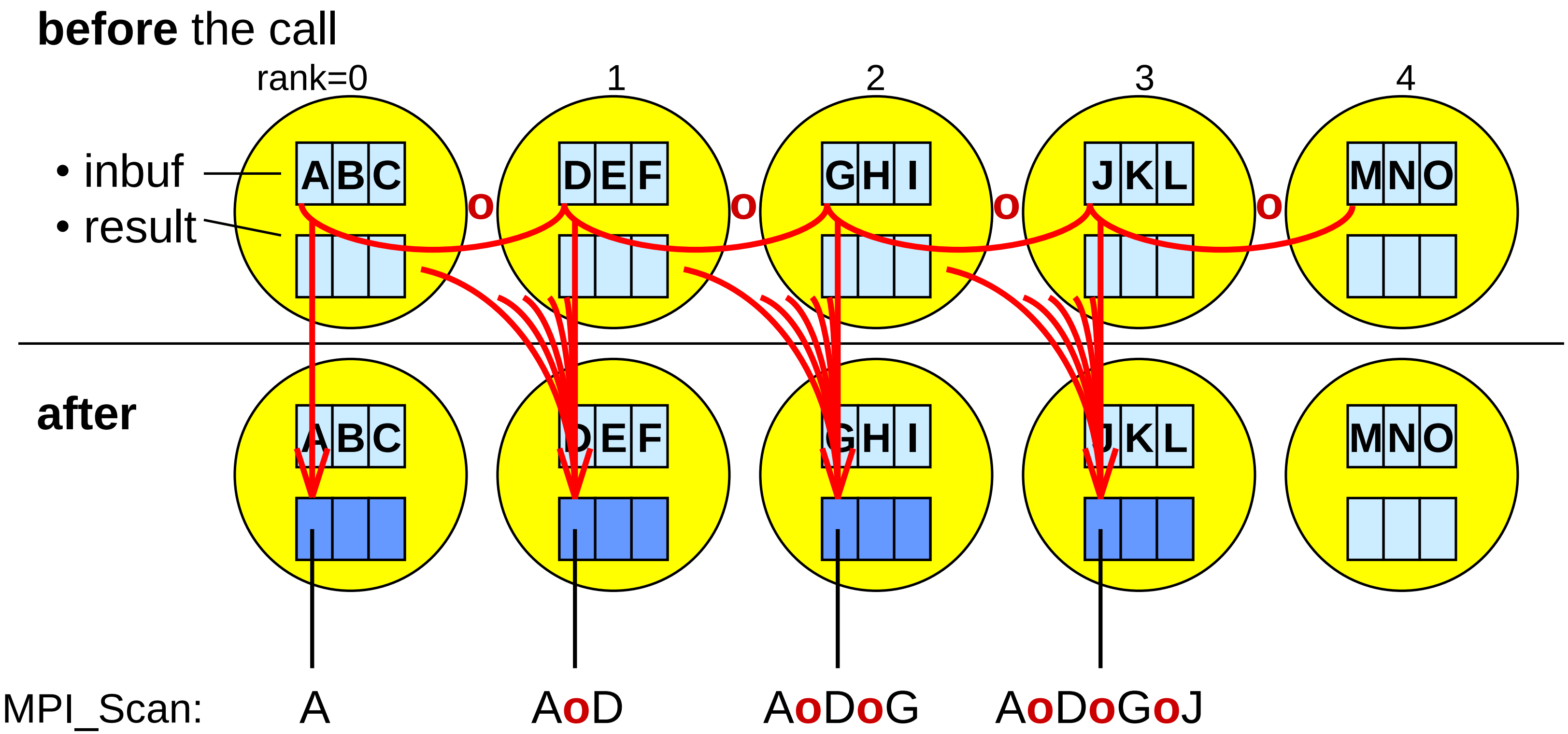
MPI_Scan and MPI_Exscan



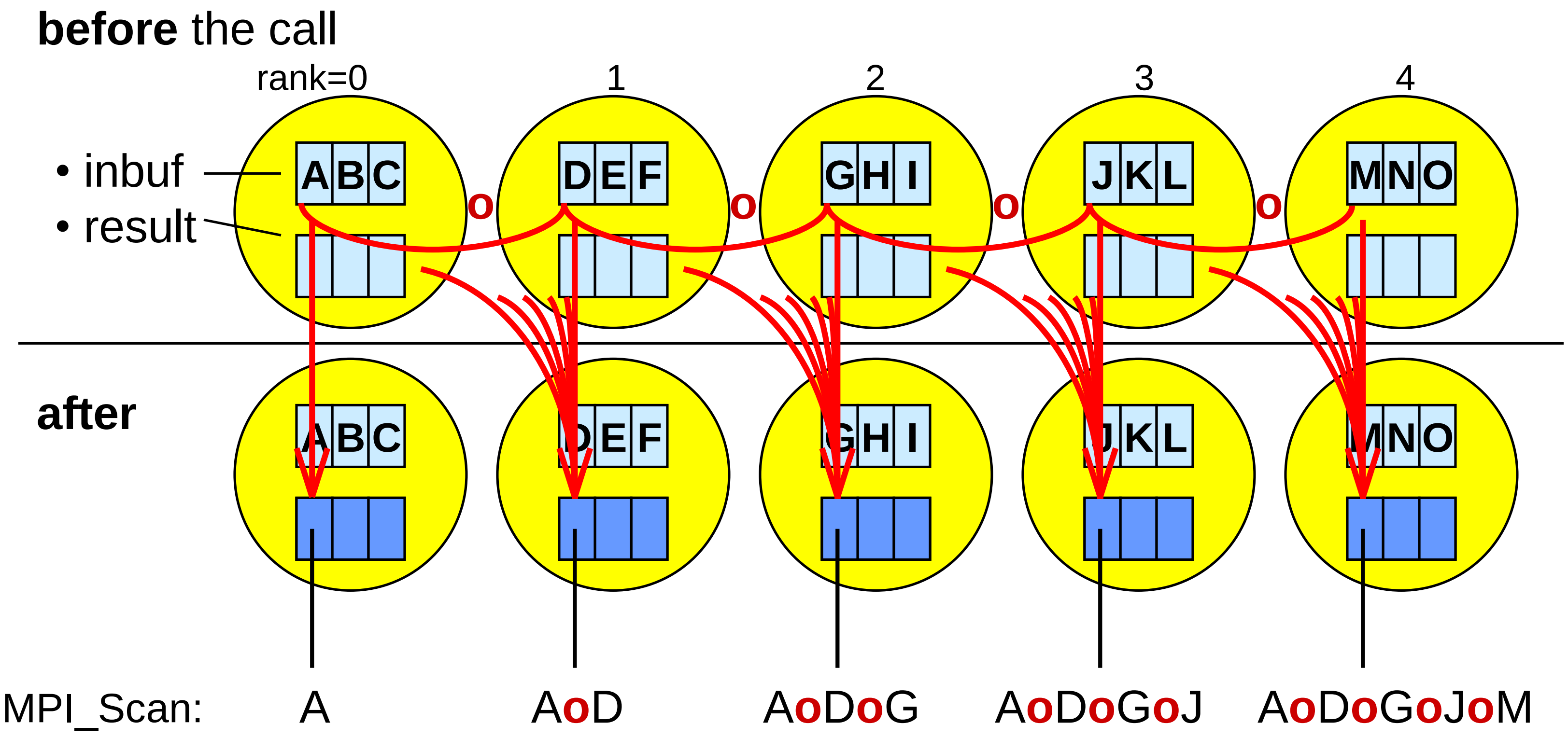
MPI_Scan and MPI_Exscan



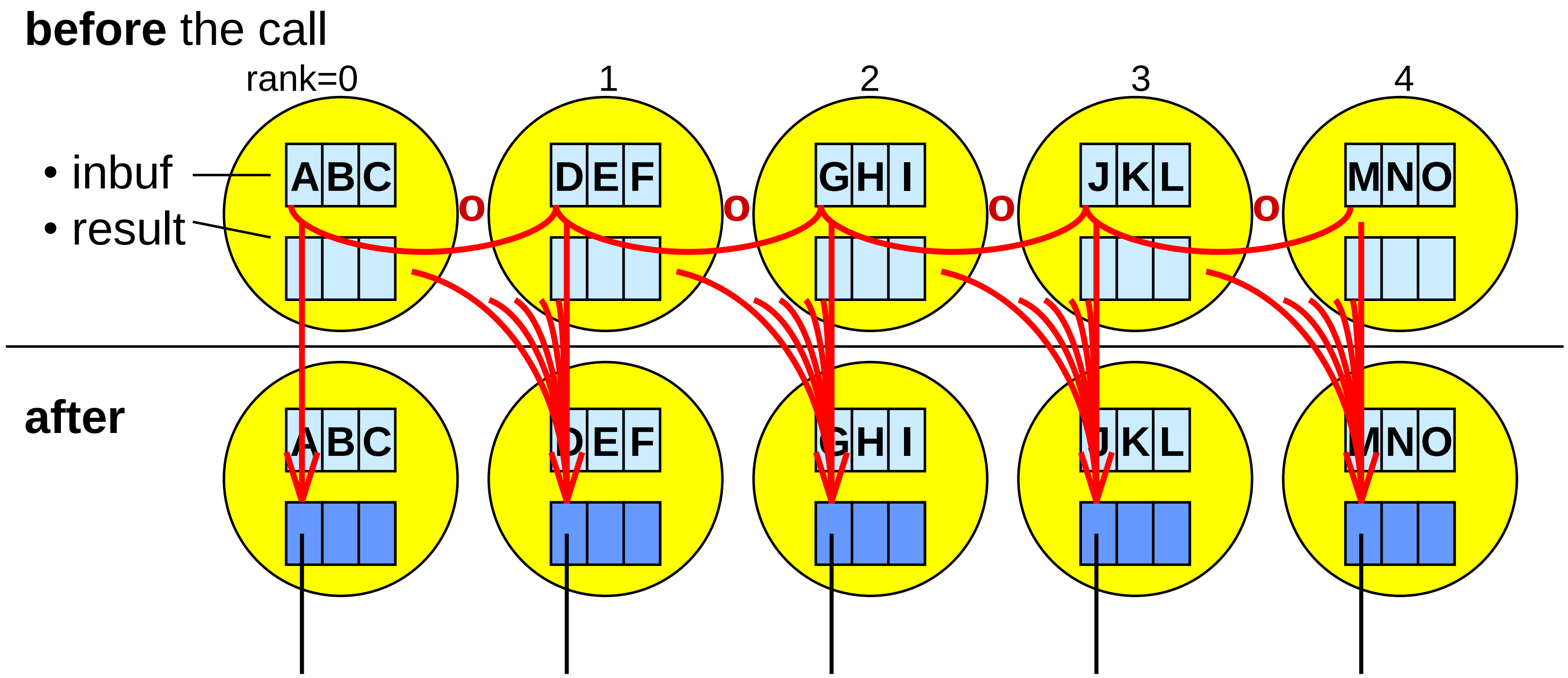
MPI_Scan and MPI_Exscan



MPI_Scan and MPI_Exscan

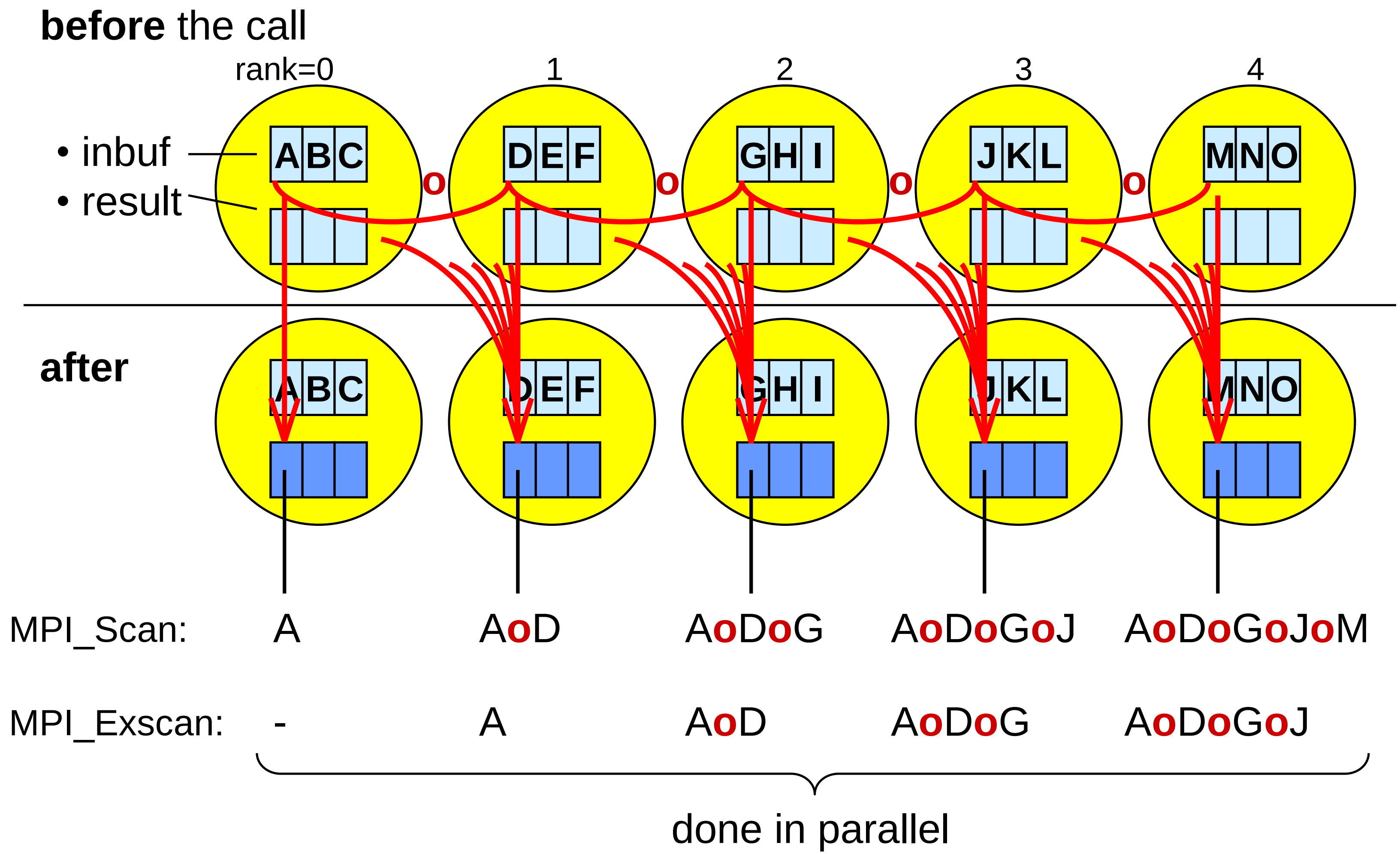


MPI_Scan and MPI_Exscan



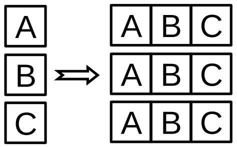
MPI_Scan:	A	AoD	AoDoG	AoDoGoJ	AoDoGoJoM
MPI_Exscan:	-	A	AoD	AoDoG	AoDoGoJ

MPI_Scan and MPI_Exscan



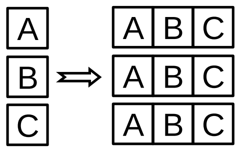
Other Collective Communication Routines

- MPI_Allgather → similar to MPI_Gather, but all processes receive the result vector

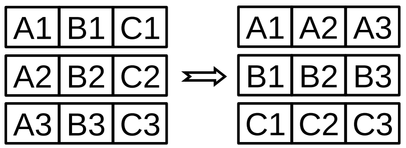


Other Collective Communication Routines

- MPI_Allgather → similar to MPI_Gather, but all processes receive the result vector

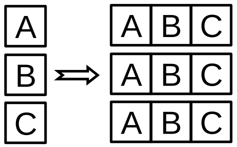


- MPI_Alltoall → each process sends messages to all processes

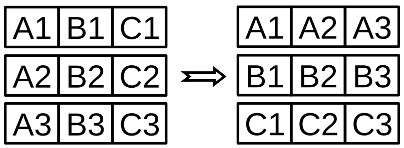


Other Collective Communication Routines

- MPI_Allgather → similar to MPI_Gather, but all processes receive the result vector



- MPI_Alltoall → each process sends messages to all processes



- MPI_.........v (Gatherv, Scatterv, Allgatherv, Alltoallv, Alltoallw)
  - → Each message has a different count and displacement
  - → array of counts and array of displs (Alltoallw: also array of types)
  - → interface does **not scale** to thousands of MPI processes!
  - → Recommendation: One should try to use data structures with same communication size on all ranks.

Exercise 2 — Global reduction

- Rewrite the pass-around-the-ring program to use the MPI global reduction to perform the global sum of all ranks of the processes in the ring (and print it from all processes).
- Use C C/Ch6/allreduce-skel.c or Fortran F_30/Ch6/allreduce-skel_30.f90 or Python PY/Ch6/allreduce-skel.py
- I.e., the pass-around-the-ring communication loop must be totally substituted by one call to the MPI collective reduction routine.
- For the argument list, of MPI_Allreduce, please look into the MPI standard:
  - Go to the end of the standard (=end of the *MPI function index of MPI-4.0*)
  - Go backward in the alphabet to MPI_Allreduce
  - Click on the underlined reference
    - MPI_Allreduce ….., 239, …… (in MPI-4.0)
      ….., 187, …… (in MPI-3.1)
  - Python: see also, e.g., mpi4py.MPI.Comm — MPI for Python 3.1.1 documentation
    - Specify **`sum`** in the same way as the **`rcv_buf`** in the ring algorithm

Advanced Exercises — Global scan and sub-groups

1. Global scan:
   - Rewrite the last program so that each process computes a partial sum, i.e., with MPI_Scan().
   - mpirun –np 5 ./a.out **| sort –n** to get the output sorted by the ranks:

     rank= 0 → sum=0
     rank= 1 → sum=1
     rank= 2 → sum=3
     rank= 3 → sum=6
     rank= 4 → sum=10

Quiz on Chapter 6-(1) – Collective communication

- Why should you use MPI collective routines?
  - ____
- MPI Collective communication: Which are the **major rules** when using **collective communication** routines and that **do not apply** to point to point communication? Please try to find at least two or three:
  1. ________________
  2. ________________
  3. ________________
  4. ________________
  5. ________________

Nonblocking Collective Communication Routines

New in MPI-3.0

MPI_I.......… **Nonblocking** variants of all collective communication:
MPI_Ibarrier, MPI_Ibcast, …

Nonblocking Collective Communication Routines

New in MPI-3.0

MPI_I.......… **Nonblocking** variants of all collective communication:
MPI_Ibarrier, MPI_Ibcast, …

- **Nonblocking** collective operations do **not match** with **blocking** collective operations

With point-to-point message passing, such matching is allowed

Nonblocking Collective Communication Routines

New in MPI-3.0

MPI_I.......... **Nonblocking** variants of all collective communication:
MPI_Ibarrier, MPI_Ibcast, …

- **Nonblocking** collective operations do **not match** with **blocking** collective operations

  With point-to-point message passing, such matching is allowed

- Collective initiation and completion are separated

Nonblocking Collective Communication Routines

New in MPI-3.0

MPI_I.......… **Nonblocking** variants of all collective communication:
MPI_Ibarrier, MPI_Ibcast, …

- **Nonblocking** collective operations do **not match** with **blocking** collective operations

  With point-to-point message passing, such matching is allowed

- Collective initiation and completion are separated
- **MPI_I…** calls are **local** (i.e., not synchronizing),
  whereas the **corresponding MPI_Wait** collectively **synchronizes**
  in same way as corresponding blocking collective procedure

Nonblocking Collective Communication Routines

New in MPI-3.0

MPI_I.......… **Nonblocking** variants of all collective communication:
MPI_Ibarrier, MPI_Ibcast, …

- **Nonblocking** collective operations do **not match** with **blocking** collective operations

  With point-to-point message passing, such matching is allowed

- Collective initiation and completion are separated
- **MPI_I…** calls are **local** (i.e., not synchronizing),
  whereas the **corresponding MPI_Wait** collectively **synchronizes**
  in same way as corresponding blocking collective procedure
- May have multiple outstanding collective communications on same communicator

Nonblocking Collective Communication Routines

New in MPI-3.0

MPI_I.......... **Nonblocking** variants of all collective communication:
MPI_Ibarrier, MPI_Ibcast, …

- **Nonblocking** collective operations do **not match** with **blocking** collective operations

With point-to-point message passing, such matching is allowed

- Collective initiation and completion are separated
- **MPI_I…** calls are **local** (i.e., not synchronizing), whereas the **corresponding MPI_Wait** collectively **synchronizes** in same way as corresponding blocking collective procedure
- May have multiple outstanding collective communications on same communicator
- Ordered initialization on each communicator

Nonblocking Collective Communication Routines

New in MPI-3.0

MPI_I.......… **Nonblocking** variants of all collective communication:
MPI_Ibarrier, MPI_Ibcast, …

- **Nonblocking** collective operations do **not match** with **blocking** collective operations

With point-to-point message passing, such matching is allowed

- Collective initiation and completion are separated
- **MPI_I…** calls are **local** (i.e., not synchronizing), whereas the **corresponding MPI_Wait** collectively **synchronizes** in same way as corresponding blocking collective procedure
- May have multiple outstanding collective communications on same communicator
- Ordered initialization on each communicator
- Parallel MPI I/O (except with shared file pointer):
The split collective interface may be deprecated in a future version of MPI

New in MPI-3.1

General progress rule of MPI

- MPI is mainly defined in a way that **progress** on communication (and …) is **required only during MPI procedure calls.**
- But then, progress is required
  - for **all** outstanding (incomplete/nonblocking) communications
  - together with operation of the current communication (…) procedure call.

General progress rule of MPI

- MPI is mainly defined in a way that **progress** on communication (and …) is **required only during MPI procedure calls.**
- But then, progress is required
  - for **all** outstanding (incomplete/nonblocking) communications
  - together with operation of the current communication (…) procedure call.
- See, e.g., in MPI-4.0
  - Sect. 3.5, page 54, and 3.7.4, page 75; Paragr.s “Progress”, esp. progress of repeated MPI_Test, p.75$_{38\text{-}40}$
  - Sect. 3.8.1 and 3.8.2 about MPI_(I)(M)PROBE
  - Sect. 3.8.4 Cancel, esp. page 94 lines 8-16 & MPI_Finalize Example 11.6, page 496$_{26\text{-}48}$ & MPI_Session_finalize, esp. page 503$_{30\text{-}47}$ and Example 11.8 on page 804
  - Sect. 4.2.2 MPI_Parrived: Same progress rule as for repeated MPI_Test, see page 111$_{31\text{-}34}$
  - Sect. 5.12: Nonblocking collectives: Same rules as for nonblocking pt-to-pt
  - Sect. 12.7.3: Progress with one-sided communication, especially the **rationale at the end**
  - Sect. 11.6: MPI and Threads
  - Sect. 14.6.3: Progress with MPI-I/O

General progress rule of MPI

- MPI is mainly defined in a way that **progress** on communication (and …) is **required only during MPI procedure calls.**
- But then, progress is required
  - for **all** outstanding (incomplete/nonblocking) communications
  - together with operation of the current communication (…) procedure call.
- See, e.g., in MPI-4.0
  - Sect. 3.5, page 54, and 3.7.4, page 75; Paragr.s “Progress”, esp. progress of repeated MPI_Test, p.75$_{38\text{-}40}$
  - Sect. 3.8.1 and 3.8.2 about MPI_(I)(M)PROBE
  - Sect. 3.8.4 Cancel, esp. page 94 lines 8-16 & MPI_Finalize Example 11.6, page 496$_{26\text{-}48}$ & MPI_Session_finalize, esp. page 503$_{30\text{-}47}$ and Example 11.8 on page 804
  - Sect. 4.2.2 MPI_Parrived: Same progress rule as for repeated MPI_Test, see page 111$_{31\text{-}34}$
  - Sect. 5.12: Nonblocking collectives: Same rules as for nonblocking pt-to-pt
  - Sect. 12.7.3: Progress with one-sided communication, especially the **rationale at the end**
  - Sect. 11.6: MPI and Threads
  - Sect. 14.6.3: Progress with MPI-I/O
- Non of these rules require progress outside of called MPI routines,
  - But MPI_Test and each MPI routine that blocks must do progress on any ongoing (i.e. nonblocking) communication

General progress rule of MPI

- MPI is mainly defined in a way that **progress** on communication (and …) is **required only during MPI procedure calls.**
- But then, progress is required
  - for **all** outstanding (incomplete/nonblocking) communications
  - together with operation of the current communication (…) procedure call.
- See, e.g., in MPI-4.0
  - Sect. 3.5, page 54, and 3.7.4, page 75; Paragr.s “Progress”, esp. progress of repeated MPI_Test, p.75$_{38-40}$
  - Sect. 3.8.1 and 3.8.2 about MPI_(I)(M)PROBE
  - Sect. 3.8.4 Cancel, esp. page 94 lines 8-16 & MPI_Finalize Example 11.6, page 496$_{26-48}$ & MPI_Session_finalize, esp. page 503$_{30-47}$ and Example 11.8 on page 804
  - Sect. 4.2.2 MPI_Parrived: Same progress rule as for repeated MPI_Test, see page 111$_{31-34}$
  - Sect. 5.12: Nonblocking collectives: Same rules as for nonblocking pt-to-pt
  - Sect. 12.7.3: Progress with one-sided communication, especially the **rationale at the end**
  - Sect. 11.6: MPI and Threads
  - Sect. 14.6.3: Progress with MPI-I/O
- Non of these rules require progress outside of called MPI routines,
  - But MPI_Test and each MPI routine that blocks must do progress on any ongoing (i.e. nonblocking) communication
- Additional progress
  - By several calls to MPI_Test(), which enables progress

General progress rule of MPI

- MPI is mainly defined in a way that **progress** on communication (and …) is **required only during MPI procedure calls.**
- But then, progress is required
  - for **all** outstanding (incomplete/nonblocking) communications
  - together with operation of the current communication (…) procedure call.
- See, e.g., in MPI-4.0
  - Sect. 3.5, page 54, and 3.7.4, page 75; Paragr.s "Progress", esp. progress of repeated MPI_Test, p.75 38-40
  - Sect. 3.8.1 and 3.8.2 about MPI_(I)(M)PROBE
  - Sect. 3.8.4 Cancel, esp. page 94 lines 8-16 & MPI_Finalize Example 11.6, page 496 26-48 & MPI_Session_finalize, esp. page 503 30-47 and Example 11.8 on page 804
  - Sect. 4.2.2 MPI_Parrived: Same progress rule as for repeated MPI_Test, see page 111 31-34
  - Sect. 5.12: Nonblocking collectives: Same rules as for nonblocking pt-to-pt
  - Sect. 12.7.3: Progress with one-sided communication, especially the **rationale at the end**
  - Sect. 11.6: MPI and Threads
  - Sect. 14.6.3: Progress with MPI-I/O
- Non of these rules require progress outside of called MPI routines,
  - But MPI_Test and each MPI routine that blocks must do progress on any ongoing (i.e. nonblocking) communication
- Additional progress
  - By several calls to MPI_Test(), which enables progress
  - Use non-standard extensions to switch on asynchronous progress
    - E.g., with MPICH:
      export MPICH_ASYNC_PROGRESS=1

Implies a helper thread and MPI_THREAD_MULTIPLE

General progress rule of MPI

- MPI is mainly defined in a way that **progress** on communication (and …) is **required only during MPI procedure calls.**
- But then, progress is required
  - for **all** outstanding (incomplete/nonblocking) communications
  - together with operation of the current communication (…) procedure call.
- See, e.g., in MPI-4.0
  - Sect. 3.5, page 54, and 3.7.4, page 75; Paragr.s "Progress", esp. progress of repeated MPI_Test, p.75$_{38\text{-}40}$
  - Sect. 3.8.1 and 3.8.2 about MPI_(I)(M)PROBE
  - Sect. 3.8.4 Cancel, esp. page 94 lines 8-16 & MPI_Finalize Example 11.6, page 496$_{26\text{-}48}$ & MPI_Session_finalize, esp. page 503$_{30\text{-}47}$ and Example 11.8 on page 804
  - Sect. 4.2.2 MPI_Parrived: Same progress rule as for repeated MPI_Test, see page 111$_{31\text{-}34}$
  - Sect. 5.12: Nonblocking collectives: Same rules as for nonblocking pt-to-pt
  - Sect. 12.7.3: Progress with one-sided communication, especially the **rationale at the end**
  - Sect. 11.6: MPI and Threads
  - Sect. 14.6.3: Progress with MPI-I/O
- Non of these rules require progress outside of called MPI routines,
  - But MPI_Test and each MPI routine that blocks must do progress on any ongoing (i.e. nonblocking) communication
- Additional progress
  - By several calls to MPI_Test(), which enables progress
  - Use non-standard extensions to switch on asynchronous progress
    - E.g., with MPICH:
      export MPICH_ASYNC_PROGRESS=1

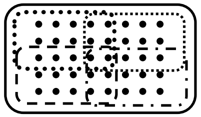
Implies a helper thread and MPI_THREAD_MULTIPLE

Opportunities with Nonblocking Collectives

- Offers opportunity to overlap

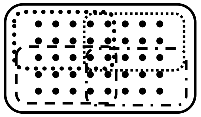
Opportunities with Nonblocking Collectives

- Offers opportunity to overlap
  - several collective communications,
    e.g., on several overlapping communicators
    - **Without deadlocks or serializations!**



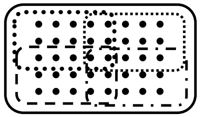
Opportunities with Nonblocking Collectives

- Offers opportunity to overlap
  - several collective communications,
    e.g., on several overlapping communicators
    - **Without deadlocks or serializations!**
  - computation and communication



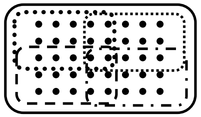
Opportunities with Nonblocking Collectives

- Offers opportunity to overlap
  - several collective communications,
    e.g., on several overlapping communicators
    - **Without deadlocks or serializations!**
  - computation and communication
    - **For this, progress is needed**



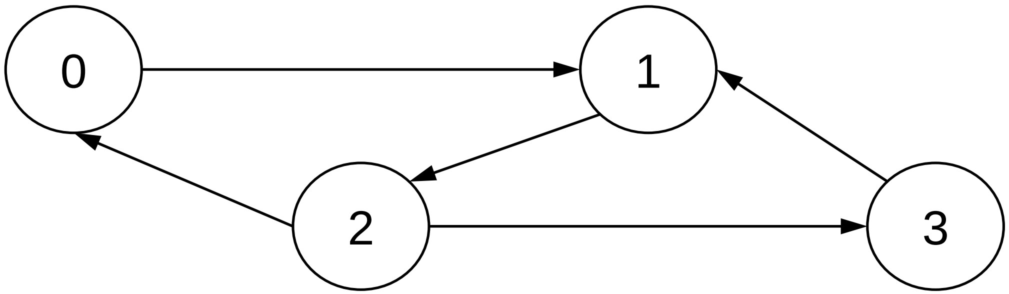
Opportunities with Nonblocking Collectives

- Offers opportunity to overlap
  - several collective communications,
    e.g., on several overlapping communicators
    - **Without deadlocks or serializations!**
  - computation and communication
    - **For this, progress is needed**
    - **See previous slide**



Nonblocking Barrier: Functional Opportunities – an Example

- The receiver
  - needs information and
  - does not know the sending processes nor the number of sending processes (**nsp**)
  - and this number is small compared to the total number.
  - The sender knows all its neighbors, which need some data.
- Non-scalable solution to exchange number of neighbors:
  - **MPI_Alltoall**, **MPI_Reduce_scatter** (array with one logical entry per process)
  - Each sender tells all processes whether they will get a message or not.



- For the example with MPI_Ibarrier on next slide, we also need the following *local inquiry procedure*:
  - **MPI_Iprobe**(int source, int tag, MPI_Comm comm, int **flag*, MPI_Status **status*);
    Python: flag = comm**.**Iprobe(source, tag, status)
  - Result: flag == non-zero or .TRUE. ➔ a message arrived and can be received with a local MPI_Recv,
    i.e., a subsequent corresponding MPI_Recv will **not** block
    flag == 0 or .FALSE. ➔ currently no incoming message with given source rank & tag & comm

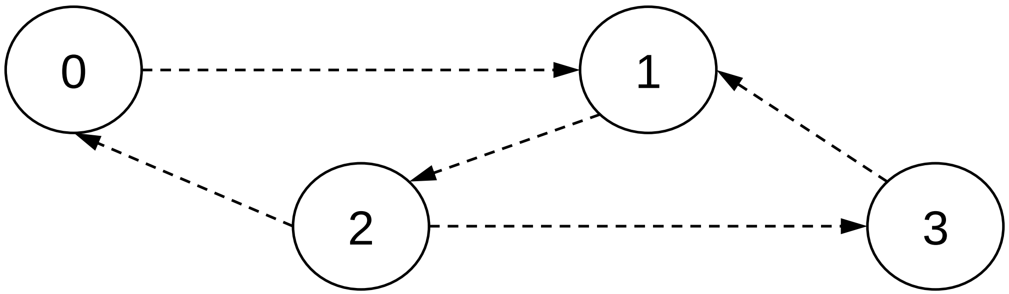
Nonblocking Barrier:
Functional Opportunities – an Example

Principles:

1. **Ssend** reports to the sender that **Recv** is called on the other side.
2. **Ibarrier** completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  - *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**
  - LOOP
    - *Process in the role being a receiver:*
      MPI_Iprobe(MPI_ANY_SOURCE,...); If there is a message then MPI_Recv for this one msg
    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed
      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)
  - UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)

Important: The S=synchronous reports back to the sender that the RECV is called!

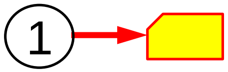


Nonblocking Barrier: Functional Opportunities – an Example

Principles:

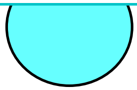
1. **Ssend** reports to the sender that **Recv** is called on the other side.
2. **Ibarrier** completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  – *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**

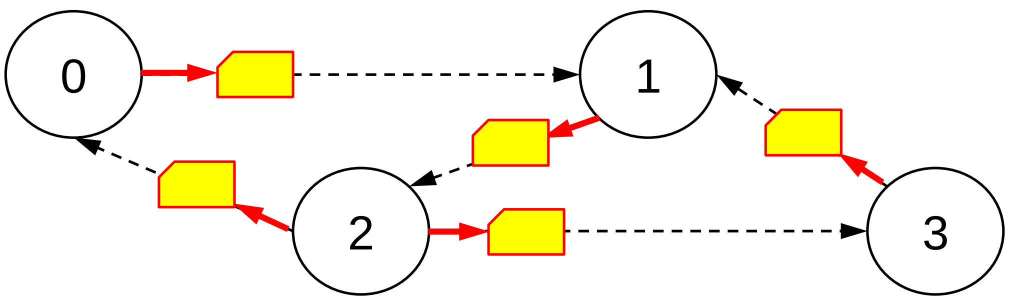


  – LOOP
    - *Process in the role being a receiver:*
      MPI_Iprobe(MPI_ANY_SOURCE,…); If there is a message then MPI_Recv for this one msg
    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed
      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)

Important: The S=synchronous reports back to the sender that the RECV is called!



  – UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)

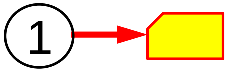


Nonblocking Barrier:
Functional Opportunities – an Example

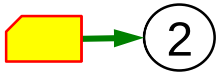
Principles:

1. **Ssend** reports to the sender that **Recv** is called on the other side.
2. **Ibarrier** completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  – *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**

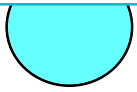


  – LOOP
    - *Process in the role being a receiver:*

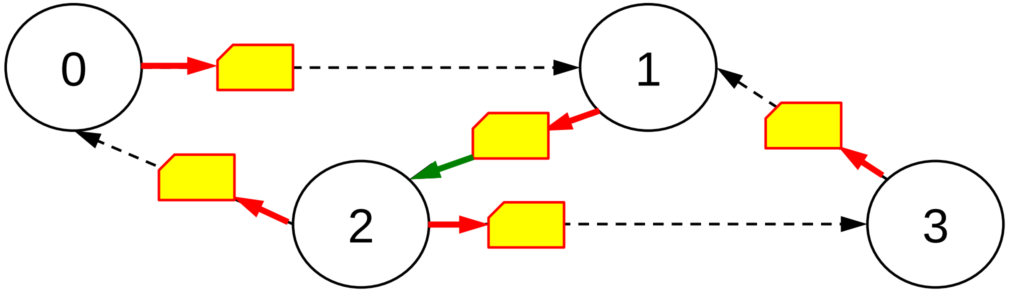


      MPI_Iprobe(MPI_ANY_SOURCE,...); If there is a message then MPI_Recv for this one msg
    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed
      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)

Important: The S=synchronous reports back to the sender that the RECV is called!



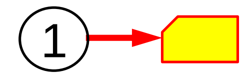
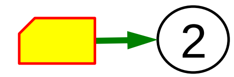
  – UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)



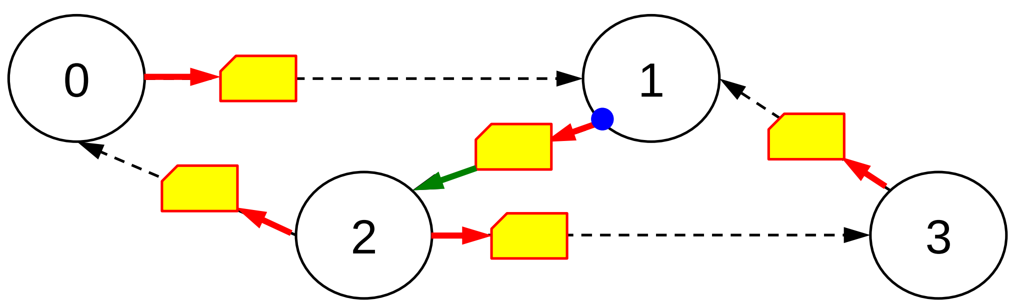
Nonblocking Barrier: Functional Opportunities – an Example

Principles:

1. **Ssend** reports to the sender that **Recv** is called on the other side.
2. **Ibarrier** completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  - *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND** 
  - LOOP
    - *Process in the role being a receiver:* 
      MPI_Iprobe(MPI_ANY_SOURCE,...); If there is a message then MPI_Recv for this one msg
    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed
      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)

      Important: The S=synchronous reports back to the sender that the RECV is called!
  - UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)

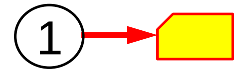


Nonblocking Barrier: Functional Opportunities – an Example

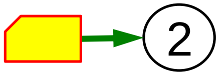
Principles:

1. **Ssend** reports to the sender that **Recv** is called on the other side.
2. **Ibarrier** completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  - *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**

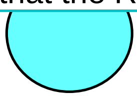


  - LOOP
    - *Process in the role being a receiver:*
      MPI_Iprobe(MPI_ANY_SOURCE,...); If there is a message then MPI_Recv for this one msg

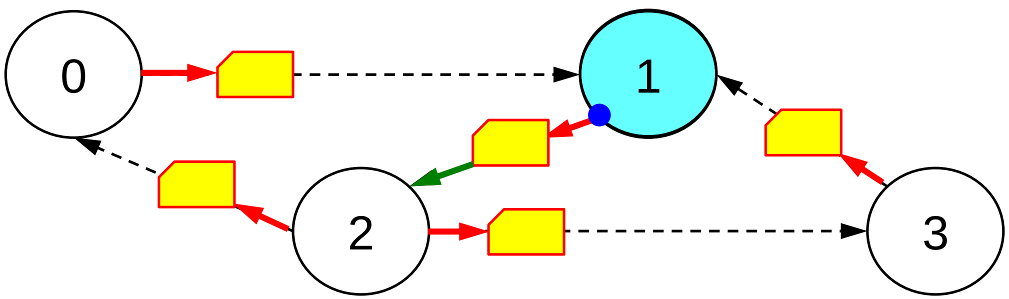


    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed
      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)

Important: The S=synchronous reports back to the sender that the RECV is called!



  - UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)

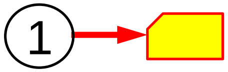


Nonblocking Barrier:
Functional Opportunities – an Example

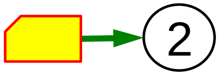
Principles:

1. **Ssend** reports to the sender that **Recv** is called on the other side.
2. **Ibarrier** completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  – *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**



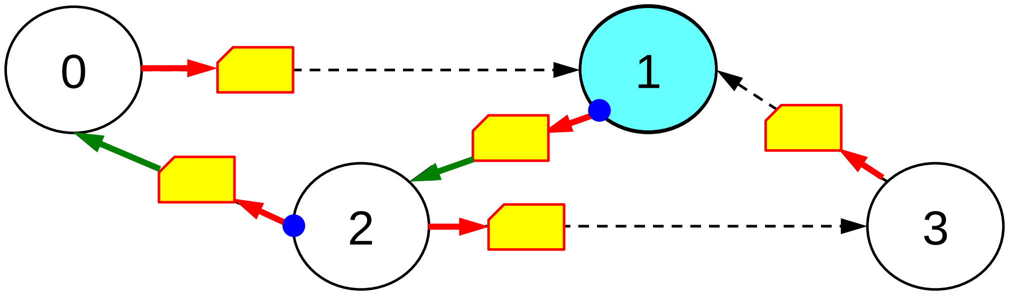
  – LOOP
    - *Process in the role being a receiver:*
      MPI_Iprobe(MPI_ANY_SOURCE,…); If there is a message then MPI_Recv for this one msg



    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed
      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)

Important: The S=synchronous reports back to the sender that the RECV is called!

  – UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)

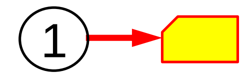


Nonblocking Barrier: Functional Opportunities – an Example

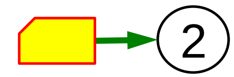
Principles:

1. **Ssend** reports to the sender that **Recv** is called on the other side.
2. **Ibarrier** completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  – *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**

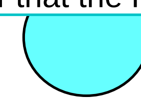


  – LOOP
    - *Process in the role being a receiver:*

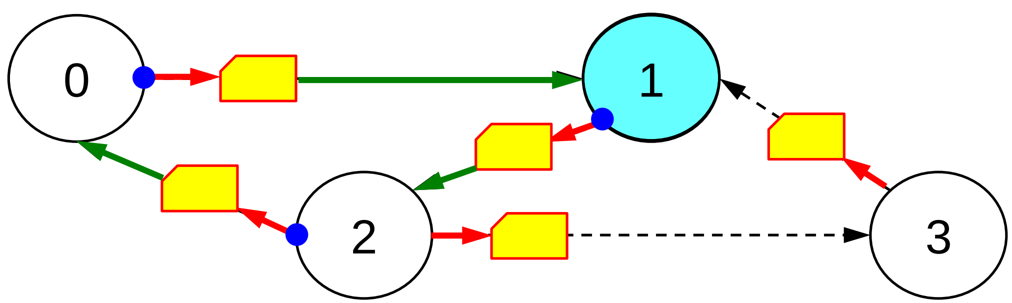


      MPI_Iprobe(MPI_ANY_SOURCE,...); If there is a message then MPI_Recv for this one msg
    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed
      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)

Important: The S=synchronous reports back to the sender that the RECV is called!



  – UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)

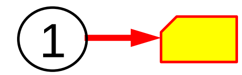


Nonblocking Barrier:
Functional Opportunities – an Example

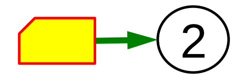
Principles:

1. **Ssend** reports to the sender that **Recv** is called on the other side.
2. **Ibarrier** completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

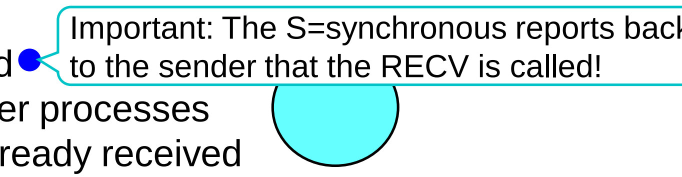
- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  – *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**



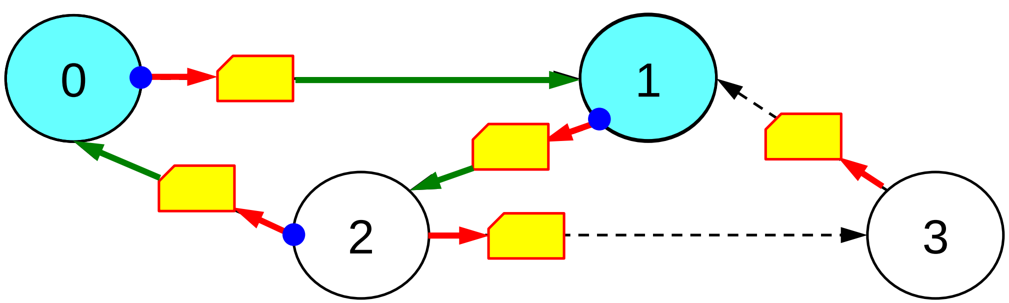
  – LOOP
    - *Process in the role being a receiver:*
      MPI_Iprobe(MPI_ANY_SOURCE,...); If there is a message then MPI_Recv for this one msg



    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed
      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)



  – UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)

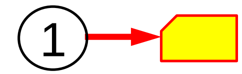


Nonblocking Barrier: Functional Opportunities – an Example

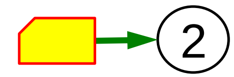
Principles:

1. **Ssend** reports to the sender that **Recv** is called on the other side.
2. **Ibarrier** completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  – *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**

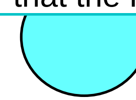


  – LOOP
    - *Process in the role being a receiver:*

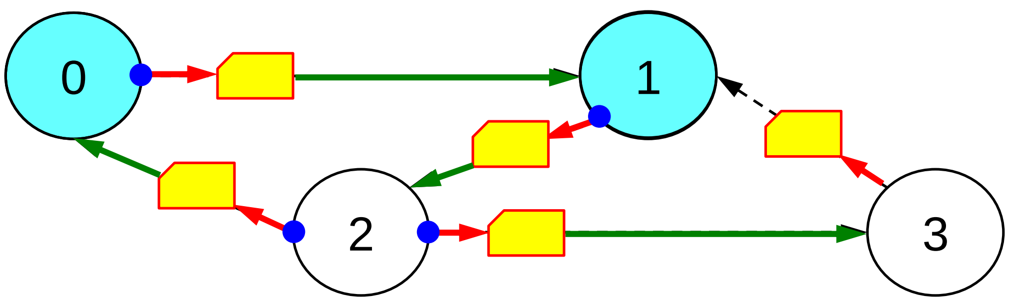


      MPI_Iprobe(MPI_ANY_SOURCE,...); If there is a message then MPI_Recv for this one msg
    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed

      Important: The S=synchronous reports back to the sender that the RECV is called!



      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)
  – UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)



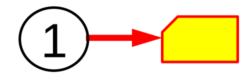
Nonblocking Barrier: Functional Opportunities – an Example

Principles:

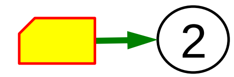
1. Ssend reports to the sender that **Recv** is called on the other side.

2. Ibarrier completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  - *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**

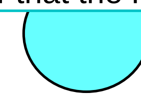


  - LOOP
    - *Process in the role being a receiver:*
      MPI_Iprobe(MPI_ANY_SOURCE,…); If there is a message then MPI_Recv for this one msg

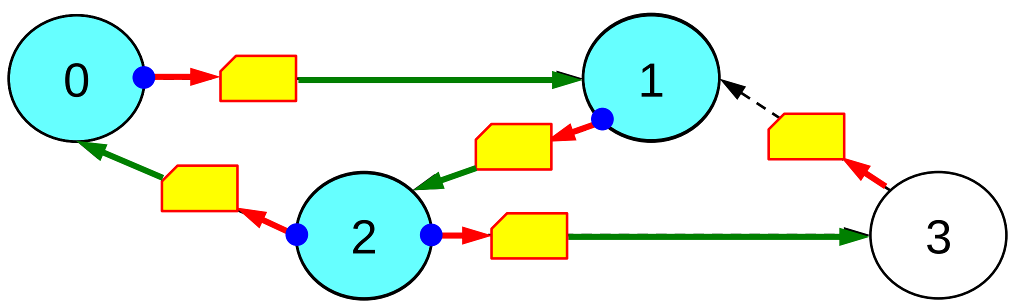


    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed
      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)

Important: The S=synchronous reports back to the sender that the RECV is called!



  - UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)



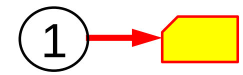
Nonblocking Barrier: Functional Opportunities – an Example

Principles:

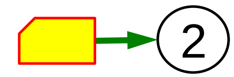
1. Ssend reports to the sender that **Recv** is called on the other side.

2. Ibarrier completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  - *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**

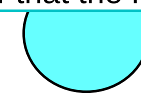


  - LOOP
    - *Process in the role being a receiver:*

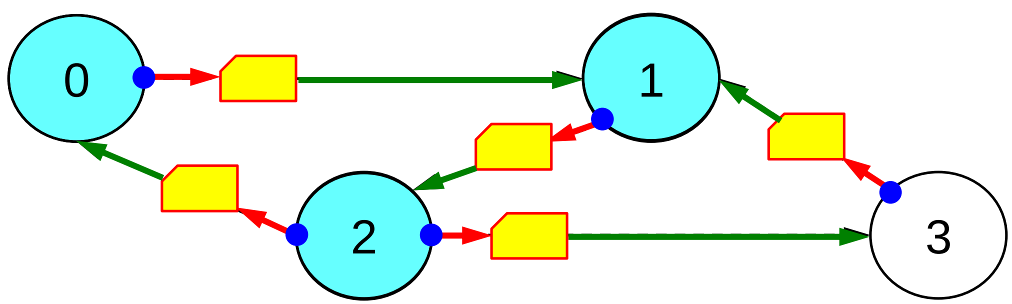


      MPI_Iprobe(MPI_ANY_SOURCE,...); If there is a message then MPI_Recv for this one msg
    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed

      Important: The S=synchronous reports back to the sender that the RECV is called!



      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)
  - UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)



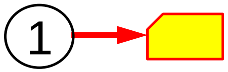
Nonblocking Barrier: Functional Opportunities – an Example

Principles:

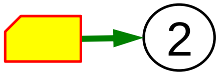
1. Ssend reports to the sender that **Recv** is called on the other side.

2. Ibarrier completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  - *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**

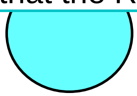


  - LOOP
    - *Process in the role being a receiver:*

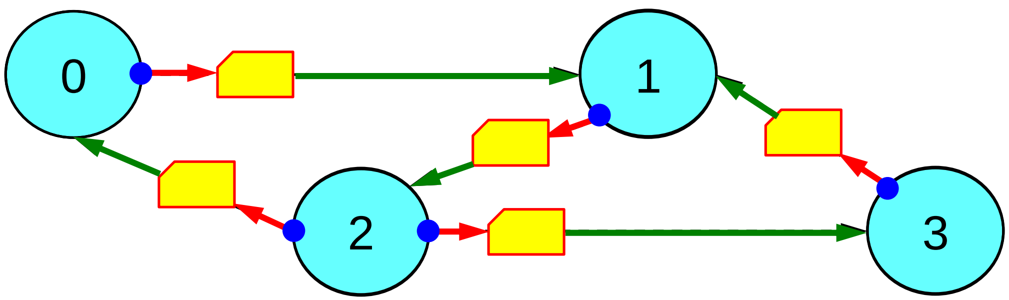


      MPI_Iprobe(MPI_ANY_SOURCE,...); If there is a message then MPI_Recv for this one msg
    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed

      Important: The S=synchronous reports back to the sender that the RECV is called!



      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)
  - UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)

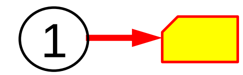


Nonblocking Barrier: Functional Opportunities – an Example

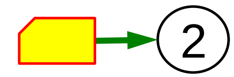
Principles:

1. **Ssend** reports to the sender that **Recv** is called on the other side.
2. **Ibarrier** completes when **all** processes reported (by starting the **Ibarrier**) that **all** their **Ssend** calls are received on their other sides, i.e., completely all **Recv** calls are called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.
- Solution with nonblocking barrier:
  - *Each process as a sender*
    - **Loop over its neighbors, sending the data with MPI_ISSEND**



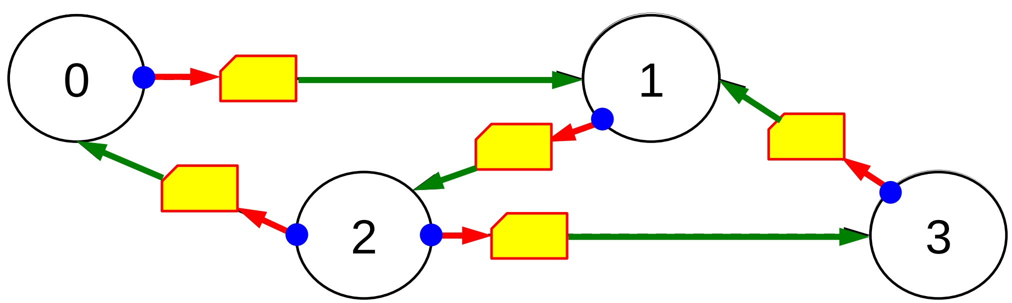
  - LOOP
    - *Process in the role being a receiver:*



      MPI_Iprobe(MPI_ANY_SOURCE,…); If there is a message then MPI_Recv for this one msg
    - *Process in the role being a sender:*
      Check whether all **Issend** calls are completed

      Important: The S=synchronous reports back to the sender that the RECV is called!

      → then start **MPI_Ibarrier** to signal to all other processes that all **MPI_Issend** of this process are already received (i.e. the corresponding **MPI_Recv** is already called)
  - UNTIL **MPI_Ibarrier** completed (i.e. all processes signaled that all receives are called)



Collective Operations for Intercommunicators

- In MPI-1, collective operations are restricted to ordinary (intra) communicators.
- In MPI-2, most collective operations are extended by an additional functionality for intercommunicators
  - e.g., Bcast on a parents-children intercommunicator: sends data from one parent process to all children.
- Intercommunicators do not apply in
  - MPI_Scan, MPI_Iscan, MPI_Exscan, MPI_Iexscan,
  - MPI_(I)Neighbor_allgather(v)
  - MPI_(I)Neighbor_alltoall(v,w)

New in MPI-3.0

Sparse Collective Operations on Process Topology

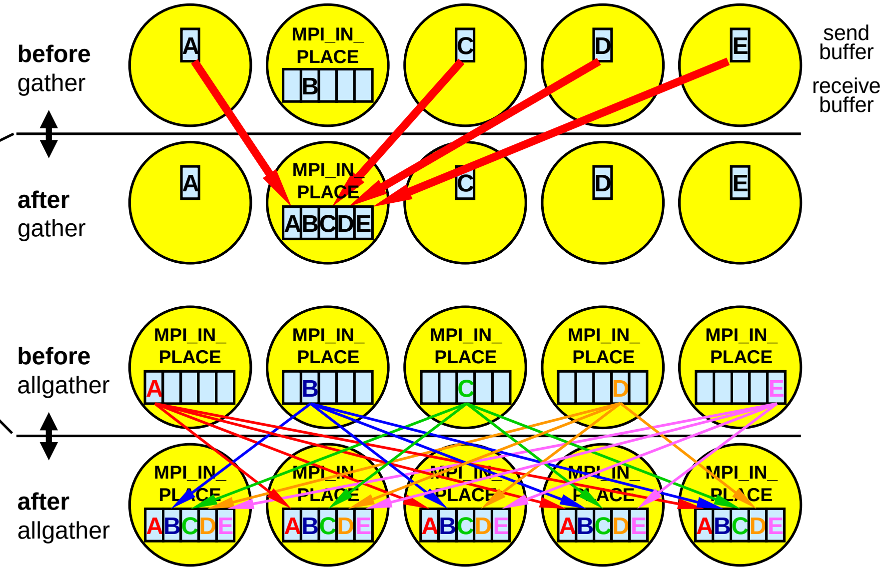
- MPI process topologies (Cartesian and (distributed) graph) usable for communication
  - MPI_(I)NEIGHBOR_ALLGATHER(V)
  - MPI_(I)NEIGHBOR_ALLTOALL(V,W)
- If the topology is the full graph, then neighbor routine is identical to full collective communication routine
  - Exception: s/rdispls in MPI_NEIGHBOR_ALLTOALLW are MPI_Aint
- Allows for optimized communication scheduling and scalable resource binding
- Cartesian topology:
  - Sequence of buffer segments is communicated with:
    - **direction=0 source, direction=0 dest, direction=1 source, direction=1 dest, …**
  - Defined only for disp=1
  - If a source or dest rank is MPI_PROC_NULL then the buffer location is still there but the content is not touched.

Extended Collective Operations — "In place" Buffer Specification

The **MPI_IN_PLACE** has two meanings:

- to **prohibit the local copy** with
  → sendbuf=**MPI_IN_PLACE**:
  - (I)GATHER(V) at root process
  - (I)ALLGATHER(V) at all processes

  → recvbuf=**MPI_IN_PLACE**:
  - (I)SCATTER(V) at root process



- to **overwrite input buffer** with the result: (sendbuf=**MPI_IN_PLACE**, input is taken from recvbuf, which is then overwitten)
  - (I)REDUCE at root
  - (I)ALLREDUCE, (I)REDUCE_SCATTER(_BLOCK), (I)SCAN, (I)EXSCAN, (I)ALLTOALL(V,W) at all processes
- Not available for
  - (I)BARRIER, (I)BCAST, (I)NEIGHBOR_ALLGATHER/ALLTOALL(V,W)
- Python: the constant is MPI.IN_PLACE

Exercise 3 — nonblocking barrier

In MPI/tasks/…

- Use C C/Ch6/ibarrier-skel.c or Fortran F_30/Ch6/ibarrier-skel_30.f90 or Python PY/Ch6/ibarrier-skel.py
- Each process sends 0-4 messages to some other processes (see number_of_dests).
- The skeletons include already the Issends of these messages.

Exercise 3

Exercise 3 — nonblocking barrier

In MPI/tasks/…

- Use **C** C/Ch6/ibarrier-skel.c or **Fortran** F_30/Ch6/ibarrier-skel_30.f90 or **Python** PY/Ch6/ibarrier-skel.py
- Each process sends 0-4 messages to some other processes (see number_of_dests).
- The skeletons include already the Issends of these messages.
- The receiving processes do not know
  - how many messages must be received, and
  - from which processes they will come.

Exercise 3

Exercise 3 — nonblocking barrier

In MPI/tasks/…

- Use **C** C/Ch6/ibarrier-skel.c or **Fortran** F_30/Ch6/ibarrier-skel_30.f90 or **Python** PY/Ch6/ibarrier-skel.py
- Each process sends 0-4 messages to some other processes (see number_of_dests).
- The skeletons include already the Issends of these messages.
- The receiving processes do not know
  - how many messages must be received, and
  - from which processes they will come.
- The skeleton also includes the Iprobe(…) **[please add the arguments]** and the Recv()

Exercise 3

Exercise 3

Exercise 3 — nonblocking barrier

In MPI/tasks/…

- Use **C** C/Ch6/ibarrier-skel.c or **Fortran** F_30/Ch6/ibarrier-skel_30.f90 or **Python** PY/Ch6/ibarrier-skel.py
- Each process sends 0-4 messages to some other processes (see number_of_dests).
- The skeletons include already the Issends of these messages.
- The receiving processes do not know
  - how many messages must be received, and
  - from which processes they will come.
- The skeleton also includes the Iprobe(…) **[please add the arguments]** and the Recv()
- You should add the sender-side part of the nonblocking barrier algorithm presented within this course chapter. Hints:
  - With which one call can you check for the completeness of all nonblocking send requests?
  - MPI_Ibarrier(*comm*, &ib_rq) should be called only once!
  - The MPI_Test(&ib_rq, …) can be done only when MPI_Ibarrier is already called (arguments →)

Exercise 3

Exercise 3 — nonblocking barrier

In MPI/tasks/…

- Use **C** C/Ch6/ibarrier-skel.c or **Fortran** F_30/Ch6/ibarrier-skel_30.f90 or **Python** PY/Ch6/ibarrier-skel.py
- Each process sends 0-4 messages to some other processes (see number_of_dests).
- The skeletons include already the Issends of these messages.
- The receiving processes do not know
  - how many messages must be received, and
  - from which processes they will come.
- The skeleton also includes the Iprobe(…) **[please add the arguments]** and the Recv()
- You should add the sender-side part of the nonblocking barrier algorithm presented within this course chapter. Hints:
  - With which one call can you check for the completeness of all nonblocking send requests?
  - MPI_Ibarrier(*comm*, &ib_rq) should be called only once!
  - The MPI_Test(&ib_rq, …) can be done only when MPI_Ibarrier is already called (arguments →)
- Please only fill in the ________ parts. Please do not modify the already given source code.

Exercise 3 — nonblocking barrier

In MPI/tasks/…

Exercise 3

- Use **C** C/Ch6/ibarrier-skel.c or **Fortran** F_30/Ch6/ibarrier-skel_30.f90 or **Python** PY/Ch6/ibarrier-skel.py
- Each process sends 0-4 messages to some other processes (see number_of_dests).
- The skeletons include already the Issends of these messages.
- The receiving processes do not know
  - how many messages must be received, and
  - from which processes they will come.
- The skeleton also includes the Iprobe(…) **[please add the arguments]** and the Recv()
- You should add the sender-side part of the nonblocking barrier algorithm presented within this course chapter. Hints:
  - With which one call can you check for the completeness of all nonblocking send requests?
  - MPI_Ibarrier(*comm*, &ib_rq) should be called only once!
  - The MPI_Test(&ib_rq, …) can be done only when MPI_Ibarrier is already called (arguments →)
- Please only fill in the ________ parts. Please do not modify the already given source code.
- mpirun -np 4 ./a.out **| sort +0 -1 +6 -7 +4r -5** (to check whether all messages are received)
- mpirun -np 4 ./a.out **| sort +0 -1 +2 -3 +4r -5 +6 -7** (to sort by processes / snd/rcv / partners)

Exercise 3

Exercise 3 — nonblocking barrier

In MPI/tasks/…

- Use **C** C/Ch6/ibarrier-skel.c or **Fortran** F_30/Ch6/ibarrier-skel_30.f90 or **Python** PY/Ch6/ibarrier-skel.py
- Each process sends 0-4 messages to some other processes (see number_of_dests).
- The skeletons include already the Issends of these messages.
- The receiving processes do not know
  - how many messages must be received, and
  - from which processes they will come.
- The skeleton also includes the Iprobe(…) **[please add the arguments]** and the Recv()
- You should add the sender-side part of the nonblocking barrier algorithm presented within this course chapter. Hints:
  - With which one call can you check for the completeness of all nonblocking send requests?
  - MPI_Ibarrier(*comm*, &ib_rq) should be called only once!
  - The MPI_Test(&ib_rq, …) can be done only when MPI_Ibarrier is already called (arguments →)
- Please only fill in the ________ parts. Please do not modify the already given source code.
- mpirun -np 4 ./a.out **| sort +0 -1 +6 -7 +4r -5** (to check whether all messages are received)
- mpirun -np 4 ./a.out **| sort +0 -1 +2 -3 +4r -5 +6 -7** (to sort by processes / snd/rcv / partners)

Exercise — nonblocking barrier — solutions

In the Ch6/solutions directory, you find

- ibarrier.c / _30.f90 / .py
  - the solution for the ../ibarrier-skel.c / _30.f90 / .py
- ibarrier-optimized.c / _30.f90 / .py
  - an optimized solution that additionally loops over the iprobe & recv
- ibarrier-optimized-test.c / _30.f90 / .py
  - same, but executes only each 10th iprobe & recv
- ibarrier-**wrong**.c, ibarrier-optimized-**wrong**.c, ibarrier-optimized-test-**wrong**.c / _30.f90 / .py
  - All *synchronous* MPI_S**s**end calls are substituted by *standard* MPI_Send.
  - Therefore, the algorithm will start the ibarrier to early.
  - And therefore may stop before all messages are received.
  - Especially the test version shows always wrong results,
    whereas the optimized version may sometimes receive all message by luck.
  - Incorrect programs may produce correct results ☹
    → therefore correct results never prove that the program is correct ☹

Advanced Exercise 4 — MPI_IN_PLACE

- Use **C** C/Ch6/in-place-skel.c or **Fortran** F_30/Ch6/in-place-skel_30.f90
- Your tasks:
  - Substitute the several **`0`** by a **`root`** variable initialized with **`root=0`**, compile and run
  - Substitute **`root=0`** by **`root=num_procs-1`**, compile and run
  - Modify your program that the **`MPI_IN_PLACE`** option is used for **`MPI_Gather`** (read the appropriate paragraph in the MPI description of **`MPI_Gather`**), compile and run

Any significant difference to your solution?