Message Passing with MPI

PPCES 2018

Joachim Protze / Marc-André Hermanns
IT Center / JARA-HPC

Slides by Hristo Iliev

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

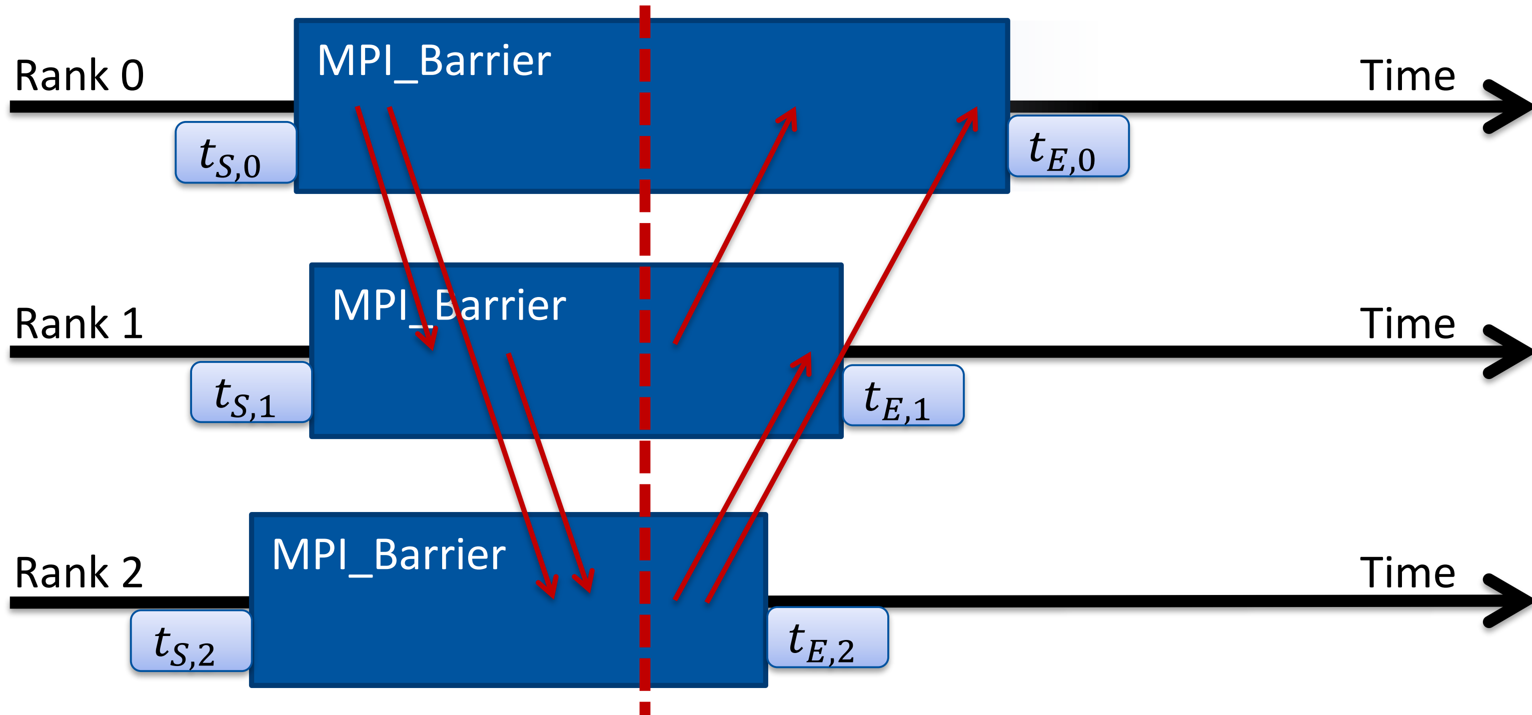
Collective Operations

- **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:**

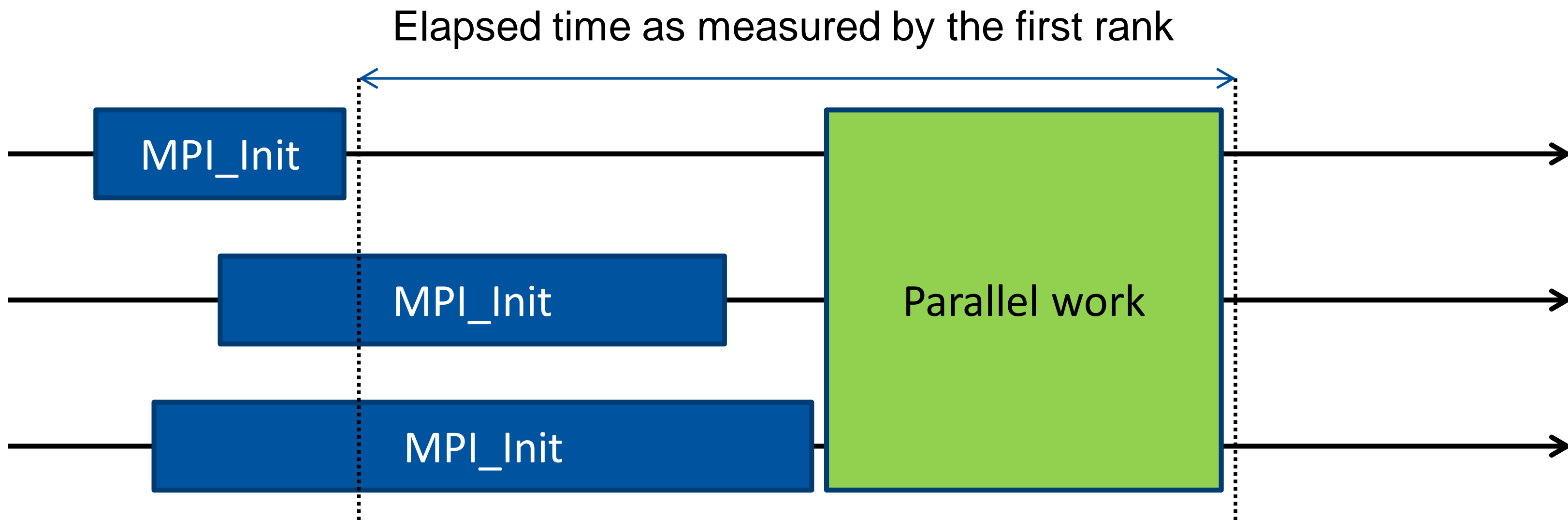
```
MPI_Barrier (MPI_Comm comm)
```



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

Barrier Synchronisation

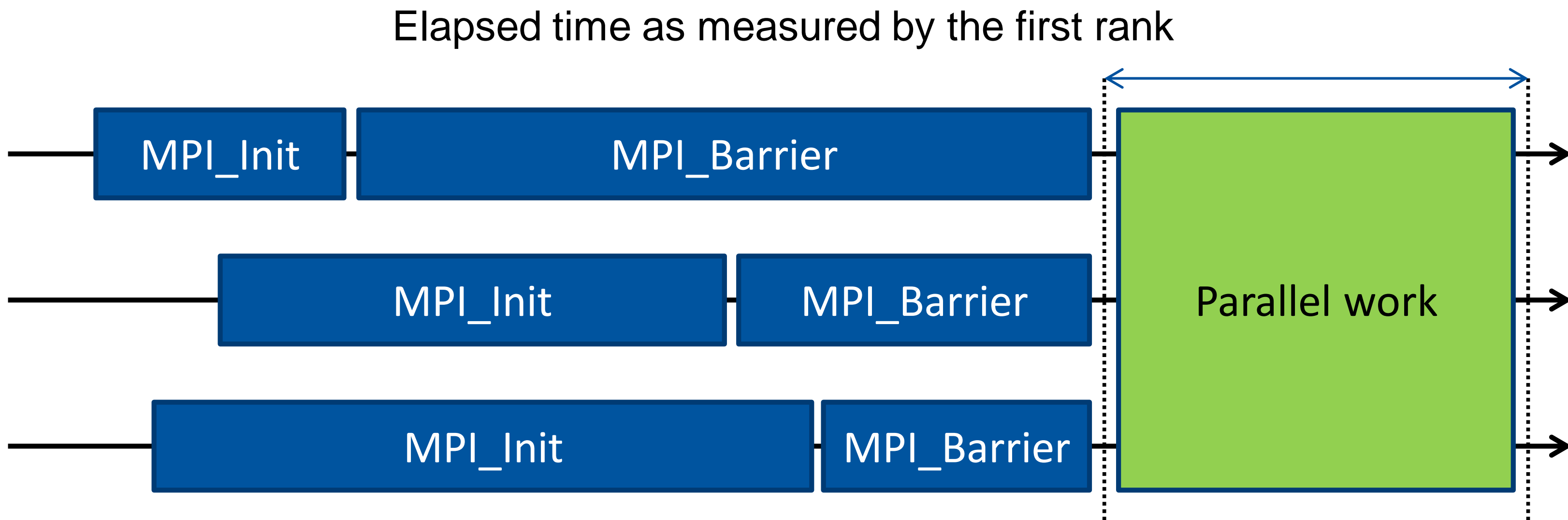
- **Useful for benchmarking**
  - → Always synchronise before taking time measurements



  - → 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 may occur, but 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 must specify the same value

→ **comm:** communicator

- **Notes:**

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

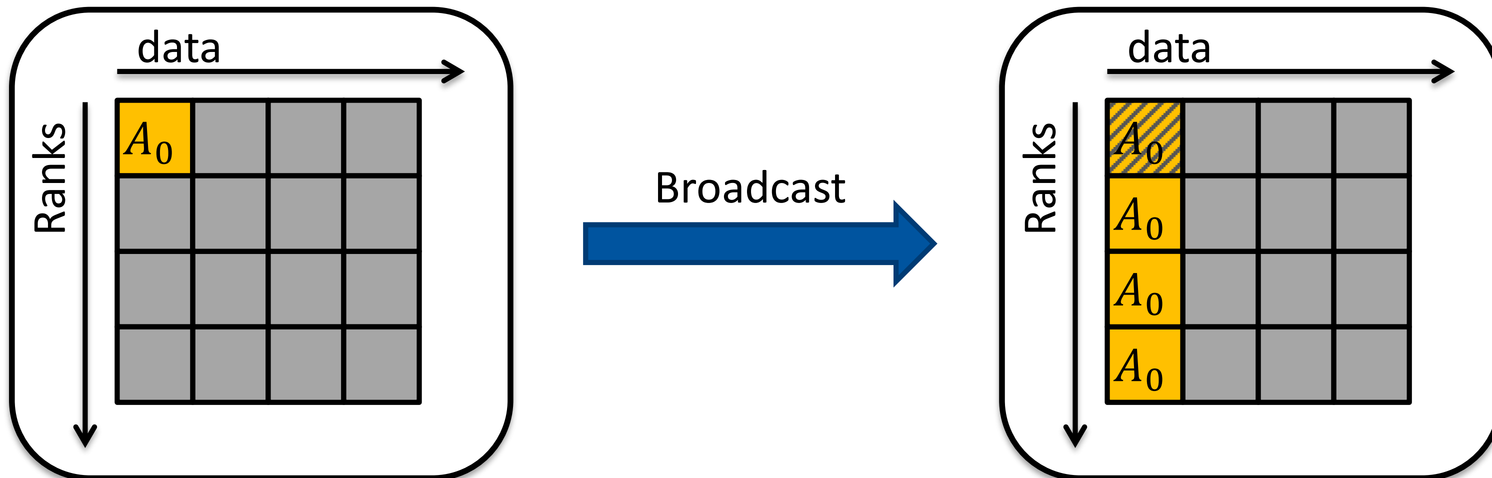
→ in rank **root**, **data** is an input argument

→ Type signatures must match across all ranks (→ User Datatypes)

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 → Deadlock
```

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:** communicator

Data Scatter

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

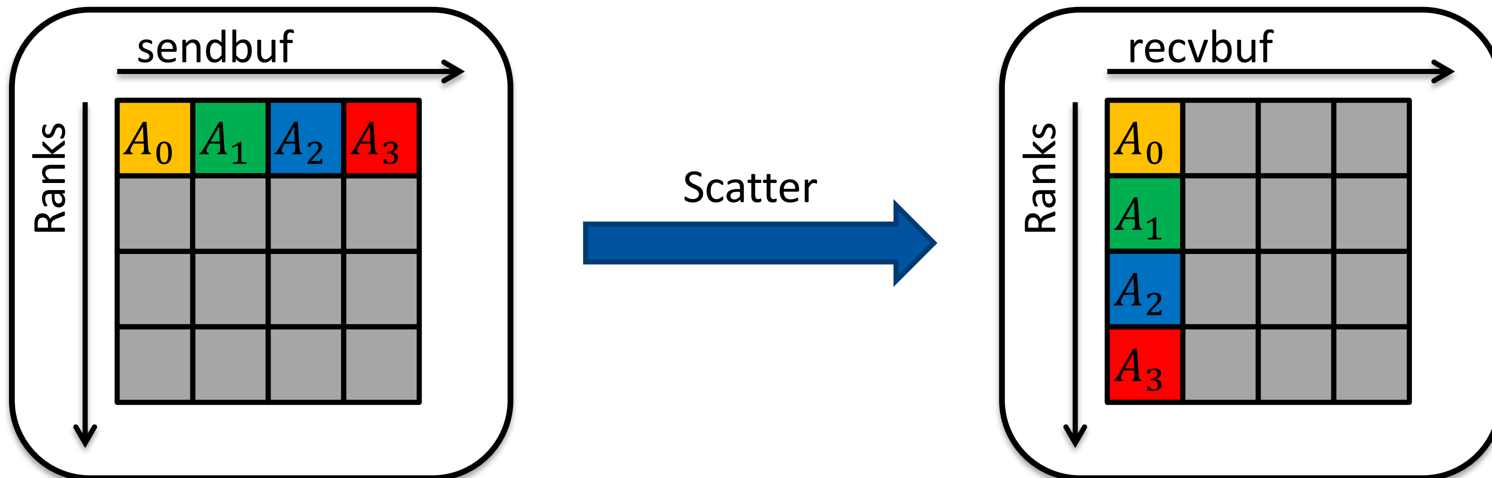
```
MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```

- **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
  - → Type signatures of must match across all ranks (→ Datatypes)

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)
```



- **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)
```

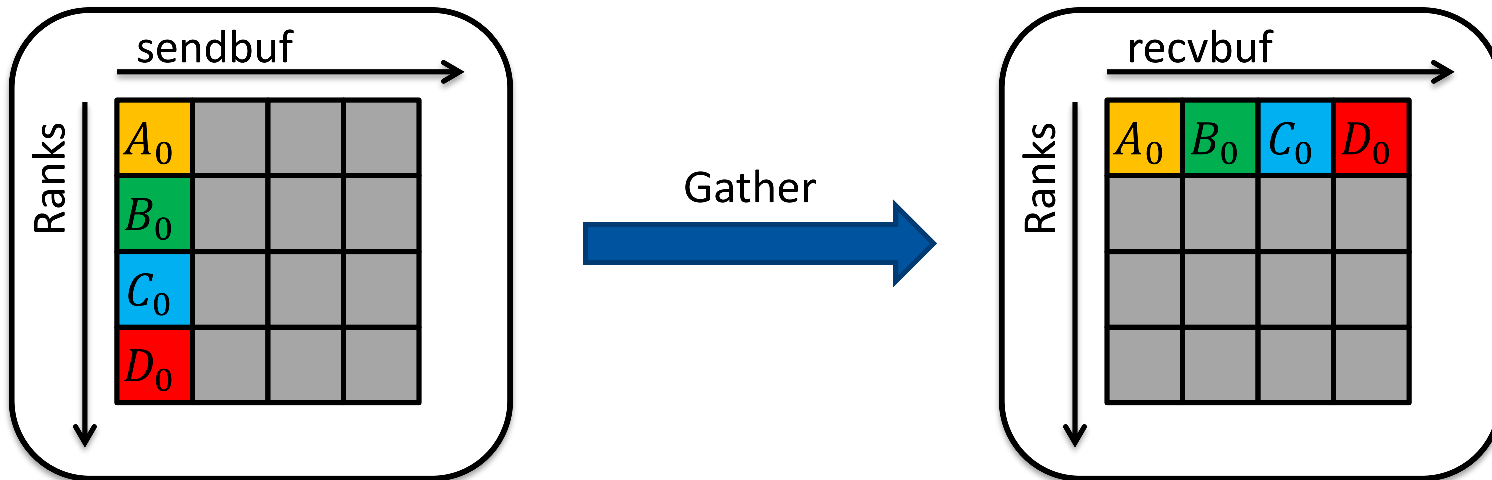
Significant at root rank only

- **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

RWTH AACHEN UNIVERSITY

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

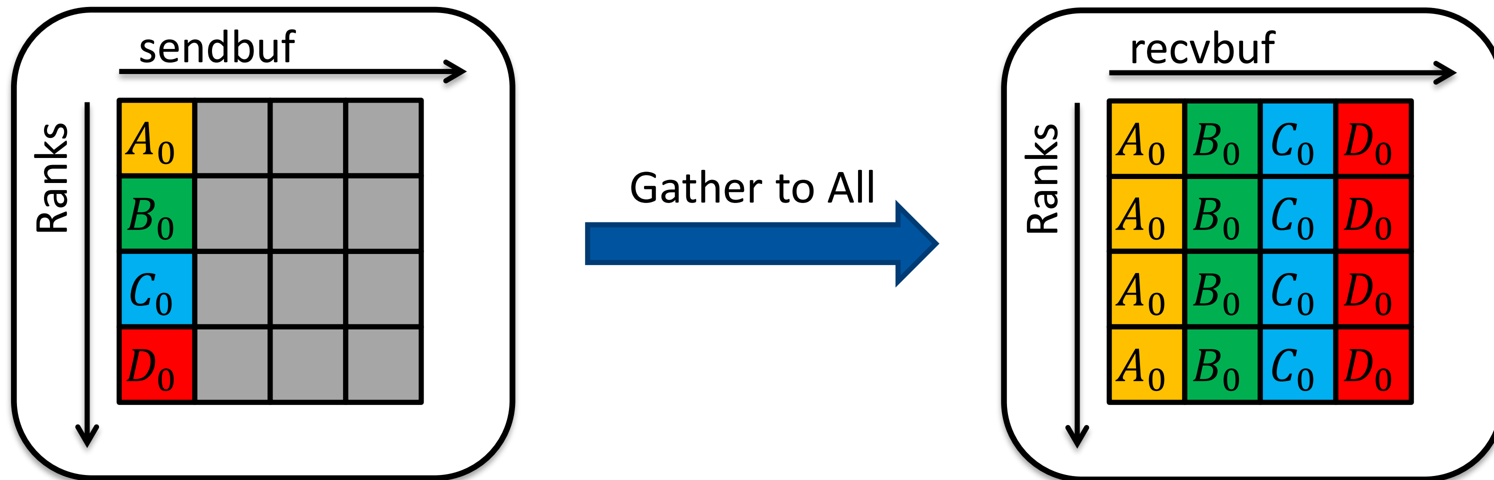
```
MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
      void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

- **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
  - → Type signatures of must match across all ranks (→ Datatypes)
  - → 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:**

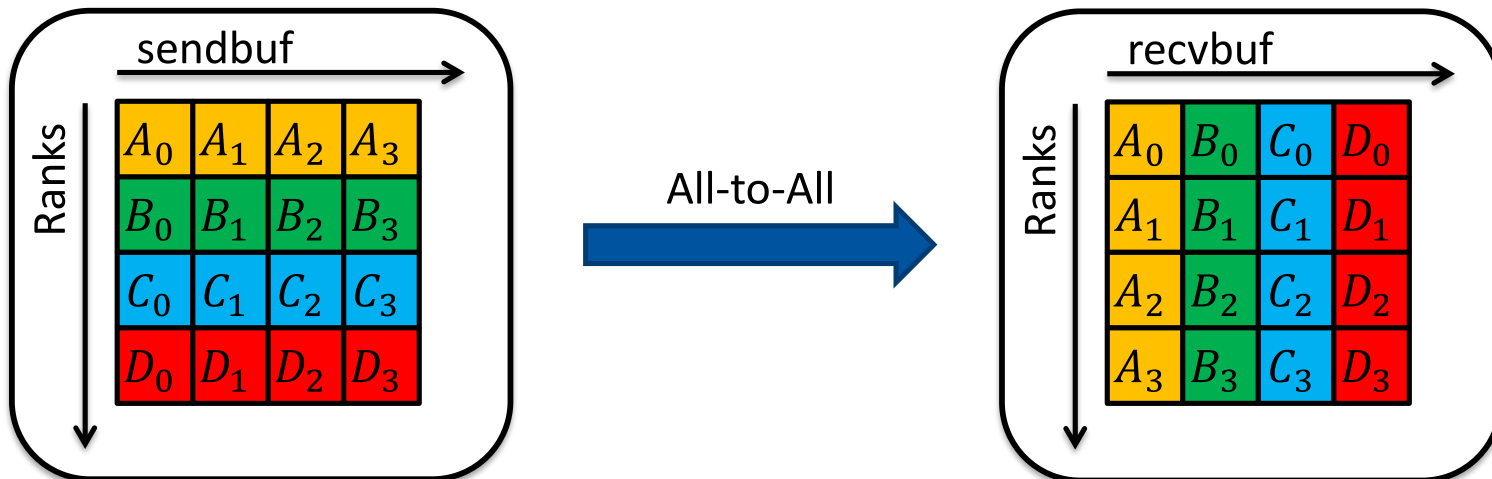
```
MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype,
       void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

- **Notes:**
  - → 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)

- **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

- **Most collectives have varying count versions**
- **Communicate individual amount of data among rank pairs**
- **May fit better for irregular problems**
  - → Regular domain decomposition where domain does not divide evenly by number of rank
  - → Irregular domain decomposition
- **May have preform less well than regular versions**
  - → More complex parameter handling
  - → Different algorithms and less potentially less well tuned

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)
```

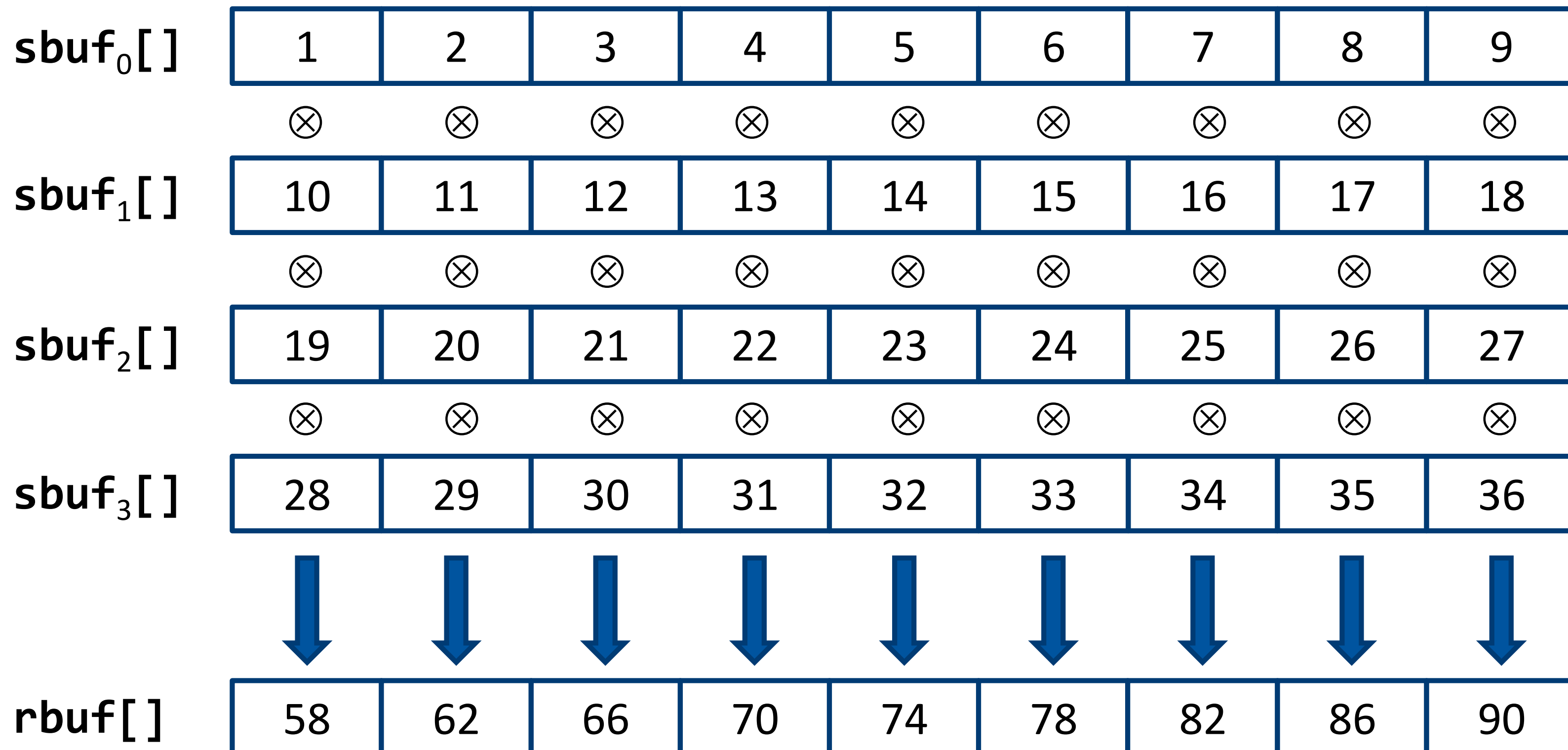
→ **sendbuf:** data to be reduced
→ **recvbuf:** location for the result(s) (significant at root only)
→ **count:** number of data elements
→ **datatype:** element datatype
→ **op:** handle of the reduction operation
→ **root:** destination rank
→ **comm:** communicator

- **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`$_{nranks-1}$`[i]`



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)
```

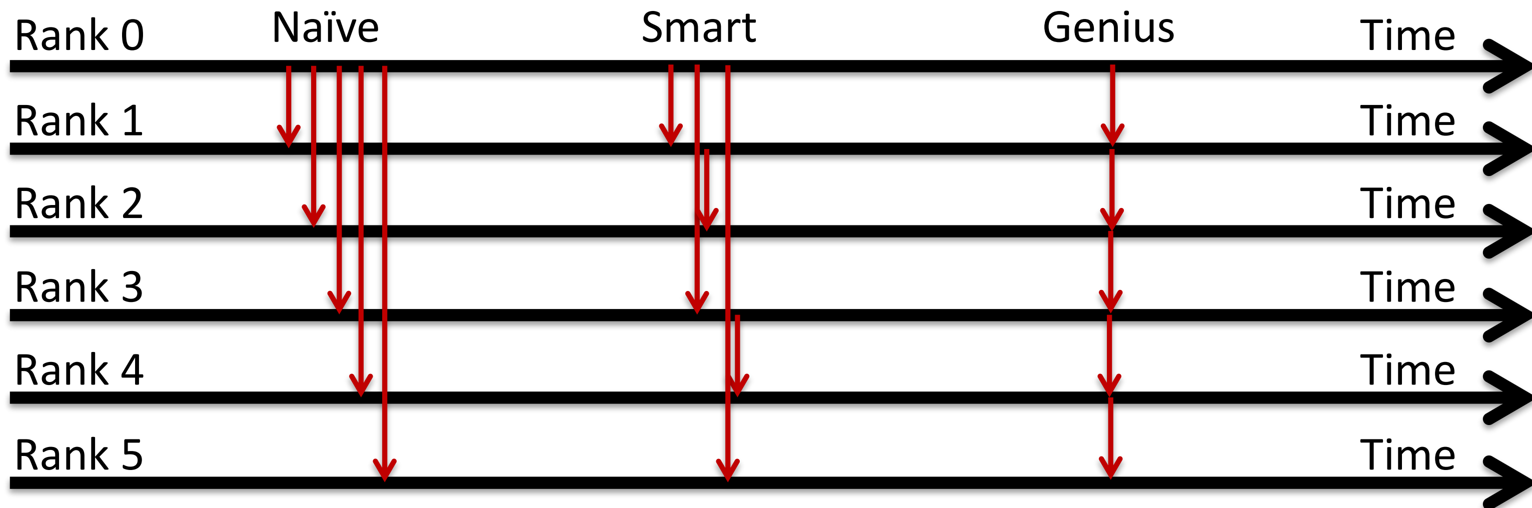
- **Notes:**
  - → every rank receives the result of the reduction operation
  - → equivalent to **MPI_Reduce + MPI_Bcast** with the same root
  - → can be slower with non-commutative operations because of the forced in-order execution (the same applies to **MPI_Reduce**)
    - → concerns non-commutative user-defined operations only

Summary: Collective Operations

- **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 explicitly synchronising MPI collective**
  - → Some may synchronize implicitly (e.g., Allgather, Allreduce)
- **Point-to-point and collective communication are independent of each other on the same communicator.**

Advantages of Collective Operations

- **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(#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

- **Defines context for each communication operation in MPI**
  - → Group of participating peers (process group)
  - → Error handlers for communication and I/O operations
  - → Local key/value cache
  - → Virtual topology information (optional)

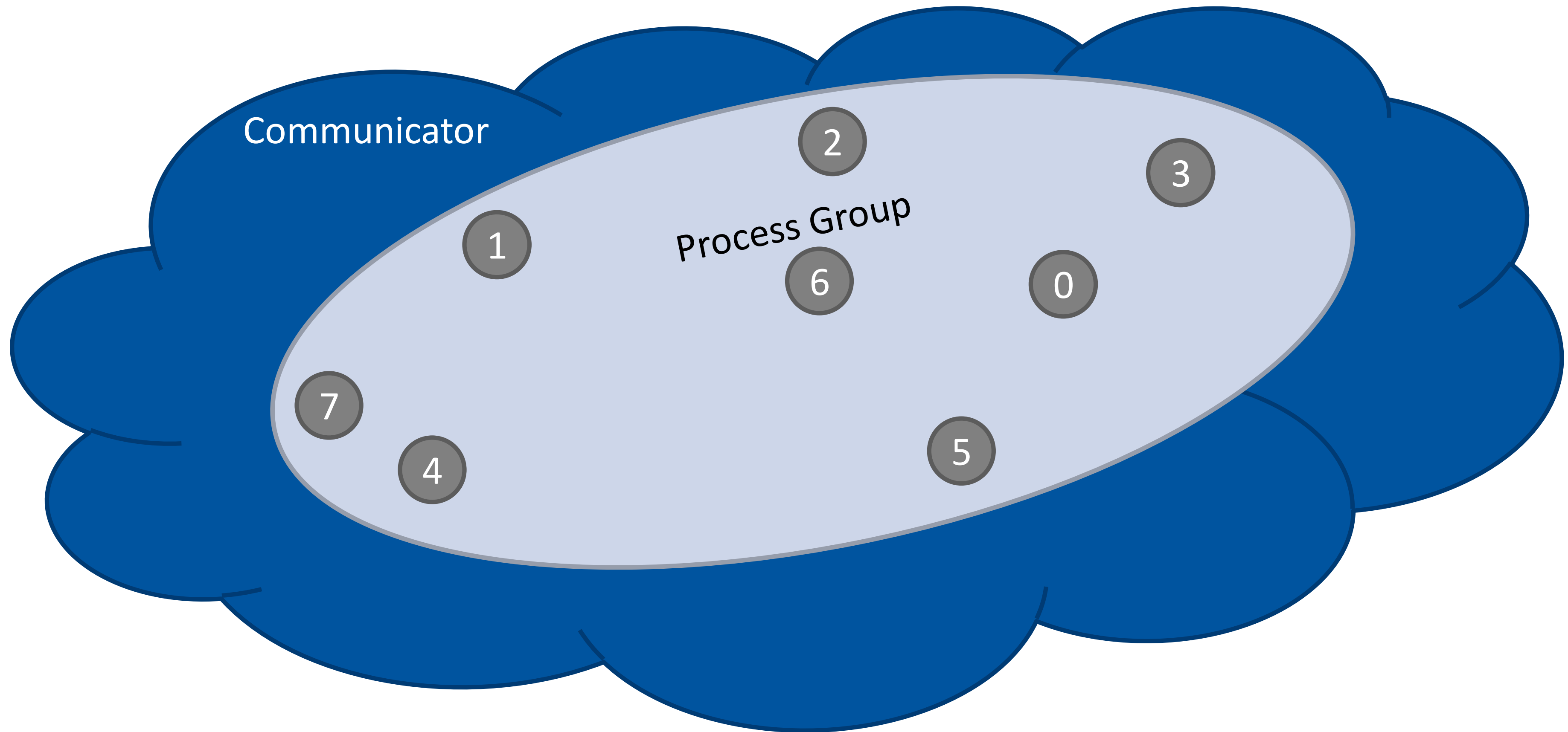
- **Two predefined communicators:**
  - → **MPI_COMM_WORLD**

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

    contains only the current process

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

Communicator creation

- **Duplicate an existing communicator**
  → MPI_Comm_dup, MPI_Comm_dup_with_info, MPI_Comm_idup

- **Create new communicator for a subgroup of a communicator**
  → MPI_Comm_create, MPI_Comm_create_group

- **Split an existing communicator**
  → MPI_Comm_split, MPI_Comm_split_type

Communicator duplication

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

```
MPI_Comm_dup (MPI_Comm comm, MPI_Comm *newcomm)
```

→ New communication context with same ranks and ordering

→ Easy isolation of encapsulated communication

→ Libraries should never communicate on MPI_COMM_WORLD directly

```
MPI_Comm_idup (MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
```

→ Non-blocking variant (hide cost of communicator creation)

→ Complete with wait/test family of calls

```
MPI_Comm_dup_with_info (MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
```

→ Pass key-value information on creation

Virtual Topologies

- **Each communicator can have an associated topology**
  - → Mapping between ranks and abstract addresses
  - → Virtual neighbourhood (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
- **Communication not restricted to neighbors**
  - → Each rank can still communicate with every other rank in the communicator

Virtual Topoligies II

- **Enable process reordering**
  - → Improved mapping of processes for neighborhood communication
- **Enable use of neighborhood collectives (not covered here)**
  - → Collective communication on ad-hoc sub-communicators
- **N-dimensional Cartesion topologies**
  - → Easy neighbor query in any dimension
  - → Transparently handles boundary conditions
- **Graph topologies**
  - → Map irregular application domains to ranks

Destroying Communicators

RWTH AACHEN UNIVERSITY

- **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

- **It is erroneous to free predefined communicators MPI_COMM_WORLD, MPI_COMM_SELF or MPI_COMM_NULL**

Communicators

- 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

MPI Datatypes

- **MPI datatypes are opaque handles**
  - → Instructions for accessing the binary content of a memory buffer

- **MPI predefines basic datatypes for each language binding**
  - → Distinct handles for C/C++ and Fortran (e.g., MPI_INT vs. MPI_INTEGER)
  - → Describe a single data element

- **More complex MPI datatypes can be constructed (derived)**
  - → Matrix rows & columns, diagonal matrices, structures

Terminology

- **Type signature**
  - → A sequence of basic datatypes described by a given type and count
  - → Example: {MPI_INT, MPI_INT, MPI_DOUBLE}

- **Type map**
  - → A sequence of basic datatypes and their displacements
  - → Example: {(MPI_INT, 0), (MPI_INT, 4), (MPI_DOUBLE, 8)}

- **Datatypes are local objects**
  - → May differ across processes
  - → Enable transparent type marshalling (encoding & decoding of data)

Terminology II

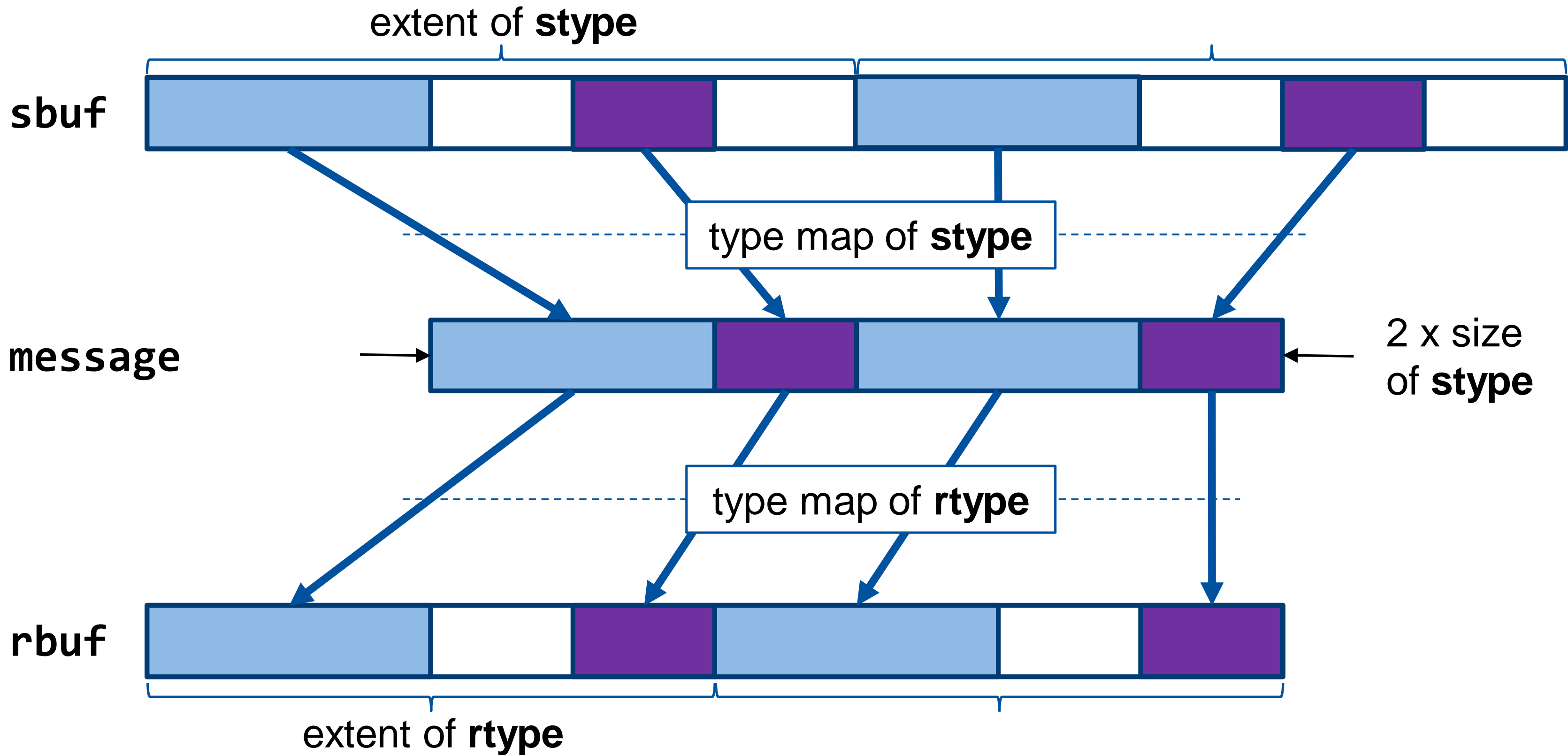
- **Lower and upper bound:**
  - → lb(datatype) = min $disp_j$
  - → ub(datatype) = max ($disp_j$ + sizeof($type_j$)) + padding
- **Extent**
  - → $extent$(datatype) = ub(datatype) – lb(datatype)
  - → The size of the step when accessing consecutive elements of that type
- **Size**
  - → $size$(datatype)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

Basic datatype example

- **Example: MPI_INT**
  - → *type map* = { (*MPI_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

Using Datatypes

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




```
MPI_Recv(rbuf, 2, 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, blocklen, 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)**

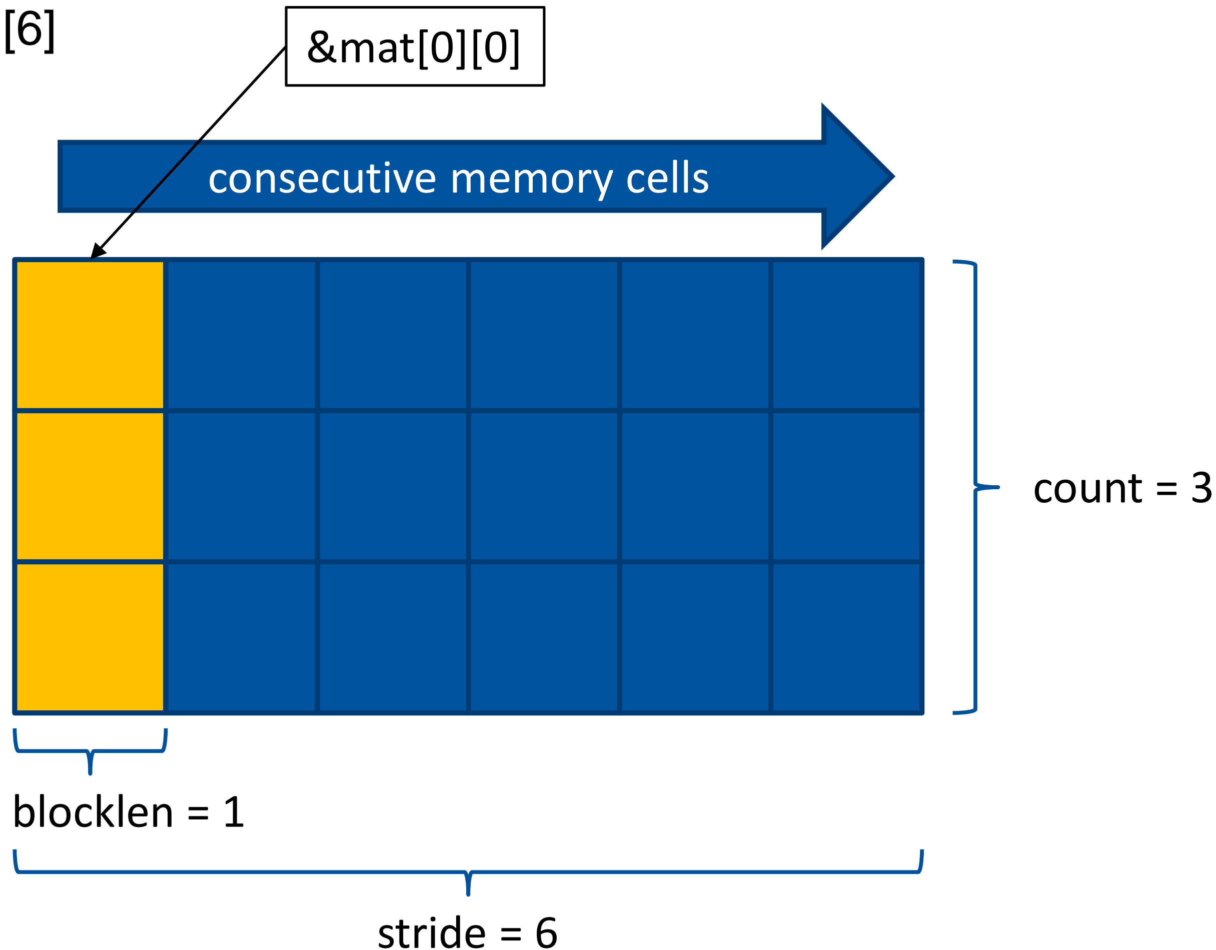
  → **stride** = row (C/C++) | column (Fortran) length (in number of elements)

  → **blocklen** = 1 (or the number of consecutive rows/columns)

  → **count** = number of rows (C/C++) | columns (Fortran)

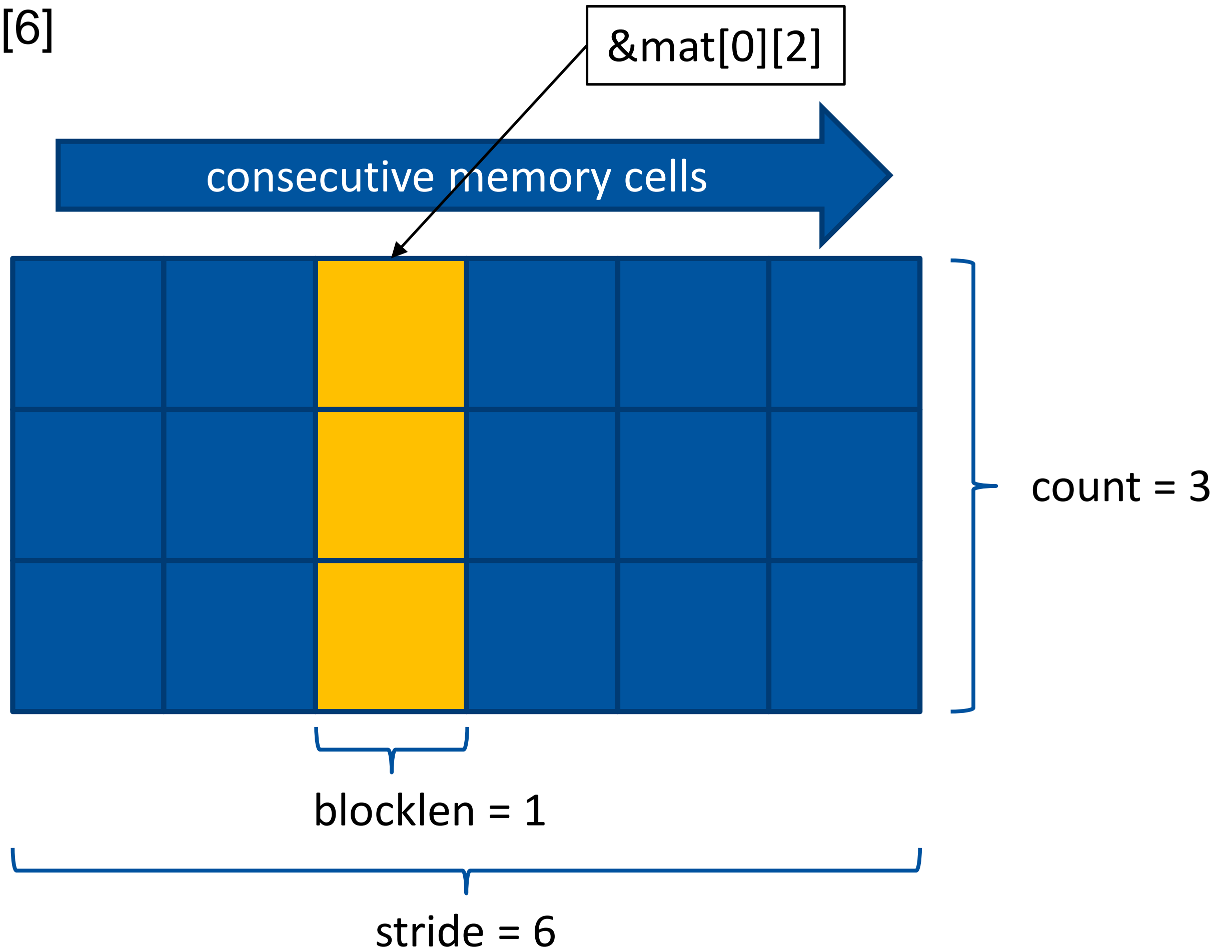
Vector Datatypes

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

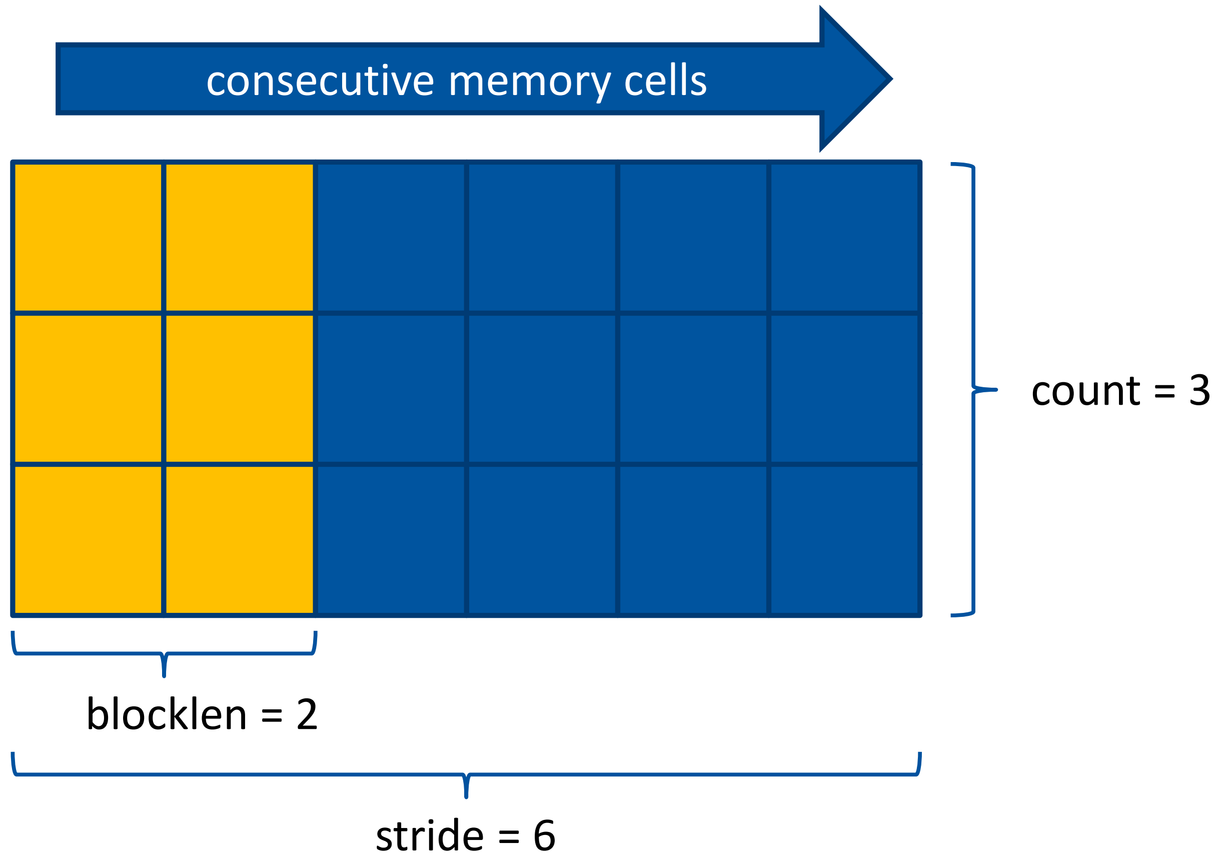


Vector Datatypes

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



- **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

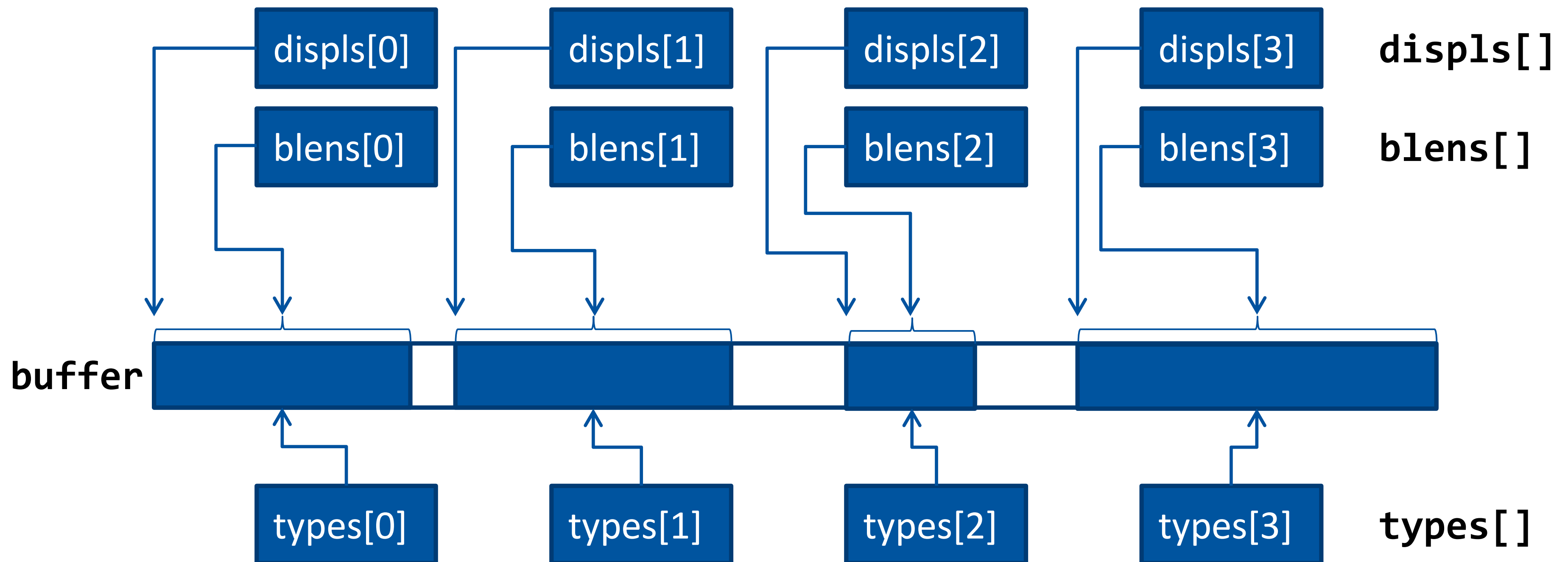
```
MPI_Type_create_struct (int count, int blens[], MPI_Aint displs[],
                        MPI_Datatype types[], MPI_Datatype *datatype)
```

- → **count:** number of blocks in the datatype
- → **blocklens[]:** 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

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)
```



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

RWTH AACHEN UNIVERSITY

- **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 single 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**

Summary: Derived Datatypes

- **Local handles describing how to access memory**
- **Can be mixed and matched on both sides of a communication operation as long as their type signatures match**
- **Lower and upper bound can be manipulated to account for padding at beginning and end**
- **Need to be committed before use in communication**
- **Language-specific handles need to be used in mixed-language applications**

Communicators