Parallel programming / computation

Sultan ALPAR

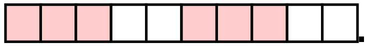
s.alpar@iitu.edu.kz

IITU

Lecture 3

Messages and Point-to-Point Communication

Messages

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
  - Basic datatype.
  - Derived datatypes .
- Derived datatypes can be built up from basic or derived datatypes.
- C types are different from Fortran types.
- Datatype handles are used to describe the type of the data in the memory.

Example: message with 5 integers

2345	**654**	**96574**	**-12**	**7676**

- **Python:** messages can be stored in
  - Objects → using send(...), recv(…), ... mpi4py routines → slow object serialization
  - Buffers as numPy arrays → using Send(…), Recv(…), … → fast communication

Lower-case methods

Upper-case methods

For other alternatives, see MPI\tasks\PY\Ch13\mpi_io_exa1-skel.py

C

MPI Basic Datatypes — C / C++

MPI Datatype handle	C datatype	Remarks
MPI_CHAR	char	Treated as printable character
MPI_SHORT	signed short int	
MPI_INT	signed int	
MPI_LONG	signed long int	
MPI_LONG_LONG	signed long long	
MPI_SIGNED_CHAR	signed char	Treated as integral value
MPI_UNSIGNED_CHAR	unsigned char	Treated as integral value
MPI_UNSIGNED_SHORT	unsigned short int	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_UNSIGNED_LONG_LONG	unsigned long long	
MPI_FLOAT	float	
MPI_DOUBLE	double	
MPI_LONG_DOUBLE	long double	
MPI_BYTE		
MPI_PACKED		

Further datatypes, see, e.g., MPI-3.1/4.0, Annex A.1

Includes also special C++ types, e.g., bool, see MPI-3.1 page 674, MPI-4.0 page 862

Python All datatype handles can be used, syntax: e.g., MPI.FLOAT

Fortran

MPI Basic Datatypes — Fortran

MPI Datatype handle	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_ LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Further datatypes, e.g., **MPI_REAL8** for **REAL*8**, see MPI-3.1/MPI-4.0, Annex A.1

Fortran

MPI Basic Datatypes — Fortran

MPI Datatype handle	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_ LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Further datatypes, e.g., **MPI_REAL8** for **REAL*8**, see MPI-3.1/MPI-4.0, Annex A.1

2345	**654**	**96574**	**-12**	**7676**

Arguments for MPI send/recv

count=5

datatype=MPI_INTEGER

Declaration of the buffers

INTEGER arr(5)

□

Fortran

MPI Basic Datatypes — Fortran

MPI Datatype handle	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_ LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Further datatypes, e.g., **MPI_REAL8** for **REAL*8**, see MPI-3.1/MPI-4.0, Annex A.1

2345	**654**	**96574**	**-12**	**7676**

Arguments for MPI send/recv

count=5

datatype=MPI_INTEGER

Declaration of the buffers

INTEGER arr(5)

For KIND-parameterized Fortran types, basic datatype handles must be generated with

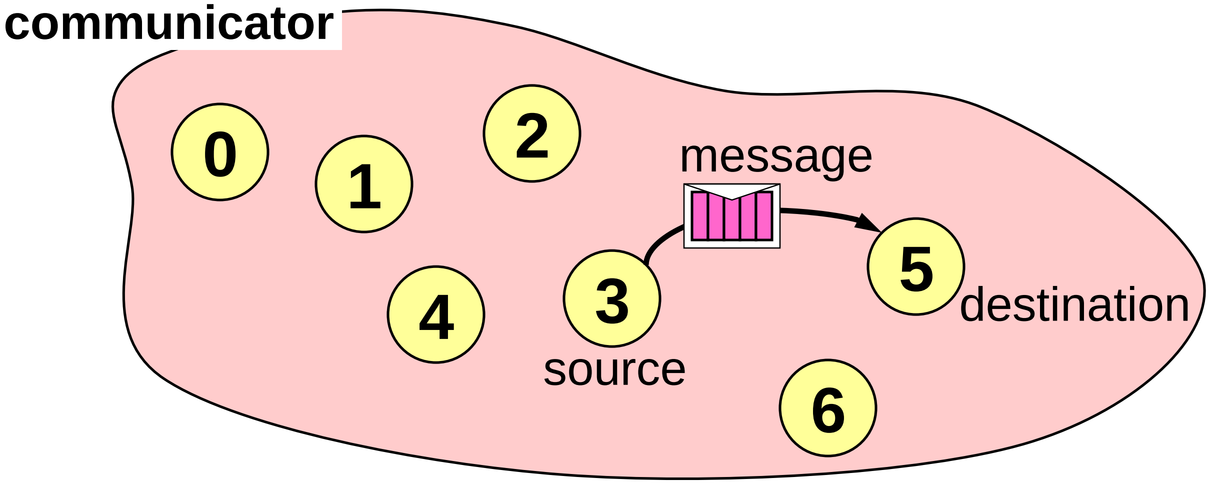
– MPI_TYPE_CREATE_F90_INTEGER

– MPI_TYPE_CREATE_F90_REAL

– MPI_TYPE_CREATE_F90_COMPLEX

Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator, e.g., MPI_COMM_WORLD.
- Processes are identified by their ranks in the communicator.



Sending a Message

C

Fortran

Python

- C/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, *ierror*)

  mpi_f08:
  ```
  TYPE(*), DIMENSION(..) :: buf
  TYPE(MPI_Datatype) :: datatype;          TYPE(MPI_Comm) :: comm
  INTEGER :: count, dest, tag;             INTEGER, OPTIONAL :: ierror
  ```
  mpi & mpif.h: <type> buf(*); INTEGER count, datatype, dest, tag, comm, ierror
- Python: comm.**Send**(buf, int dest, int tag=0)
  comm.**send**(obj, int dest, int tag=0)

- buf is the starting point of the message with count elements, each described with datatype.
- dest is the rank of the destination process within the communicator comm.
- tag is an additional nonnegative integer piggyback information, additionally transferred with the message.
- The tag can be used by the program to distinguish different types of messages.
- Python: – buf must implement the Python buffer protocol, e.g., numPy arrays
  - ◦ buf can be buf or (buf, datatype) or (buf, count, datatype)
  - ◦ with C datatypes in Python syntax, e.g., MPI.INT, MPI.FLOAT, …

  – obj is any Python object that can be serialized with the pickle method

Receiving a Message

C

- C/C++: int MPI_Recv(void **buf*, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm,
  MPI_Status **status*)

Fortran

- Fortran: MPI_RECV(*buf*,count,datatype, source, tag, comm, *status*, *ierror*)

  mpi_f08:
  TYPE(*), DIMENSION(..) :: buf
  INTEGER :: count, source, tag
  TYPE(MPI_Datatype) :: datatype; TYPE(MPI_Comm) :: comm
  TYPE(MPI_Status) :: status; INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: <type> buf(*); INTEGER count, datatype, source, tag, comm, ierror
  INTEGER status(MPI_STATUS_SIZE)

Python

- Python: comm.Recv(*buf*, int source=ANY_SOURCE, int tag=ANY_TAG, Status *status*=None)
  obj = comm.recv(buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
  Status *status*=None)

  buf is only a temporary buffer, deprecated since version 3.0.0

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.

Receiving a Message

C

- C/C++: int MPI_Recv(void **buf*, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm,
  MPI_Status **status*)

Fortran

- Fortran: MPI_RECV(*buf*,count,datatype, source, tag, comm, *status*, *ierror*)

  mpi_f08:
  ```
  TYPE(*), DIMENSION(..) :: buf
  INTEGER :: count, source, tag
  TYPE(MPI_Datatype) :: datatype;          TYPE(MPI_Comm) :: comm
  TYPE(MPI_Status) :: status;              INTEGER, OPTIONAL :: ierror
  ```

  mpi & mpif.h:
  ```
  <type> buf(*);  INTEGER count, datatype, source, tag, comm, ierror
  INTEGER status(MPI_STATUS_SIZE)
  ```

Python

- Python: comm.**R**ecv(*buf*, int source=ANY_SOURCE, int tag=ANY_TAG, Status *status*=None)
  obj = comm.**r**ecv(buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
  Status *status*=None)

  buf is only a temporary buffer, deprecated since version 3.0.0

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.
- Envelope information is returned in *status*.
- On can pass MPI_STATUS_IGNORE instead of a status argument.

Receiving a Message

C

- C/C++:
```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
                         int source, int tag, MPI_Comm comm,
                         MPI_Status *status)
```

Fortran

- Fortran:
```
MPI_RECV(buf,count,datatype, source, tag, comm, status, ierror)
```
mpi_f08:
```
TYPE(*), DIMENSION(..) :: buf
INTEGER :: count, source, tag
TYPE(MPI_Datatype) :: datatype;            TYPE(MPI_Comm) :: comm
TYPE(MPI_Status) :: status;                INTEGER, OPTIONAL :: ierror
```
mpi & mpif.h:
```
<type> buf(*);  INTEGER count, datatype, source, tag, comm, ierror
INTEGER status(MPI_STATUS_SIZE)
```

Python

- Python:
```
comm.Recv(buf, int source=ANY_SOURCE, int tag=ANY_TAG, Status status=None)
obj = comm.recv(buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
                                                  Status status=None)
```
buf is only a temporary buffer, deprecated since version 3.0.0

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.
- Envelope information is returned in *status*.
- On can pass MPI_STATUS_IGNORE instead of a status argument.
- Output arguments are printed *blue-cursive*.

□

Receiving a Message

C

- C/C++: int MPI_Recv(void *_buf_, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm,
  MPI_Status *_status_)

Fortran

- Fortran: MPI_RECV(_buf_,count,datatype, source, tag, comm, _status_, _ierror_)

  mpi_f08:
  TYPE(*), DIMENSION(..) :: buf
  INTEGER :: count, source, tag
  TYPE(MPI_Datatype) :: datatype; TYPE(MPI_Comm) :: comm
  TYPE(MPI_Status) :: status; INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: <type> buf(*); INTEGER count, datatype, source, tag, comm, ierror
  INTEGER status(MPI_STATUS_SIZE)

Python

- Python: comm.Recv(_buf_, int source=ANY_SOURCE, int tag=ANY_TAG, Status _status_=None)
  obj = comm.recv(buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
  Status _status_=None)

  buf is only a temporary buffer, deprecated since version 3.0.0

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.
- Envelope information is returned in _status_.
- On can pass MPI_STATUS_IGNORE instead of a status argument.
- Output arguments are printed _blue-cursive_.
- **Message matching rule:** receives only if comm, source, and tag match.

count, datatype is **not** part of this matching rule

Receiving a Message

C

- C/C++: int MPI_Recv(void **buf*, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm,
  MPI_Status **status*)

Fortran

- Fortran: MPI_RECV(*buf*,count,datatype, source, tag, comm, *status*, *ierror*)

  mpi_f08: TYPE(*), DIMENSION(..) :: buf
  INTEGER :: count, source, tag
  TYPE(MPI_Datatype) :: datatype; TYPE(MPI_Comm) :: comm
  TYPE(MPI_Status) :: status; INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: <type> buf(*); INTEGER count, datatype, source, tag, comm, ierror
  INTEGER status(MPI_STATUS_SIZE)

Python

- Python: comm.**R**ecv(*buf*, int source=ANY_SOURCE, int tag=ANY_TAG, Status *status*=None)
  obj = comm.**r**ecv(buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
  Status *status*=None)

  buf is only a temporary buffer, deprecated since version 3.0.0

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.
- Envelope information is returned in *status*.
- On can pass MPI_STATUS_IGNORE instead of a status argument.
- Output arguments are printed *blue-cursive*.
- **Message matching rule:** receives only if comm, source, and tag match.
- Python: **S**end requires that the matching receive is a **R**ecv / ditto for **s**end and **r**ecv

count, datatype is **not** part of this matching rule

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.

Requirements for Point-to-Point Communications

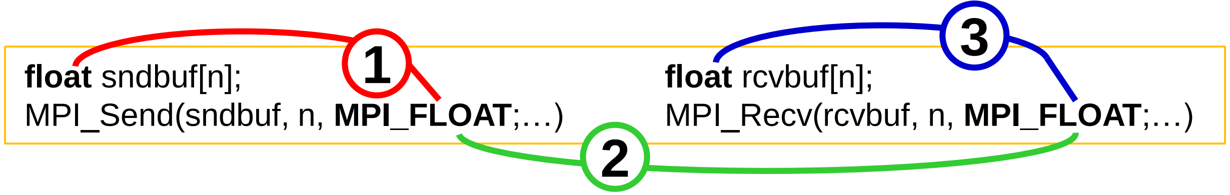
For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Type matching:



float sndbuf[n];
MPI_Send(sndbuf, n, **MPI_FLOAT**;…)

float rcvbuf[n];
MPI_Recv(rcvbuf, n, **MPI_FLOAT**;…)

① Send-buffer’s (C or Fortran) type must match with the send datatype handle

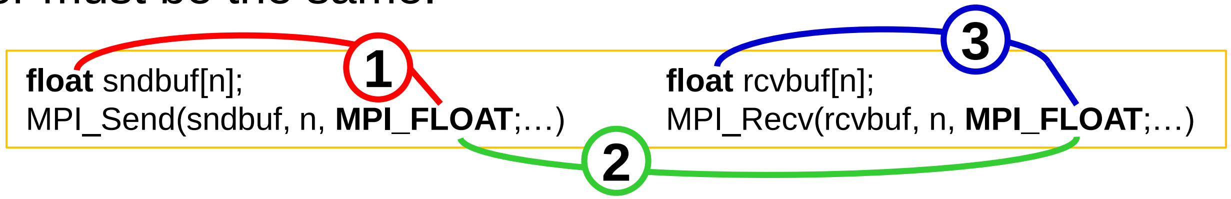
② Send datatype handle must match with the receive datatype handle

③ Receive datatype handle must match with receive-buffer’s (C or Fortran) type

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Type matching:



```
float sndbuf[n];
MPI_Send(sndbuf, n, MPI_FLOAT;…)
```

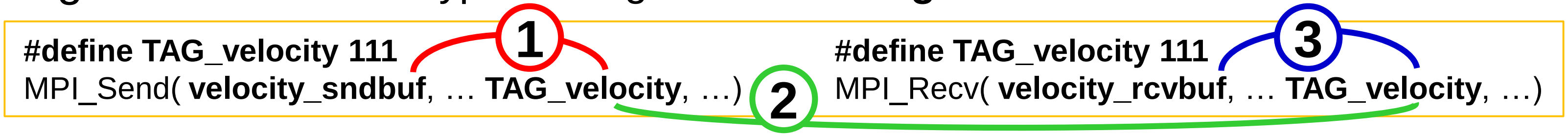
```
float rcvbuf[n];
MPI_Recv(rcvbuf, n, MPI_FLOAT;…)
```

① Send-buffer’s (C or Fortran) type must match with the send datatype handle

② Send datatype handle must match with the receive datatype handle

③ Receive datatype handle must match with receive-buffer’s (C or Fortran) type

- Tags must match → typical usage: **different tags for different data**



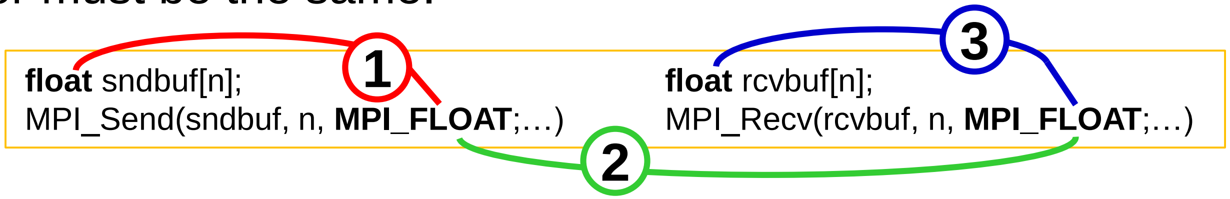
```
#define TAG_velocity 111
MPI_Send( velocity_sndbuf, … TAG_velocity, …)
```

```
#define TAG_velocity 111
MPI_Recv( velocity_rcvbuf, … TAG_velocity, …)
```

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Type matching:



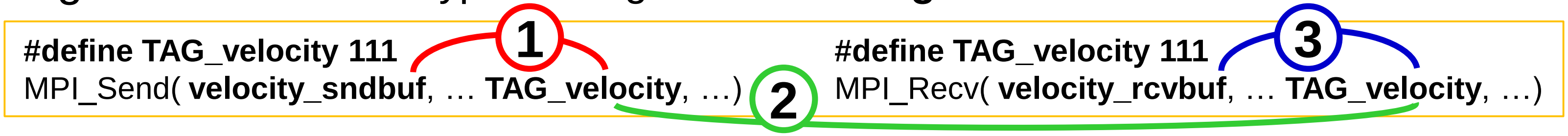
```
float sndbuf[n];                                float rcvbuf[n];
MPI_Send(sndbuf, n, MPI_FLOAT;…)                MPI_Recv(rcvbuf, n, MPI_FLOAT;…)
```

① Send-buffer’s (C or Fortran) type must match with the send datatype handle

② Send datatype handle must match with the receive datatype handle

③ Receive datatype handle must match with receive-buffer’s (C or Fortran) type

- Tags must match → typical usage: **different tags for different data**



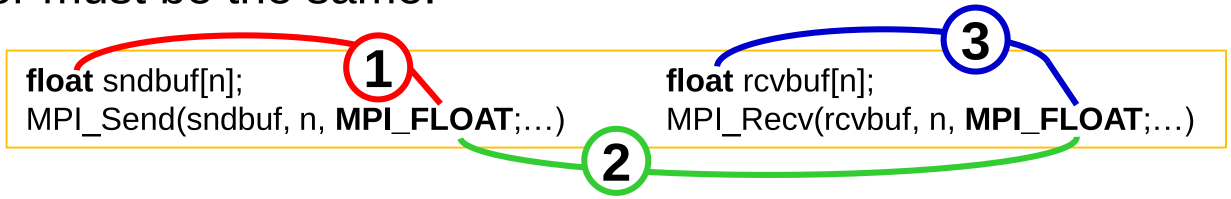
```
#define TAG_velocity 111                                #define TAG_velocity 111
MPI_Send( velocity_sndbuf, … TAG_velocity, …)           MPI_Recv( velocity_rcvbuf, … TAG_velocity, …)
```

→ The velocity message will never be received in, e.g., a temperature array

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Type matching:



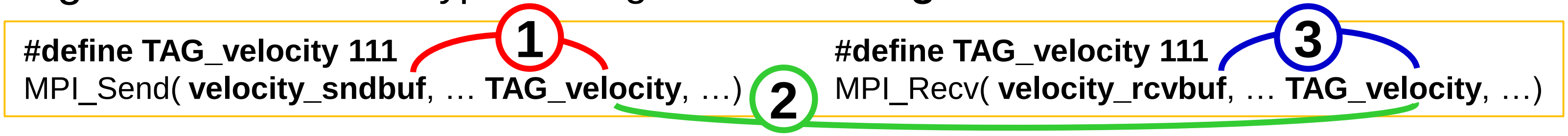
```
float sndbuf[n];                              float rcvbuf[n];
MPI_Send(sndbuf, n, MPI_FLOAT,…)              MPI_Recv(rcvbuf, n, MPI_FLOAT,…)
```

① Send-buffer’s (C or Fortran) type must match with the send datatype handle

② Send datatype handle must match with the receive datatype handle

③ Receive datatype handle must match with receive-buffer’s (C or Fortran) type

- Tags must match → typical usage: **different tags for different data**

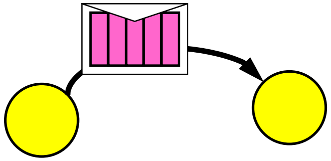


```
#define TAG_velocity 111                              #define TAG_velocity 111
MPI_Send( velocity_sndbuf, … TAG_velocity, …)         MPI_Recv( velocity_rcvbuf, … TAG_velocity, …)
```

→ The velocity message will never be received in, e.g., a temperature array

- Receiver’s buffer must be large enough.

Data conversion in heterogeneous clusters



MPI_Send(send_buf, …)

MPI_Recv(recv_buf, …)

	4 byte int	2 byte	2 byte	8 byte long long int
send_buf =	0x4321	0xBA	0xFC	0x87654321

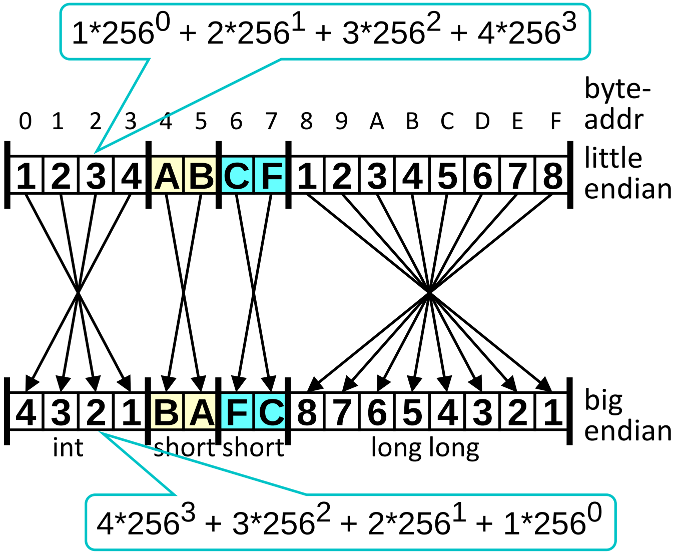
stored in a memory with little endian representation

sent to a process with big endian representation

→ **data conversion in MPI_Send or MPI_Recv**

recv_buf =	0x4321	0xBA	0xFC	0x87654321

same data values, but different internal representation



$1{*}256^0 + 2{*}256^1 + 3{*}256^2 + 4{*}256^3$

0 1 2 3 4 5 6 7 8 9 A B C D E F
byte-addr
1 2 3 4 A B C F 1 2 3 4 5 6 7 8
little endian
4 3 2 1 B A F C 8 7 6 5 4 3 2 1
big endian
int
short
short
long long

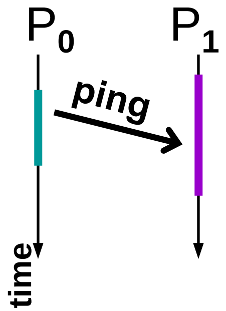
$4{*}256^3 + 3{*}256^2 + 2{*}256^1 + 1{*}256^0$

Note, most clusters are homogeneous
→ conversion is not needed
→ no additional communication overhead for this

Exercise 1 — One Ping

In MPI/tasks/…

- Use: C C/Ch3/ping-skel.c or Fortran F_30/Ch3/ping-skel_30.f90 or Python PY/Ch3/ping-skel.py (hint: use send & recv)
- Write a program according to the time-line diagram:
  - Process 0 sends a message to process 1 (ping)
- We prepare a benchmark program → don't care on buffer contents
  - Just send 1 float (in C) / REAL (in Fortran) / [None] (in Python)

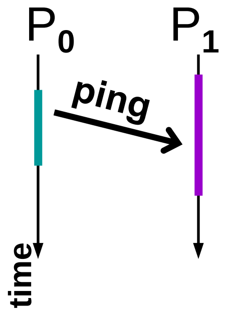


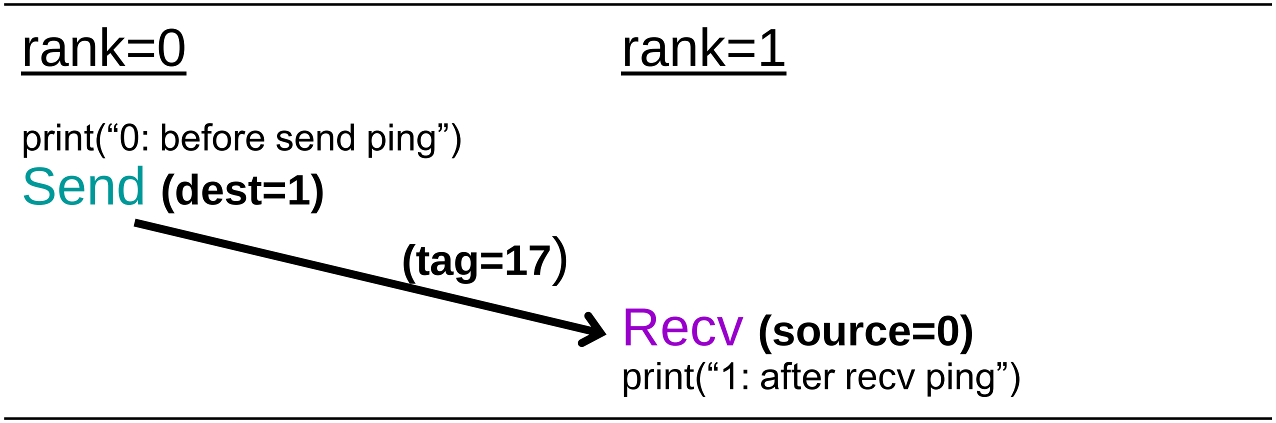
Exercises 1+2

Exercise 1 — One Ping

In MPI/tasks/…

- Use: **C** C/Ch3/ping-skel.c or **Fortran** F_30/Ch3/ping-skel_30.f90 or **Python** PY/Ch3/ping-skel.py (hint: use **s**end & **r**ecv)
- Write a program according to the time-line diagram:
  - Process 0 sends a message to process 1 (ping)
- We prepare a benchmark program → don’t care on buffer contents
  - Just send 1 float (in C) / REAL (in Fortran) / [None] (in Python)





rank=0

print(“0: before send ping”)
Send **(dest=1)**

(tag=17)

rank=1

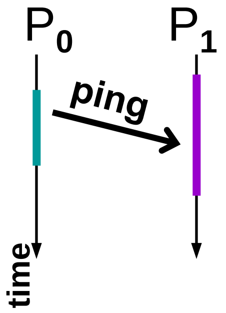
Recv **(source=0)**
print(“1: after recv ping”)

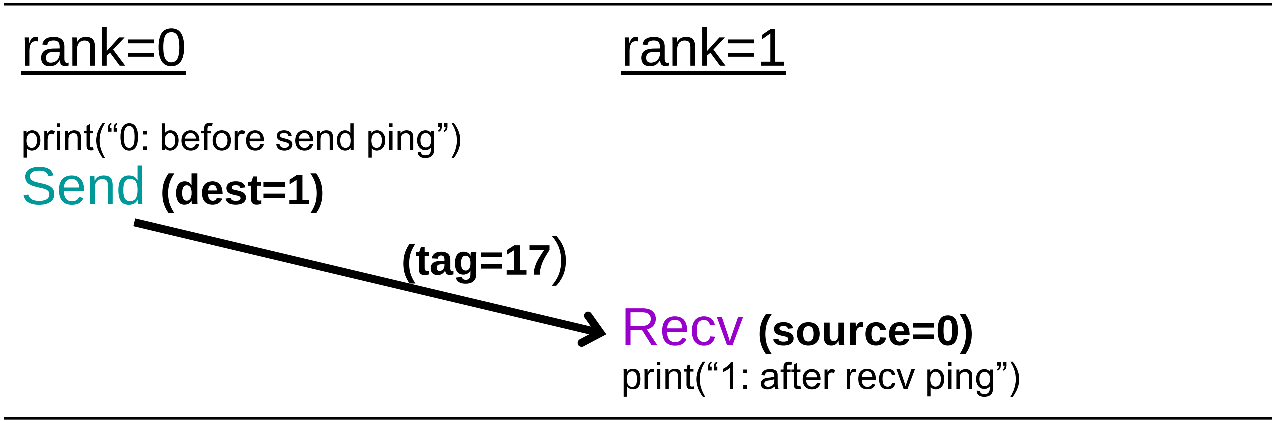
Exercises 1+2

Exercise 1 — One Ping

In MPI/tasks/…

- Use: C C/Ch3/ping-skel.c or Fortran F_30/Ch3/ping-skel_30.f90 or Python PY/Ch3/ping-skel.py (hint: use send & recv)
- Write a program according to the time-line diagram:
  - Process 0 sends a message to process 1 (ping)
- We prepare a benchmark program → don't care on buffer contents
  - Just send 1 float (in C) / REAL (in Fortran) / [None] (in Python)





rank=0

print("0: before send ping")
Send (dest=1)

(tag=17)

rank=1

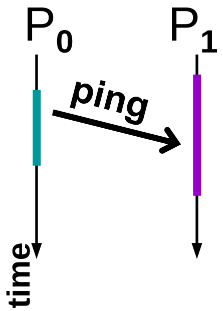
Recv (source=0)
print("1: after recv ping")

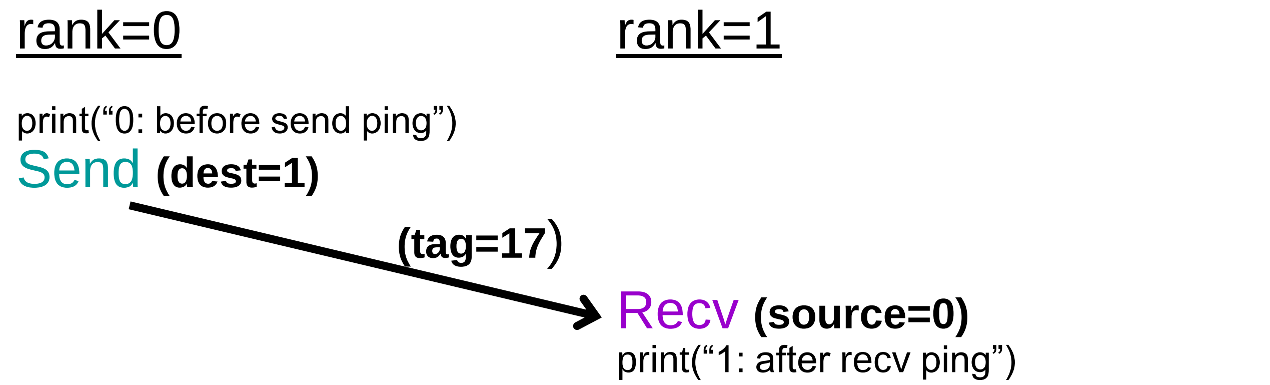
```
if (my_rank==0)          /* i.e., emulated multiple program */
      MPI_Send( ... dest=1 ...)
else
      MPI_Recv( ... source=0 ...)
fi
```

Exercise 1 — One Ping

In MPI/tasks/…

- Use: C C/Ch3/ping-skel.c or Fortran F_30/Ch3/ping-skel_30.f90 or Python PY/Ch3/ping-skel.py (hint: use send & recv)
- Write a program according to the time-line diagram:
  - – Process 0 sends a message to process 1 (ping)
- We prepare a benchmark program → don't care on buffer contents
  - – Just send 1 float (in C) / REAL (in Fortran) / [None] (in Python)





rank=0

rank=1

print("0: before send ping")

Send (dest=1)

(tag=17)

Recv (source=0)

print("1: after recv ping")

```
if (my_rank==0)             /* i.e., emulated multiple program */
    MPI_Send( ... dest=1 ...)
else
    MPI_Recv( ... source=0 ...)
fi
```

Start with only 2 processes:
mpirun -np 2 …

Exercise 2 — One ping pong

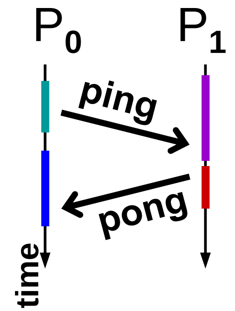
- Before starting this exercise 2, you should have **compared your result of exercise 1** with **ping.c** / **_30.f90** / **.py** in the solution sub-directory

---

Exercise 2:

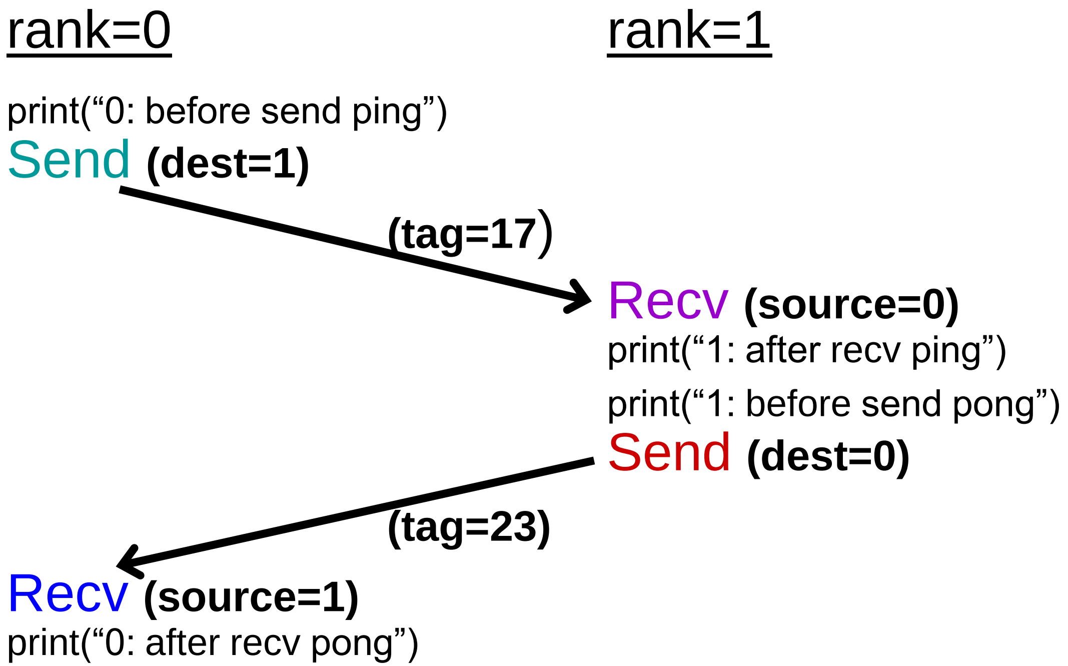
- Use: C C/Ch3/pingpong-skel.c or Fortran F_30/Ch3/pingpong-skel_30.f90 or Python PY/Ch3/pingpong-skel.py (hint: use send & recv)

- Write a program according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message,
    process 1 sends a message back to process 0 (pong)



- For details, see next slide

Exercise 2 — One ping pong



rank=0

print(“0: before send ping”)

Send (dest=1)

(tag=17)

Recv (source=1)

print(“0: after recv pong”)

rank=1

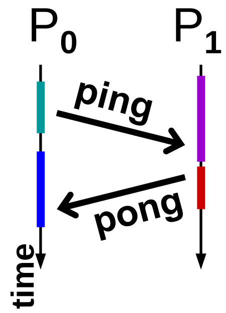
Recv (source=0)

print(“1: after recv ping”)

print(“1: before send pong”)

Send (dest=0)

(tag=23)

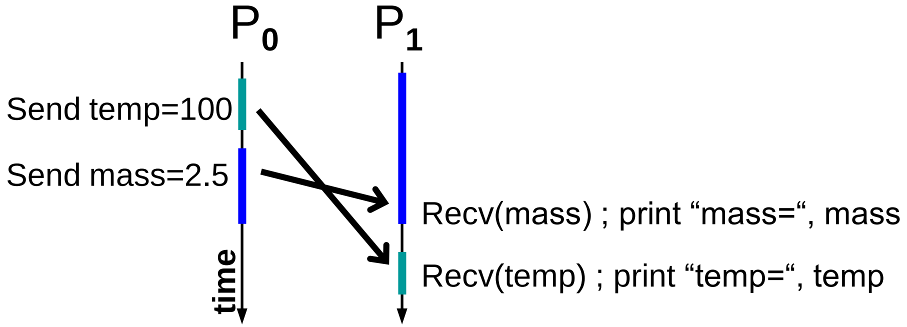


```
if (my_rank==0)              /* i.e., emulated multiple program */
    MPI_Send( ... dest=1 ...)
    MPI_Recv( ... source=1 ...)
else
    MPI_Recv( ... source=0 ...)
    MPI_Send( ... dest=0 ...)
fi
```

Advanced Exercise 2b — Overtaking messages

Advanced Exercises 2b

- Use: C C/Ch3/overtake-skel.c or Fortran F_30/Ch3/overtake-skel_30.f90 or Python PY/Ch3/overtake-skel.py (hint: use send & recv)
- Write a program according to the time-line diagram:



P_0
P_1
Send temp=100
Send mass=2.5
time
Recv(mass) ; print "mass=", mass
Recv(temp) ; print "temp=", temp

- Use float in C / REAL in Fortran for temp and mass
- 1st test: use same tags for both messages → expected: wrong result
- 2nd test: use different tags → correct result

Remarks:

- The complete rules for overtaking messages will come at the end of the chapter.
- Solutions: C / F_30 / PY/Ch3/solutions/overtake.c / _30.f90 / .py
- Later we'll learn that this program may also cause a deadlock, because MPI_Send may synchronize; see additional solutions overtake-arr.c / -arr_30.f90 / -arr.py

Wildcarding

Use case:
A manager or I/O process waits for
(and receives) results
from some worker processes

- Receiver can wildcard.
- To receive from any source — source = MPI_ANY_SOURCE
- To receive from any tag — tag = MPI_ANY_TAG
- Actual source and tag are returned in the receiver's *status* parameter.

Wildcarding

Use case:
A manager or I/O process waits for (and receives) results from some worker processes

- Receiver can wildcard.
- To receive from any source — source = MPI_ANY_SOURCE
- To receive from any tag — tag = MPI_ANY_TAG
- Actual source and tag are returned in the receiver's *status* parameter.

- With info assertions **New in MPI-4.0**
  - "mpi_assert_no_any_source" = "true" and/or
  - "mpi_assert_no_any_tag" = "true"

  stored on the communicator usind MPI_Comm_set_info(),
  - an MPI application can tell the MPI library that it will never use MPI_ANY_SOURCE and/or MPI_ANY_TAG on this communicator

  → may enable lower latencies.
- Other assertions:
  - "mpi_assert_exact_length" = "true" → receive buffer must have exact length
  - "mpi_assert_allow_overtaking" = "true" → message order need not to be preserved

Communication Envelope

- Envelope information is returned from MPI_RECV in *status*.

C

- C/C++: MPI_Status status;
  status.MPI_SOURCE
  status.MPI_TAG
  status.MPI_ERROR *)

Fortran

- Fortran:
  mpi_f08: TYPE(MPI_Status) :: status
  status%MPI_SOURCE
  status%MPI_TAG
  status%MPI_ERROR *)
- mpi & mpif.h: INTEGER status(MPI_STATUS_SIZE)
  status(MPI_SOURCE)
  status(MPI_TAG)
  status(MPI_ERROR) *)

special indexes

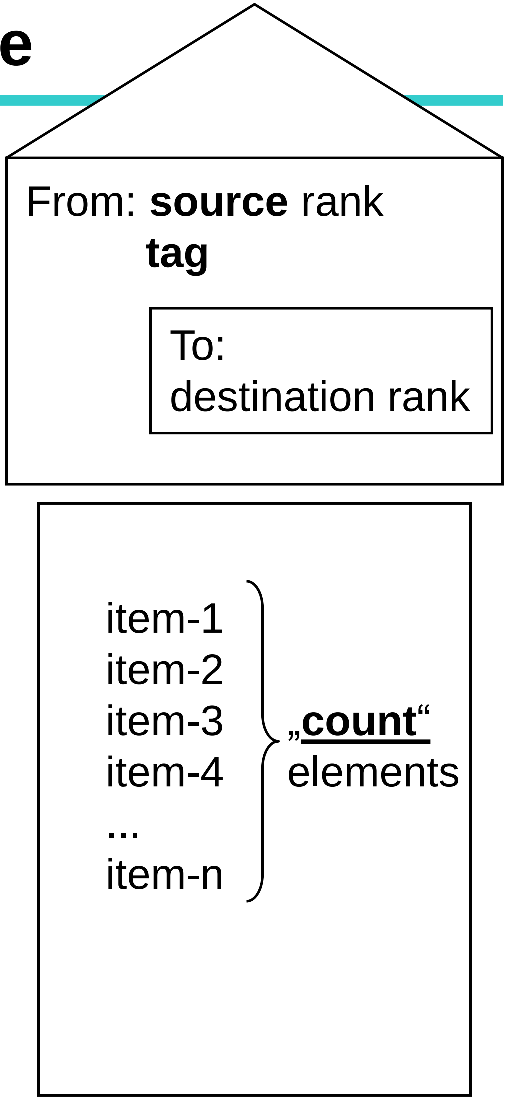
Python

- Python: status.Get_source()
  status.Get_tag(), …

See also MPI-2.2 page 32, lines 14-23

- count via MPI_GET_COUNT()

*) See slide on MPI_Waitall, …


From: **source** rank
tag

To:
destination rank

item-1
item-2
item-3
item-4
...
item-n

„**count**“
elements

Receive Message Count

C

- C/C++: int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

Fortran

- Fortran: MPI_GET_COUNT(status, datatype, *count*, *ierror*)

```
mpi_f08:   TYPE(MPI_Status)        :: status
           TYPE(MPI_Datatype)      :: datatype
           INTEGER                 :: count
           INTEGER, OPTIONAL       :: ierror
```

```
mpi & mpif.h:  INTEGER status(MPI_STATUS_SIZE), datatype, count, ierror
```

Python

- Python: *count* = status.Get_count(Datatype datatype=BYTE)

Caution:

```
buf = np.zeros((100,), dtype=np.double)
comm.Send((buf, 5, MPI.DOUBLE), ….)
comm.Recv((buf, 100, MPI.DOUBLE),…., status)
count = status.Get_count(MPI.DOUBLE) # → 5
count = status.Get_count() # → 40
```

Communication Modes

- Send communication modes:
  - synchronous send → MPI_**S**SEND
  - buffered [asynchronous] send → MPI_**B**SEND
  - standard send → MPI_**SEND**
  - Ready send → MPI_**R**SEND

- for different use cases
- with different performance

- Receiving all modes → MPI_**RECV**

Communication Modes — Definitions

Send mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	

Communication Modes — Definitions

Send mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	local call, i.e., always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH For additional risks, see *progress* slides in course chapter 18 *Best practice.* **New in MPI-4.1** Automatic buffering and buffering methods on communicator and session level.

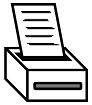
Communication Modes — Definitions

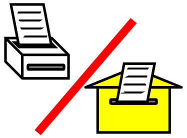
Send mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	local call, i.e., always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH For additional risks, see *progress* slides in course chapter 18 *Best practice.* **New in MPI-4.1** Automatic buffering and buffering methods on communicator and session level.
Standard send **MPI_SEND**	Either synchronous or buffered	uses an internal buffer

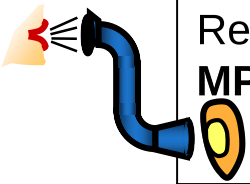
Communication Modes — Definitions

Send mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	local call, i.e., always completes (unless an error occurs), irrespective of receiver **New in MPI-4.1**	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH For additional risks, see *progress* slides in course chapter 18 *Best practice.* Automatic buffering and buffering methods on communicator and session level.
Standard send **MPI_SEND**	Either synchronous or buffered	uses an internal buffer
Ready send **MPI_RSEND**	May be started **only** if the matching receive is already posted!	highly dangerous!

Communication Modes — Definitions







Send mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	local call, i.e., always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH For additional risks, see *progress* slides in course chapter 18 *Best practice.* **New in MPI-4.1** Automatic buffering and buffering methods on communicator and session level.
Standard send **MPI_SEND**	Either synchronous or buffered	uses an internal buffer
Ready send **MPI_RSEND**	May be started **only** if the matching receive is already posted!	highly dangerous!
Receive **MPI_RECV**	Completes when a message has arrived	same routine for all communication modes

Rules for the communication modes

- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode

  → all risks of synchronous send

for any message size

Rules for the communication modes

- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode

  → all risks of synchronous send

for any message size

- Synchronous send (**MPI_SSEND**)
  - risk of deadlock
  - risk of serialization
  - risk of waiting —> idle time
  - high latency / best bandwidth

minimal transfer time
only for **long** messages

□

Rules for the communication modes

- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode

  → all risks of synchronous send

for any message size

- Synchronous send (**MPI_SSEND**)
  - risk of deadlock
  - risk of serialization
  - risk of waiting —> idle time
  - high latency / best bandwidth

minimal transfer time **only** for **long** messages

- Buffered send (**MPI_BSEND**)
  - low latency / bad bandwidth

minimal transfer time **only** for **short** messages

Rules for the communication modes

- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode

  → all risks of synchronous send

for any message size

- Synchronous send (**MPI_SSEND**)
  - risk of deadlock
  - risk of serialization
  - risk of waiting —> idle time
  - high latency / best bandwidth

minimal transfer time **only** for **long** messages

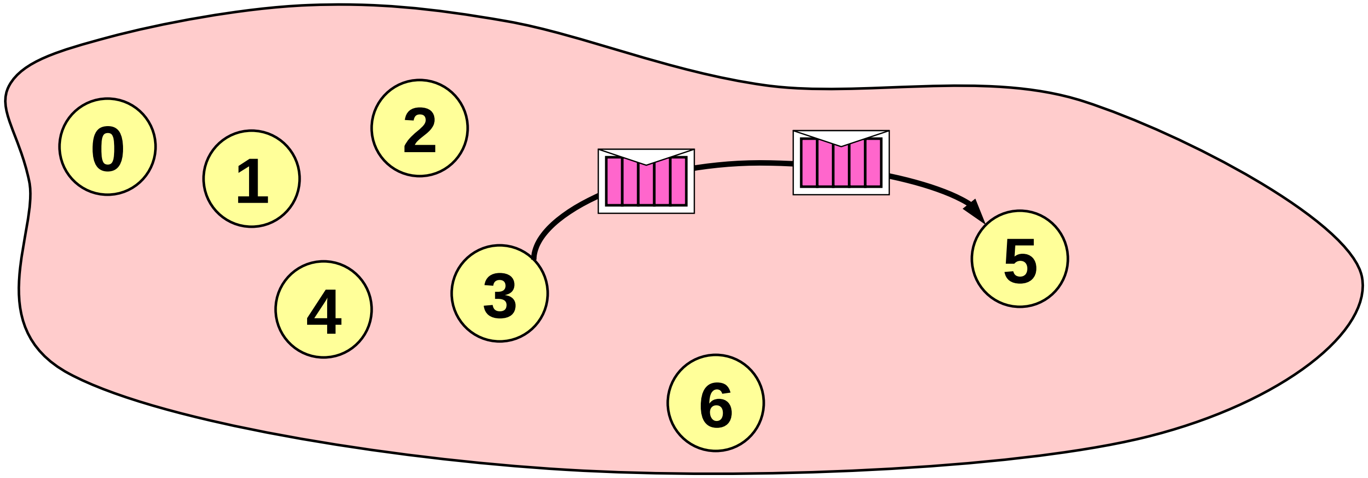
- Buffered send (**MPI_BSEND**)
  - low latency / bad bandwidth

minimal transfer time **only** for **short** messages

- Ready send (**MPI_RSEND**)
  - use **never**, except you have a *200% guarantee* that Recv is already called in the current version and all future versions of your code,
  - may be the fastest,
  - for a use case, see later → Chapter 4 (nonblocking) → Quiz E

Message Order Preservation

- Rule for messages on the same connection,
  i.e., same communicator, source, and destination rank:
- **Messages do not overtake each other.**
- This is true even for non-synchronous sends.



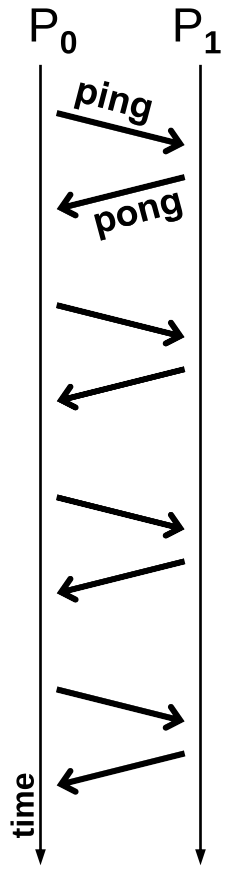
- If both receives match both messages, then the order is preserved.

Exercises 3-5

Exercise 3 — Ping pong benchmark

Use: C C/Ch3/pingpong-bench-skel.c or Fortran F_30/Ch3/pingpong-bench-skel_30.f90
Python PY/Ch3/pingpong-bench-skel.py

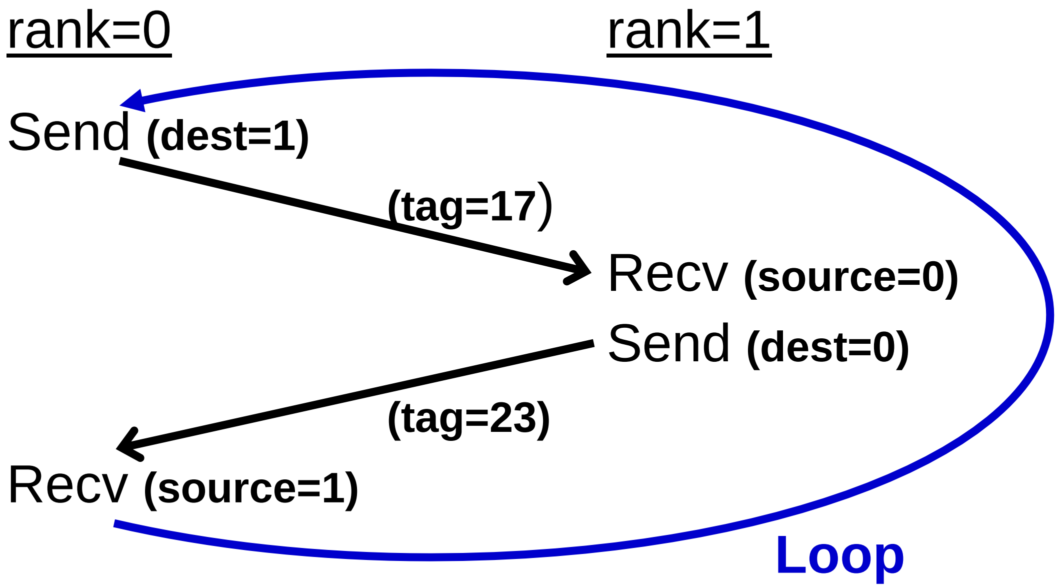
- Write a program according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message,
    process 1 sends a message back to process 0 (pong)
- Repeat this ping-pong with a loop of length 50
- Add timing calls before and after the loop:
- C/C++: *double MPI_Wtime*(void); [1)]
- Fortran: *DOUBLE PRECISION FUNCTION MPI_WTIME*()
- Python: *time* = MPI.Wtime()
- MPI_WTIME returns a wall-clock time in seconds.
- Only at process 0,
  - print out the transfer time of **one** message
  - in µs, i.e., delta_time / (2*50) * 1e6
- See also next slide



Removed in MPI-4.1

~~[1)] One of the rare routines that can be implemented as macros in C, see MPI-3.1 / MPI-4.0, Sect.2.6.4, page 20 / 26~~

Exercise 3 — Ping pong benchmark



rank=0

rank=1

Send **(dest=1)**

(tag=17)

Recv **(source=0)**

Send **(dest=0)**

(tag=23)

Recv **(source=1)**

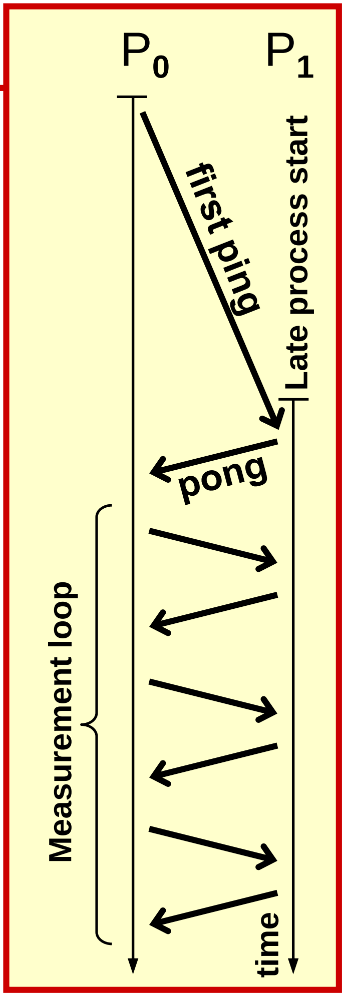
Loop

```
if (my_rank==0)              /* i.e., emulated multiple program */
     MPI_Send( ... dest=1 ...)
     MPI_Recv( ... source=1 ...)
else
     MPI_Recv( ... source=0 ...)
     MPI_Send( ... dest=0 ...)
fi
```

Exercises 4+5 (advanced): Ping pong latency and bandwidth

Exercise 4

- Exclude startup time problems from measurements:
  - – Execute a first ping-pong outside of the measurement loop



Exercise 5

- latency = transfer time for short messages
- bandwidth = message size (in bytes) / transfer time
- Print out message transfer time and bandwidth
  - – for following send modes:
    - **for standard send (MPI_Send)**
    - **for synchronous send (MPI_Ssend)**
  - – for following message sizes:
    - **8 bytes (e.g., one double or double precision value)**
    - **512 B (= 8*64 bytes)**
    - **32 kB (= 8*64**2 bytes)**
    - **2 MB (= 8*64**3 bytes)**

C

unlimit or
ulimit -s 200000
once before calling **mpirun**

Quiz on Chapter 3 – Point-to-point communication

A. How many different MPI point-to-point send modes (=blocking APIs) exist?

B. Which one requires that you first use MPI_Buffer_attach?

C. Which one is recommended for smallest latency and highest bandwidth both together?

D. If your buffer is an array `buf` with 5 double precision values that you want to send? How do you describe your message in the call to MPI_Send
   - in C (or Python)?
   - in Fortran?

E. When calling MPI_Recv to receive this message which count values would be correct?

F. When I use one of the MPI send routines, how many messages do I send?

G. Which is the predefined communicator that can be used to exchange a message from process rank 3 to process rank 5?

H. If you send two messages msg1 and msg2 from rank 3 to rank 5, is it possible that the second one can overtake, i.e., be received before the first one?

I. Do you remember the major risks of synchronous send?

J. Has standard send the same risks?

K. What is the major use case for tags?