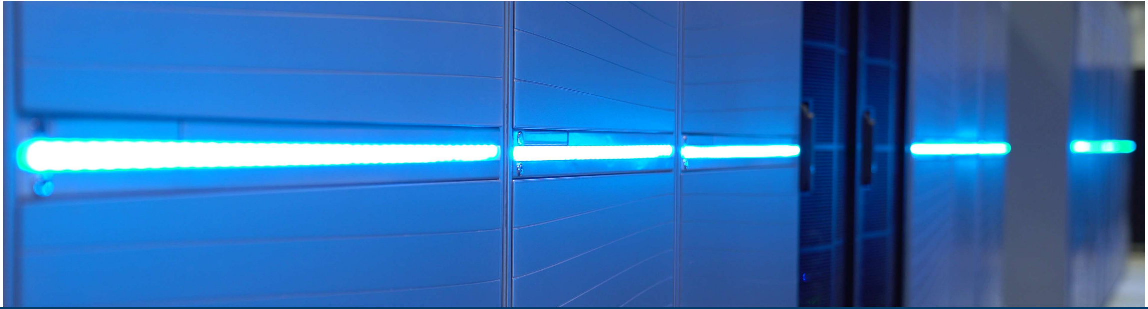


Part V: Collective Communication

Member of the Helmholtz Association

JÜLICH
Forschungszentrum

COLLECTIVE COMMUNICATION [MPI-4.0, 2.4, 6.1]

Collective

A procedure is collective if all processes in a group need to invoke the procedure.

- Collective communication implements certain communication patterns that involve all processes in a group
- Synchronization may or may not occur (except for `MPI_Barrier`)
- No tags are used
- No `MPI_Status` values are returned
- Receive buffer size must match the total amount of data sent (c.f. point-to-point communication where receive buffer capacity is allowed to exceed the message size)
- Point-to-point and collective communication do not interfere

CLASSIFICATION [MPI-4.0, 6.2.2]

One-to-all

`MPI_Bcast`, `MPI_Scatter`, `MPI_Scatterv`

All-to-one

`MPI_Gather`, `MPI_Gatherv`, `MPI_Reduce`

All-to-all

`MPI_Allgather`, `MPI_Allgatherv`, `MPI_Alltoall`, `MPI_Alltoallv`, `MPI_Alltoallw`, `MPI_Allreduce`, `MPI_Reduce_scatter`, `MPI_Barrier`

Other

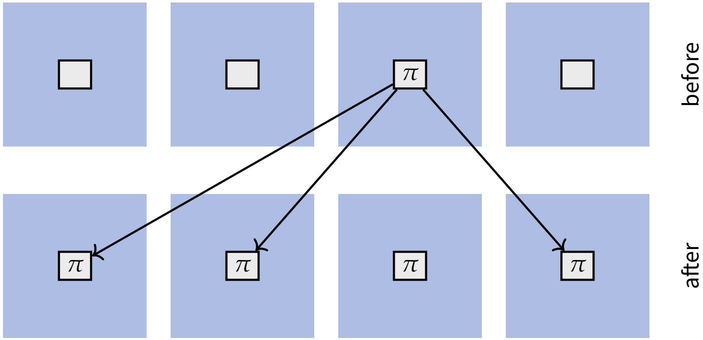
`MPI_Scan`, `MPI_Exscan`

REFERENCES

- MPI standard documentation:
  `https://www.mpi-forum.org/docs/`
- mpich guidebook:
  `https://www.mpich.org/static/docs/v3.3/`
- MPI for Python - mpi4py:
  `https://mpi4py.readthedocs.io/en/stable/`

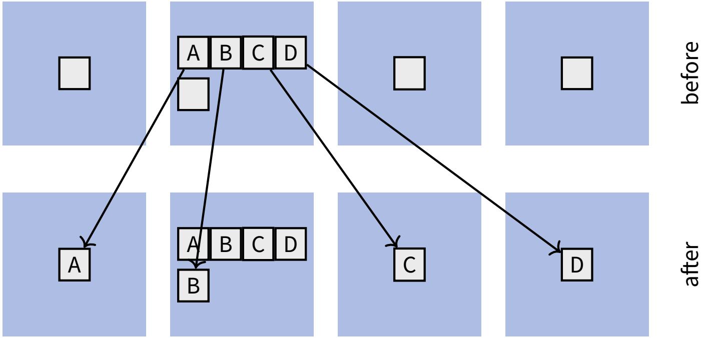
BROADCAST [MPI-4.0, 6.4]

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
 ↪  MPI_Comm comm)
```



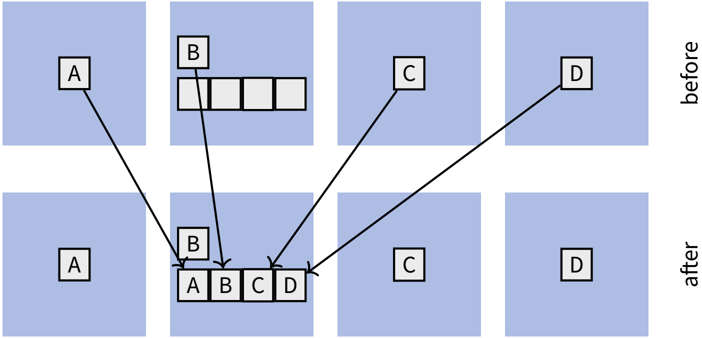
SCATTER [MPI-4.0, 6.6]

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



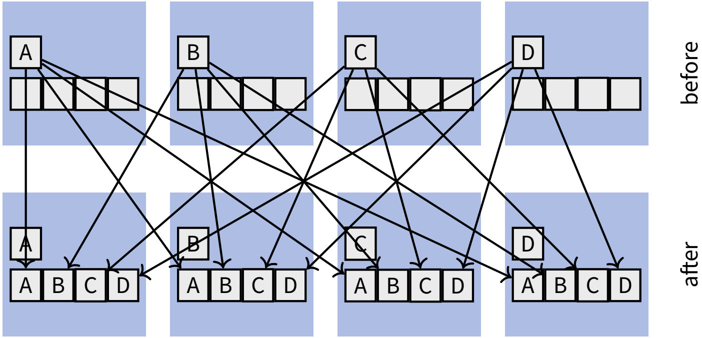
GATHER [MPI-4.0, 6.5]

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



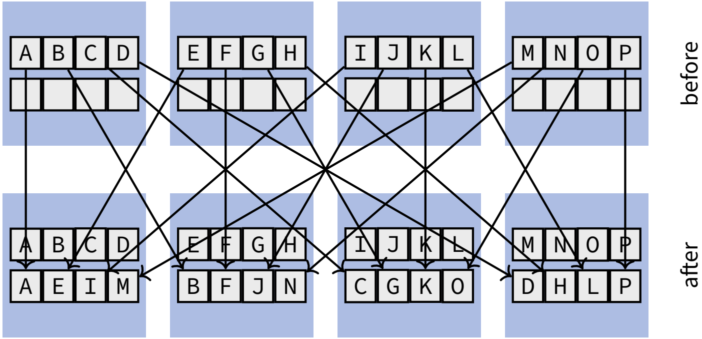
ALLGATHER [MPI-4.0, 6.7]

```
int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype
↪   sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm
↪   comm)
```



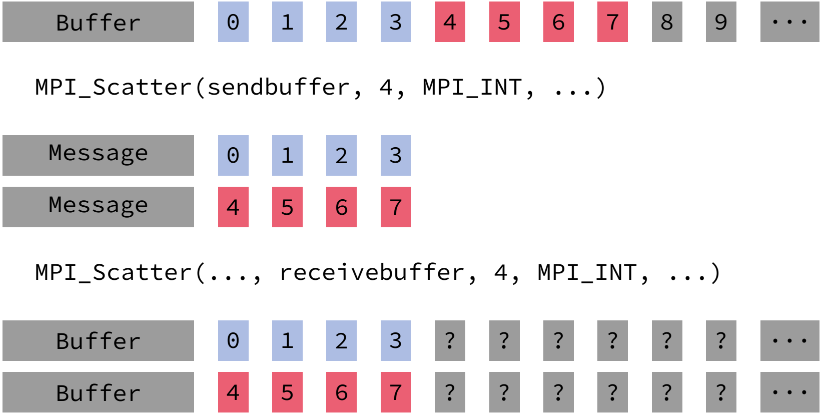
ALL-TO-ALL SCATTER/GATHER [MPI-4.0, 6.8]

```
int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
↪   void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```



MESSAGE ASSEMBLY

Single Message Size



Buffer 0 1 2 3 4 5 6 7 8 9 ···

```
MPI_Scatter(sendbuffer, 4, MPI_INT, ...)
```

Message 0 1 2 3

Message 4 5 6 7

```
MPI_Scatter(..., receivebuffer, 4, MPI_INT, ...)
```

Buffer 0 1 2 3 ? ? ? ? ? ? ···

Buffer 4 5 6 7 ? ? ? ? ? ? ···

DATA MOVEMENT VARIANTS [MPI-4.0, 6.5 – 6.8]

Routines with variable counts (and datatypes):

- `MPI_Scatterv`: scatter into parts of variable length
- `MPI_Gatherv`: gather parts of variable length
- `MPI_Allgatherv`: gather parts of variable length onto all processes
- `MPI_Alltoallv`: exchange parts of variable length between all processes
- `MPI_Alltoallw`: exchange parts of variable length and datatype between all processes

DATA MOVEMENT SIGNATURES

Varying Message Size

```
int MPI_Scatterv(const void *sendbuf, const int *sendcounts, const int
 ↪  *displs, MPI_Datatype sendtype, void *recvbuf, int recvcount,
 ↪  MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Same high-level pattern as before.

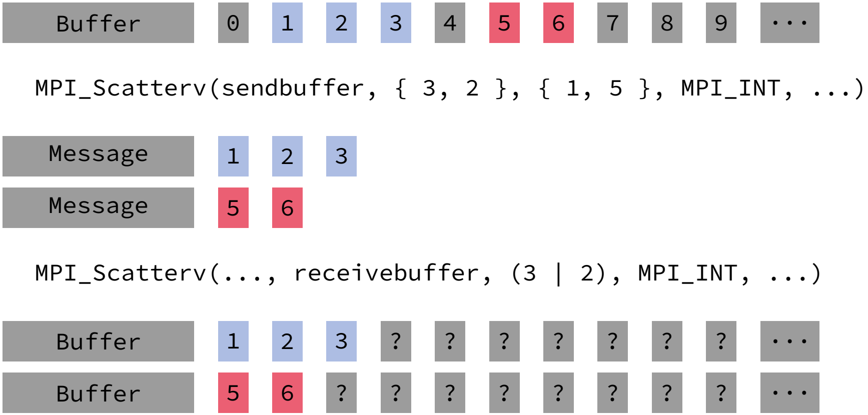
In addition to `send/recvbuffer` following is specified:

- `send/recvcounts` array of length: number of MPI tasks that holds an individual `count` of number of message elements to be send
- `send/recvdispls` array of length: number of MPI tasks that holds the `displacements` (in units of message elements) from the beginning of the buffer at which to start taking elements

Note: Overlapping blocks
The blocks for different messages in send buffers can overlap. In receive buffers, they must not.

MESSAGE ASSEMBLY

Varying Message Size



Buffer 0 1 2 3 4 5 6 7 8 9 ···

```
MPI_Scatterv(sendbuffer, { 3, 2 }, { 1, 5 }, MPI_INT, ...)
```

Message 1 2 3

Message 5 6

```
MPI_Scatterv(..., receivebuffer, (3 | 2), MPI_INT, ...)
```

Buffer 1 2 3 ? ? ? ? ? ? ? ···

Buffer 5 6 ? ? ? ? ? ? ? ? ···

GLOBAL REDUCTION OPERATIONS [MPI-4.0, 6.9]

Associative operations over distributed data

$d_0 \oplus d_1 \oplus d_2 \oplus \ldots \oplus d_{n-1}$, where

d_i, data of process with rank i

$\oplus$, associative operation

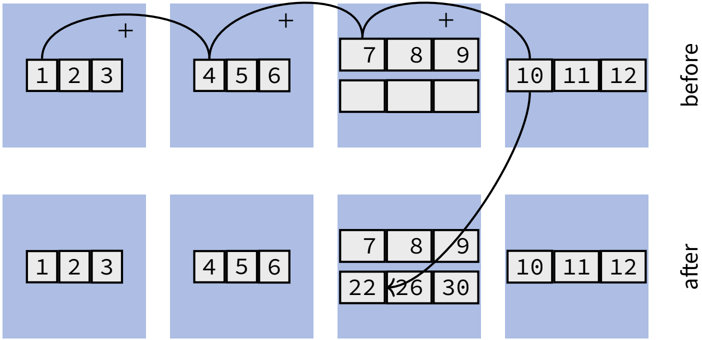
Examples for $\oplus$:

- Sum $+$ and product $\times$
- Maximum max and minimum min
- User-defined operations

Note: Order of application is not defined, watch out for floating point rounding.

REDUCE [MPI-4.0, 6.9.1]

Explanation



REDUCE [MPI-4.0, 6.9.1]

Signature

C

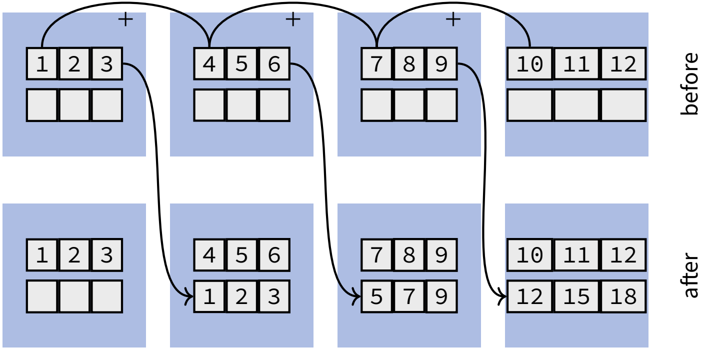
```
int MPI_Reduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype
 ↪  datatype, MPI_Op op, int root, MPI_Comm comm)
```

F08

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
type(*), dimension(..), intent(in) :: sendbuf
type(*), dimension(..) :: recvbuf
integer, intent(in) :: count, root
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Op), intent(in) :: op
type(MPI_Comm), intent(in) :: comm
integer, optional, intent(out) :: ierror
```

EXCLUSIVE SCAN [MPI-4.0, 6.11.2]

Explanation



EXCLUSIVE SCAN [MPI-4.0, 6.11.2]

Signature

C

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

F08

```
MPI_Exscan(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
```

PREDEFINED OPERATIONS [MPI-4.0, 6.9.2]

Name	Meaning
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 the first rank that holds it
MPI_MINLOC	Minimum and the first rank that holds it

REDUCTION VARIANTS [MPI-4.0, 6.9 – 6.11]

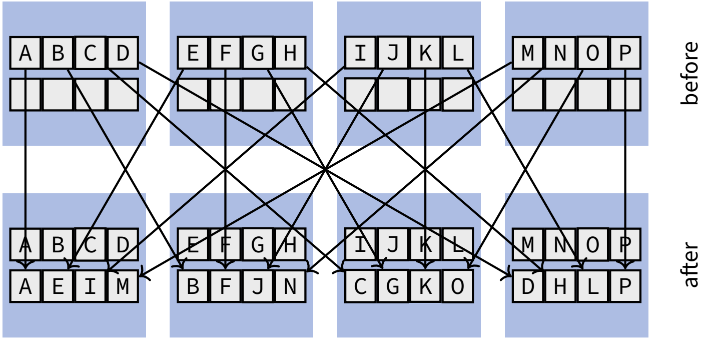
Routines with extended or combined functionality:

- `MPI_Allreduce`: perform a global reduction and copy the result onto all processes
- `MPI_Reduce_scatter`: perform a global reduction then copy different parts of the result onto all processes
- `MPI_Scan`: perform a global prefix reduction, include own data in result

IN PLACE MODE

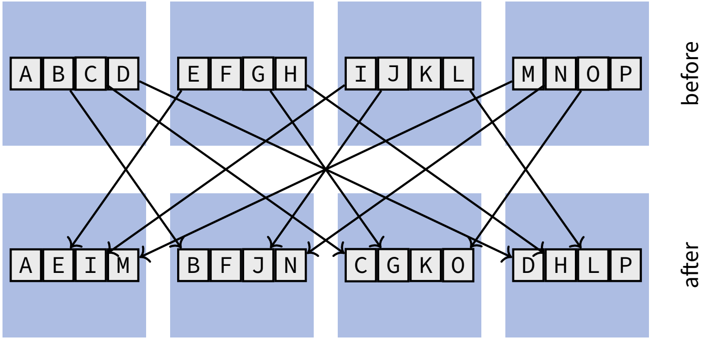
- Collectives can be used in in place mode with only one buffer to conserve memory
- The special value `MPI_IN_PLACE` is used in place of either the send or receive buffer address
- `count` and `datatype` of that buffer are ignored

IN PLACE ALL-TO-ALL SCATTER/GATHER



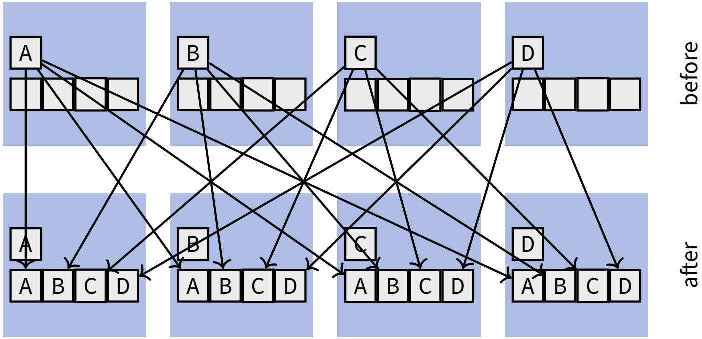
If `MPI_IN_PLACE` is used for `sendbuf` on all processes, `sendcount` and `sendtype` are ignored and the input data is assumed to already be in the correct position in `recvbuf`.

IN PLACE ALL-TO-ALL SCATTER/GATHER



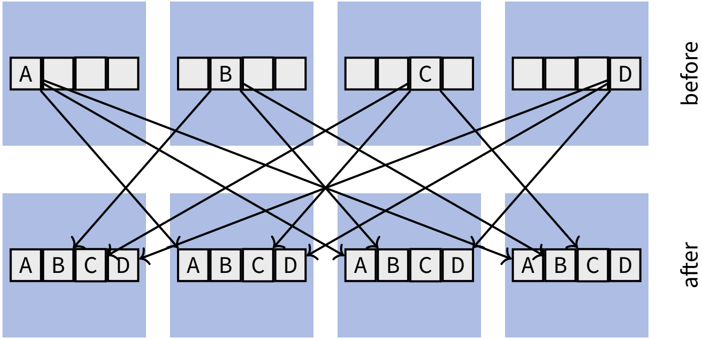
If `MPI_IN_PLACE` is used for `sendbuf` on all processes, `sendcount` and `sendtype` are ignored and the input data is assumed to already be in the correct position in `recvbuf`.

IN PLACE ALLGATHER



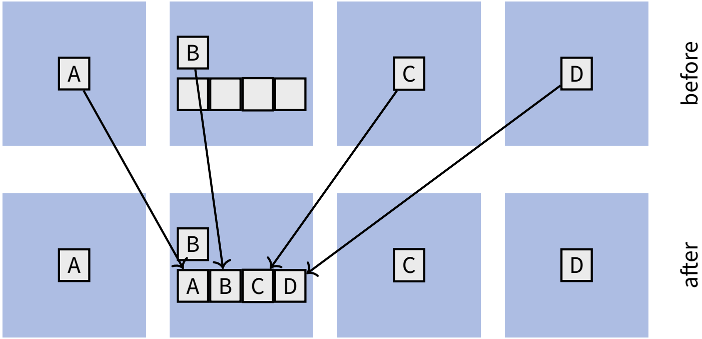
If `MPI_IN_PLACE` is used for `sendbuf` on all processes, `sendcount` and `sendtype` are ignored and the input data is assumed to already be in the correct position in `recvbuf`.

IN PLACE ALLGATHER



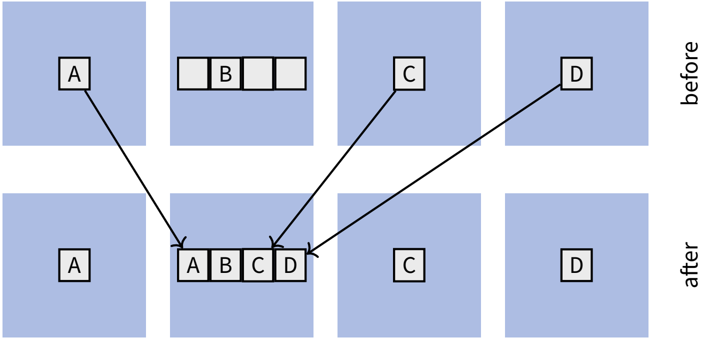
If `MPI_IN_PLACE` is used for `sendbuf` on all processes, `sendcount` and `sendtype` are ignored and the input data is assumed to already be in the correct position in `recvbuf`.

IN PLACE GATHER



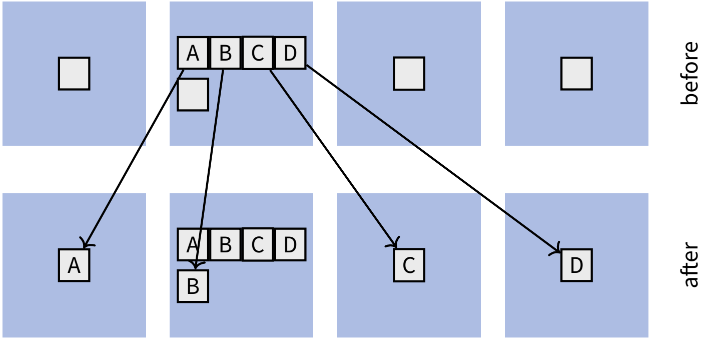
If MPI_IN_PLACE is used for sendbuf on the root process, sendcount and sendtype are ignored on the root process and the root process will not send data to itself.

IN PLACE GATHER



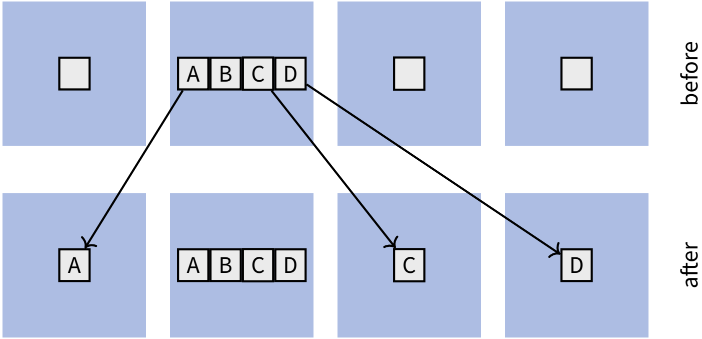
If `MPI_IN_PLACE` is used for `sendbuf` on the root process, `sendcount` and `sendtype` are ignored on the root process and the root process will not send data to itself.

IN PLACE SCATTER



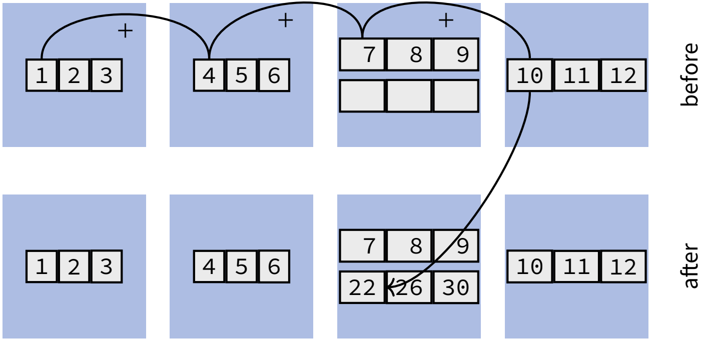
If `MPI_IN_PLACE` is used for `recvbuf` on the root process, `recvcount` and `recvtype` are ignored and the root process does not send data to itself

IN PLACE SCATTER



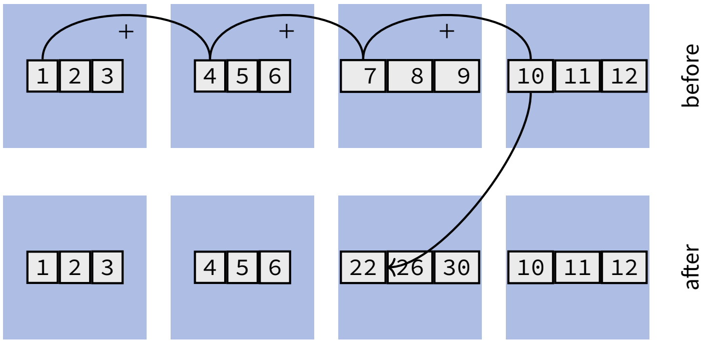
If `MPI_IN_PLACE` is used for `recvbuf` on the root process, `recvcount` and `recvtype` are ignored and the root process does not send data to itself

IN PLACE REDUCE



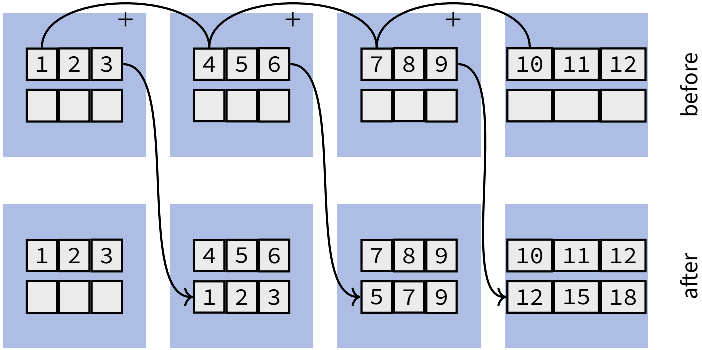
If `MPI_IN_PLACE` is used for `sendbuf` on the root process, the input data for the root process is taken from `recvbuf`.

IN PLACE REDUCE



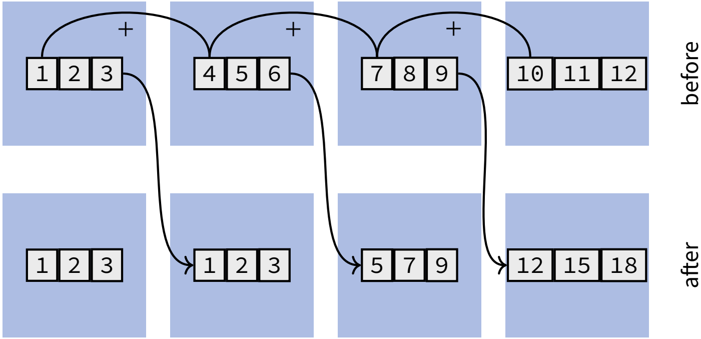
If `MPI_IN_PLACE` is used for `sendbuf` on the root process, the input data for the root process is taken from `recvbuf`.

IN PLACE EXCLUSIVE SCAN



If `MPI_IN_PLACE` is used for `sendbuf` on all the processes, the input data is taken from `recvbuf` and replaced by the results.

IN PLACE EXCLUSIVE SCAN



If MPI_IN_PLACE is used for sendbuf on all the processes, the input data is taken from recvbuf and replaced by the results.

BARRIER [MPI-4.0, 6.3]

C

```c
int MPI_Barrier(MPI_Comm comm)
```

F08

```fortran
MPI_Barrier(comm, ierror)
type(MPI_Comm), intent(in) :: comm
integer, optional, intent(out) :: ierror
```

Explicitly synchronizes all processes in the group of a communicator by blocking until all processes have entered the procedure.

NONBLOCKING COLLECTIVE COMMUNICATION

Blocking

A procedure is blocking if return from the procedure indicates that the user is allowed to reuse resources specified in the call to the procedure.

Nonblocking

All calls are local and return immediately. All associated send buffers and buffers associated with input arguments should not be modified, and all associated receive buffers should not be accessed, until the collective operation completes. The call returns a request handle, which must be passed to a completion call.

Advantages/optimisation:

- overlap communication and computation
- overlap communication and communication: perform collective operations on overlapping communicators (incl. same communicator) and point-to-point communication
- avoid synchronizing effects

PROPERTIES

For all blocking collective calls a nonblocking counterpart exists.

- Nonblocking calls have an extra `request` handel
- Nonblocking calls have are indicated by an extra 'I' letter (for immediate) before in call name: `MPI_I<collective call>`
- Nonblocking collective operation is only complete upon passing through completion routines (`MPI_Wait`, ...)
- All processes must call collective operations (blocking and nonblocking) in the same order per communicator

NONBLOCKING BROADCAST [MPI-4.0, 6.12.2]

Blocking operation

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
 ↪  MPI_Comm comm)
```

Nonblocking operation

```
int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root,
 ↪  MPI_Comm comm, MPI_Request* request)
```

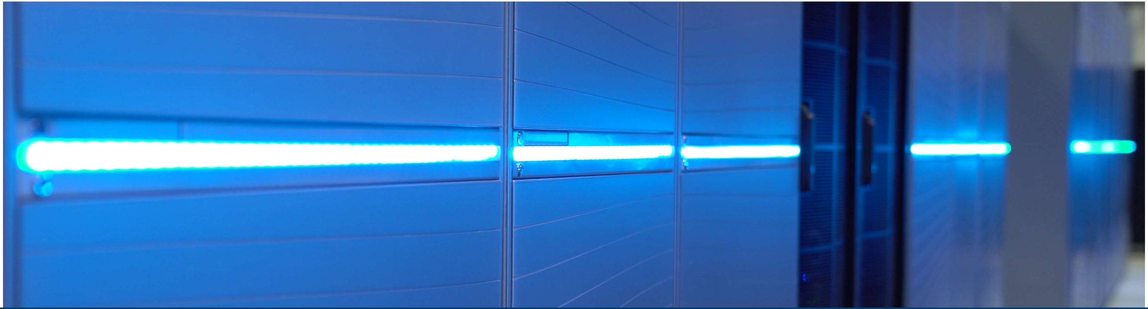
NONBLOCKING BROADCAST [MPI-4.0, 6.12.2]

F08

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
type(*), dimension(..) :: buffer
integer, intent(in) :: count, root
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Comm), intent(in) :: comm
integer, optional, intent(out) :: ierror
```

F08

```
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
type(*), dimension(..), asynchronous :: buffer
integer, intent(in) :: count, root
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Comm), intent(in) :: comm
type(MPI_Request), intent(out) :: request
integer, optional, intent(out) :: ierror
```



Part VI: Collective Communication - Exercises

EXERCISE 1

Exercise 5 – Collective Communication

5.1 Do it yourself

Write your own MPI parallel code using provided `skeleton.{c|F90|py}` with the following criteria:

- The MPI program should produce a sum of the rank of all processes.
- All processes should carry the summed value.
- The MPI program should only contain collective calls.

There are multiple ways to achieve the end result. Experiment with different collective calls.

EXERCISE - ADVANCED

Exercise 6 – Collective Communication

6.1 Redistribution of Points with Collectives

In the file `redistribute.{c|f90|py}` implement the function `redistribute` which should work as follows:

1. All processes call the function collectively and pass in an array of 1000 random numbers, generated from a uniform random distribution on $[0, 1)$.
2. Impose the following rule to each process:
   - Partition $[0, 1)$ among the `nranks` processes: process i gets partition $[i/nranks, (i + 1)/nranks)$.
3. Redistribute the points, so that every process is left with only those points that lie inside its partition and return them from the function.

Guidelines:

- Use collectives, either `MPI_Gather` and `MPI_Scatter` or `MPI_Alltoall(v)`
- It helps to partition the points so that consecutive blocks can be sent to other processes
- `MPI_Alltoall` can be used to distribute the information that is needed to call `MPI_Alltoallv`
- Dynamic memory management could be necessary

The file contains tests that will check your implementation.

Use: `MPI_Alltoall`, `MPI_Alltoallv`