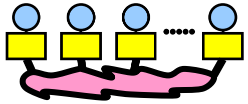
Messages and Point-to-Point Communication

Sub Routine

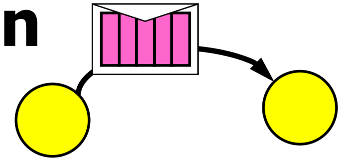
1. MPI Overview



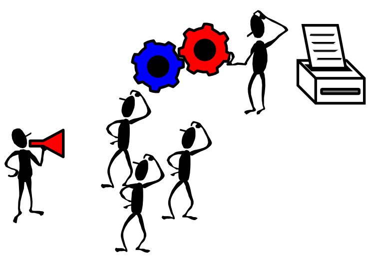
2. Process model and language bindings

MPI_Init()
MPI_Comm_rank()

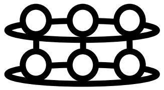
3. **Messages and point-to-point communication**
   - **the MPI processes can communicate**



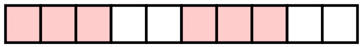
4. Non-blocking communication



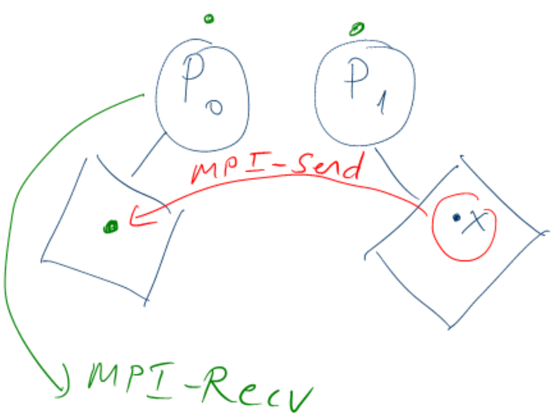
5. Collective communication
6. Virtual topologies



7. Derived datatypes



8. Case study



P2P Comm

ICHEC
Irish Centre for High-End Computing

Messages

- A message contains a number of elements of some particular datatype.
  int, float, char, struct, ---
- MPI datatypes:
  MPI_INT, MPI_FLOAT, MPI_CHAR, ...
  - Basic datatype.
  - Derived datatypes
- C types are different ~~from Fortran types~~.
- Datatype handles are used to describe the type of the data in the memory.

float d[7];

float *d = (float *) malloc(7 * sizeof(float))

Example: message with 5 integers

int *array =

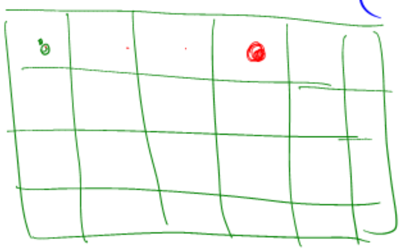
2345	654	96574	-12	7676

array ⟺ &array[0]   &array[1]   &array[4]

C

Row oriented (based)

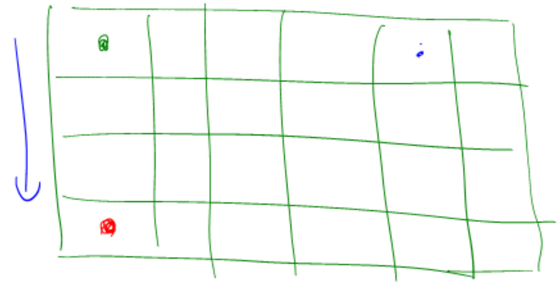
a



4 x 6

F

a



column oriented (based)

a[0]

a[3]

a[0]

a[3]

i k j $\longrightarrow$???

MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	(Mixed)

MPI Datatype	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	

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

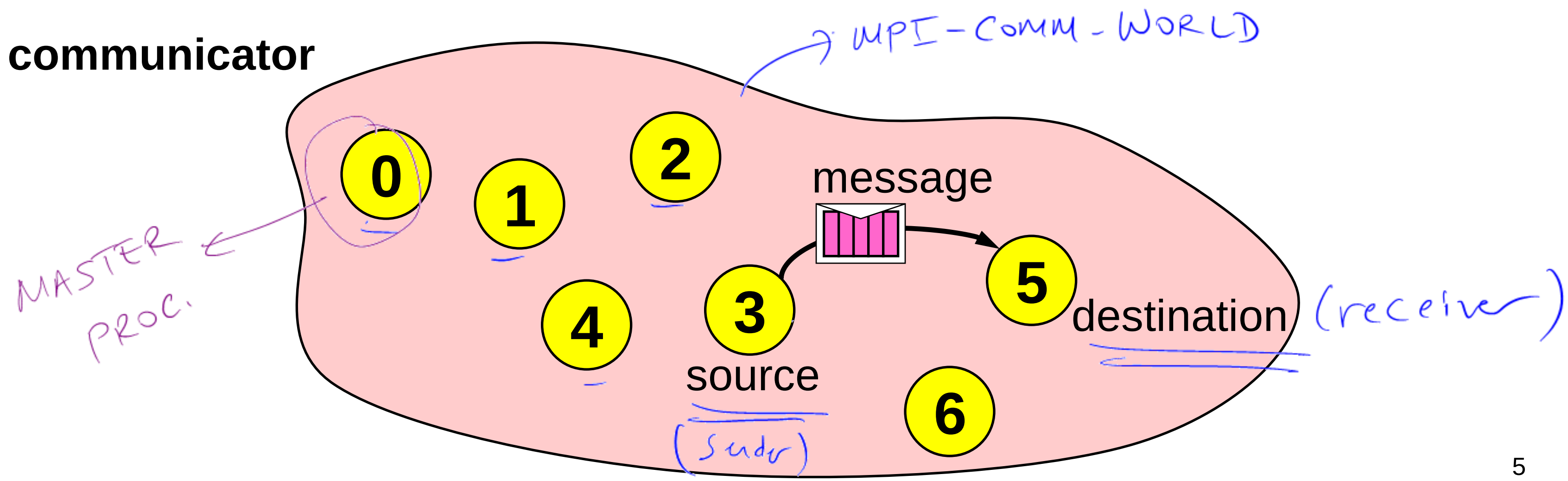
count=5
datatype=MPI_INTEGER

INTEGER arr(5)

P2P

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.



how many

(array) address data to be sent.

Sending a Message

- C: int MPI_Send(void *buf, int count, MPI_Datatype datatype,
  int dest, int tag, MPI_Comm comm)

where to　　77年 666　　→ MPI-COMM-WORLD

- Fortran: MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, *IERROR*)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

- 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.

Gelen veri hangi adrese alınacak

Receiving a Message

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

- Fortran: MPI_RECV(*BUF*, COUNT, DATATYPE, SOURCE, TAG,
  COMM, *STATUS*, *IERROR*)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM
  INTEGER STATUS(MPI_STATUS_SIZE), IERROR

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank <u>source</u> in <u>comm</u>.
- Envelope information is returned in <u>*status*</u>.
- Output arguments are printed *blue-cursive*.
- Only messages with matching <u>tag</u> are received.

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.
- Tags must match.
- Message datatypes must match. ?!
- Receiver's buffer must be large enough.

Wildcards

- 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.

P_0 → P_1

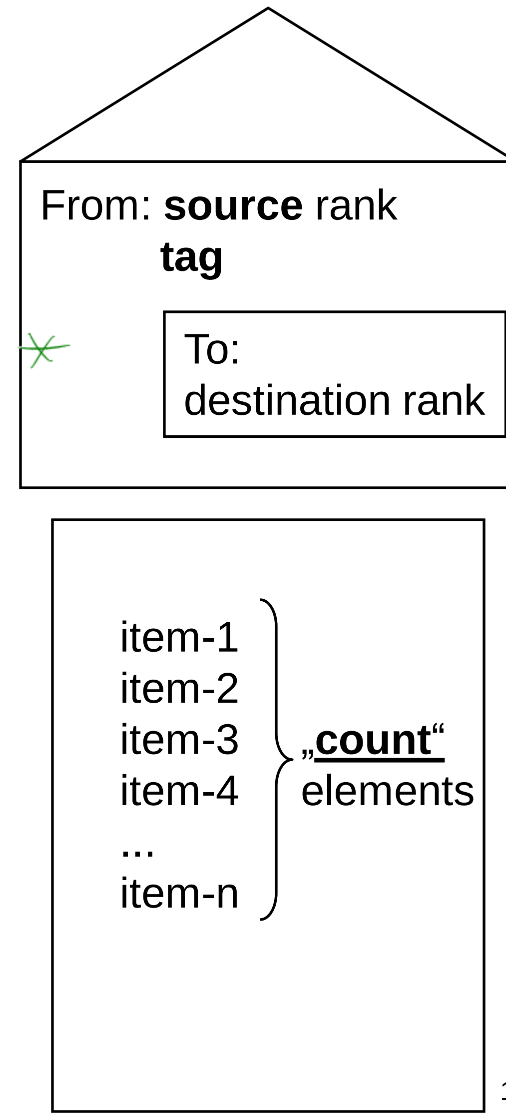
Communication Envelope → Zarf

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

MPI_Status

- C: status.MPI_SOURCE →
  status.MPI_TAG →
  <u>count</u> via MPI_Get_count()

- Fortran: status(MPI_SOURCE)
  status(MPI_TAG)
  <u>count</u> via MPI_GET_COUNT()



Receive Message Count

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

  Output parameter.

- Fortran: MPI_GET_COUNT(STATUS, DATATYPE, *COUNT*, *IERROR*)

  INTEGER STATUS(MPI_STATUS_SIZE)
  INTEGER DATATYPE, COUNT, IERROR

COMMUNICATION IN MPI

P2P

MPI-Send
MPI-Recv

Blocking

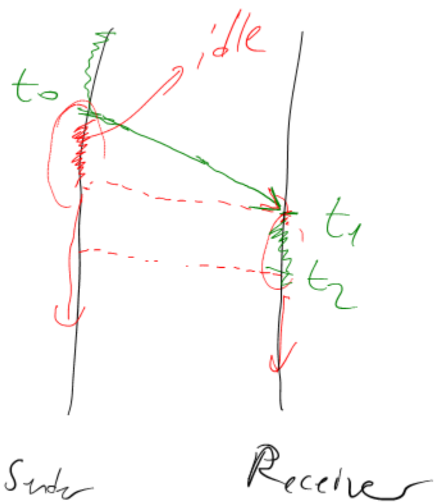
Non-blocking

MPI-Isend
Immediate
MPI-IRecv

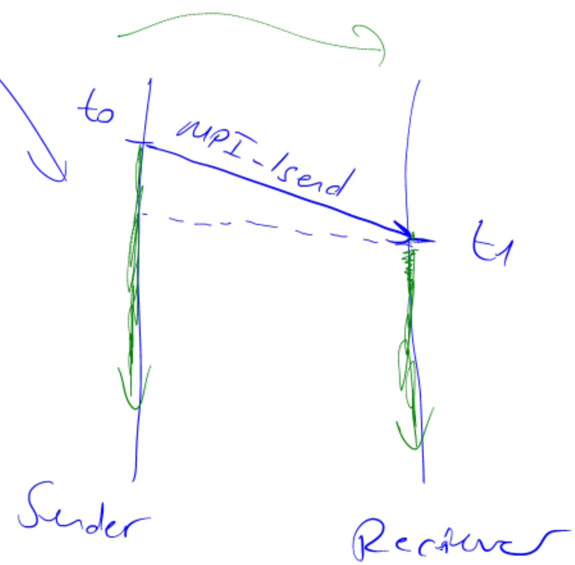
Collective

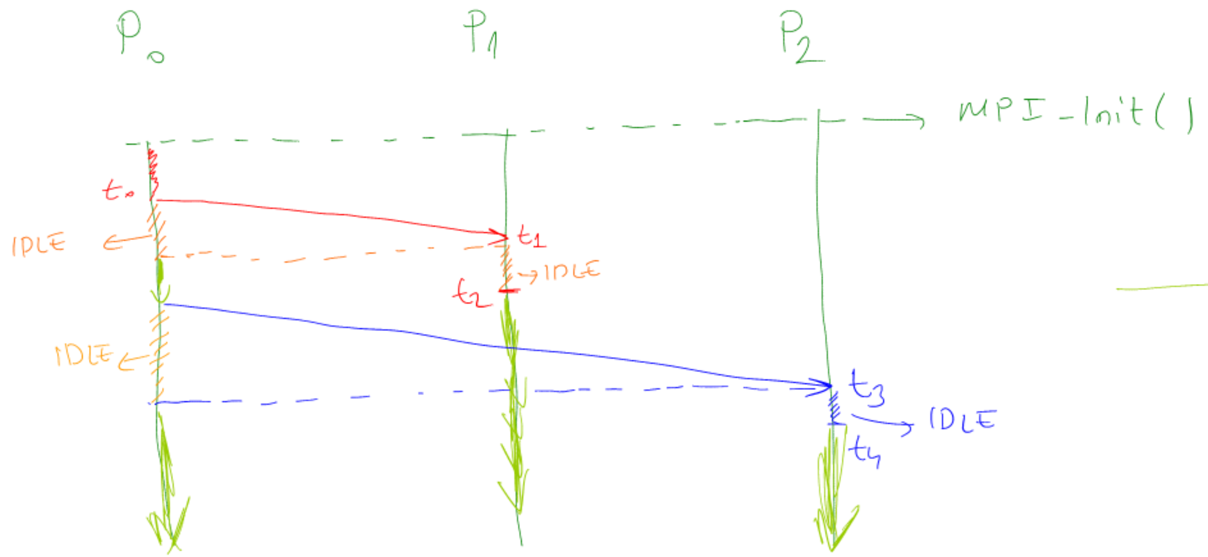
Next week !!!

Blocking : Safer
Non-blocking : Faster

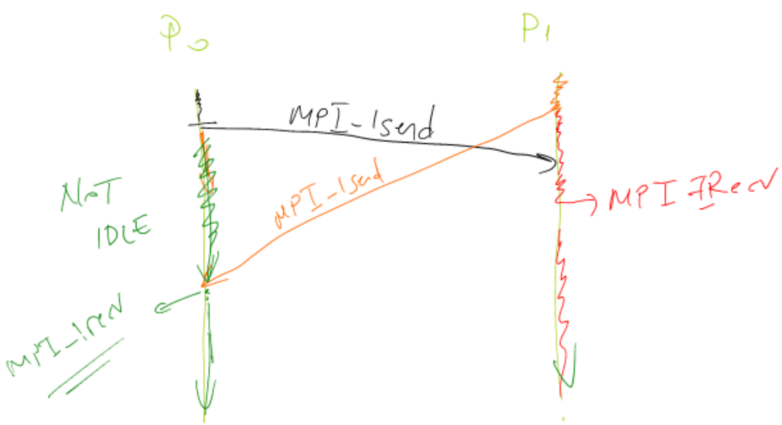


time





→ Blocking



→ Non-blocking

(Avoiding deadlock)

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

- Receiving all modes → MPI_**RECV**

Communication Modes — Definitions

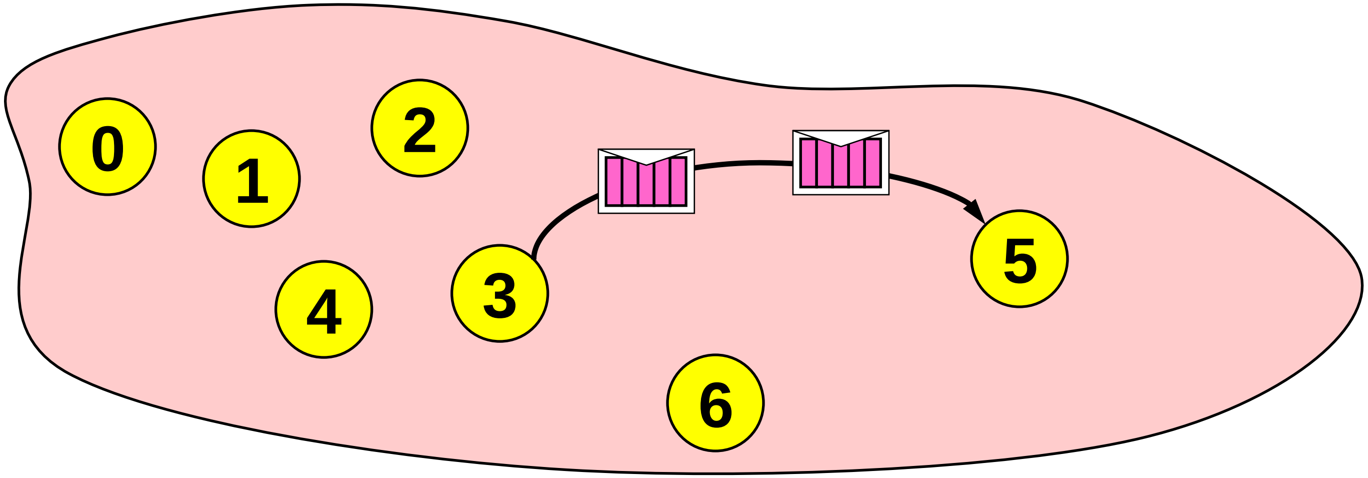
Sender modes	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Synchronous **MPI_SEND**	Standard send. Either uses an internal buffer or buffered	
Ready send **MPI_RSEND**	May be started **only** if the matching receive is already posted!	highly dangerous!
Receive **MPI_RECV**	Completes when a the message (data) has arrived	

Rules for the communication modes

- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode
  - —> risks with synchronous send
- Synchronous send (**MPI_SSEND**)
  - risk of deadlock
  - risk of serialization
  - risk of waiting —> idle time
  - high latency / best bandwidth
- Buffered send (**MPI_BSEND**)
  - low latency / bad bandwidth
- 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

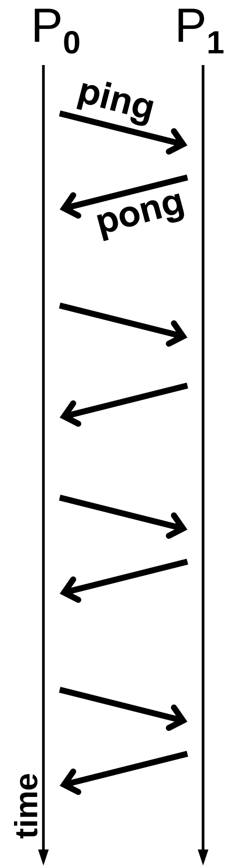
Message Order Preservation

- 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.

Exercise — Ping pong



- 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: *double MPI_Wtime*(void);
- Fortran: *DOUBLE PRECISION FUNCTION MPI_WTIME*()
- MPI_WTIME returns a wall-clock time in seconds.
- At process 0, print out the transfer time of **one** message
  - in seconds
  - in µs.

Ping - Pong Example : Wimbledon Final Federer vs Nadal

(0.88) Federer

(0.91) Nadal

int ball = 0;

1 ball++;

2 ball++;

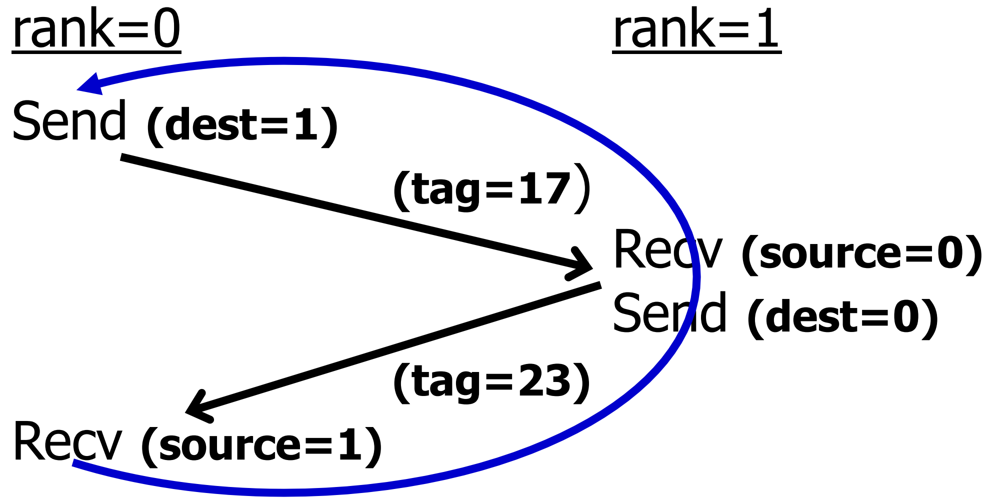
ball++;

ball % 2 == 1 $\Rightarrow$ Federer wins

else

$\Rightarrow$ Nadal wins

Exercise — Ping pong




```
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
```

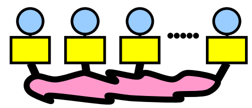
Advanced Exercise - Measure latency and bandwidth

- latency = transfer time for zero length 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)

Chap.4 Non-Blocking Communication

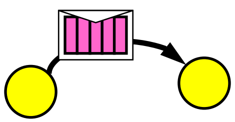
1. MPI Overview



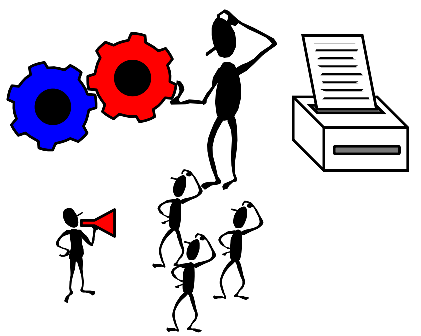
2. Process model and language bindings

MPI_Init()
MPI_Comm_rank()

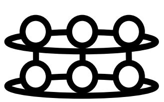
3. Messages and point-to-point communication



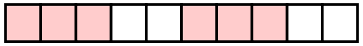
4. **Non-blocking communication**
   – **to avoid idle time and deadlocks**



5. Collective communication
6. Virtual topologies



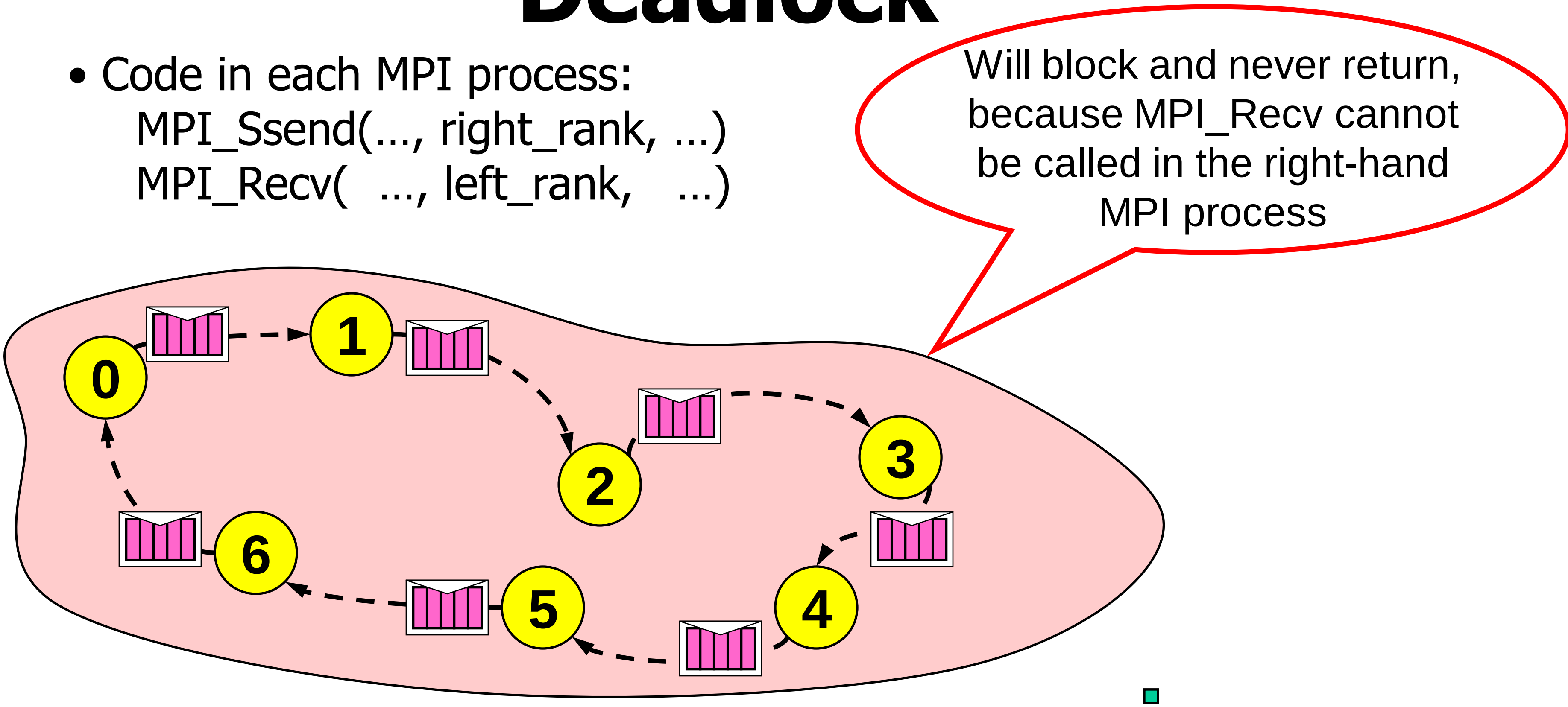
7. Derived datatypes



8. Case study

Deadlock

- Code in each MPI process:
  MPI_Ssend(…, right_rank, …)
  MPI_Recv(…, left_rank, …)



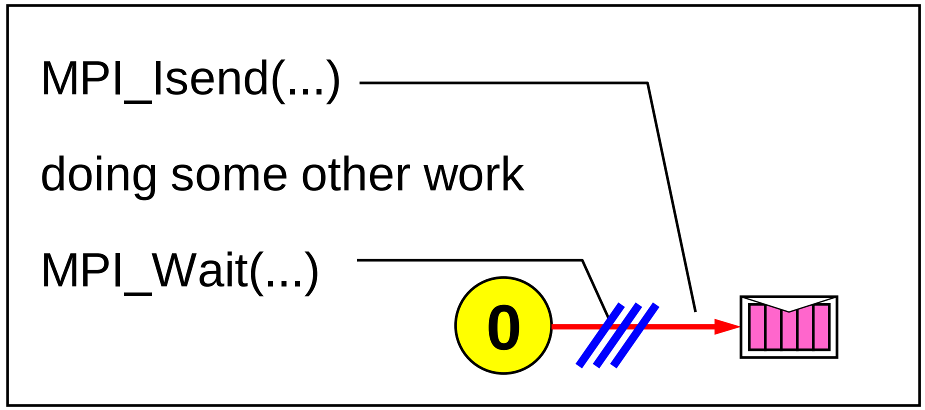
- Same problem with standard send mode (MPI_Send), if MPI implementation chooses synchronous protocol

Non-Blocking Communications

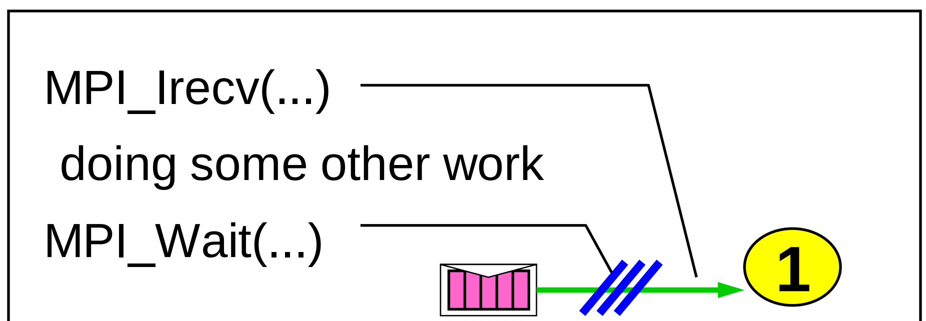
- Separate communication into three phases:
- Initiate non-blocking communication
  - returns **I**mmediately
  - routine name starting with MPI_**I**…
- Do some work
  - "latency hiding"
- Wait for non-blocking communication to complete

Non-Blocking Examples

- Non-blocking **send**



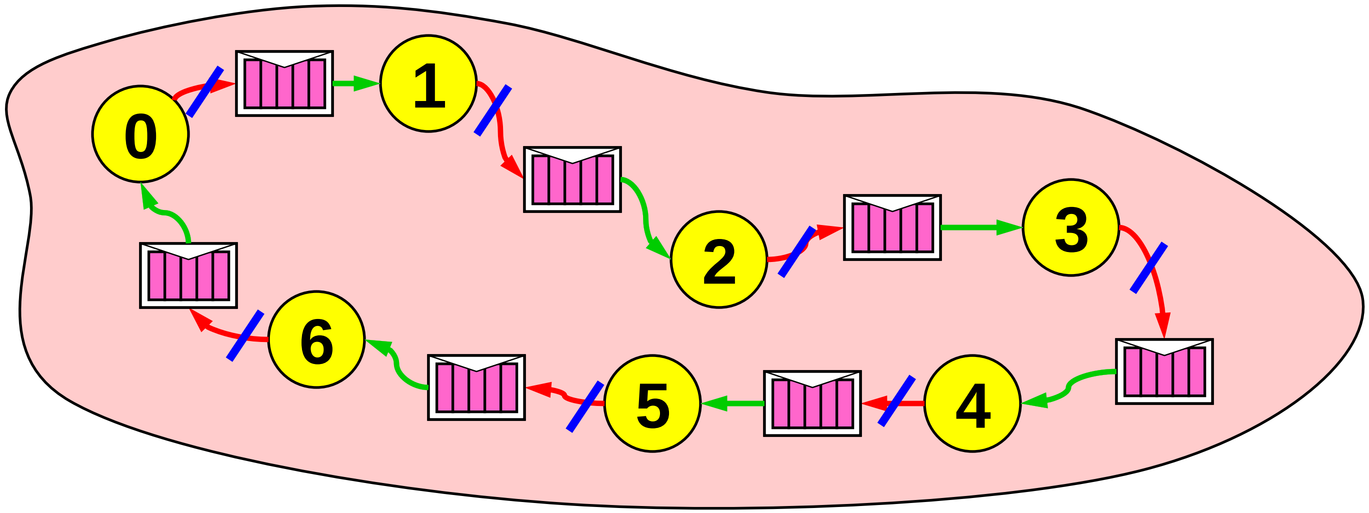
- Non-blocking **receive**



/// = waiting until operation locally completed

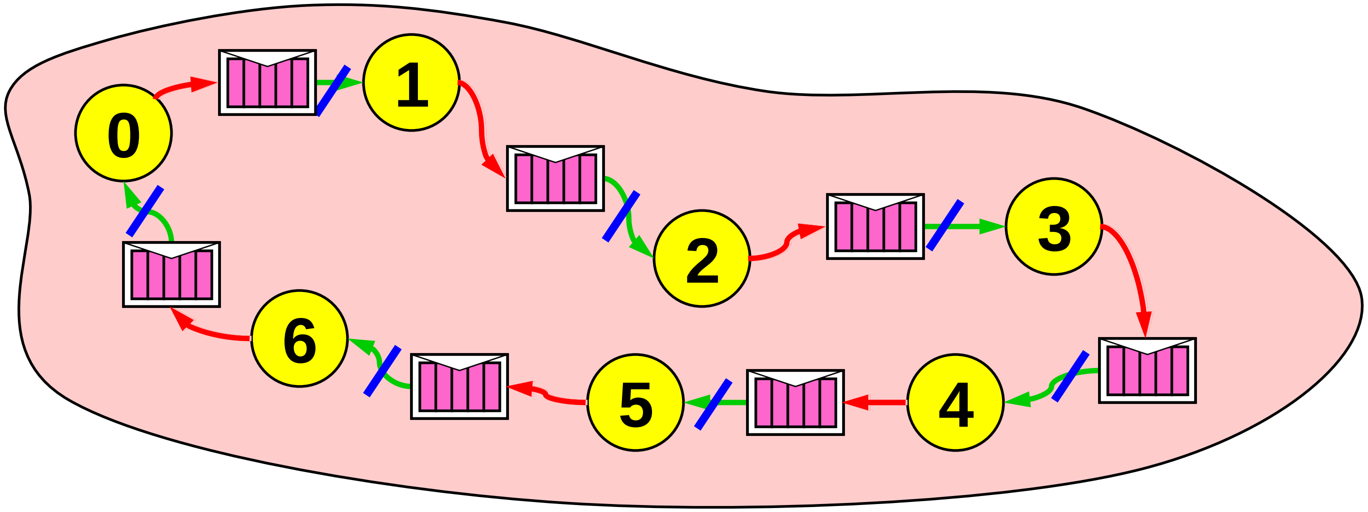
Non-Blocking Send

- Initiate non-blocking send
  → in the ring example: Initiate non-blocking send to the right neighbor
- Do some work:
  → in the ring example: Receiving the message from left neighbor
- Now, the message transfer can be completed
- Wait for non-blocking send to complete /



Non-Blocking Receive

- Initiate non-blocking receive
  - in the ring example: Initiate non-blocking receive from left neighbor
- Do some work:
  - in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for non-blocking receive to complete



Handles, already known

- Predefined handles
  - defined in mpi.h / mpif.h
  - communicator, e.g., MPI_COMM_WORLD
  - datatype, e.g., MPI_INT, MPI_INTEGER, …

- Handles **can** also be stored in local variables
  - memory for datatype handles
    - in C: MPI_Datatype
    - in Fortran: INTEGER
  - memory for communicator handles
    - in C: MPI_Comm
    - in Fortran: INTEGER

Request Handles

Request handles

- are used for non-blocking communication
- **must** be stored in local variables

  C: MPI_Request

  Fortran: INTEGER

- the value
  - **is generated** by a non-blocking communication routine
  - **is used** (and freed) in the MPI_WAIT routine

Non-blocking Synchronous Send

- C:

  MPI_Issend(buf, count, datatype, dest, tag, comm,
  OUT &*request_handle*);

  MPI_Wait(INOUT &request_handle, &*status*);

- Fortran:

  CALL MPI_ISSEND(buf, count, datatype, dest, tag, comm,
  OUT *request_handle*, *ierror*)

  CALL MPI_WAIT(INOUT request_handle, *status*, *ierror*)

- <u>buf</u> must not be used between <u>Issend</u> and <u>Wait</u> (in all progr. languages)
  MPI 1.1, page 40, lines 44-45
- "<u>Issend</u> + <u>Wait</u> directly after Issend" is equivalent to blocking call (<u>Ssend</u>)
- <u>status</u> is not used in <u>Issend</u>, but in <u>Wait</u> (with send: nothing returned)
- Fortran problems, see MPI-2, Chap. 10.2.2, pp 284-290

Non-blocking Receive

- C:

    MPI_Irecv(*buf*, count, datatype, source, tag, comm,
    OUT &*request_handle*);

    MPI_Wait(INOUT &request_handle, &*status*);

- Fortran:

    CALL MPI_IRECV (*buf*, count, datatype, source, tag, comm,
    OUT *request_handle*, *ierror*)

    CALL MPI_WAIT(INOUT request_handle, *status*, *ierror*)

- <u>buf</u> must not be used between <u>Irecv</u> and <u>Wait</u> (in all progr. languages)

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking.
- A blocking send can be used with a non-blocking receive,
  and vice-versa.
- Non-blocking sends can use any mode
  - standard – MPI_ISEND
  - synchronous – MPI_ISSEND
  - buffered – MPI_IBSEND
  - ready – MPI_IRSEND

Completion

- C:

  MPI_Wait(&request_handle, &*status*);

  MPI_Test(&request_handle, &*flag*, &*status*);

- Fortran:

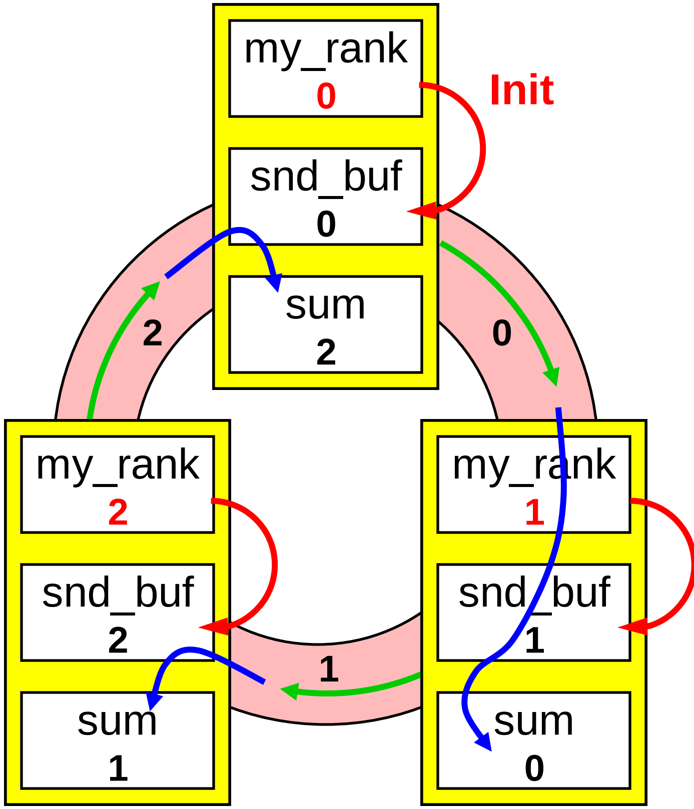
  CALL MPI_WAIT(request_handle, *status*, *ierror*)

  CALL MPI_TEST(request_handle, *flag*, *status*, *ierror*)

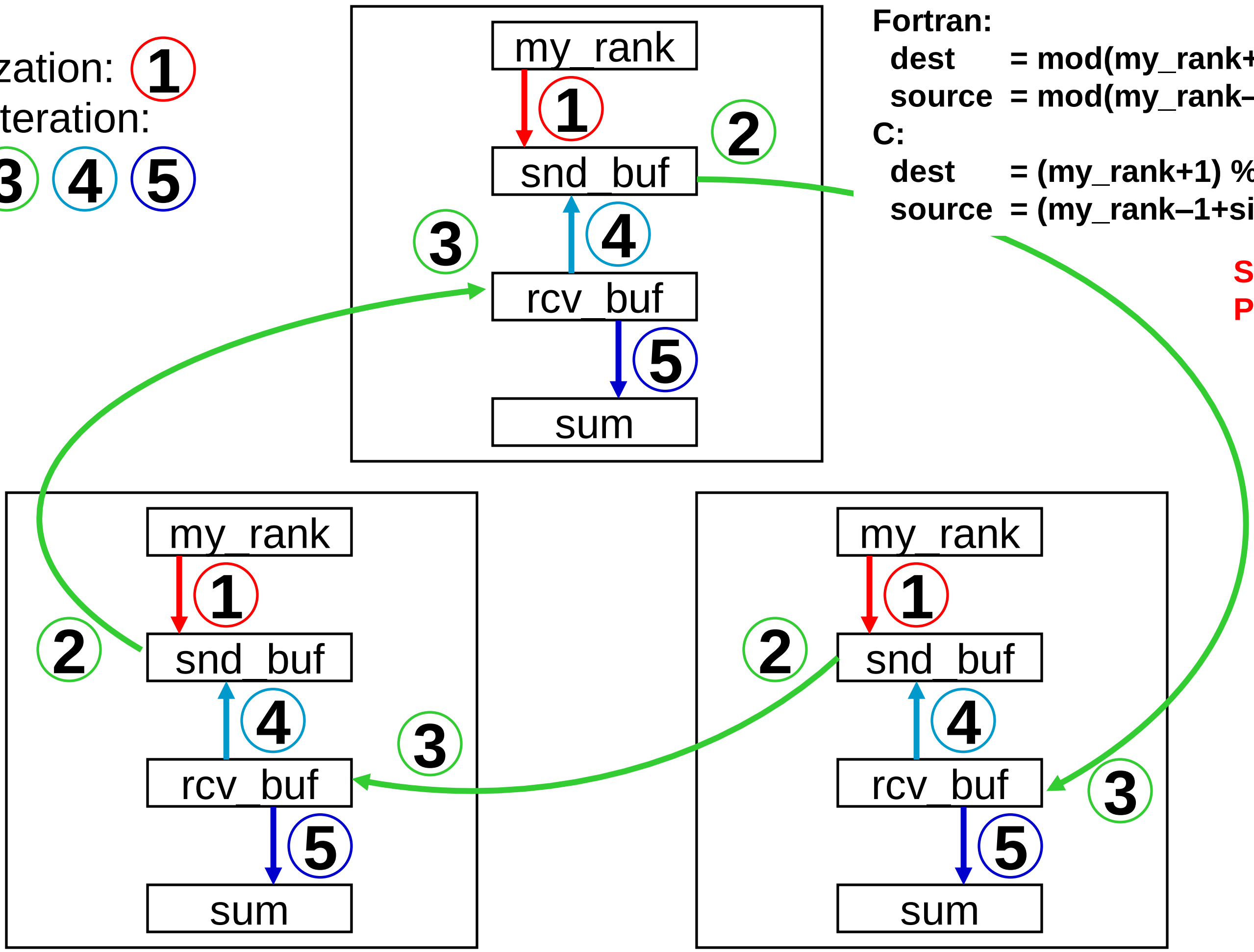
- one must
  - WAIT or
  - loop with TEST until request is completed, i.e., flag == 1 or .TRUE.

Exercise — Rotating information around a ring

- A set of processes are arranged in a ring.
- Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- Each process passes this on to its neighbor on the right.
- Each processor calculates the sum of all values.
- Keep passing it around the ring until the value is back where it started, i.e.
- each process calculates sum of all ranks.
- Use non-blocking MPI_Issend
  - to avoid deadlocks
  - to verify the correctness, because blocking synchronous send will cause a deadlock



Initialization: 1
Each iteration: 2 3 4 5



Fortran:

```
dest     = mod(my_rank+1,size)
source   = mod(my_rank–1+size,size)
```

C:

```
dest     = (my_rank+1) % size;
source   = (my_rank–1+size) % size;
```

Single Program !!!