Parallel programming / computation

Sultan ALPAR
s.alpar@iitu.edu.kz

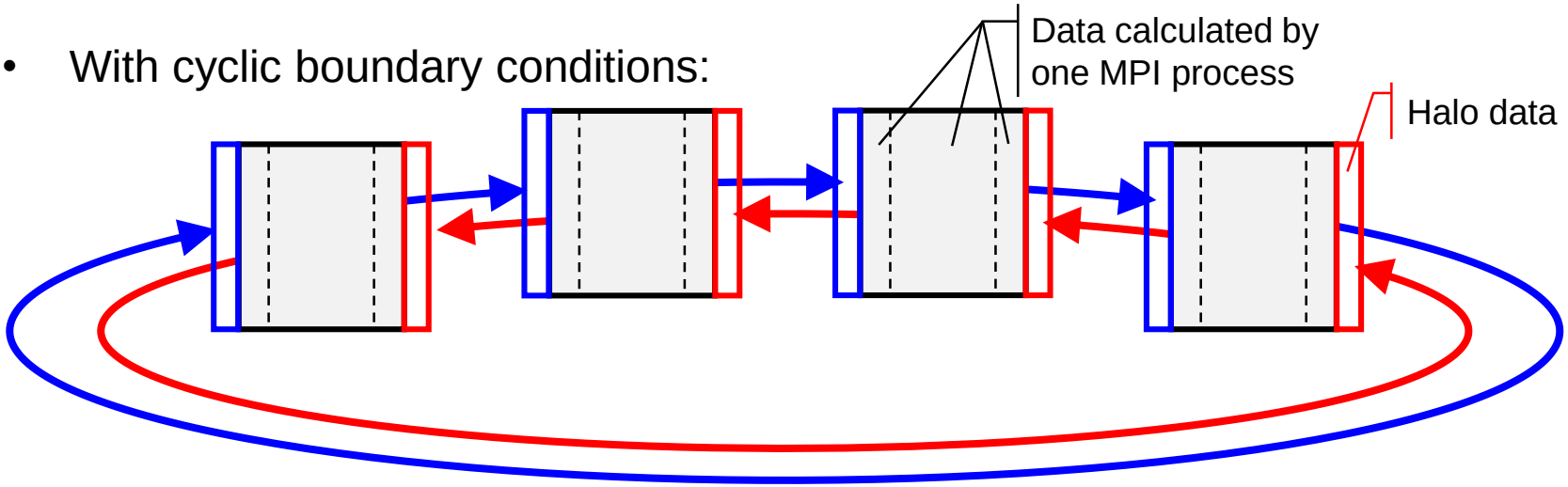
IITU

Lecture 4

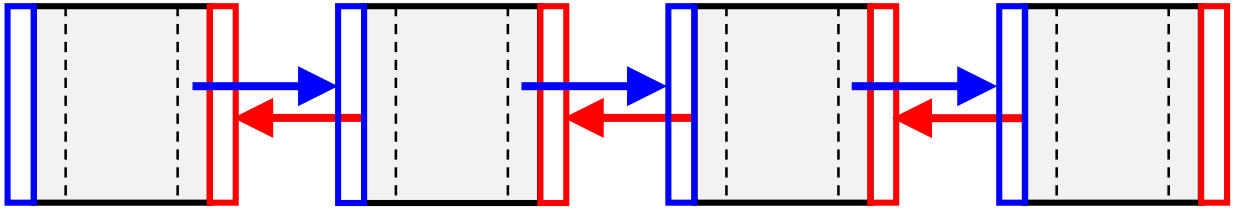
Nonblocking Communication

Task: Halo Communication

- With cyclic boundary conditions:

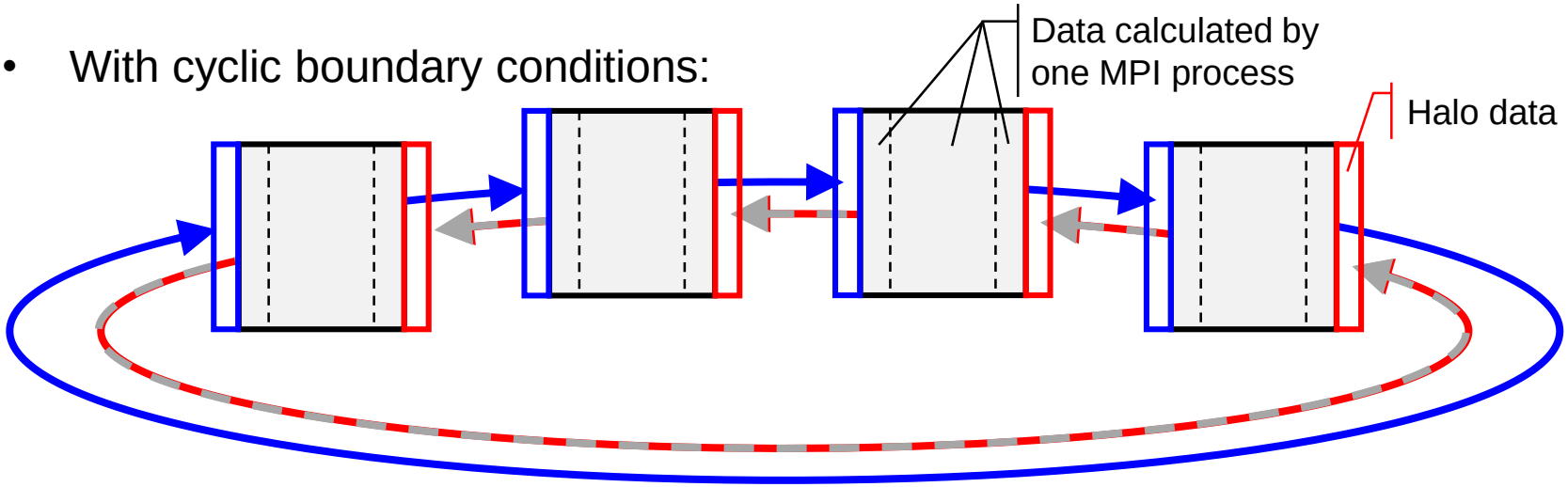


- Non-cyclic:

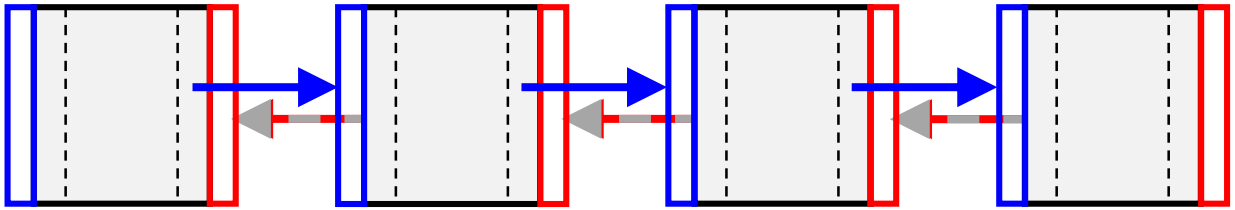


Task: Halo Communication

- With cyclic boundary conditions:



- Non-cyclic:



Let‘s concentrate on the blue direction (from left to right, clockwise)

Blocking Routines → Risk of Deadlocks & Serializations

For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

Blocking Routines → Risk of Deadlocks & Serializations

For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

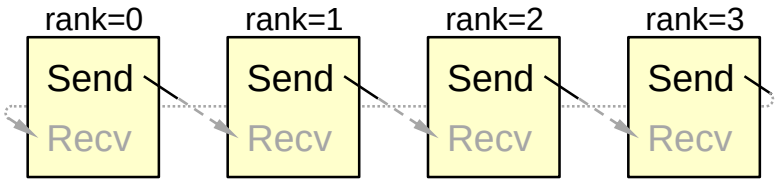
If the MPI library chooses the **synchronous** protocol,
i.e. MPI_Send waits until MPI_Recv is called, then:

Blocking Routines → Risk of Deadlocks & Serializations

For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol,
i.e. MPI_Send waits until MPI_Recv is called, then:

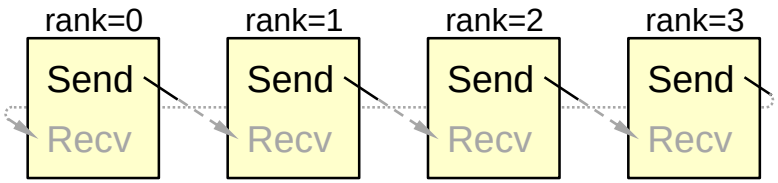


Blocking Routines → Risk of Deadlocks & Serializations

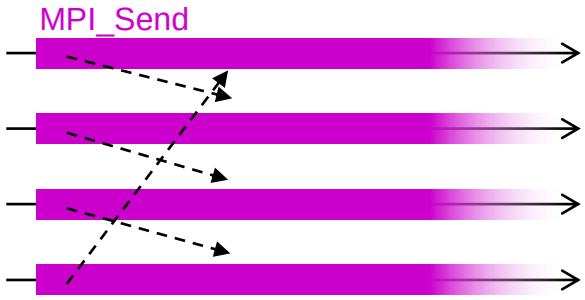
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol,
i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes



Rank

0

1

2

3

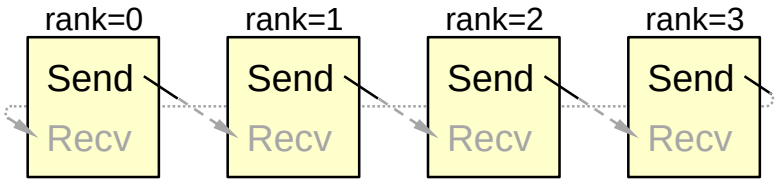
→ **Deadlock**

Blocking Routines → Risk of Deadlocks & Serializations

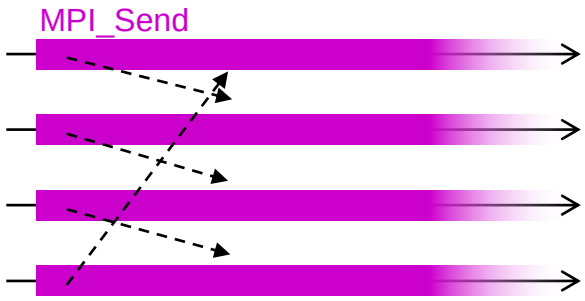
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

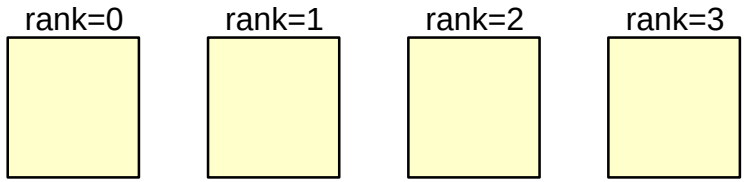


→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:

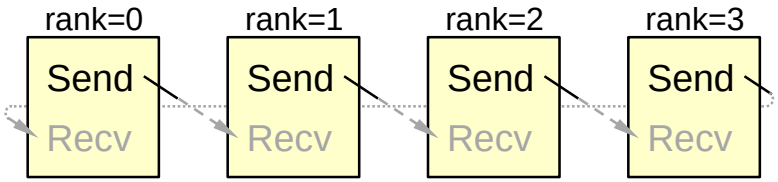


Blocking Routines → Risk of Deadlocks & Serializations

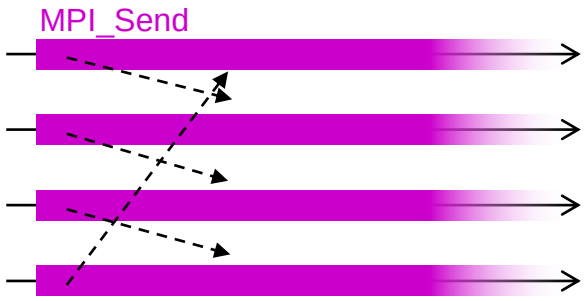
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

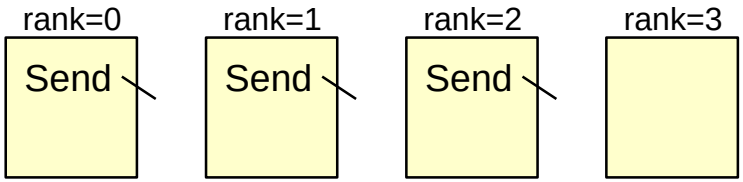


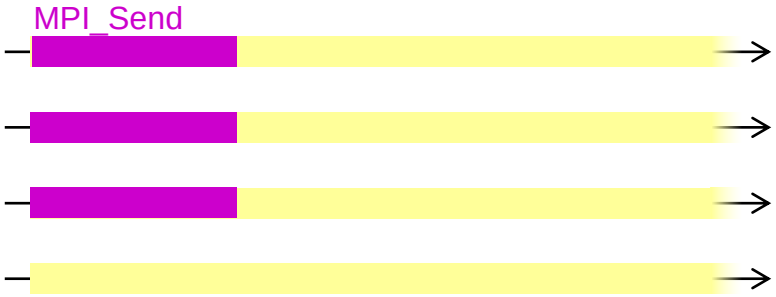
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



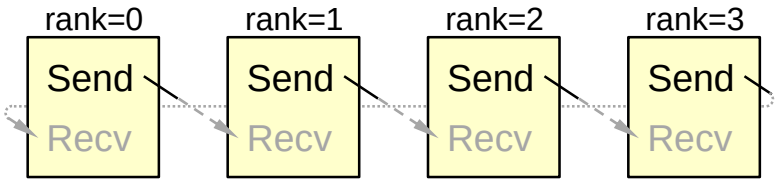


Blocking Routines → Risk of Deadlocks & Serializations

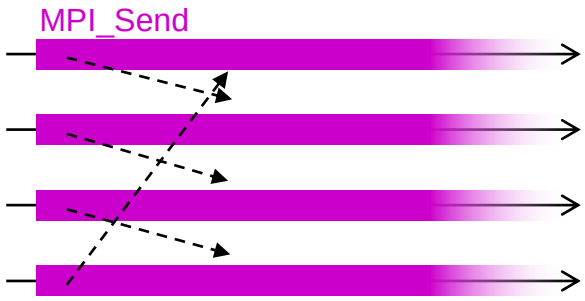
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

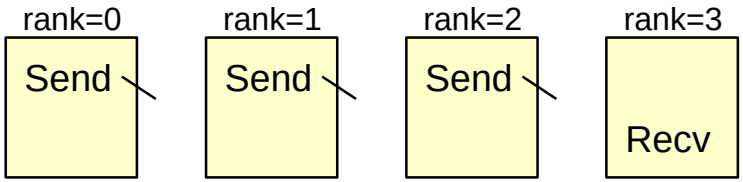


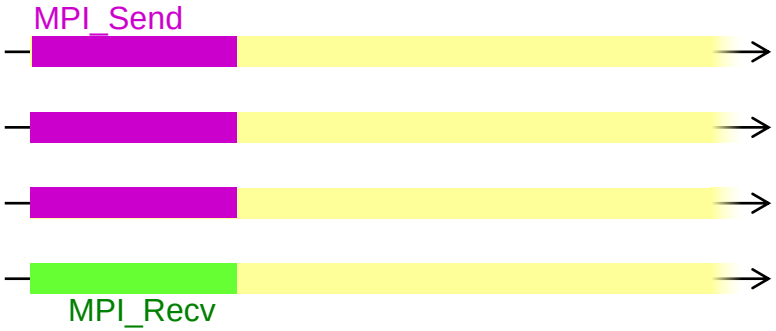
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



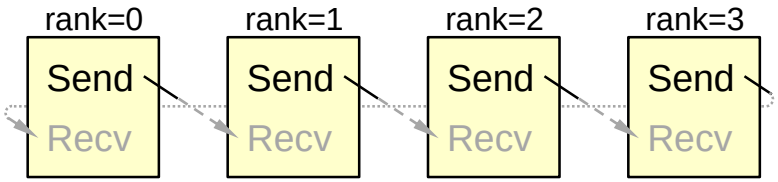


Blocking Routines → Risk of Deadlocks & Serializations

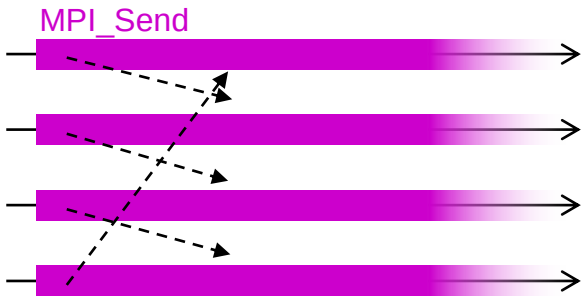
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

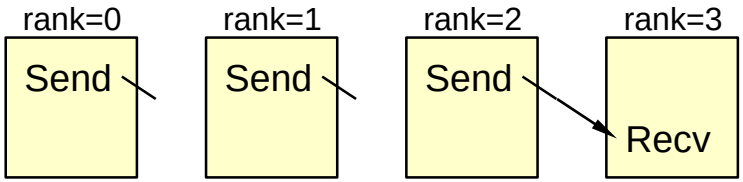


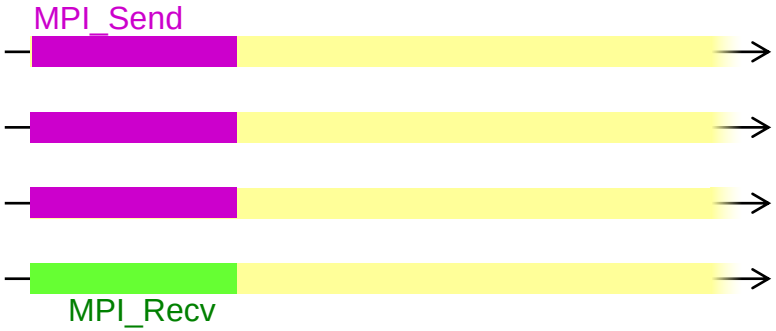
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



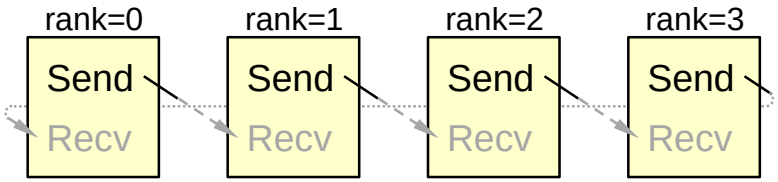


Blocking Routines → Risk of Deadlocks & Serializations

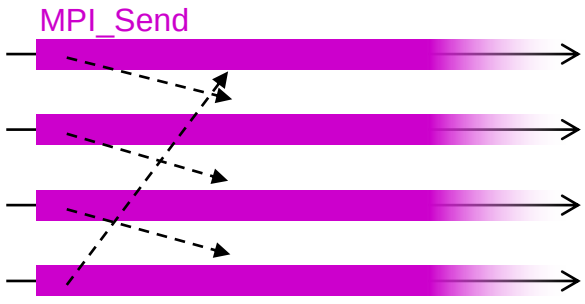
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

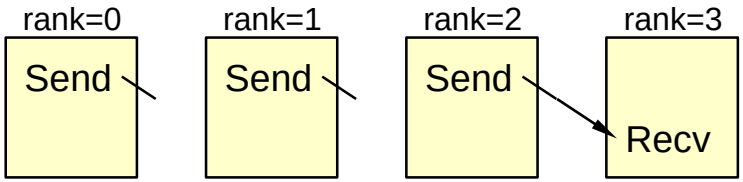


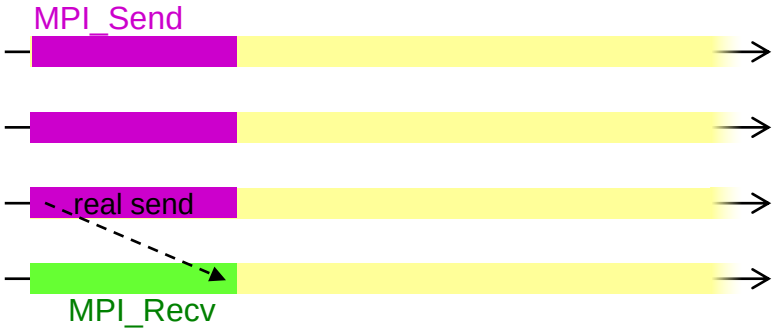
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



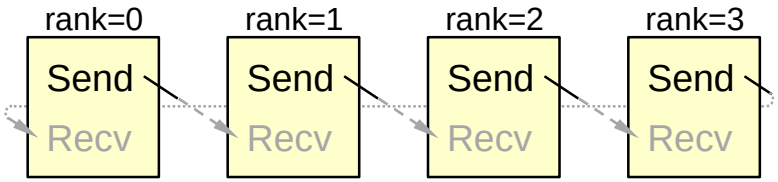


Blocking Routines → Risk of Deadlocks & Serializations

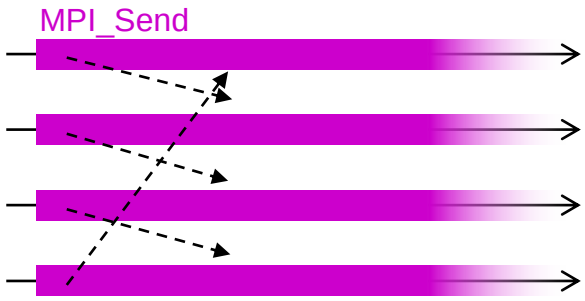
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

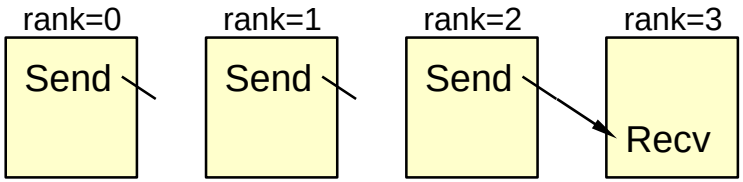


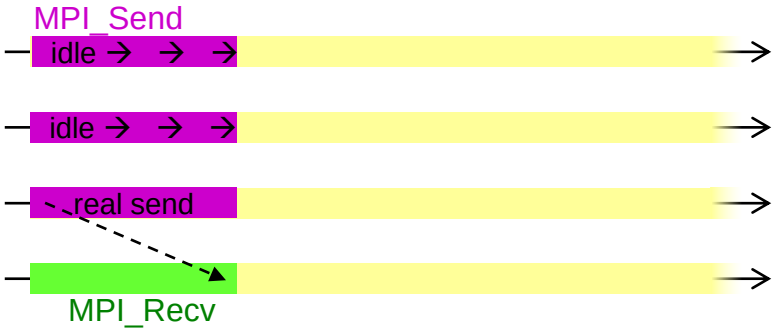
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



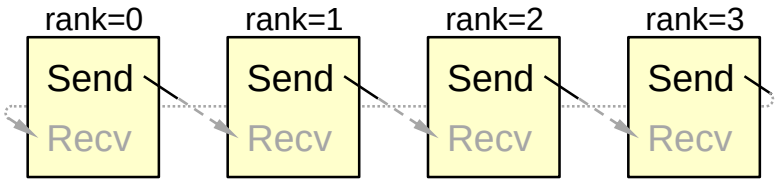


Blocking Routines → Risk of Deadlocks & Serializations

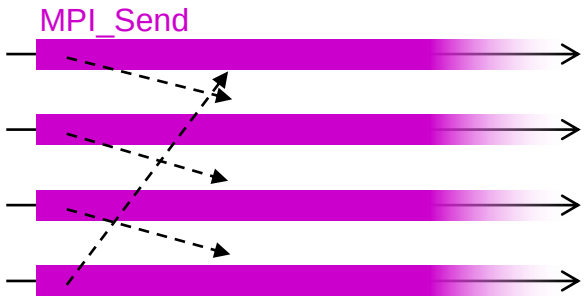
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

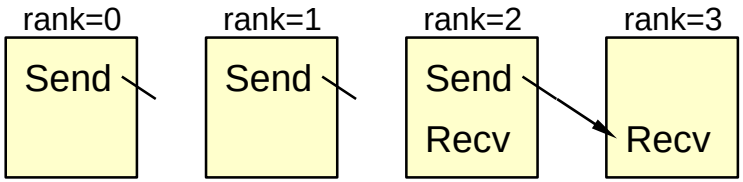


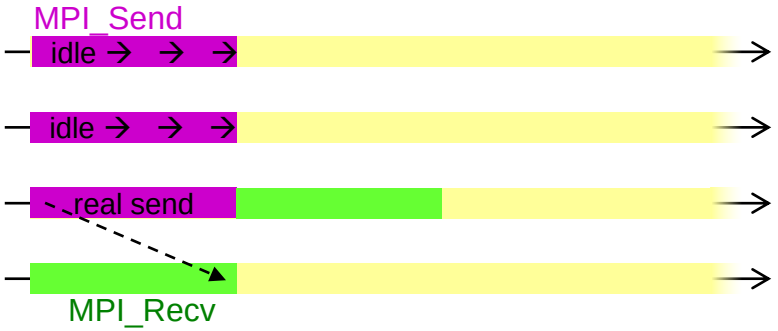
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



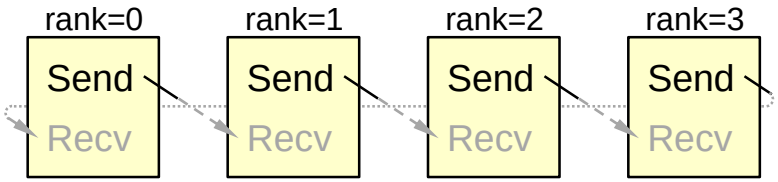


Blocking Routines → Risk of Deadlocks & Serializations

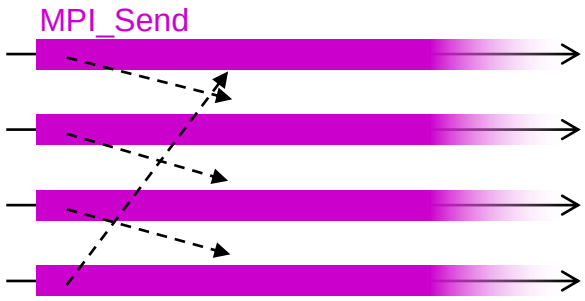
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

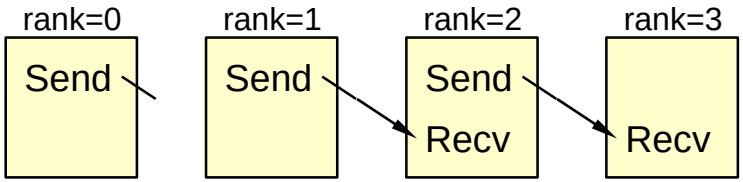


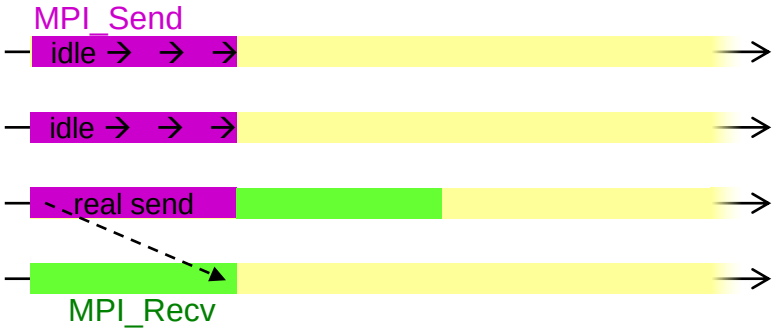
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



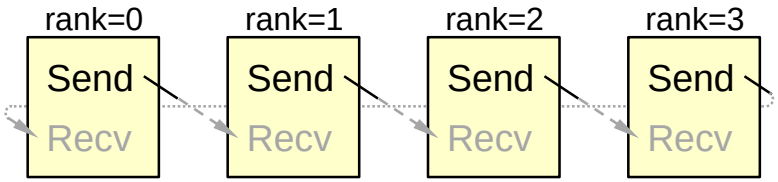


Blocking Routines → Risk of Deadlocks & Serializations

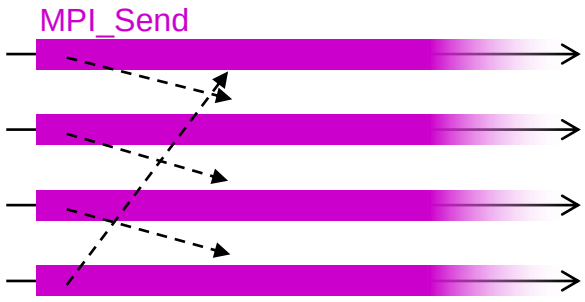
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

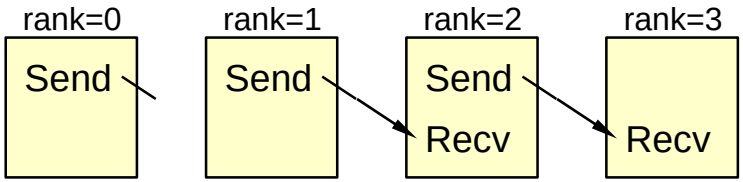


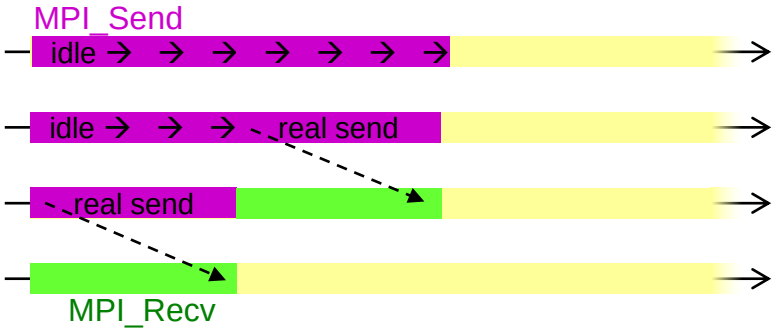
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



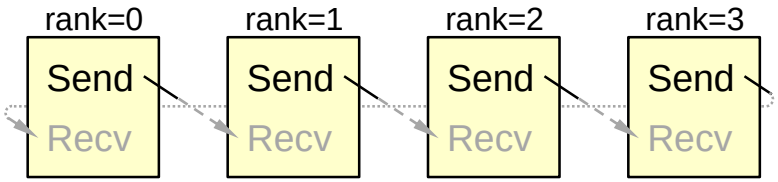


Blocking Routines → Risk of Deadlocks & Serializations

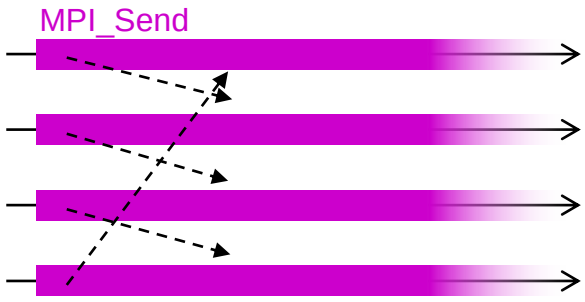
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

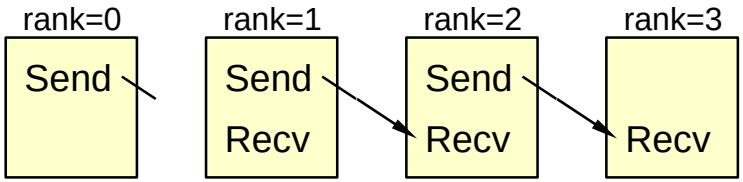


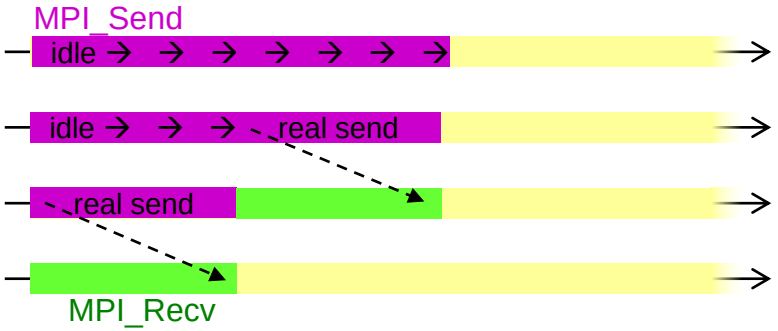
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



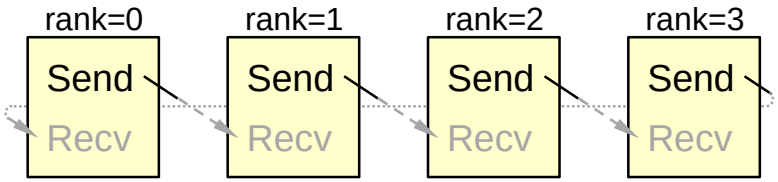


Blocking Routines → Risk of Deadlocks & Serializations

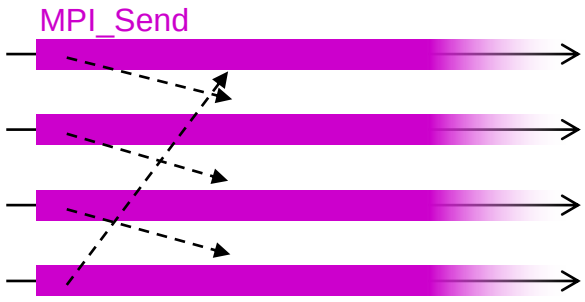
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

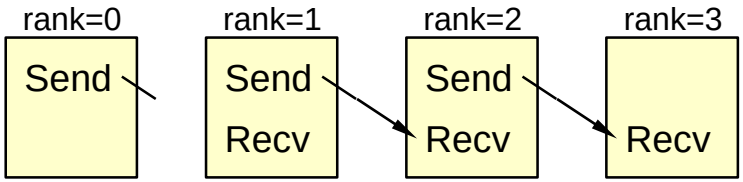


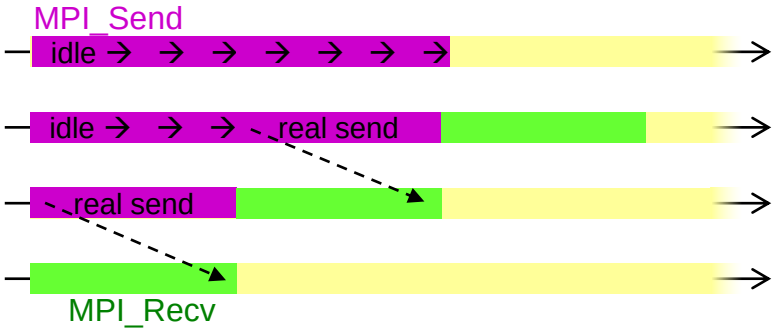
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



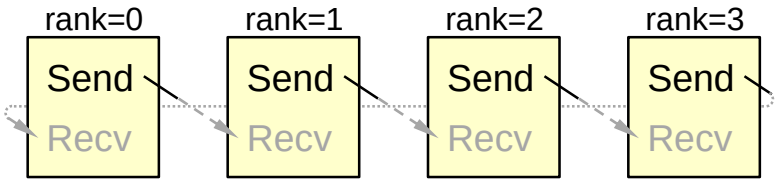


Blocking Routines → Risk of Deadlocks & Serializations

For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



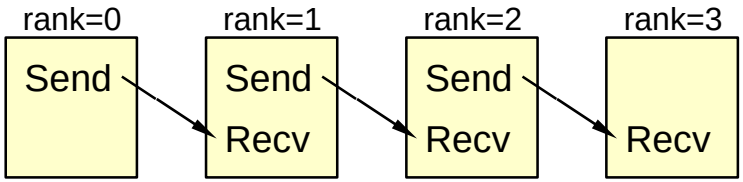
Timelines of all processes

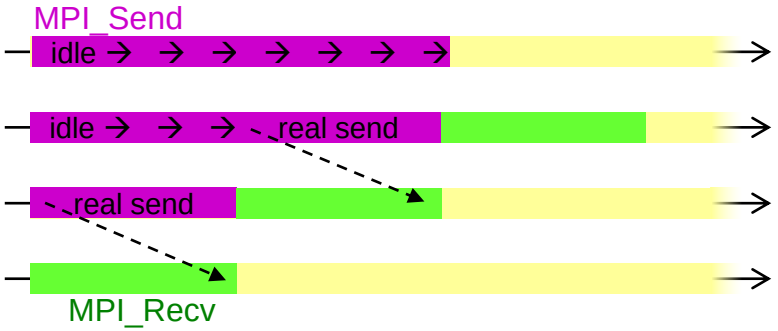
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:



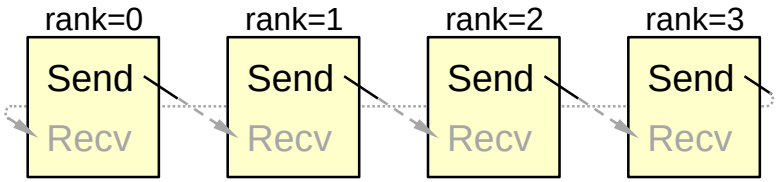


Blocking Routines → Risk of Deadlocks & Serializations

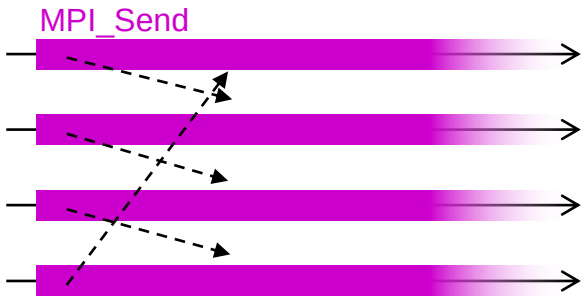
For cyclic boundary:

```
MPI_Send(…, right_rank, …)
MPI_Recv(…, left_rank,  …)
```

If the MPI library chooses the **synchronous** protocol, i.e. MPI_Send waits until MPI_Recv is called, then:



Timelines of all processes

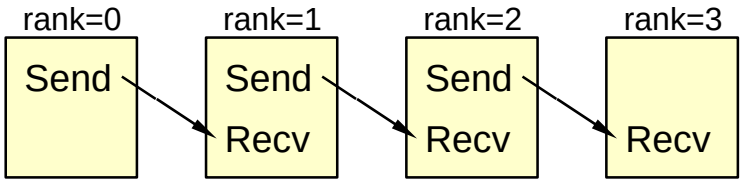


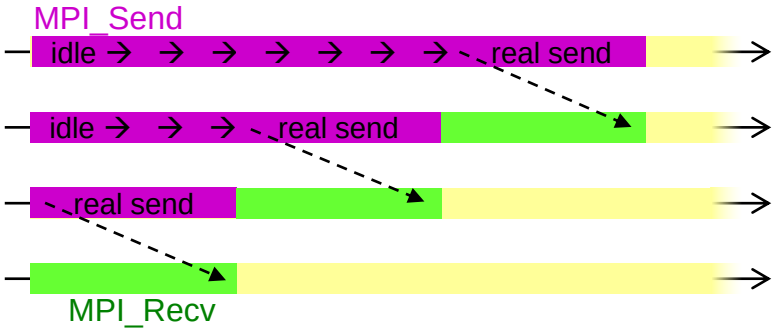
→ **Deadlock**

For non-cyclic boundary:

```
if (myrank < size-1) MPI_Send(…, right_rank, …);
if (myrank > 0) MPI_Recv(…, left_rank,  …);
```

If the MPI library chooses the **synchronous** protocol:

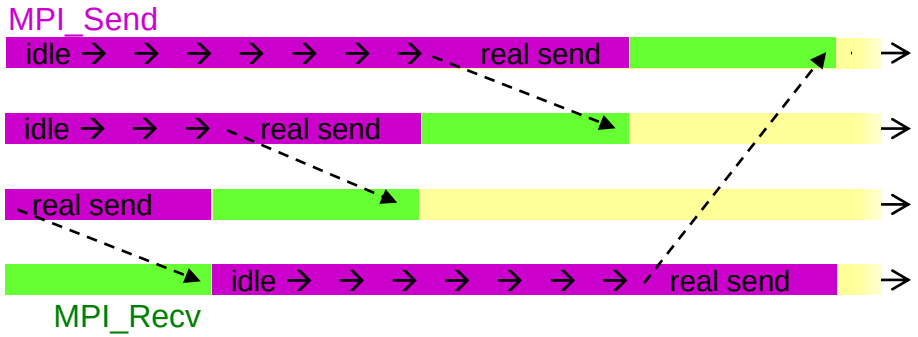




→ **Serialization**

Cyclic communication – other bad ideas

```
if (myrank < size-1) {
   MPI_Send(…, right_rank, …);
   MPI_Recv(…, left_rank,  …);
} else {
   MPI_Recv(…, left_rank,  …);
   MPI_Send(…, right_rank, …);
}
```

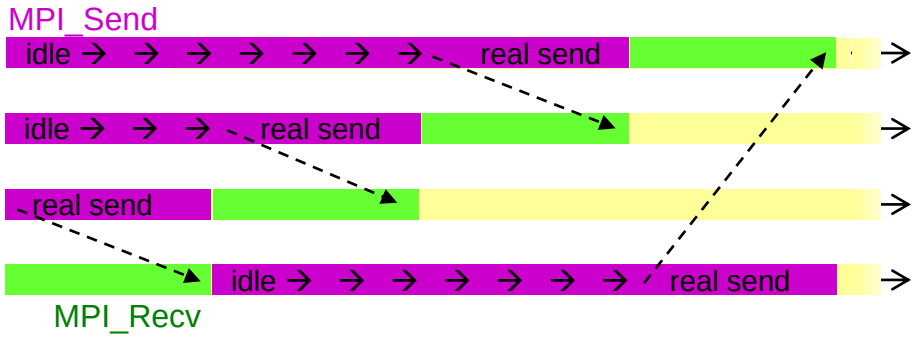


→ **Serialization**

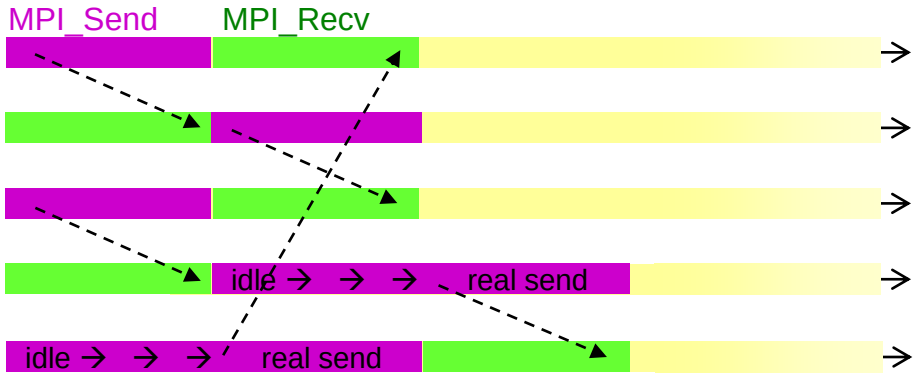
(If the MPI library chooses the synchronous protocol)

Cyclic communication – other bad ideas

```
if (myrank < size-1) {
   MPI_Send(…, right_rank, …);
   MPI_Recv(…, left_rank,  …);
} else {
   MPI_Recv(…, left_rank,  …);
   MPI_Send(…, right_rank, …);
}
```




```
if (myrank%2 == 0) {
   MPI_Send(…, right_rank, …);
   MPI_Recv(…, left_rank,  …);
} else {
   MPI_Recv(…, left_rank,  …);
   MPI_Send(…, right_rank, …);
}
```



→ **Serialization**

(If the MPI library chooses the synchronous protocol)

Non-Blocking Communications

Separate communication into **three phases**:

- Initiate nonblocking communication
  - returns immediately
  - routine name starting with MPI_**I**…

  → it is local,
  i.e., it returns independently of any other process' activity

"**I**" stands for
- **I**mmediately (=local) and
- **I**ncomplete (=nonblocking[1)])

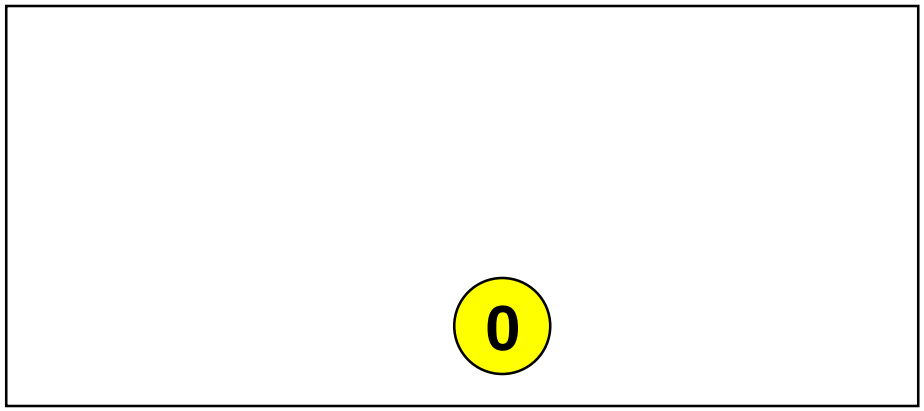
- Do some work (perhaps involving other communications?)
- Wait for nonblocking communication to **complete**, i.e.,
  - the send buffer is read out, or
  - the receive buffer is filled in

[1)] The definition of nonblocking is clarified in MPI-4.0
→ course Chapter 15 *Probe, Persistent Requests, Request_free, Cancel*

Non-Blocking Examples

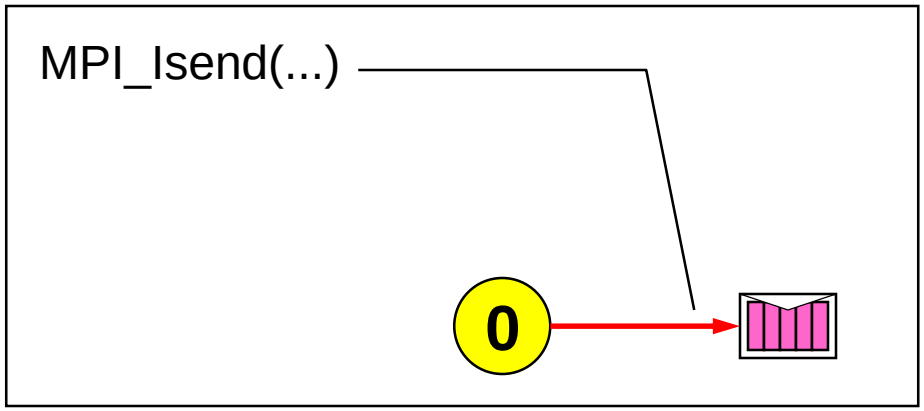
Non-Blocking Examples

- Nonblocking **send**



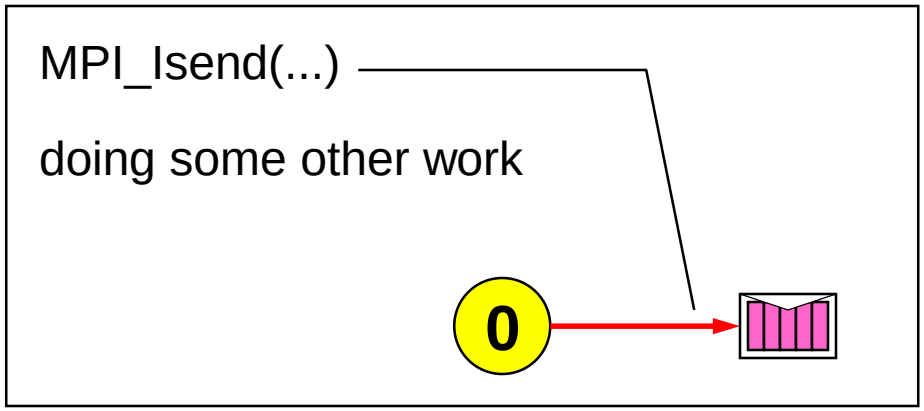
Non-Blocking Examples

- Nonblocking **send**



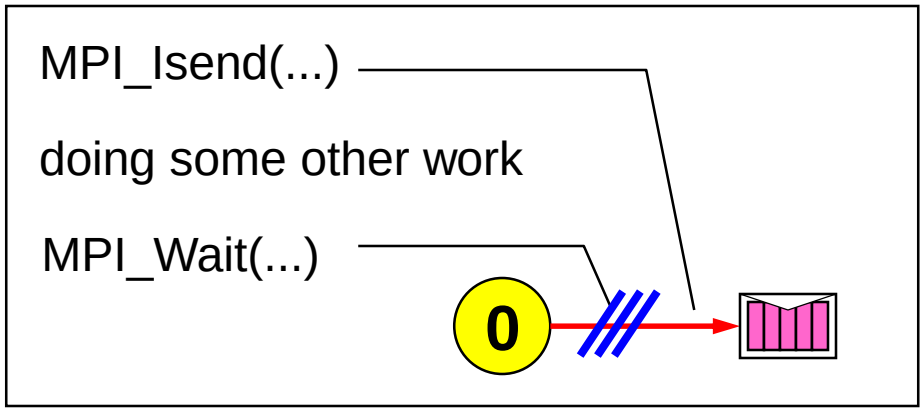
Non-Blocking Examples

- Nonblocking **send**



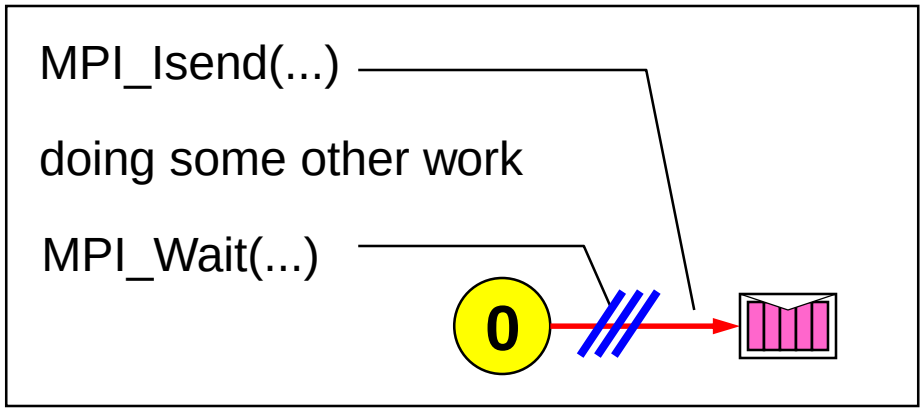
Non-Blocking Examples

- Nonblocking **send**

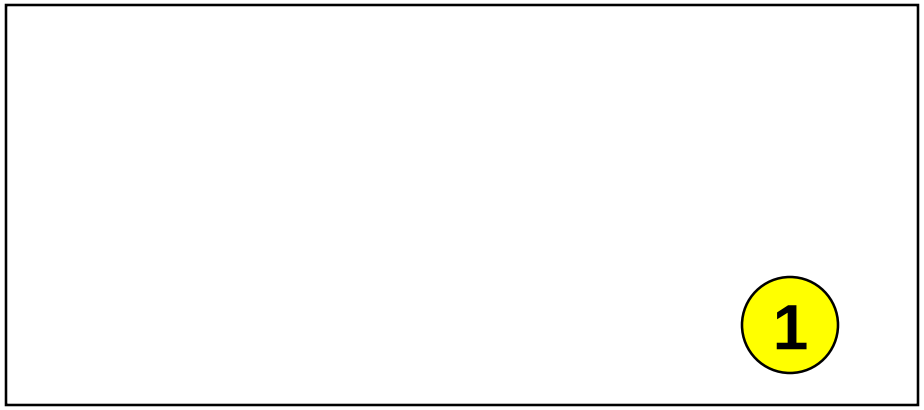


Non-Blocking Examples

- Nonblocking **send**

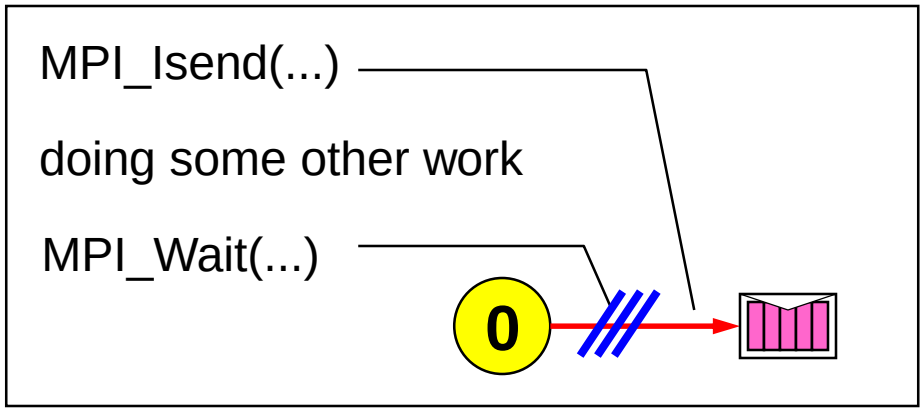


- Nonblocking **receive**

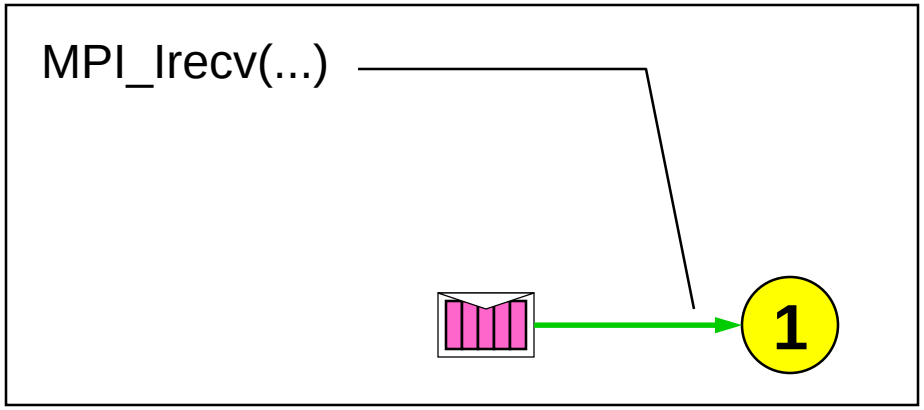


Non-Blocking Examples

- Nonblocking **send**

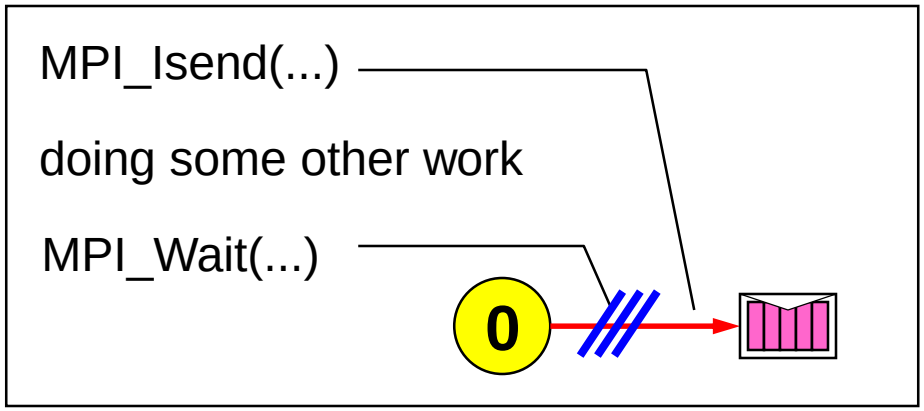


- Nonblocking **receive**

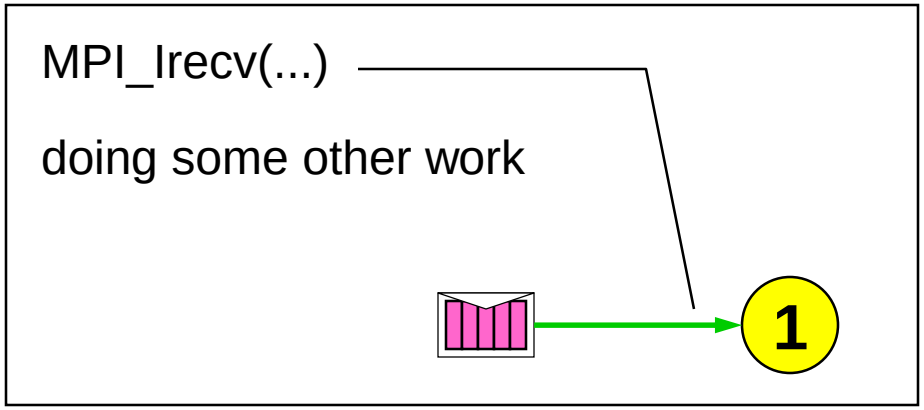


Non-Blocking Examples

- Nonblocking **send**

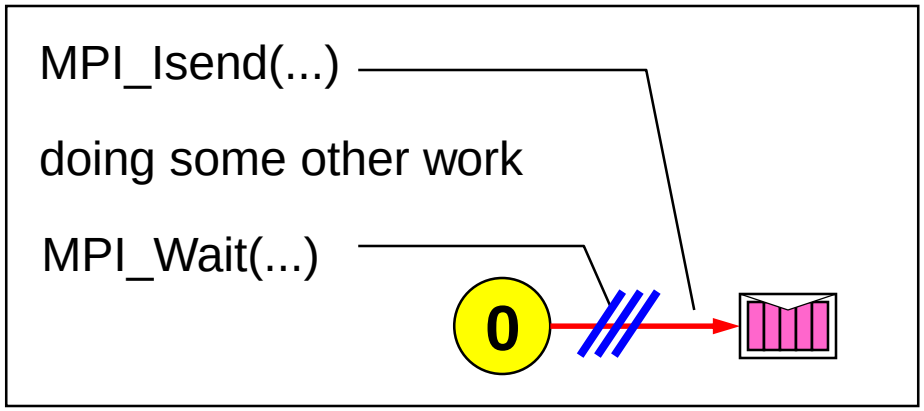


- Nonblocking **receive**

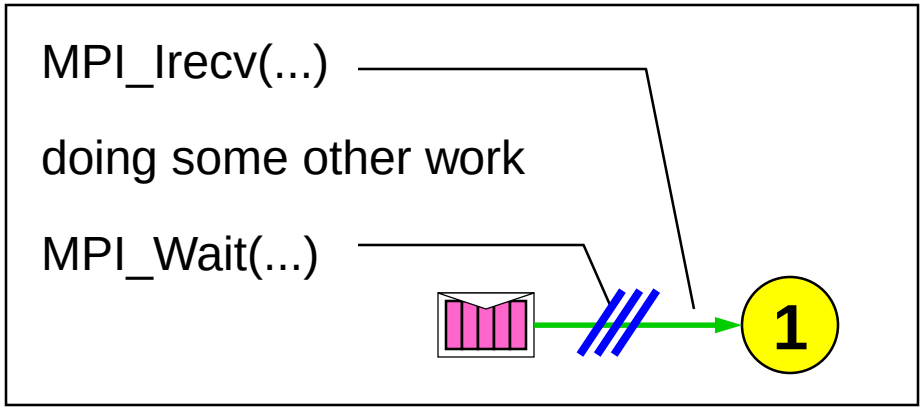


Non-Blocking Examples

- Nonblocking **send**

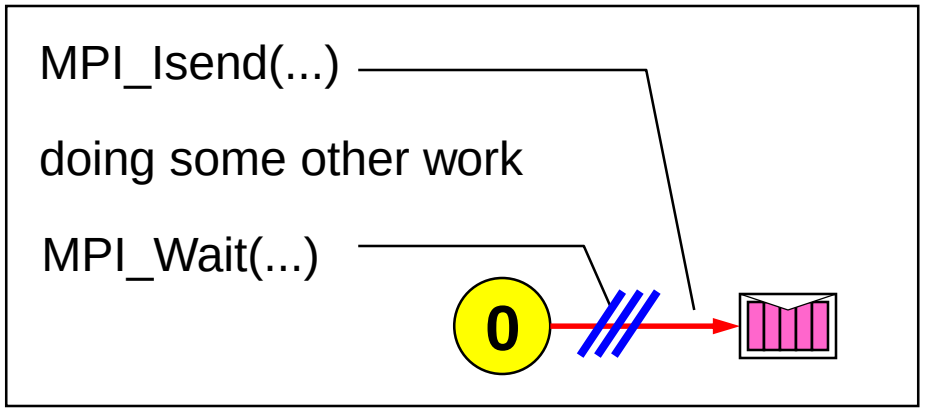


- Nonblocking **receive**

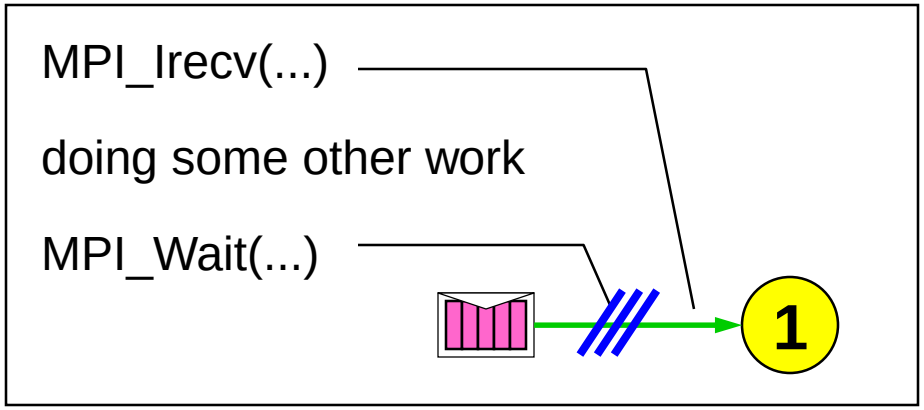


Non-Blocking Examples

- Nonblocking **send**



- Nonblocking **receive**



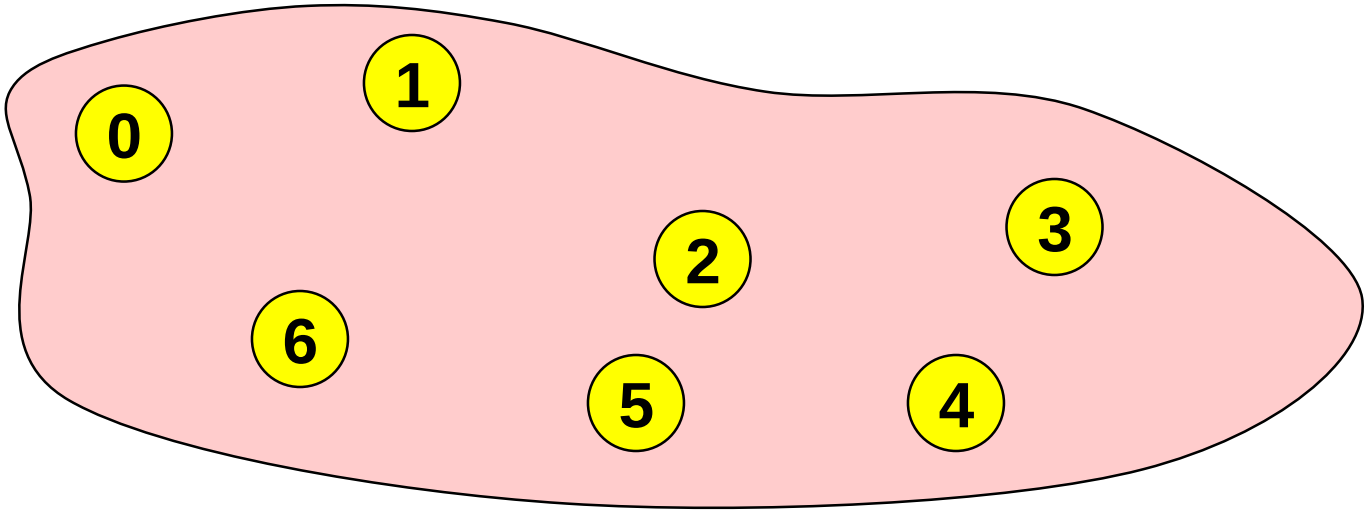
 = waiting until operation locally completed

Non-Blocking Send

- Initiate nonblocking send

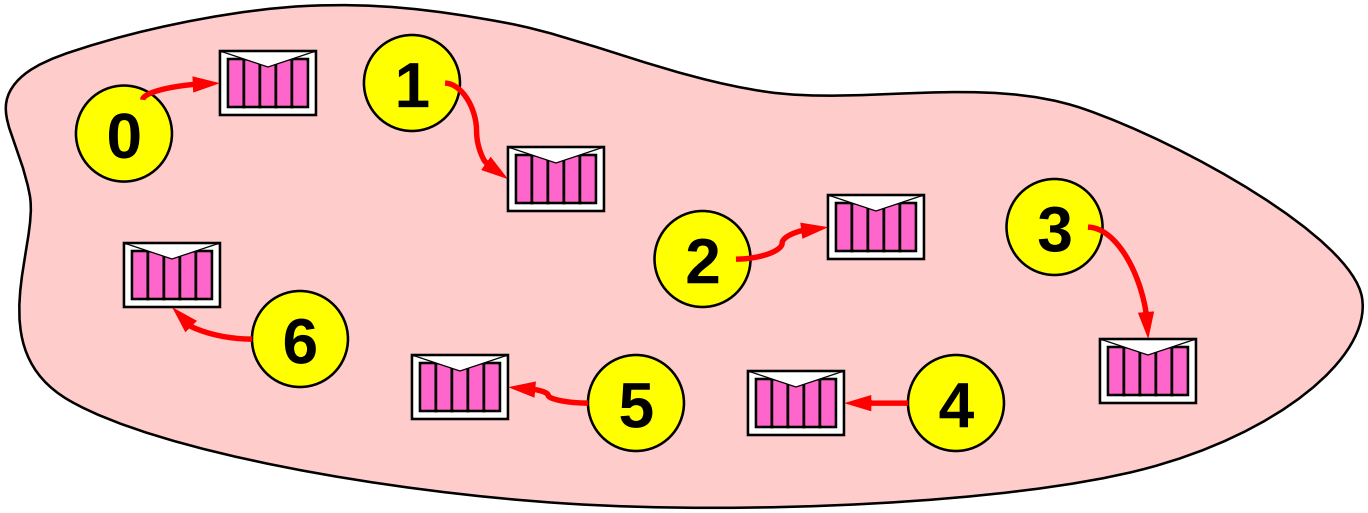
  in the ring example: Initiate nonblocking send to the right neighbor
- Do some work:

  in the ring example: Receiving the message from left neighbor
- Now, the message transfer can be completed
- Wait for nonblocking send to complete



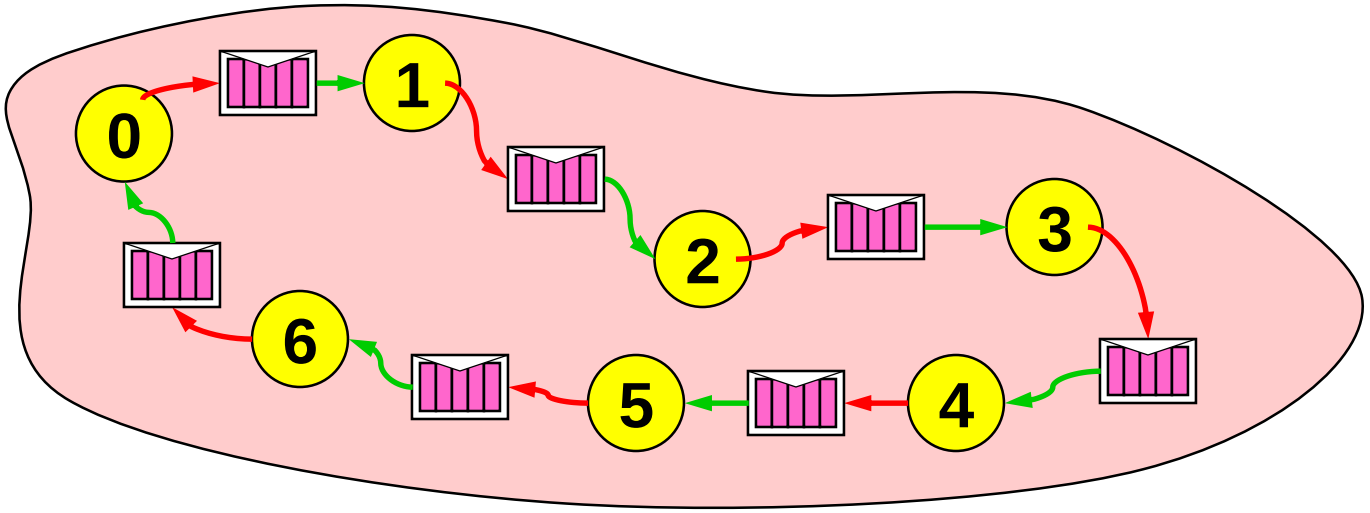
Non-Blocking Send

- Initiate nonblocking send
  - → in the ring example: Initiate nonblocking 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 nonblocking send to complete



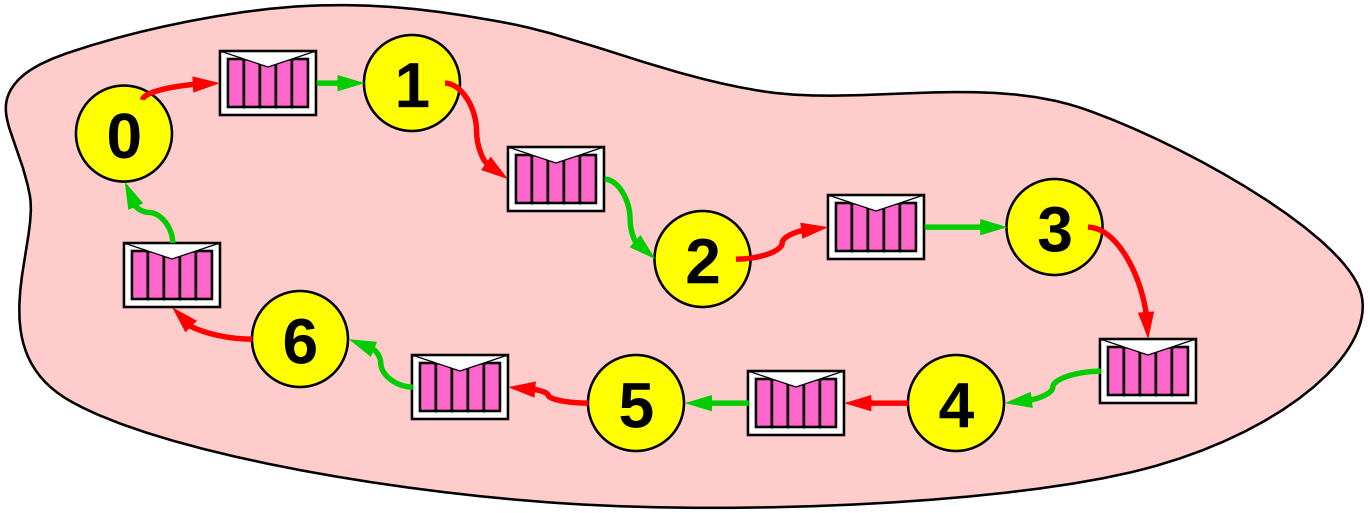
Non-Blocking Send

- Initiate nonblocking send
  - → in the ring example: Initiate nonblocking 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 nonblocking send to complete



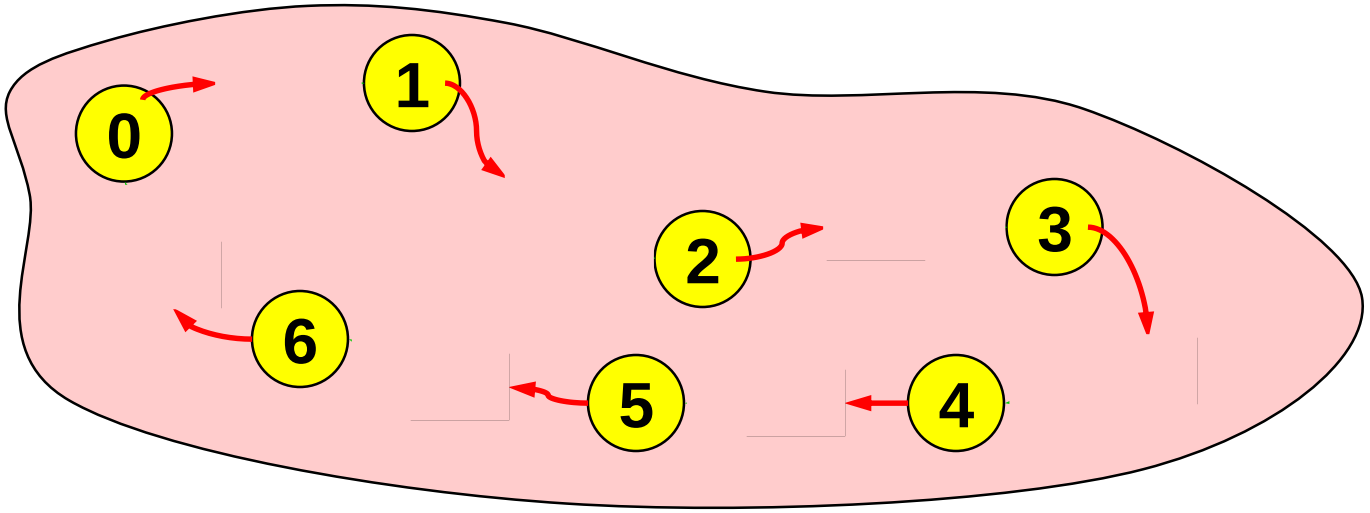
Non-Blocking Send

- Initiate nonblocking send
  - → in the ring example: Initiate nonblocking 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 nonblocking send to complete



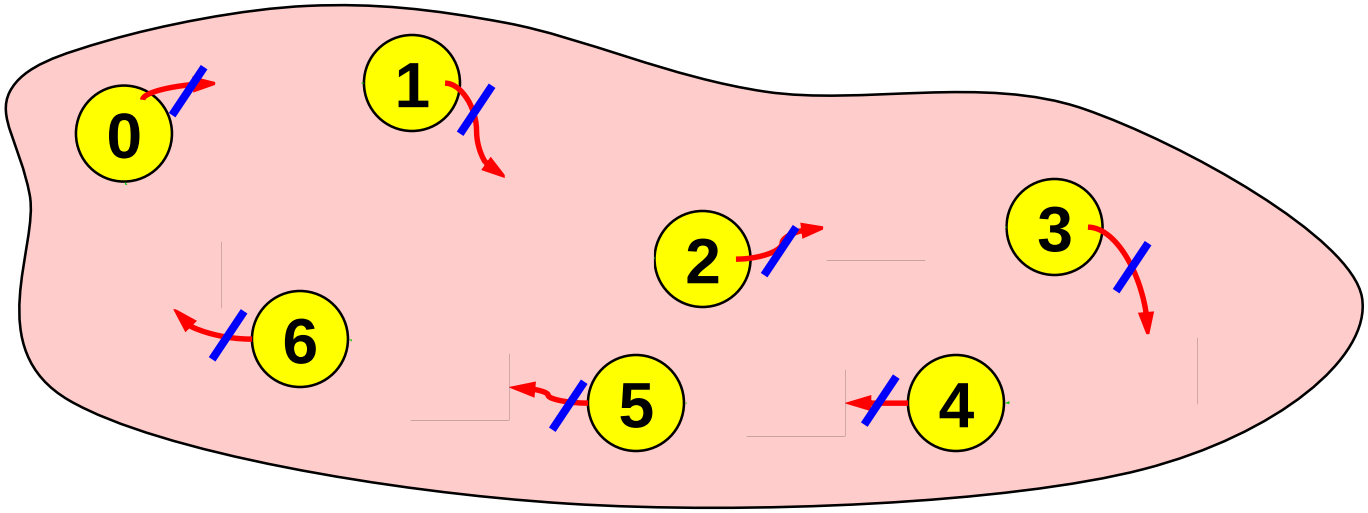
Non-Blocking Send

- Initiate nonblocking send
  - → in the ring example: Initiate nonblocking 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 nonblocking send to complete



Non-Blocking Send

- Initiate nonblocking send
  - in the ring example: Initiate nonblocking 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 nonblocking send to complete

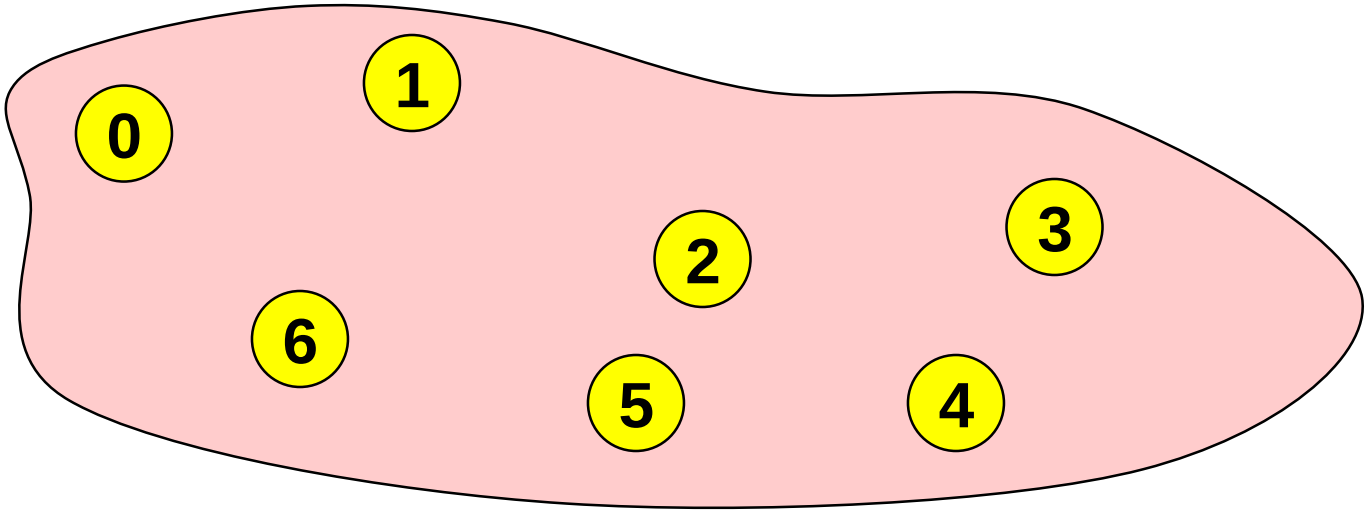


Non-Blocking Receive

- Initiate nonblocking **receive**

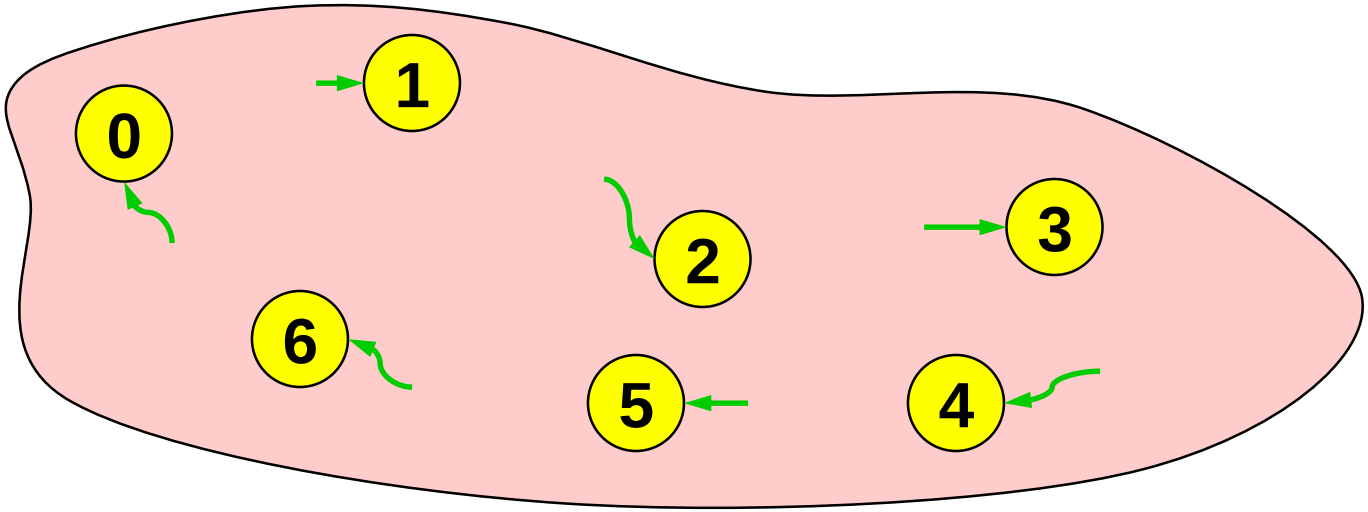
  in the ring example: Initiate nonblocking receive from left neighbor
- Do some work:

  in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for nonblocking receive to complete



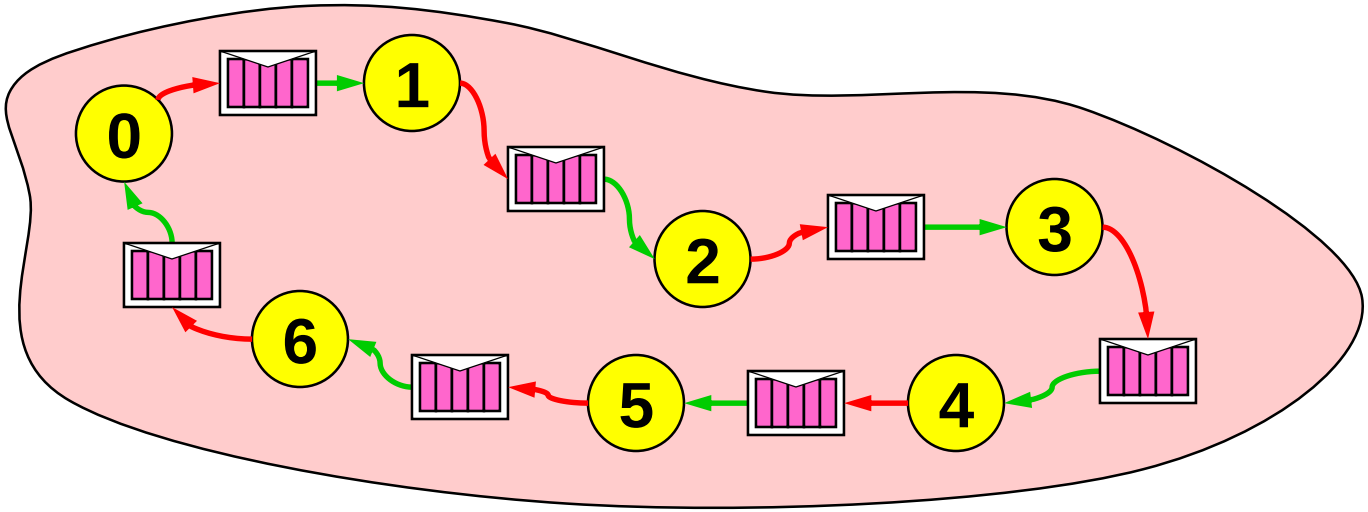
Non-Blocking Receive

- Initiate nonblocking **receive**
  - → in the ring example: Initiate nonblocking 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 nonblocking receive to complete



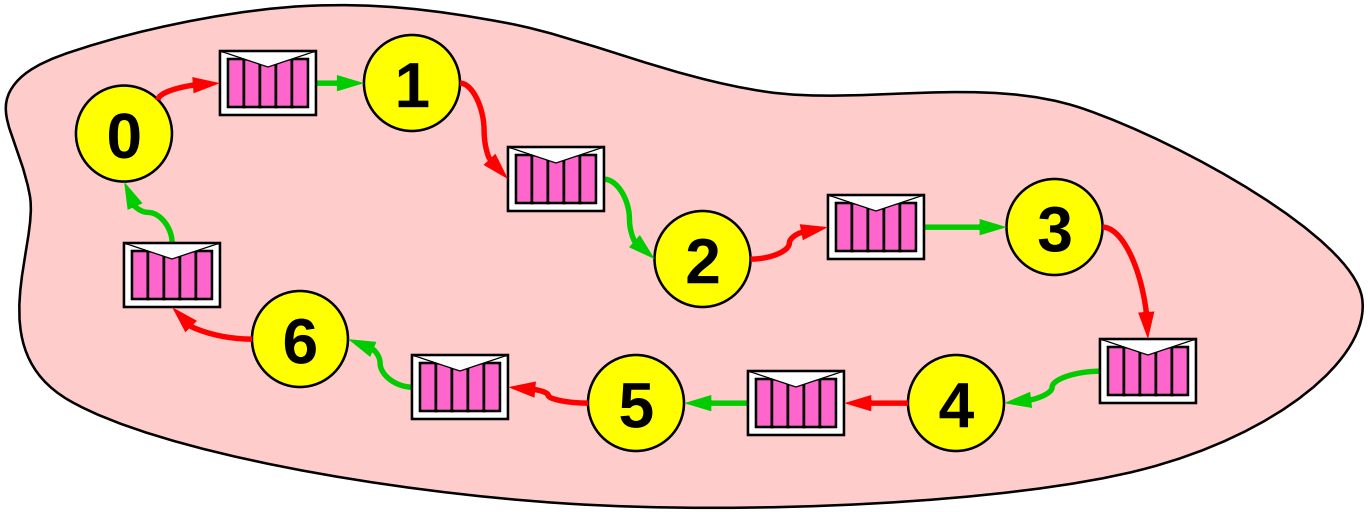
Non-Blocking Receive

- Initiate nonblocking **receive**
  - in the ring example: Initiate nonblocking 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 nonblocking receive to complete



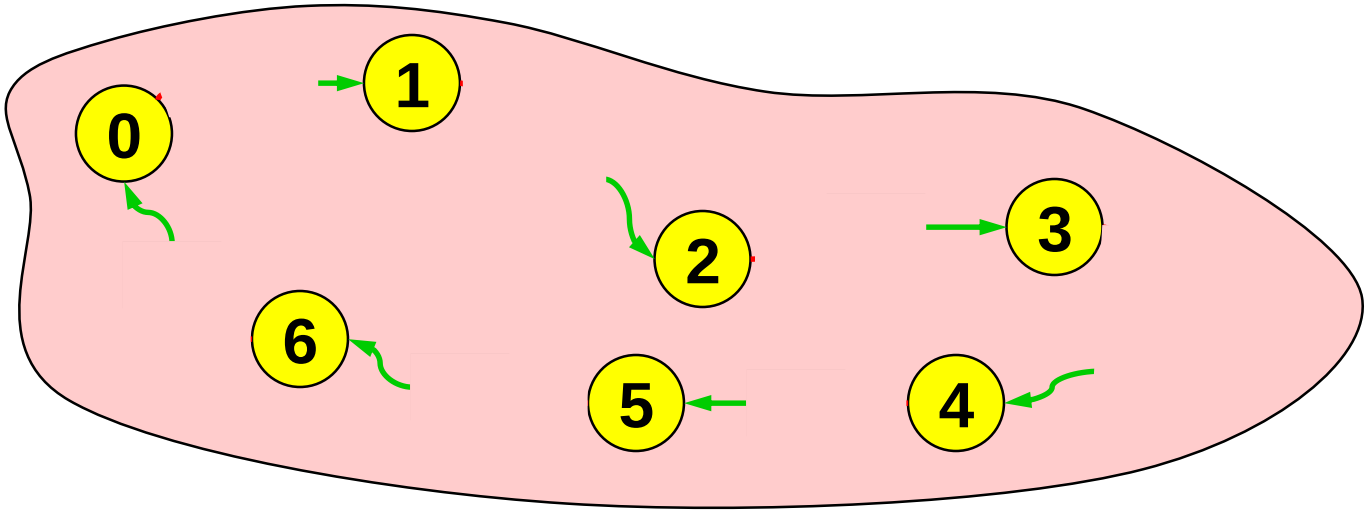
Non-Blocking Receive

- Initiate nonblocking **receive**
  - in the ring example: Initiate nonblocking 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 nonblocking receive to complete



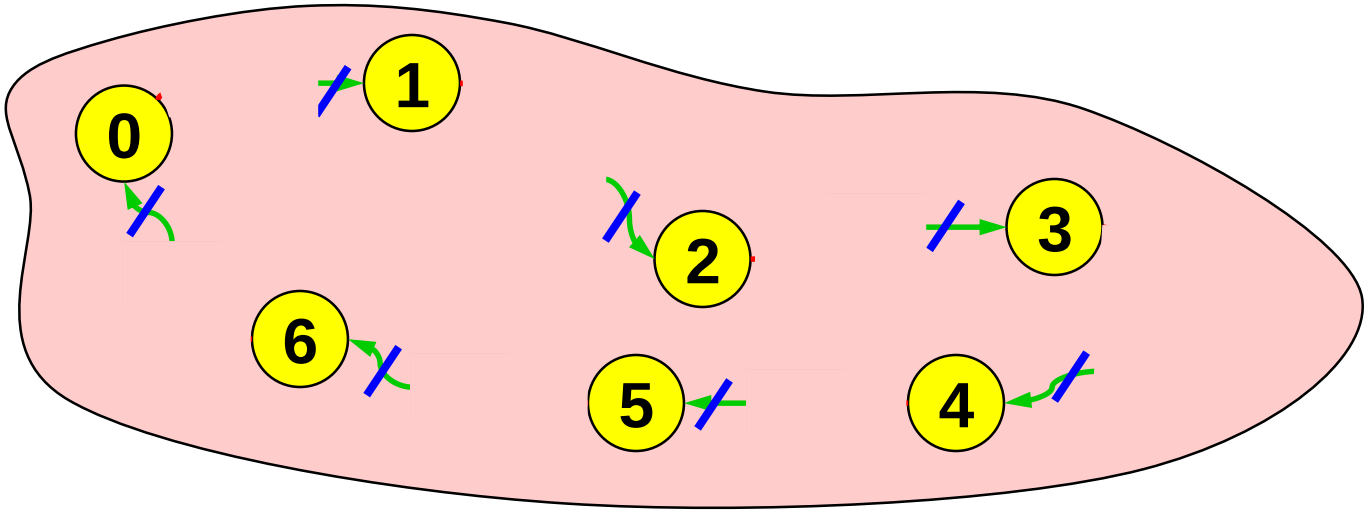
Non-Blocking Receive

- Initiate nonblocking **receive**
  - → in the ring example: Initiate nonblocking 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 nonblocking receive to complete

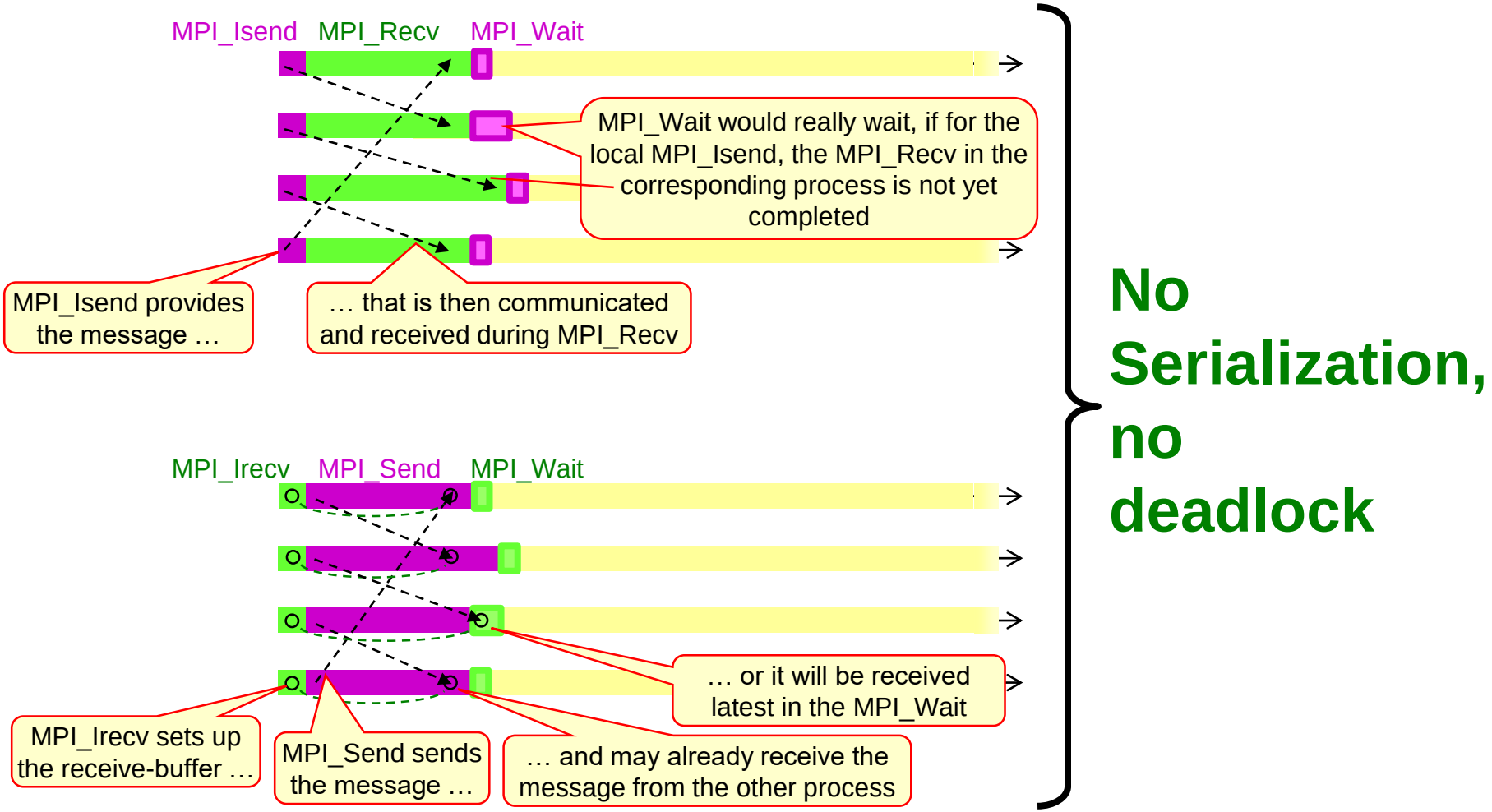


Non-Blocking Receive

- Initiate nonblocking **receive**
  - in the ring example: Initiate nonblocking 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 nonblocking receive to complete



Timelines for both solutions



Exercise 1 — Rotating information around a ring

- This exercise is a **preparation of many exercises** during the whole course.

Exercise 1 — Rotating information around a ring

- This exercise is a **preparation of many exercises** during the whole course.
- It is the **smallest example of halo communication**:
  - snd_buf = the real data that is to sent to the neighbor
  - rcv_buf = a halo
- Instead of huge double precision arrays, our data is just **1 integer**
- It is communicating in a ring.

Exercise 1 — Rotating information around a ring

- This exercise is a **preparation of many exercises** during the whole course.
- It is the **smallest example of halo communication**:
  - snd_buf = the real data that is to sent to the neighbor
  - rcv_buf = a halo
- Instead of huge double precision arrays, our data is just **1 integer**
- It is communicating in a ring.
- Our source code is **wrong**, because it uses MPI_Send and MPI_Recv in a naïve way.
- Wrong programs may work ☺, here because we send only a small message.
- But it is still wrong ☹.

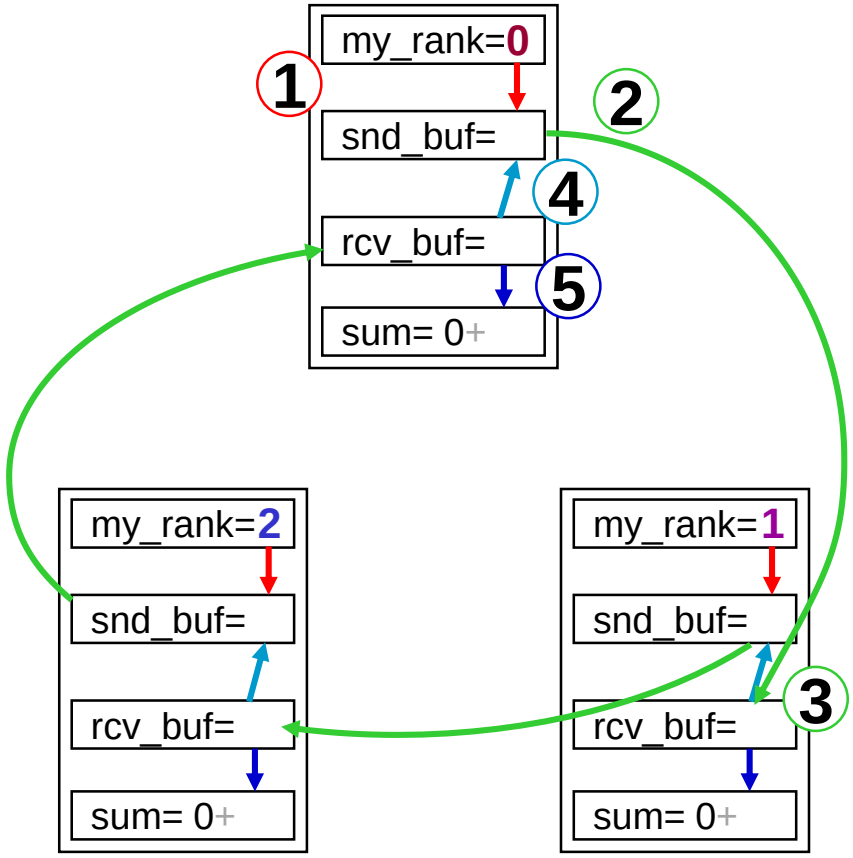
Exercise 1

Exercise 1 — Rotating information around a ring

- This exercise is a **preparation of many exercises** during the whole course.
- It is the **smallest example of halo communication**:
  - snd_buf = the real data that is to sent to the neighbor
  - rcv_buf = a halo
- Instead of huge double precision arrays, our data is just **1 integer**
- It is communicating in a ring.
- Our source code is **wrong**, because it uses MPI_Send and MPI_Recv in a naïve way.
- Wrong programs may work ☺, here because we send only a small message.
- But it is still wrong ☹.
- We start with C C/Ch4/ring-WRONG2.c or Fortran F_30/Ch4/ring-WRONG2_30.f90 or Python PY/Ch4/ring-WRONG2.py
- It uses two different buffers for send and receive.
- Just for fun, the program send the my_rank values around the ring in a loop with #process iterations and sums up all values coming along.

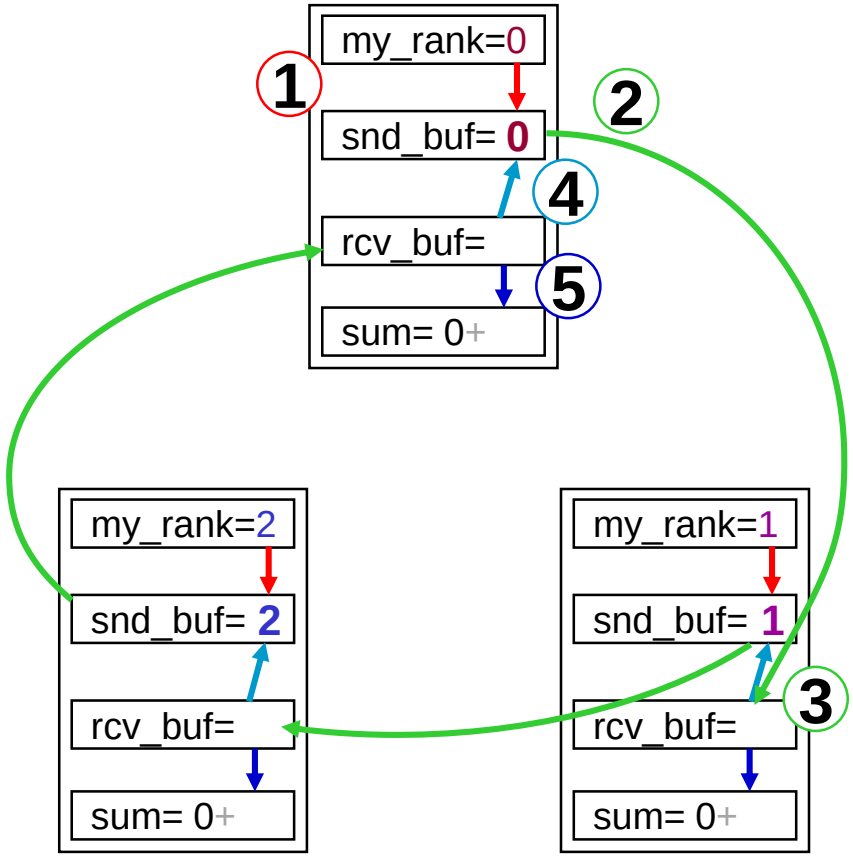
Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2)+(3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.
- ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
  - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol



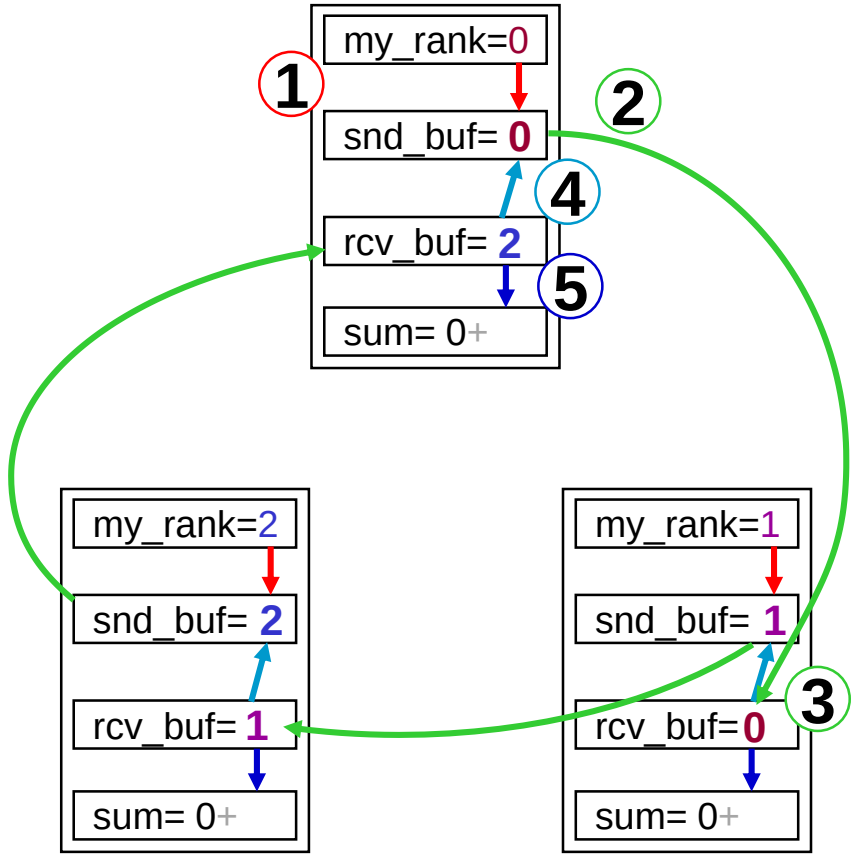
Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2)+(3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.
- ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
  - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol



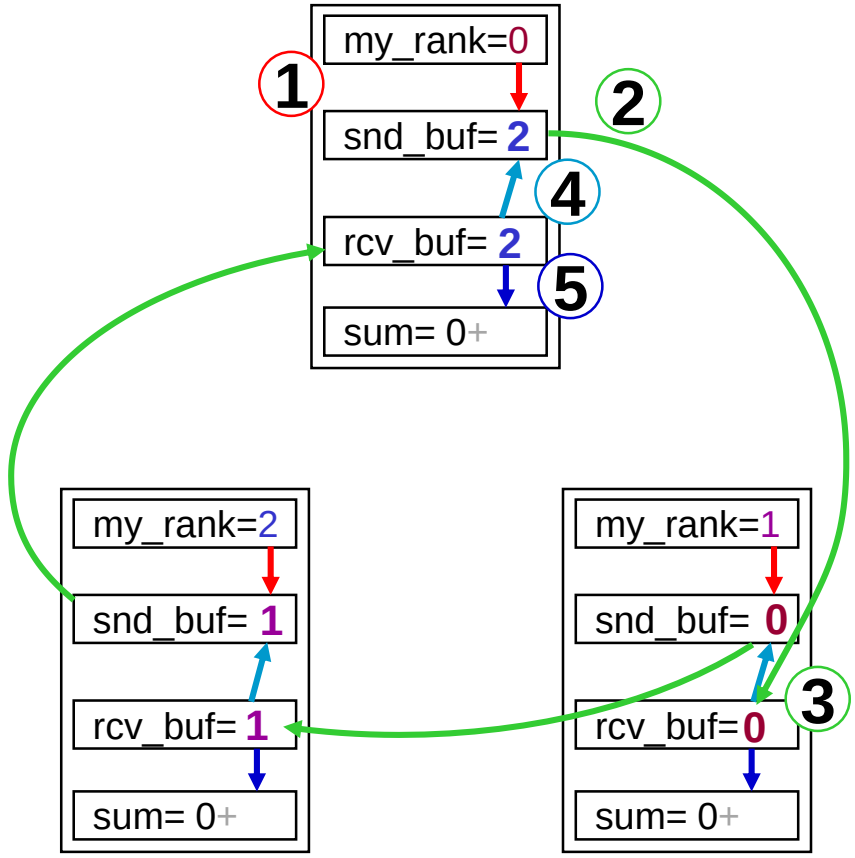
Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2)+(3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.
- ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
  - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol



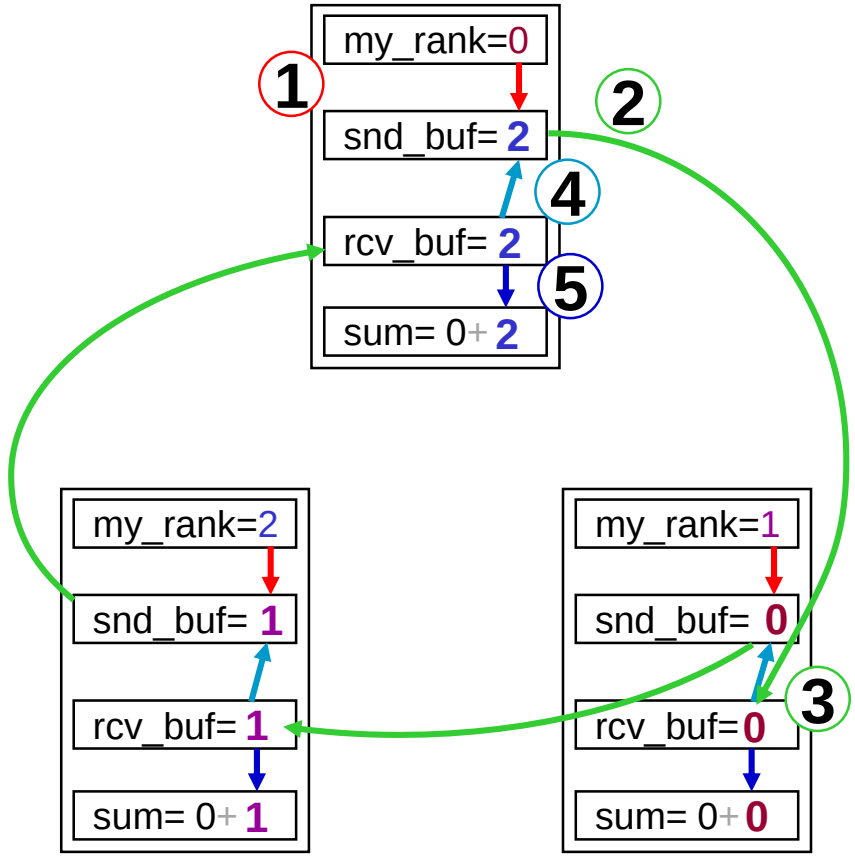
Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2)+(3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.
- ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
  - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol



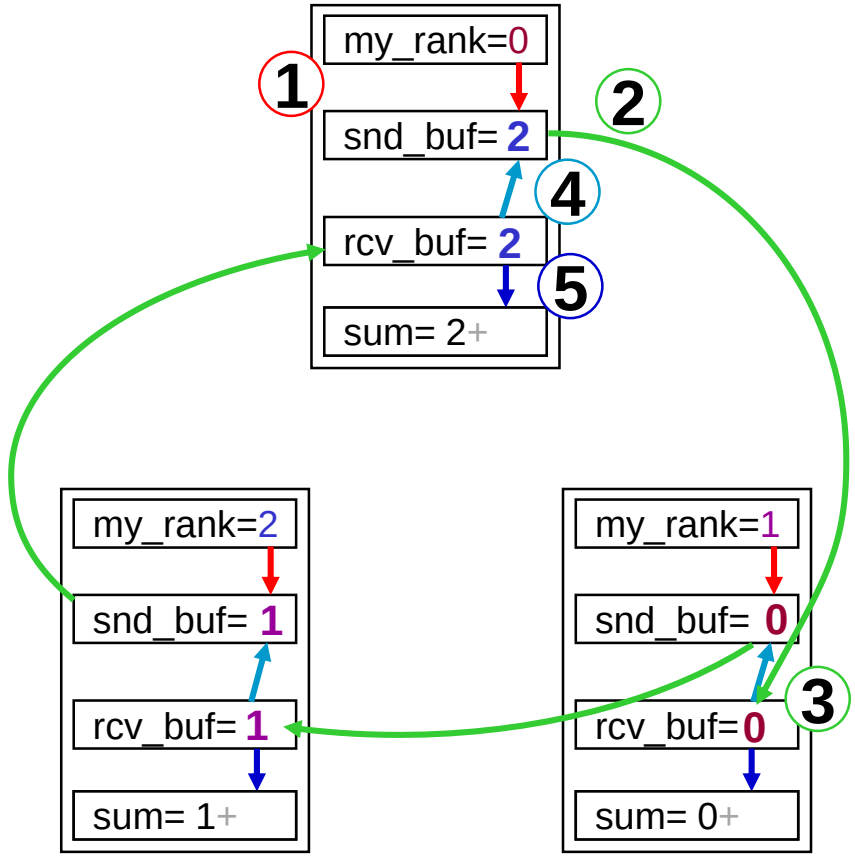
Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2)+(3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.
- ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
  - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol



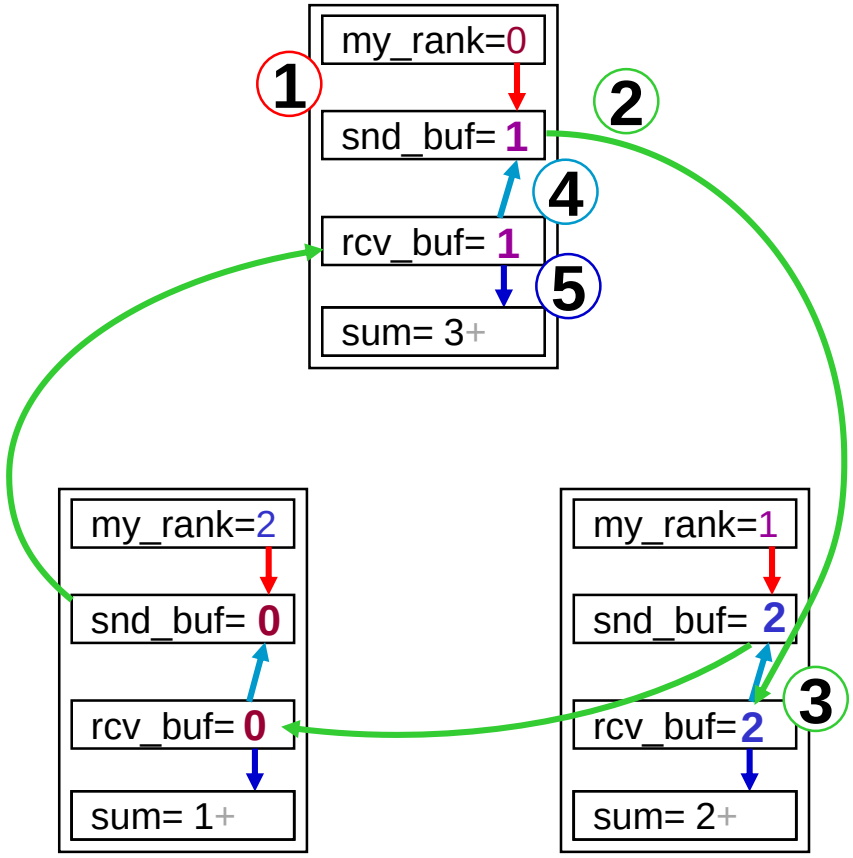
Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2)+(3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.
- ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
  - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol



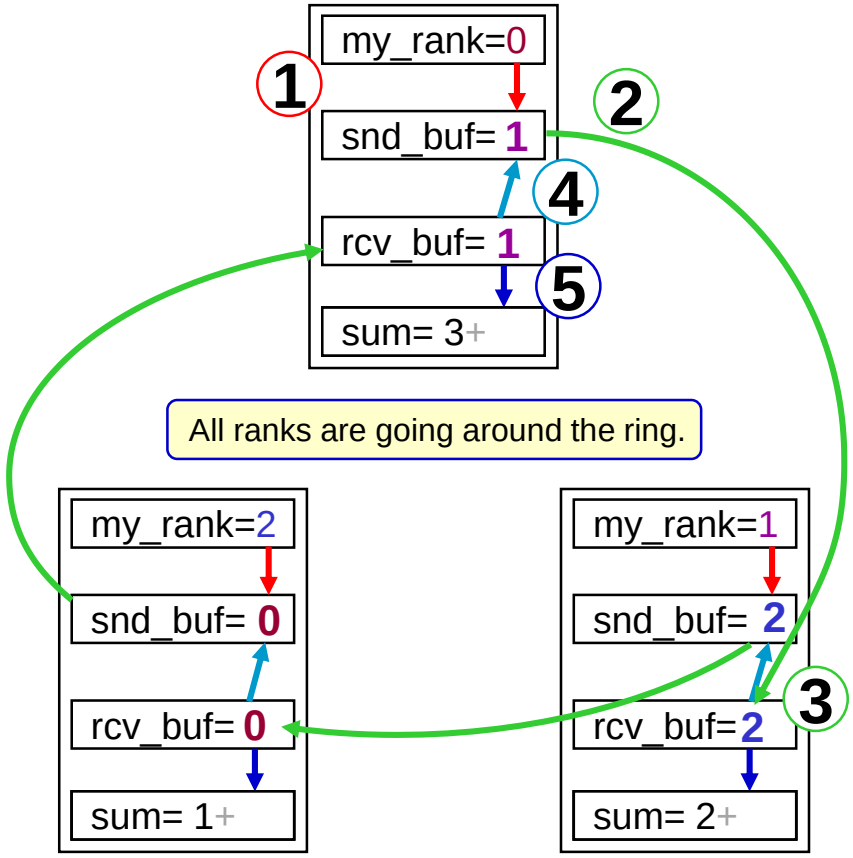
Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2)+(3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.
- ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
  - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol



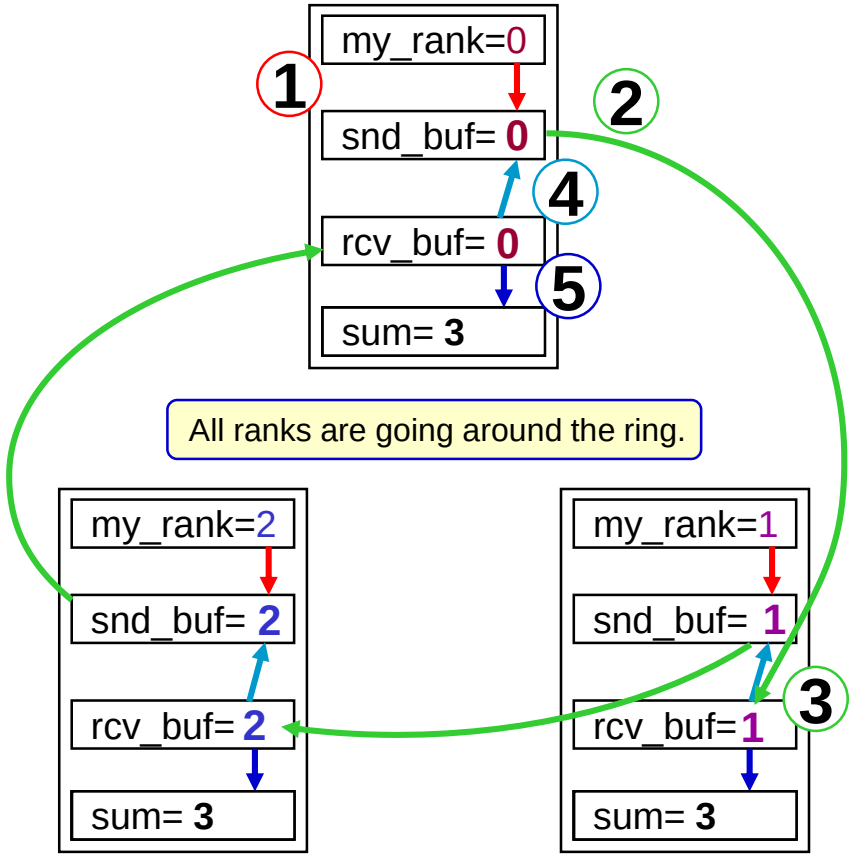
Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- 1 Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- 2 + 3 Each process passes this on to its neighbor on the right.
- 4 Preparation of next iteration.
- 5 Each process calculates the sum of all values.
- Repeat "2 - 5" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.
- ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
  - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol



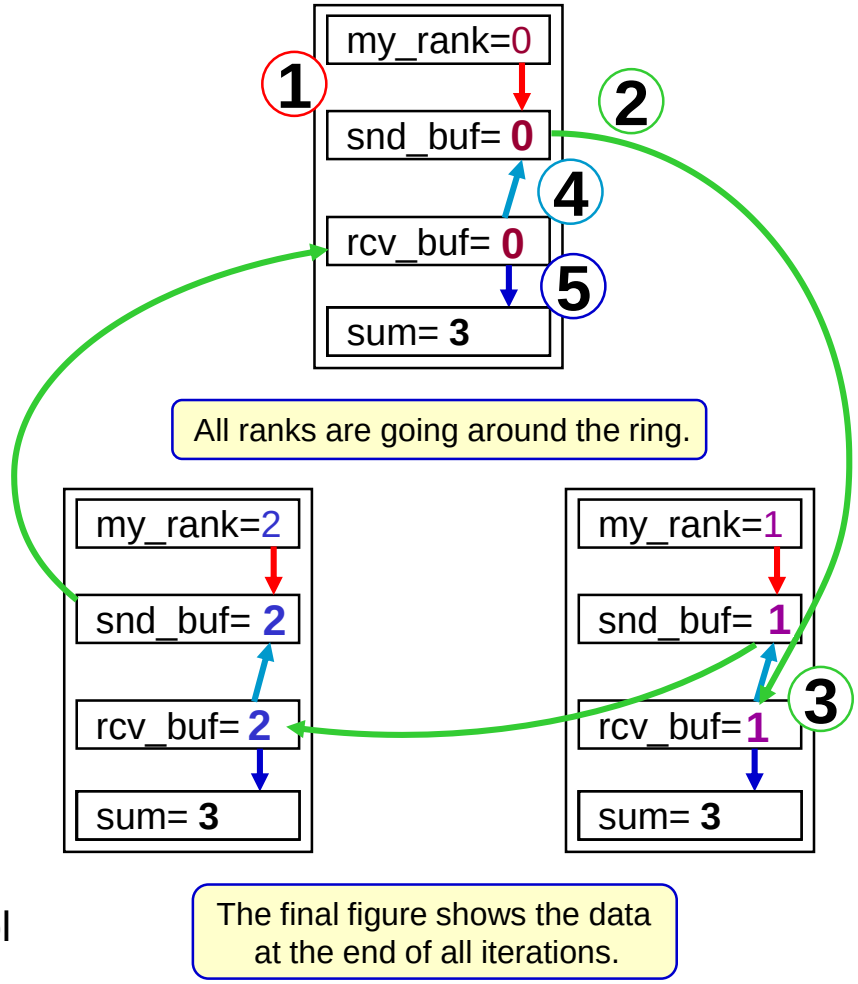
Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2)+(3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.
- ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
  - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol

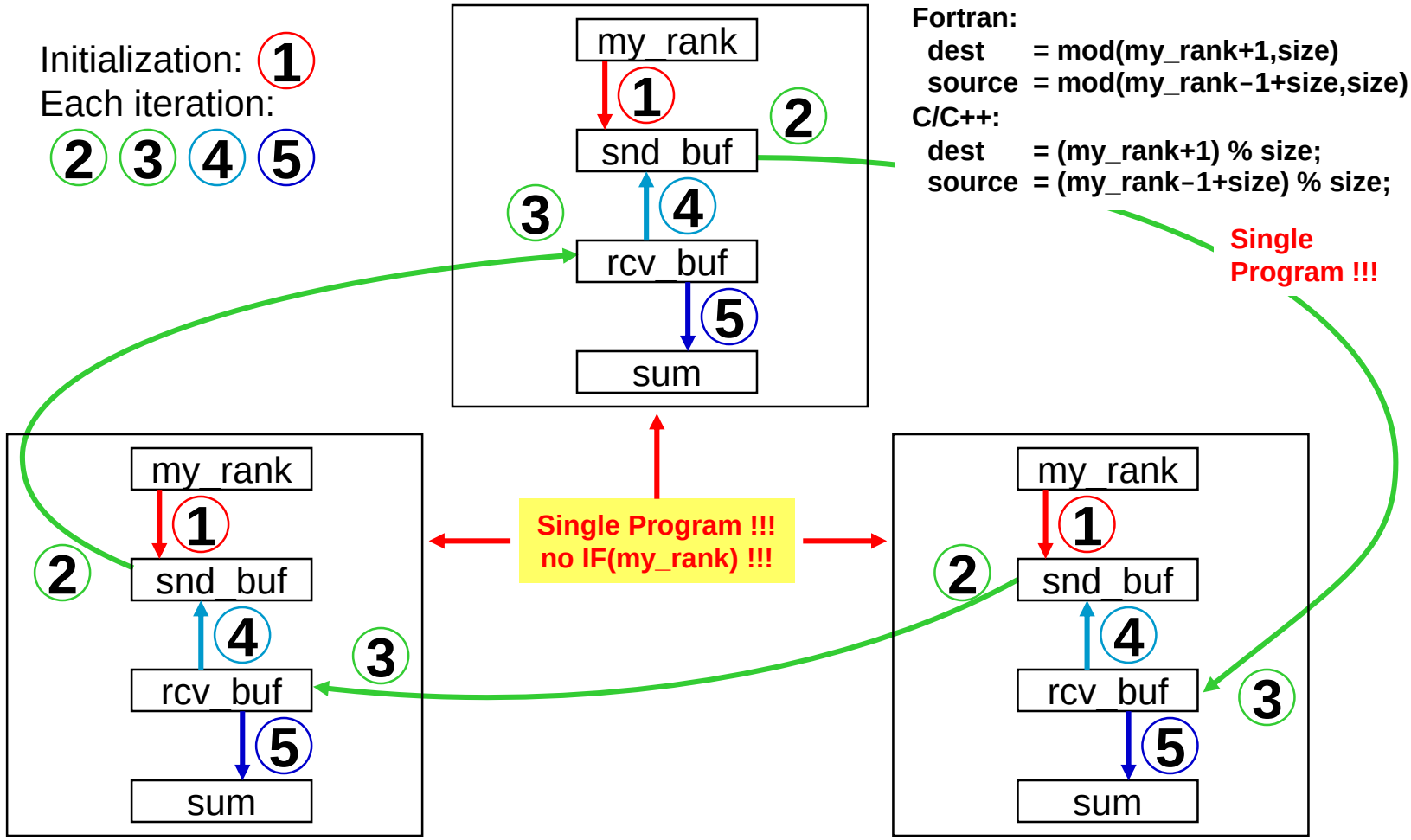


Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2)+(3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.
- ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
  - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol



Exercise 1 — Rotating information around a ring



Exercise 1 — Rotating information around a ring

- We start with a more simple version with only one common buffer
  C C/Ch4/ring-WRONG**1**.c or Fortran F_30/Ch4/ring-WRONG**1**_30.f90
  or Python PY/Ch4/ring-WRONG**1**.py

- Now substitute MPI_Send by MPI_Ssend → your **ring-WRONG1-Ss.c / _30.f90 / .py**
  - Expected result: Deadlock
  - Please try to run it and kill it (e.g., with Ctrl+c or Strg+c)

- Do same in version with 2 buffers
  - With 3 processes sum(all ranks) = 0+1+2 = 3,
    with 4 processes sum=6, with 5 processes sum=10, …
  - Please compile and run the program with 3, 4, 5 processes.
    C C/Ch4/ring-WRONG**2**.c or Fortran F_30/Ch4/ring-WRONG**2**_30.f90
    or Python PY/Ch4/ring-WRONG**2**.py
  - Substitute MPI_Send by MPI_Ssend → your **ring-WRONG2-Ss.c / _30.f90 / .py**
  - Keep the Ssend, but resolve deadlock through a serialization
    → your **ring-WRONG2-serialized.c / _30.f90 / .py**
    - **still wrong, not due to deadlock, but due to bad performance**
    - **Please compile and run the program with 3, 4, 5 processes: same results as in the first experiment.**

Handles, already known

- Predefined handles
  - defined in mpi.h / mpi_f08 / mpi & mpif.h
  - communicator, e.g., MPI_COMM_WORLD
  - datatype, e.g., MPI_INT, MPI_INTEGER, …
- Handles **can** also be stored in local variables

C / Fortran / Python

- memory for datatype handles
  - in C/C++: MPI_Datatype
  - in Fortran:
    - mpi_f08: TYPE(MPI_Datatype)
    - mpi & mpif.h: INTEGER
  - in Python: automatically, e.g., my_dtype = MPI.FLOAT

C / Fortran / Python

- memory for communicator handles
  - in C/C++: MPI_Comm
  - in Fortran:
    - mpi_f08: TYPE(MPI_Comm)
    - mpi & mpif.h: INTEGER
  - in Python: automatically, e.g., comm_world = MPI.COMM_WORLD

Request Handles

Request handles

- are used for nonblocking communication
- **must** be stored in local variables
  - in C/C++: MPI_Request
  - in Fortran:
    - mpi_f08: TYPE(MPI_Request)
    - mpi & mpif.h: INTEGER
  - in Python: automatically

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

Nonblocking Synchronous Send

C

- C/C++: MPI_Issend(&buf, count, datatype, dest, tag, comm,
  [OUT] &*request_handle*);

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

Fortran

- Fortran: ....., ASYNCHRONOUS :: buf

  New in MPI-3.0

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

  CALL MPI_WAIT([INOUT] request_handle, *status*, *ierror*)
  IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING)
  & CALL MPI_F_SYNC_REG(buf)

  New in MPI-3.0

Python

- Python: *request* = comm_world.Issend(…) / status = MPI.Status(); request.Wait(*status*)

Nonblocking Synchronous Send

C

- C/C++: MPI_Issend(&buf, count, datatype, dest, tag, comm,
  [OUT] &*request_handle*);
  ↓
  MPI_Wait([INOUT] &request_handle, &*status*);

Fortran

- Fortran: ….., ASYNCHRONOUS :: buf
  New in MPI-3.0
  CALL MPI_ISSEND(buf, count, datatype, dest, tag, comm,
  [OUT] *request_handle*, *ierror*)
  ↓
  CALL MPI_WAIT([INOUT] request_handle, *status*, *ierror*)
  IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING)
  New in MPI-3.0
  & CALL MPI_F_SYNC_REG(buf)

Python

- Python: *request* = comm_world.Issend(…) / status = MPI.Status(); request.Wait(*status*)

- buf must not be modified between Issend and Wait (in all progr. languages)
  (In MPI-2.1, this restriction was stronger: “should not access”, see MPI-2.1, page 52, lines 5-6)

Nonblocking Synchronous Send

C

- C/C++: MPI_Issend(&buf, count, datatype, dest, tag, comm,
  [OUT] &*request_handle*);
  ↓
  MPI_Wait([INOUT] &request_handle, &*status*);

Fortran

- Fortran: ….., ASYNCHRONOUS :: buf
  CALL MPI_ISSEND(buf, count, datatype, dest, tag, comm,
  [OUT] *request_handle*, *ierror*)
  ↓
  CALL MPI_WAIT([INOUT] request_handle, *status*, *ierror*)
  IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING)
  & CALL MPI_F_SYNC_REG(buf)

New in MPI-3.0

New in MPI-3.0

Python

- Python: *request* = comm_world.Issend(…) / status = MPI.Status(); request.Wait(*status*)

- buf must not be modified between Issend and Wait (in all progr. languages)
  (In MPI-2.1, this restriction was stronger: "should not access", see MPI-2.1, page 52, lines 5-6)
- "Issend + Wait directly after Issend" is equivalent to blocking call (Ssend)

Nonblocking Synchronous Send

C

- C/C++: MPI_Issend(&buf, count, datatype, dest, tag, comm,
  [OUT] &*request_handle*);
  ↓
  MPI_Wait([INOUT] &request_handle, &*status*);

Fortran

- Fortran: ….., ASYNCHRONOUS :: buf
  CALL MPI_ISSEND(buf, count, datatype, dest, tag, comm,
  [OUT] *request_handle*, *ierror*)
  ↓
  CALL MPI_WAIT([INOUT] request_handle, *status*, *ierror*)
  IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING)
  & CALL MPI_F_SYNC_REG(buf)

New in MPI-3.0

New in MPI-3.0

Python

- Python: *request* = comm_world.Issend(…) / status = MPI.Status(); request.Wait(*status*)

- buf must not be modified between Issend and Wait (in all progr. languages)
  (In MPI-2.1, this restriction was stronger: "should not access", see MPI-2.1, page 52, lines 5-6)
- "Issend + Wait directly after Issend" is equivalent to blocking call (Ssend)
- Nothing returned in status (because send operations have no status)

Nonblocking Synchronous Send

C

- C/C++: MPI_Issend(&buf, count, datatype, dest, tag, comm,
  [OUT] &*request_handle*);
  ↓
  MPI_Wait([INOUT] &request_handle, &*status*);

Fortran

- Fortran: ….., ASYNCHRONOUS :: buf
  CALL MPI_ISSEND(buf, count, datatype, dest, tag, comm,
  [OUT] *request_handle*, *ierror*)
  ↓
  CALL MPI_WAIT([INOUT] request_handle, *status*, *ierror*)
  IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING)
  & CALL MPI_F_SYNC_REG(buf)

New in MPI-3.0

New in MPI-3.0

Python

- Python: *request* = comm_world.Issend(…) / status = MPI.Status(); request.Wait(*status*)

- buf must not be modified between Issend and Wait (in all progr. languages)
  (In MPI-2.1, this restriction was stronger: "should not access", see MPI-2.1, page 52, lines 5-6)
- "Issend + Wait directly after Issend" is equivalent to blocking call (Ssend)
- Nothing returned in status (because send operations have no status)

Fortran

- Fortran problems, see MPI-3.1 / MPI-4.0, Chap. 17.1.10-19 / 19.1.10-19, pp 631-648 / 817-834, and slides at the end of this course chapter

Nonblocking Receive

C

- C/C++: MPI_Irecv (&*buf*, count, datatype, source, tag, comm,
  [OUT] &*request_handle*);
  ⬇
  MPI_Wait(**[INOUT]** &request_handle, &*status*);

Fortran

- Fortran: ….., ASYNCHRONOUS :: buf **New in MPI-3.0**
  CALL MPI_IRECV (*buf*, count, datatype, source, tag, comm,
  [OUT] *request_handle*, *ierror*)
  ⬇
  CALL MPI_WAIT(**[INOUT]** request_handle, *status*, *ierror*)
  IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) **New in MPI-3.0**
  & CALL MPI_F_SYNC_REG(buf)

Python

- Python: *request* = comm_world.Irecv(…) / status = MPI.Status(); request.Wait(*status*)

- buf must not be used between Irecv and Wait (in all progr. languages)
- Message status is returned in Wait

Fortran

- Fortran problems, see MPI-3.1 / MPI-4.0, Chap. 17.1.10-19 / 19.1.10-19, pp 631-648 / 817-834, and slides at the end of this course chapter

Blocking and Non-Blocking

- Send and receive can be blocking or nonblocking.
- A blocking send can be used with a nonblocking receive, and vice-versa.
- Nonblocking sends can use any mode
  - standard – MPI_ISEND
  - synchronous – MPI_ISSEND
  - buffered – MPI_IBSEND
  - ready – MPI_IRSEND
- Synchronous mode affects completion, i.e. MPI_Wait / MPI_Test, not initiation, i.e., MPI_I….
- The nonblocking operation immediately followed by a matching wait is equivalent to the blocking operation, except the Fortran problems.

Completion

C

- C/C++:

  or MPI_STATUS_IGNORE

  MPI_Wait(&request_handle, &*status*);

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

Fortran

- Fortran:

  or MPI_STATUS_IGNORE

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

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

Python

- Python:

  or None

  status = MPI.Status(); request_handle.Wait(*status*)

  status = MPI.Status(); *flag* = request_handle.Test(*status*)

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

C Fortran Python

Multiple Non-Blocking Communications

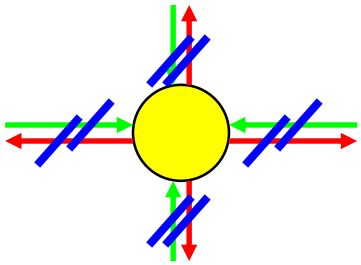
You have several request handles:

- Wait or test for completion of **one** message
  - **MPI_Waitany / MPI_Testany**
- Wait or test for completion of **all** messages
  - **MPI_Waitall / MPI_Testall *)**
- Wait or test for completion of **at least one** messages
  - **MPI_Waitsome / MPI_Testsome *)**
- Argument lists: MPI_Wait…(count, array_of_requests, array_of_statuses)
  MPI_Test…(count, array_of_requests, flag, array_of_statuses)
  in Python: MPI.Request.Wait…(array_of_requests, array_of_statuses)
  flag = MPI.Request.Test…(array_of_requests, array_of_statuses)

C: int
Fortran: LOGICAL

or MPI_STATUSES_IGNORE

or statuses=None



***)** Each status contains an additional error field.
This field is only used if MPI_ERR_IN_STATUS is returned (also valid for send operations).

Other MPI features: Send-Receive in one routine

- MPI_Sendrecv & MPI_Sendrecv_replace
  - Combines the triple "MPI_Irecv + Send + Wait" into one routine
  - See advanced exercise at the end of course Chapter 12-(1) *Derived Datatypes*

New in MPI-4.0

- Nonblocking MPI_Isendrecv & MPI_Isendrecv_replace
  - Whereas blocking MPI_Sendrecv was used to prevent
    - **serializations and**
    - **deadlocks,**
  - the nonblocking MPI_Isendrecv can be used, e.g., to parallelize the existing communication calls in multiple directions → e.g., to minimize idle times if only some neighbors are delayed

Performance options

Which is the fastest neighbor communication?

- MPI_Irecv + MPI_Send
- MPI_Irecv + MPI_Isend
- MPI_Isend + MPI_Recv
- MPI_Isend + MPI_Irecv
- MPI_Sendrecv
- Several MPI_Isendrecv together
- MPI_Neighbor_alltoall → see course Chapter 9-(2) *Virtual Topologies*

Caution: In the exercise, we use the *synchronous* MPI_I**s**send() only to demonstrate a deadlock if the nonblocking routine is not correctly used.

A real application would use *standard* **Isend() !!!**
Never *synchronous* I**s**send() **!!!**

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming
but
efficiency of MPI application-programming is not portable!

Use cases for nonblocking operations

- To prevent serializations and deadlocks
  (as if overlapping of communication with other communication)
  - Now also described in the intro of MPI-4.0 Section 3.7 Nonblocking Communication

New in MPI-4.0

- Real overlapping of
  - several communications
  - communication and computation

Exercise 2 — Rotating information around a ring

Exercises 2+3

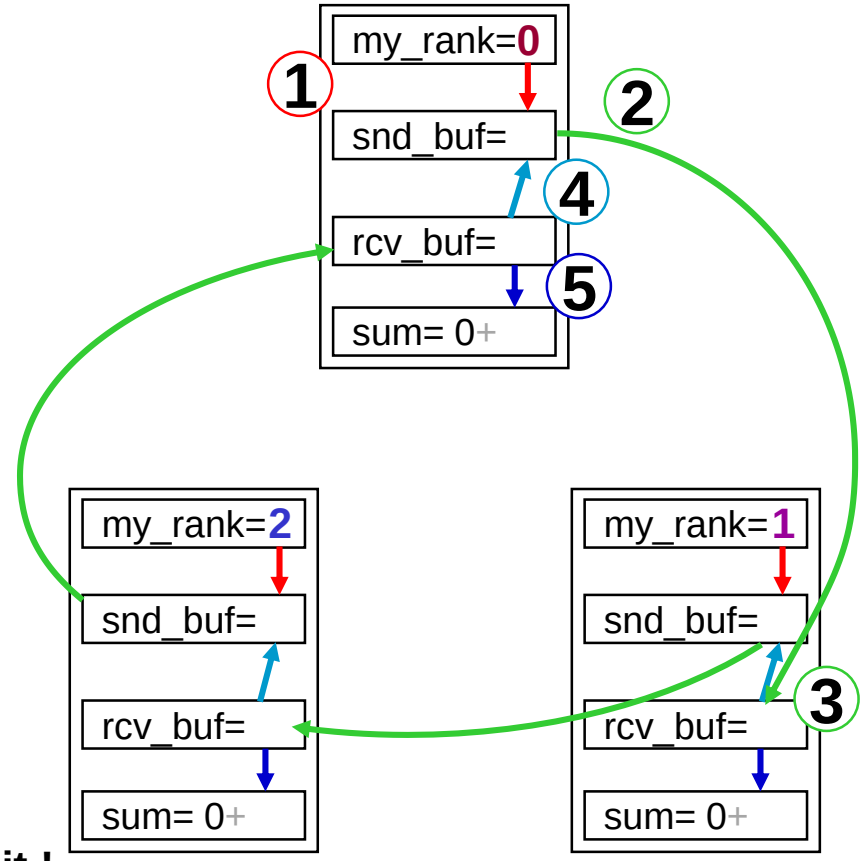
- A set of processes are arranged in a ring.
- 1 Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- 2 + 3 Each process passes this on to its neighbor on the right.
- 4 Preparation of next iteration.
- 5 Each process calculates the sum of all values.
- Repeat "2 - 5" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

In MPI/tasks/…

C Use C/Ch4/ring-skel.c

Fortran or F_30/Ch4/ring-skel_30.f90

Python or PY/Ch4/ring-skel.py



- **Use nonblocking MPI_Issend + MPI_Wait !**
  - **to avoid deadlocks**
  - **to verify the correctness,** because blocking synchronous send will cause a deadlock
- **Keep normal blocking MPI_Recv !**

Exercise 2 — Rotating information around a ring

Exercises 2+3

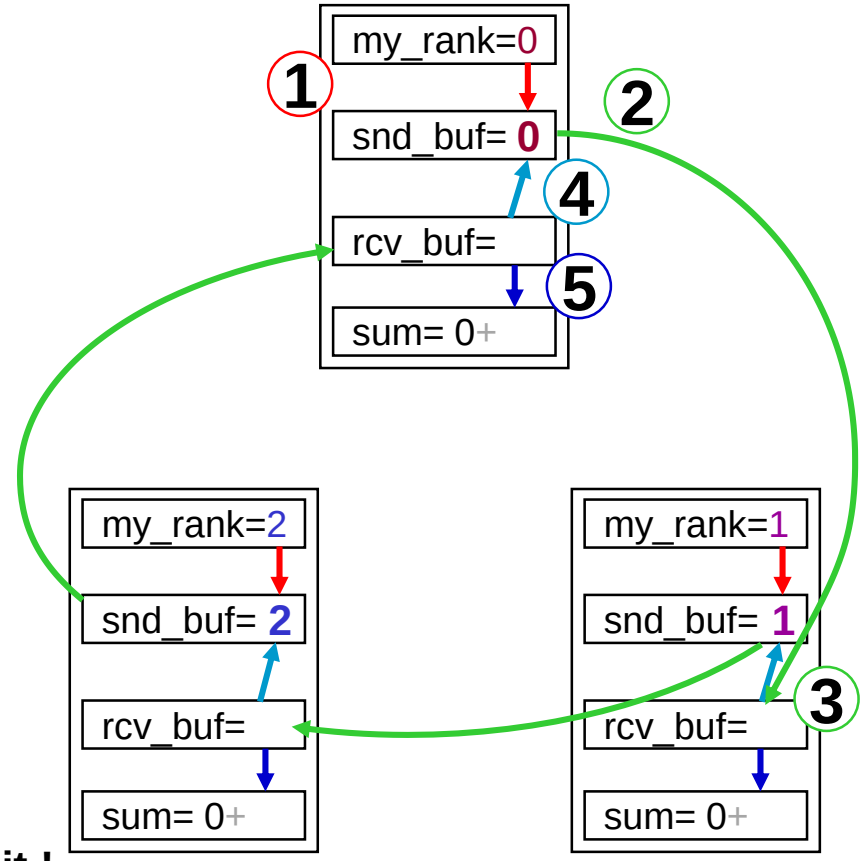
- A set of processes are arranged in a ring.
- 1 Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- 2 + 3 Each process passes this on to its neighbor on the right.
- 4 Preparation of next iteration.
- 5 Each process calculates the sum of all values.
- Repeat "2 - 5" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

In MPI/tasks/…

C Use C/Ch4/ring-skel.c

Fortran or F_30/Ch4/ring-skel_30.f90

Python or PY/Ch4/ring-skel.py



- **Use nonblocking MPI_Issend + MPI_Wait !**
  - **to avoid deadlocks**
  - **to verify the correctness,** because blocking synchronous send will cause a deadlock
- **Keep normal blocking MPI_Recv !**

Exercise 2 — Rotating information around a ring

Exercises 2+3

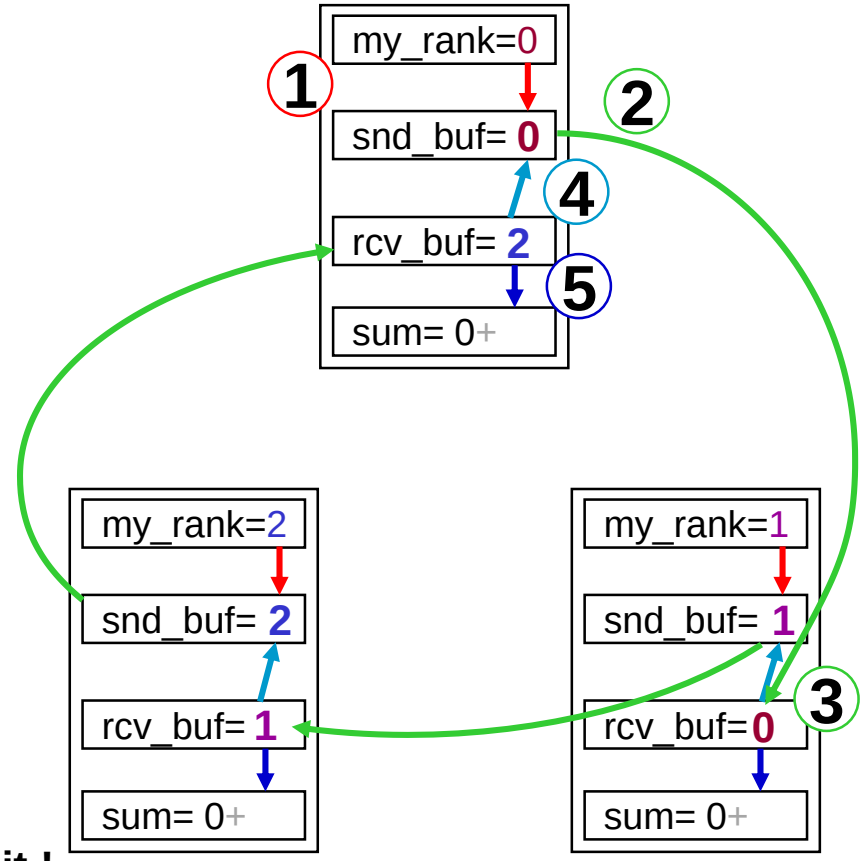
- A set of processes are arranged in a ring.
- 1 Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- 2 + 3 Each process passes this on to its neighbor on the right.
- 4 Preparation of next iteration.
- 5 Each process calculates the sum of all values.
- Repeat "2 - 5" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

In MPI/tasks/…

C Use C/Ch4/ring-skel.c

Fortran or F_30/Ch4/ring-skel_30.f90

Python or PY/Ch4/ring-skel.py



- **Use nonblocking MPI_Issend + MPI_Wait !**
  - **to avoid deadlocks**
  - **to verify the correctness,** because blocking synchronous send will cause a deadlock
- **Keep normal blocking MPI_Recv !**

Exercise 2 — Rotating information around a ring

Exercises 2+3

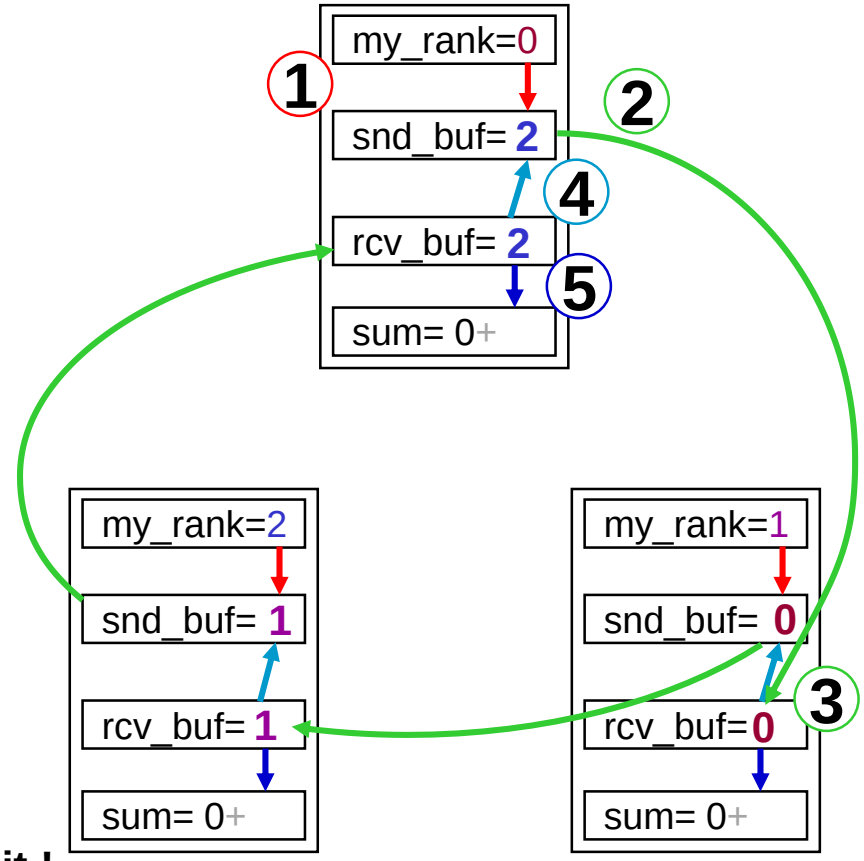
- A set of processes are arranged in a ring.
- 1 Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- 2 + 3 Each process passes this on to its neighbor on the right.
- 4 Preparation of next iteration.
- 5 Each process calculates the sum of all values.
- Repeat "2 - 5" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

In MPI/tasks/…

C Use C/Ch4/ring-skel.c

Fortran or F_30/Ch4/ring-skel_30.f90

Python or PY/Ch4/ring-skel.py



- **Use nonblocking MPI_Issend + MPI_Wait !**
  - **to avoid deadlocks**
  - **to verify the correctness,** because blocking synchronous send will cause a deadlock
- **Keep normal blocking MPI_Recv !**

Exercise 2 — Rotating information around a ring

Exercises 2+3

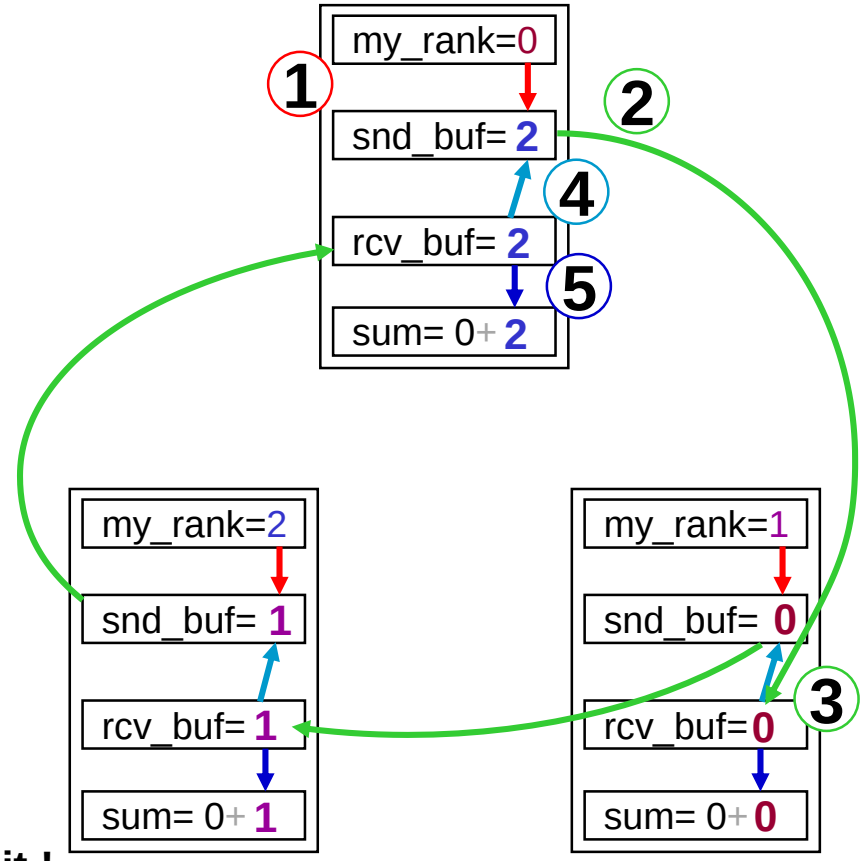
- A set of processes are arranged in a ring.
- 1 Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- 2 + 3 Each process passes this on to its neighbor on the right.
- 4 Preparation of next iteration.
- 5 Each process calculates the sum of all values.
- Repeat "2 - 5" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

In MPI/tasks/…

C Use C/Ch4/ring-skel.c

Fortran or F_30/Ch4/ring-skel_30.f90

Python or PY/Ch4/ring-skel.py



- **Use nonblocking MPI_Issend + MPI_Wait !**
  - **to avoid deadlocks**
  - **to verify the correctness,** because blocking synchronous send will cause a deadlock
- **Keep normal blocking MPI_Recv !**

Exercise 2 — Rotating information around a ring

Exercises 2+3

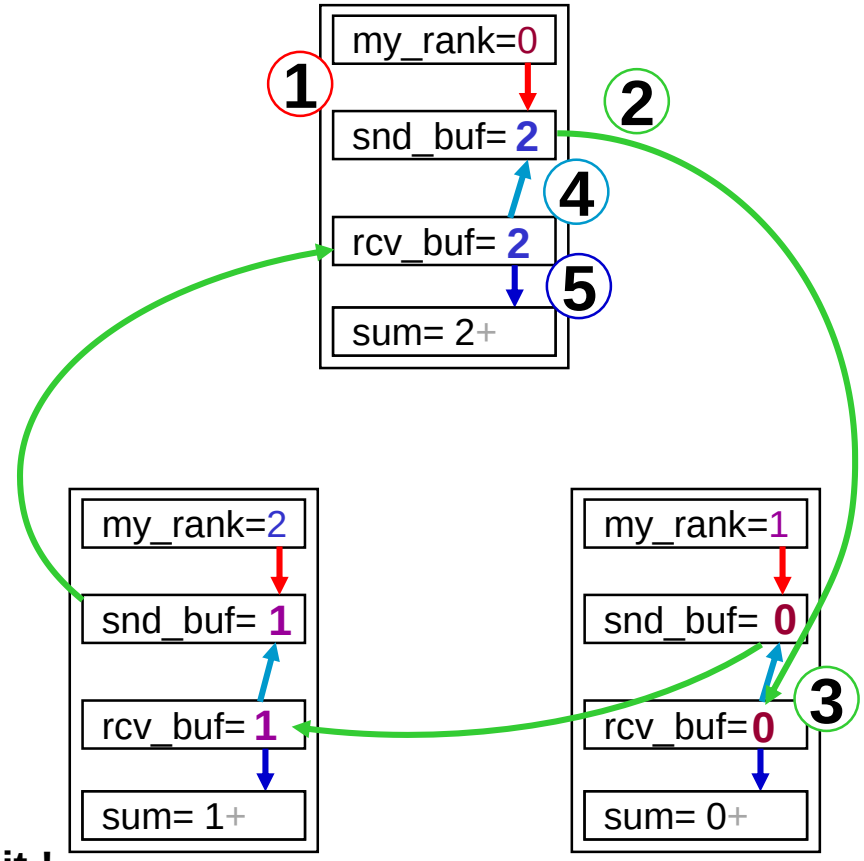
- A set of processes are arranged in a ring.
- 1 Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- 2 + 3 Each process passes this on to its neighbor on the right.
- 4 Preparation of next iteration.
- 5 Each process calculates the sum of all values.
- Repeat "2 - 5" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

In MPI/tasks/…

C Use C/Ch4/ring-skel.c

Fortran or F_30/Ch4/ring-skel_30.f90

Python or PY/Ch4/ring-skel.py



- **Use nonblocking MPI_Issend + MPI_Wait !**
  - **to avoid deadlocks**
  - **to verify the correctness,** because blocking synchronous send will cause a deadlock
- **Keep normal blocking MPI_Recv !**

Exercise 2 — Rotating information around a ring

Exercises 2+3

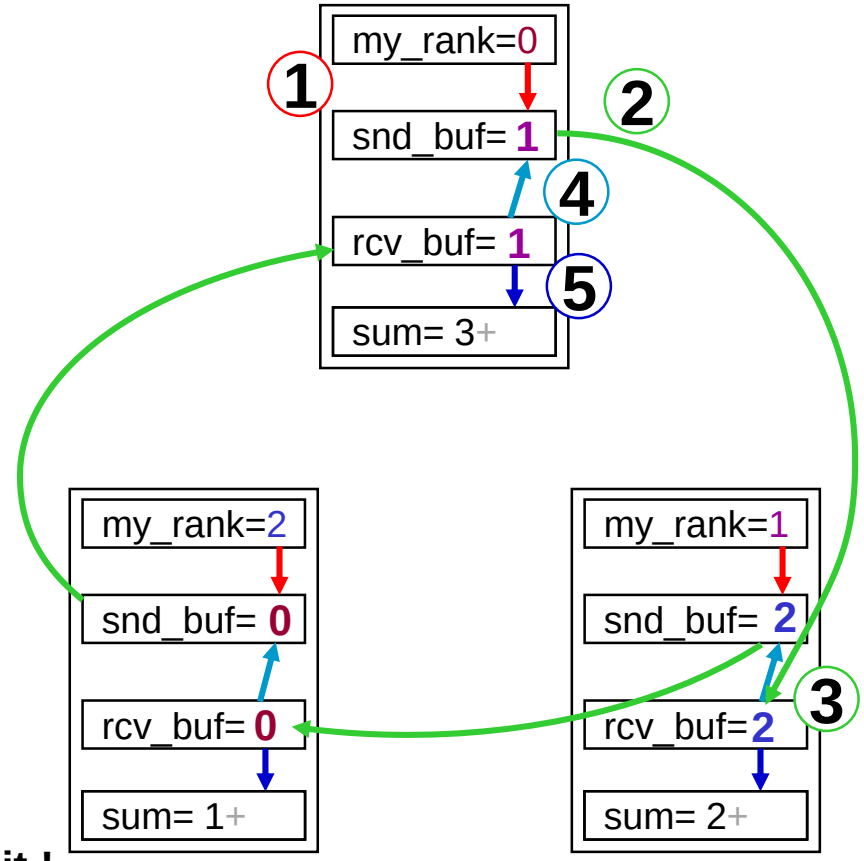
- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2)+(3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

In MPI/tasks/…

C Use C/Ch4/ring-skel.c

Fortran or F_30/Ch4/ring-skel_30.f90

Python or PY/Ch4/ring-skel.py



- **Use nonblocking MPI_Issend + MPI_Wait !**
  - **to avoid deadlocks**
  - **to verify the correctness,** because blocking synchronous send will cause a deadlock
- **Keep normal blocking MPI_Recv !**

Exercise 2 — Rotating information around a ring

Exercises 2+3

- A set of processes are arranged in a ring.
- (1) Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- (2) + (3) Each process passes this on to its neighbor on the right.
- (4) Preparation of next iteration.
- (5) Each process calculates the sum of all values.
- Repeat "(2)-(5)" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

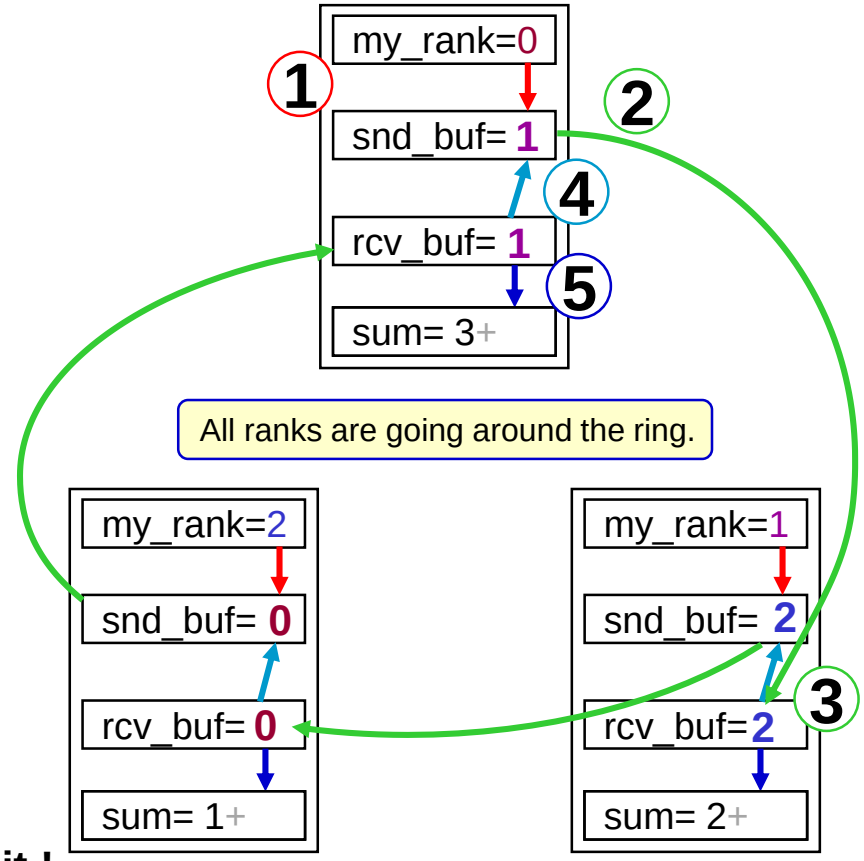
In MPI/tasks/…

C Use C/Ch4/ring-skel.c

Fortran or F_30/Ch4/ring-skel_30.f90

Python or PY/Ch4/ring-skel.py

- **Use nonblocking MPI_Issend + MPI_Wait !**
  - **to avoid deadlocks**
  - **to verify the correctness,** because blocking synchronous send will cause a deadlock
- **Keep normal blocking MPI_Recv !**



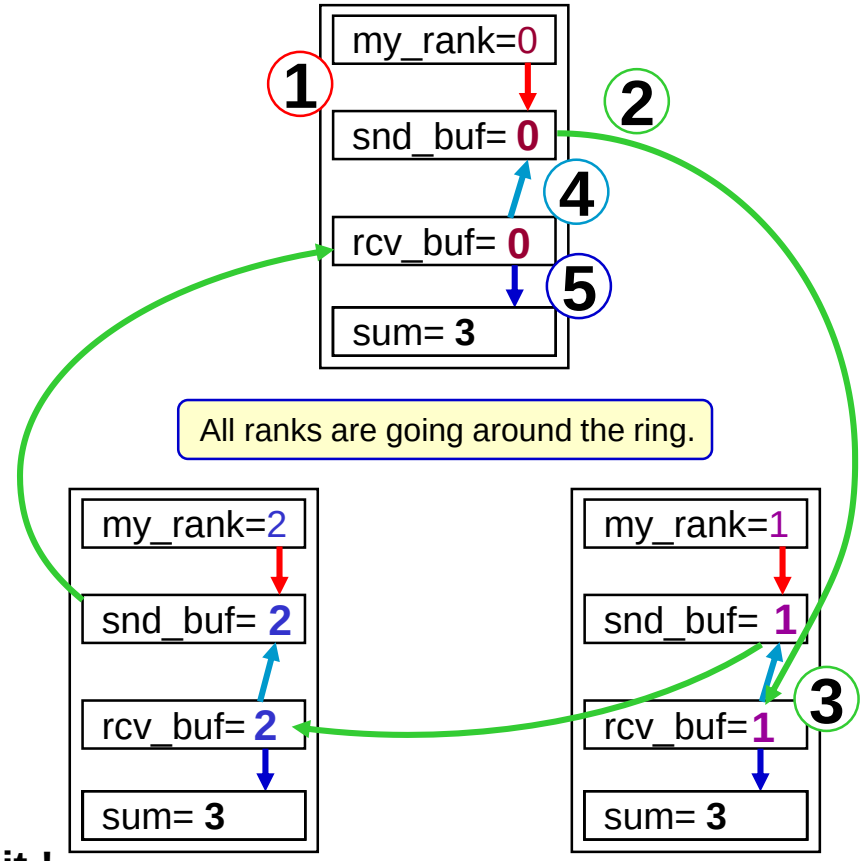
Exercise 2 — Rotating information around a ring

Exercises 2+3

- A set of processes are arranged in a ring.
- 1 Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- 2 + 3 Each process passes this on to its neighbor on the right.
- 4 Preparation of next iteration.
- 5 Each process calculates the sum of all values.
- Repeat "2 - 5" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

In MPI/tasks/…

C Use C/Ch4/ring-skel.c
Fortran or F_30/Ch4/ring-skel_30.f90
Python or PY/Ch4/ring-skel.py



- **Use nonblocking MPI_Issend + MPI_Wait !**
  - **to avoid deadlocks**
  - **to verify the correctness,** because blocking synchronous send will cause a deadlock
- **Keep normal blocking MPI_Recv !**

Exercise 2 — Rotating information around a ring

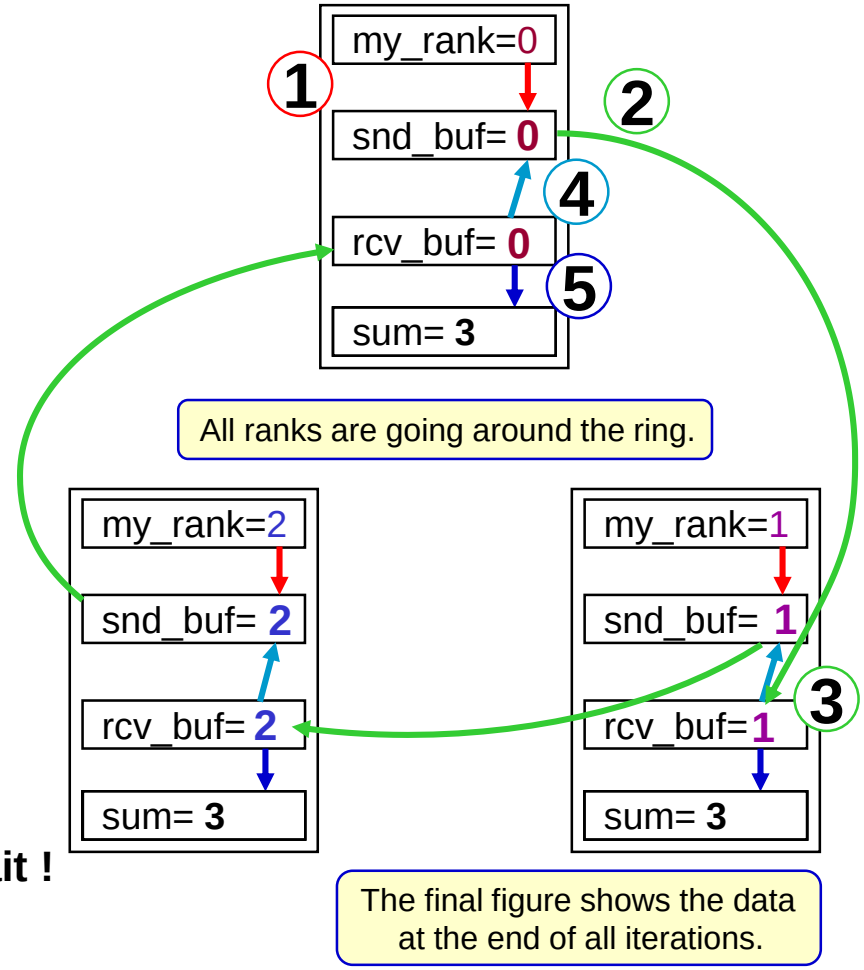
Exercises 2+3

- A set of processes are arranged in a ring.
- 1 Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- 2 + 3 Each process passes this on to its neighbor on the right.
- 4 Preparation of next iteration.
- 5 Each process calculates the sum of all values.
- Repeat "2 - 5" with "size" iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

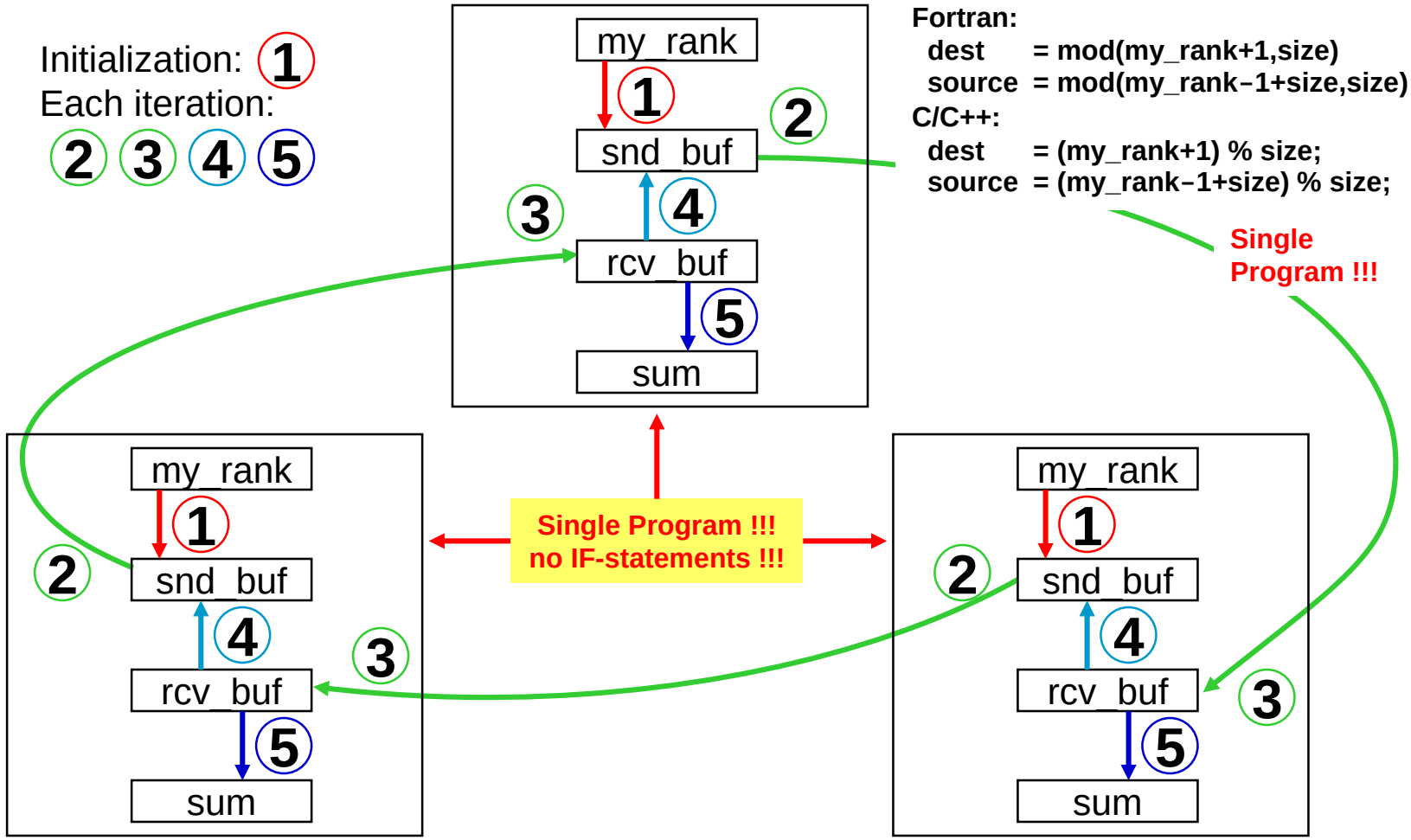
In MPI/tasks/…

C Use C/Ch4/ring-skel.c
Fortran or F_30/Ch4/ring-skel_30.f90
Python or PY/Ch4/ring-skel.py

- **Use nonblocking MPI_Issend + MPI_Wait !**
  - **to avoid deadlocks**
  - **to verify the correctness,** because blocking synchronous send will cause a deadlock
- **Keep normal blocking MPI_Recv !**



Exercise 2 — Rotating information around a ring



Fortran: Do not forget MPI-3.0 → **…, ASYNCHRONOUS :: …_buf** and **IF(.NOT.MPI_...) CALL MPI_F_SYNC_REG(…)**

Exercises 3 (advanced) — Irecv instead of Issend

- Substitute the Issend–Recv–Wait method by the Irecv–Ssend–Wait method in your ring program.
  - Solution: MPI/tasks/C/Ch4/solutions/ring_advanced_irecv_ssend.c
    and MPI/tasks/F_30/Ch4/solutions/ring_advanced_irecv_ssend_30.f90

- Or

- Substitute the Issend–Recv–Wait method by the Irecv–Issend–Waitall method in your ring program.

Quiz A+B on Chapter 4 – Nonblocking communication

A. MPI nonblocking communication: Which are the three major use-cases and please sort from most important to less important:

1. _____
2. _____
3. _____

B. MPI/participant-test/ring-test.c or MPI/participant-test/ring-test_30.f90

Is this **loop body** correctly programmed with MPI

– Yes / No

For the case that you answered with "No",

– which bug(s) do you see?
– And how could it/they be resolved?
– Which correction(s) would you apply (still using nonblocking communication)?

C

```
MPI_Isend(&buffer,1,MPI_INT,right,
     17, MPI_COMM_WORLD, &request);
MPI_Recv (&buffer,1,MPI_INT, left,
     17, MPI_COMM_WORLD, &status);
sum += buffer;
```

Full program on next slide

Fortran

```
CALL MPI_Isend(buffer,1,MPI_INTEGER,&
 &   right,17,MPI_COMM_WORLD,request)
CALL MPI_Recv (buffer,1,MPI_INTEGER,&
 &   left, 17,MPI_COMM_WORLD,status)
sum = sum + buffer
```

Python

```
request=comm_world.Issend((buffer,1,MPI.INT),
                          dest=right, tag=17)
comm_world.Recv((buffer,1,MPI.INT),
           source=left, tag=17, status=status)
sum += buffer
```

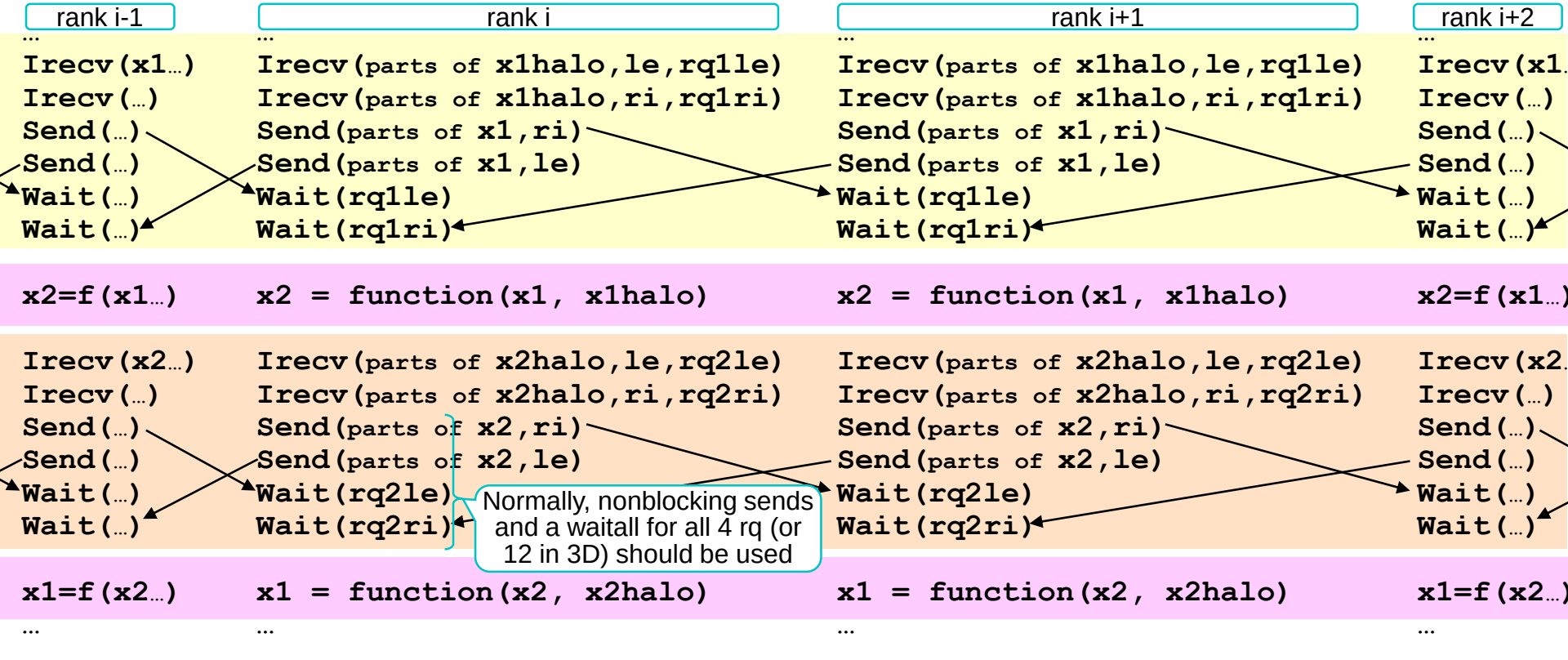
Quiz C+D on Chapter 4 – Nonblocking communication

C. Which **point-to-point** communication pattern would you prefer for the halo communication of a 3-dimensional time step integration application?
   - _____
   - _____
   - _____

D. Which send methods are only for very special cases?
   - _____ / for _____ / disadvantage _____ / prefer instead _____
   - _____ / for _____ / disadvantage _____ / prefer instead _____
   - _____ / for _____ / disadvantage _____ / prefer instead _____

Quiz E on Chapter 4 – Double buffering and MPI_Rsend

E. Let's have a look at several iteration of our halo exchange and x_new = function(x_old):
   - We use here x1 and x2 instead of x_new and x_old and show two iterations
   - The halos are explicitly named with x1halo and x2halo, and le=left, ri=right



Who can see, what must we change that we can use **MPI_Rsend** instead of MPI_Send?