Message Passing with MPI

PPCES 2016

Hristo Iliev

IT Center / JARA-HPC

Agenda

- **Motivation**
- **Part 1**
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- **Part 2**
  - → Collective operations
  - → Communicators
  - → User datatypes
- **Part 3**
  - → Hybrid parallelisation
  - → Common parallel patterns

Collective Operations

- **MPI collective operations involve all ranks in a given communication context (communicator) at the same time**

- **All ranks in the communicator must make the same MPI call for the operation to succeed**
  NB: There should be only one call per MPI rank (i.e. not per thread)

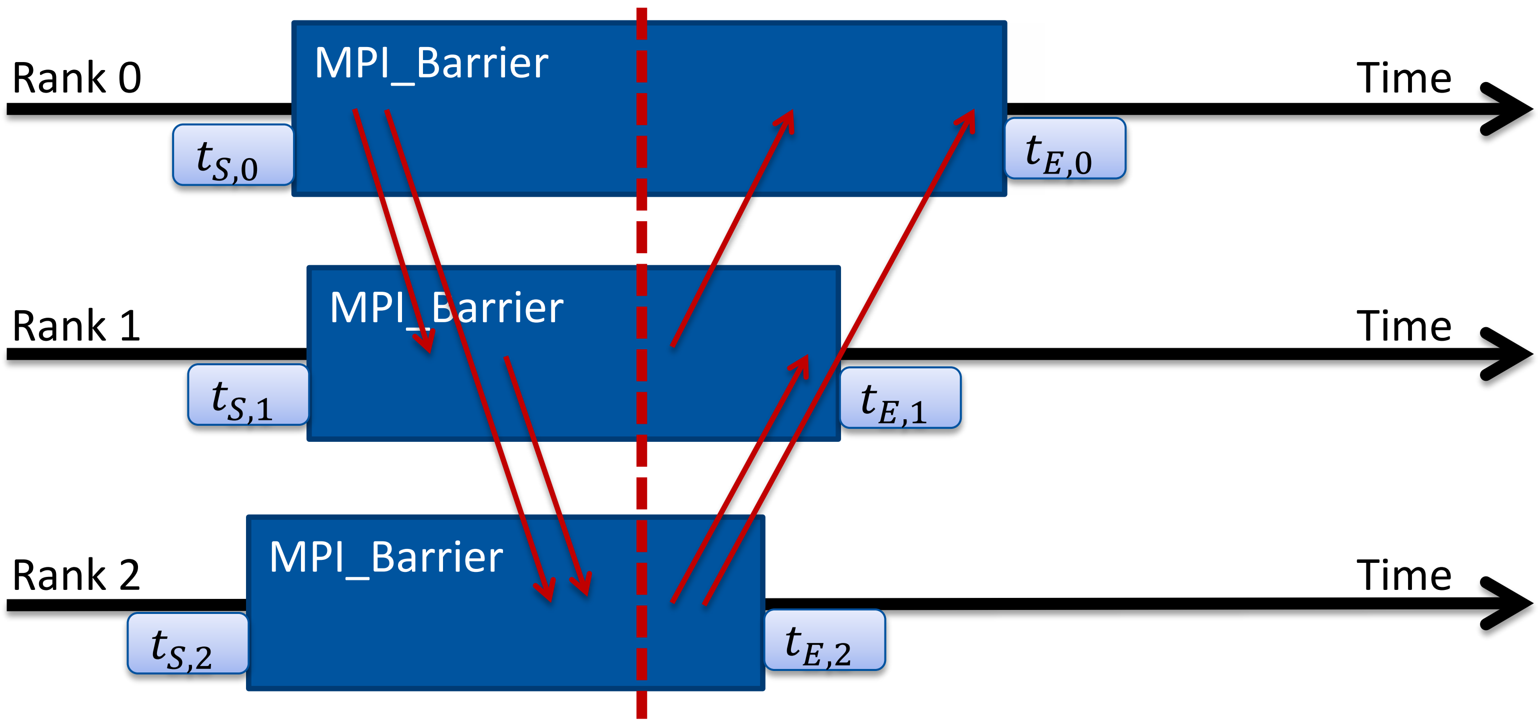
- **Some collective operations are globally synchronous**
  → The MPI standard allows for early return in some ranks

- **Collective operations are provided as convenience to the end user and can be (and often are) implemented with basic point-to-point communication primitives**
  → But they are usually tuned to deliver the best system performance

Barrier Synchronisation

- **The only explicit synchronisation operation in MPI:**

```
MPI_Barrier (MPI_Comm comm)
```

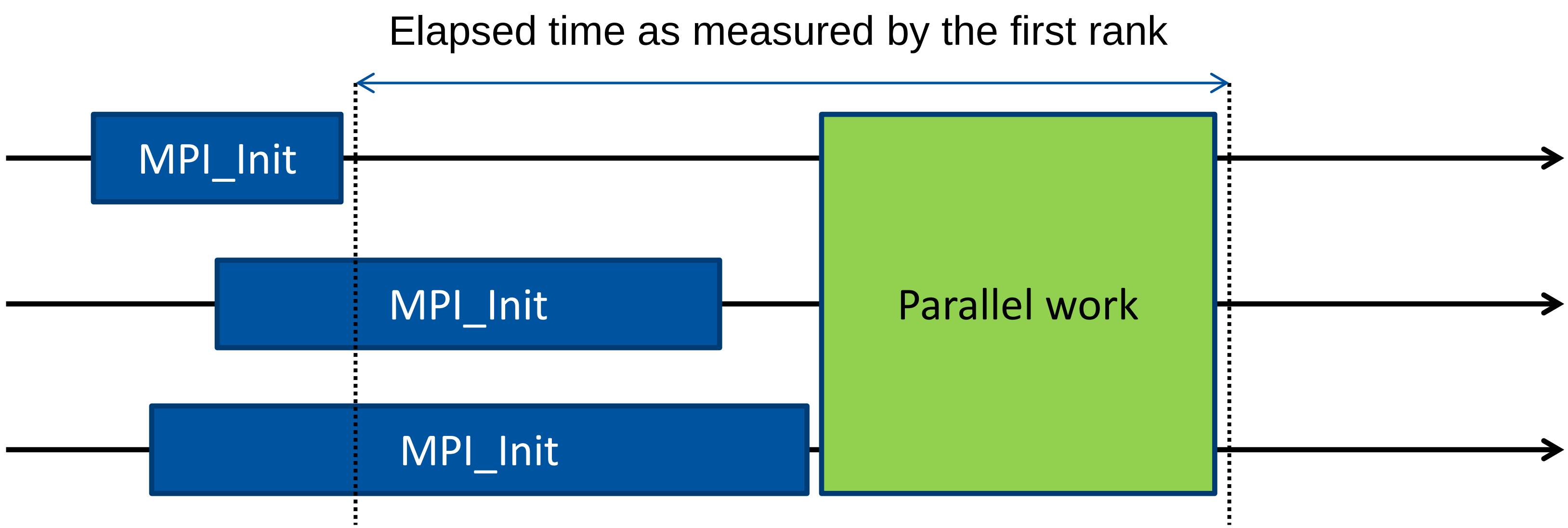


$$\max(t_{S,0}; t_{S,1}; t_{S,2}) < \min(t_{E,0}; t_{E,1}; t_{E,2})$$

Barrier Synchronisation

Useful for benchmarking

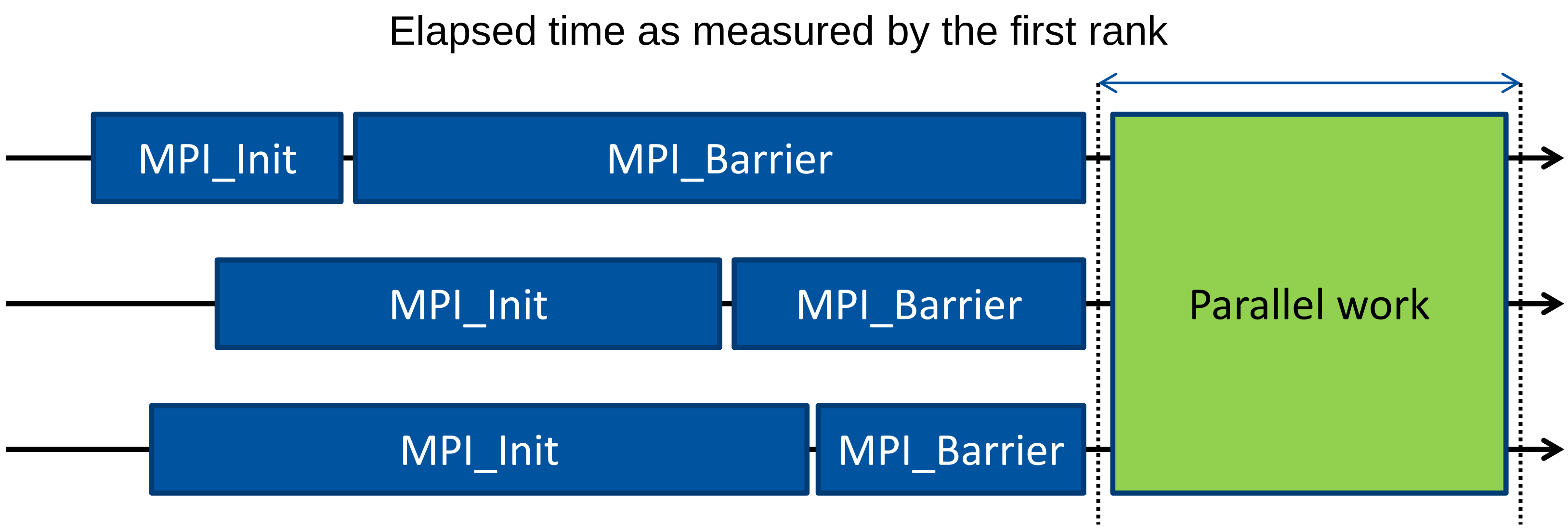
→ Always synchronise before taking time measurements



→ Huge discrepancy between the actual work time and the measurement

Barrier Synchronisation

- **Useful for benchmarking**
  - → Always synchronise before taking time measurements



  - → Dispersion of the barrier exit times is usually quite low

Data Replication (Broadcast)

- **Replicate data from one rank to all other ranks:**

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

→ **data:** data to be sent at **root**; place to put the data in all other ranks
→ **count:** number of data elements
→ **dtype:** elements' datatype
→ **root:** source rank; all ranks <u>must</u> specify the same value
→ **comm:** communication context

- **Notes:**

→ in all ranks but **root**, **data** is an output argument

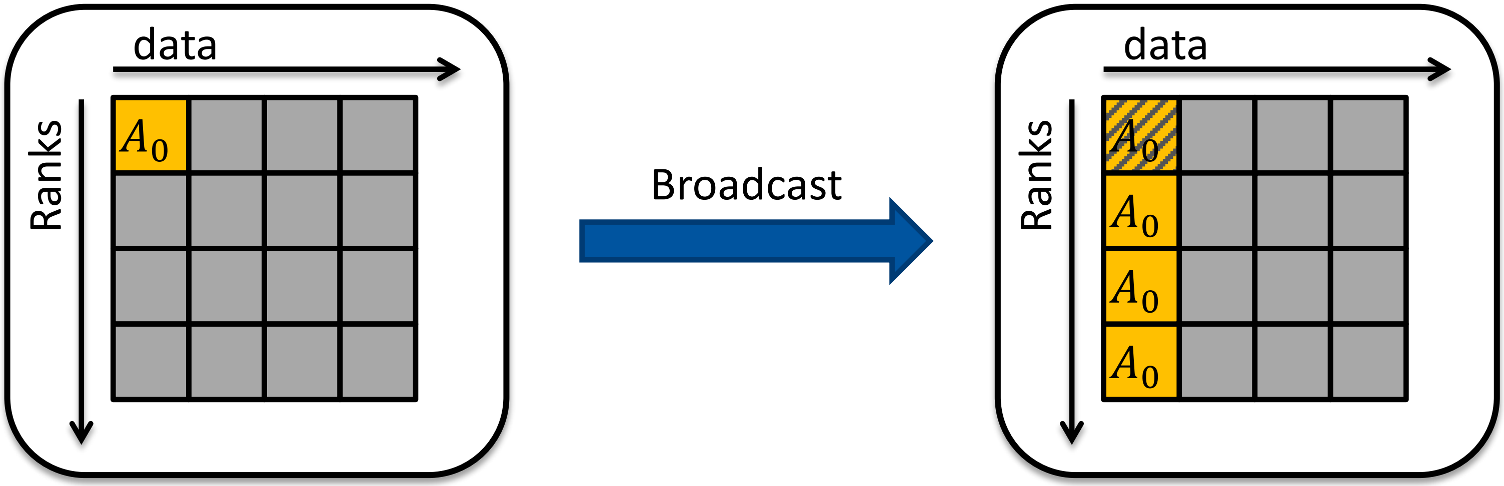
→ in rank **root**, **data** is an input argument

→ **MPI_Bcast** completes only after all ranks in **comm** have made the call

Data Replication (Broadcast)

- **Replicate data from one rank to all other ranks:**

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



Data Replication (Broadcast)

- **Replicate data from one rank to all other ranks:**

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

→ example use:

```
int ival;

if (rank == 0)
   ival = read_int_from_user();

MPI_Bcast(&ival, 1, MPI_INT, 0, MPI_COMM_WORLD);

// WRONG
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
```

Data Replication (Broadcast)

- **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);
}
```

Data Scatter

- **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

→ **sendbuf:** data to be distributed
→ **sendcount:** size of each chunk in data elements
→ **sendtype:** source datatype
→ **recvbuf:** buffer for data reception
→ **recvcount:** number of elements to receive
→ **recvtype:** receive datatype
→ **root:** source rank
→ **comm:** communication context

Data Scatter

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

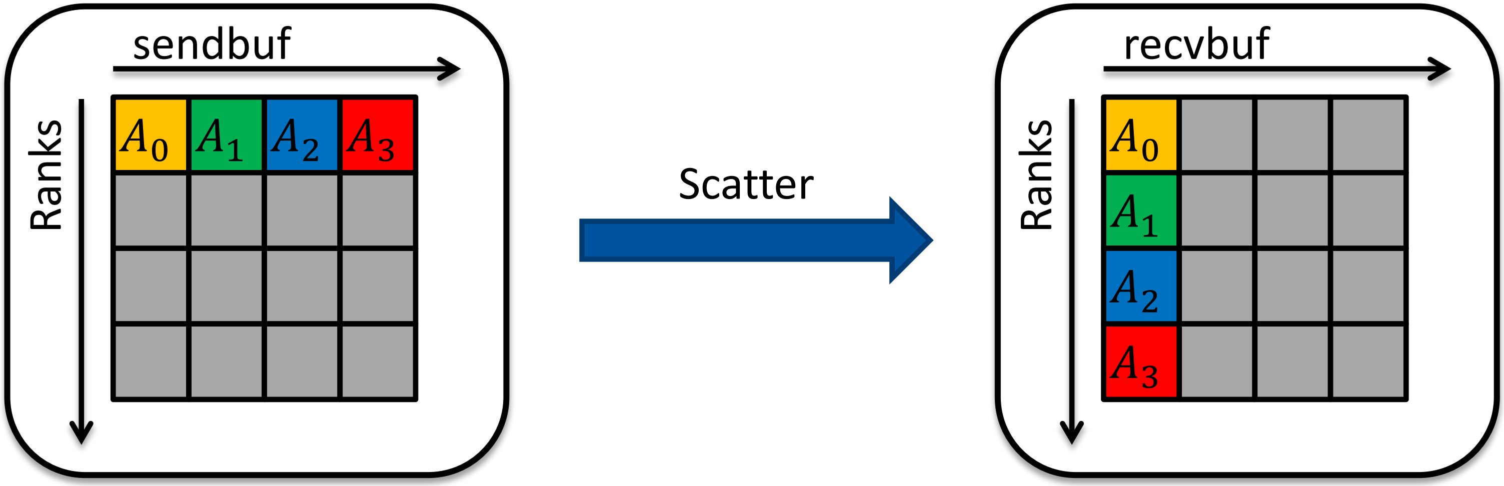
```
MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
       void *recvbuf, int recvcount, MPI_Datatype recvtype,
       int root, MPI_Comm comm)
```

- **Notes:**
  - → **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 order following the receiver's rank
  - → **root** also sends one data chunk to itself
  - → for each chunk the amount of data sent must match the receive size, i.e. if **sendtype == recvtype** holds, then **sendcount == recvcount** must hold too

Data Scatter

- 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)
```



Data Scatter

- **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 in the root rank

→ **recvbuf** is written into in all ranks

→ example use:

```
// Assume there are 10 MPI ranks
int bigdata[100];
int localdata[10];

MPI_Scatter(bigdata, 10, MPI_INT,      // send buffer, root only
            localdata, 10, MPI_INT,    // receive buffer
            0, MPI_COMM_WORLD);
```

Data Gather

- **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)
```

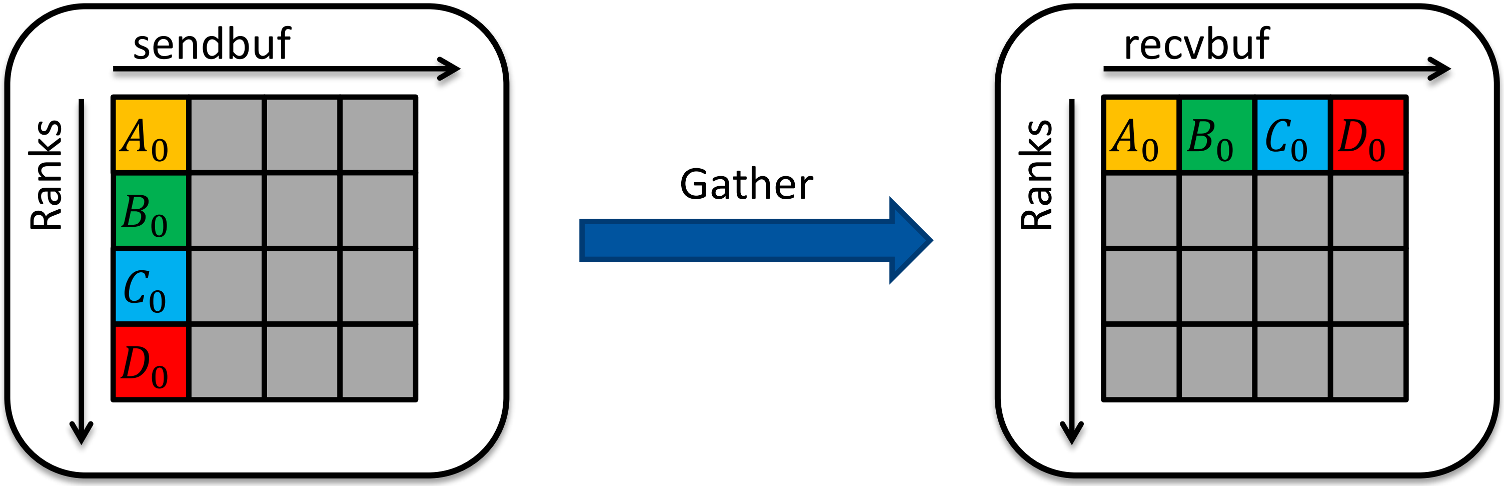
Significant at root rank only

- **The opposite operation of 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
  - → for each chunk the receive size must match the amount of data sent

Data Gather

- **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)
```



Gather-to-All

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

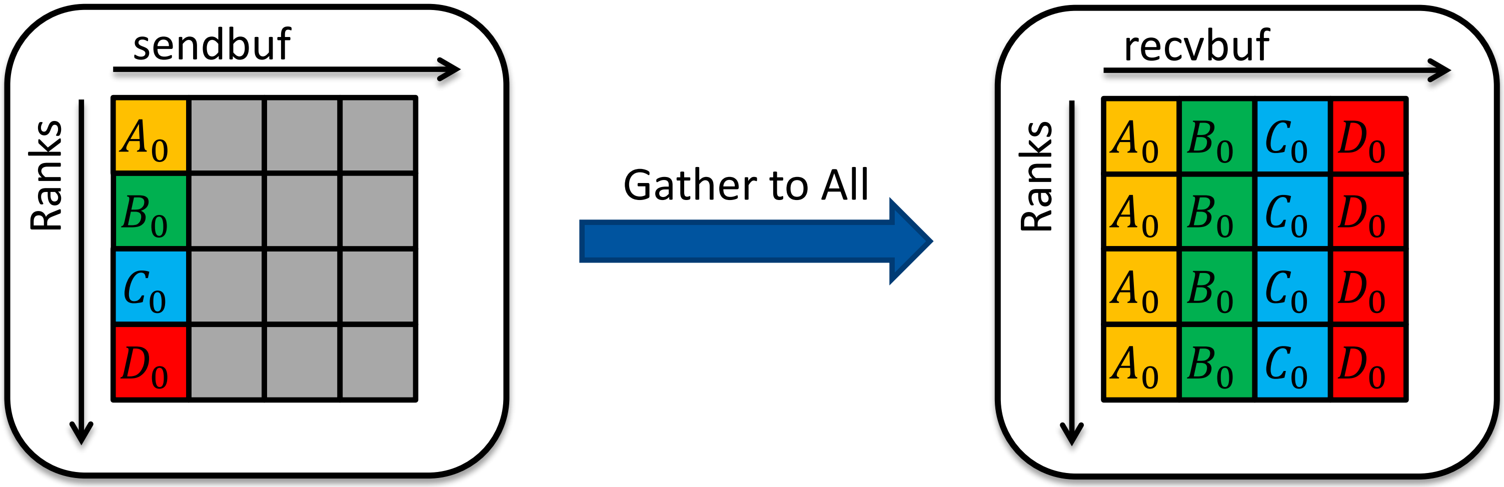
```
MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
       void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

- **Note:**
  - → 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
  - → for each chunk the receive size must match the amount of data sent
  - → equivalent to **MPI_Gather + MPI_Bcast**, but possibly more efficient

Gather-to-All

- **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)
```



All-to-All

- **Combined scatter and gather operation:**

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

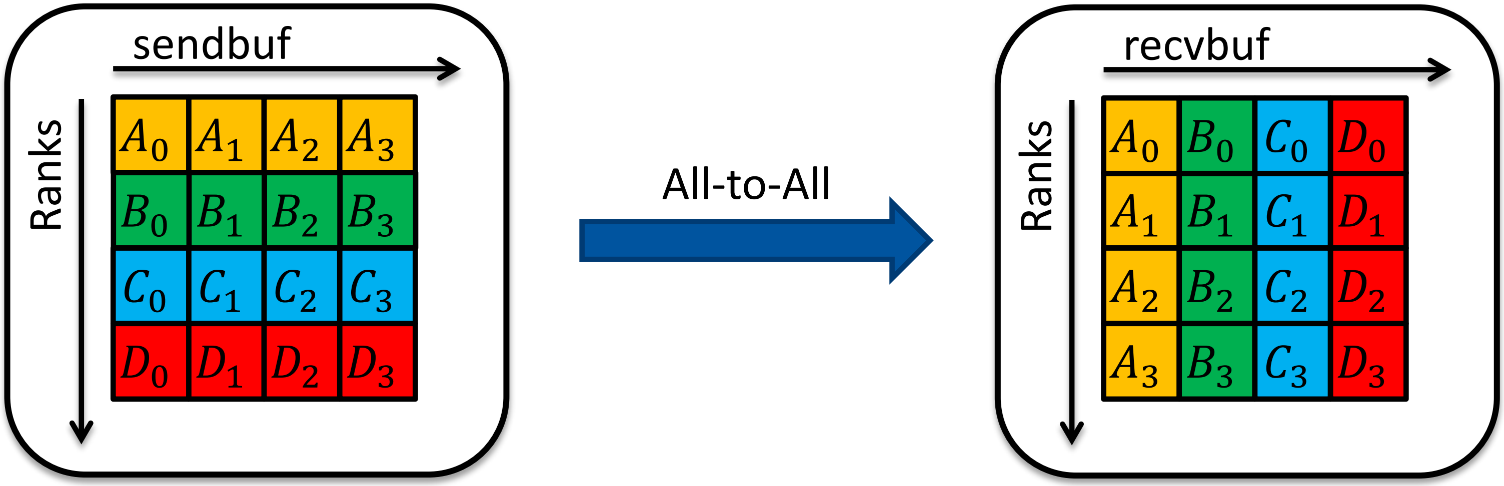
- **Notes:**
  - → no root rank – each rank distributes its **sendbuf** to every rank in the communicator (including itself)
  - → data chunks are taken in increasing order of the receiver's rank
  - → data chunks are stored in increasing order of the sender's rank
  - → almost equivalent to **MPI_Scatter + MPI_Gather**
    (one cannot mix data from separate collective operations)

All-to-All

- **Combined scatter and gather operation:**

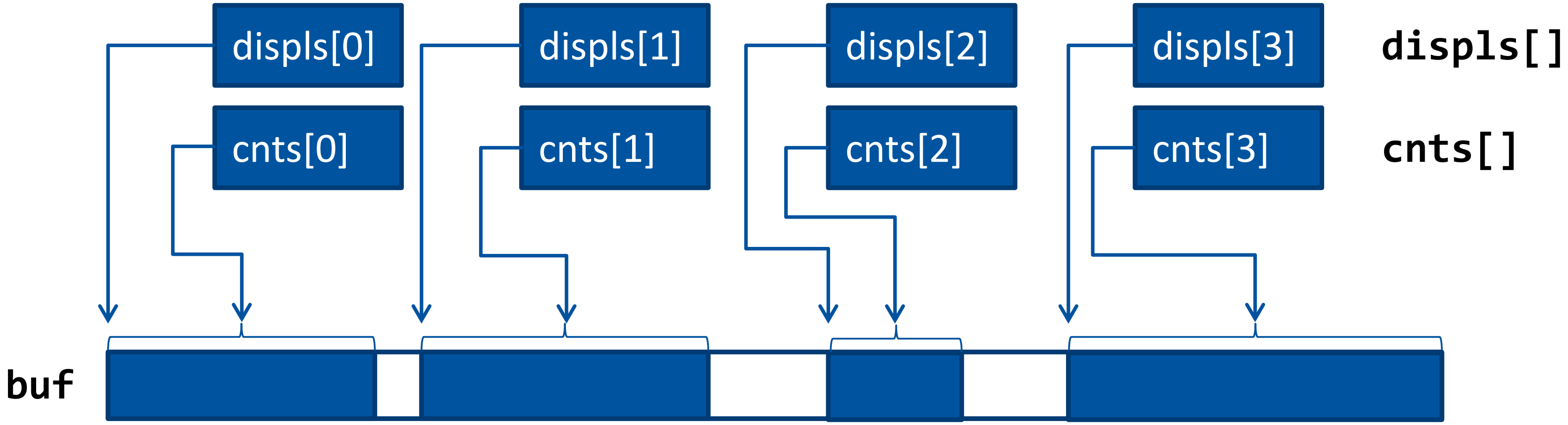
```
MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype,
       void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

- **Note: a kind of global chunked transpose**



Varying Counts

- **Position and length of each chunk can be explicitly specified with the so-called varying count (-v) versions**
  - → Displacement and count in units of data elements specified for each chunk



- **Useful when the problem size is not divisible by the number of MPI processes or when dealing with irregular domain decomposition**

Varying Counts

- Most collectives have varying count versions

```
MPI_Scatterv (void *sbuf, int *scnts, int *sdispls, MPI_Datatype stype,
       void *rbuf, int rcount, MPI_Datatype rtype,
        int root, MPI_Comm comm)
```

```
MPI_Gatherv (void *sbuf, int scount, MPI_Datatype stype,
       void *rbuf, int *rcnts, int *rdispls, MPI_Datatype rtype,
       int root, MPI_Comm comm)
```

```
MPI_Allgatherv (void *sbuf, int *snts, int *sdispls, MPI_Datatype stype,
       void *rbuf, int rcount, MPI_Datatype rtype,
        MPI_Comm comm)
```

```
MPI_Alltoallv (void *sbuf, int *scnts, int *sdispls, MPI_Datatype stype,
       void *rbuf, int *rcnts, int *rdispls, MPI_Datatype rtype,
        MPI_Comm comm)
```

Global Reduction

- **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:** number of data elements
→ **datatype:** element datatype
→ **op:** handle of the reduction operation
→ **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

Global reduction

- **Element-wise and cross-rank operation**

  → `rbuf[i] = sbuf`$_0$`[i]` op `sbuf`$_1$`[i]` op `sbuf`$_2$`[i]` op … `sbuf`$_{\text{nranks-1}}$`[i]`

sbuf$_0$[]	1	2	3	4	5	6	7	8	9
sbuf$_1$[]	10	11	12	13	14	15	16	17	18
sbuf$_2$[]	19	20	21	22	23	24	25	26	27
sbuf$_3$[]	28	29	30	31	32	33	34	35	36
rbuf[]	58	62	66	70	74	78	82	86	90

$\otimes$ = `MPI_SUM`

Global Reduction

- **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
…	…

- **Users can create their own reduction operations, but that goes beyond the scope of the course**

Global Reduction

- **Perform an arithmetic reduction and broadcast the result:**

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

- **Notes:**
  - → no root rank – every rank receives the result of the reduction operation
  - → equivalent to **MPI_Reduce + MPI_Bcast** with the same root
  - → can be slower with non-commutative operations because of the forced in-order execution (the same applies to **MPI_Reduce**)
    - → concerns non-commutative user-defined operations only

Collective Calls

- **All ranks in the communicator must call the MPI collective operation for it to complete successfully:**
  - → both data sources (root) 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

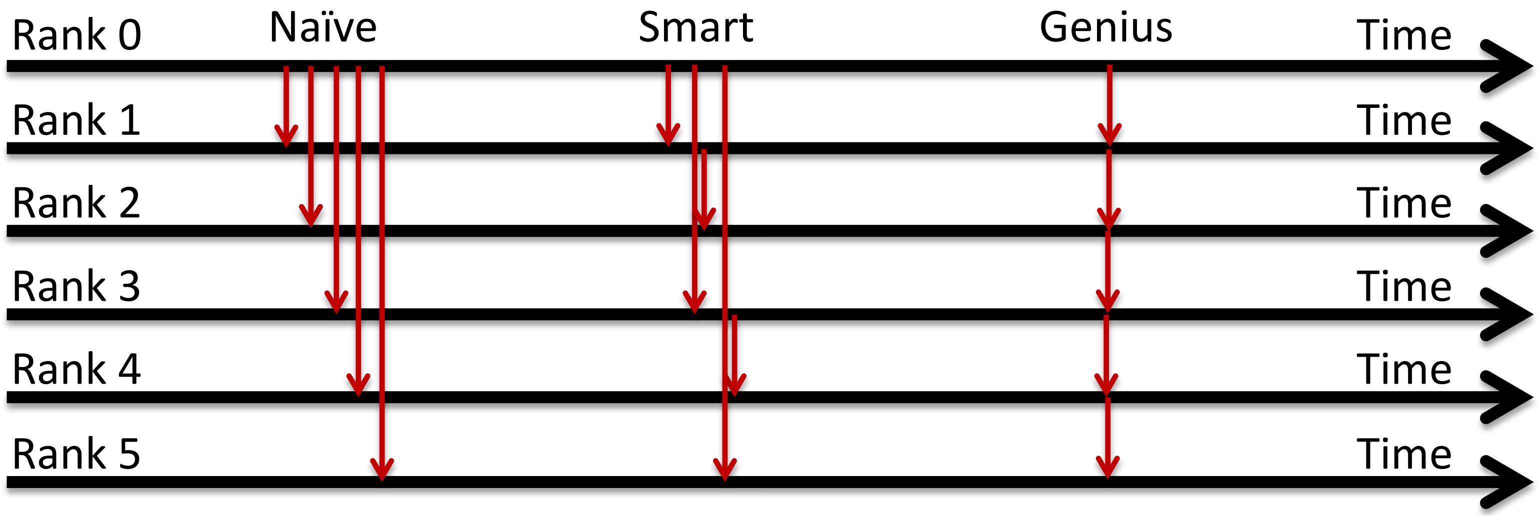
- **The sequence of collective calls must be the same in all ranks**

- **MPI_Barrier is the only true synchronising MPI**

- **One cannot use MPI_Recv to receive data sent by MPI_Scatter / MPI_Alltoall**
- **One cannot use MPI_Send to send data to MPI_Gather / MPI_Allgather / MPI_Alltoall**

Advantages

- **Collective operations implement portably common SPMD patterns**
- **Implementation-specific magic, but standard behaviour**
- **Example: Broadcast**
  - → Naïve: root sends separate message to every other rank, O(#ranks)
  - → Smart: tree-based hierarchical communication, O(log(#ranks))
  - → Genius: pipelined segmented transport, O(1)



Collective Operations

?

Agenda

- Motivation
- Part 1
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- **Part 2**
  - → Collective operations
  - → Communicators
  - → User datatypes
- Part 3
  - → Hybrid parallelisation
  - → Common parallel patterns

Communication Contexts

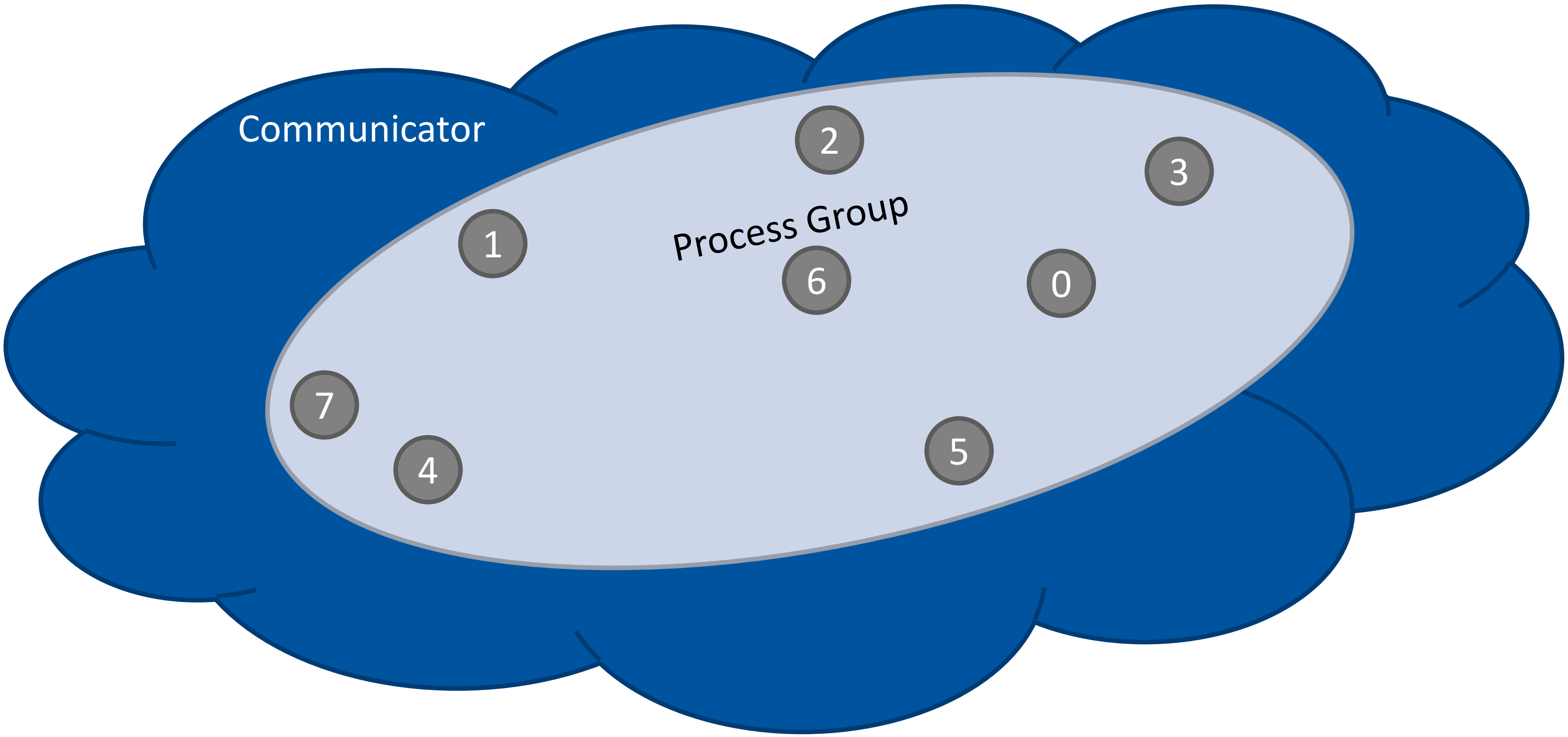
- **Each communication operation in MPI happens in a certain context:**
  - → Group of participating peers (process group)
  - → Error handlers for communication and I/O operations
  - → Local key/value store
  - → Optional virtual topology
- **MPI always provides two predefined contexts:**
  - → **MPI_COMM_WORLD**

    contains all processes launched **initially** as part of the MPI program
  - → **MPI_COMM_SELF**

    contains only the current process
- **A unique communication endpoints consists of a communicator handle and a rank from that communicator**

Communicators

- **Communicator – process group – ranks**



Query Operations

- **Obtain the size of the process group of a given communicator:**

```
MPI_Comm_size (MPI_Comm comm, int *size)
```

→ ranks in the group are numbered from 0 to size-1

- **Obtain the rank of the calling process in the given communicator:**

```
MPI_Comm_rank (MPI_Comm comm, int *rank)
```

- **Special “null” rank – MPI_PROC_NULL**

→ member of any communicator

→ can be sent messages to – results in a no-op

→ can be received messages from – zero-size message tagged **MPI_ANY_TAG**

→ use it to write symmetric code and handle process boundaries

Message Envelope Matching

- **Recall: message envelope**

	Sender	Receiver
Source	Implicit	Explicit, wildcard possible (**MPI_ANY_SOURCE**)
Destination	Explicit	Implicit
Tag	Explicit	Explicit, wildcard possible (**MPI_ANY_TAG**)
Communicator	Explicit	Explicit

- **Cross-communicator messaging is not possible**
  - → messages sent in one communicator can only be received by ranks in the same communicator
  - → communicators can be used to isolate communication to prevent interference and tag clashes – useful when writing parallel libraries

- **Simple flat addressing using MPI_COMM_WORLD often suffices**

Virtual Topologies

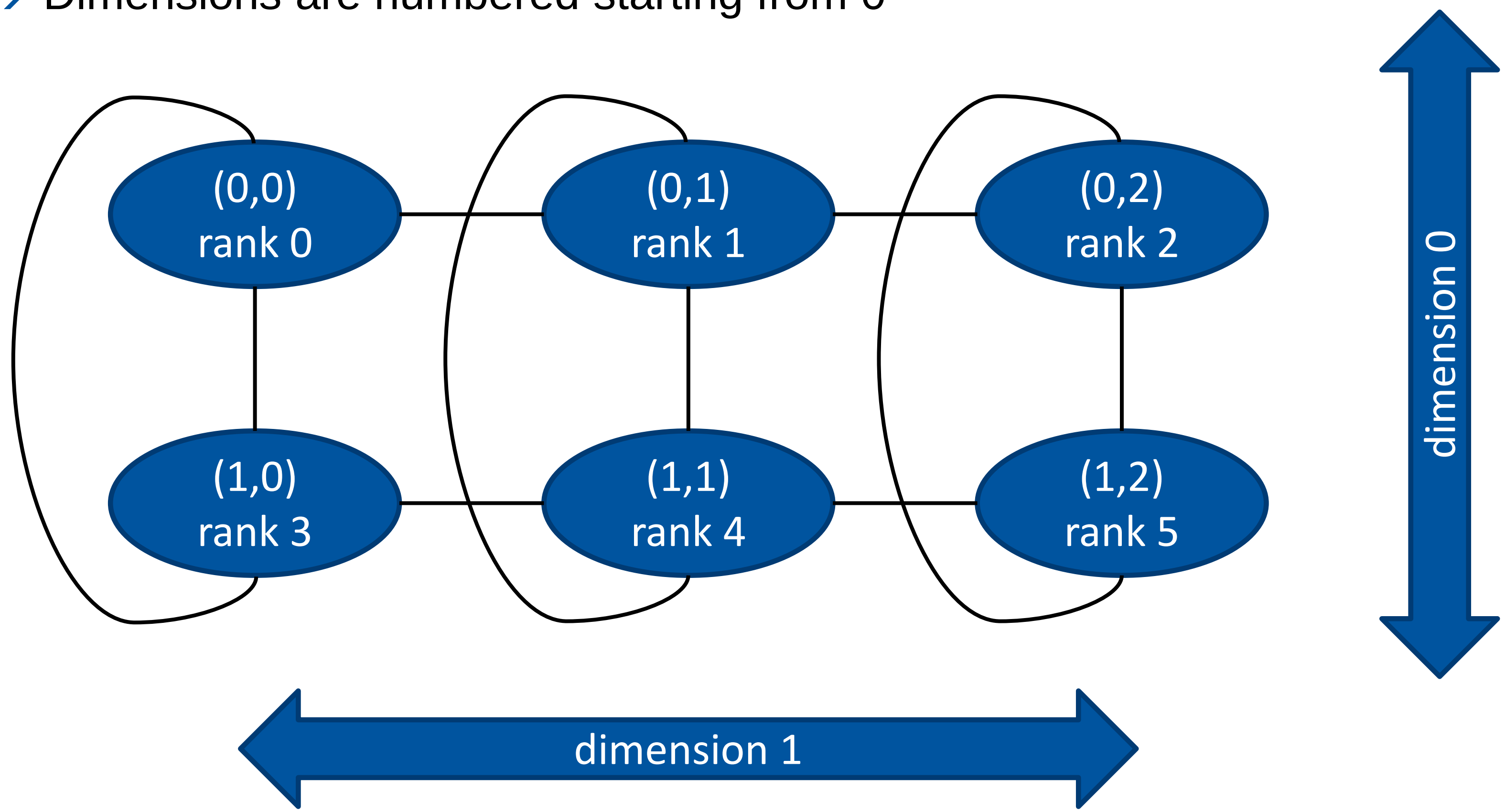
- **Each communicator can have an associated topology**
  → Mapping between ranks and abstract addresses
  → Virtual connectivity (neighbour links) information

- **Three different topology kinds:**
  → no topology – e.g. **MPI_COMM_WORLD**
  → Cartesian topology – regular *n*-dimensional grid
  → graph topology – general connectivity graph

- **Not having a neighbour link in the topology does not prevent ranks from communicating with each other**

Cartesian Topology

- **Regular n-dimensional grid**
  - → Dimensions are numbered starting from 0



Construct a Cartesian topology

```
MPI_Cart_create (MPI_Comm old_comm, int ndims, int dims[], int periods[],
                 int reorder, MPI_Comm *comm_cart)
```

→ Creates a new communicator **comm_cart** from the process group of **old_comm** with an **ndims**-dimensional Cartesian topology attached

→ **dims[]** – specifies the number of nodes in each dimension

→ **periods[]** – specifies the periodicity of each dimension (boolean array)

→ **reorder** – if set to true (non-zero), hints the MPI runtime to reorder the ranks in the new communicator so that their virtual connectivity matches as closely as possible the physical one; otherwise ranks are kept

Balanced Cartesian Distribution

- **Create a balance distribution of a number of processes**

```
MPI_Dims_create (int nnodes, int ndims, int dims[])
```

→ Computes the most balanced way to arrange **nnodes** ranks into an **ndims**-dimensional grid

→ Non-zero elements of **dims** specify the number of nodes in the corresponding dimension

→ Zero elements are filled with the optimal number of nodes in the corresponding dimension

→ Error if the product of non-zero elements of **dims** does not divide **nnodes**

Balanced Cartesian Distribution

- **Create a balance distribution of a number of processes**

```
MPI_Dims_create (int nnodes, int ndims, int dims[])
```

→ Factors **nnodes / product({1} U {non-zero elements of dims})**

→ The computed sizes are set in non-increasing order

  →the lowest-numbered dimension receives the biggest size

→ Example (taken from the MPI standard):

dims before call	function call	dims on return
(0,0)	`MPI_Dims_create(6, 2, dims)`	(3,2)
(0,0)	`MPI_Dims_create(7, 2, dims)`	(7,1)
(0,3,0)	`MPI_Dims_create(6, 3, dims)`	(2,3,1)
(0,3,0)	`MPI_Dims_create(7, 3, dims)`	erroneous call

Coordinate Conversion

- **Translate Cartesian coordinate tuples into ranks**

```
MPI_Cart_rank (MPI_Comm comm, int coords[], int *rank)
```

→ **comm** – Cartesian communicator

→ **coords** – an array of at least **ndims** elements – Cartesian coordinates

→ **rank** – corresponding process rank in **comm**

- **Translate ranks into Cartesian coordinate tuples**

```
MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims, int coords[])
```

→ **coords** – an array of **maxdims** elements to receive the coordinates

→ **maxdims** should be equal to or larger than **ndims**

Cartesian Shift

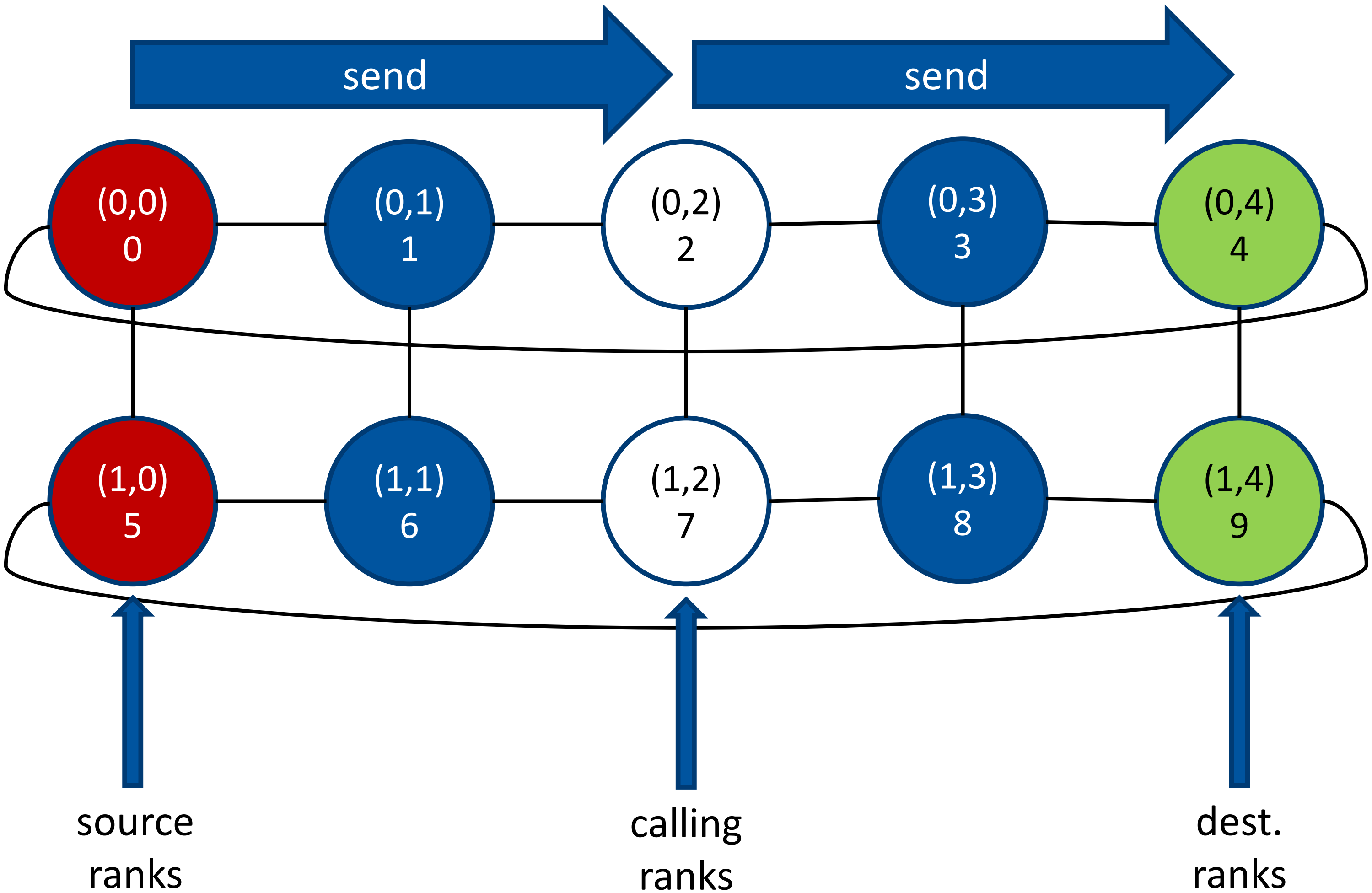
Find ranks of neighbour processes

```
MPI_Cart_shift (MPI_Comm comm, int dir, int disp, int *source, int *dest)
```

- → Computes the ranks of neighbours to communicate with in order to perform data shift (e.g. using **MPI_Sendrecv**) at distance of **disp** in direction **dir**
- → Equivalent to:
  - → obtain the Cartesian coordinates of the calling process
  - → translate (…, $\text{coord}_{\text{dir}}$ + disp, …) into rank **dest**
  - → translate (…, $\text{coord}_{\text{dir}}$ – disp, …) into rank **source**
- → If the source or the destination lies beyond a non-periodic boundary, the corresponding rank is returned as **MPI_PROC_NULL**

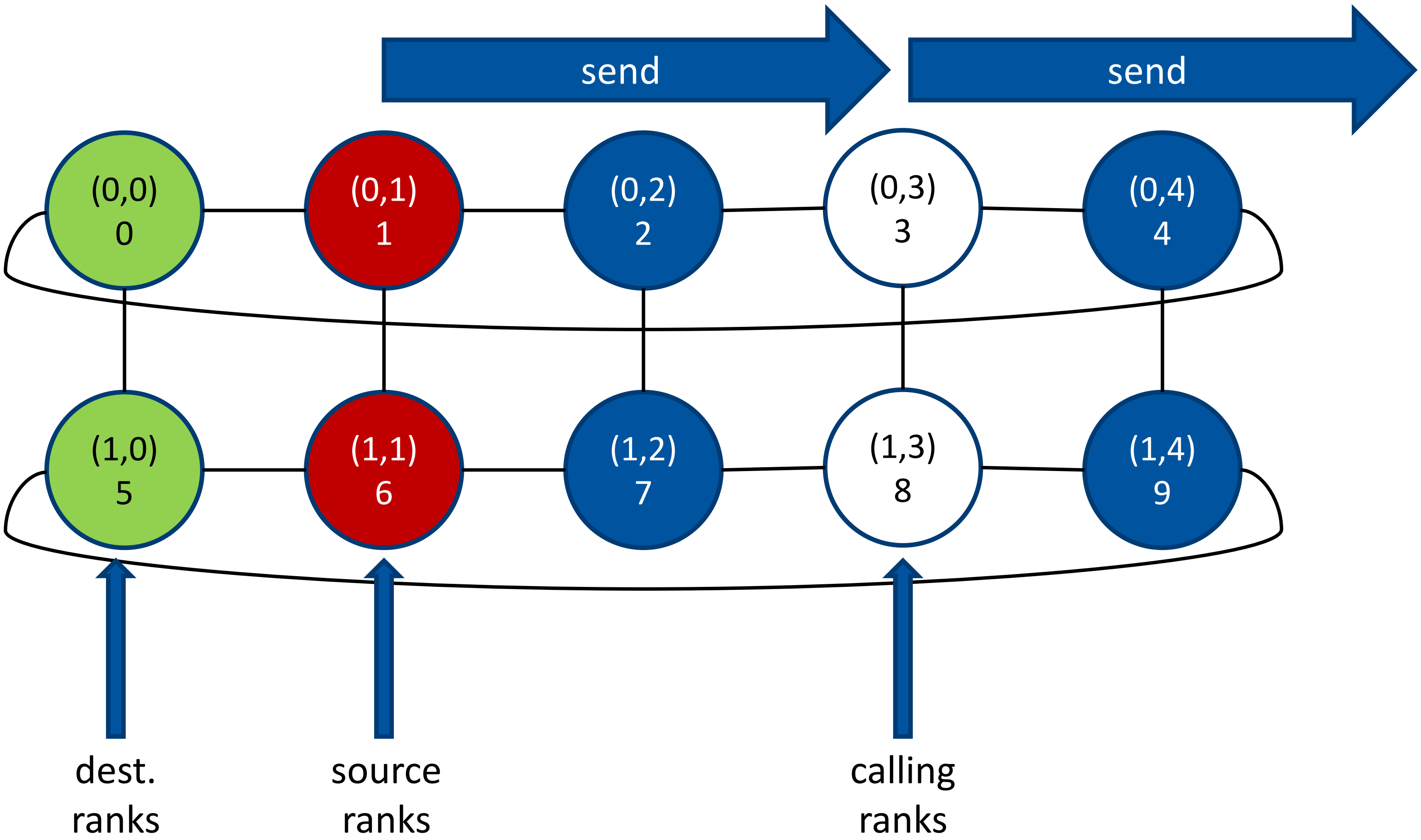
Cartesian Shift

- **Example: periodic boundary (dir = 1, disp = 2)**



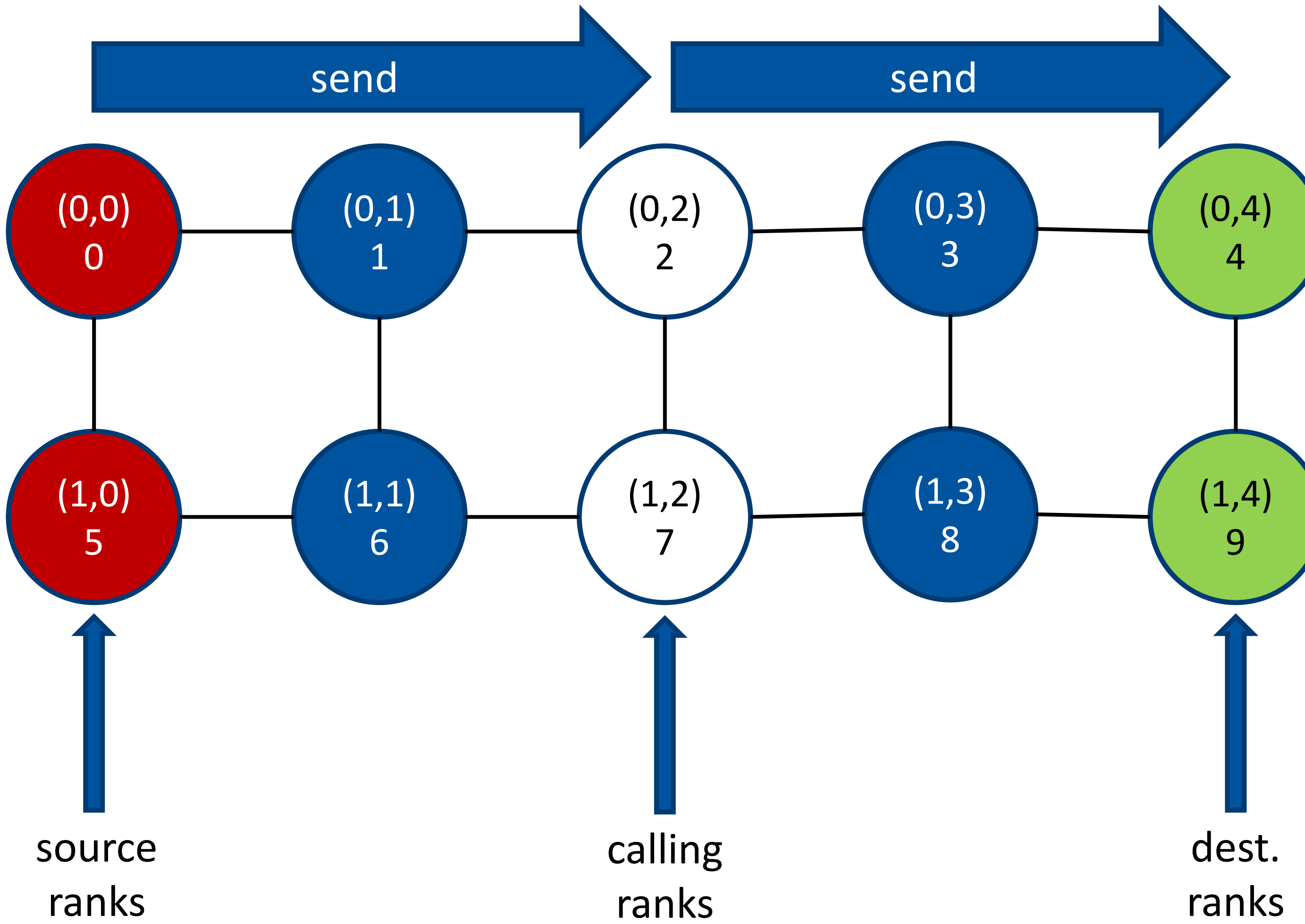
Cartesian Shift

- **Example: periodic boundary (dir = 1, disp = 2)**



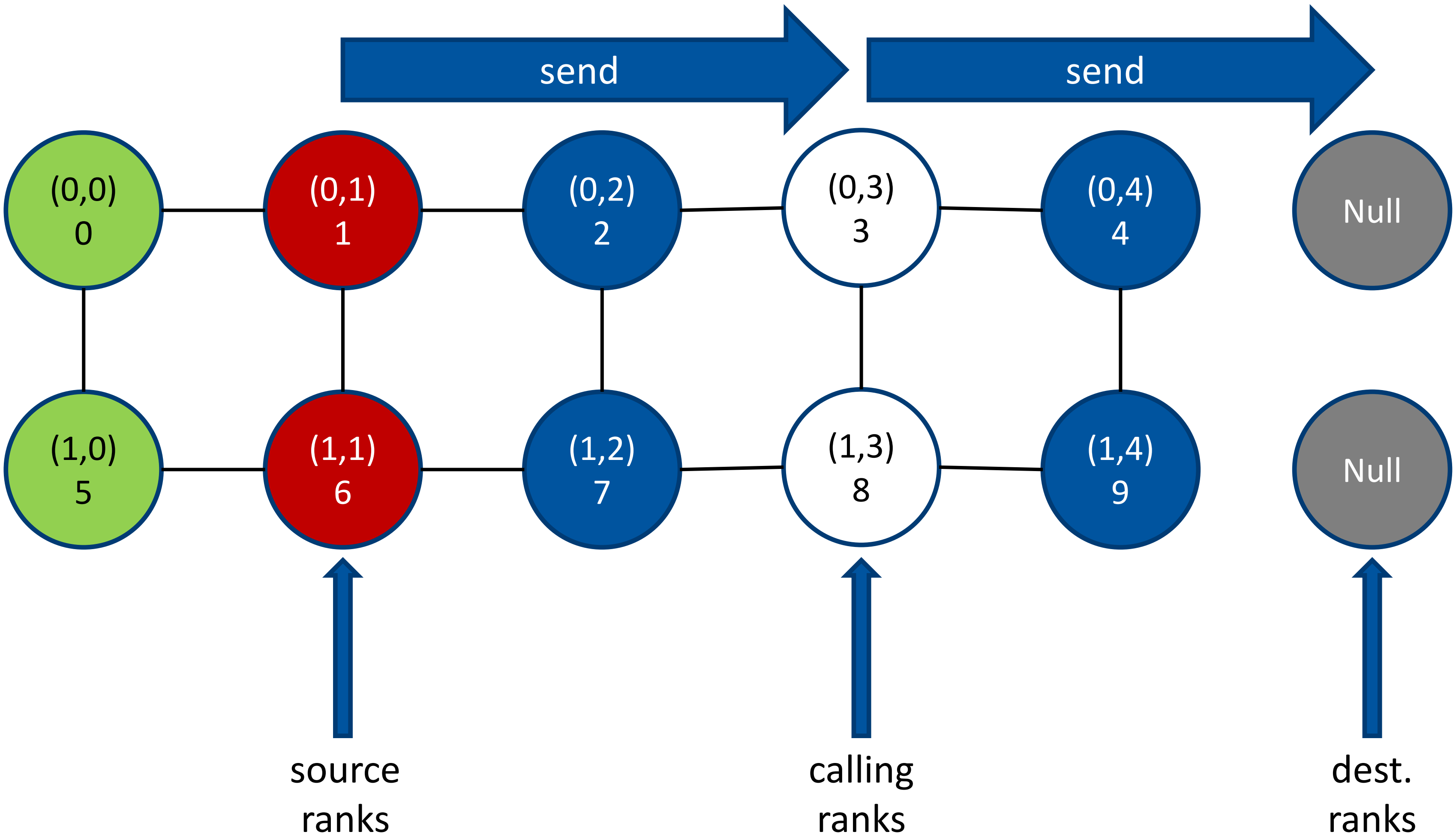
Cartesian Shift

- **Example: non-periodic boundary (dir = 1, disp = 2)**



Cartesian Shift

- Example: non-periodic boundary (dir = 1, disp = 2)



Cartesian Partitioning

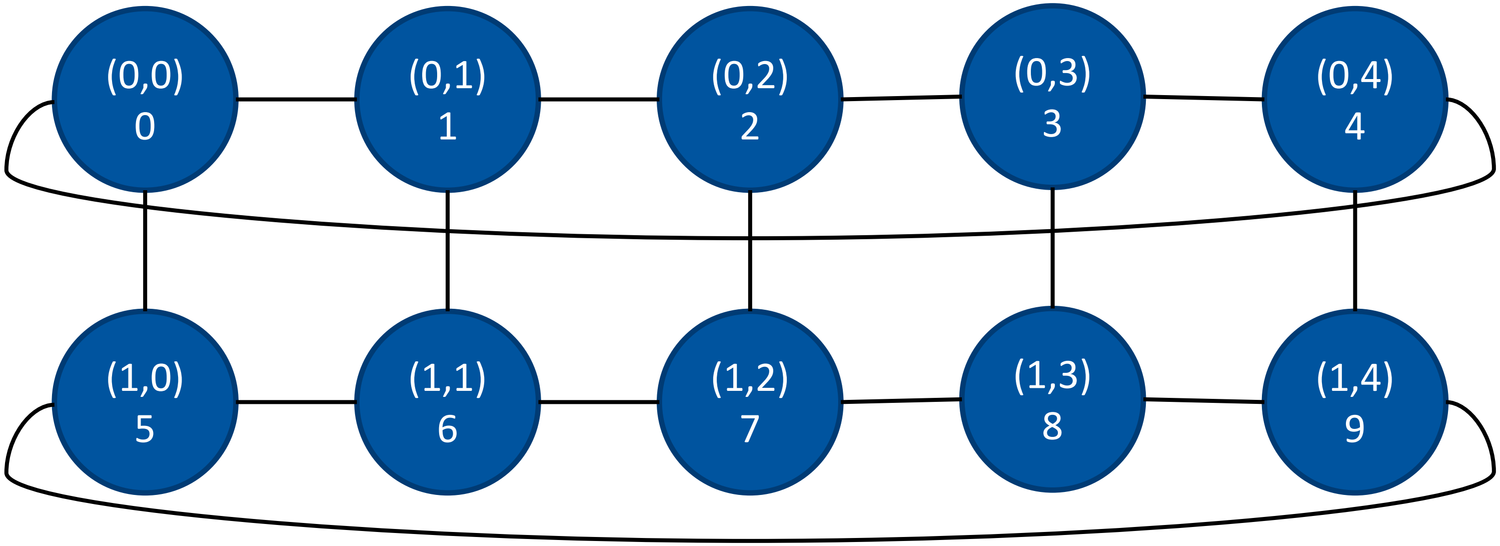
- **Split a Cartesian communicator along some dimensions**

```
MPI_Cart_sub (MPI_Comm comm, int remain_dims[], MPI_Comm *newcomm)
```

  - → **remain_dims** – boolean array; a true value flags particular dimension as being preserved by the operation (i.e. no splitting along that dimension)
  - → Creates a new Cartesian subcommunicator for each node in non-preserved dimensions
  - → Nodes with the same coordinate along non-preserved dimensions become members of the same subcommunicator
  - → Periodicity of the preserved dimensions is carried on into the newly created subcommunicators

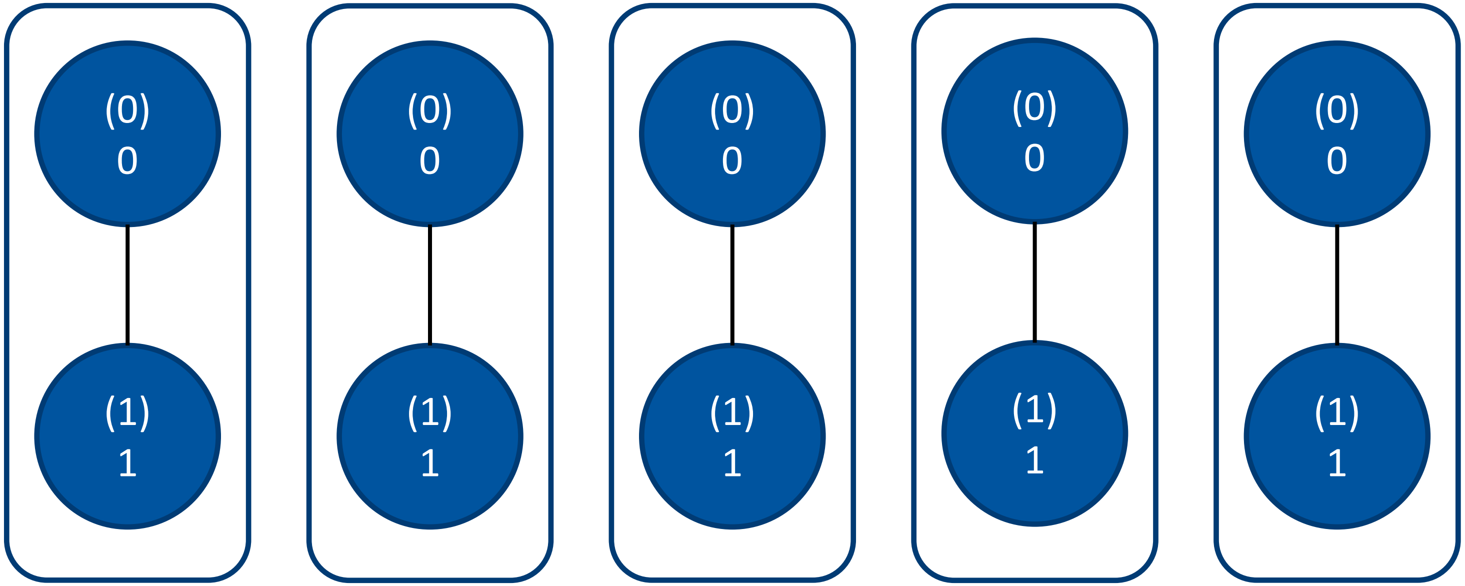
Cartesian Partitioning

- **Example: initial 2x5 Cartesian topology**



Cartesian Partitioning

- **Example: remain_dims = { true, false }**

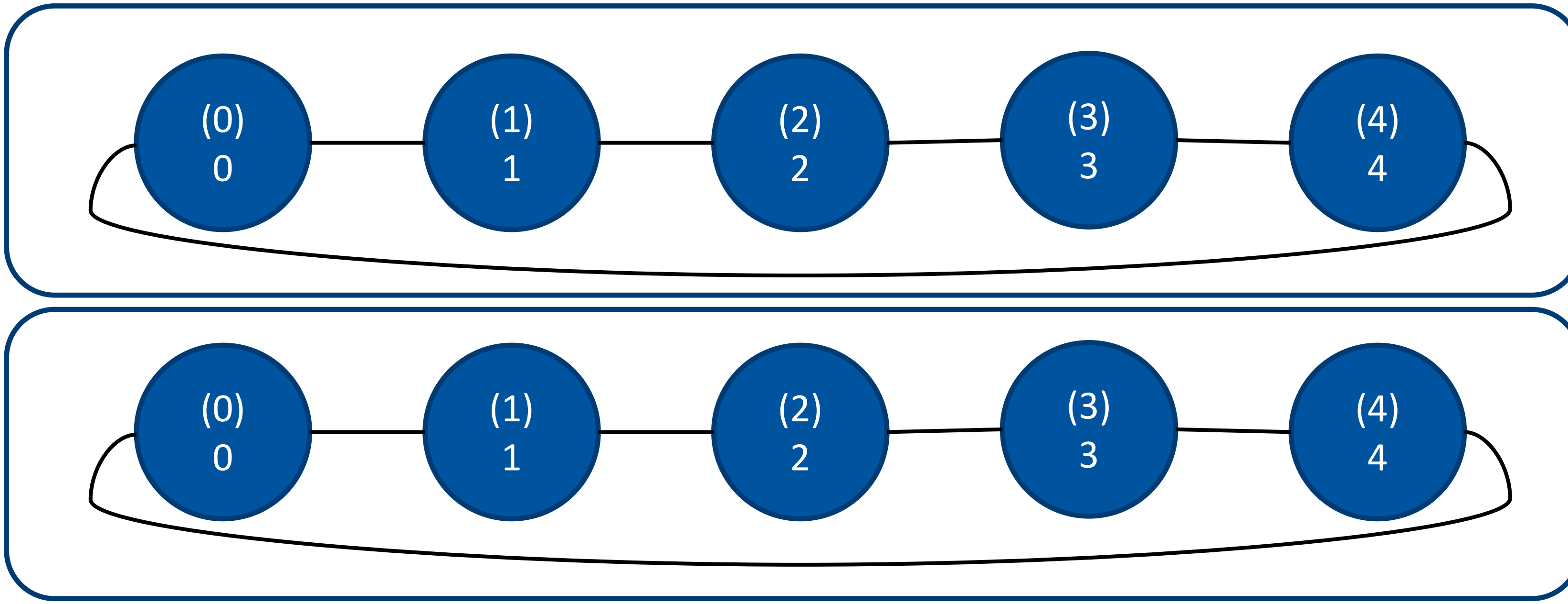


→ Five one-dimensional subcommunicators created as a result

→ Each subcommunicator contains 2 processes

Cartesian Partitioning

- **Example: remain_dims = { false, true }**

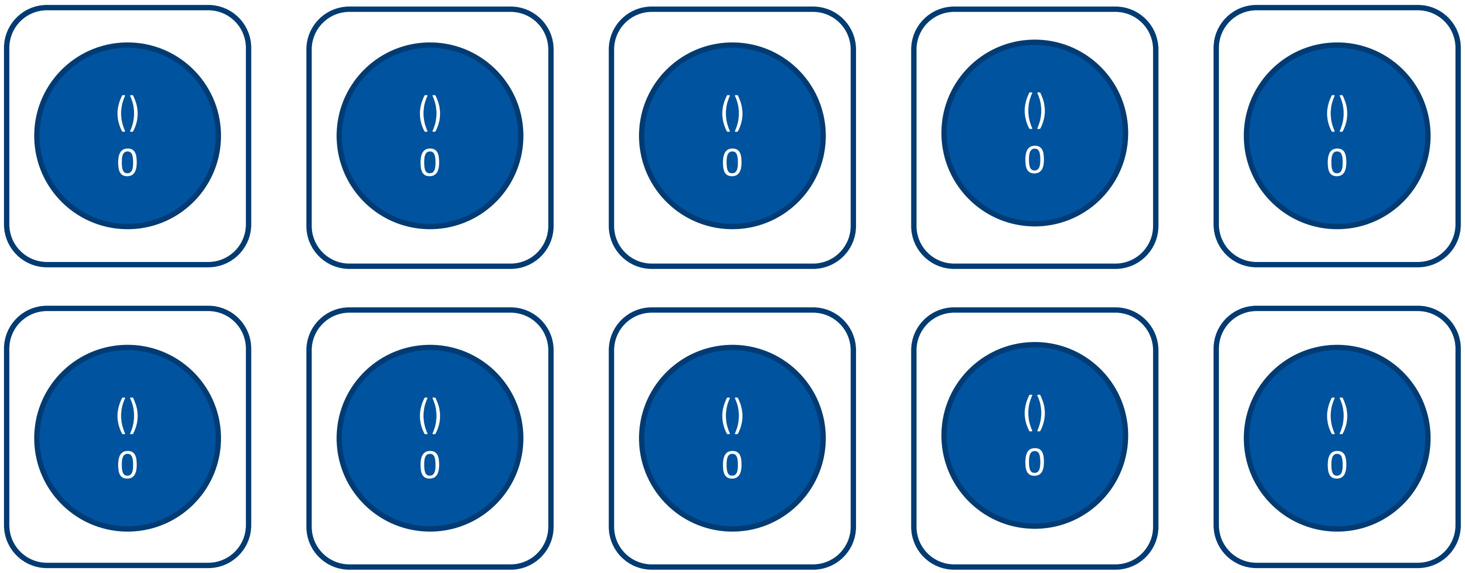


→ Two one-dimensional subcommunicators created as a result

→ Each subcommunicator contains 5 processes

Cartesian Partitioning

- **Example: remain_dims = { false, false }**



→ Ten zero-dimensional subcommunicators created as a result

→ Each subcommunicator contains only one process

Destroying Communicators

- **Communicators take up memory and other precious resources**
- **Should be freed once no longer needed**

```
MPI_Comm_free (MPI_Comm *comm)
```

→ Marks **comm** for deletion

→ **comm** is set to **MPI_COMM_NULL** on return

→ The actual communicator object is only deleted once all pending operations are completed

- **Do not try to free predefined communicators such as MPI_COMM_WORLD, MPI_COMM_SELF or MPI_COMM_NULL**

Communicators

?

Agenda

- **Motivation**
- **Part 1**
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- **Part 2**
  - → Collective operations
  - → Communicators
  - → User datatypes
- **Part 3**
  - → Hybrid parallelisation
  - → Common parallel patterns

User Datatypes

- **Basic MPI datatypes can be combined into complex user datatypes**
  - → User (derived) datatypes can be further combined into even more complex derived datatypes
- **MPI datatypes are essentially instructions for accessing the binary content of the buffer**
  - → type sequence – *(basic data type, displacement)*
    - → displacements are relative to the beginning of the memory buffer and can be positive or negative
  - → type map – { $(type_0, disp_0)$, …, $(type_{n-1}, disp_{n-1})$ }
  - → type signature – { $type_0$, …, $type_{n-1}$ }
- **The type signature at the sender <u>must</u> match that at the receiver**

Datatype Internals

- **Lower and upper bound:**
  - → lb(datatype) = min $disp_j$
  - → ub(datatype) = max ($disp_j$ + sizeof($type_j$)) + padding
- **Extent**
  - → $extent$(datatype) = ub(datatype) – lb(datatype)
  - → The span in memory from the first to the last basic element
  - → The size of the step when accessing consecutive elements of that type
- **Size**
  - → $size$(datatype) = sum sizeof($type_j$)
  - → The total amount of bytes taken by the datatype, not counting any gaps in it

Datatype Internals

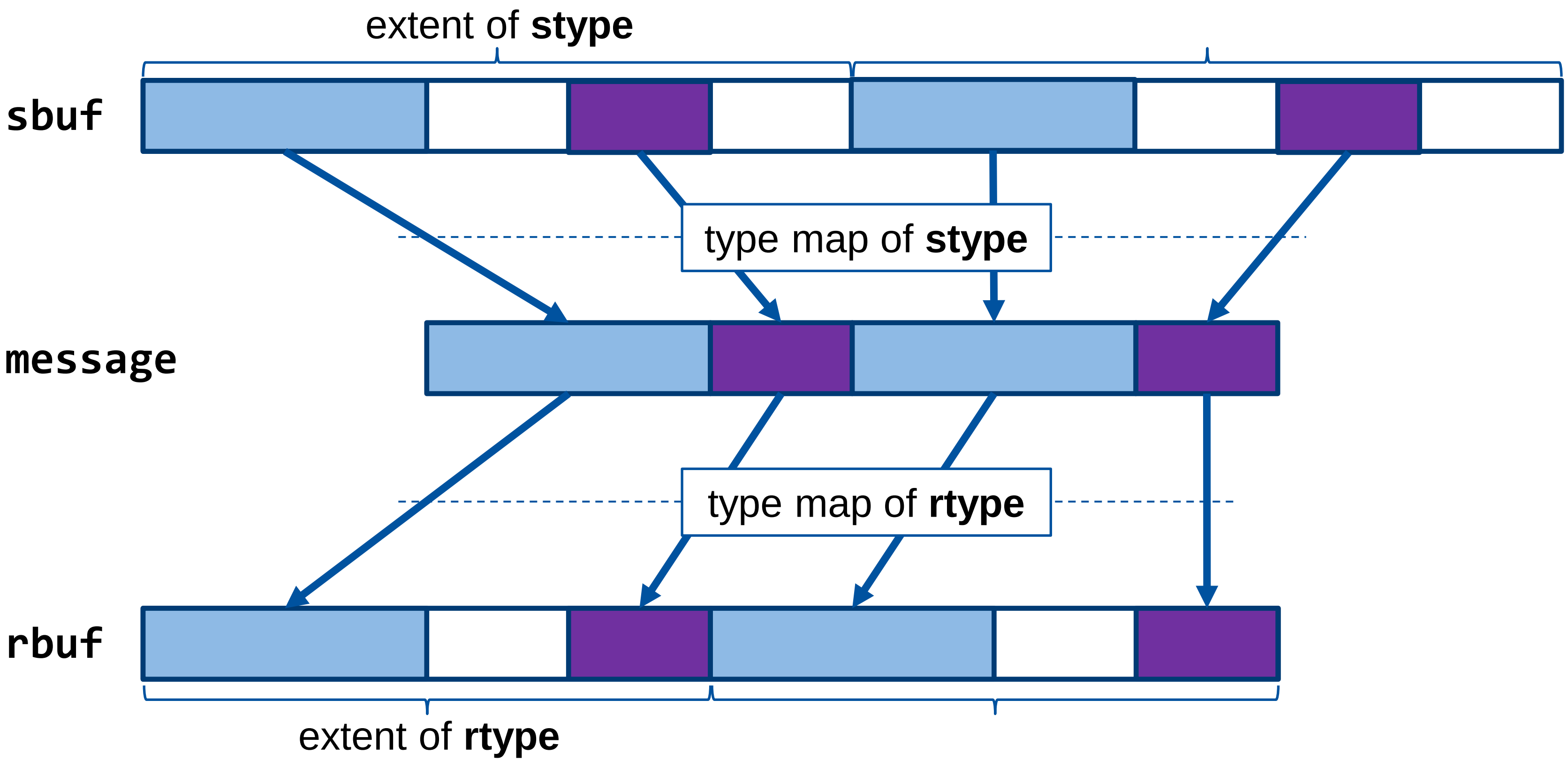
- **Example: MPI_INT**
  → *type map* = { (*int*, 0) }
  → *lb* = 0
  → *ub* = 4
  → *extent* = 4 bytes
  → *size* = 4 bytes

- **All predefined basic MPI datatypes have lower bound 0, i.e. data is flush with the buffer start**

- **Platform-specific alignment rules are taken into account**
  → The upper bound is therefore adjusted if necessary

Datatype Internals

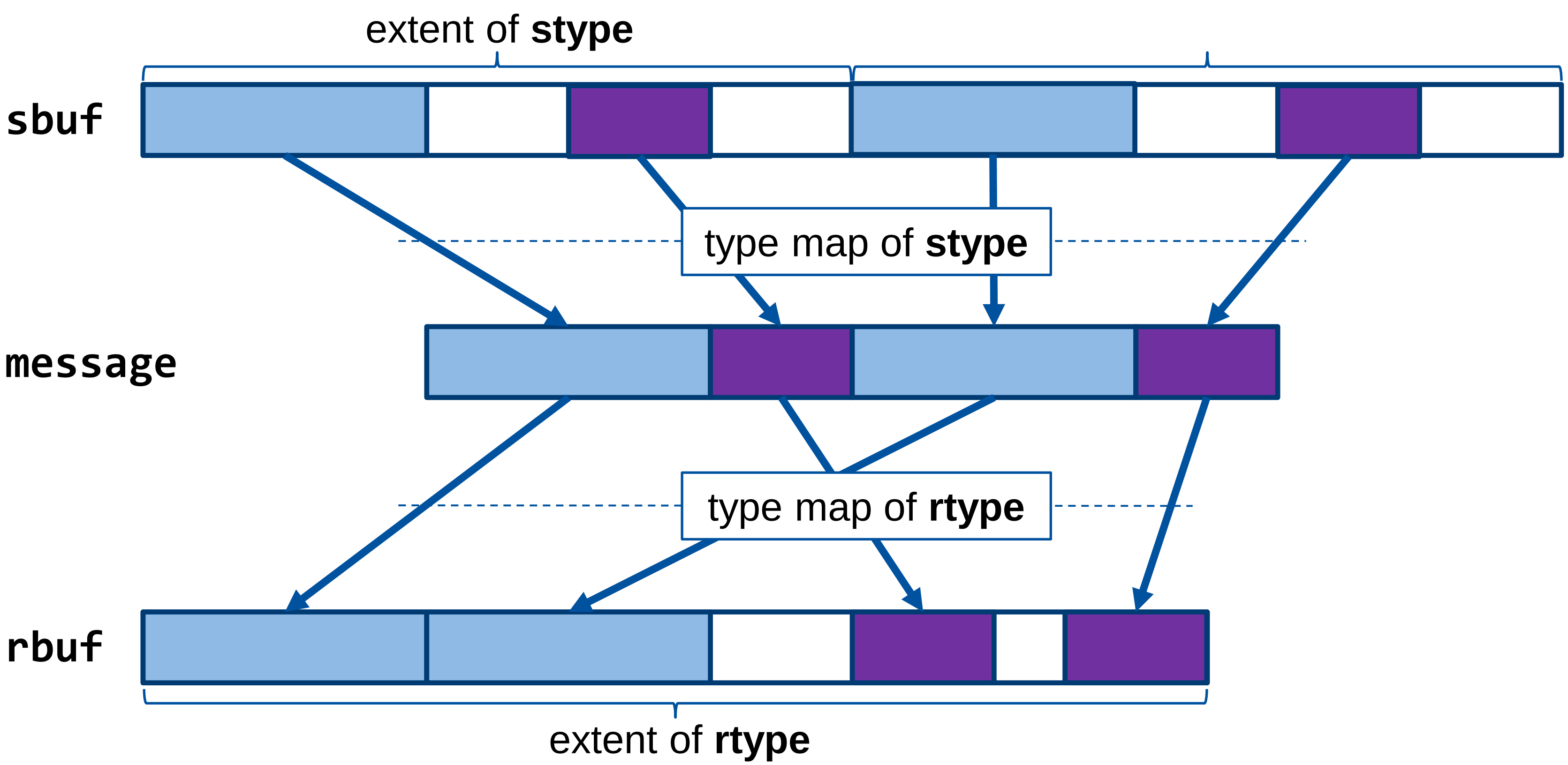
```
MPI_Send(sbuf, 2, stype, dest, 0, MPI_COMM_WORLD);
```




```
MPI_Recv(rbuf, 2, rtype, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Datatype Internals

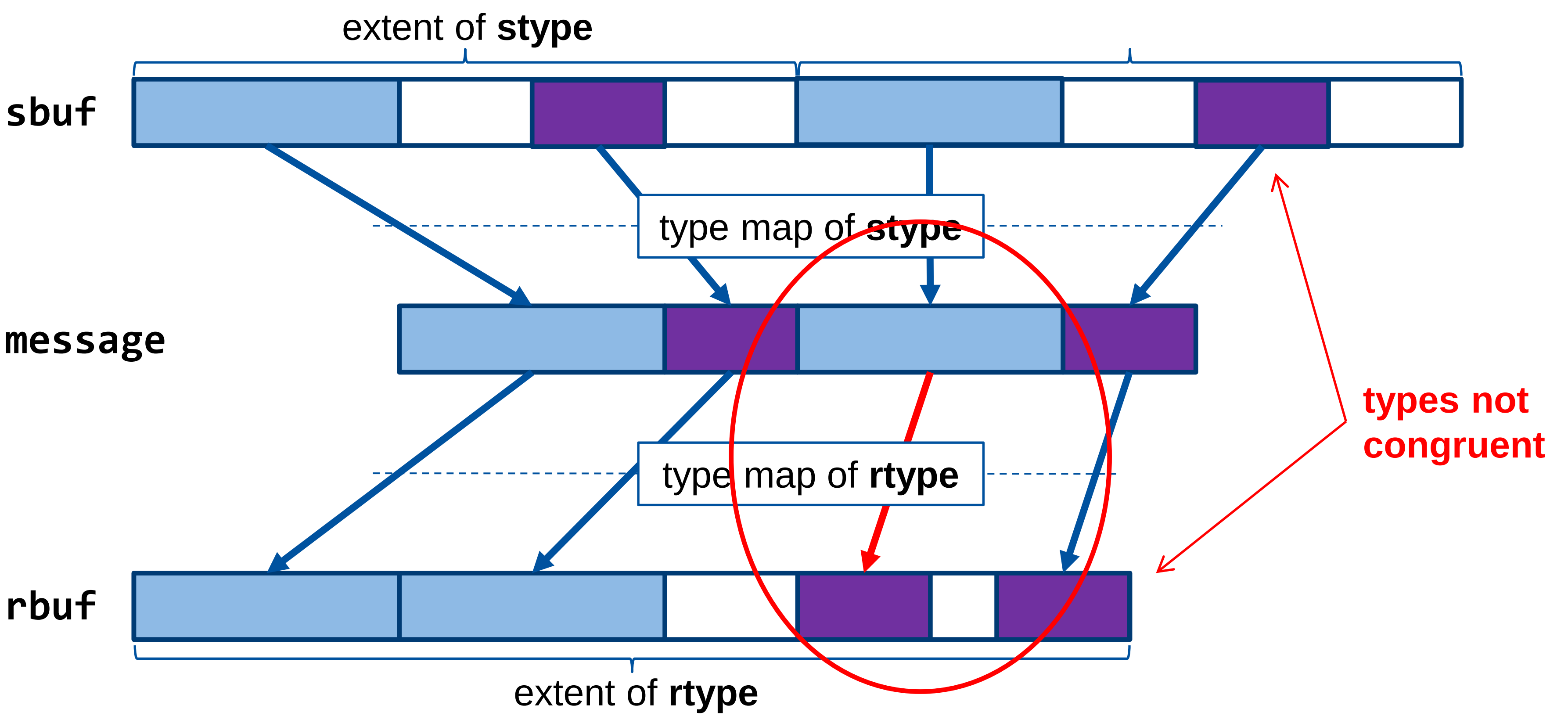
```
MPI_Send(sbuf, 2, stype, dest, 0, MPI_COMM_WORLD);
```




```
MPI_Recv(rbuf, 1, rtype, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Datatype Internals

```
MPI_Send(sbuf, 2, stype, dest, 0, MPI_COMM_WORLD);
```




```
MPI_Recv(rbuf, 1, rtype, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Contiguous Datatypes

- **Create a sequence of elements of an existing datatype**

```
MPI_Type_contiguous (int count, MPI_Type oldtype, MPI_Type *newtype)
```

→ The new datatype represents a contiguous sequence of **count** elements of **oldtype**

→ The elements are separated from each other by the extent of **oldtype**

→ A send/receive of one element of **newtype** is congruent with a receive/send of **count** elements of **oldtype**

- **Useful for sending entire matrix rows (C/C++) or columns (Fortran)**

Vector Datatypes

- **Create a sequence of equally spaced blocks of elements**

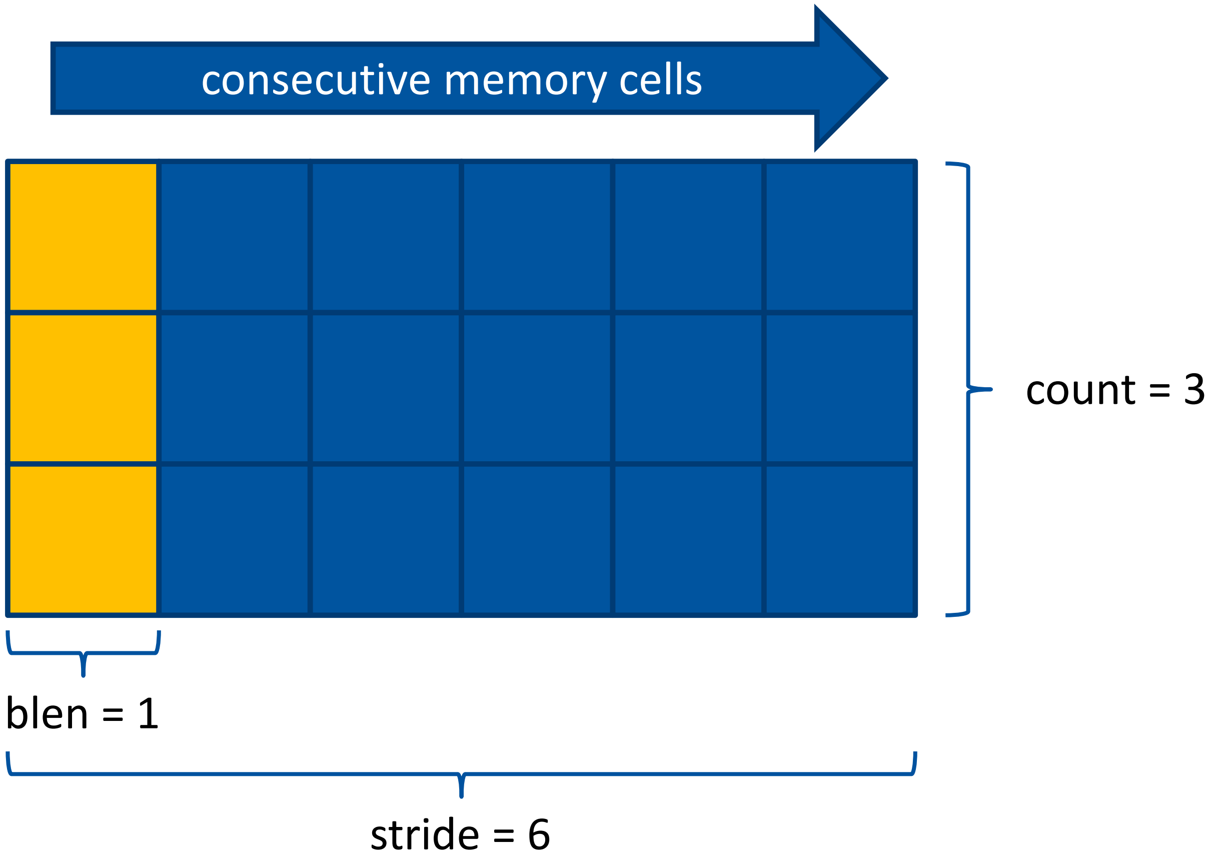
```
MPI_Type_vector (count, blen, stride, oldtype, newtype)
```

  → The new datatype represents a sequence of **count** blocks, each containing **blen** elements of the old datatype

  → Every two consecutive blocks are separated by **stride** elements each

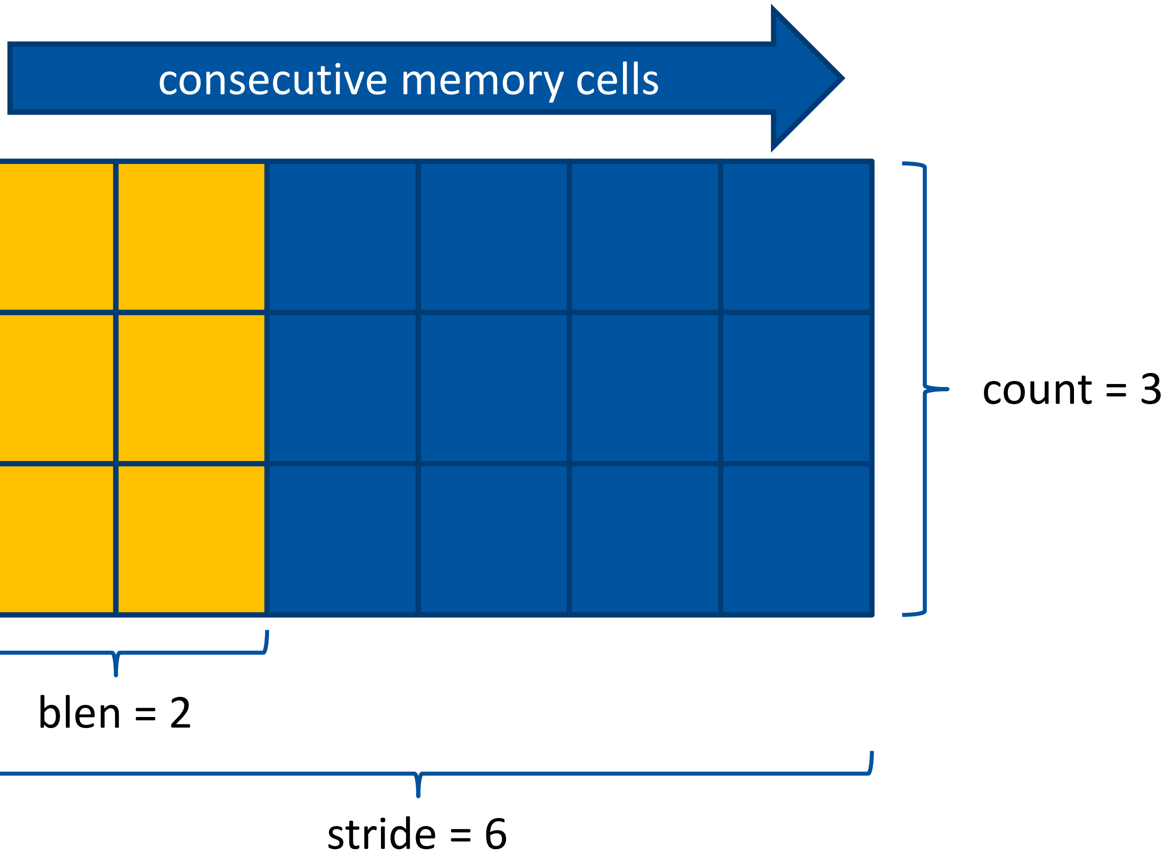
- **Useful for sending matrix columns (C/C++) or rows (Fortran)**
  → **stride** = row (C/C++) | column (Fortran) length (in number of elements)
  → **blen** = 1 (or the number of consecutive rows/columns)
  → **count** = number of rows (C/C++) | columns (Fortran)

- **Example: single column of a C/C++ matrix**
  - → mat[3][6]



Vector Datatypes

- **Example: two consecutive columns of a C/C++ matrix**
  - → mat[3][6]



Structure Datatypes

- **The most generic datatype**
  - → Useful for C/C++ structures and Fortran derived data type / COMMON blocks

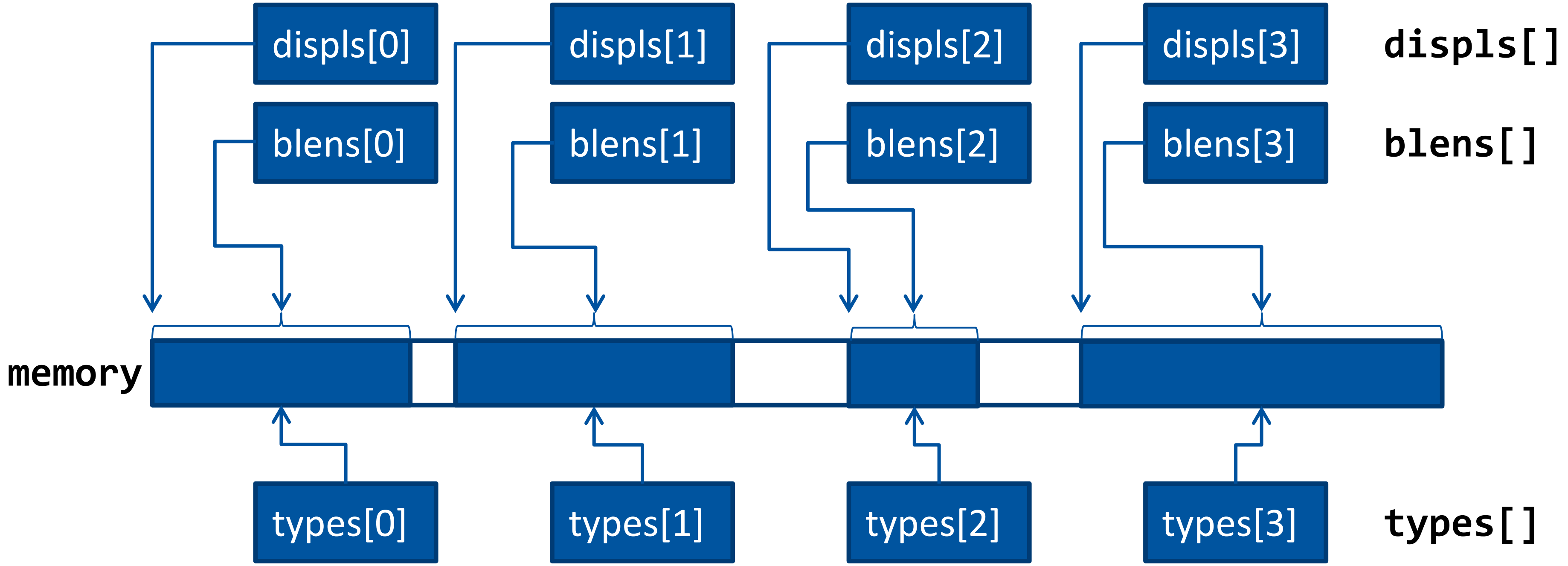
```
MPI_Type_create_struct (int count, int blens[], MPI_Aint displs[],
                        MPI_Datatype types[], MPI_Datatype *datatype)
```

- → **count:** number of blocks in the datatype
- → **blens[]:** number of elements in each block
- → **displs[]:** displacement in bytes from the start of each block
- → **types[]:** datatype of the elements in each block
- → **datatype:** handle of the new datatype

The most generic datatype

→ Useful for C/C++ structures and Fortran derived data type / COMMON blocks

```
MPI_Type_create_struct (int count, int blens[], MPI_Aint displs[],
                        MPI_Datatype types[], MPI_Datatype *datatype)
```



Structure Datatypes

- **The most generic datatype**
  - → Corresponds to C/C++ struct

```
typedef struct {
  float mass;
  double pos[3];
  char sym;
} Particle;

int blens[] = { 1, 3, 1 };
MPI_Aint displs[] = { offsetof(Particle, mass),
                      offsetof(Particle, pos),
                      offsetof(Particle, sym) };
MPI_Type types[] = { MPI_FLOAT, MPI_DOUBLE, MPI_CHAR };

MPI_Type particle_type;
MPI_Type_create_struct(3, blens, displs, types, &particle_type);
```

Using Derived Datatypes

- **Register a datatype for use with communication operations:**

```
MPI_Type_commit (MPI_Datatype *datatype)
```

  → A datatype must be committed before it can be used in communications

  → All predefined datatypes are already committed

  → Intermediate datatypes, i.e. ones used for building more complex datatypes but not used in communication, can be left uncommitted

- **Deregister and free a datatype:**

```
MPI_Type_free (MPI_Datatype *datatype)
```

  → Derived datatypes, build from the freed datatype, are not affected

  → **datatype** set to **MPI_TYPE_NULL** upon successful return

Structure Datatypes

- **The most generic datatype**

```
typedef struct {
  float mass;
  double pos[3];
  char sym;
} Particle;

int blens[] = { 1, 3, 1 };
MPI_Aint displs[] = { offsetof(Particle, mass),
                      offsetof(Particle, pos),
                      offsetof(Particle, sym) };
MPI_Type types[] = { MPI_FLOAT, MPI_DOUBLE, MPI_CHAR };

MPI_Type particle_struct;
MPI_Type_create_struct(3, blens, displs, types, &particle_struct);
MPI_Type_commit(&particle_struct);
```

- **particle_struct can now be used to send a scalar of type Particle**

Structure Datatypes

- **Resize to the true size of the structure**

```
int blens[] = { 1, 3, 1 };
MPI_Aint displs[] = { offsetof(Particle, mass),
                      offsetof(Particle, pos),
                      offsetof(Particle, sym) };
MPI_Type types[] = { MPI_FLOAT, MPI_DOUBLE, MPI_CHAR };

MPI_Type particle_struct;
MPI_Type_create_struct(3, blens, displs, types, &particle_struct);
// No need to commit particle_struct - not used in communication

MPI_Aint true_size = sizeof(Particle);
MPI_Type_create_resized(particle_struct, 0, true_size, &particle_type);
MPI_Type_commit(&particle_type);
```

- **MPI_Type_create_resized takes an existing datatype and creates a new one with modified lower bound and extent**

Using Derived Datatypes

- **Datatypes can be mixed and matched on both sides of a communication operation as long as their type signatures match**
  - → E.g. one can send 10 **MPI_INT** elements and receive them as a single element of a contiguous datatype with **count = 10** and **oldtype** = **MPI_INT**
  - → Extra care should be taken when using derived datatypes in collective operations

- **If the amount of data in the received message is not enough to build an integral number of elements of a derived datatype, a count of MPI_UNDEFINED is returned by MPI_Get_count**

Communicators

?