

# MPI in Small Bites

HPC.NRW Competence Network

# Blocking Collective Communication

HPC.NRW Competence Network

## MPI in Small Bites

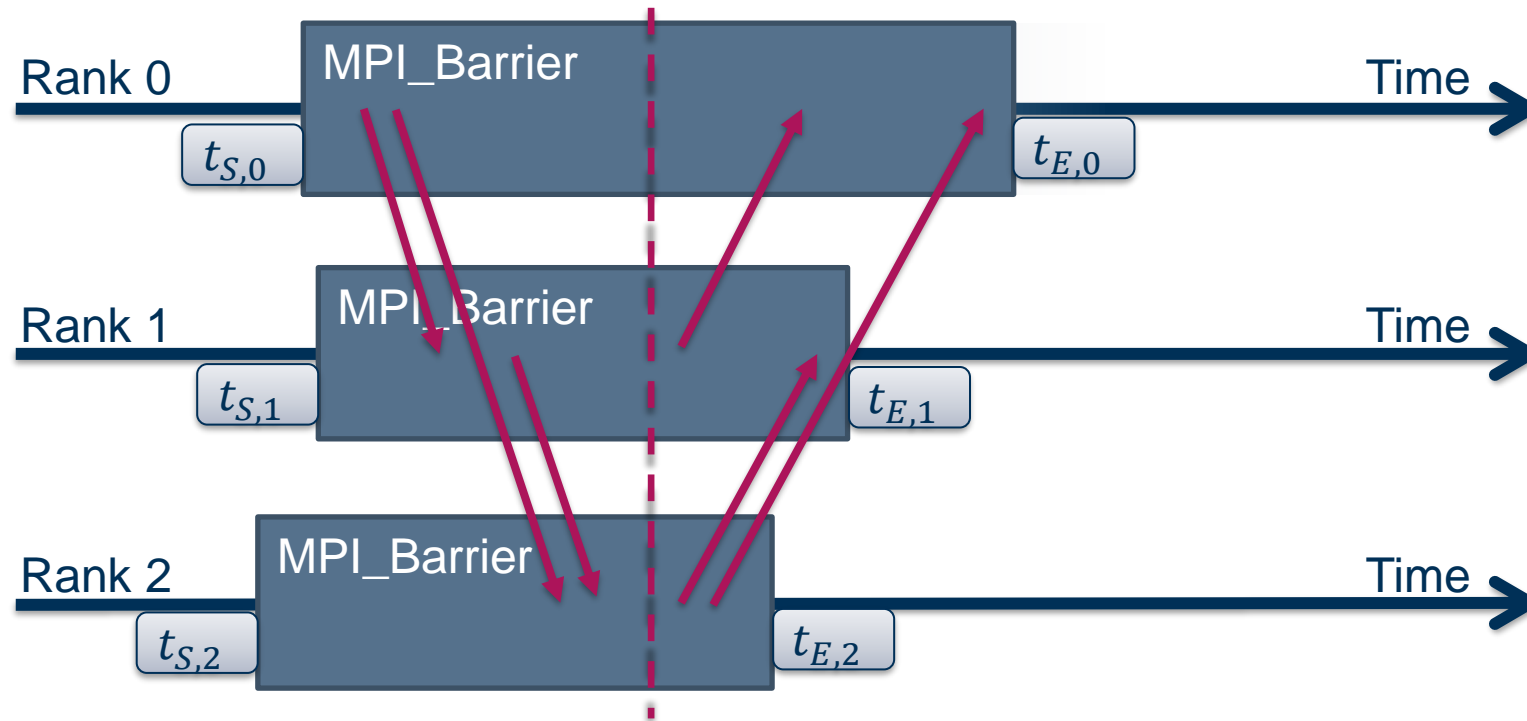
- Involve all ranks in a given communicator
  - Create a smaller communicator for collective communication in a subgroup
- All ranks must call the same MPI operation to succeed
  - There should be only one call per MPI rank (i.e., not per thread)
- Process synchronization behaviour is implementation specific
  - The MPI standard may allow for early return on some ranks
- Implement common group-communication patterns
  - Usually tuned to deliver the best system performance
  - **Do not reinvent the wheel!**



# Barrier Synchronisation

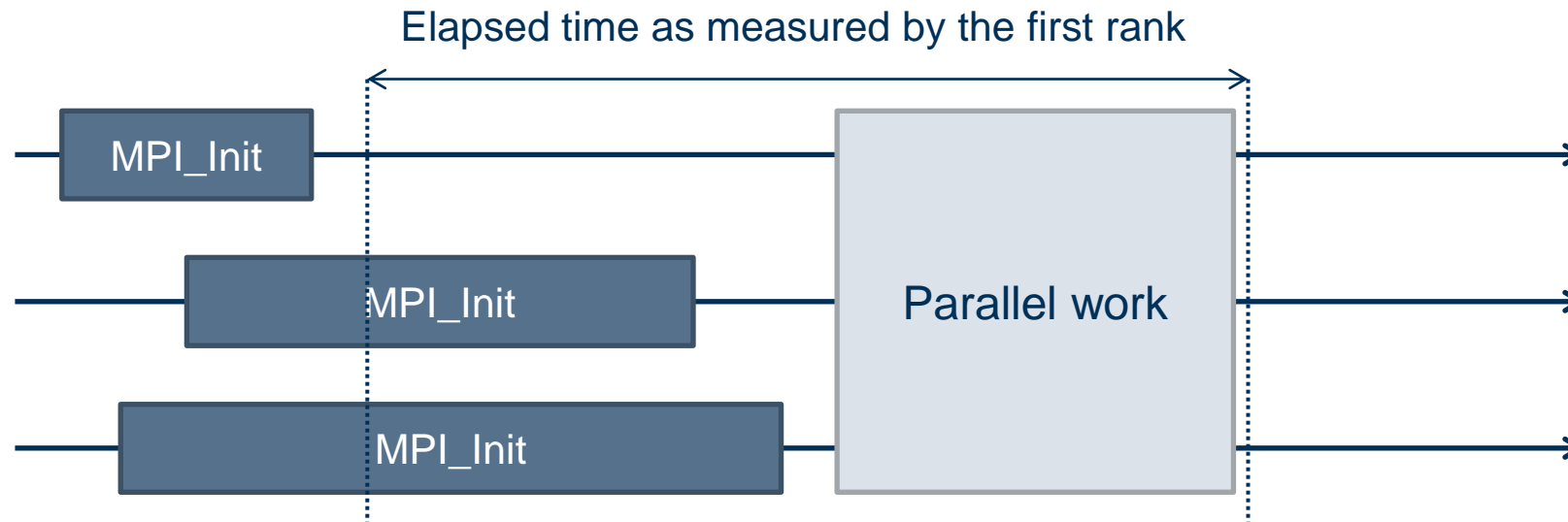
- The only explicit synchronisation operation in MPI:

`MPI_Barrier (MPI_Comm comm)`



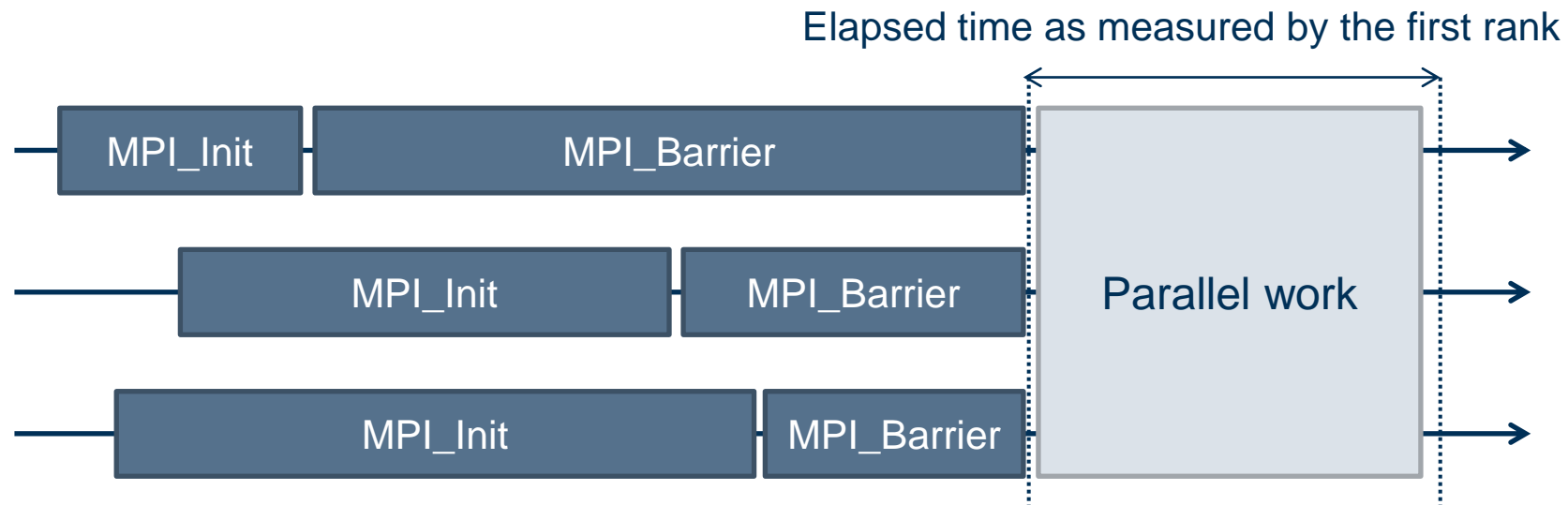
$$\max(t_{S,0}; t_{S,1}; t_{S,2}) < \min(t_{E,0}; t_{E,1}; t_{E,2})$$

- Useful for benchmarking
  - Always synchronise before taking time measurements



- Huge discrepancy between the actual work time and the measurement

- Useful for benchmarking
  - Always synchronise before taking time measurements



- Huge discrepancy between the actual work time and the measurement

# Broadcast (one-to-many data replication)

- Replicate data from one rank to all other ranks:

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

What?

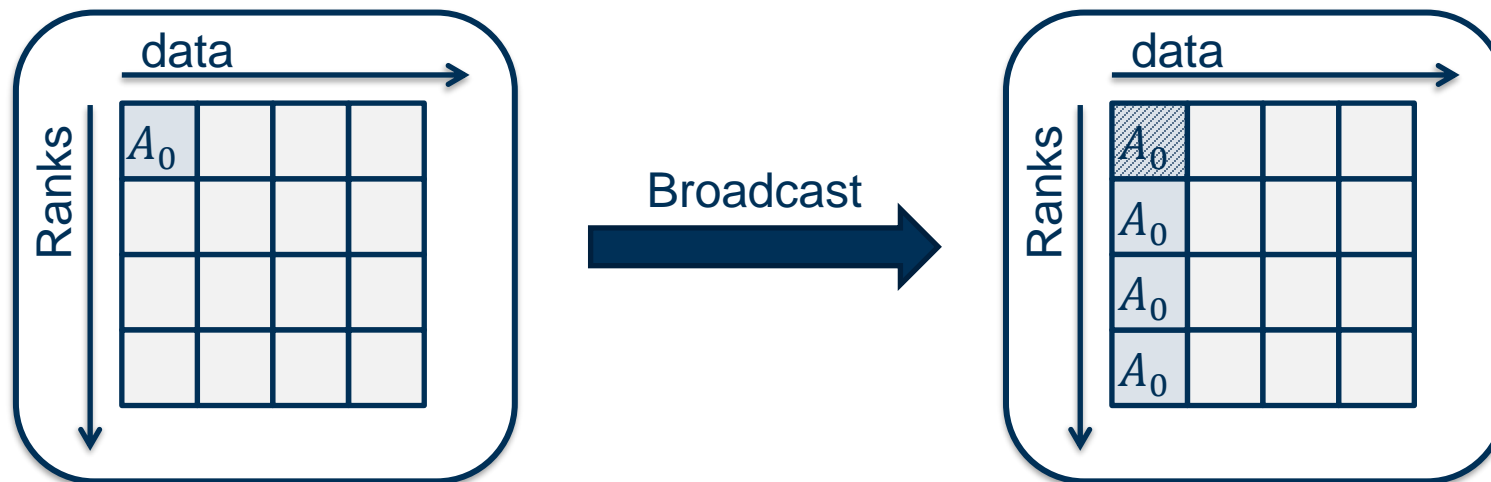
From whom?

- **data:** send buffer at **root** rank; receive buffer on all other ranks
  - **count:** number of data elements
  - **datatype:** elements' datatype
  - **root:** source rank; all ranks must specify the same value
  - **comm:** communicator
- 
- On all ranks but **root**, **data** is an output argument
  - On rank **root**, **data** is an input argument
  - Type signatures must match across all ranks (→ Datatypes)

# Broadcast (one-to-many data replication)

- Replicate data from one rank to all other ranks:

```
MPI_Bcast (void *data, int count, MPI_Datatype dtype,  
           int root, MPI_Comm comm)
```





# Broadcast (one-to-many data replication) – Wrong usage

- Replicate data from one rank to all other ranks:

```
MPI_Bcast (void *data, int count, MPI_Datatype dtype,  
           int root, MPI_Comm comm)
```

- Example:

```
int ival;  
  
if (rank == 0)  
    ival = read_int_from_user();  
  
MPI_Bcast(&ival, 1, MPI_INT, 0, MPI_COMM_WORLD);  
  
// WRONG USAGE!  
if (rank == 0) {  
    ival = read_int_from_user();  
    MPI_Bcast(&ival, 1, MPI_INT, 0, MPI_COMM_WORLD);  
}  
  
// The other ranks do not call MPI_Bcast → Deadlock
```

# Broadcast (one-to-many data replication) – Naïve Implementation


```
void broadcast (void *data, int count, MPI_Type dtype,
               int root, MPI_Comm comm)
{
    int rank, nprocs, i;

    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &nprocs);
    if (rank == root) {
        for (i = 0; i < nprocs; i++)
            if (i != root)
                MPI_Send(data, count, dtype, i, TAG_BCAST, comm);
    }
    else
        MPI_Recv(data, count, dtype, root, TAG_BCAST, comm,
                MPI_STATUS_IGNORE);
}
```

# Scatter (one-to-many data distribution)

- Distribute **chunks** of data from one rank to all ranks:

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



Significant at root  
rank only

# Scatter (one-to-many data distribution)

- Distribute **chunks** of data from one rank to all ranks:

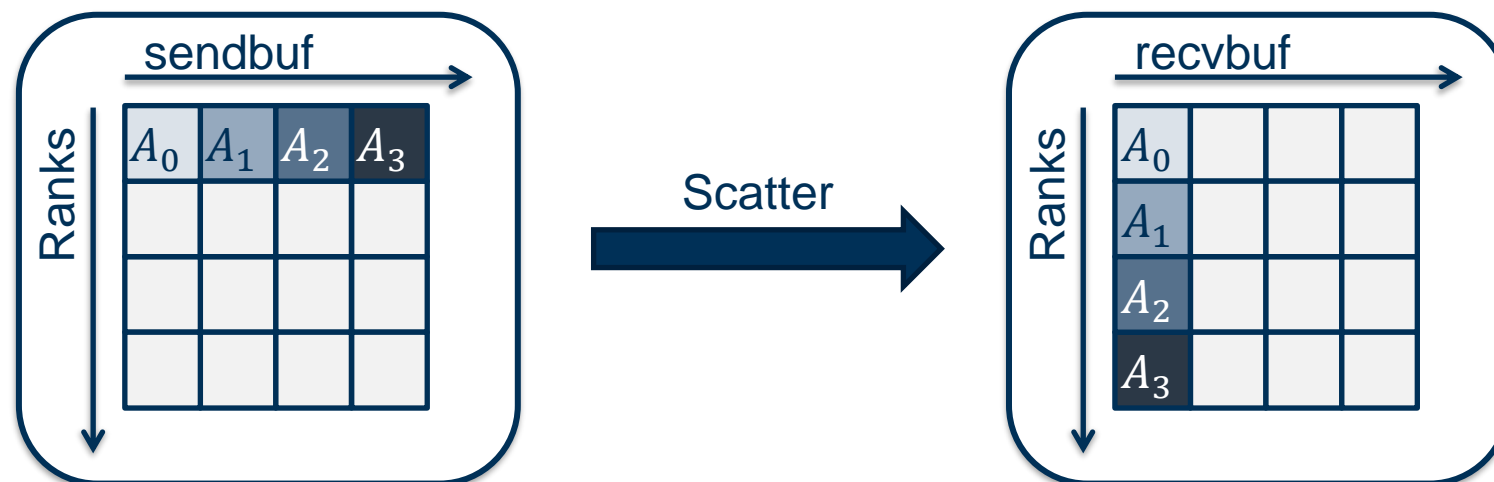
```
MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,  
            void *recvbuf, int recvcount, MPI_Datatype recvtype,  
            int root, MPI_Comm comm)
```

- **sendbuf** must be large enough in order to supply **sendcount** elements of data to **each** rank in the communicator
- Data chunks are taken in **increasing rank order** following
- **root** also sends one data chunk to itself
- **Type signatures** of must match across all ranks (→ Datatypes)

# Scatter (one-to-many data distribution)

- Distribute chunks of data from one rank to all ranks:

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



# Scatter (one-to-many data distribution) – Example

- Distribute chunks of data from one rank to all ranks:

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

- **sendbuf** is only accessed on the root rank
- **recvbuf** is written into in all ranks

- Example:

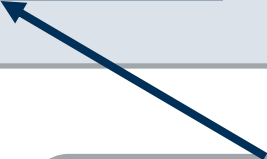
```
int bigdata[100];           // 10x10 elements  
int localdata[10];  
  
MPI_Scatter(bigdata, 10, MPI_INT,           // send buffer, root only  
            localdata, 10, MPI_INT,       // receive buffer  
            0, MPI_COMM_WORLD);
```

# Gather (many-to-one data distribution)

- Collect chunks of data from all ranks in one place:

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

- The inverse operation to MPI\_Scatter
- **recvbuf** must be large enough to hold **recvcount** elements from each rank
- **root** also receives one data chunk from itself
- Data chunks are stored in **increasing order** of the sender's rank
- Type signature of **sendcount** and **sendtype** must match **recvcount** and **recvtype**

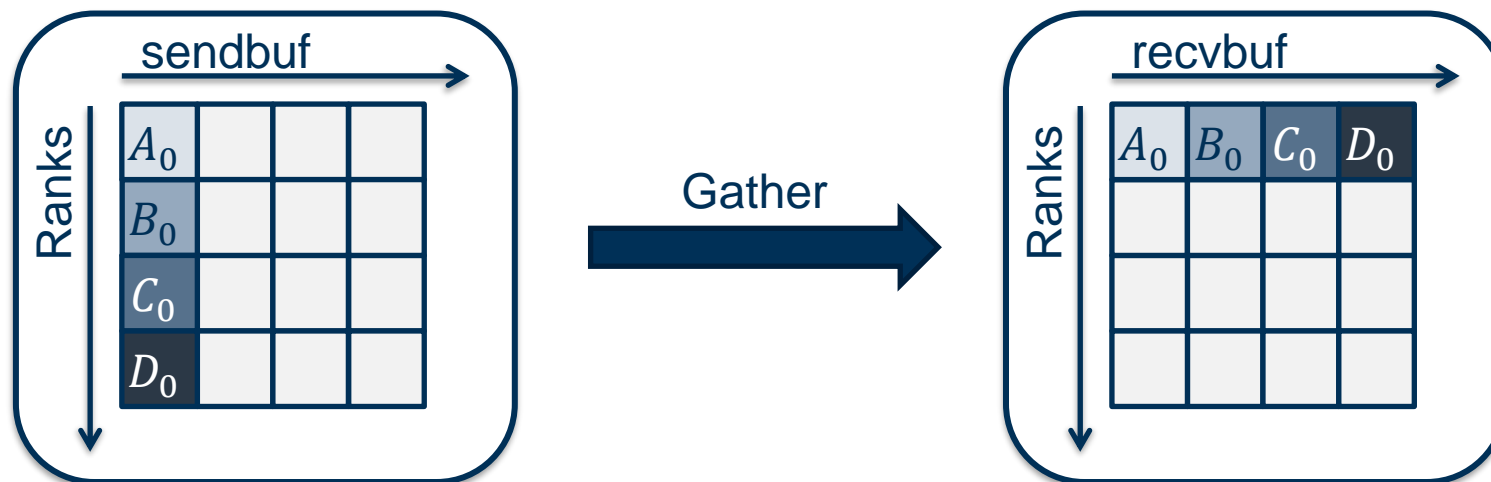


Significant at root rank only

# Gather (many-to-one data distribution)

- Collect chunks of data from all ranks in one place:

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





# Allgather (many-to-many data distribution)

- Collect chunks of data from all ranks in all ranks:

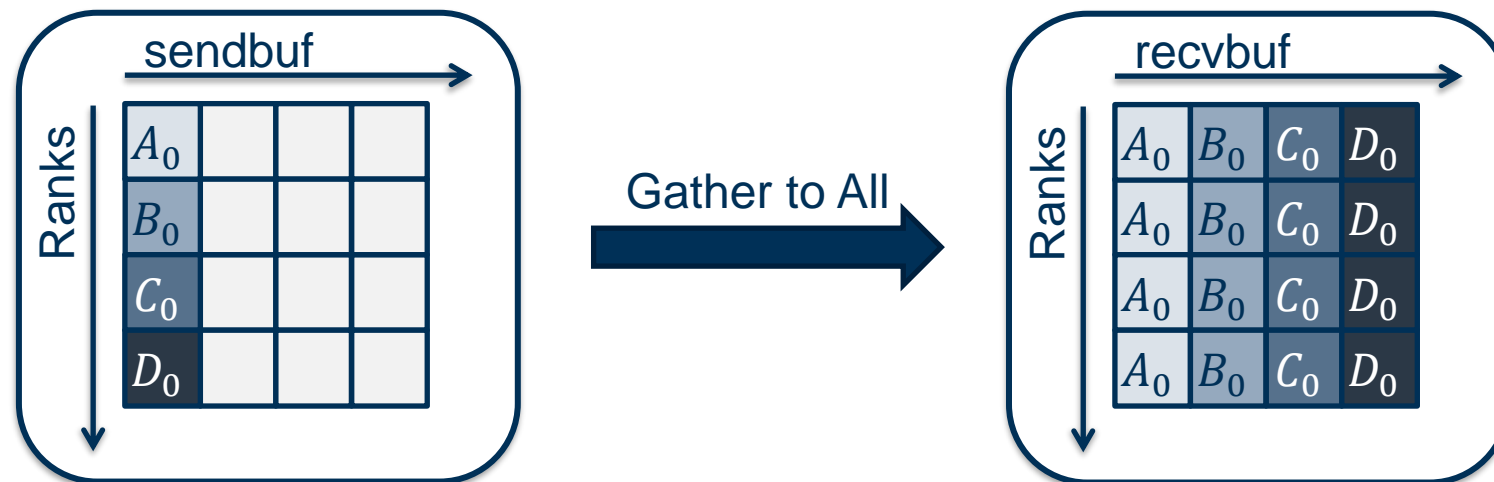
```
MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype,  
              void *recvbuf, int recvcount, MPI_Datatype recvtype,  
              MPI_Comm comm)
```

- No **root** rank – all ranks receive a copy of the gathered data
- Each rank also receives one data chunk from itself
- Data chunks are stored in **increasing order** of sender's rank
- **Type signatures of must match across all ranks (→ Datatypes)**
- Logically equivalent to **MPI\_Gather + MPI\_Bcast**, but potentially more efficient

# Allgather (many-to-many data distribution)

- Collect chunks of data from all ranks in all ranks:

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



- Combined scatter and gather operation:

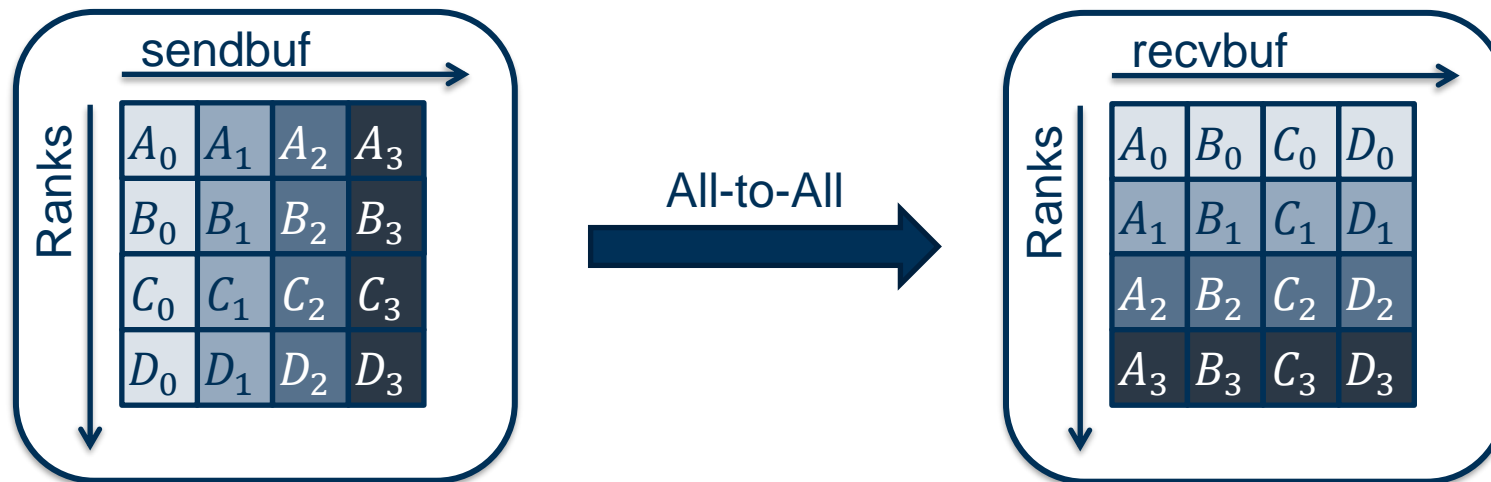
```
MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype,  
             void *recvbuf, int recvcount, MPI_Datatype recvtype,  
             MPI_Comm comm)
```

- Each rank distributes its **sendbuf** to every rank in the communicator (including itself)
- Data chunks are **read** in increasing order of the receiver's rank
- Data chunks are **stored** in increasing order of the sender's rank
- Almost equivalent to multiple **MPI\_Scatter + MPI\_Gather**

# All-to-All (many-to-many data distribution)

- Combined scatter and gather operation:

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



- Perform an arithmetic reduction operation while gathering data

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

- **sendbuf:** data to be reduced
- **recvbuf:** location for the result(s) (significant at root only)
- **count/datatype:** type signature of data
- **op:** reduction operation handle
- **root:** destination rank
- **comm:** communicator
- Result is computed in- or out-of-order depending on the operation:
  - All predefined operations are *associative* and *commutative*
  - Beware of non-commutative effects on floats

- Some predefined operation handles:

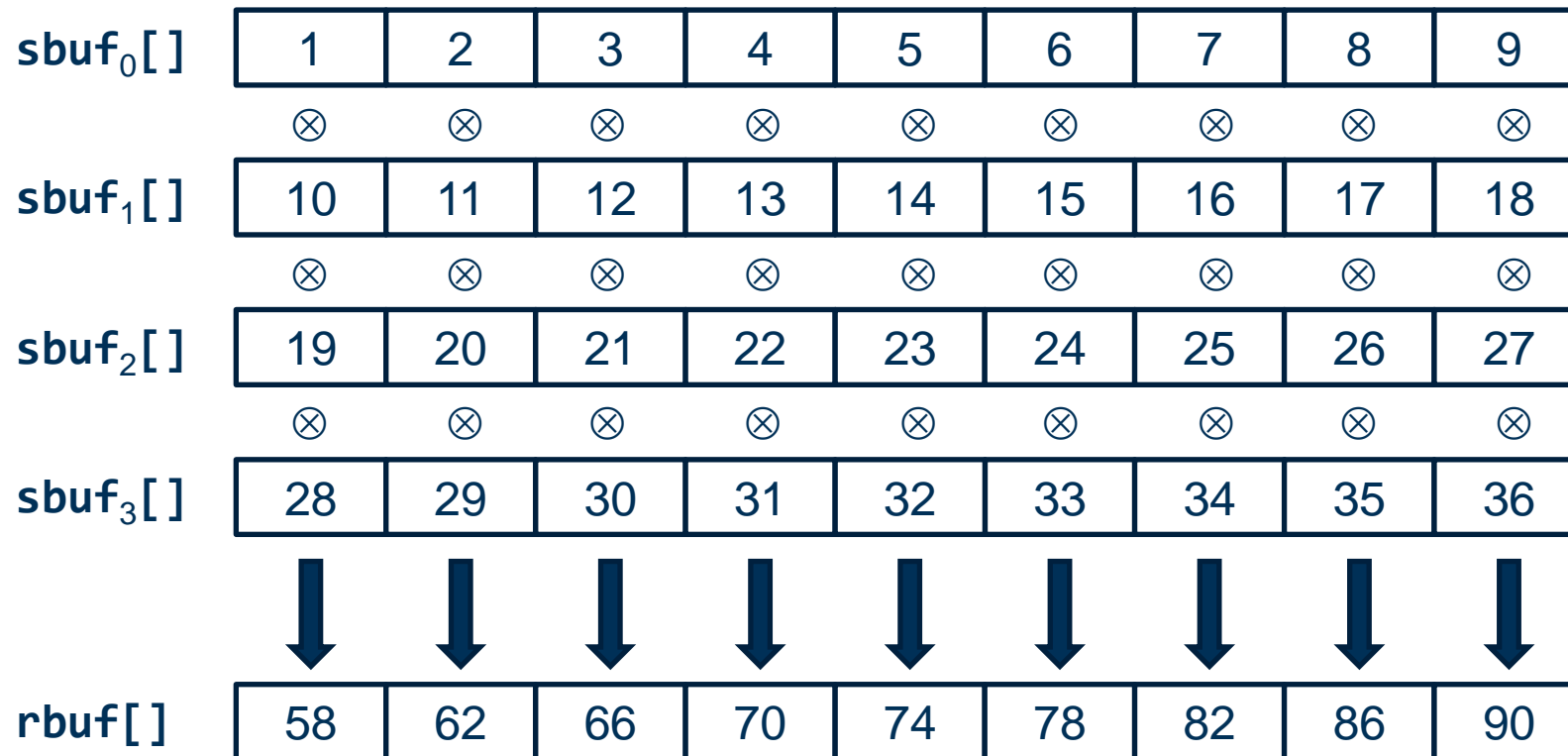
MPI_Op	Result value
MPI_MAX	Maximum value
MPI_MIN	Minimum value
MPI_SUM	Sum of all values
MPI_PROD	Product of all values
MPI_LAND	Logical AND of all values
MPI_BAND	Bit-wise AND of all values
MPI_LOR	Logical OR of all values
...	...

- User-define operators possible (not covered here)

# Reduce (many-to-one data reduction) – Example

– Element-wise and cross-rank operation

–  $rbuf[i] = sbuf_0[i] \text{ op } sbuf_1[i] \text{ op } sbuf_2[i] \text{ op } \dots \text{ op } sbuf_{nranks-1}[i]$



⊗ = MPI\_SUM

# Allreduce (many-to-many data reduction)

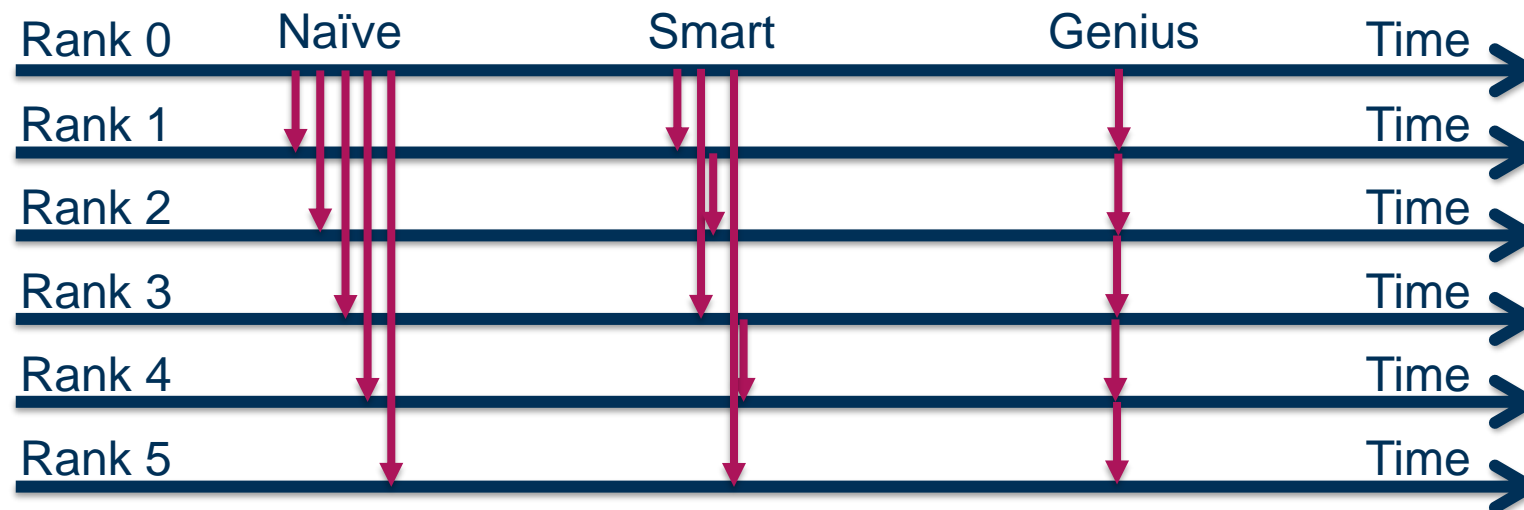
- Reduction result available on all ranks:

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

- Logically equivalent to **MPI\_Reduce** + **MPI\_Bcast** with the same root



- Collective operations implement common SPMD patterns portably
- Platform/Vendor-specific implementation, but standard behaviour
- Example: Broadcast
  - Naïve: root sends separate message to every other rank,  $O(\#\text{ranks})$
  - Smart: tree-based hierarchical communication,  $O(\log(\#\text{ranks}))$
  - Genius: pipelined segmented transport,  $O(1)$



- All ranks in the communicator must call the MPI collective operation
  - Both, data sources and data receivers have to make the same call and supply the same value for the root rank where needed
  - Observe the significance of each argument
- The sequence of collective calls must be the same on all ranks
- MPI\_Barrier is the only explicitly synchronising MPI collective
  - Some may synchronize implicitly (e.g., Allgather, Allreduce)
- Communication paradigms are independent of each other
  - Collective communication does not interfere with point-to-point communication on the same communicator