GWDG – Kurs
Parallel Programming with MPI

MPI
Collective Operations

Oswald Haan
ohaan@gwdg.de

Learning Objectives

- Why Provide Collective Operations
- In Addition to Send-Receive Message Passing
- MPI Functions for Collective Operations:

MPI_BARRIER,
MPI_BCAST, MPI_SCATTER, MPI_GATHER,
MPI_REDUCE

Outline

- Why Collective Operations?
- Data-Patterns of Collectives
- Syntax of Communcation Collectives:

  MPI_BARRIER
  MPI_BCAST
  MPI_SCATTER
  MPI_GATHER
  and variants

- Reduction Collectives

  MPI_REDUCE
  MPI_REDUCE_SCATTER
  MPI_SCAN, MPI_EXSCAN

Motivation for Collective Operations

- Process 0 provides an input value:
  copy this value to all processes

- All processes compute a number:
  collect these numbers into an array in process 0

- All processes compute a number:
  determine the maximum of these numbers

MPI provides single function calls to perform the necessary multiple point-to-point message passing communication for collective communication patterns

Motivation for Collective Operations

The implementation can provide different realizations for collective operations, optimized with respect to the underlying hardware and to the data layout in the application.

For Intel MPI implementation, see

https://software.intel.com/en-us/mpi-developer-reference-linux-i-mpi-adjust-family-environment-variables

For OpenMPI, see

Choosing a Specific Collective Algorithm Implementation in OpenMPI

and

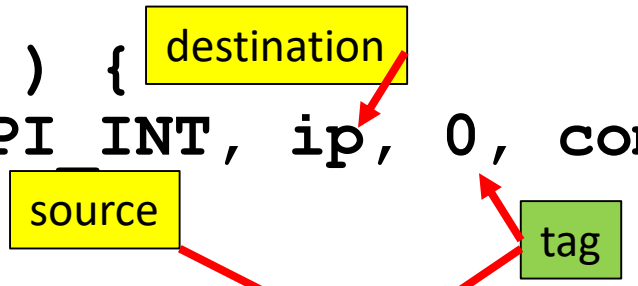
https://www.open-mpi.org/faq/?category=tuning#mca-params

Motivation for Collective Operations

Example: Process 0 copies data to all processes

Point-to-point message passing:

```
if ( me == 0 ) {
   for ( ip=1;ip<np;ip++ ) {
     MPI_Send(&val, 1, MPI_INT, ip, 0, com );}
}
 else  {
     MPI_Recv(&val, 1, MPI_INT, 0, 0, com,
                 MPI_STATUS_IGNORE );
}
```

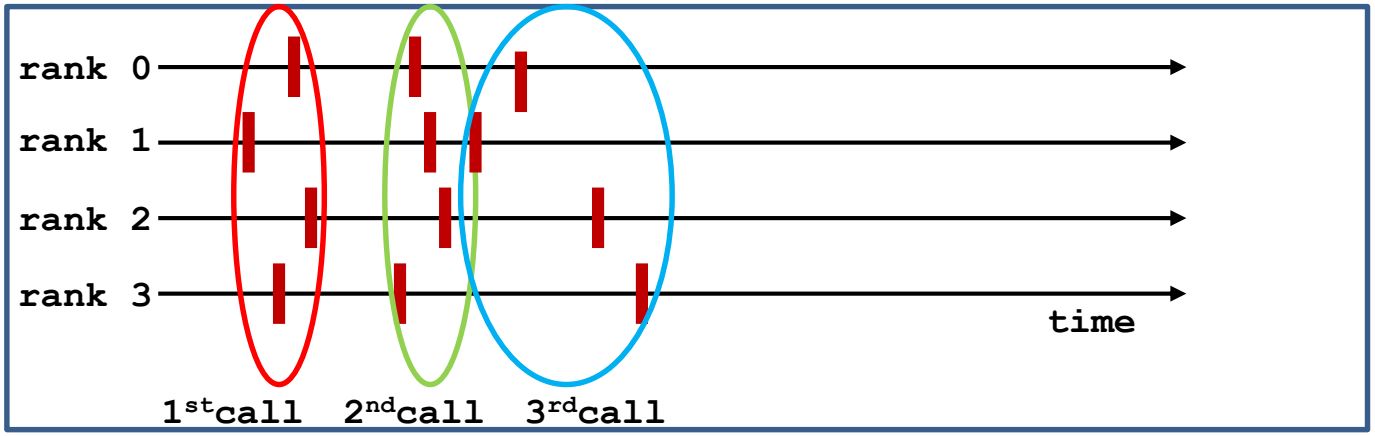


Collective:

```
MPI_Bcast(&val, 1, MPI_Type int, 0, com );
```

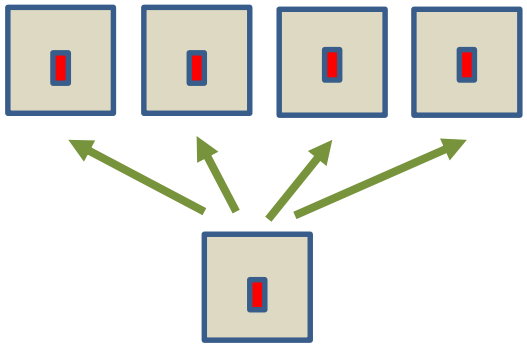
Characeristics of Collective Operations

- 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.

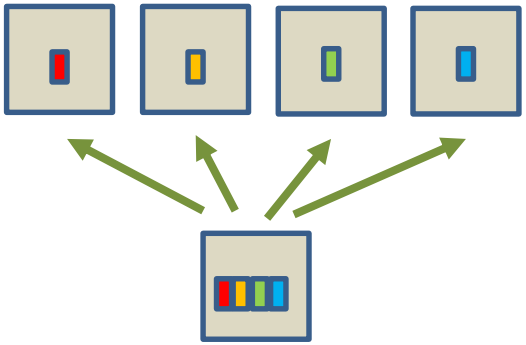


- In MPI-1.0 – MPI-2.2, all collective operations are blocking.
- Non-blocking versions since MPI-3.0.
- buffers on all processes must have exactly the same size.

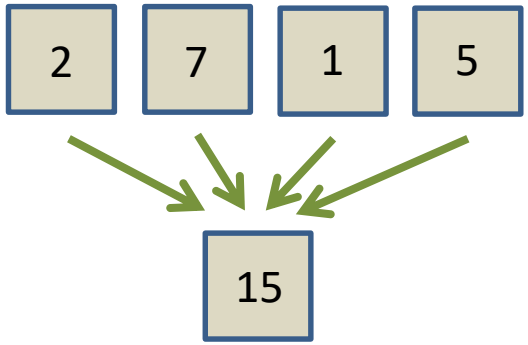
Basic Types of Collective Operations



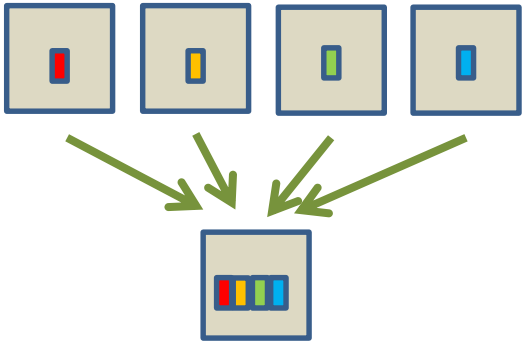
broadcast



scatter



reduce



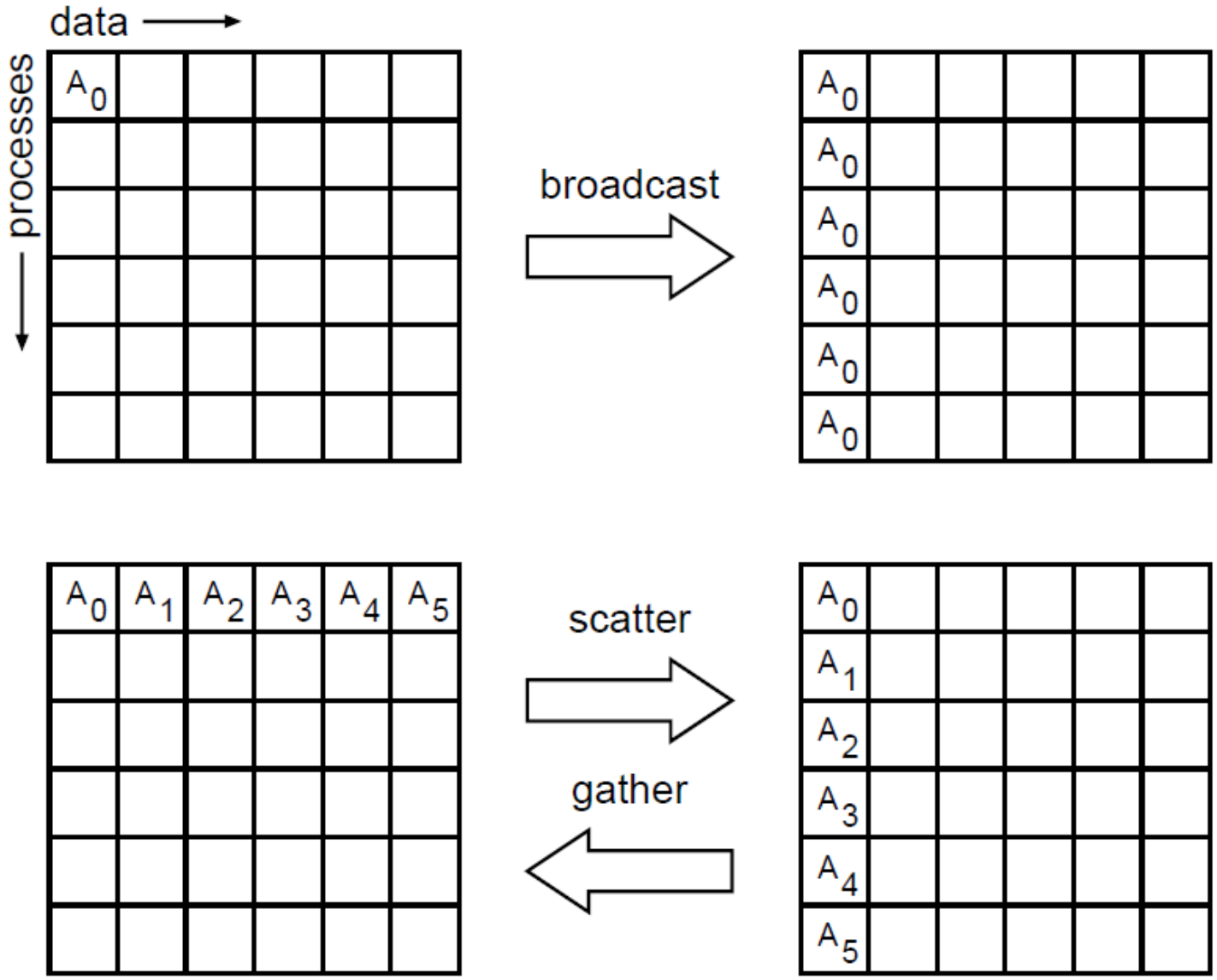
gather

Classification of Collective Operations

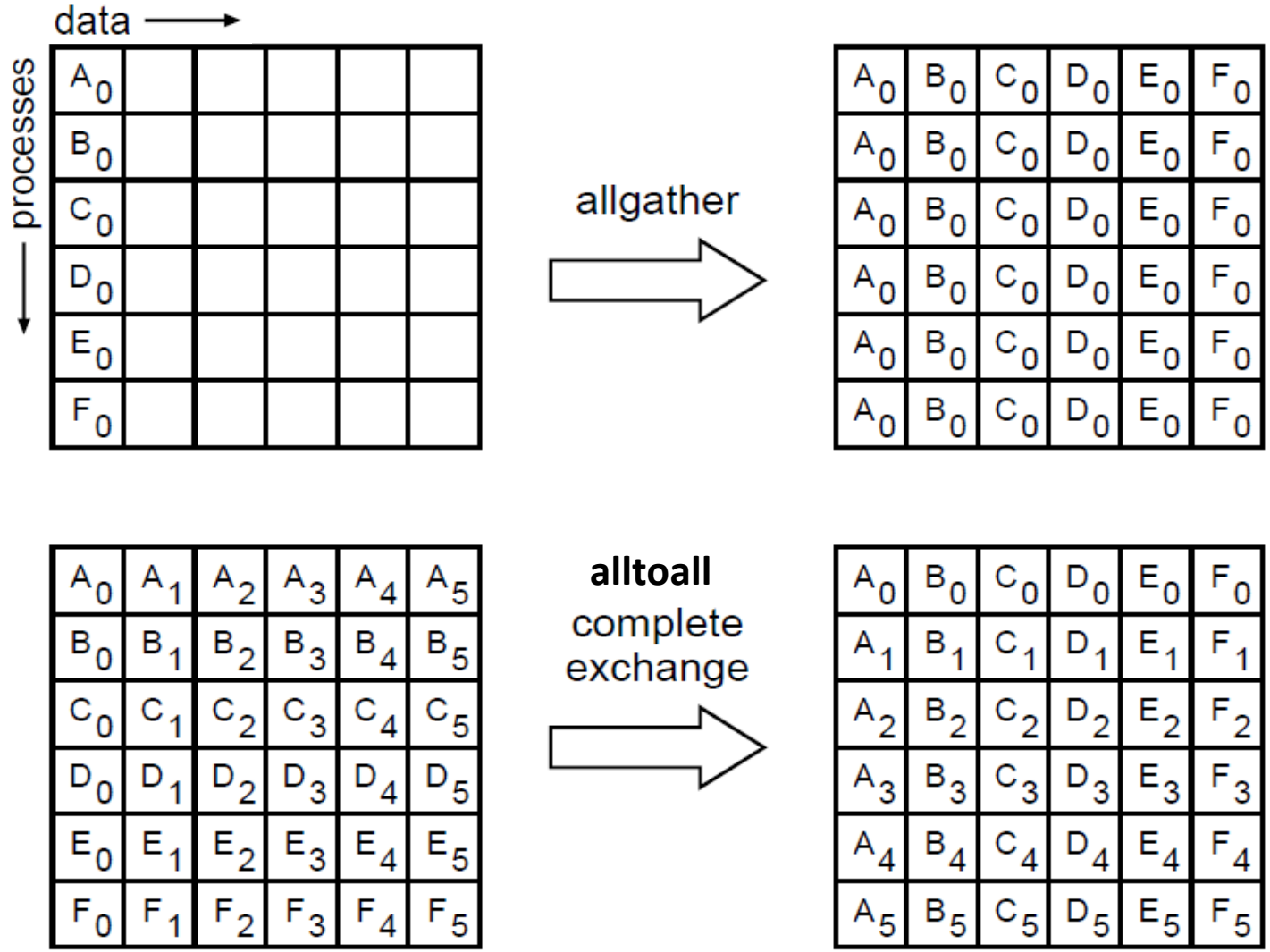
MPI_BARRIER:	Synchronisation
MPI_BCAST:	Send from one process to all processes
MPI_GATHER:	gather data from all processes on one process
MPI_SCATTER:	scatter data from one process to all processes
MPI_ALLGATHER:	gather data from all processes, broadcast them to all processes
MPI_ALLTOALL:	exchange data 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<= i, j<i

one → all all → one all → all

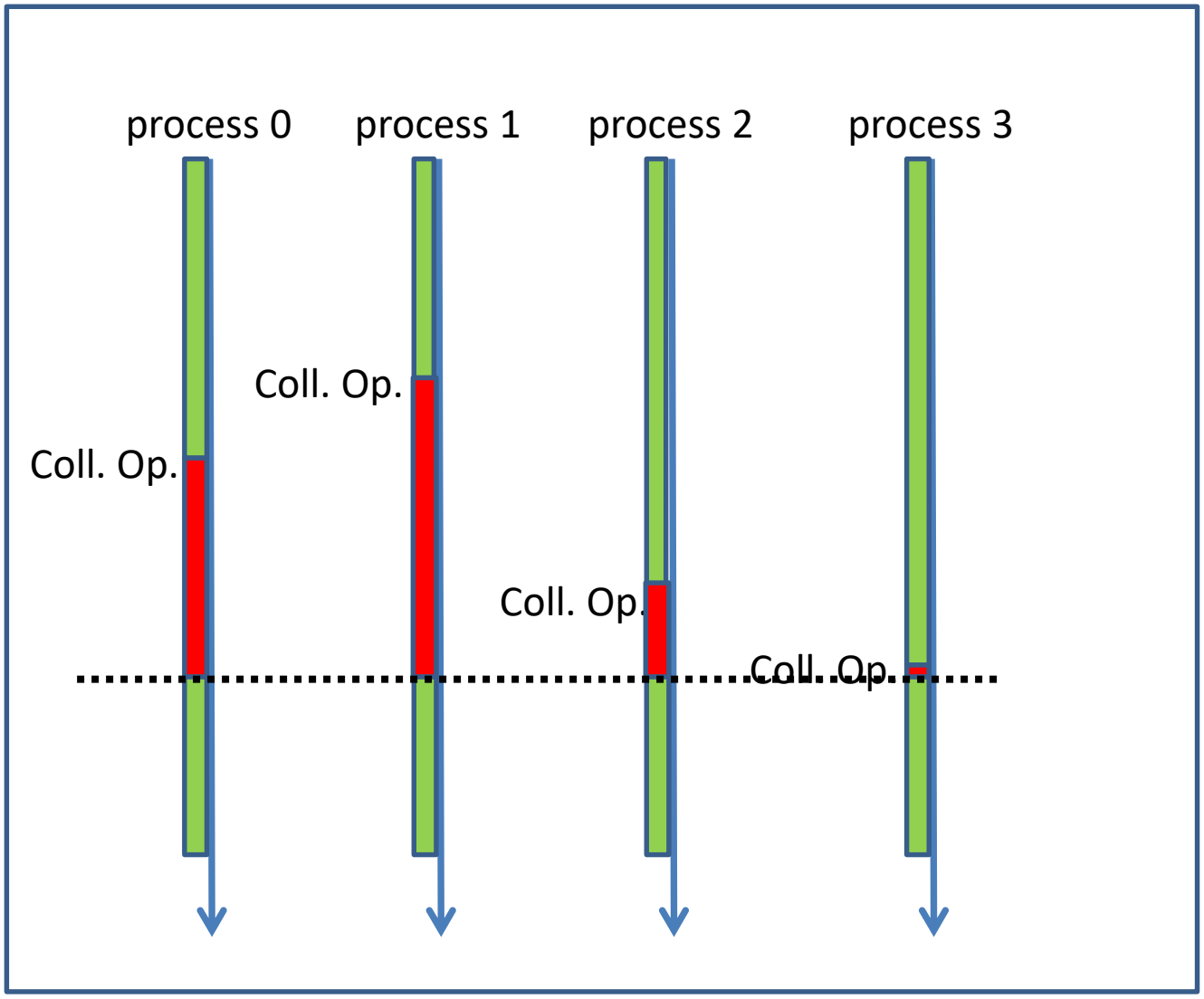
Data Flow in Collective Communication



Kollektive Operationen: Datenfluss



Collective Operations are Blocking



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

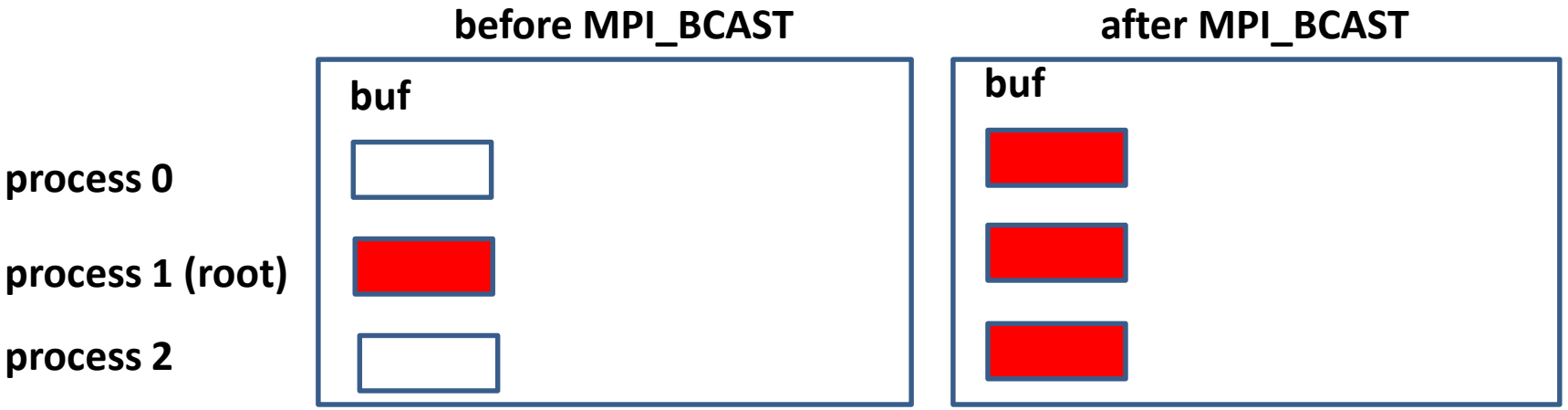
MPI_BCAST: Broadcast from **root**

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

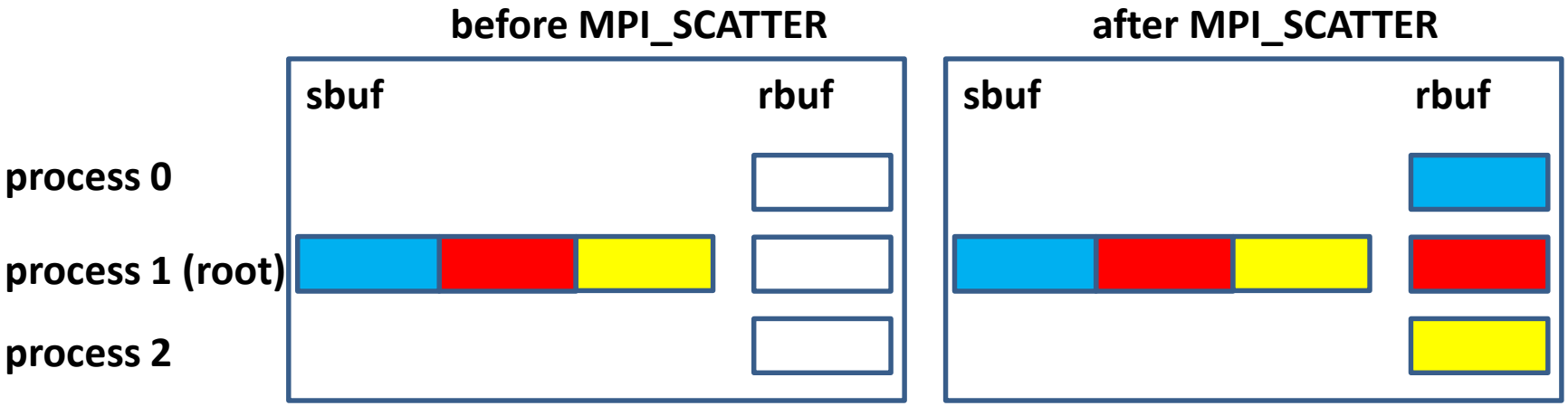


MPI_SCATTER: Scatter from **root**

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



Restrictions for Arguments in MPI_SCATTER

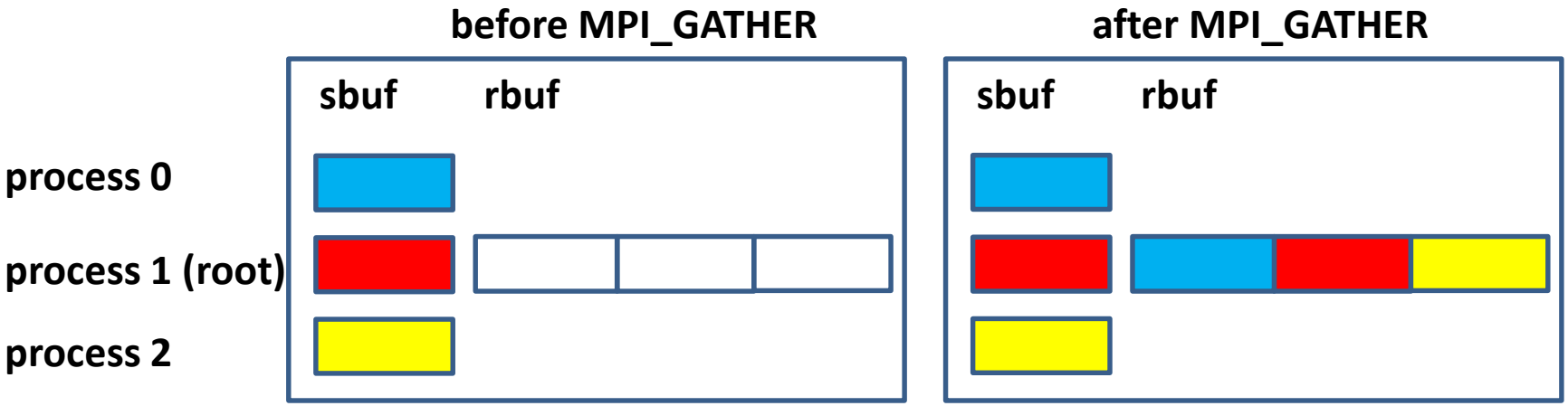
- All processes must supply the same values for **`root`** and **`comm`**
- **`r_data_size = rcount*size(rtype)`** on all processes
  must be equal to
  `s_data_size = scount*size(stype)` on process **`root`**
- **`sbuf`** is ignored an all non-**`root`** processes
- The total size of data scattered 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:
```
robj = 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 an all non-**`root`** processes
- The total size of data gathered on process **`root`** is
  `nproc * r_data_size`

Other Collective Communication Routines

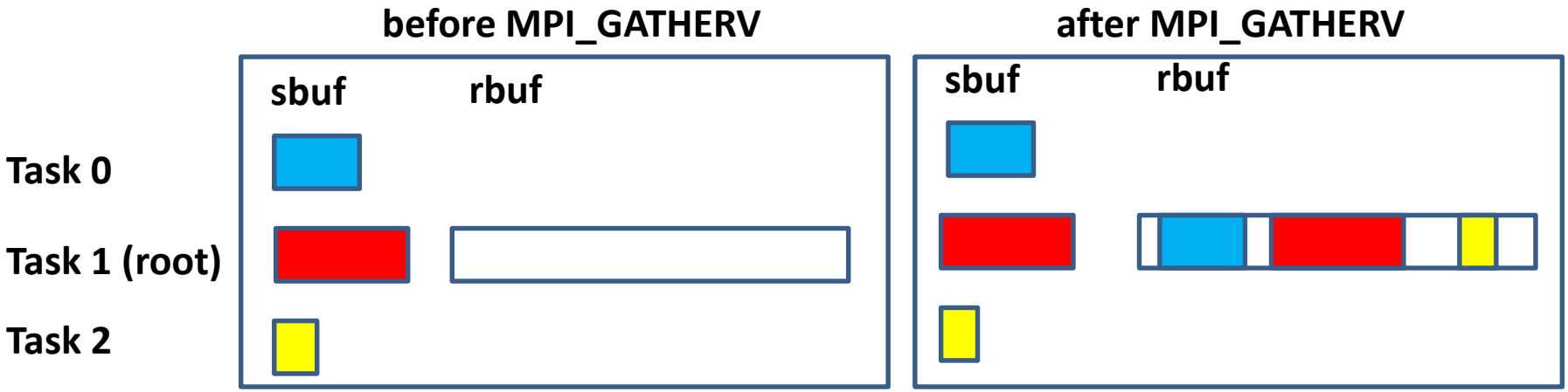
- MPI_ALLGATHER similar to MPI_GATHER,
  - but all processes receive the result vector
  - therefore no **`root`** argument
- MPI_ALLTOALL
  - each process sends messages to all processes
- MPI_GATHERV, _SCATTERV, _ALLGATHERV, _ALLTOALLV
  - collective communication routines with variable data layouts
  - 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(*), dipls(*), 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
  `scount * size(stype)`,
  must be equal to the number of bytes received in the **`i`**-th block in **`rbuf`** on **root**, determined by
  `rcounts(i) * size(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.**

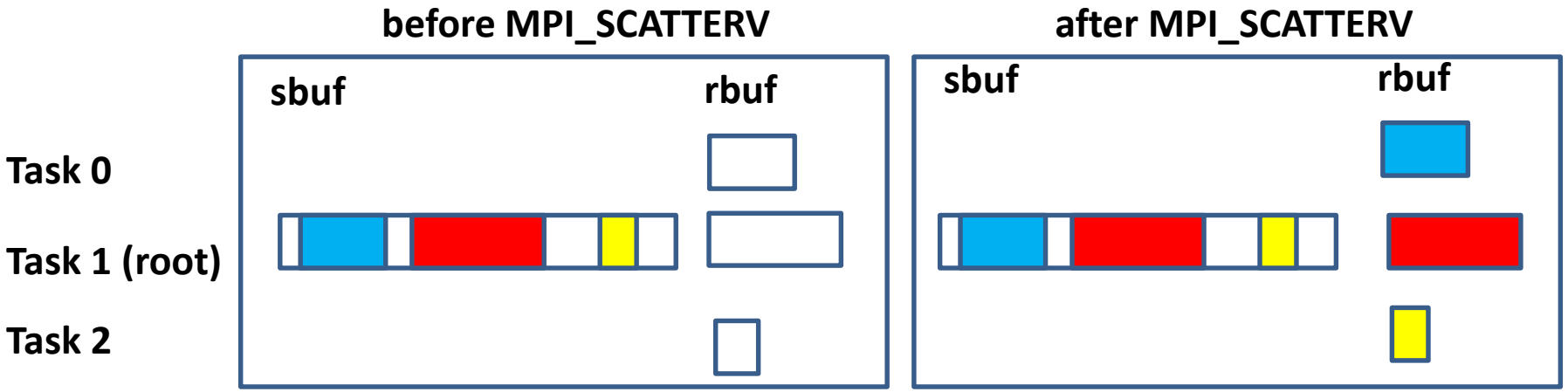
- **`rbuf, rcounts, displs`** will be ignored on all **non-root** tasks .

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(*), dipls(*), 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
  rcount * size(rtype),
  must be equal to the number of bytes sent from the **i**-th block in **sbuf** on **root**, determined by
  scounts(i) * size(stype)

- The **i**-th data block in **sbuf** has **scounts(i)** elements of type **stype**, is located at a distance of **scounts(i)** elements from the start address of **sbu**f 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, if the data to be gathered from the processes are already on their place in the **`rbuf`** of this process.

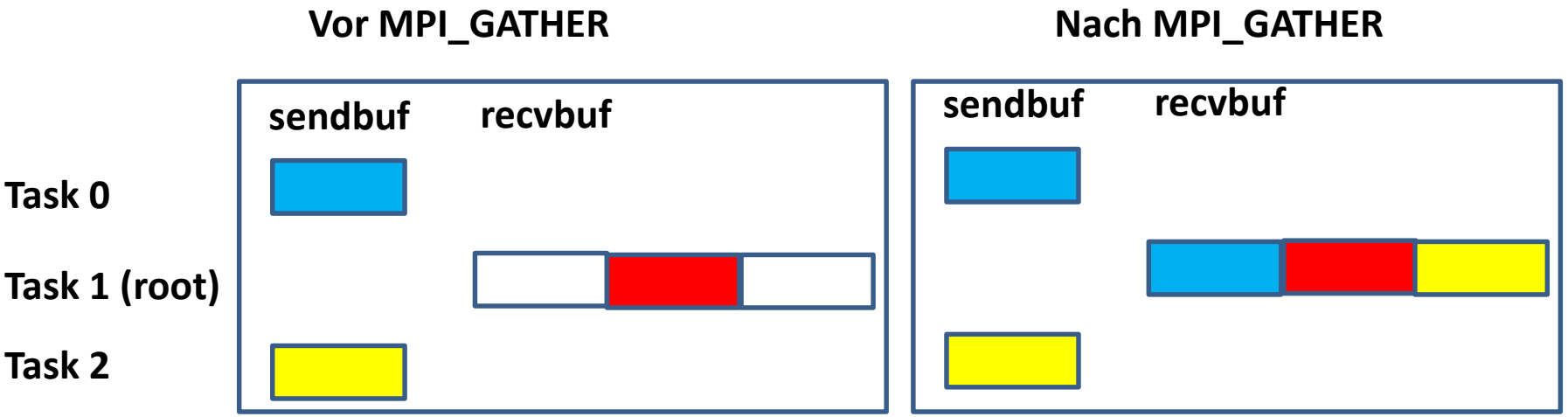
MPI_GATHER: mit MPI_IN_PLACE auf root

non-root-Task

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

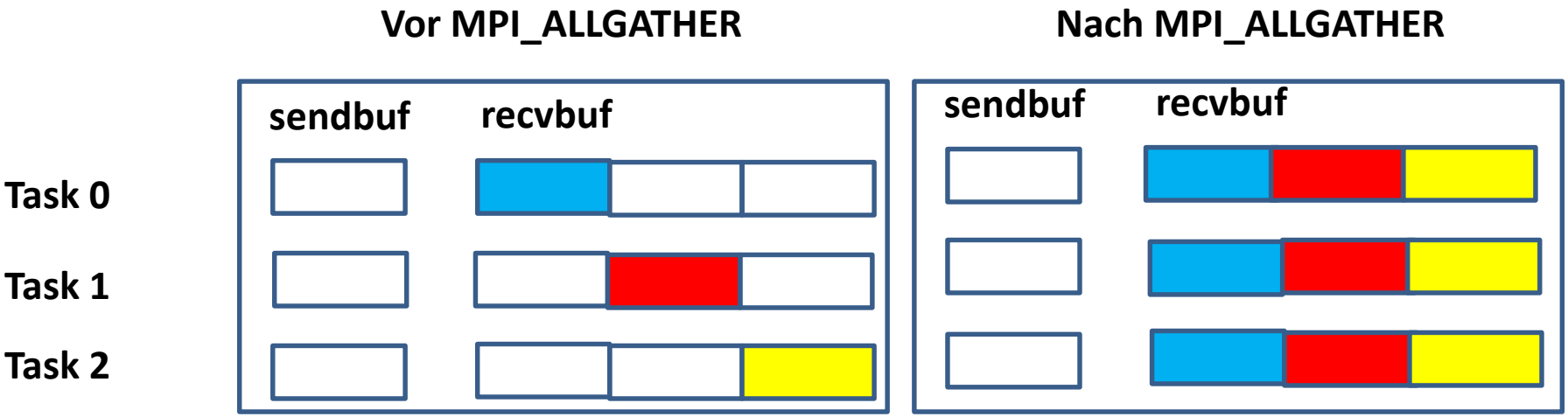
root-Task

MPI_GATHER(MPI_IN_PLACE, sendcount, sendtype, recvbuf, recvcount,
recvtype, root, comm)



MPI_ALLGATHER mit MPI_IN_PLACE

MPI_ALLGATHER(MPI_IN_PLACE, sendcount, sendtype,
recvbuf, recvcount,recvtype, comm)



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:

red**o** = d**0** **o** d**1** **o** d**2** **o** d**3** **o** …

– d**i** = data in process rank i

- **single variable, or**
- **vector,** i.e. d**i** = (d**i**1,d**i**2,d**i**3,…)

– **o** = associative operation

(the result does not depend on the order of the operands)

If the d**i** ‘s are vectors:

d**i** = (d**i**1, d**i**2, d**i**3, …)

the result of the reduce operation also is a vector:

red**o** **= (** d**0**1 **o** d**1**1 **o** d**2**1 **o** d**3**1 **o** … ,
d**0**2 **o** d**1**2 **o** d**2**2 **o** d**3**2 **o** … ,
d**0**3 **o** d**1**3 **o** d**2**3 **o** d**3**3 **o** … ,
…)

– Examples:

- **global sum or product, global maximum or minimum, …**

Predefined Reduction Operations

Name (*fortran,c*)	(*mpi4py*)	Meaning
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

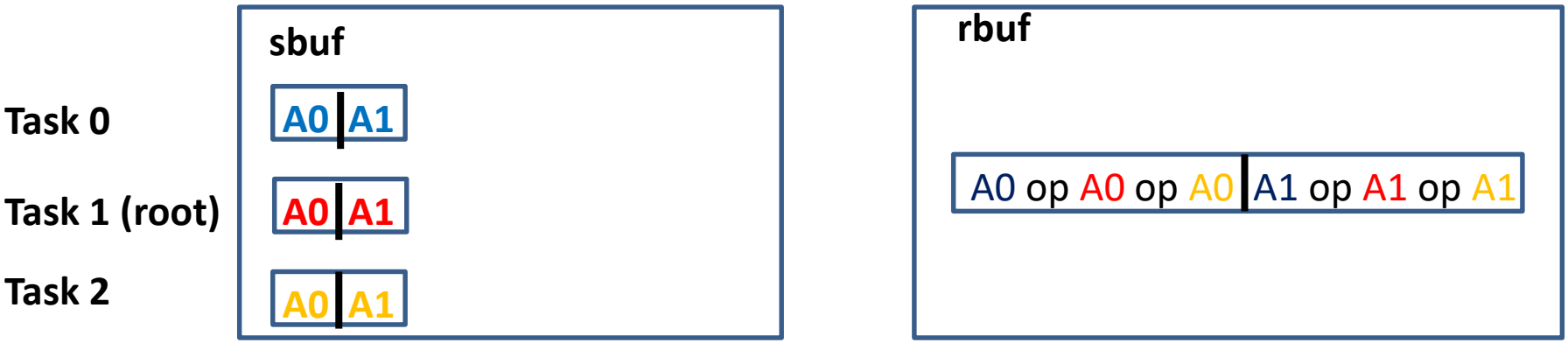
Reduction Operations

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`**

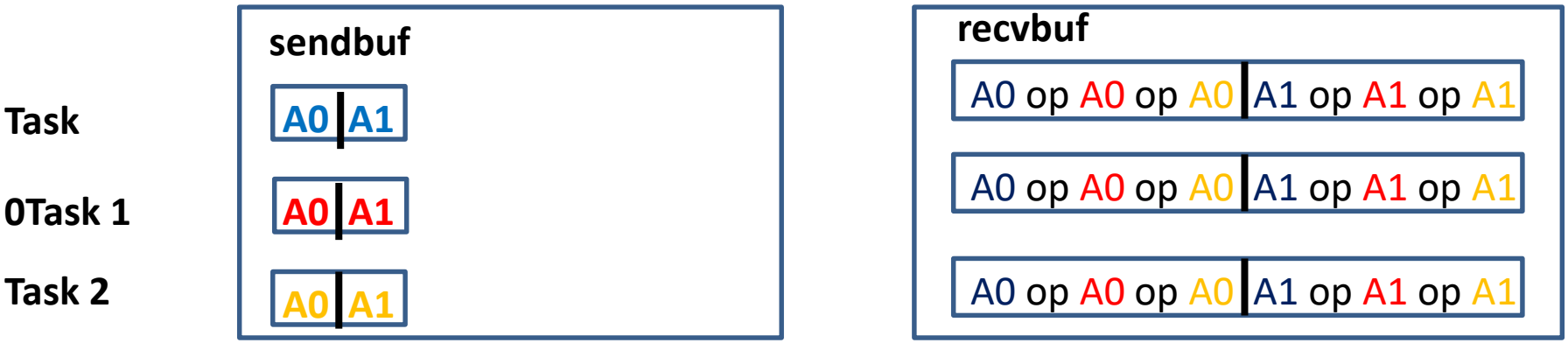


Variants of Reduction Operations

- 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**

Reduction Operations

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



Reduction Operations

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)

Reduction Operations

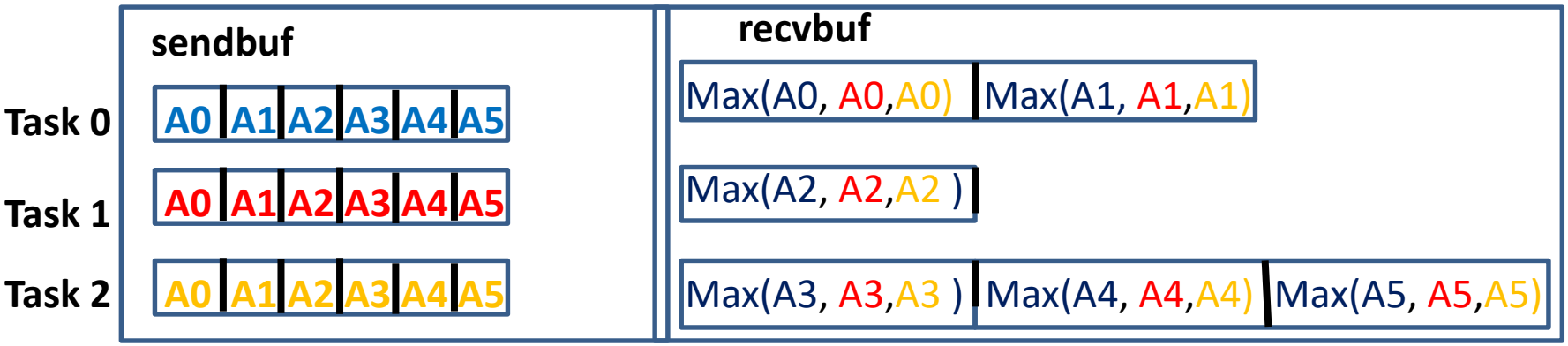
MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

Example: 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

recvcount(0)+ . . . +recvcount(nproc-1)

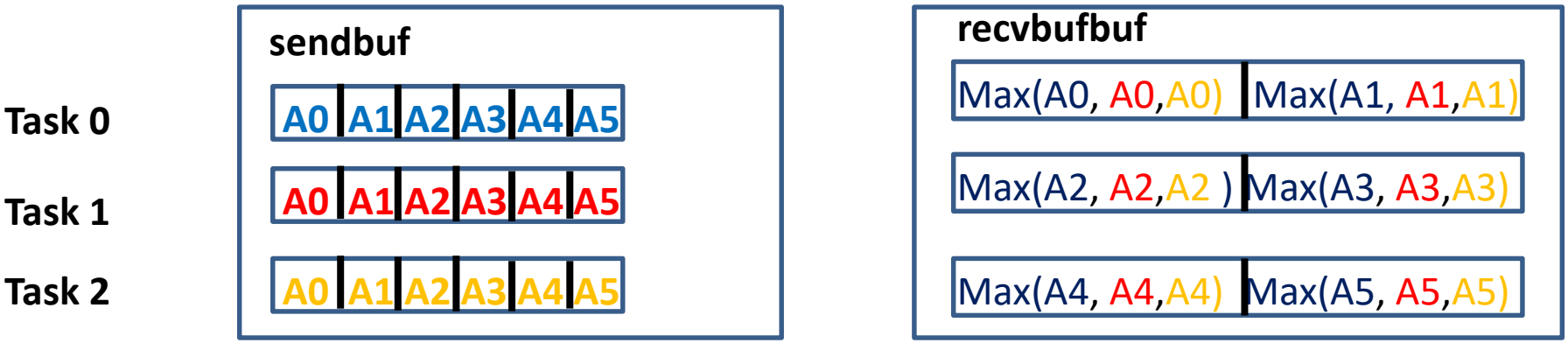


Reduction Operations

MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm)
z.B. recvcount = 2, op = MPI_MAX

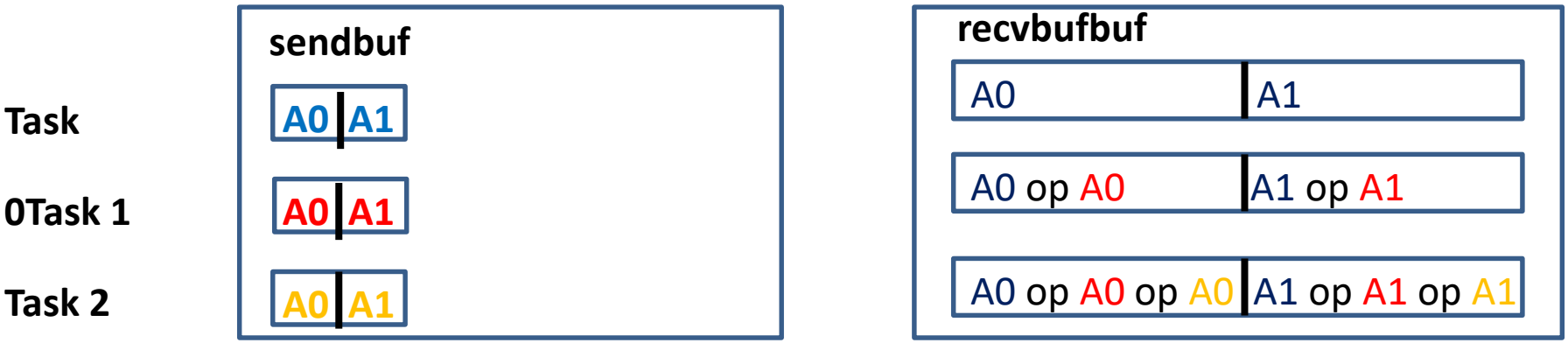
The number of elements in sendbuf to be reduced over nproc tasks is

nproc*recvcount



Reduction Operations

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



Reduction Operations

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