GWDG – Kurs
Parallel Programming with  MPI

# Point-to-Point Communication

Oswald Haan

ohaan@gwdg.de

## Learning Objectives

- MPI Functions for Message Passing:

  `MPI_SEND, MPI_RECV, MPI_ISEND, MPI_IRECV, MPI_PROBE, MPI_WAIT, MPI_TEST`

- The Difference between Blocking and Non_Blocking Message Passing

- Avoiding Deadlocks in Message Passing

# Outline

- Message Passing in MPI
- Content of Messages: Data and Envelope
- Basic Functions for Blocking Message Exchange:
  **MPI_SEND, MPI_RECV**
- Distinguish Messages to be Received:

  **MPI_ANY_..., MPI_STATUS, MPI_PROBE**
- Semantic of Blocking Message Exchange

  Buffered vs. Synchronous Send
- Non-Blocking Message Exchange:

  Why and How, Request Objects
- Basic Functions for Non-Blocking Message Exchange:
  **MPI_ISEND, MPI_IRECV, MPI_WAIT, MPI_TEST**
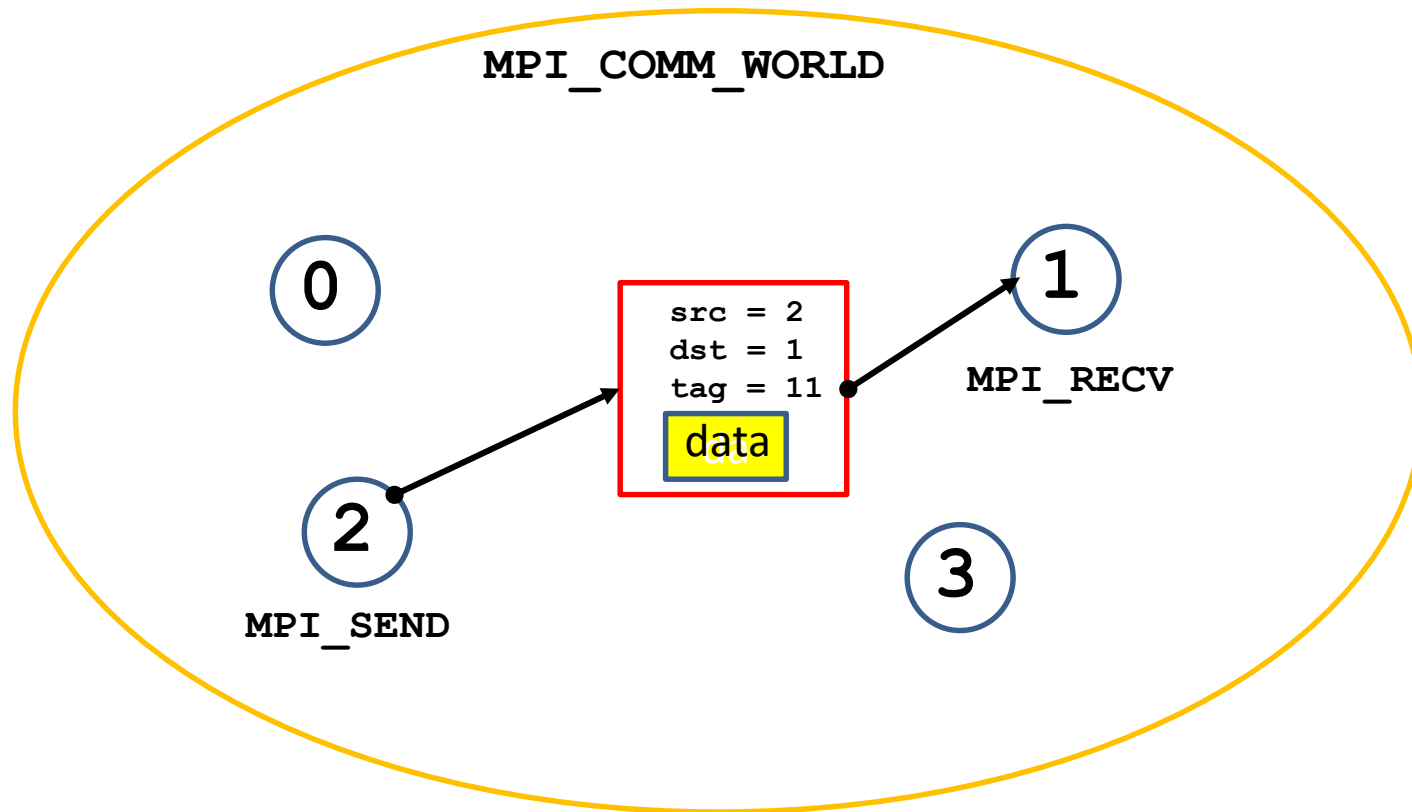- Semantic of Non-Blocking Message Exchange

# Message Passing

Basic mechanism of the message passing programming model: Transfer a message between two processes in **two steps**:

1.  On the source process: Sending the message from memory to the destination process

2.  On the destination process: Receiving the message from the source process to memory

step 1: send

system buffer

memory

step 2: receive

memory

source process

destination  process
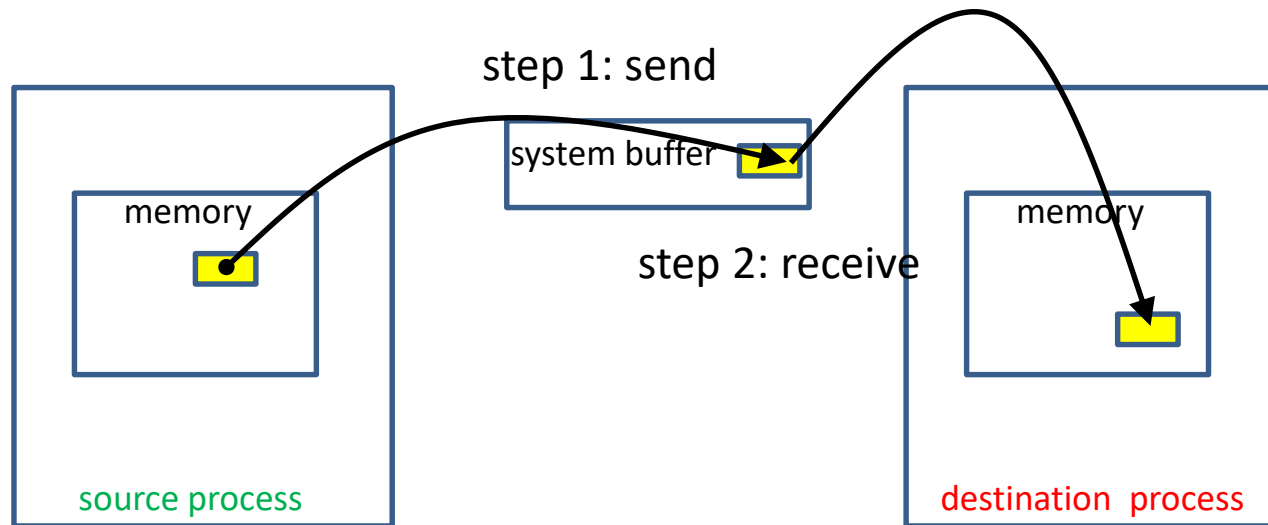
# MPI Setup for Message Passing



MPI_COMM_WORLD

0

```
src = 2
dst = 1
tag = 11
```

data

1

MPI_RECV

2

MPI_SEND

3

# MPI specification: Message Content

A MPI message contains:

   a number of elements of the **same** datatype.

MPI datatypes:

- basic datatype
  basic C types are different from basic Fortran types

- derived datatypes
  derived datatypes can be built up from basic or
  recursively from derived datatypes.

- datatype handles are used to describe
  the data layout of a datatype in memory.

# Basic MPI-Datatypes: Fortran

| MPI datatype | Fortran datatype |
|---|---|
| MPI_INTEGER | INTEGER |
| MPI_REAL | REAL |
| MPI_DOUBLE_PRECISION | DOUBLE PRECISION |
| MPI_LOGICAL | LOGICAL |
| MPI_CHARACTER | CHARACTER |
| MPI_COMPLEX | COMPLEX |

# Basic MPI Datatypes: C

| MPI datatype | C datatype |
|---|---|
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_CHAR | char |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED_CHAR | unsigned char |

# Basic MPI-datatypes: mpi4py

| MPI datatype | correspond to C datatype |
|---|---|
| MPI.INT | signed int |
| MPI.LONG | signed long int |
| MPI.FLOAT | float |
| MPI.DOUBLE | double |
| MPI.LONG_DOUBLE | long double |
| MPI.CHAR | char |

# MPI specification: Message Envelope

Each MPI message content is accompanied by a <span style="color:red">message envelope</span>, which contains additional information necessary or useful for the data transfer:

- the communicator, to which source and destination processes belong,

- the ranks of message-source and message-destination processes in this communicator,

- an identifier (tag), which can be used to differentiate between messages.

> The message tag must be specified by the user as an integer
> in the range [0, UB], where the value of the implementation dependent
> upper bound UB is equal to the MPI constant MPI_TAG_UB.
> The MPI standard requires UB to be at least 32767.

# Blocking Send Operation: MPI_SEND

C:

```
int MPI_Send(void *buf, int count,
                MPI_Datatype datatype,
                int dest, int tag, MPI_Comm comm)
```

Fortran:

```
MPI_SEND(buf, count, datatype, dest, tag,
            comm, ierr)
<type> buf(*)
INTEGER count, datatype, dest, tag, comm, ierr
```

# Blocking Send Operation: MPI_SEND

mpi4py:

      `comm.send(obj, dest, tag = 0 )`

      `obj` : Python object; `dest:` integer , `tag`: integer,

                                          optional (default: 0)

The Python object contains all information about
number and types of data-elements


      `comm.Send(ar, dest, tag = 0 )`

      `ar`   : NumPy array; `dest:` integer , `tag`: integer,

                                          optional (default: 0)

The NumPy array contains information about number and types of its
elements.
A section of the array can be selected by specifying:

      `ar[startindex:stopindex]`

# Blocking Receive Operation: MPI_RECV

C:

```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype,
             int src, int tag, MPI_Comm comm,
             MPI_Status *status)
```

Fortran:

```
MPI_RECV(buf, count, datatype, src, tag,
         comm, status, ierr)
<type> buf(*)
INTEGER count, datatype, src, tag, comm, ierr
INTEGER status(MPI_STATUS_SIZE)
```

# Blocking Receive Operation: MPI_RECV

mpi4py:

```
obj = comm.recv(buf=None, source= ANY_SOURCE,
                          tag= ANY_TAG, status=None)
```

**obj** : python object, **source**,  **tag**: integer

status: MPI.status() or None


```
comm.Recv(ar, source= ANY_SOURCE,
          tag= ANY_TAG, status=None)
```

**ar** : numpy array; **dest**, **tag**: integer

status: MPI.status() or None

# MPI_RECV  Restriction

The size of data to be recieved must be equal or greater than the size of the pending message:


`recvcount*size(recvtype)`  must be equal or greater than

`sendcount*size(sendtype)`


otherwise the program stops with an error

# „Wild Cards" for **MPI_RECV**

MPI_RECV copies a pending message into the memory of the calling process, starting at address **_buf_**  --

if the source and tag attributes of the message envelope
conform with the source and tag arguments
in the call to MPI_RECV.

With "wild card" arguments for source and/or for tag  a pending message will be received regardless of its attributed source and/or tag values.

C, Fortran :    **MPI_ANY_SOURCE, MPI_ANY_TAG**
mpi4py         **MPI.ANY_SOURCE, MPI.ANY_TAG**

# The `status` argument in `MPI_RECV`

The actual source and tag of the received message, and its actual size can be retrieved from the data-structure `status` returned as an argument in the call to MPI_RECV.

If the information from status is not needed, the argument can be ignored, saving memory space for this data structure and the processing cost to produce its content.

C, Fortran:      use  `MPI_STATUS_IGNORE`
                 as   `status`  argument

mpi4py:          use  `status = None`
                 or omit the  `status`  argument

# Retrieving Message Properties
## from the Status Argument `stat`

| | **C** | **Fortran** | **mpi4py** |
|---|---|---|---|
| type of **stat** | `MPI_Status stat` | `integer stat(MPI_STATUS_SIZE)` | `stat = MPI.Status()` |
| source | `stat.MPI_SOURCE` | `stat(MPI_SOURCE)` | `stat.Get_source()` |
| tag | `stat.MPI_TAG` | `stat(MPI_TAG)` | `stat.Get_tag()` |
| error | `stat.MPI_ERROR` | `stat(MPI_ERROR)` | `stat.Get_error()` |
| count | `MPI_Get_count (&stat,datatype,&count)` | `call MPI_GET_COUNT (stat,datatype,count,ierr)` | `stat.Get_elements (datatype)` |
| size | `---` | `---` | `stat.Get_size()` |

# Probing a Message before Receiving it

The properties of a pending message can be retrieved with the MPI_PROBE routine before actually copying the message to memory with MPI_RECV.

C:

```
MPI_Probe(source, tag, comm, &status);
```

Fortran:

```
call MPI_PROBE(source, tag, comm, status, ierr)
```

mpi4py:

```
res = comm.Probe(source=ANY_SOURCE, tag=ANY_TAG,
                 status=None)
```

   `res` :  Literal  **(True)**


**source** and **tag** can be wild cards!

Probing the information in status can be used to
- adjust the count of elements to be received to the size of the message
- select messages from particular sources / with particular tags .

# Semantics of Blocking Point-to-Point Communication

A call to MPI_SEND is **blocking**:

- It completes, if the message data from the send buffer are copied to another location and therefore the send buffer can be safely reused.
  - It is **local**, if the message data are copied to a temporary buffer in the source or destination process: it can complete before a matching MPI_RECV has been called in the destination process.
  - It is **non-local**, if the message data are copied directly to the receive buffer of a matching MPI_RECV call: it can complete only after a matching MPI_RECV has been called in the destination process.
- The choice between buffered and direct sending the message with MPI_SEND is **implementation dependent**

A call to MPI_RECV is **blocking**:

- It completes, if the data from a matching message are completely copied into the receive buffer.
  - It is always **non-local**: it can complete only after a matching MPI_SEND has been called in the source process.

# Explicit Modi for blocking Send

MPI provides three additional send routines with prescribed behavior:

**Buffered Send**:                                                **MPI_BSEND(sbuf,...**
A user defined temporary buffer `temp` must be provided with  a call to
          **MPI_BUFFER_ATTACH(temp,size)**

Is **local**: it completes, when `sbuf`  has been copied to `temp`  in the sending process

*Low latency  /  low bandwidth  (additional data copying)*


**Synchronous Send:**                                         **MPI_SSEND(sbuf,...**
Is **non-local:** it returns, when `sbuf`  has been copied to `rbuf` in the memory of the destination process

*High latency  (establishing communication channel)  /  high bandwidth*


**Ready Send:**                                                 **MPI_RSEND(sbuf,...**
Is **non-local**: it fails unless a matching MPI_RECV has been called in the destination process.


**Standard Send:**                                              **MPI_SEND(sbuf,...**
Buffered  for short, synchronous  for long messages

*Best of two worlds / but danger of deadlock*

# Syntax for Buffered Send

**Fortran**:

```
call MPI_BUFFER_ATTACH(temp, bsize, ierr)
call MPI_BSEND( sendbuf, count, datatype, ...
call MPI_BUFFER_DETACH(temp, bsize, ierr)
```

`bsize` is the size in Bytes of the temporary buffer to be attached.

The temporary buffer space `temp` must be declared to contain at least `bsize` Bytes.

`bsize` >= # Bytes for the send buffer  + MPI_BSEND_OVERHEAD

where MPI_BSEND_OVERHEAD, is a predefined  upper bound of Bytes used for administration of buffered send.

# Syntax for Buffered Send

C:

```
MPI_Buffer_attach(void* temp, int bsize);
MPI_Bsend( void* sendbuf, int count, ...);
MPI_Buffer_detach(void* temp_ptr, int* bsize);
```

**bsize** is the size in Bytes of the temporary buffer to be attached.

The temporary buffer space with address pointer **temp** must be declared to contain at least **bsize** Bytes,
**temp_ptr** is the pointer to the address in memory, where
the value of the address pointer **temp** is stored.

**bsize** >= # Bytes for the send buffer + MPI_BSEND_OVERHEAD

where MPI_BSEND_OVERHEAD, is a predefined upper bound of Bytes used for administration of buffered send.

# Syntax for Buffered Send

**mpi4py**:

```
sendbuf = numpy.ones(nl,dtype = np.float64)
ntemp = nl + MPI.BSEND_OVERHEAD/8
temp = np.empty(ntemp,dtype = np.float64)


MPI.Attach_buffer(temp)
comm.Bsend(sendbuf,dest,tag=0)
MPI.Detach_buffer()
```
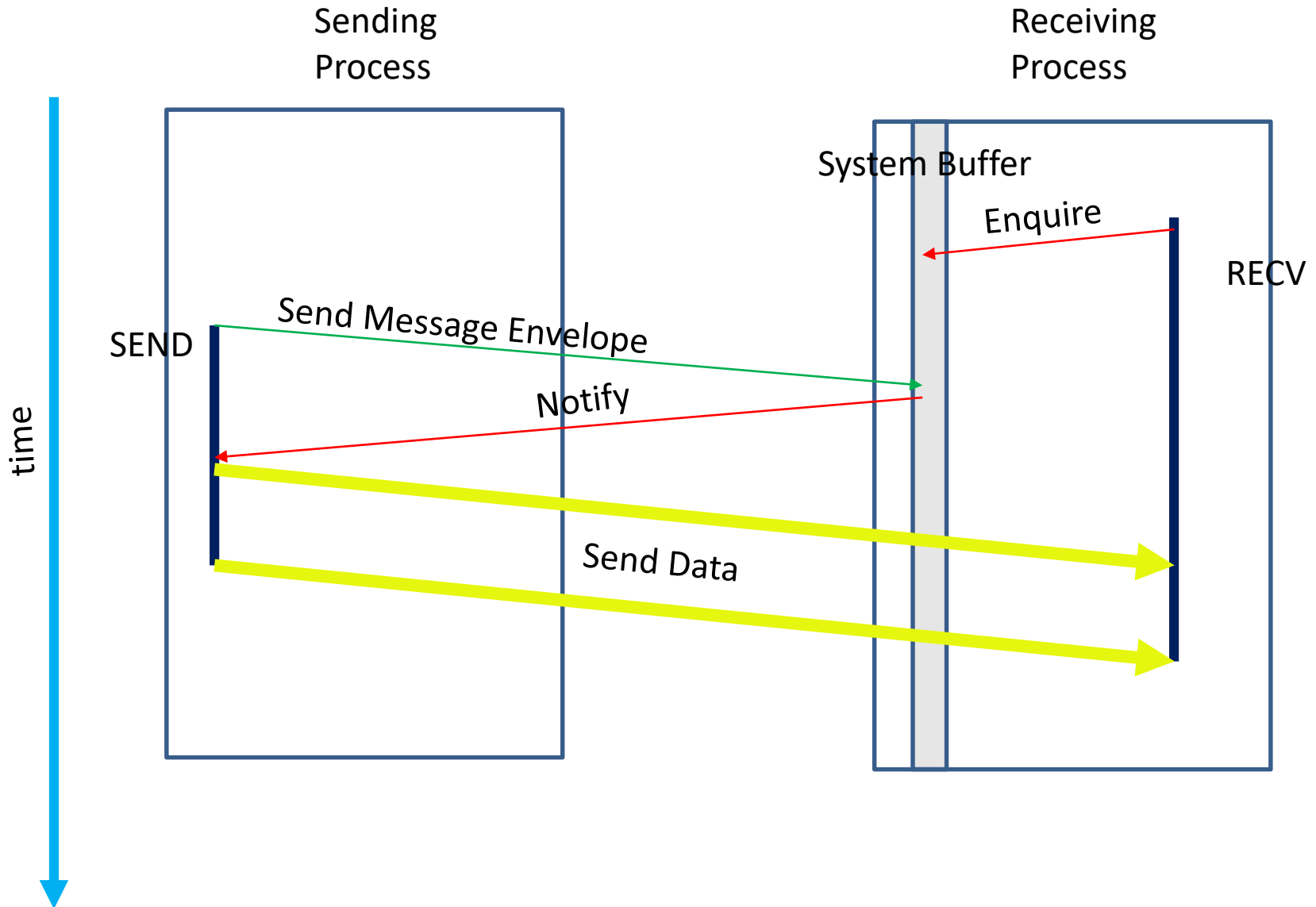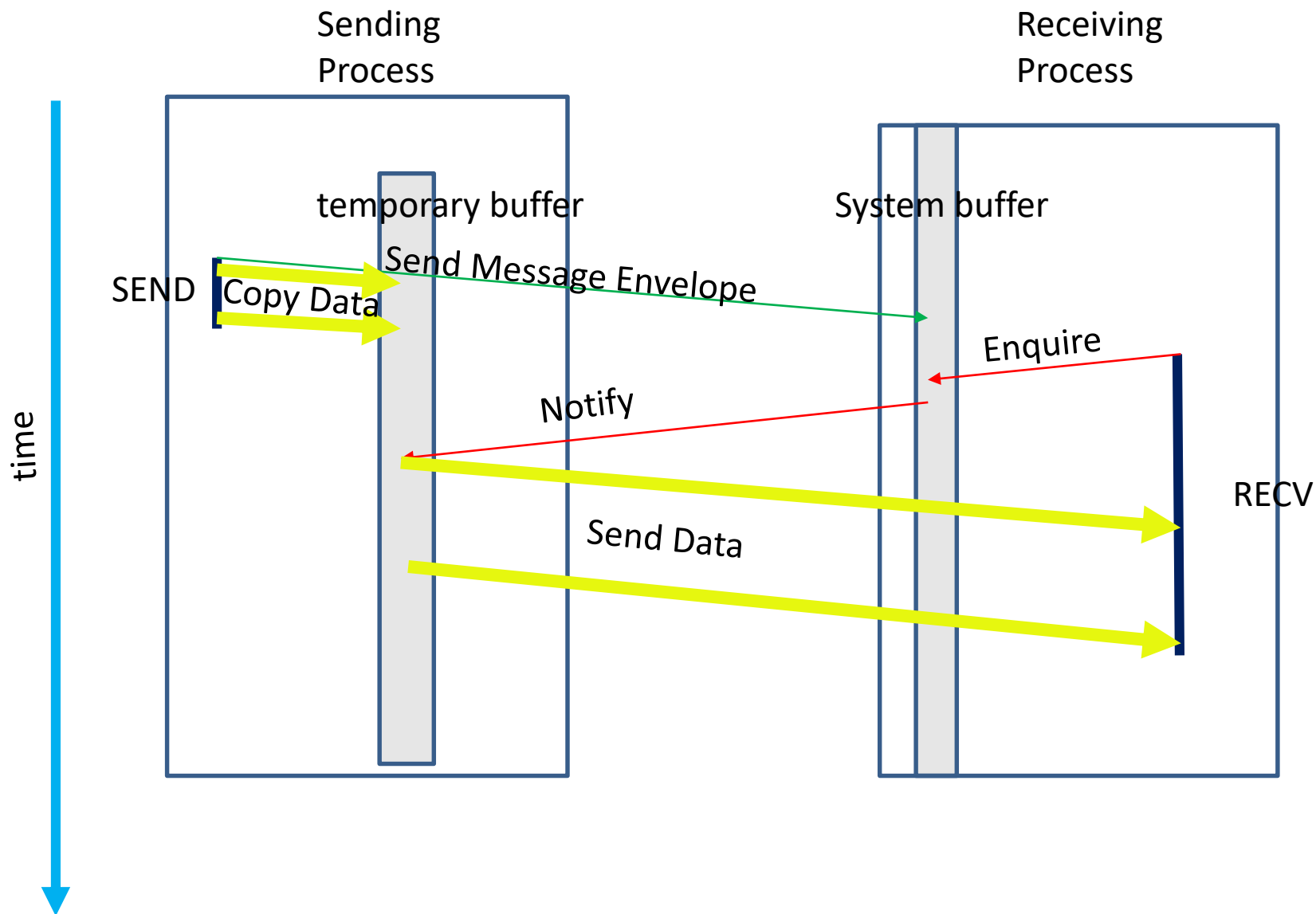
#Bytes of temp  =  # Bytes for the send buffer  + `MPI.BSEND_OVERHEAD`
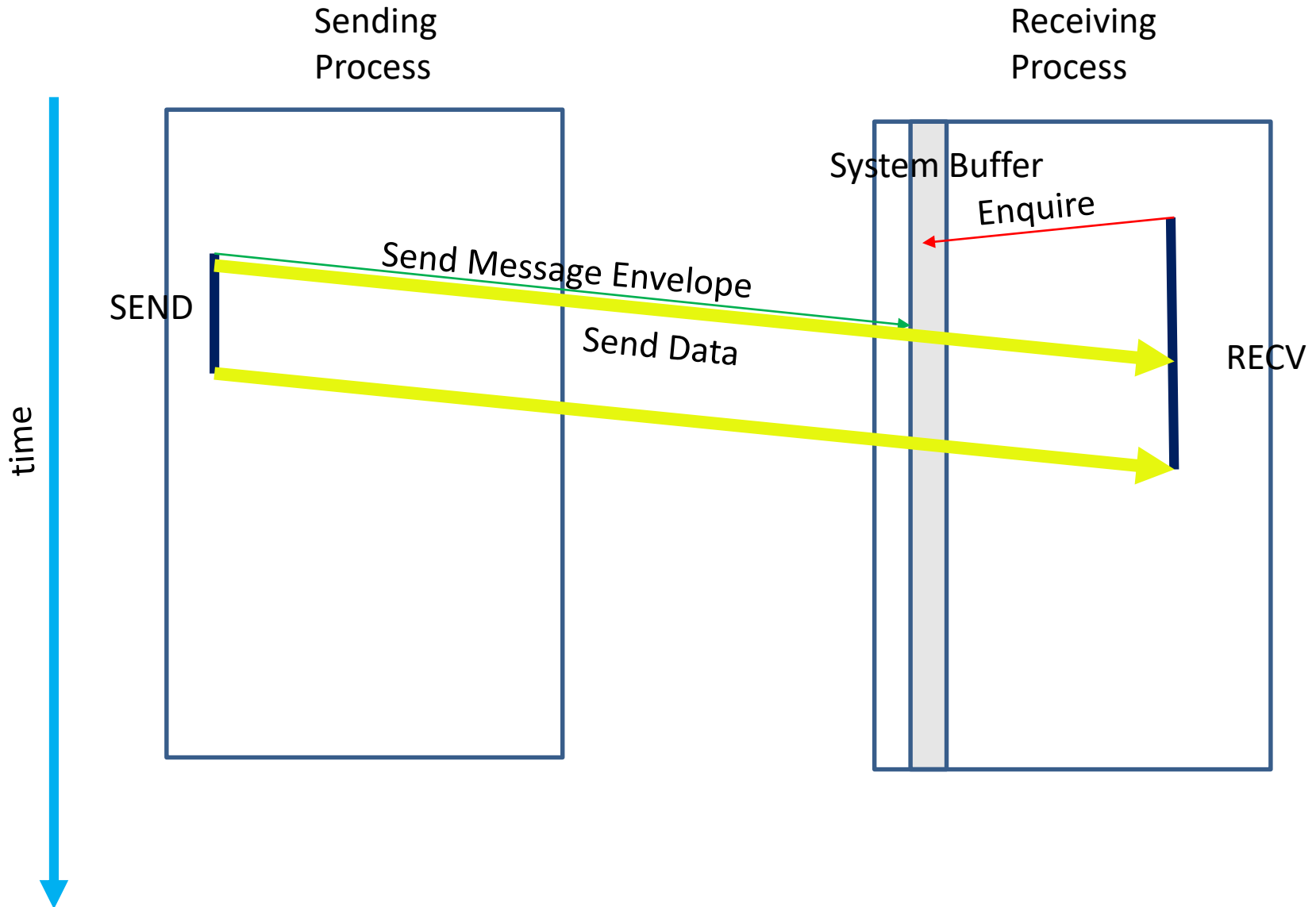where MPI.BSEND_OVERHEAD, is a predefined  upper bound of Bytes used for administration of buffered send.

# Synchronous Send

# Buffered Send



Sending Process

Receiving Process

time

temporary buffer

System buffer

SEND

Copy Data

Send Message Envelope

Enquire

Notify

RECV

Send Data

# Ready Send

Sending Process

Receiving Process

System Buffer

Enquire

SEND

Send Message Envelope
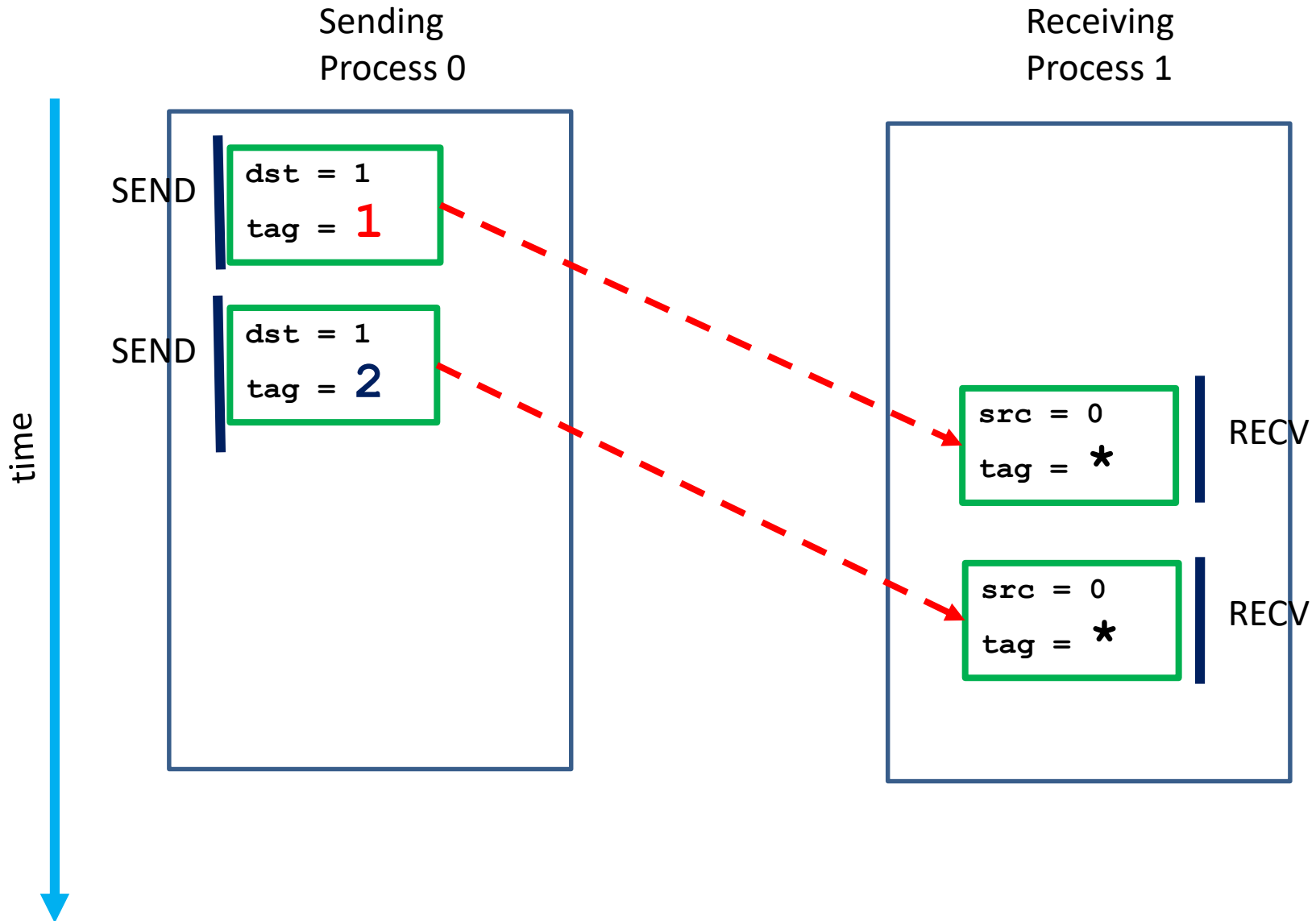
Send Data

RECV

time

# Message Order Preservation

Rule for multiple messages on the same connection,

i.e. same communicator, source, and destination rank:

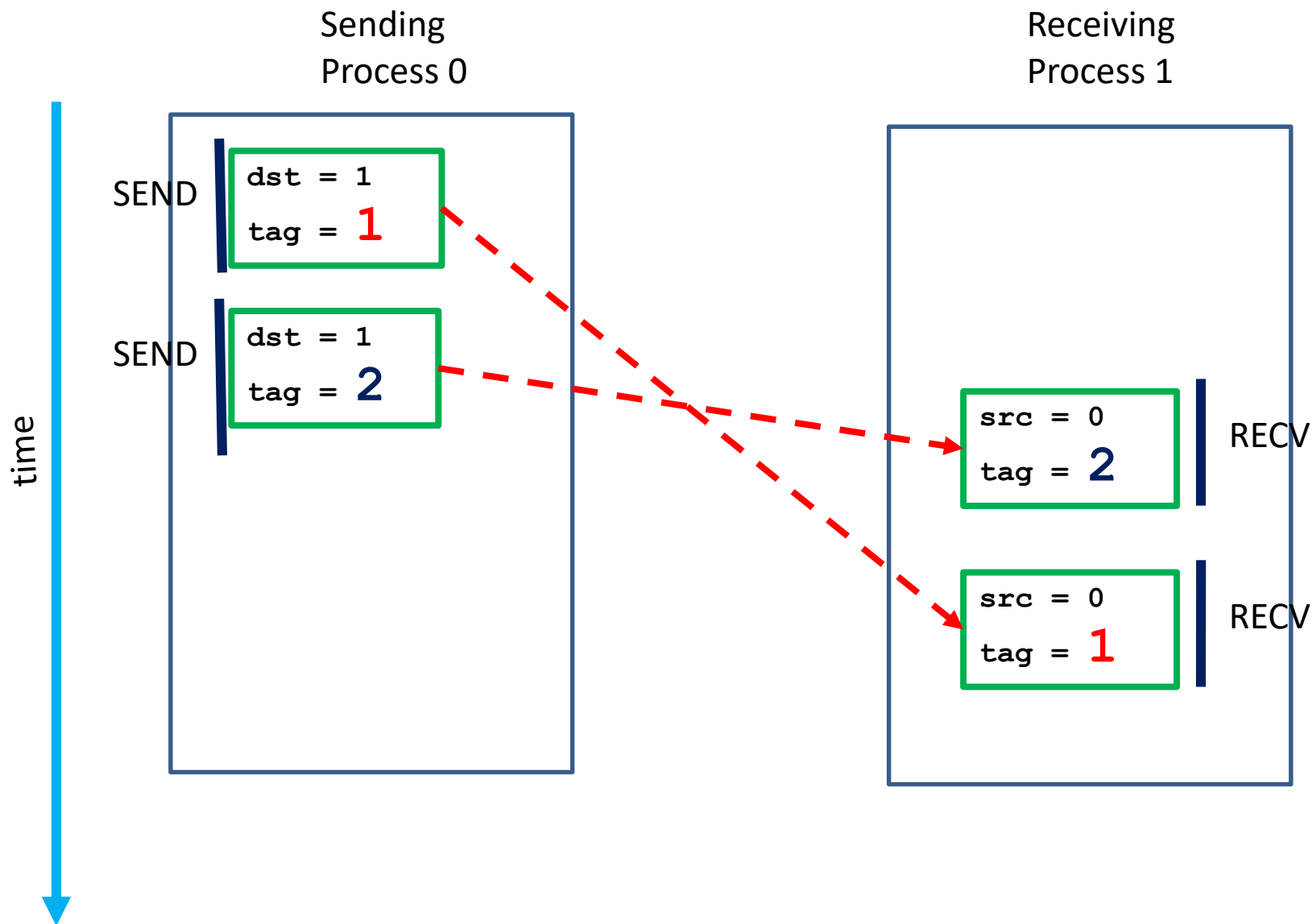•If several receives match multiple messages, then the sending order is preserved.

**Messages do not overtake each other.**

• This is true even for non-synchronous sends.
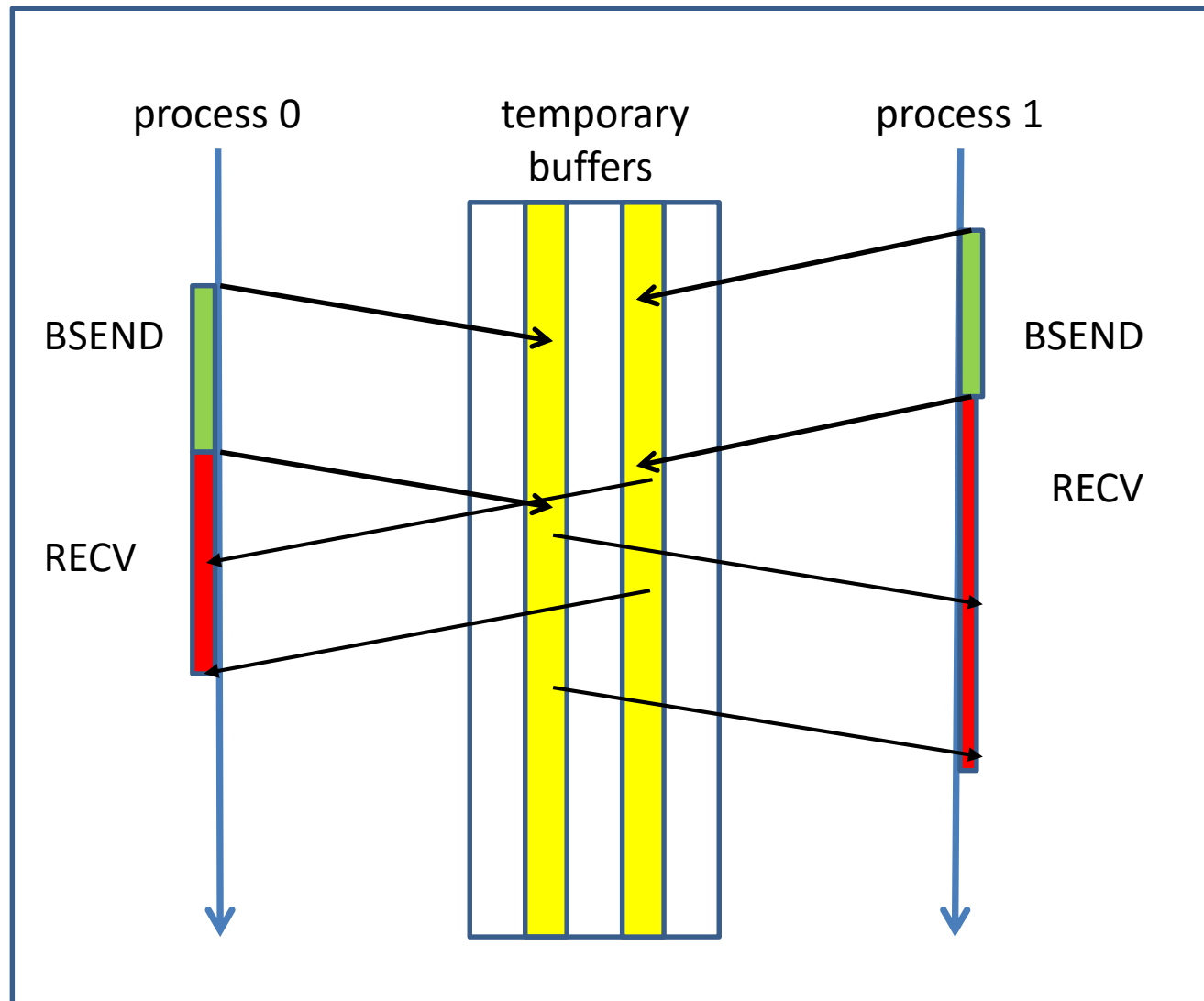
# Receives Match Both Sends

Sending
Process 0

Receiving
Process 1

time

SEND
```
dst = 1
tag = 1
```

SEND
```
dst = 1
tag = 2
```

```
src = 0
tag = *
```
RECV

```
src = 0
tag = *
```
RECV

# Each Receive Matches One Send

Sending
Process 0

Receiving
Process 1

time

SEND
```
dst = 1
tag = 1
```

SEND
```
dst = 1
tag = 2
```

```
src = 0
tag = 2
```
RECV

```
src = 0
tag = 1
```
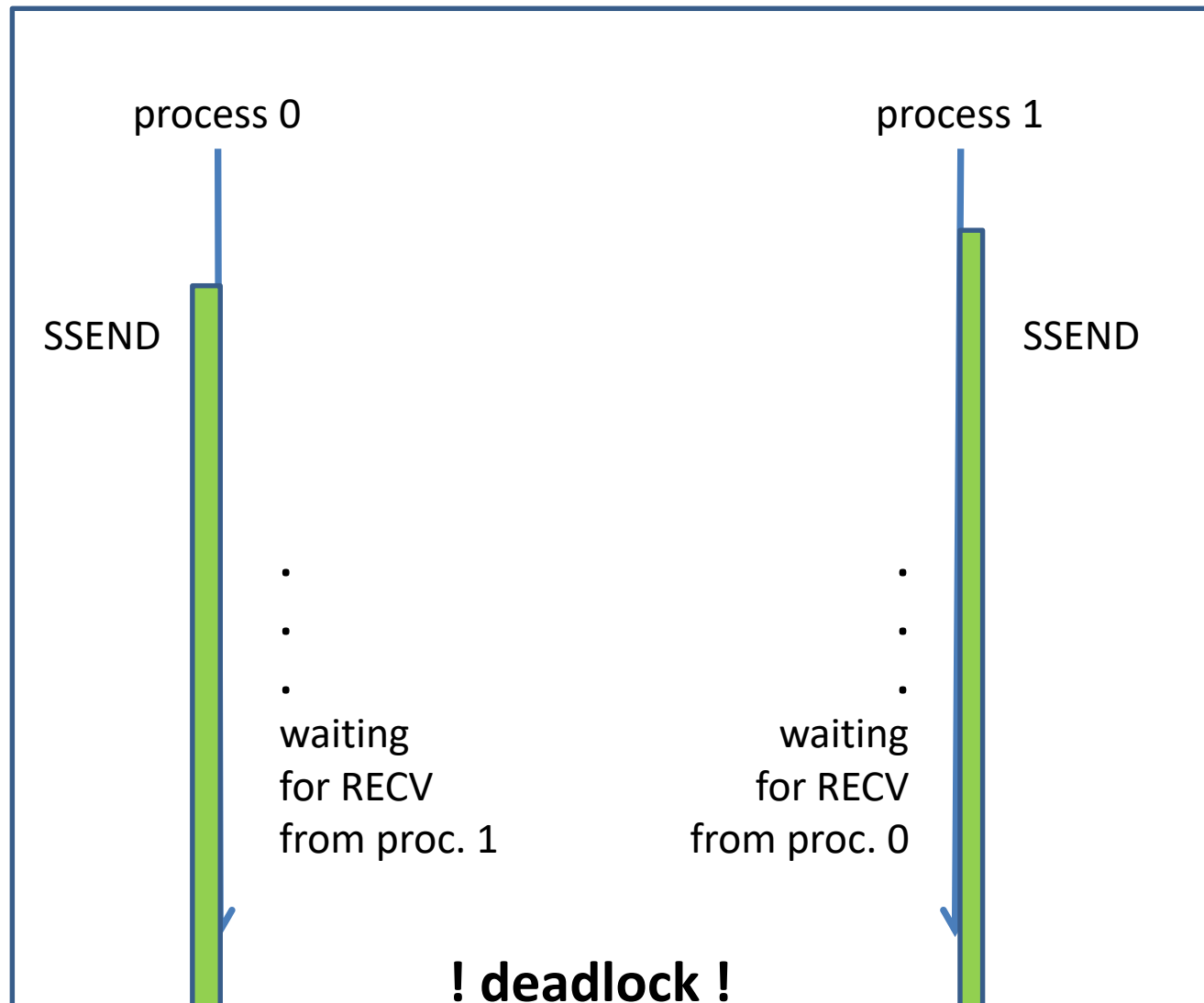RECV

# Message Exchange

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag,
                    comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag,
                    comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag,
                    comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag,
                    comm, status, ierr)
END IF
```
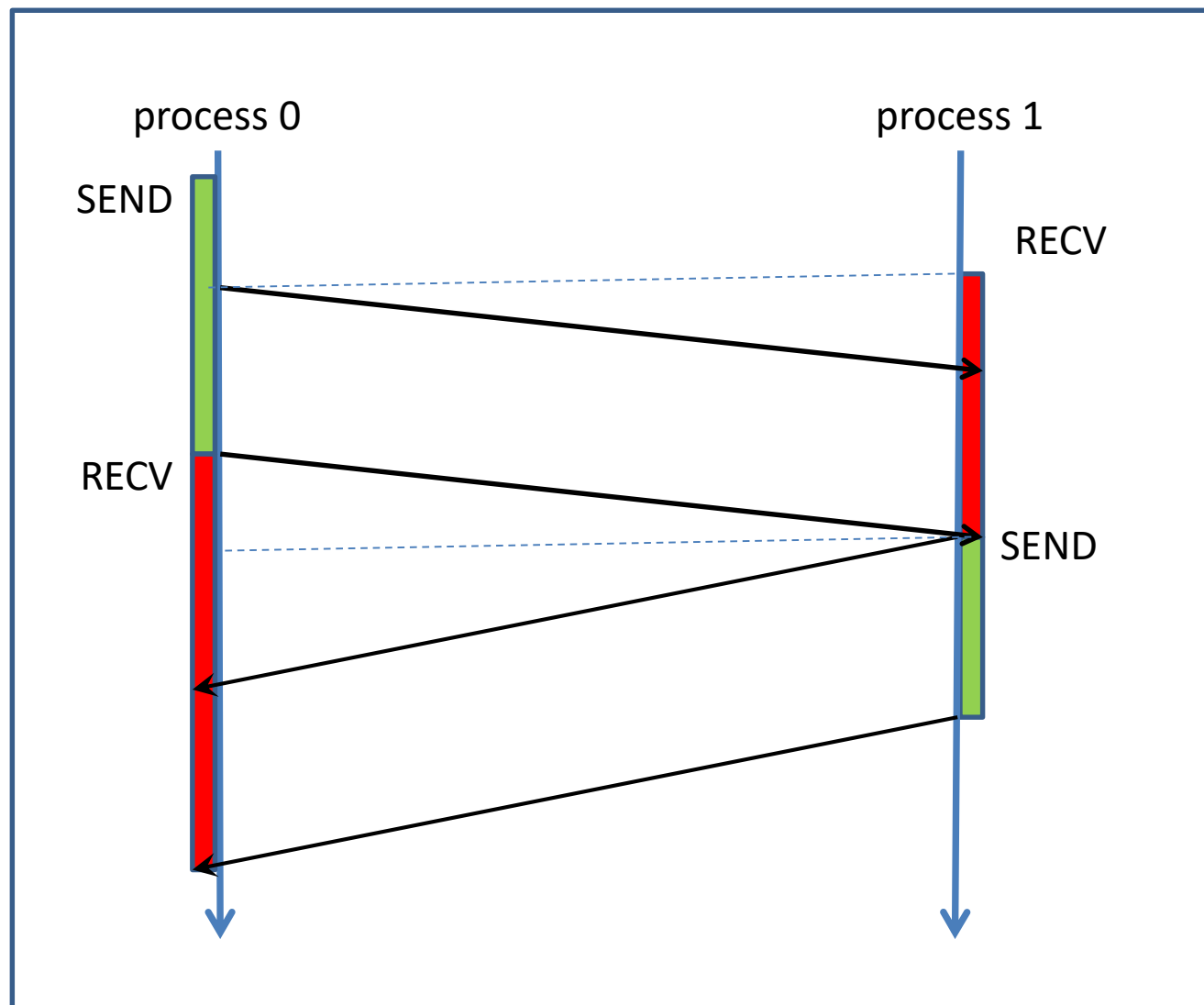
# Message Exchange with Buffered SEND

# Message Exchange with Synchronous SEND

process 0

process 1

SSEND

SSEND

.
.
.
waiting
for RECV
from proc. 1

.
.
.
waiting
for RECV
from proc. 0

**! deadlock !**

Parallel Programming with MPI

# No Deadlock with Reversed Order

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
     CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag,
                     comm, ierr)
     CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag,
                     comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
     CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag,
                     comm, status, ierr)
     CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag,
                     comm, ierr)
END IF
```

# Message Exchange without Deadlock

# Message Exchange with MPI_SENDRECV

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
        ipn = 1
ELSE IF (rank.EQ.1) THEN
        ipn = 0
END IF
call MPI_SENDRECV( sendbuf, count, MPI_REAL, ipn, tag
                 , recvbuf, count, MPI_REAL, ipn, tag
                 , comm, status, ierr )
```

# Non-Blocking
# Point-to-Point Communication

# Blocking vs. Non-Blocking

## Blocking

Calls to blocking point-to point send and receive routines return after completion of the intended data movement:

- The message passing is completed in the sending process, when the data from the send buffer are copied to a different place (either to a temporary buffer or to the receive buffer  in the receiving process)

- The message passing is completed in the receiving process, when the data from the send buffer are stored in the receive buffer.

## Non-Blocking

Calls to non-blocking routines return immidiately.
Completion of the communication must be monitored by
MPI routines for testing or waiting for completion.

- User responsibility:
  data in send or receive buffer are not to be used until the completion ot the transaction has been assured

# ***WHY*** Non-Blocking Communication

- **Overlap** communication and working with  data not involved in the communication

- **Break** deadlocks and serialization

- **Avoid** temporary buffering of messages

# *HOW* Non-Blocking Communication

Split communication into two operations

1. **Posting**: initiate non-blocking communication routine
   - returns **i**mmediately
   - routine name starting with MPI_**I** …

   *compute and communicate data*
   *not involved in the posted routine*

2. **Waiting**: monitor the progress of the posted communication
   - MPI_WAIT blocks until completion of the communication
   - MPI_TEST returns the status of the communication

# The Request Object

A **request object** is used to identify a non-blocking communication

A **request handle** refering to the **request object** generated with the call to `MPI_I`… is returned in an argument in the calling sequence
The **request handle** argument in the MPI_WAIT and MPI_TEST routines identifies the specific communication process to be monitored

# Nonblocking Communication

Nonblocking communication calls **cooperate** with blocking communication calls:

A message sent by MPI_ISEND can be received by MPI_RECV

A message sent by MPI_SEND can be received by MPI_IRECV

# Syntax for Non-Blocking SEND

C:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm, MPI_Request *req)
```

Fortran:

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, req, ierr)
      <type> buf(*)
      INTEGER count, datatype, dest, tag, comm, req, ierr
```

mpi4 py:

```
req=comm.isend(buf,dest,tag=0)
      buf : any Python object
req=comm.Isend(buf,dest,tag=0)
      buf : buffer-like object, e.g. numpy-array
```

# Syntax for Non-Blocking RECV

**C**:

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
       int source, int tag, MPI_Comm comm, MPI_Request *req)
```

**Fortran**:

```
MPI_IRECV(buf, count, datatype, source, tag, comm, req, ierr)
      <type> buf(*)
      INTEGER count, datatype, source, tag, comm, req, ierr
```

**mpi4 py**:

```
req=comm.irecv(buf=None, source=ANY_SOURCE, tag=ANY_TAG)
      buf : optional
req=comm.Irecv(buf, source=ANY_SOURCE, tag=ANY_TAG)
      buf : buffer-like object, e.g. numpy-array
```

*No status argument for posting a nonblocking receive*

# Waiting for Completion

C:

```
int  MPI_Wait( MPI_Request *req
                   , MPI_Status *status )
```

Fortran:

```
MPI_WAIT(req, status, ierr)
INTEGER req, status(MPI_STATUS_SIZE), ierror
```

mpi4py:

```
buf=MPI.Request.wait(req, status=None) or
buf=req.wait(status=None)
        buf: any Python object

MPI.Request.Wait(req, status=None) or
req.Wait(status=None)
```

- WAIT is blocking until the operation, which created `req,` has completed.
- If `req` was created by `IRECV, status` contains information about sender, tag and length of the message; if it was created by `ISEND`, `status` is undefined

# Testing for Completion

C:

```
int MPI_Test( MPI_Request *req, int *flag
                , MPI_Status *status )
```

Fortran:

```
MPI_TEST(req, flag, status, ierr)
    INTEGER req, status(MPI_STATUS_SIZE), ierror
    LOGICAL flag
```

mpi4py:

```
flag = MPI.Request.Test(req, status=None)
[flag,obj] = MPI.Request.test(req, status=None)
```

- TEST is non-blocking.
- returns **flag=true (1 for C)**, if the operation, which created **request**, has completed.
- returns **flag=false (0 for C)**, if the operation, which created **request**, has not yet completed.
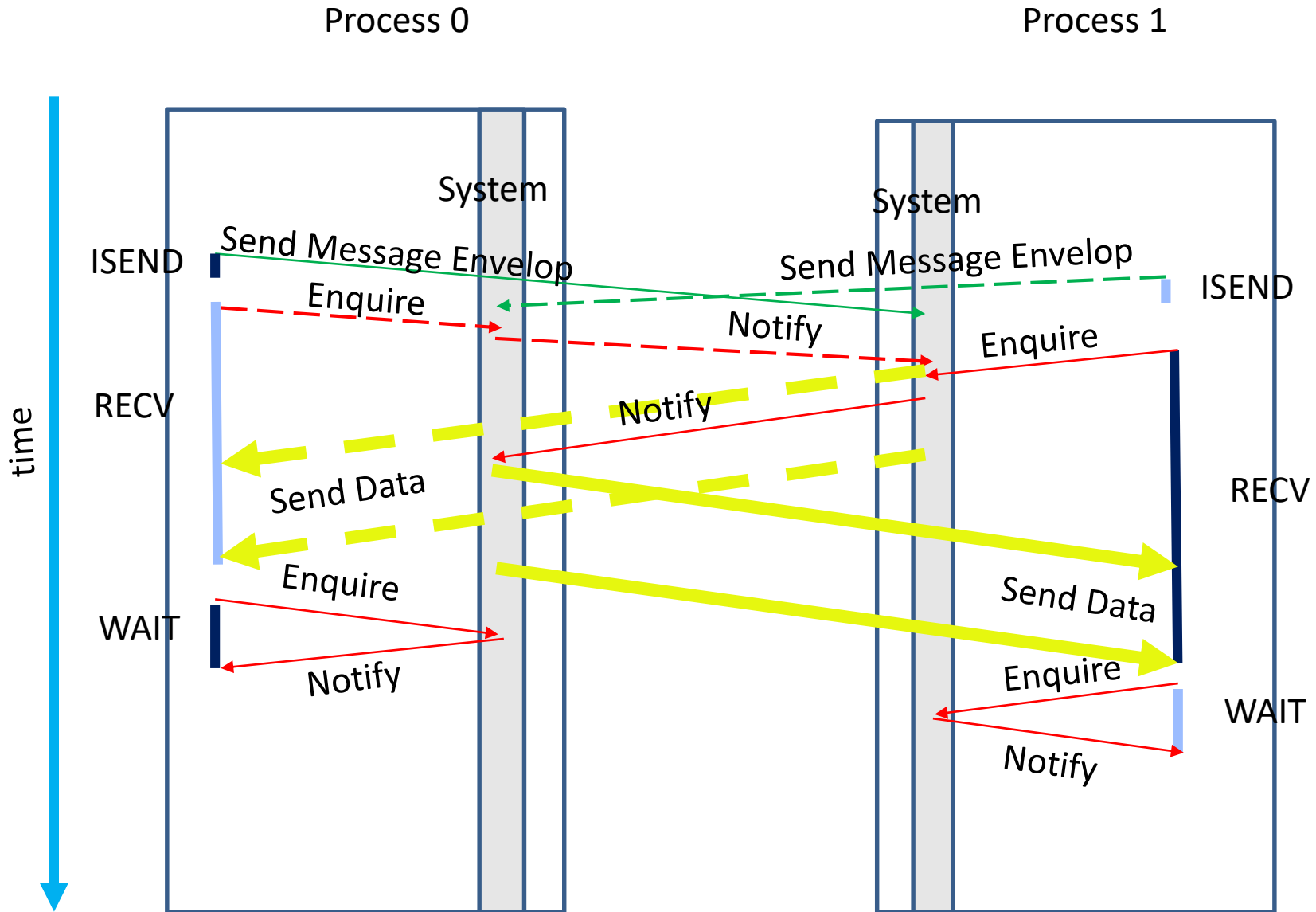- If the operation has completed, **status** is as in **MPI_WAIT**

# ISEND

Parallel Programming with MPI

# IRECV

Parallel Programming with MPI
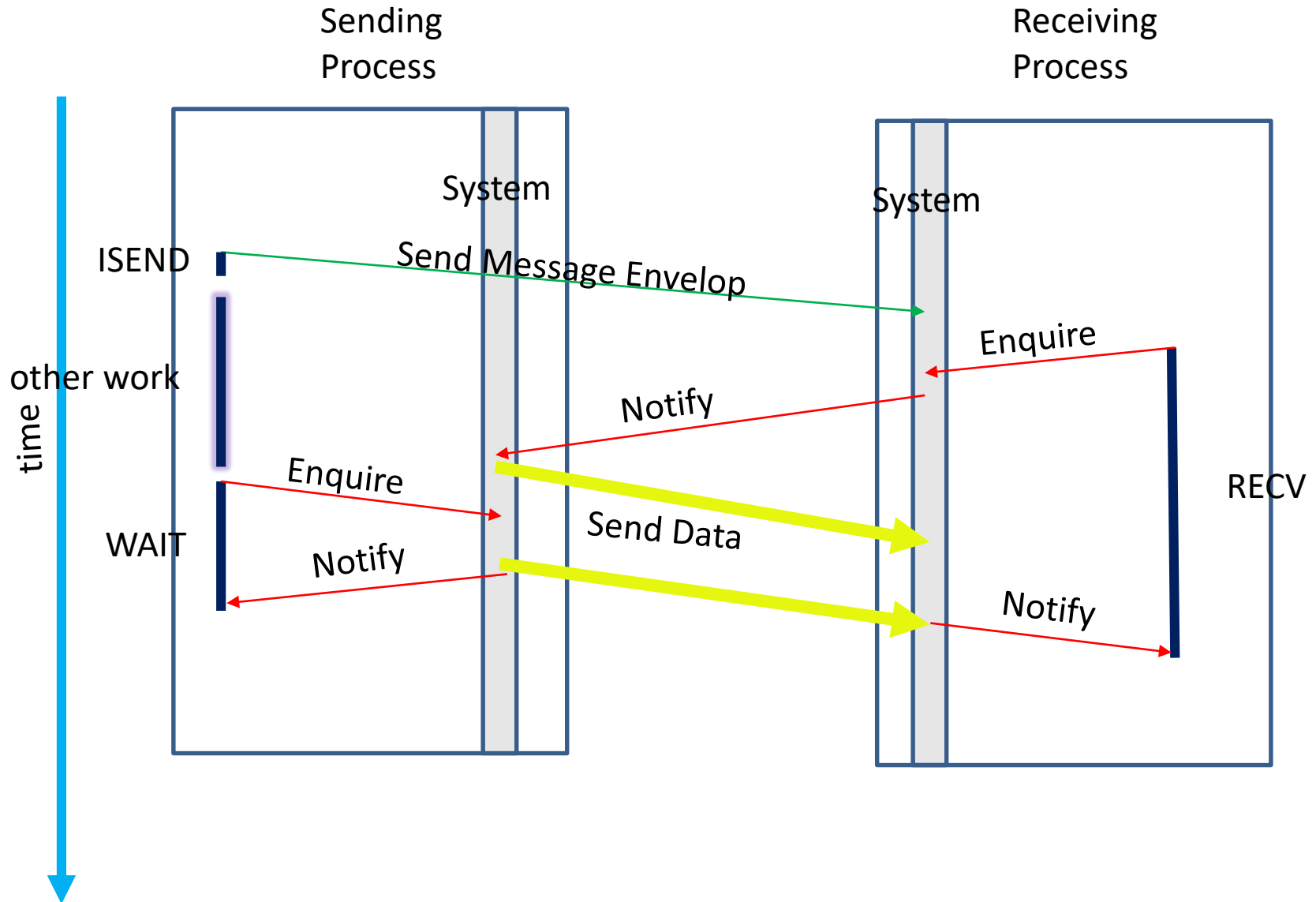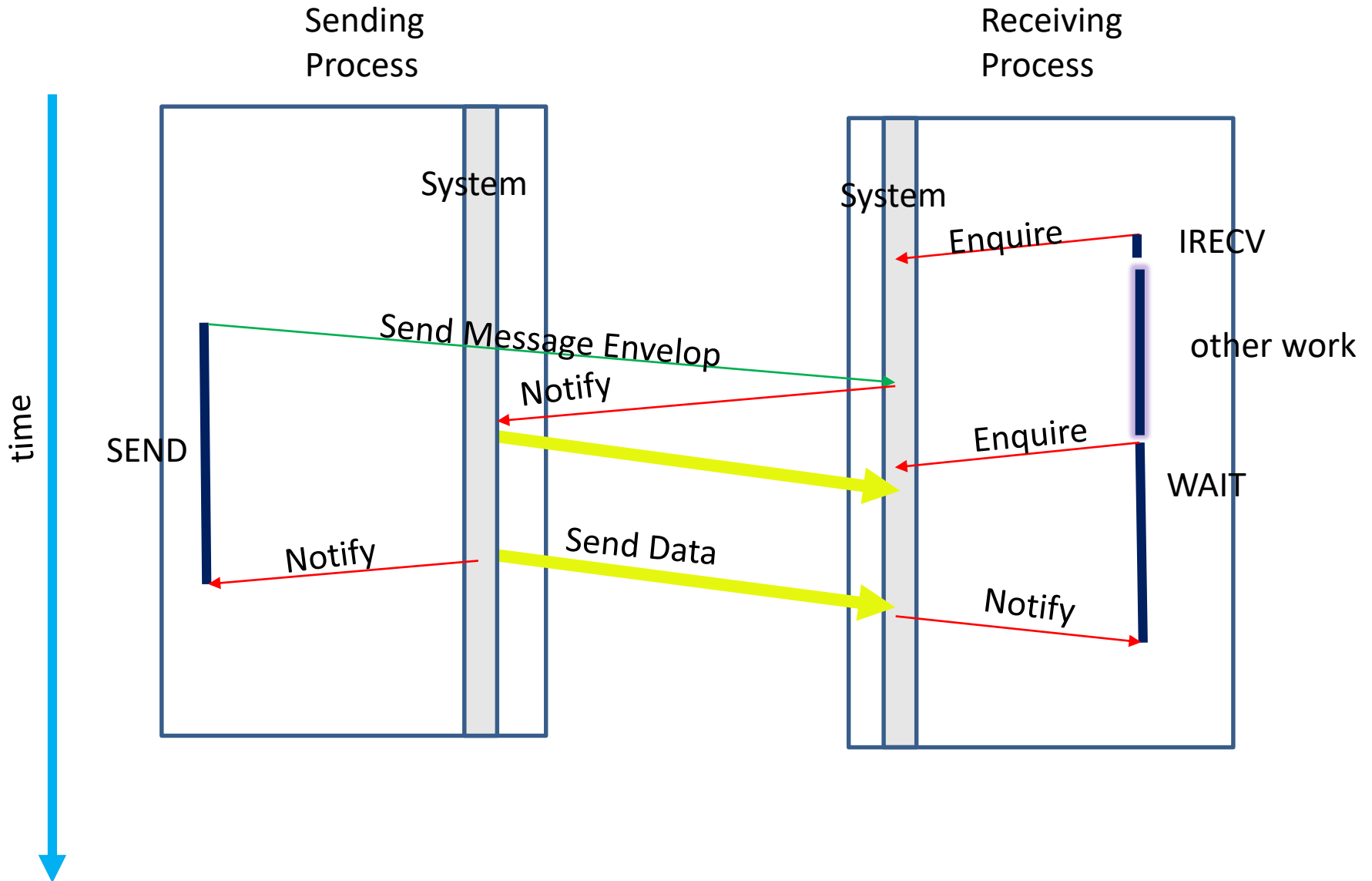
# Message Exchange with ISEND

# Summary for python Syntax

Communucation of **buffer-like object** (e.g. numpy-array) :

| | |
|---|---|
| *Nonblocking send* | `req=comm.Isend(buf,dest,tag=0)` |
| *Nonblocking receive* | `req=comm.Irecv(buf,source,tag=0)` |
| *Wait for completion* | `MPI.Request.Wait(req, status)` |
| ***Test** for completion* | `flag = MPI.Request.Test(req, status)` |

`Communicated object `**`buf`**` as parameter in Isend and Irecv`

# Summary for python Syntax

Communcation of **general python object** :

*Blocking send*        `comm.send(`**`buf`**`,dest,tag)`
*Blocking receive*      **`buf`**`=comm.recv(source,tag)`

```
Communicated object buf as parameter in      send
                        as return value in    recv
```

*Nonblocking send*      **`req`**`=comm.isend(`**`buf`**`,dest,tag)`
*Nonblocking receive*     **`req`**`=comm.irecv(source,tag)`
*Wait for completion*     **`buf`**`=MPI.Request.wait(`**`req`**`,status)`
***Test*** *for completion*    `[flag,`**`buf`**`]=MPI.Request.test(`**`req`**`,status)`

```
Communicated object buf as parameter in      isend
                        as return value in    wait or test
```

# Summary for python Syntax

Communcation of **buffer-like object** (e.g. numpy-array) :
:

| | |
|---|---|
| *Blocking send* | `comm.Send(`**`buf`**`,dest,tag)` |
| *Blocking receive* | `comm.Recv(`**`buf`**`,source,tag)` |

```
Communicated object buf  as parameter in       Send
                         as parameter in       Recv
```

| | |
|---|---|
| *Nonblocking send* | **`req`**`=comm.Isend(`**`buf`**`,dest,tag)` |
| *Nonblocking receive* | **`req`**`=comm.Irecv(`**`buf`**`,source,tag)` |
| *Wait for completion* | `MPI.Request.Wait(`**`req`**`,status)` |
| *Test for completion* | `flag=MPI.Request.Test(`**`req`**`,status)` |

```
Communicated object buf  as parameter in       Isend
                         as parameter in       Irecv
```