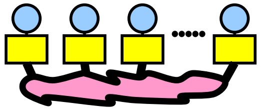
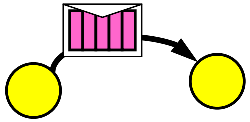
Chap.4 Non-Blocking Communication

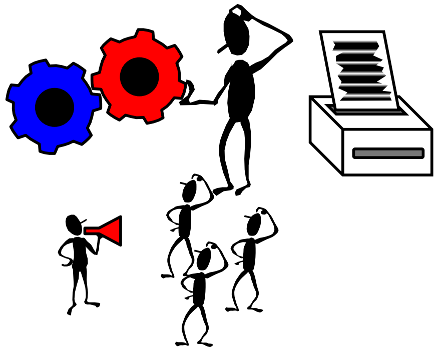
1. MPI Overview 

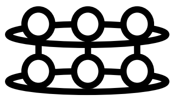
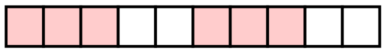
2. Process model and language bindings

   MPI_Init()
   MPI_Comm_rank()

3. Messages and point-to-point communication 

4. **Non-blocking communication**
   - **to avoid idle time and deadlocks**

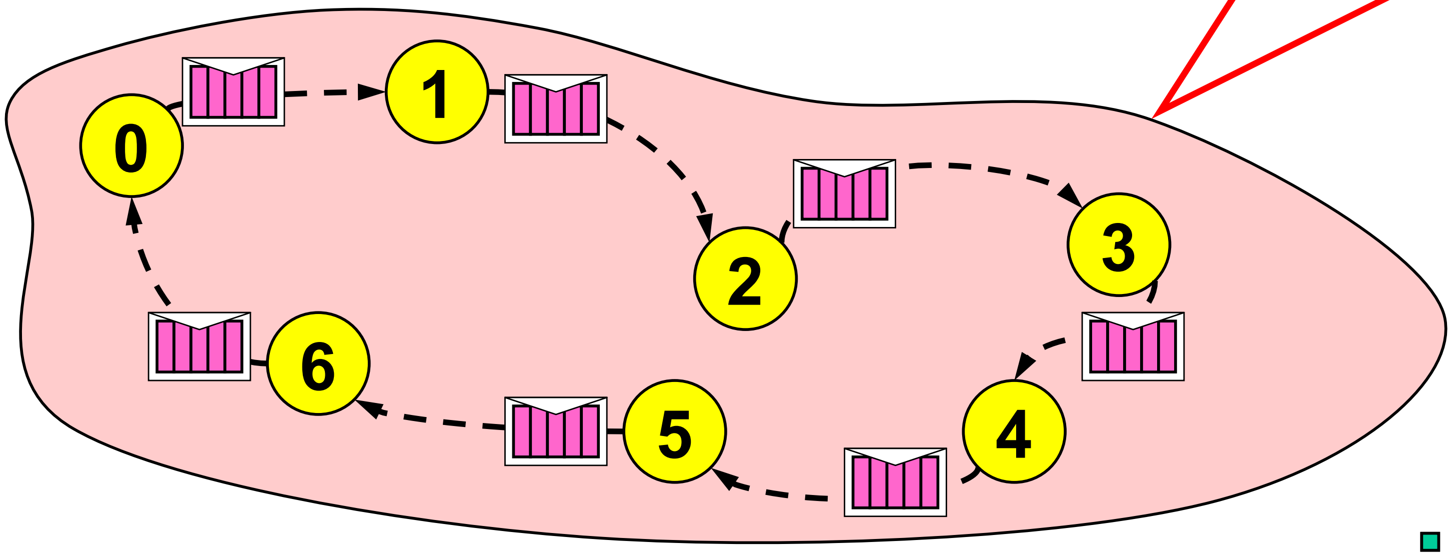


5. Collective communication
6. Virtual topologies 
7. Derived datatypes 
8. Case study

Deadlock

- Code in each MPI process:
  MPI_Ssend(…, right_rank, …)
  MPI_Recv(…, left_rank,   …)

Will block and never return, because MPI_Recv cannot be called in the right-hand MPI process



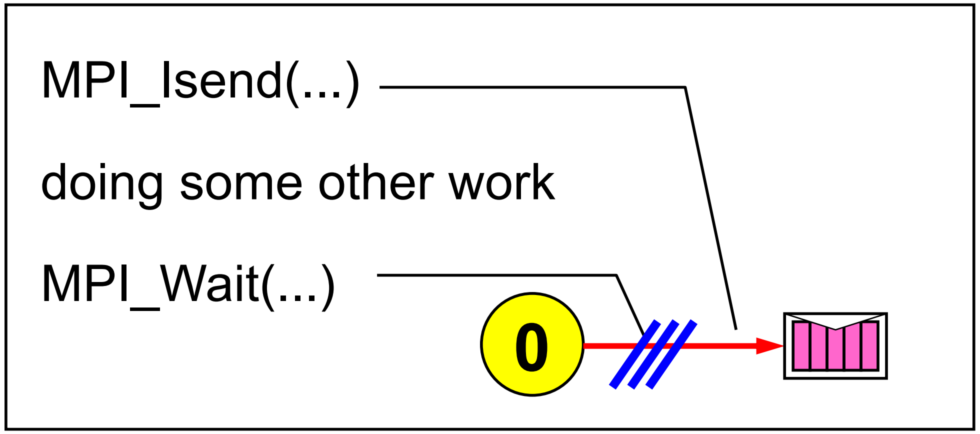
- Same problem with standard send mode (MPI_Send), if MPI implementation chooses synchronous protocol

Non-Blocking Communications

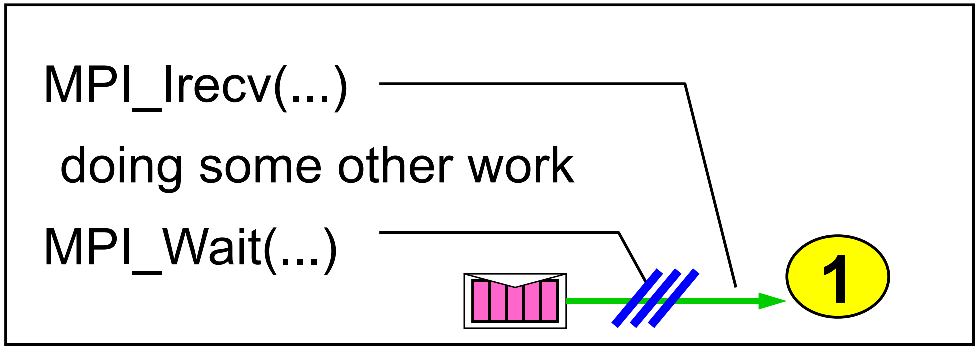
- Separate communication into three phases:
- Initiate non-blocking communication
  - returns **I**mmediately
  - routine name starting with MPI_**I**…
- Do some work
  - “latency hiding”
- Wait for non-blocking communication to complete

Non-Blocking Examples

- Non-blocking **send**



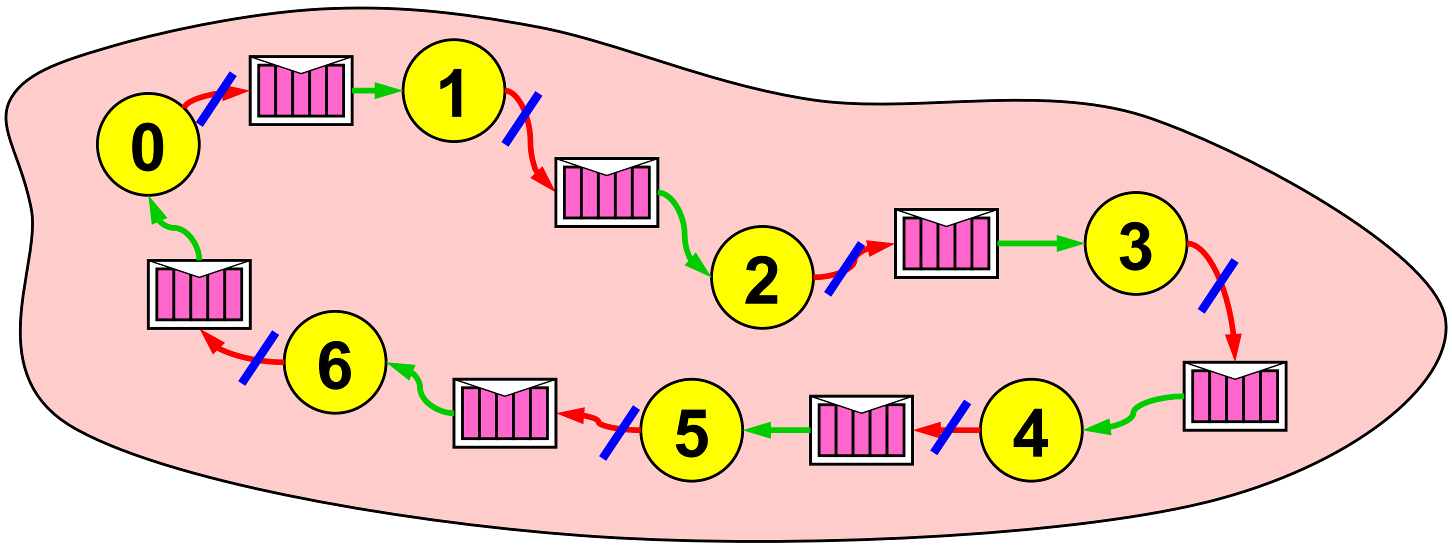
- Non-blocking **receive**



= waiting until operation locally completed

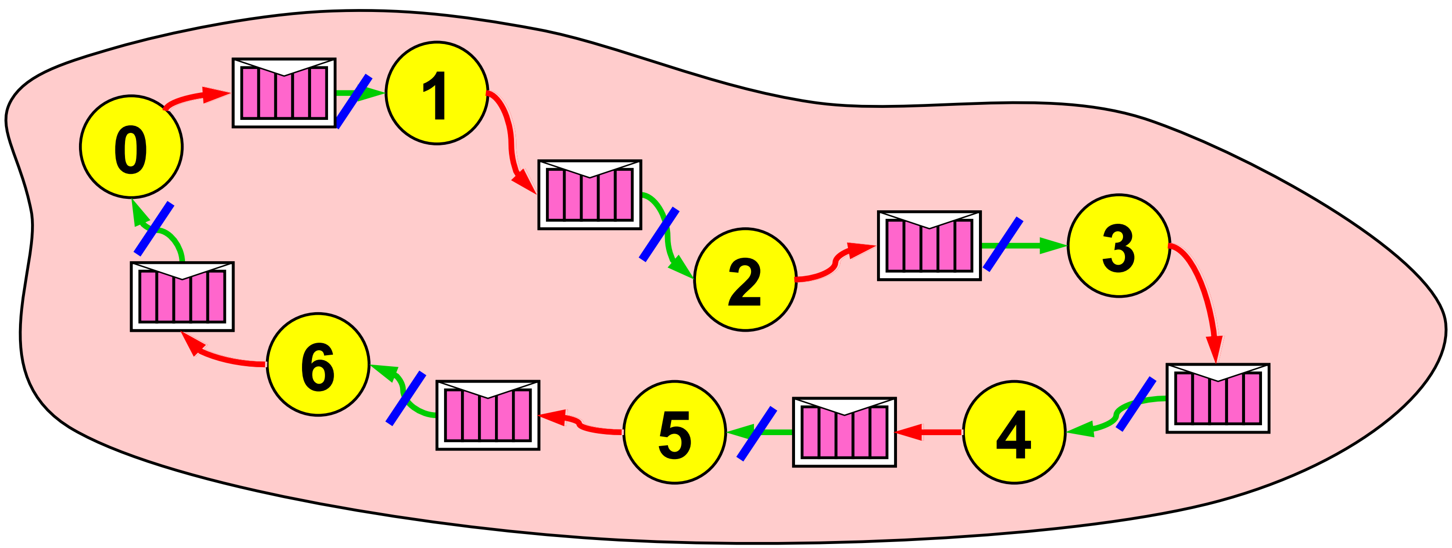
Non-Blocking Send

- Initiate non-blocking send
  in the ring example: Initiate non-blocking send to the right neighbor
- Do some work:
  in the ring example: Receiving the message from left neighbor
- Now, the message transfer can be completed
- Wait for non-blocking send to complete



Non-Blocking Receive

- Initiate non-blocking receive
  in the ring example: Initiate non-blocking receive from left neighbor
- Do some work:
  in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for non-blocking receive to complete



Handles, already known

- Predefined handles
  - defined in mpi.h / mpif.h
  - communicator, e.g., MPI_COMM_WORLD
  - datatype, e.g., MPI_INT, MPI_INTEGER, …
- Handles **can** also be stored in local variables
  - memory for datatype handles
    - in C: MPI_Datatype
    - in Fortran: INTEGER
  - memory for communicator handles
    - in C: MPI_Comm
    - in Fortran: INTEGER

Request Handles

Request handles

- are used for non-blocking communication
- **must** be stored in local variables

  C: MPI_Request

  Fortran: INTEGER

- the value
  - **is generated** by a non-blocking communication routine
  - **is used** (and freed) in the MPI_WAIT routine

Non-blocking Synchronous Send

- C:

  MPI_Issend(buf, count, datatype, dest, tag, comm,
  OUT &*request_handle*);

  MPI_Wait(INOUT &request_handle, &*status*);

- Fortran:

  CALL MPI_ISSEND(buf, count, datatype, dest, tag, comm,
  OUT *request_handle*, *ierror*)

  CALL MPI_WAIT(INOUT request_handle, *status*, *ierror*)

- <u>buf</u> must not be used between <u>Issend</u> and <u>Wait</u> (in all progr. languages)
  MPI 1.1, page 40, lines 44-45
- “<u>Issend</u> + <u>Wait</u> directly after Issend” is equivalent to blocking call (<u>Ssend</u>)
- <u>status</u> is not used in <u>Issend</u>, but in <u>Wait</u> (with send: nothing returned)
- Fortran problems, see MPI-2, Chap. 10.2.2, pp 284-290

Non-blocking Receive

- C:

  MPI_Irecv(*buf*, count, datatype, source, tag, comm,
  OUT &*request_handle*);

  MPI_Wait(INOUT &request_handle, &*status*);

- Fortran:

  CALL MPI_IRECV (*buf*, count, datatype, source, tag, comm,
  OUT *request_handle*, *ierror*)

  CALL MPI_WAIT(INOUT request_handle, *status*, *ierror*)

- buf must not be used between Irecv and Wait (in all progr. languages)

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking.
- A blocking send can be used with a non-blocking receive,
  and vice-versa.
- Non-blocking sends can use any mode
  - standard – MPI_ISEND
  - synchronous – MPI_ISSEND
  - buffered – MPI_IBSEND
  - ready – MPI_IRSEND

Completion

- C:

  MPI_Wait(&request_handle, &*status*);

  MPI_Test(&request_handle, &*flag*, &*status*);

- Fortran:

  CALL MPI_WAIT(request_handle, *status*, *ierror*)

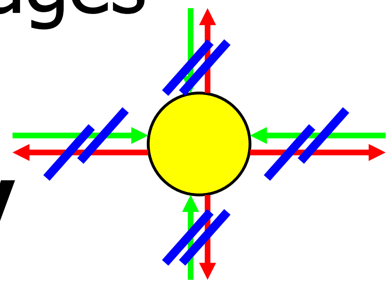
  CALL MPI_TEST(request_handle, *flag*, *status*, *ierror*)

- one must
  - WAIT or
  - loop with TEST until request is completed, i.e., flag == 1 or .TRUE.

Multiple Non-Blocking Communications

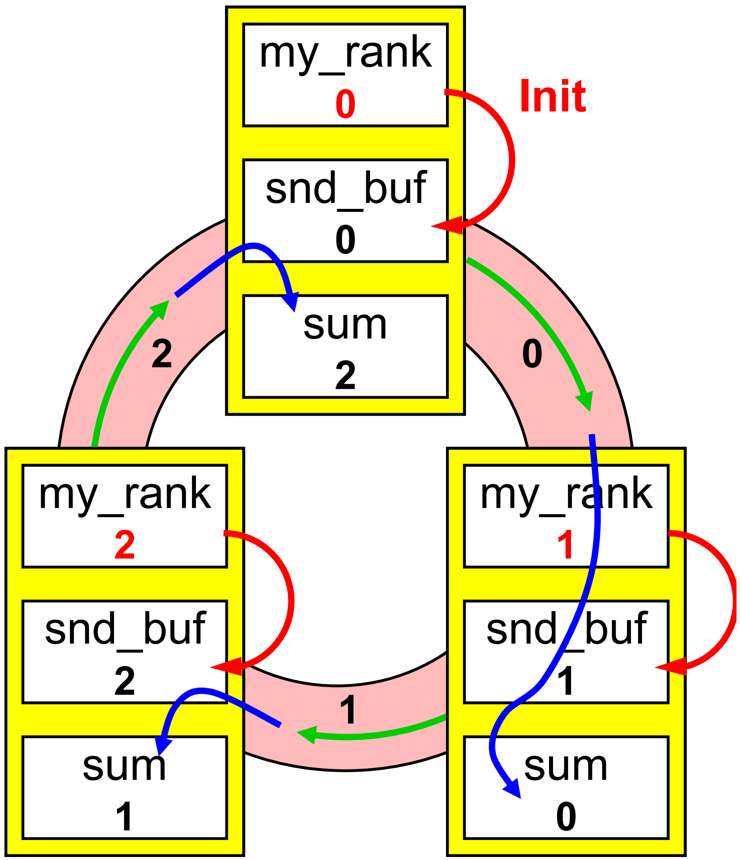
You have several request handles:

- Wait or test for completion of **one** message
  - **MPI_Waitany / MPI_Testany**
- Wait or test for completion of **all** messages
  - **MPI_Waitall / MPI_Testall**
- Wait or test for completion of **as many** messages as possible
  - **MPI_Waitsome / MPI_Testsome**



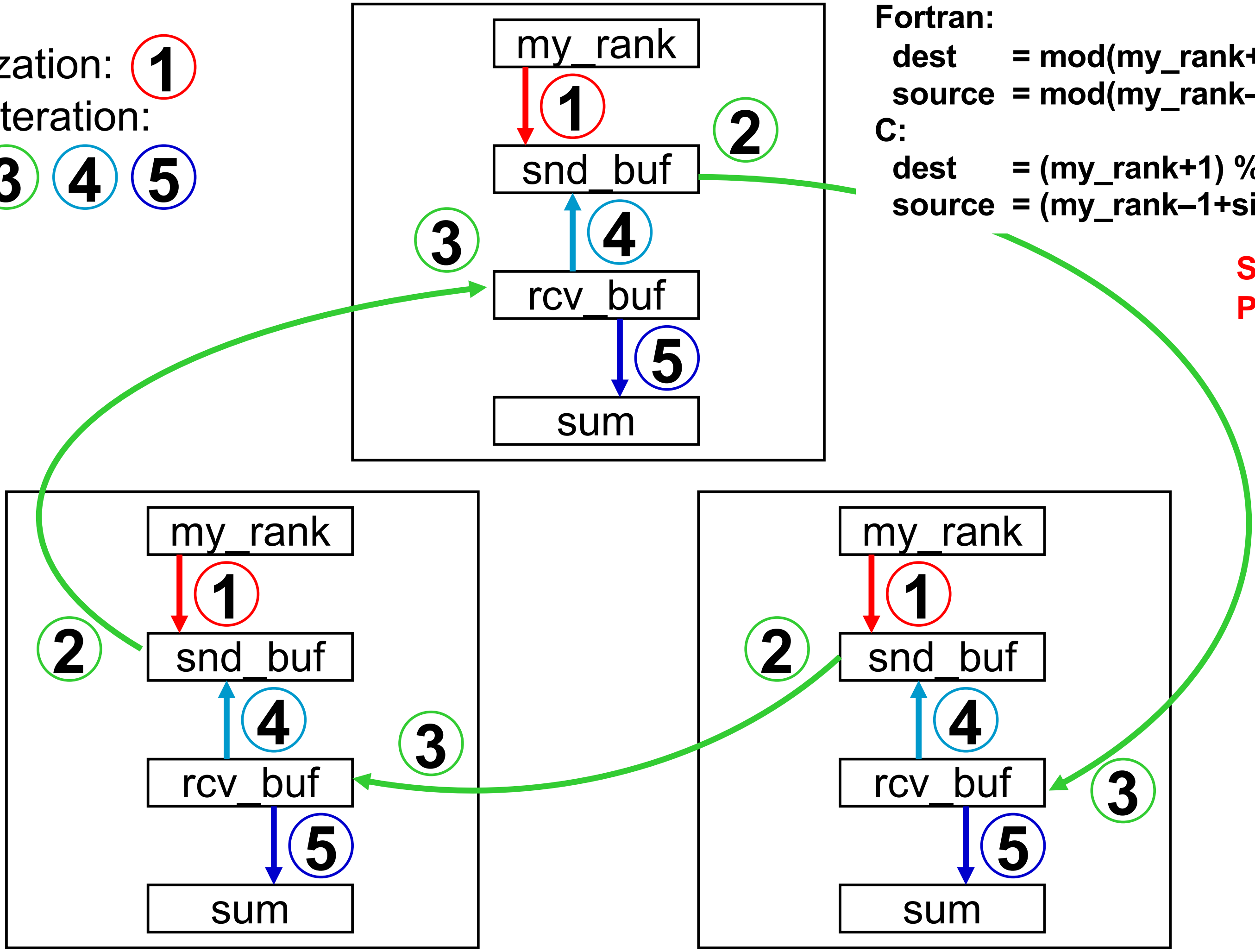
Exercise — Rotating information around a ring

- A set of processes are arranged in a ring.
- Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- Each process passes this on to its neighbor on the right.
- Each processor calculates the sum of all values.
- Keep passing it around the ring until the value is back where it started, i.e.
- each process calculates sum of all ranks.
- Use non-blocking MPI_Issend
  - to avoid deadlocks
  - to verify the correctness, because blocking synchronous send will cause a deadlock



Initialization: 1

Each iteration: 2 3 4 5



Fortran:

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

C:

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

Single Program !!!

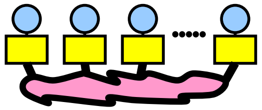
Advanced Exercises — Irecv instead of Issend

- Substitute the Issend–Recv–Wait method by the Irecv–Ssend–Wait method in your ring program.

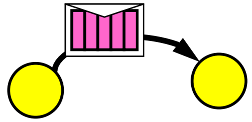
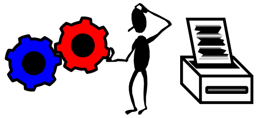
- Or

- Substitute the Issend–Recv–Wait method by the Irecv–Issend–Waitall method in your ring program.

Chap.5 Collective Communication

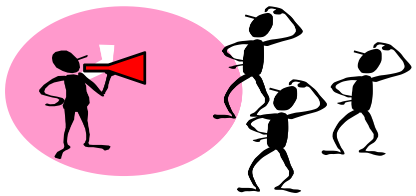
1. MPI Overview 
2. Process model and language bindings

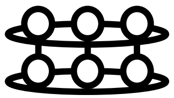
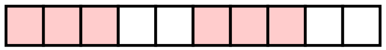
```
MPI_Init()
MPI_Comm_rank()
```

3. Messages and point-to-point communication 
4. Non-blocking communication 

5. Collective communication

e.g.,broadcast



6. Virtual topologies 
7. Derived datatypes 
8. Case study

Collective Communication

- Communications involving a group of processes.
- Must be called by all processes in a communicator.
- Examples:
  – Barrier synchronization.
  – Broadcast, scatter, gather.
  – Global sum, global maximum, etc.

Characteristics of Collective Communication

- Optimised Communication routines involving a group of processes
- Collective action over a communicator, i.e. all processes must call the collective routine.
- Synchronization may or may not occur.
- All collective operations are blocking.
- No tags.
- Receive buffers must have exactly the same size as send buffers.

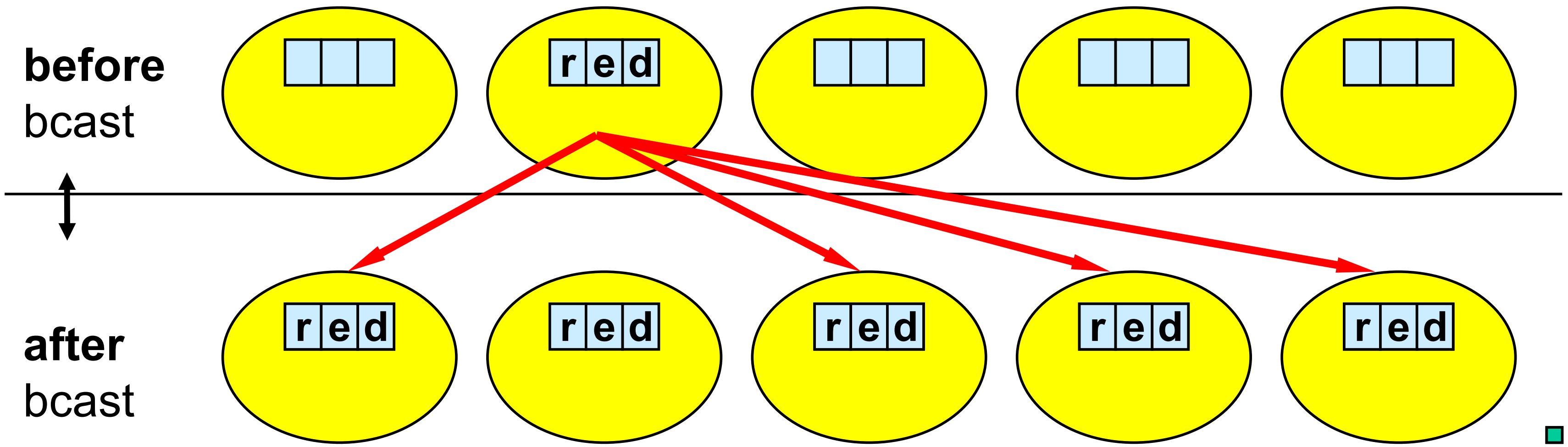
Barrier Synchronization

- C: int MPI_Barrier(MPI_Comm comm)
- Fortran: MPI_BARRIER(COMM, *IERROR*)
  INTEGER COMM, IERROR
- 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.

Broadcast

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

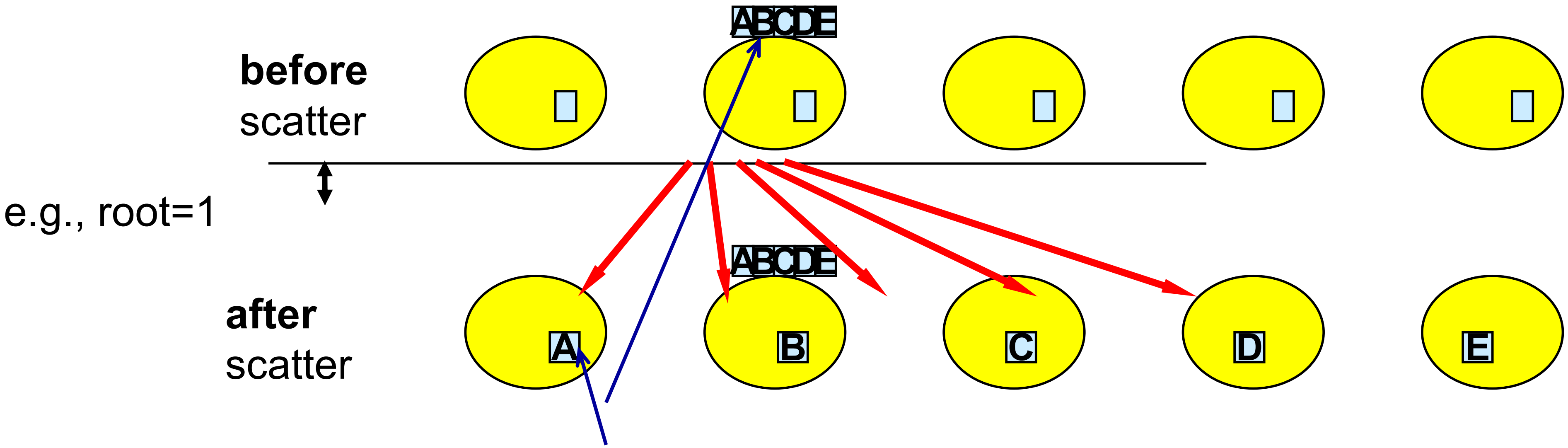
- Fortran:
```
MPI_Bcast(BUF, COUNT, DATATYPE, ROOT, COMM, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, ROOT
INTEGER COMM, IERROR
```



e.g., root=1

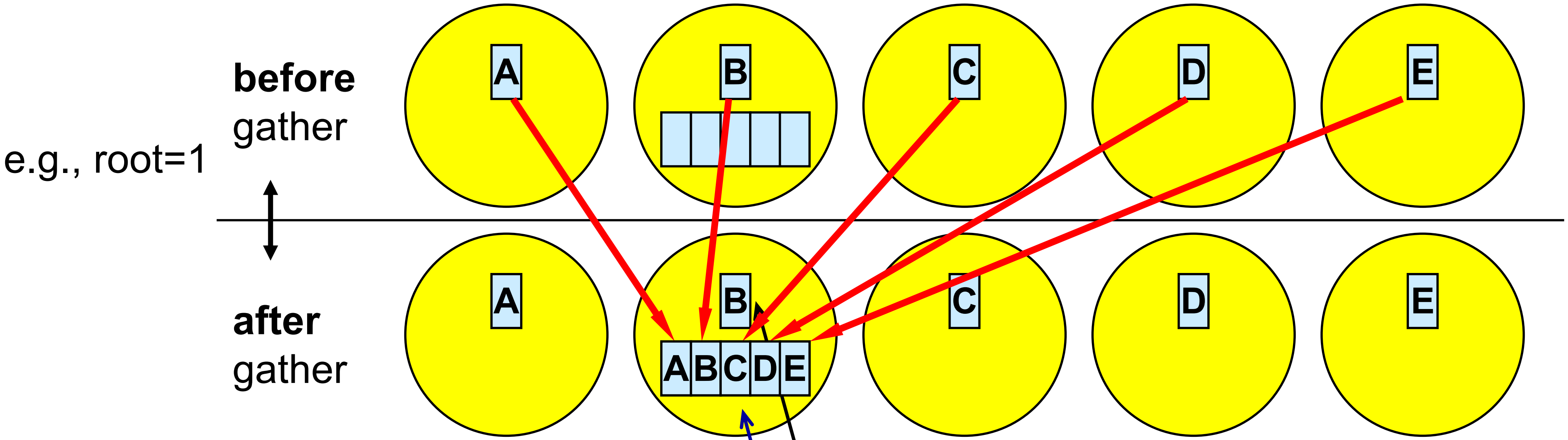
- **rank of the sending process (i.e., root process)**
- **must be given identically by all processes**

Scatter



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

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)
    <type>      SENDBUF(*), RECVBUF(*)
    INTEGER     SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE
INTEGER         ROOT, COMM, IERROR
```

Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- $d_0 \ \mathbf{o}\ d_1 \ \mathbf{o}\ d_2 \ \mathbf{o}\ d_3 \ \mathbf{o}\ \ldots \ \mathbf{o}\ d_{s-2} \ \mathbf{o}\ d_{s-1}$
  - d_i = data in process rank i
    - single variable, or
    - vector
  - $\mathbf{o}$ = 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:
  - $[(d_0 \ \mathbf{o}\ d_1) \ \mathbf{o}\ (d_2 \ \mathbf{o}\ d_3)] \ \mathbf{o}\ [\ldots \ \mathbf{o}\ (d_{s-2} \ \mathbf{o}\ d_{s-1})]$
  - $((((((d_0 \ \mathbf{o}\ d_1) \ \mathbf{o}\ d_2) \ \mathbf{o}\ d_3) \ \mathbf{o}\ \ldots\ ) \ \mathbf{o}\ d_{s-2}) \ \mathbf{o}\ d_{s-1})$

MPI

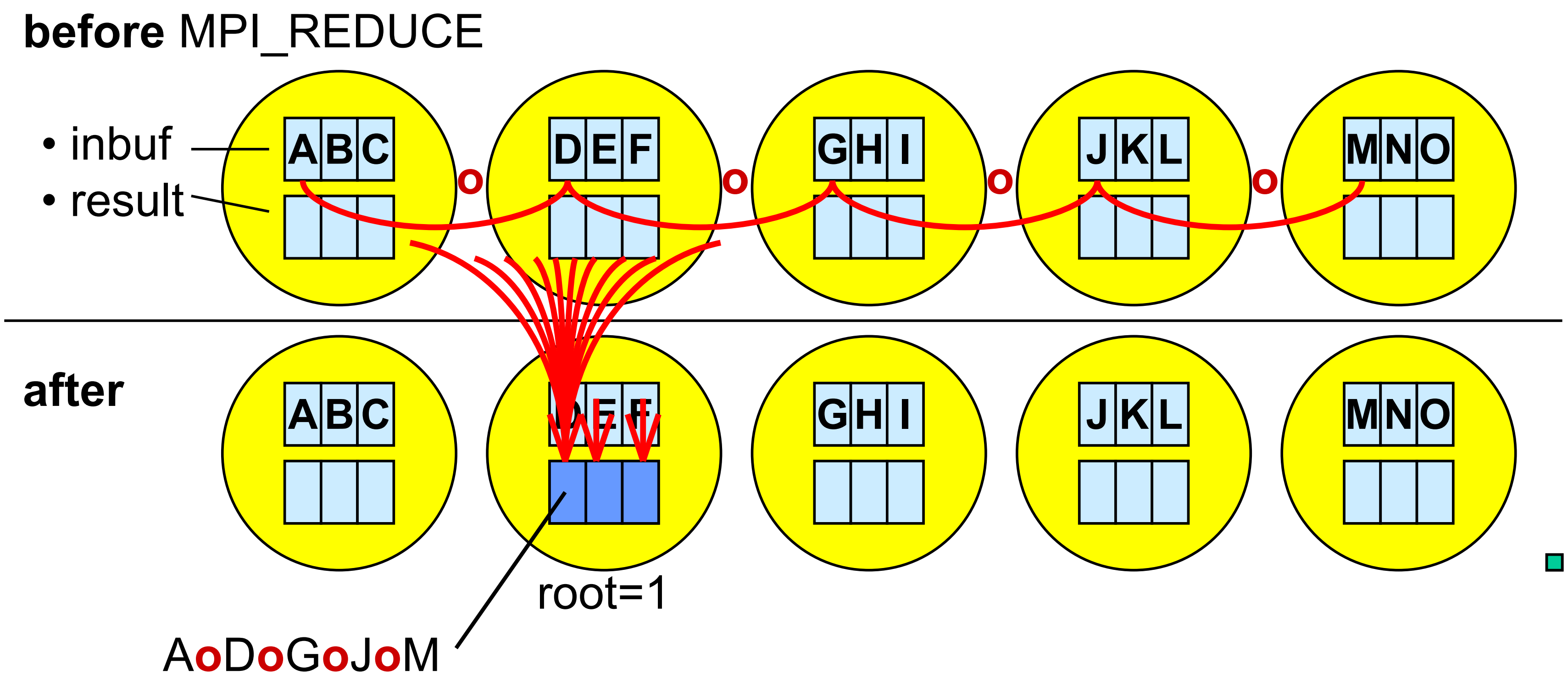
Example of Global Reduction

- Global integer sum.
- Sum of all inbuf values should be returned in *resultbuf.*
- C: root=0;
MPI_Reduce(&inbuf, &*resultbuf*, 1, MPI_INT,
MPI_SUM,root, MPI_COMM_WORLD);
- Fortran: root=0
MPI_REDUCE(inbuf, *resultbuf*, 1, MPI_INTEGER,
MPI_SUM, root, MPI_COMM_WORLD, *IERROR*)
- The result is only placed in *resultbuf* at the root process.

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
• inbuf
• result
A B C
o
D E F
o
G H I
o
J K L
o
M N 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-1 standard
- Registering a user-defined reduction function:
  - C: MPI_Op_create(MPI_User_function *func, int commute, MPI_Op **op*)
  - Fortran: MPI_OP_CREATE(FUNC, COMMUTE, *OP*, *IERROR*)
- COMMUTE tells the MPI library whether FUNC is commutative.

Example

```
typedef struct {
double real, imag;
}Complex
```

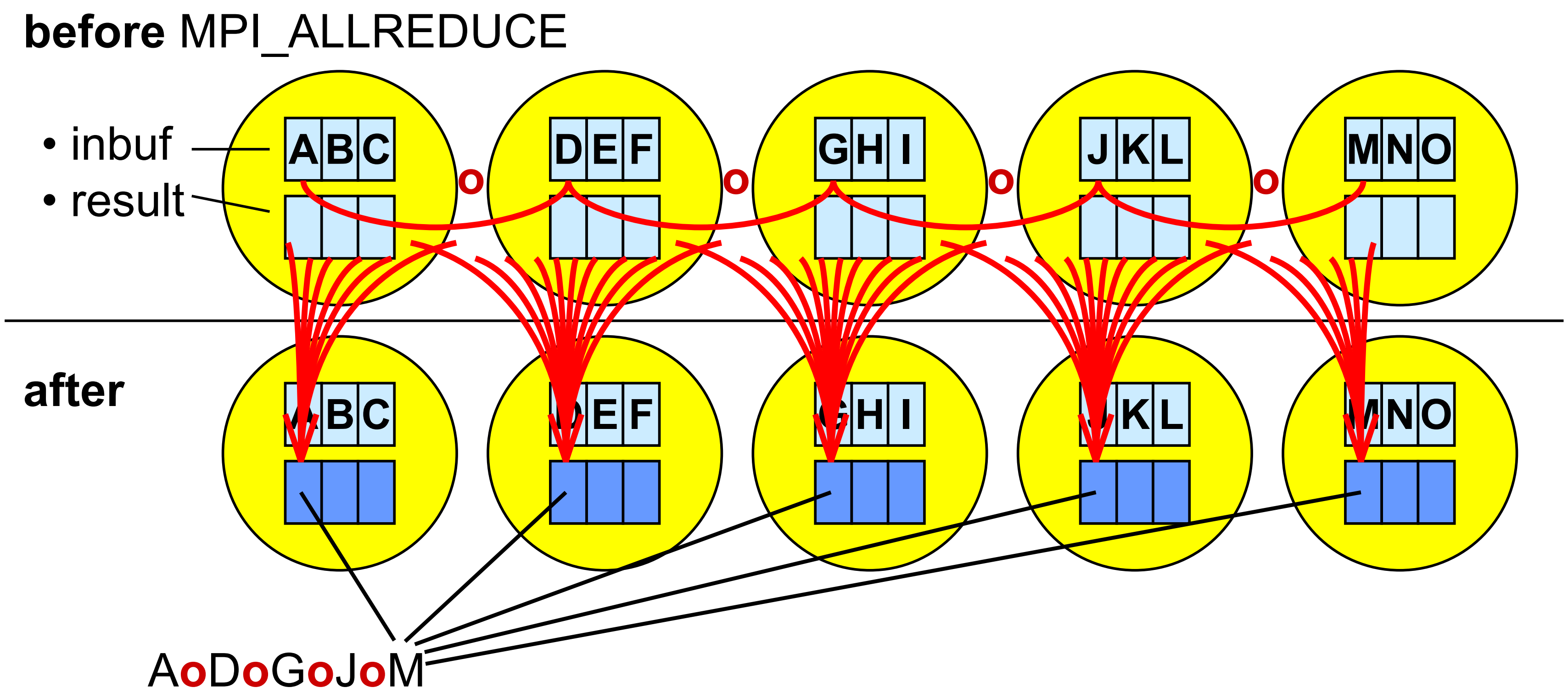
- **Complex** a[100], answer[100];
- MPI_Op **myOp**
- MPI_Datatype ctype;
- MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
- MPI_Type_commit(&ctype);
- MPI_Op_create(**myProd** , True, & **myOp**);
- MPI_Reduce(a, answer, 100, ctype, **myOp** root, comm);

```
void myProd ( Complex *in,
    Complex *inout, int *len,
    MPI_Datatype *dptr )
    {
        int i;
        Complex c;
        for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real -
        inout->imag*in->imag;
        c.imag = inout->real*in->imag +
        inout->imag*in->real;
        *inout = c;
        in++; inout++;
        }
    }
```

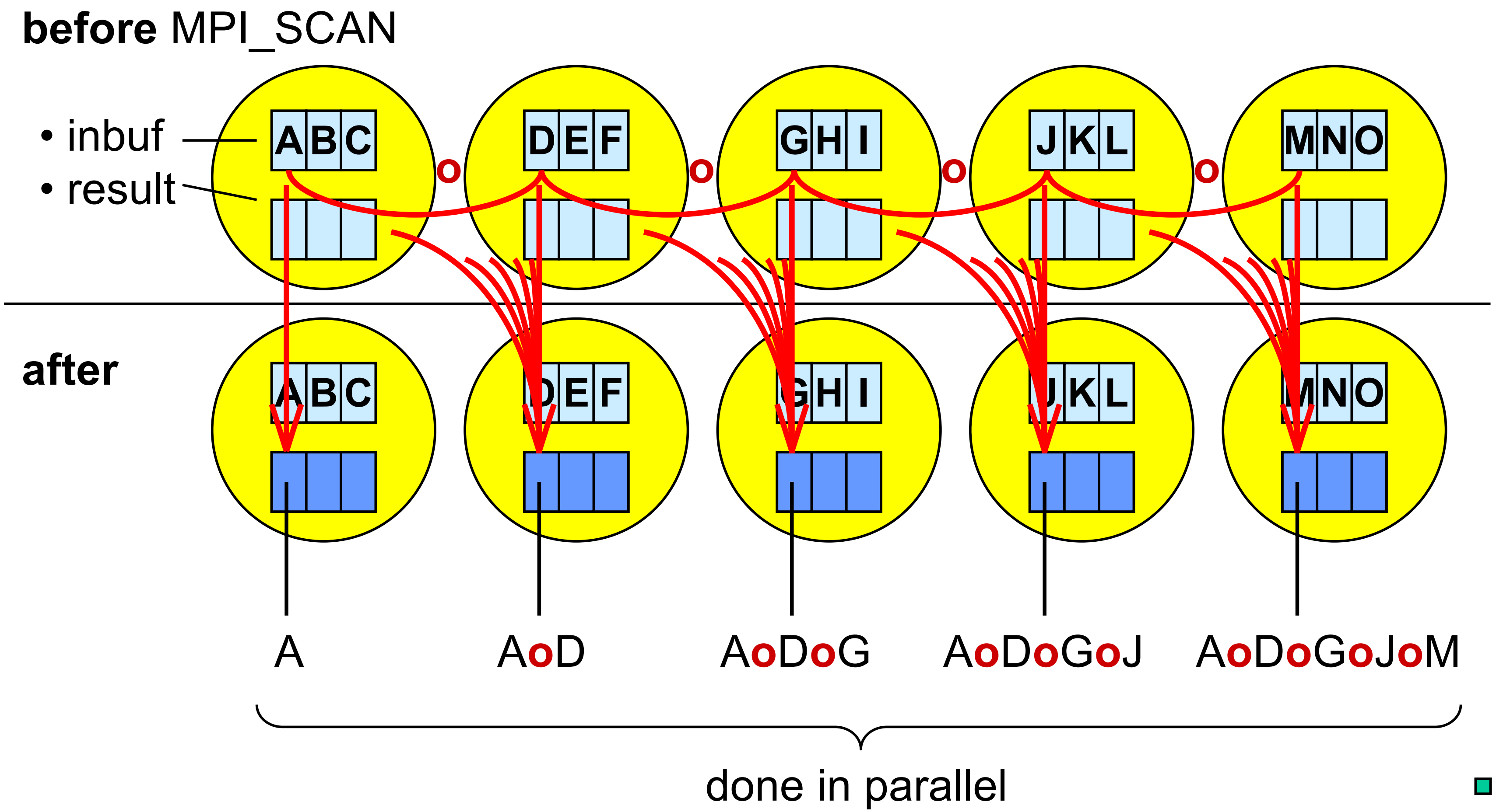
Variants of Reduction Operations

- MPI_ALLREDUCE
  - no root,
  - returns the result in all processes
- 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_ALLREDUCE



MPI_SCAN


before MPI_SCAN
• inbuf
• result
ABC
DEF
GHI
JKL
MNO
o
o
o
o
after
ABC
DEF
GHI
JKL
MNO
A
AoD
AoDoG
AoDoGoJ
AoDoGoJoM
done in parallel

Exercise — 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.
- Use the results from Chap. 4:
  ~course00/MPI-I/examples/fortran/ring.f
  or
  ~course00/MPI-I/examples/c/ring.c
- I.e., the pass-around-the-ring communication loop must be totally substituted by one call to the MPI collective reduction routine.

Advanced Exercises — Global scan and sub-groups

- Global scan:
  - Rewrite the last program so that each process computes a partial sum.
  - Rewrite in a way that each process prints out its partial result in the correct order:

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