HPC.NRW

NATIONALES HOCHLEISTUNGS RECHNEN

EDIH Rheinland

WEST AI
KI-Servicezentrum

MPI in Small Bites

PPCES 2024

HPC.NRW Competence Network

THE COMPETENCE NETWORK FOR HIGH PERFORMANCE COMPUTING IN NRW.

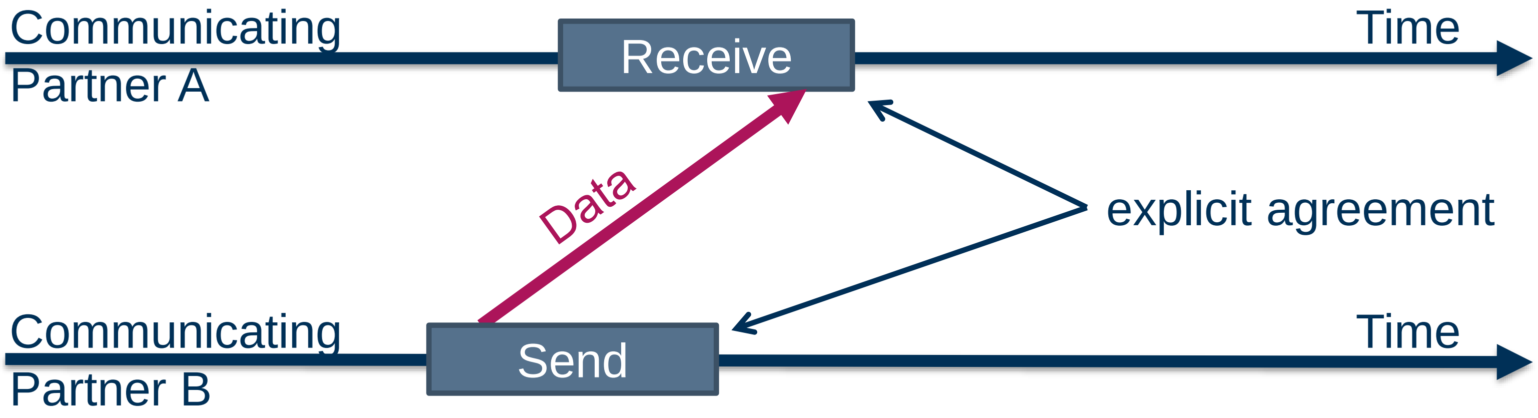
Blocking Point-to-Point Communication

HPC.NRW Competence Network

MPI in Small Bites

Point-to-Point Communication

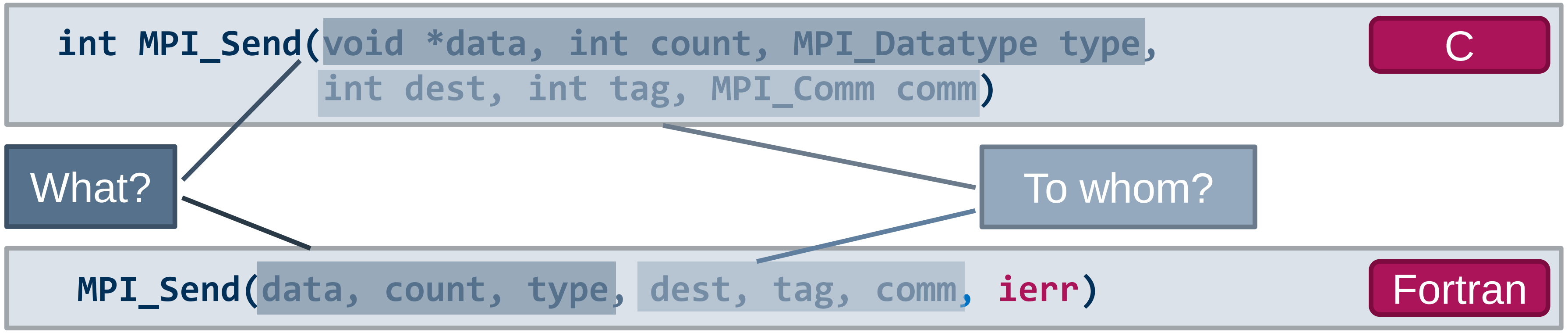
- The goal is to enable communication between processes that share no memory space



- Explicit message passing requires:
  - Send and receive primitives (operations)
  - Known addresses of both the sender and the receiver
  - Specification of what has to be sent/received

Sending Data

- Sending a message:



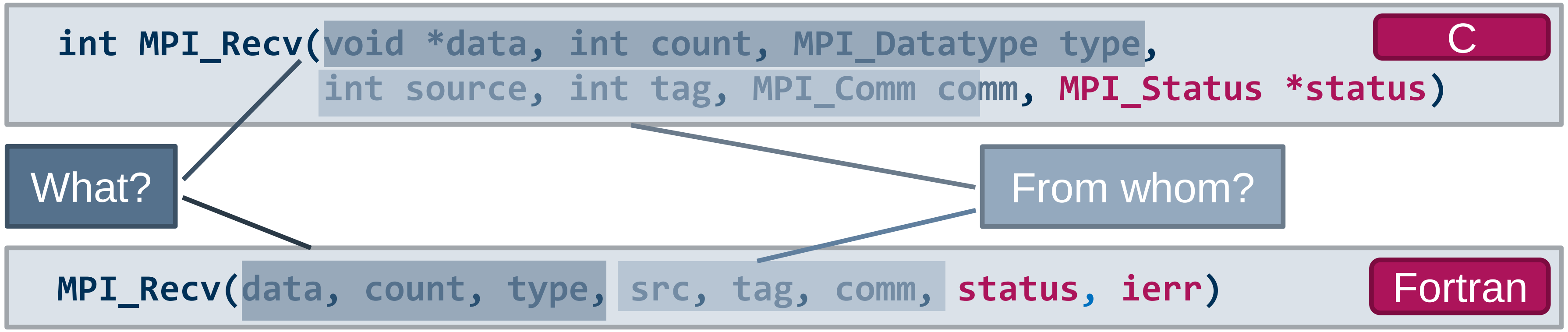
```
int MPI_Send(void *data, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm)
```

```
MPI_Send(data, count, type, dest, tag, comm, ierr)
```

- **data:** location in memory of the data to be sent
- **count:** number of elements of `type` to be sent
- **type:** handle of the *MPI datatype* of the buffer content
- **dest:** rank of the receiver
- **tag:** additional identification of the message
  ranges from 0 to `MPI_TAG_UB` (implementation dependant, but not less than 32767)
- **comm:** communication context (communicator handle)

Receiving Data

- Receiving a message:



```
int MPI_Recv(void *data, int count, MPI_Datatype type,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_Recv(data, count, type, src, tag, comm, status, ierr)
```

– **data:**	location of the receive buffer
– **count:**	size of the receive buffer in data elements
– **type:**	Handle of the MPI datatype of the data elements
– **source:**	rank of the sender or **`MPI_ANY_SOURCE`** (wildcard)
– **tag:**	message tag or **`MPI_ANY_TAG`** (wildcard)
– **comm:**	communication context
– **status:**	status of the receive operation or **`MPI_STATUS_IGNORE`**

MPI as an SPMD Environment

✓ Provide dynamic identification of all peers
  - Who am I and who else is also working on this problem?

✓ Provide robust mechanisms to exchange data
  - Whom to send data to / From whom to receive the data?
  - How much data?
  - What kind of data?
  - Has the data arrived?

Only local completion information available.

3. Provide synchronisation mechanisms
  - Have all processes reached same point in the program execution flow?

✓ Provide methods to launch and control a set of processes
  - How do we start multiple processes and get them to work together?

✓ Portability

Message Envelope and Matching

- Message matching is performed using the message envelope
- Send operation

```
int MPI_Send (void *data, int count, MPI_Datatype type,
              int dest, int tag, MPI_Comm comm)
```

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

- Receive operation

```
int MPI_Recv (void *data, int count, MPI_Datatype type,
              int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Message Envelope and Matching

- Correct reception of MPI messages is also dependent on the data type.
- Recall:

```
MPI_Send (void *data, int count, MPI_Datatype type,
          int dest, int tag, MPI_Comm comm)
```

```
MPI_Recv (void *data, int count, MPI_Datatype type,
          int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- Type signatures must match
  - May not be verified by MPI library (source for unpredictable errors!)
- One send operation is matched with one receive operation
  - Messages **do not** aggregate (no single receive for multiple sends)
  - Messages **do not** separate (no multiple receives for a single send)

Message Size and Status Object

- The receive buffer must be able to fit the entire message
  - send count ≤ receive count **OK** (check effective message length with status)
  - send count > receive count **ERROR** (message truncation)

- Message size inquiry:

```
MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)
```

  - Number of integral elements of type **`datatype`** in the message reference by **`status`**
  - If message size not divisible by size of given datatype size: **`MPI_UNDEFINED`**

Status Object Fields

HPC.NRW

- The MPI status object contains information about the message

C

```c
MPI_Status status;
…
status.MPI_SOURCE // message source rank
status.MPI_TAG    // message tag
status.MPI_ERROR  // receive status code
```

Fortran

```fortran
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
…
status(MPI_SOURCE)  ! message source rank
status(MPI_TAG)     ! message tag
status(MPI_ERROR)   ! receive status code
```

Fortran 2008

```fortran
TYPE(MPI_Status) :: status
…
status%MPI_SOURCE   ! message source rank
status%MPI_TAG      ! message tag
status%MPI_ERROR    ! receive status code
```

Checking for Message Availability (no threads)

- Checking for message:

```
MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status *status)
```

  - Do **NOT** use with multiple communicating threads (alternatives covered in a separate part)
  - Message is **not received**, separate call to **`MPI_Recv`** needed
  - Message envelope and size stored in **`status`** object

```
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
```

  - Checks for **any** message in the given communicator (wildcards)
  - Receive **must** use specific values from **`status`** to receive the inquired message

```
MPI_Status status;

MPI_Probe(MPI_AN                           M_WORLD, &status);
… allocate                          essage size …
MPI_Recv(bu          e, MPI_INT, MPI_ANY_SOURCE, 0,
         MPI_COMM_WORLD, &status);
```

Incorrect program

Checking for Message Availability (no threads)

- Checking for message:

```
MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status *status)
```

  - Do **NOT** use with multiple communicating threads (alternatives covered in a separate part)
  - Message is **not received**, separate call to **`MPI_Recv`** needed
  - Message envelope and size stored in **`status`** object

```
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
```

  - Checks for **any** message in the given communicator (wildcards)
  - Receive **must** use specific values from **`status`** to receive the inquired message

```
MPI_Status status;

MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
… allocate buffer based on message size …
MPI_Recv(buffer, size, MPI_INT, status.MPI_SOURCE, 0,
         MPI_COMM_WORLD, &status);
```

Use envelope data from status.

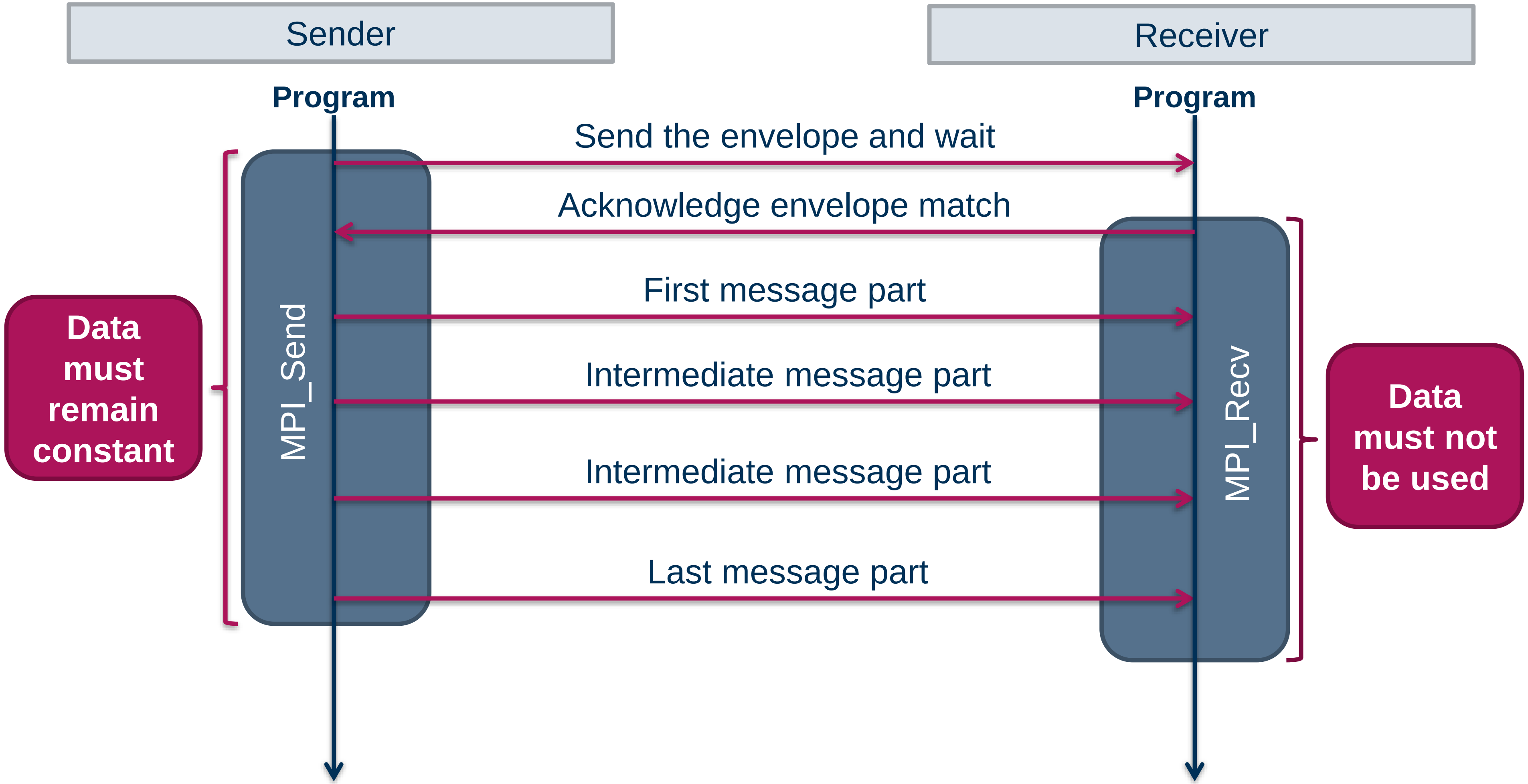
Operation Completion

- MPI operations complete locally once the message buffer is no longer in use by the MPI library and is thus free for reuse

- Send operations complete:
  - once the message is constructed ***and***
    - placed completely onto the network ***or***
    - buffered completely (by MPI, the OS, the network, …)
- Receive operations complete:
  - once the entire message has arrived and has been placed into the buffer

- Blocking MPI procedures only return once the corresponding operation has completed
  - **MPI_Send** and **MPI_Recv** are blocking

Blocking send (w/o buffering) and receive calls:



MPI Point-to-Point Send Modes

	Call	Semantics
Buffered / Asynchronous	`MPI_Bsend`	Uses extra (**user-provided**) buffer space to copy the message buffer and returns to the user. Message transfer may happen at a later point using the buffer.
Rendezvous / Synchronous	`MPI_Ssend`	Explicitly waits for the receiver to **start** the receive process
Standard	`MPI_Send`	May follow buffered (**library-internal buffer**) or synchronous semantics depending on implementation, input, and/or runtime situation
Ready	`MPI_Rsend`	Sender assumes the receive to be posted on remote process

MPI Point-to-Point Send Modes

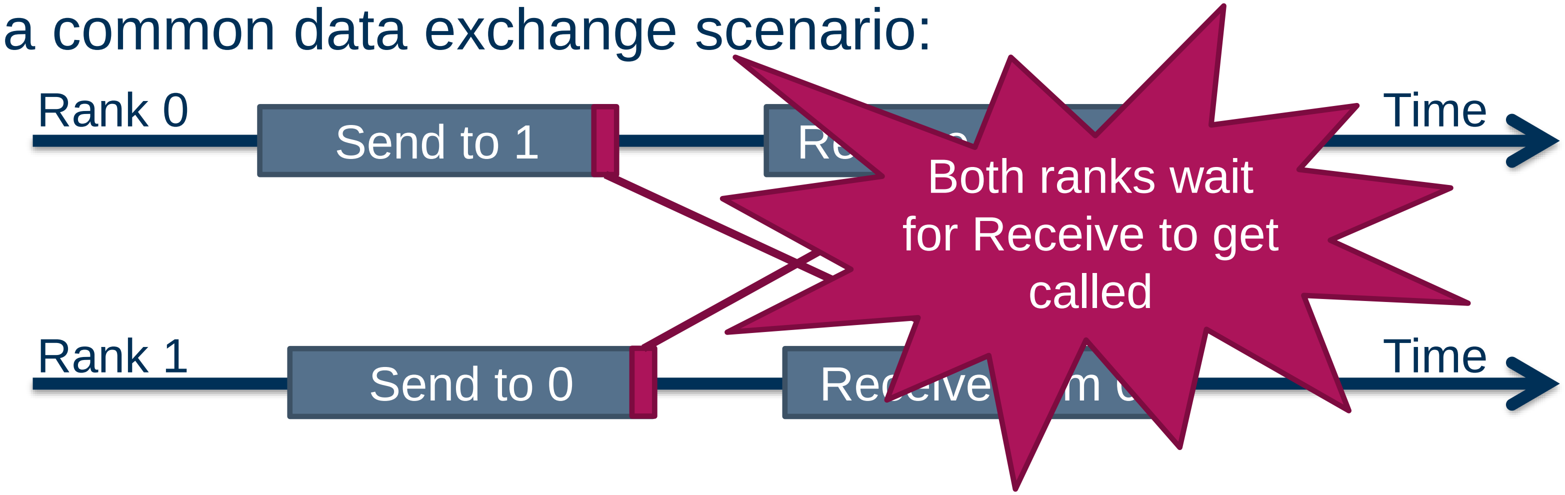
	Call	Semantics
Buffered / Asynchronous	`MPI_Bsend`	Uses extra (**user-provided**) buffer space to copy the message buffer and returns to the user. Message transfer may happen at a later point using the buffer.
Rendezvous / Synchronous	`MPI_Ssend`	Explicitly waits for the receiver to **start** the receive process
Standard	`MPI_Send`	May follow buffered (**library-internal buffer**) or synchronous semantics depending on implementation, input, and/or runtime situation
Ready	`MPI_Rsend`	Sender assumes the receive to be posted on remote process

Try to avoid! Needed guarantees hard to ensure, and often no benefit observable.

Deadlocks

- Both `MPI_Send` and `MPI_Recv` calls are blocking:
  - Standard send operation has two (implementation specific) modes of operation:
    - Buffering the message → Asynchronous completion
    - Waiting for the receiver to start receiving → Synchronous completion
  - **Never rely on any implementation-specific behaviour!**

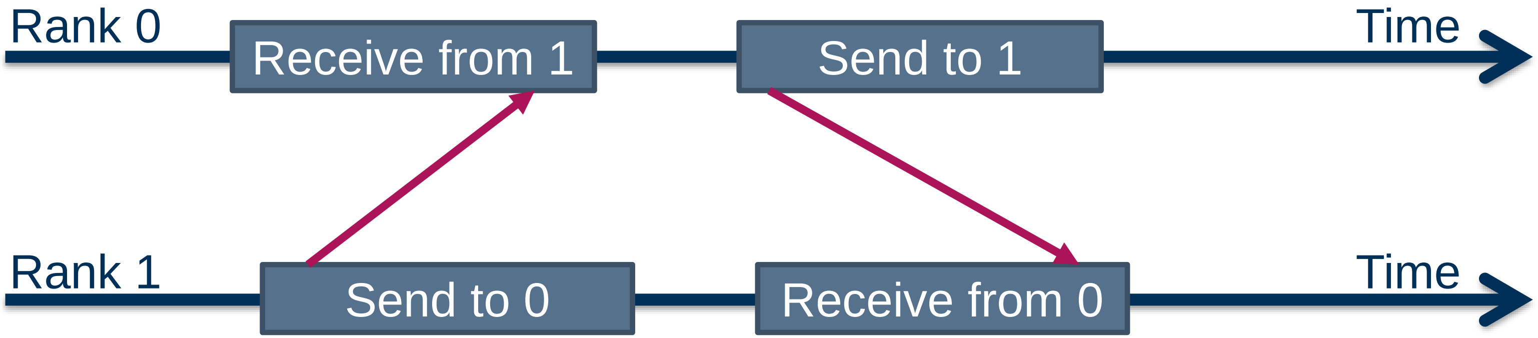
- Deadlock in a common data exchange scenario:



Deadlocks

- Both `MPI_Send` and `MPI_Recv` calls are blocking:
  - Standard send operation has two (implementation specific) modes of operation:
    - Buffering the message → Asynchronous completion
    - Waiting for the receiver to start receiving → Synchronous completion
  - **Never rely on any implementation-specific behaviour!**

- Deadlock prevention in a common data exchange scenario:



Combined Send and Receive

```
MPI_Sendrecv (void *senddata, int sendcount, MPI_Datatype sendtype,
              int dest, int sendtag, void *recvdata, int recvcount,
              MPI_Datatype recvtype, int source, int recvtag,
              MPI_Comm comm, MPI_Status *status)
```

- Sends one message and receives one message (in any order) without deadlocking (unless unmatched)
- Send and receive buffers **must not** overlap!

```
MPI_Sendrecv_replace (void *data, int count, MPI_Datatype datatype,
                      int dest, int sendtag, int source, int recvtag,
                      MPI_Comm comm, MPI_Status *status)
```

- Using the same memory location, elements count and datatype for both operations
- Often slower than `MPI_Sendrecv`

Message Ordering

- Order is preserved for point-to-point operations
  - in a given communicator
  - between any pair of processes

- Probe/receive returns the earliest matching message

- Order is **not** guaranteed for
  - Messages sent within different communicators
  - Messages arriving from different senders
  - Messages sent from different threads even with identical envelopes (logically concurrent)

Blocking Point-to-Point Communication Summary

- Communication primitives for data exchange between two processes
- Blocking communication returns on **local** completion
  - An operation is locally complete when arguments to MPI can be re-used / deallocated
- Message order guaranteed between two processes on the same communicator
- Different send modes exist to tweak communication pattern
  - Use ‘standard’ send (MPI_Send) if unsure or unless another mode is explicitly needed
- Use other means than ‘buffered’ mode to avoid deadlock (avoid the extra copy)
  - Combined send-receive calls
  - Explicit communication patterns (may reduce maintainability and impact performance)
  - Non-blocking communication (covered in another part)