

GWDG – Kurs  
Parallel Programming with MPI

# MPI

## Collective Operations

Oswald Haan  
oahan@gwdg.de

# Collective Operations

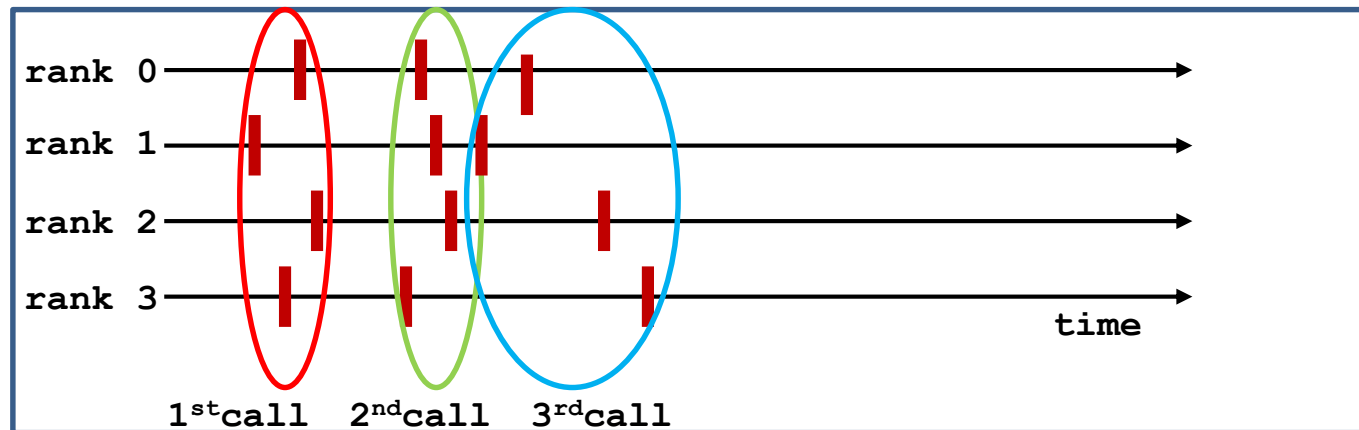
Collective operations involve all processes in a communicator.

Types of collective operations:

- Synchronization : MPI\_BARRIER
- Communication : MPI\_BCAST, MPI\_GATHER, MPI\_SCATTER
- Reduction : MPI\_REDUCE

# Characteristics of Collective Operations

- All processes of a communicator must participate, i.e. must call the collective routine.
- On a given communicator, the n-th collective call must match on all processes of the communicator. Therefore, no tags needed for collective operations.
- If one or more processes of a communicator do not participate in a given collective operation, the program will hang.



- In MPI-1.0 – MPI-2.2, all collective operations are blocking.
- Non-blocking versions since MPI-3.0.
- buffers on all processes must have exactly the same size.

# Classification of Collective Operations

**MPI\_BARRIER:**

Synchronisation

**MPI\_BCAST:**

Send from one process to all processes

**MPI\_GATHER:**

gather data from all processes on one process

**MPI\_SCATTER:**

scatter data from one process to all processes

**MPI\_ALLGATHER:**

gather data from all processes, broadcast them to all processes

**MPI\_ALLTOALL:**

exchange data between all processes

**MPI\_REDUCE:**

reduction over all processes, result goes to one process

**MPI\_ALLREDUCE:**

reduction over all processes, result is broadcasted to all processes

**MPI\_REDUCE\_SCATTER:**

reduction over all processes, result is scattered to all processes

**MPI\_SCAN, MPI\_EXSCAN:**

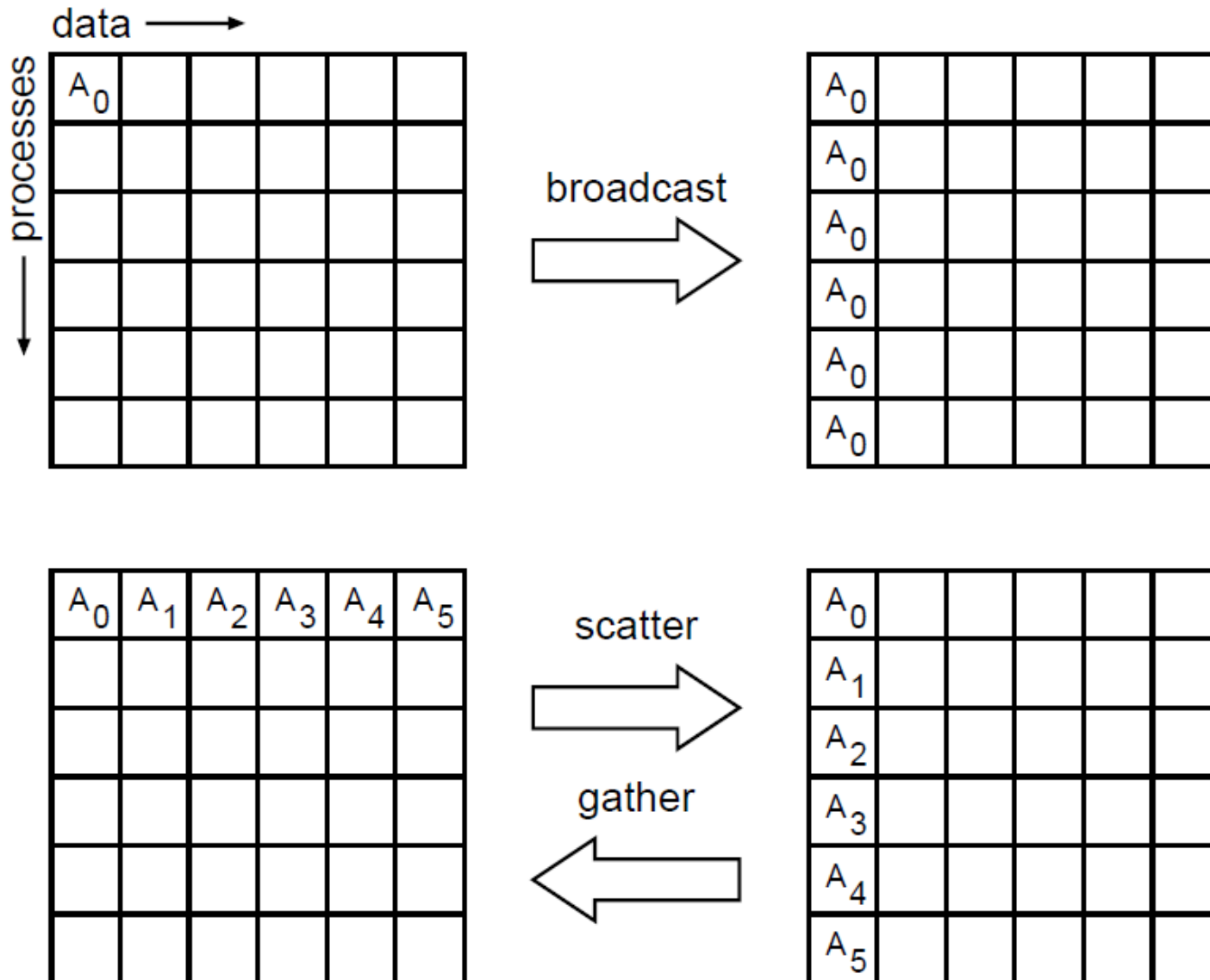
process  $i$  receives result from reduction over processes with  $j \leq i$ ,  $j < i$

one → all

all → one

all → all

# Data Flow in Collective Communication



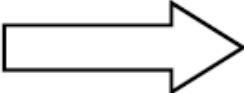
# Kollektive Operationen: Datenfluss

data →

processes ↓

A <sub>0</sub>					
B <sub>0</sub>					
C <sub>0</sub>					
D <sub>0</sub>					
E <sub>0</sub>					
F <sub>0</sub>					

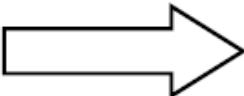
allgather



A <sub>0</sub>	B <sub>0</sub>	C <sub>0</sub>	D <sub>0</sub>	E <sub>0</sub>	F <sub>0</sub>
A <sub>0</sub>	B <sub>0</sub>	C <sub>0</sub>	D <sub>0</sub>	E <sub>0</sub>	F <sub>0</sub>
A <sub>0</sub>	B <sub>0</sub>	C <sub>0</sub>	D <sub>0</sub>	E <sub>0</sub>	F <sub>0</sub>
A <sub>0</sub>	B <sub>0</sub>	C <sub>0</sub>	D <sub>0</sub>	E <sub>0</sub>	F <sub>0</sub>
A <sub>0</sub>	B <sub>0</sub>	C <sub>0</sub>	D <sub>0</sub>	E <sub>0</sub>	F <sub>0</sub>
A <sub>0</sub>	B <sub>0</sub>	C <sub>0</sub>	D <sub>0</sub>	E <sub>0</sub>	F <sub>0</sub>

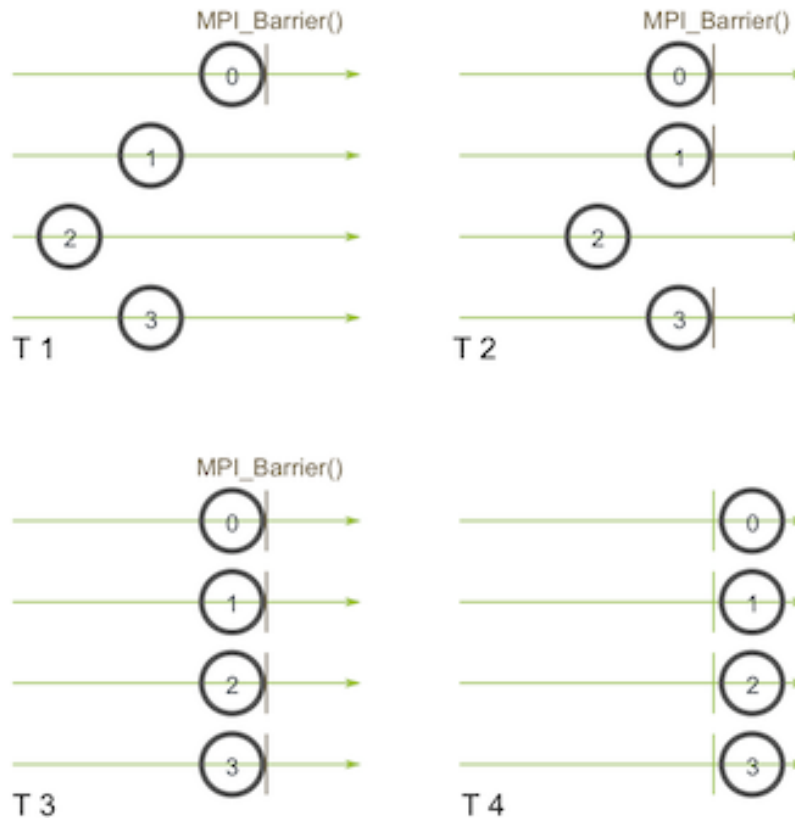
A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>
B <sub>0</sub>	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>
C <sub>0</sub>	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>	C <sub>4</sub>	C <sub>5</sub>
D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>
E <sub>0</sub>	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>
F <sub>0</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>	F <sub>4</sub>	F <sub>5</sub>

alltoall  
complete  
exchange



A <sub>0</sub>	B <sub>0</sub>	C <sub>0</sub>	D <sub>0</sub>	E <sub>0</sub>	F <sub>0</sub>
A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	D <sub>1</sub>	E <sub>1</sub>	F <sub>1</sub>
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	D <sub>2</sub>	E <sub>2</sub>	F <sub>2</sub>
A <sub>3</sub>	B <sub>3</sub>	C <sub>3</sub>	D <sub>3</sub>	E <sub>3</sub>	F <sub>3</sub>
A <sub>4</sub>	B <sub>4</sub>	C <sub>4</sub>	D <sub>4</sub>	E <sub>4</sub>	F <sub>4</sub>
A <sub>5</sub>	B <sub>5</sub>	C <sub>5</sub>	D <sub>5</sub>	E <sub>5</sub>	F <sub>5</sub>

# MPI\_BARRIER: Synchronisation



Quelle: <http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication>

# MPI\_BARRIER: Synchronisation

C: `MPI_Barrier( MPI_Comm comm )`

FORTRAN: `MPI_BARRIER( comm, ierror )`  
`INTEGER comm, ierror`

mpi4py: `comm.Barrier( )`

- MPI\_BARRIER is usually not needed, because synchronization will be effected by other MPI routines
- MPI\_BARRIER is useful for debugging and timing purposes



# MPI\_BCAST: Broadcast from **root**

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

FORTRAN: `MPI_BCAST( buf, count, datatype, root, comm, ierror )  
<type>buf(*), INTEGER count, datatype, root, comm, ierror`

mpi4py: `robj = comm.bcast(sobj, root= 0)  
comm.Bcast(ar, root= 0)`

- All processes must supply the same values for **root** and **comm**
- **data\_size = count \* size (datatype)**  
must be the same on all processes

before MPI\_BCAST

after MPI\_BCAST



# MPI\_SCATTER: Scatter from root

C: `MPI_Scatter( void *sbuf, int scount, MPI_Type stype  
                  , void *rbuf, int rcount, MPI_Type rtype  
                  , int root, MPI_Comm comm )`

FORTRAN: `MPI_SCATTER( sbuf, scount, stype, rbuf, rcount, rtype  
                          , root, comm, ierror )`

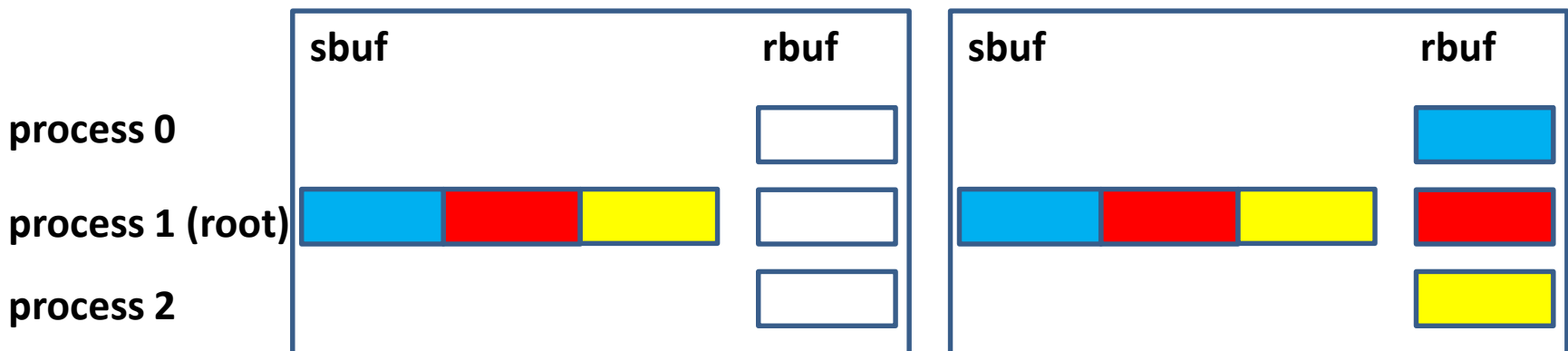
`<type>sbuf(*), rbuf(*)`

`INTEGER scount, stype, rcount, rtype, comm, ierror`

mpi4py: `robject = comm.scatter(sendobj = sobj, recvobj=None, root= 0)  
comm.Scatter(sar, rar, root= 0)`

before MPI\_SCATTER

after MPI\_SCATTER



# Restrictions for Arguments in MPI\_SCATTER

- All processes must supply the same values for **root** and **comm**
- **r\_data\_size = rcount\*size(rtype)** on all processes  
must be equal to  
**s\_data\_size = scount\*size(stype)** on process **root**
- **sbuf** is ignored on all non-**root** processes
- The total size of data scattered from process **root** is  
**nproc \* s\_data\_size**

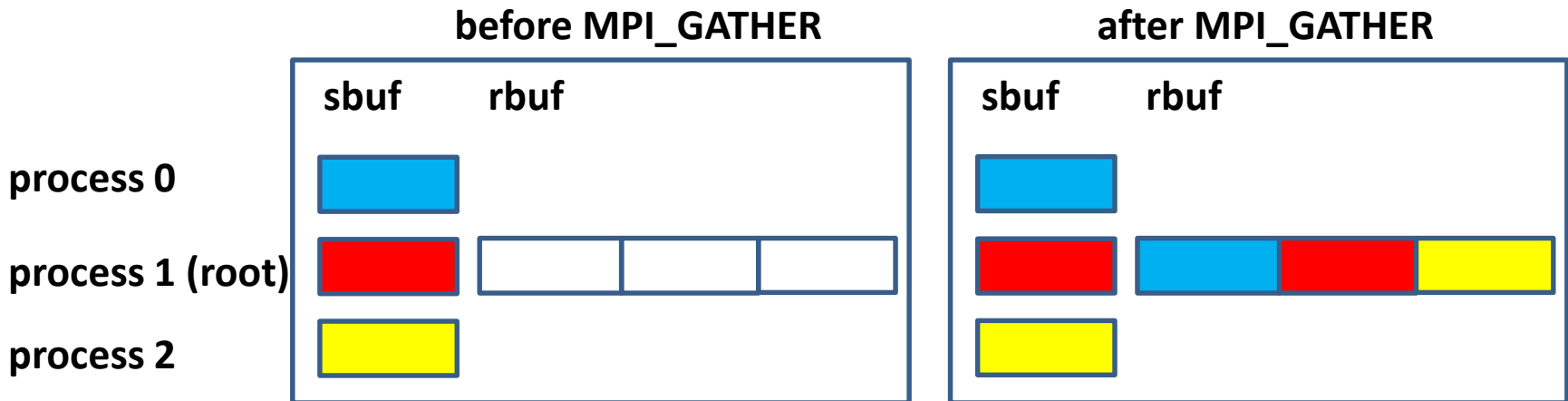
# MPI\_GATHER: Gather to root

C: `MPI_Gather( void *sbuf, int scount, MPI_Type stype  
              , void *rbuf, int rcount, MPI_Type rtype  
              , int root, MPI_Comm comm )`

Fortran: `MPI_GATHER( sbuf, scount, stype, rbuf, rcount, rtype  
                      , root, comm, ierror )`  
`<type>sbuf(*), rbuf(*)`

`INTEGER scount, stype, rcount, rtype, comm, ierror`

mpi4py: `robject = comm.gather(sendobj = sobj, recvobj=None, root= 0)  
          comm.Gather(sar, rar, root= 0)`



# Restrictions for Arguments in MPI\_GATHER

- All processes must supply the same values for **root** and **comm**
- **s\_data\_size = scount\*size(stype)** on all processes must be equal to  
**r\_data\_size = rcount\*size(rtype)** on process **root**
- **rbuf** is ignored on all non-**root** processes
- The total size of data gathered on process **root** is  
**nproc \* r\_data\_size**

# Other Collective Communication Routines

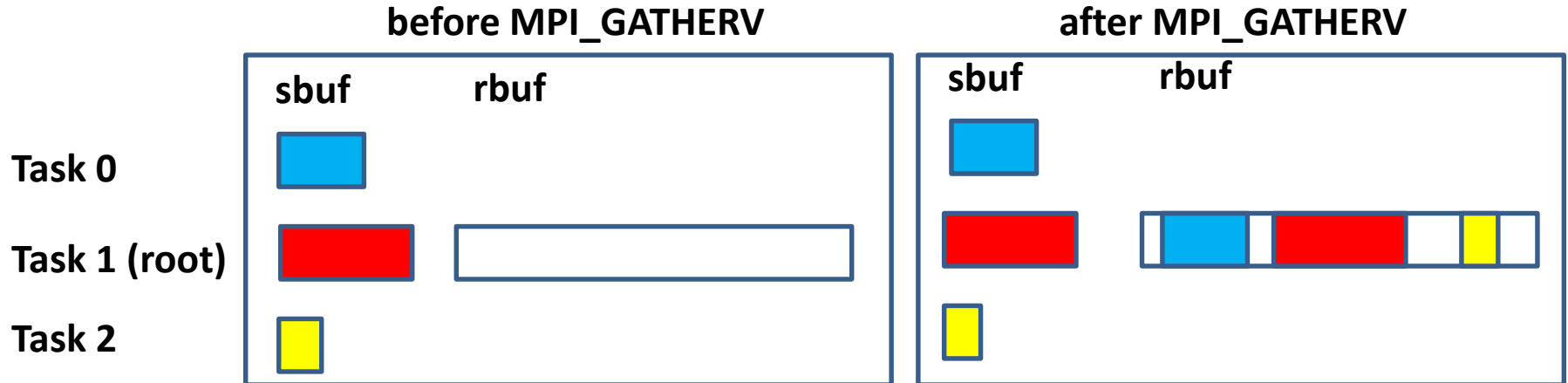
- MPI\_ALLGATHER similar to MPI\_GATHER,
  - but all processes receive the result vector
  - therefore no **root** argument
- MPI\_ALLTOALL
  - each process sends messages to all processes
- MPI\_GATHERV, \_SCATTERV, \_ALLGATHERV, \_ALLTOALLV, \_ALLTOALLW
  - Vector variants of collective communication routines
  - The counts of elements is different for each process,
  - different displacements of the element to be scattered from the send buffer resp. different displacements of the elements to be gathered in the receive buffer can be prescribed
  - Identical **array of counts** and **array of displacements** must be given as arguments in the call on all processes

# MPI\_GATHERV : Gather to root

```
C: MPI_Gatherv( void *sbuf, int scount, MPI_Type stype
               , void *rbuf, int *rcounts, int *displs, MPI_Type rtype
               , int root, MPI_Comm comm )
```

```
Fortran: MPI_GATHERV( sbuf, scount, stype
                    , rbuf, rcounts, displs, rtype, root, comm, ierr )
<type>sbuf(*), rbuf(*)
INTEGER scount, stype, rcounts(*), displs(*), rtype, comm, ierr
```

```
mpi4py: comm.Gatherv(sar, rar, root= 0)
rar = [recvdata,rcounts,dspls,dtype]
```



# Restrictions for Arguments in MPI\_GATHERV

- The number of bytes in **sbuf** send from task **i**, determined by  $\mathbf{scount} * \mathbf{size}(\mathbf{stype})$ , must be equal to the number of bytes received in the **i**-th block in **rbuf** on **root**, determined by  $\mathbf{rcounts}(i) * \mathbf{size}(\mathbf{rtype})$
- The data block **sbuf** from task **i** will be stored in **rbuf** on **root** with displacement of **displs(i)** elements of type **rtype** from address **rbuf**.
- **rbuf**, **rcounts**, **displs** will be ignored on all **non-root** tasks .

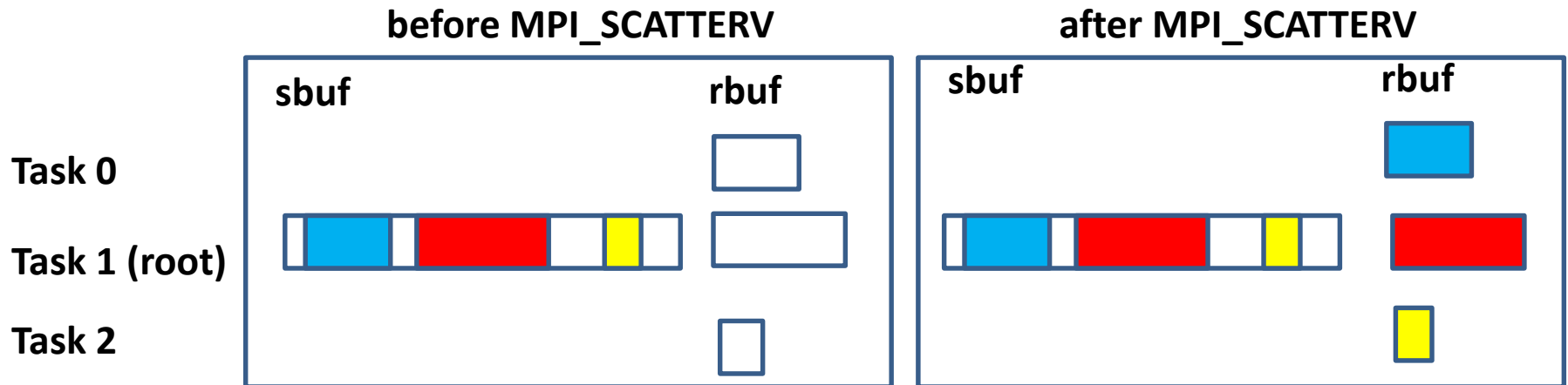


# MPI\_SCATTERV : Scatter from root

C: `MPI_Scatterv( void *sbuf, int *scounts, int *displs, MPI_Type stype  
                  , void *rbuf, int rcount, MPI_Type rtype  
                  , int root, MPI_Comm comm )`

Fortran: `MPI_SCATTERV( sbuf, scounts, displs, stype  
                          , rbuf, rcount, rtype, root, comm, ierr )`  
`<type>sbuf(*) , rbuf(*)`  
`INTEGER scounts(*), displs(*), stype, rcount, rtype, comm, ierr`

mpi4py: `comm.Scatterv(sar, rar, root= 0)`  
`sar = [senddata,scounts,dspls,dtype]`



# Restrictions for Arguments in MPI\_SCATTERV

- The number of bytes in **rbuf** received on task **i**, determined by  $\mathbf{rcount} * \mathbf{size}(\mathbf{rtype})$ , must be equal to the number of bytes sent from the **i**-th block in **sbuf** on **root**, determined by  $\mathbf{scounts}(i) * \mathbf{size}(\mathbf{stype})$
- The **i**-th data block in **sbuf** has  $\mathbf{scounts}(i)$  elements of type **stype**, is located at a distance of  $\mathbf{scounts}(i)$  elements from the start address of **sbuf** and will be stored in the receive buffer **rbuf** on process **i** as **rcount** elements of type **rtype**
- **sbuf**, **scounts**, **displs** will be ignored on all **non-root** tasks .

# In Place Variants

- In place variant of MPI\_GATHER
  - The value MPI\_IN\_PLACE can be provided as argument for **sbuf** in the root process, if the root data to be gathered are already on their place in **rbuf** of the root
- In place variant of MPI\_ALLGATHER
  - The value MPI\_IN\_PLACE can be provided as argument for **sbuf** in all processes root process, if the data to be gathered from a process are already on their place in the **rbuf** of this process.

# MPI\_GATHER: mit MPI\_IN\_PLACE auf root

## nicht-root-Task

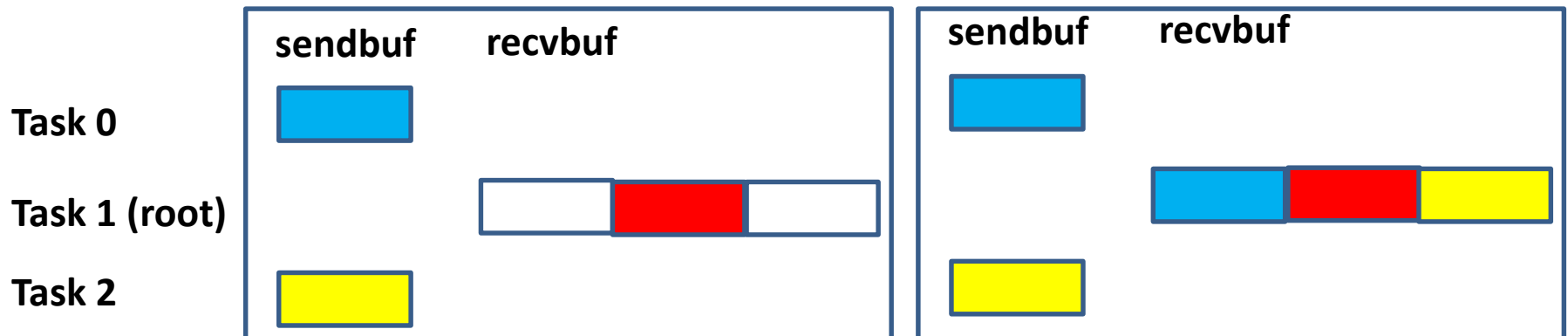
MPI\_GATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount,  
recvtype, root, comm)

## root-Task

MPI\_GATHER(MPI\_IN\_PLACE, sendcount, sendtype, recvbuf, recvcount,  
recvtype, root, comm)

Vor MPI\_GATHER

Nach MPI\_GATHER



# MPI\_ALLGATHER mit MPI\_IN\_PLACE

MPI\_ALLGATHER(MPI\_IN\_PLACE, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

Vor MPI\_ALLGATHER

Nach MPI\_ALLGATHER



# Global Reduction

An example:

Add the results `local_res` computed in each of 3 tasks to a total result:

```
total_res = local_res_0 + local_res_1 + local_res_2
```

```
INTEGER loc_res, total_res, all_res(3)
```

! Gather local results into the array `all_res` on the root process with `MPI_GATHER`

```
MPI_GATHER( loc_res, 1, MPI_INTEGER  
           , all_res, 1, MPI_INTEGER  
           , root , comm, ierror )
```

! Add the elements of `all_res` on the root process

```
if (myid.eq.root) then  
    total_res = all_res(1)+all_res(2)+all_res(3)  
end if
```

# Global Reduction with MPI\_REDUCE

```
INTEGER loc_res, total_res  
call MPI_REDUCE( loc_res, total_res, 1, MPI_INTEGER  
                , MPI_SUM  
                , root , comm, ierror )
```

# Global Reduction Operations

To perform a global reduce operation across data on all processes of a communicator:

$\text{redo} = d_0 \circ d_1 \circ d_2 \circ d_3 \circ \dots$

–  $d_i$  = data in process rank  $i$

- **single variable, or**
- **vector, i.e.  $d_i = (d_{i1}, d_{i2}, d_{i3}, \dots)$**

–  $\circ$  = associative operation

If  $d_i$  are vectors, the result of the reduce operation also is a vector:

$\text{redo} = ( d_{01} \circ d_{11} \circ d_{21} \circ d_{31} \circ \dots ,$   
 $d_{02} \circ d_{12} \circ d_{22} \circ d_{32} \circ \dots ,$   
 $d_{03} \circ d_{13} \circ d_{23} \circ d_{33} \circ \dots ,$   
 $\dots )$

– Examples:

- **global sum or product**
- **global maximum or minimum**
- **global user-defined operation**



# Predefined Reduction Operations

Name		Meaning
<i>(fortran,c)</i>	<i>(mpi4py)</i>	
MPI_MAX	MPI.MAX	maximum
MPI_MIN	MPI.MIN	minimum
MPI_SUM	MPI.SUM	sum
MPI_PROD	MPI.PROD	product
MPI_LAND	MPI.LAND	logical and
MPI_BAND	MPI.BAND	bit-wise and
MPI_LOR	MPI.LOR	logical or
MPI_BOR	MPI.BOR	bit-wise or
MPI_LXOR	MPI.LXOR	logical exclusive or (xor)
MPI_BXOR	MPI.BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	MPI.MAXLOC	max value and location
MPI_MINLOC	MPI.MINLOC	min value and location

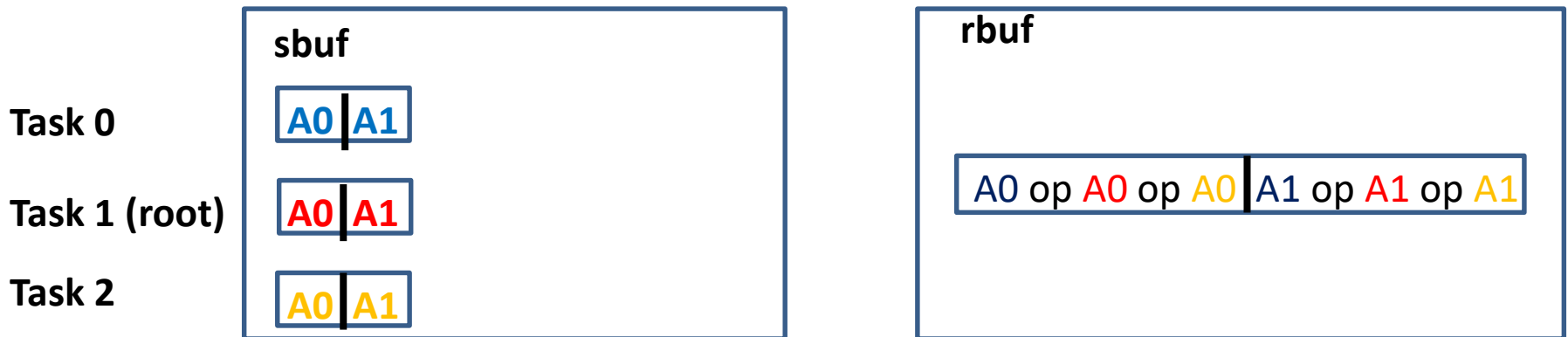
# Reduction Operations

C: `MPI_Reduce( void *sbuf, void *rbuf, int count  
                  , MPI_Datatype datatype, MPI_Op op  
                  , int root, MPI_Comm comm )`

FORTRAN: `MPI_REDUCE( sbuf, rbuf, count, datatype, op, root  
                          , comm, ierror )`  
`<type> sbuf(*), rbuf(*)`  
`INTEGER count, datatype, op, root, comm, ierror`

mpi4py: `comm.Reduce(sbuf, rbuf, op=oper root= 0)`

- All processes must supply the same values for **count**, **root**, **comm** and **datatype**



# Variants of Reduction Operations

- `MPI_ALLREDUCE`
  - returns the result in all processes
  - no root argument
- `MPI_REDUCE_SCATTER_BLOCK` and `MPI_REDUCE_SCATTER`
  - result vector of the reduction operation is scattered to the processes into the result buffers
- `MPI_SCAN`
  - result at process with rank  $i$  :  
=reduction of sbuf-values from rank 0 to rank  $i$
- `MPI_EXSCAN`
  - result at process with rank  $i$  :  
=reduction of sbuf-values from rank 0 to rank  **$i-1$**

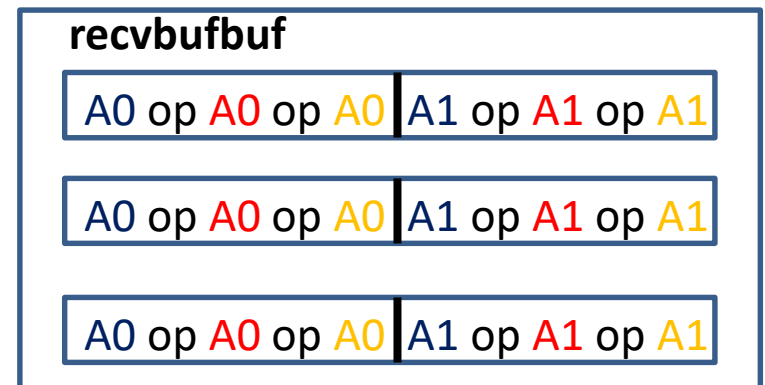
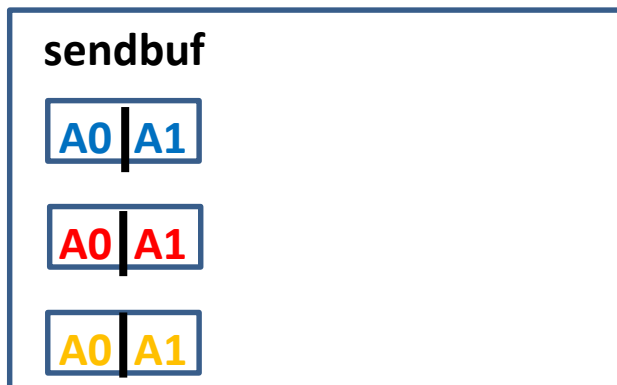
# Reduktions-Operationen

MPI\_ALLREDUCE( sendbuf, recvbuf, count, datatype, op, comm)

Task 0

Task 1

Task 2



# Reduktions-Operationen

`MPI_REDUCE_SCATTER( sendbuf, recvbuf, recvcounts, datatype,  
op, comm)`

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements of the result distributed to each process.
IN	datatype	data type of elements of send and receive buffers (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

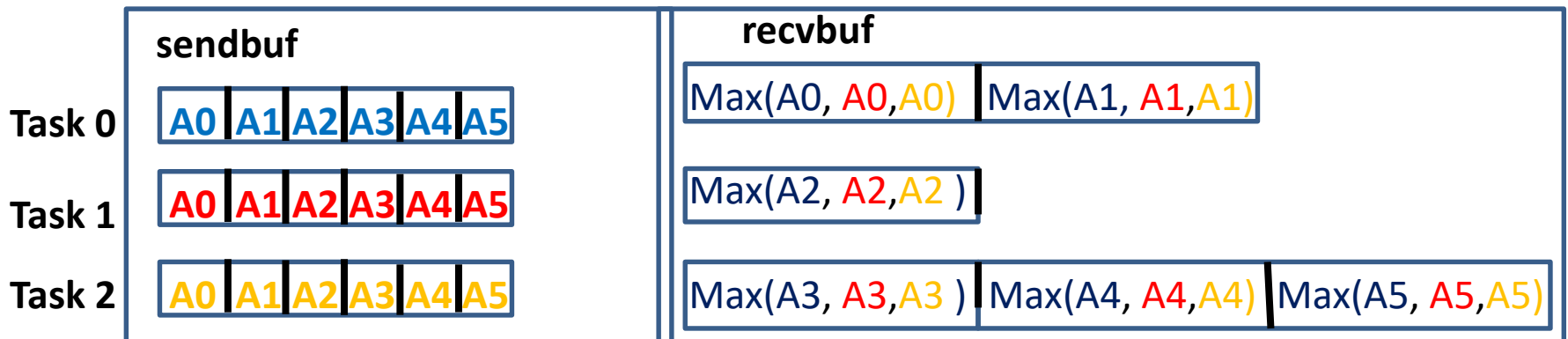
# Reduktions-Operationen

`MPI_REDUCE_SCATTER( sendbuf, recvbuf, recvcounts, datatype,  
op, comm)`

z.B. `recvcounts(0) = 2, recvcounts(1) = 1, recvcounts(2) = 3`  
`op = MPI_MAX`

The number of elements in sendbuf to be reduced over  
nproc tasks is

$$\text{recvcount}(0) + \dots + \text{recvcount}(\text{nproc}-1)$$



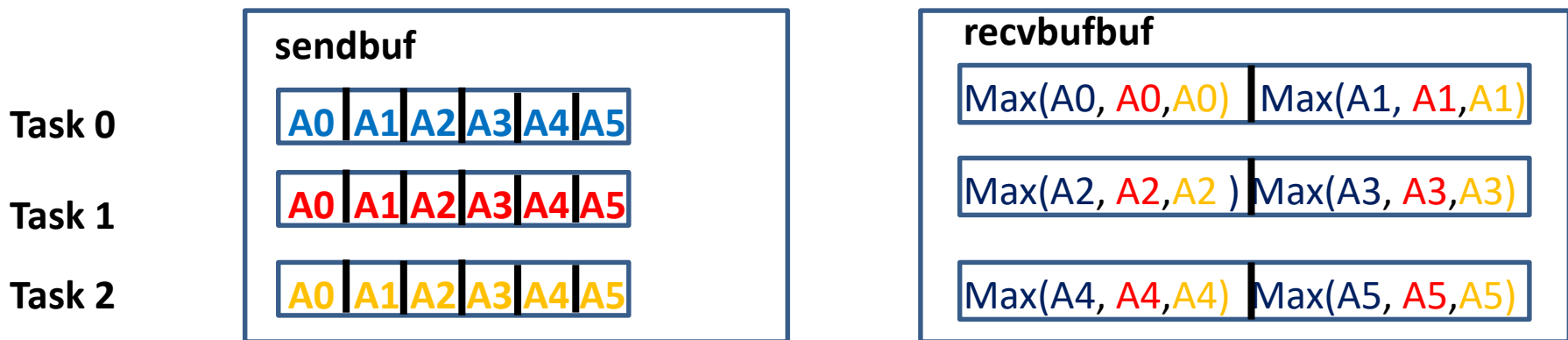
# Reduktions-Operationen

MPI\_REDUCE\_SCATTER\_BLOCK( sendbuf, recvbuf, recvcnt,  
datatype, op, comm)

z.B. recvcnt = 2, op = MPI\_MAX

The number of elements in sendbuf to be reduced over  
nproc tasks is

$$nproc * recvcnt$$



# Reduktions-Operationen

MPI\_SCAN( sendbuf, recvbuf, count, datatype, op, comm)

Task 0

Task 1

Task 2

