SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Introduction to the Message Passing Interface (MPI) (basics)

Claudia Blaas-Schenner

VSC Research Center, TU Wien

VIENNA SCIENTIFIC CLUSTER

01/2022

Univerza v Ljubljani

TECHNISCHE UNIVERSITÄT WIEN

CINECA

VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Co-funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

who is this speaker ?

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Claudia Blaas-Schenner

- affiliated at the **VSC Research Center of TU Wien, Austria** (since 2014)
- responsible for **skills development & training and education in HPC**

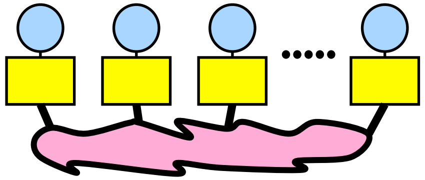
- background in physics (TU Wien, Uni Vienna, TU Dresden, ASC Prague)
- specialized in **computational** materials science (PhD from TU Wien 1996)
- wrote my **first parallel program** in a summer school in 1991 (with PVM)

- active **member of the MPI forum** (= standardization body for MPI the Message Passing Interface)
- chapter chair for **MPI Terms and Conventions** that is essential for the MPI standard as a whole

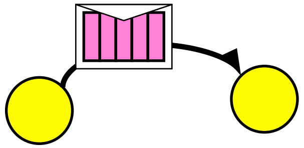
- **main interests** in efficient use od HPC systems, performance optimization, and performance portability of parallel codes
- claudia.blaas-schenner@tuwien.ac.at

MPI basics – agenda

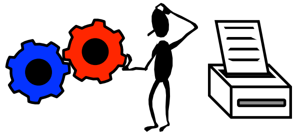
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



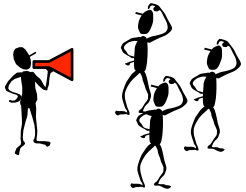
- **messages and point-to-point communication**
  - the MPI processes can communicate



- **nonblocking communication** → MPI & **Fortran**
  - to avoid idle times, serializations, and deadlocks



- **collective communication**
  - e.g. broadcast, reduction, …



- **MPI basics – summary**

goals and **scope** of MPI

- message-passing interface
- source-code portability
- allow efficient implementations
- a great deal of functionality

current version (June 9, 2021)

MPI-4.0

available libraries are for MPI-3.1

info & acknowledgements

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

These **slides** are a modified subset of the MPI course developed by

Rolf Rabenseifner, High-Performance Computing Center Stuttgart (HLRS).

Also the **hands-on labs** are developed by **Rolf Rabenseifner**, HLRS, and can be downloaded from the HLRS website:
https://fs.hlrs.de/projects/par/par_prog_ws/practical/MPI31single.tar.gz

https://fs.hlrs.de/projects/par/par_prog_ws/practical/MPI31single.zip

!

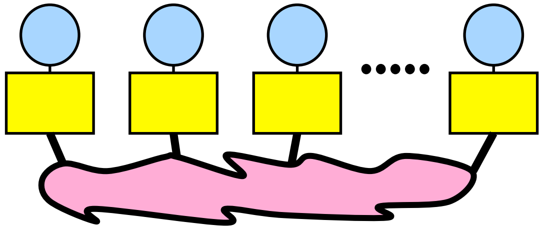
The **MPI standard document** (MPI 4.0, June 9, 2021) is available from the MPI forum:

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf → available libraries for **MPI-3.1**

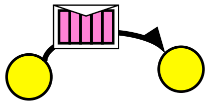
python (not part of the MPI standard): https://mpi4py.readthedocs.io/

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

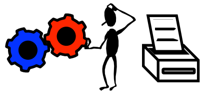
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



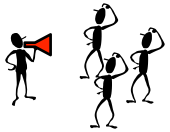
- **messages and point-to-point communication**
  - the MPI processes can communicate



- **nonblocking communication**
  - to avoid idle times, serializations, and deadlocks



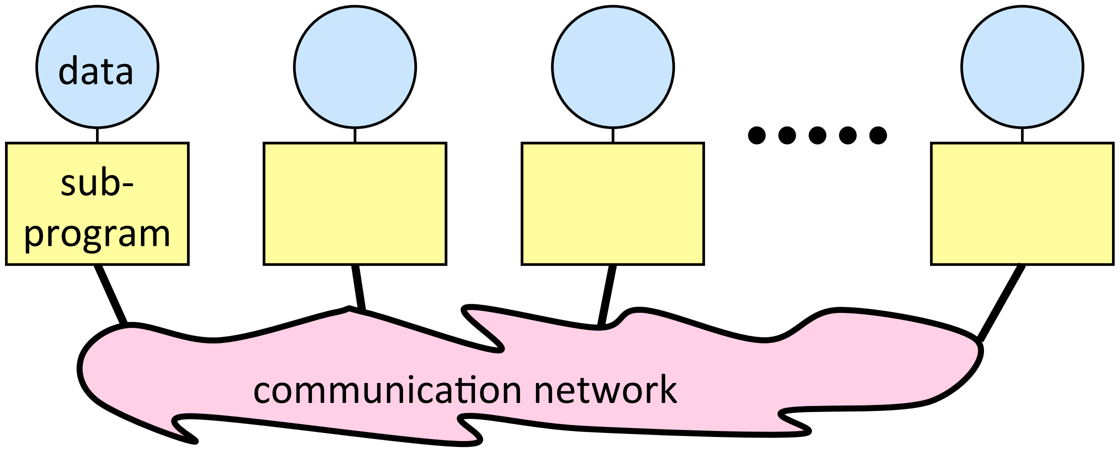
- **collective communication**
  - e.g. broadcast, reduction, …



- **MPI basics – summary**

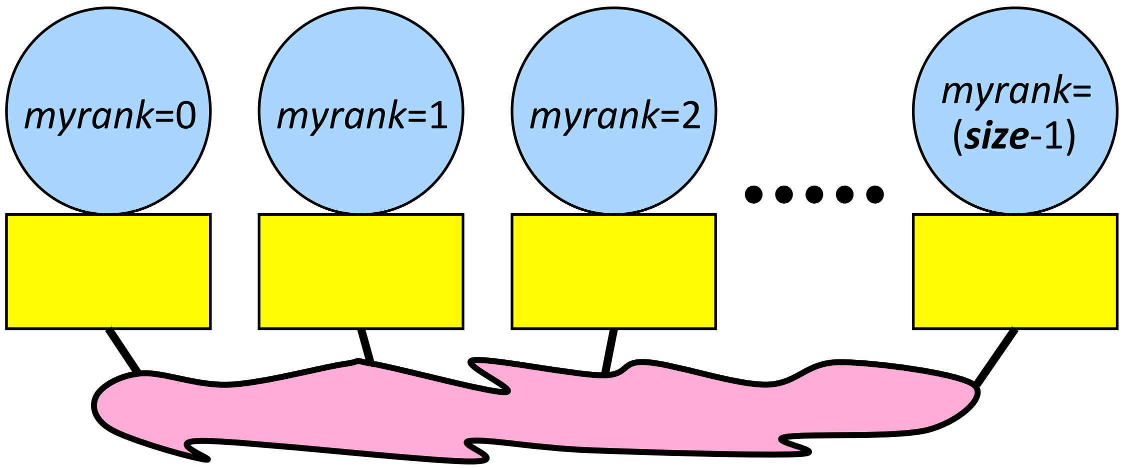
each processor in a message passing program runs a ***sub-program***

- written in a conventional sequential language, e.g., C, Fortran, or python
- typically the same on each processor (SPMD), all variables are private
- communicate via special send & receive routines (***message passing***)



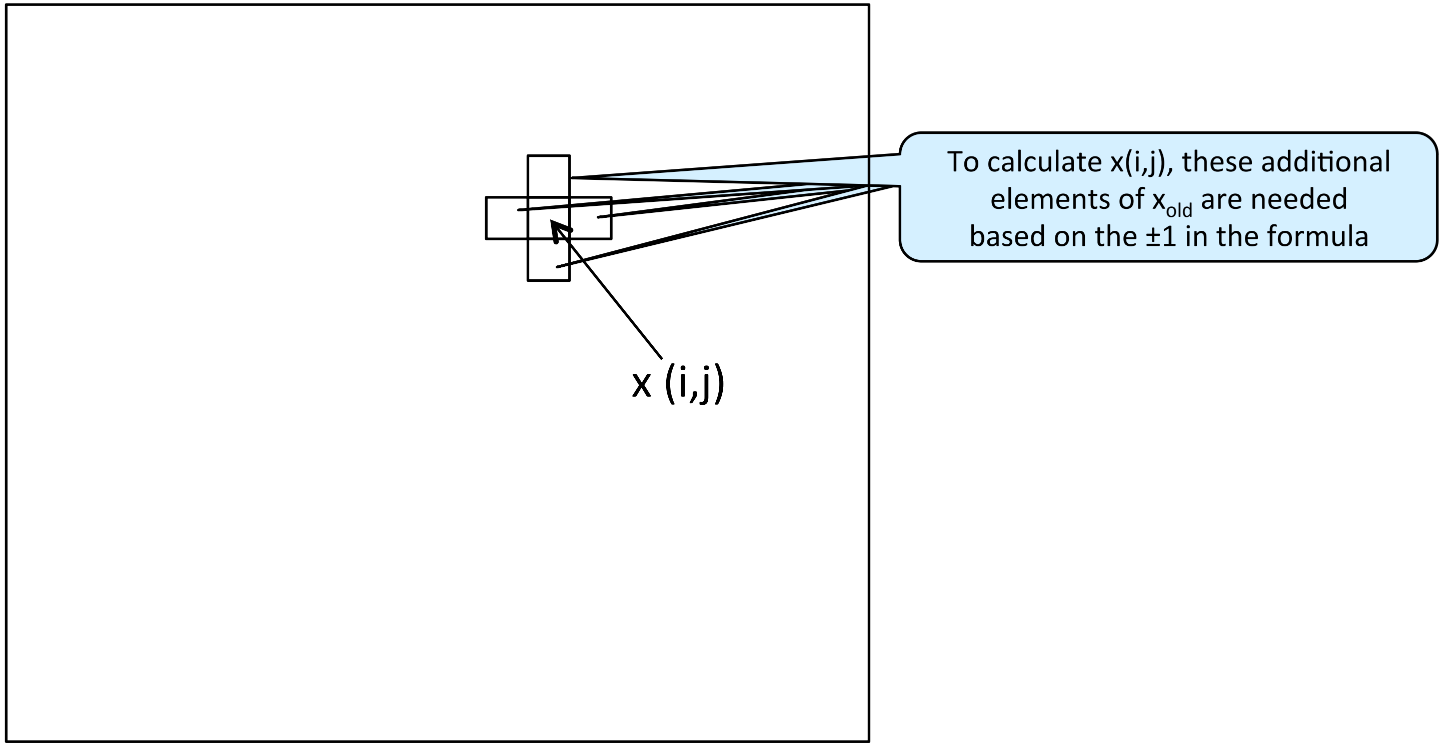
data & work distribution

- the system of ***size*** processes is started by special MPI initialization program
- the value of ***myrank*** is returned by special library routine
- all distribution decisions are based on ***myrank***



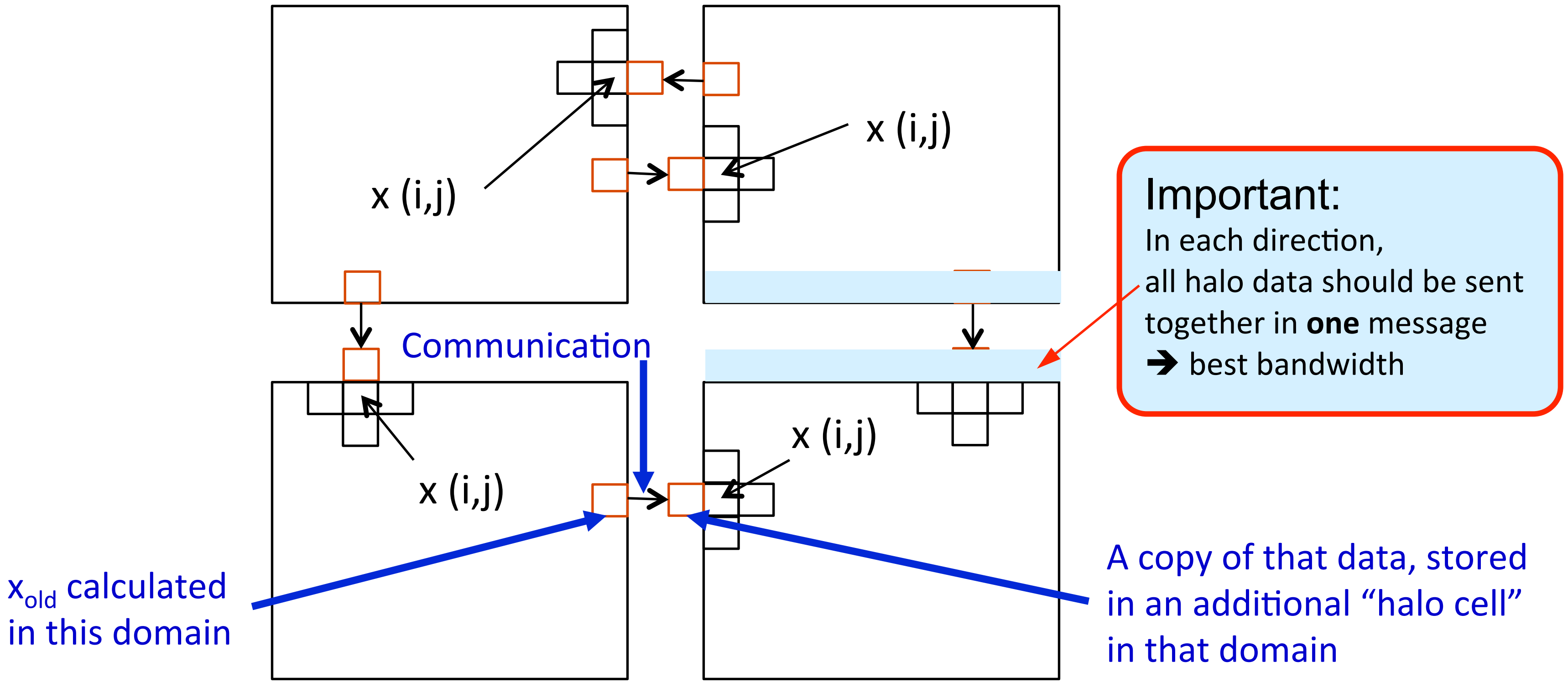
domain decomposition – serial

- $x(i,j) = f\ (x_{old}(i,j),\ x_{old}(i-1,j),\ x_{old}(i+1,j),\ x_{old}(i,j-1),\ x_{old}(i,j+1))$



domain decomposition – parallel

- $x(i,j) = f\ (x_{old}\ (i,j),\ x_{old}(i-1,j),\ x_{old}\ (i+1,j),\ x_{old}\ (i,j-1),\ x_{old}\ (i,j+1))$



Important:
In each direction,
all halo data should be sent together in **one** message
➔ best bandwidth

MPI process model

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- must be linked with an MPI library → `mpicc, mpiicc, ...` `mpif90, mpiifort, ...`
- must use include file of this MPI library →

```
#include <mpi.h>        C/C++
```

```
use mpi_f08             Fortran

use mpi
include ´mpif.h´
```

```
from mpi4py import MPI  py
```

- must be started with the MPI startup tool →

```
mpirun, mpiexec, srun,...
mpirun  -n # ./a.out
```

MPI function format & language bindings

C/C++

```
error = MPI_Xxxxxx(parameter,...);
MPI_Xxxxxx(parameter,...);
```

Fortran

```
call MPI_Xxxxxx(parameter,...,ierror)

             with mpi_f08 ierror is optional
        with mpi & mpif.h ierror is mandatory
```

python

result_value_or_object = input_mpi_object.mpi_action(parameter,...)

! not part of the MPI standard !

```
comm = MPI.COMM_WORLD
comm.Send(...)
```

(numpy) OR `comm.send(...)`

https://mpi4py.readthedocs.io/

MPI standard } – language independend

each routine } – programming languages: C / Fortran mpi_f08 / mpi & mpif.h

initializing & finalizing MPI

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

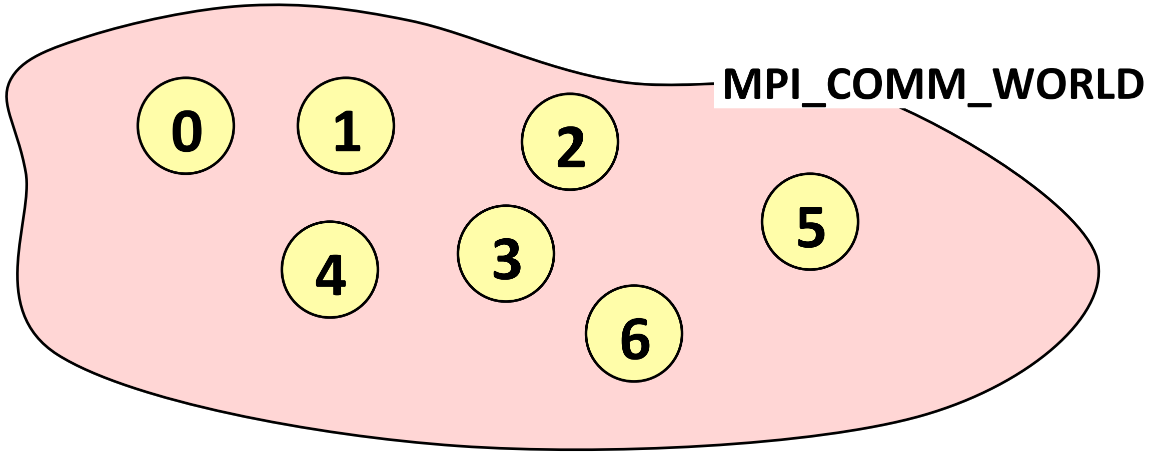
C/C++
```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
MPI_Init(&argc, &argv);
...
MPI_Finalize();
}
```

Fortran
```fortran
program xxxxx
use mpi_f08
implicit none

call MPI_INIT(ierror)
...
call MPI_FINALIZE(ierror)
end program
```

python
```python
from mpi4py import MPI
MPI_Init(), MPI_Init_thread(), MPI_Finalize()     } mpi4py
MPI_Is_initialized(), MPI_Is_finalized()           }
```

communicator MPI_COMM_WORLD

- all processes (= sub-programs) of one MPI program are combined in the **communicator MPI_COMM_WORLD** (predefined handle)
- **size** is the number of processes in a communicator
- each process has its own **rank** in a communicator
  starting with 0 – ending with (size-1)



rank

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- **rank** – identifies the different processes – basis for any work and data distribution

C/C++

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
→      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Fortran

```
MPI_COMM_RANK(comm, rank, ierror)
mpi_f08:          TYPE(MPI_Comm) :: comm
                  INTEGER :: rank
                  INTEGER, OPTIONAL :: ierror
mpi & mpif.h:     INTEGER comm, rank, ierror
→ call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
```

python

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

size

- **size** – how many processes are contained within a communicator?

C/C++

```
int MPI_Comm_size(MPI_Comm comm, int *size)
→      MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Fortran

```
MPI_COMM_SIZE(comm, size, ierror)
mpi_f08:           TYPE(MPI_Comm) :: comm
                   INTEGER :: size
                   INTEGER, OPTIONAL :: ierror
mpi & mpif.h:      INTEGER comm, size, ierror
→ call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
```

python

```
comm = MPI.COMM_WORLD
size = comm.Get_size()
```

exercise: Hello world!

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

1

- write a minimal MPI program that prints "Hello world!" by each MPI process
- compile and run it on a single processor
- run it on several processors in parallel

```
Hello world
Hello world
Hello world
Hello world
```

2

- modify your program so that
  - every process writes its rank and the size of MPI_COMM_WORLD
  - only process ranked 0 in MPI_COMM_WORLD prints "Hello world"

```
I am 2 of 4
Hello world
I am 0 of 4
I am 3 of 4
I am 1 of 4
```

- why is the sequence of the output non-deterministic?
- run the version tests provided…

cd ~/##/MPI/**C**/1_hello/
cd ~/##/MPI/**F**/1_hello/
cd ~/##/MPI/**P**/1_hello/

[1] hello-skel*
[2] myrank-skel*
[a] version_test*

} see: solutions/

solution: Hello world! C/C++

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
   int my_rank, size;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   if (my_rank == 0)
   {
     printf ("Hello world!\n");
   }
   printf("I am process %i out of %i\n", my_rank, size);

   MPI_Finalize();
}
```

solution: Hello world! **Fortran**

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
PROGRAM hello
  USE mpi_f08
  IMPLICIT NONE

  INTEGER my_rank, size

  CALL MPI_Init()

  CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
  CALL MPI_Comm_size(MPI_COMM_WORLD, size)

  IF (my_rank .EQ. 0) THEN ; WRITE(*,*) 'Hello world!, ; END IF
  WRITE(*,*) 'I am process', my_rank, ' out of', size

  CALL MPI_Finalize()
END PROGRAM
```

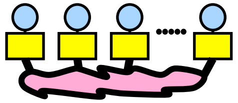
solution: Hello world! python

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

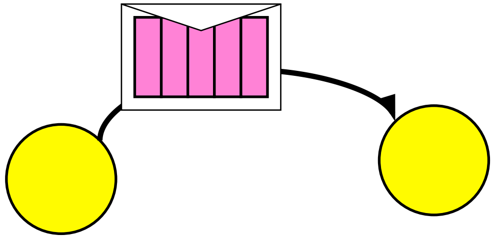
```
from mpi4py import MPI

comm_world = MPI.COMM_WORLD

my_rank = comm_world.Get_rank()
size = comm_world.Get_size()

if (my_rank == 0):
   print("Hello World!")

print(f"I am process {my_rank} out of {size}”)
```

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

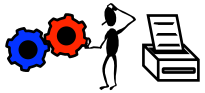
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



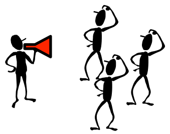
- **messages and point-to-point communication**
  - the MPI processes can communicate



- **nonblocking communication**
  - to avoid idle times, serializations, and deadlocks



- **collective communication**
  - e.g. broadcast, reduction, …

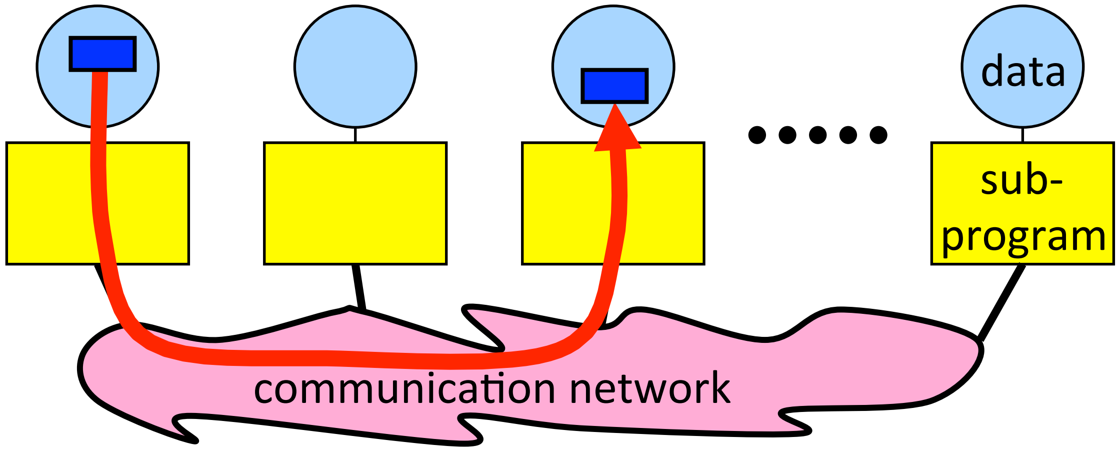


- **MPI basics – summary**

point-to-point communication

- **messages** are packets of data moving between MPI processes
- necessary information for the message passing system:

– sending process	– receiving process	} i.e., the ranks
– source location	– destination location	
– source data type	– destination data type	
– source data size	– destination buffer size	



messages

- a message contains a number of elements of some particular datatype
- MPI datatypes:
  - basic datatypes
  - 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

python: messages can be stored in

objects → `comm.send(…)` → slow (serialization)

numpy arrays → `comm.Send(…)` → fast communication

example: message with 5 integers

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

MPI basic datatypes

C/C++

python

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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-4.0, Annex A.1

example: message with 5 integers

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

arguments for MPI send/recv

count=5

datatype=MPI_INT

declaration of the buffers

int arr[5];

python: all C datatype handles can be used, syntax: e.g., MPI.FLOAT

MPI basic datatypes Fortran

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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, see, e.g., MPI-4.0, Annex A.1

example: message with 5 integers

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

arguments for MPI send/recv

count=5

datatype=MPI_INTEGER

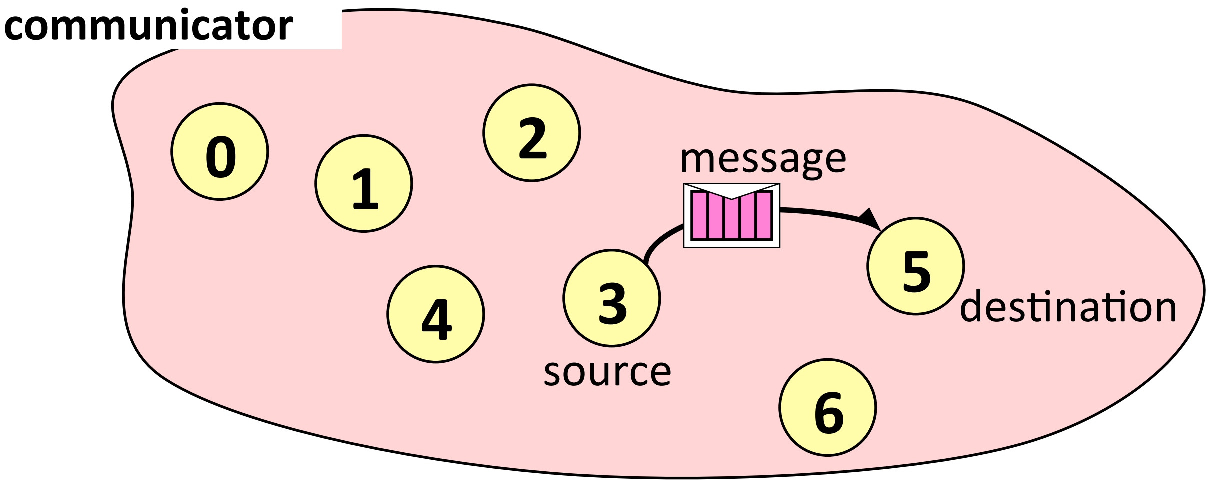
declaration of the buffers

INTEGER arr(5)

point-to-point communication

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- 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


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

```
MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
mpi_f08:      TYPE(*), DIMENSION(..) :: buf
              TYPE(MPI_Datatype) :: datatype
              TYPE(MPI_Comm) :: comm                          Fortran
              INTEGER :: count, dest, tag
              INTEGER, OPTIONAL :: ierror
mpi & mpif.h: <type> buf(*)
              INTEGER count, datatype, dest, tag, comm, ierror
```

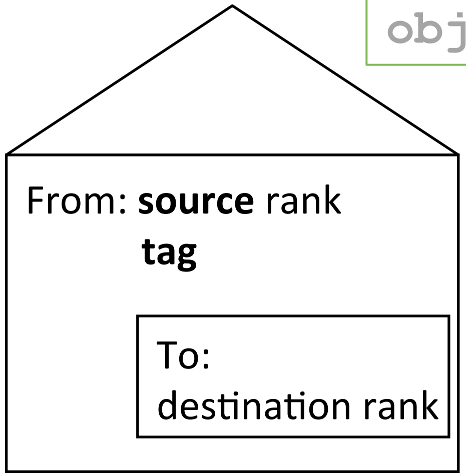
```
comm.Send(buf, int dest, int tag=0)   { buf                      python
comm.send(obj, int dest, int tag=0)   { (buf,datatype)
                                      { (buf,count,datataype)
```

receiving a message

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

```
MPI_RECV(buf, count, datatype, source, tag,
         comm, status, ierror)
```
Fortran

```
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)
```
python

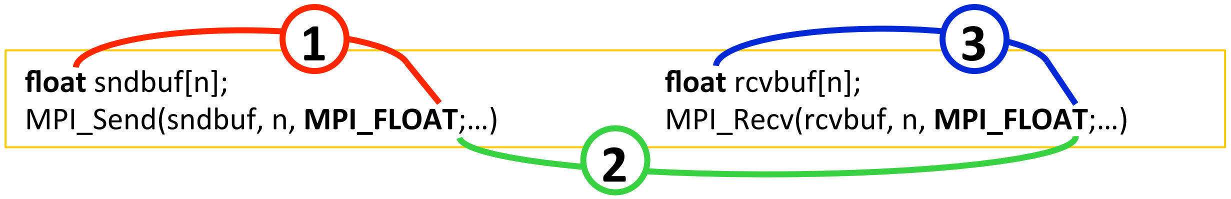


- to receive from any source — <u>source</u> = MPI_ANY_SOURCE
- to receive from any tag — <u>tag</u> = MPI_ANY_TAG
- actual source and tag are returned in <u>status</u>
- if not interested pass MPI_STATUS_IGNORE

requirements for p2p communication

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- sender must specify a valid destination rank
- receiver must specify a valid source rank
- the communicator must be the same
- tags must match
- type matching:



  (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

- receiver's buffer must be large enough

communiation modes

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Sender mode	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
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

← **debugging**

← **production**

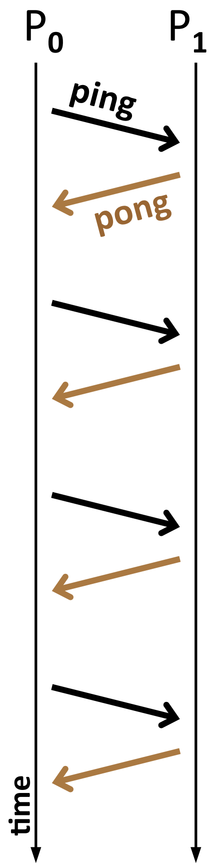
exercise: ping pong

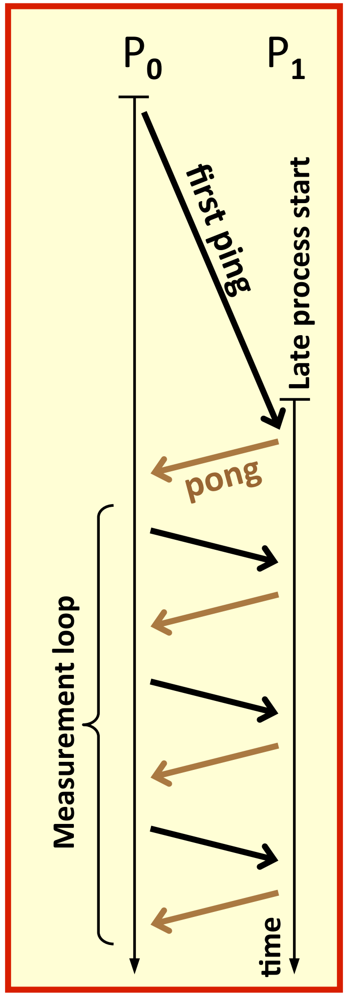
- write a program according to the time-line diagram:
  - 1 – process 0 sends a message to process 1 (**ping**)
  - 2 – after receiving this message, process 1 sends a message back to process 0 (**pong**)

message = 1 float | REAL

- 3 repeat this ping-pong with a loop of length 50
- add **timing** calls before and after the loop:
- C/C++: *double MPI_Wtime*(void);
- 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
- 4 first ping-pong before the timing loop

no printing inside of timing loop





cd ~/##/MPI/**C**/2_pingpong/
cd ~/##/MPI/**F**/2_pingpong/
cd ~/##/MPI/**P**/2_pingpong/

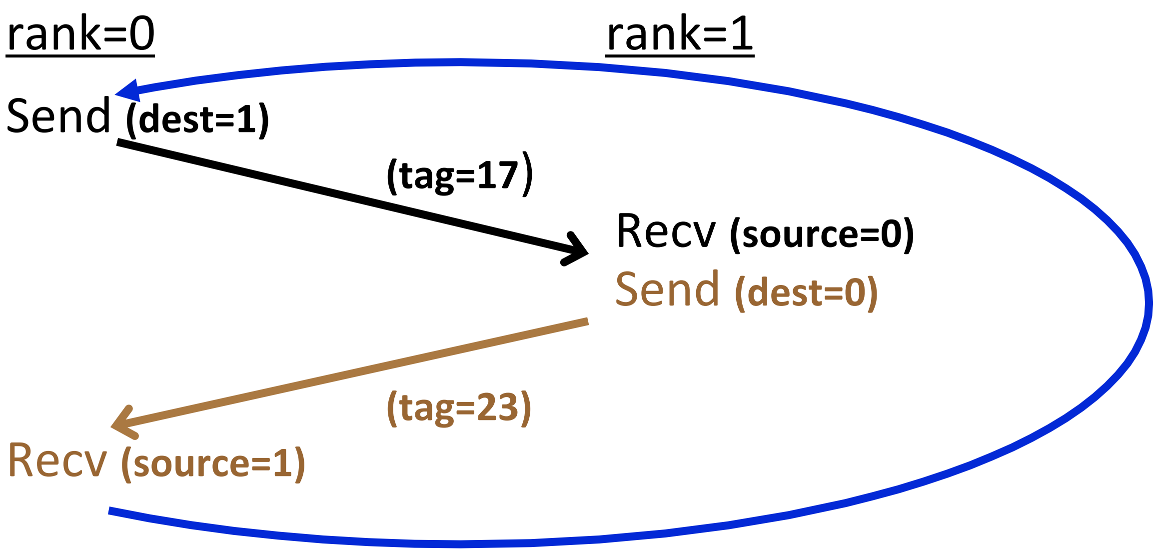
[1] ping-skel*
[2] pingpong-skel*
[3+] pingpong-bench-skel*

see: solutions/

try with SEND & SSEND
python: send & ssend
python: Send & Ssend

exercise: ping pong

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP




```
if (my_rank==0)
  MPI_Send( ... dest=1 ...)
  MPI_Recv( ... source=1 ...)
else
  MPI_Recv( ... source=0 ...)
  MPI_Send( ... dest=0 ...)
fi
```

solution: ping pong C/C++

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
start = MPI_Wtime();

for (i = 1; i <= 50; i++)
{
  if (my_rank == 0)
  {
     MPI_Send(buffer, 1, MPI_FLOAT, 1, 17, MPI_COMM_WORLD);
     MPI_Recv(buffer, 1, MPI_FLOAT, 1, 23, MPI_COMM_WORLD, &status);
  }
  else if (my_rank == 1)
  {
     MPI_Recv(buffer, 1, MPI_FLOAT, 0, 17, MPI_COMM_WORLD, &status);
     MPI_Send(buffer, 1, MPI_FLOAT, 0, 23, MPI_COMM_WORLD);
  }
}

finish = MPI_Wtime();

if (my_rank == 0)
  printf("Time for one messsage: %f micro seconds.\n",
           finish - start) / (2 * 50) * 1e6 );
```

solution: ping pong

Fortran

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
start = MPI_Wtime()
DO i = 1, 50
   IF (my_rank .EQ. 0) THEN
      CALL MPI_Send(buffer, 1, MPI_REAL, 1, 17, MPI_COMM_WORLD)
      CALL MPI_Recv(buffer, 1, MPI_REAL, 1, 23, MPI_COMM_WORLD, status)
   ELSE IF (my_rank .EQ. 1) THEN
      CALL MPI_Recv(buffer, 1, MPI_REAL, 0, 17, MPI_COMM_WORLD, status)
      CALL MPI_Send(buffer, 1, MPI_REAL, 0, 23, MPI_COMM_WORLD)
   END IF
END DO
finish = MPI_Wtime()
IF (my_rank .EQ. proc_a) THEN
   WRITE(*,*) 'One message:',(finish-start)/(2*50)*1e6,' micro seconds'
ENDIF
```

solution: ping pong

python

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
from mpi4py import MPI

number_of_messages = 50
buffer = 0.0
status = MPI.Status()

comm_world = MPI.COMM_WORLD
my_rank = comm_world.Get_rank()
```

```
start = MPI.Wtime()

for i in range(1, number_of_messages+1):
    if (my_rank == 0):
        comm_world.send(buffer, dest=1, tag=17)
        buffer = comm_world.recv(source=1, tag=23, status=status)
    elif (my_rank == 1):
        buffer = comm_world.recv(source=0, tag=17)
        comm_world.send(buffer, dest=0, tag=23)

finish = MPI.Wtime()

if (my_rank == 0):
    msg_transfer_time = ((finish - start) / (2 * number_of_messages)) * 1e6
    print(f"Time for one messsage: {msg_transfer_time:f} micro seconds.")
```

solution: ping pong

python
numpy

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

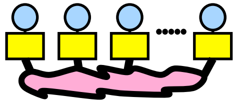
```python
from mpi4py import MPI
import numpy as np

number_of_messages = 50
buffer = np.array([0], dtype='f')
status = MPI.Status()

comm_world = MPI.COMM_WORLD
my_rank = comm_world.Get_rank()
```

```python
start = MPI.Wtime()

for i in range(1, number_of_messages+1):
    if (my_rank == 0):
        comm_world.Send((buffer,1,MPI.FLOAT), dest=1, tag=17)
        comm_world.Recv((buffer,1,MPI.FLOAT), source=1, tag=23, status=status)
    elif (my_rank == 1):
        comm_world.Recv((buffer,1,MPI.FLOAT), source=0, tag=17, status=status)
        comm_world.Send((buffer,1,MPI.FLOAT), dest=0, tag=23)

finish = MPI.Wtime()

if (my_rank == 0):
    msg_transfer_time = ((finish - start) / (2 * number_of_messages)) * 1e6
    print(f"Time for one messsage: {msg_transfer_time:f} micro seconds.")
```

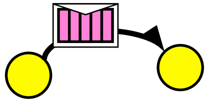
nonblocking communication

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

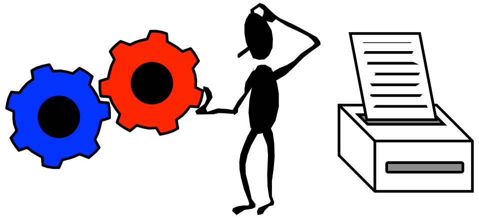
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



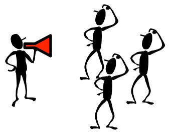
- **messages and point-to-point communication**
  - the MPI processes can communicate



- **nonblocking communication**
  - to avoid idle times, serializations, and deadlocks



- **collective communication**
  - e.g. broadcast, reduction, …



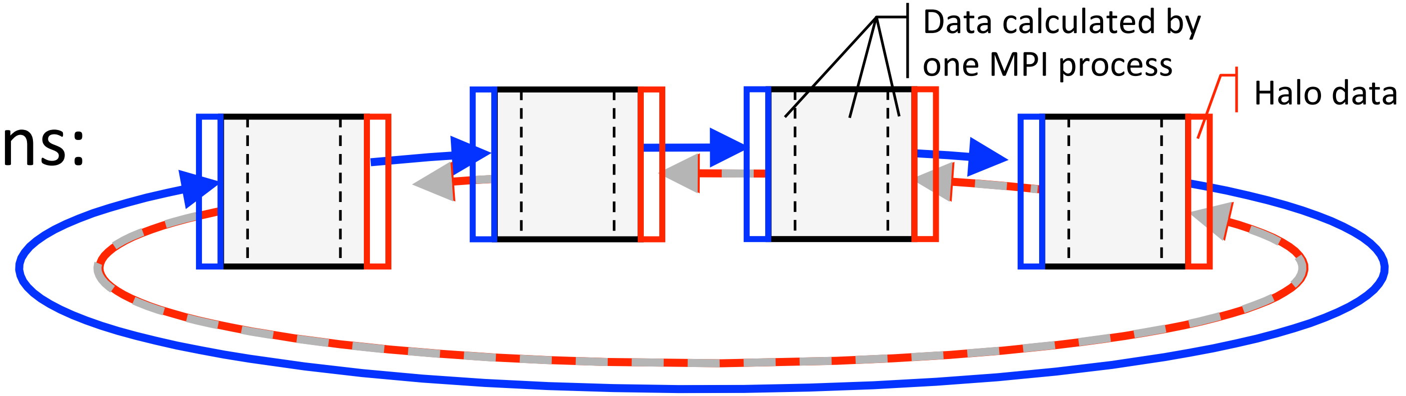
- **MPI basics – summary**

nonblocking communication

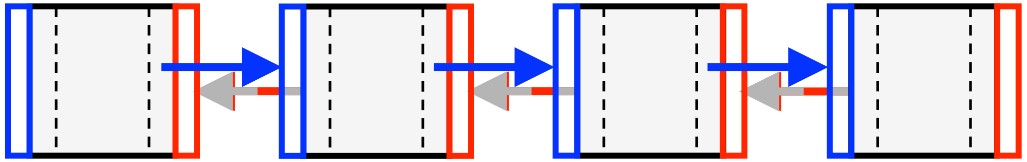
→ **to avoid idle times, serializations and deadlocks**

→ **halo communication**

cyclic boundary conditions:



non-cyclic boundary:

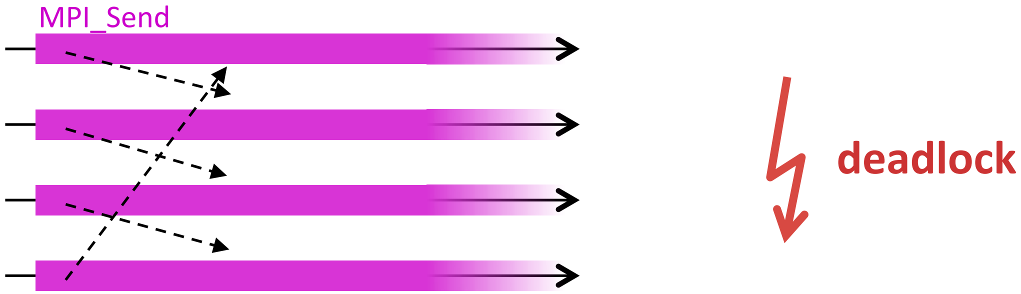


if the MPI library chooses the synchronous protocol

timelines of all processes

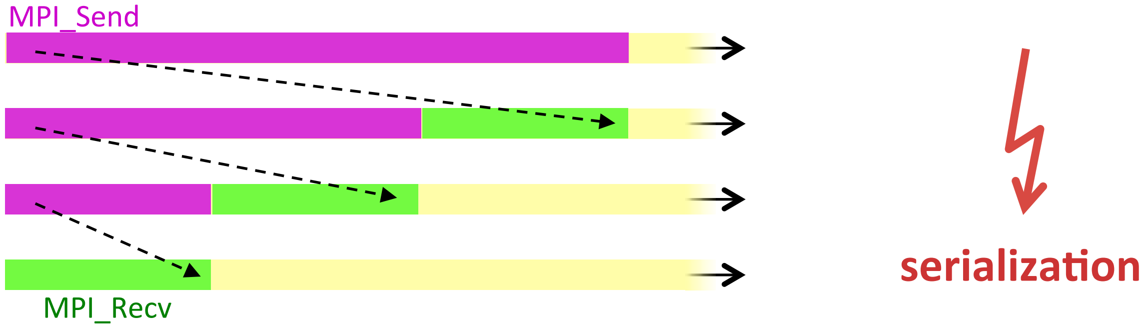
cyclic boundary:

```
MPI_Send(..., right, ...)
MPI_Recv(  ..., left,  ...)
```



non-cyclic boundary:

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



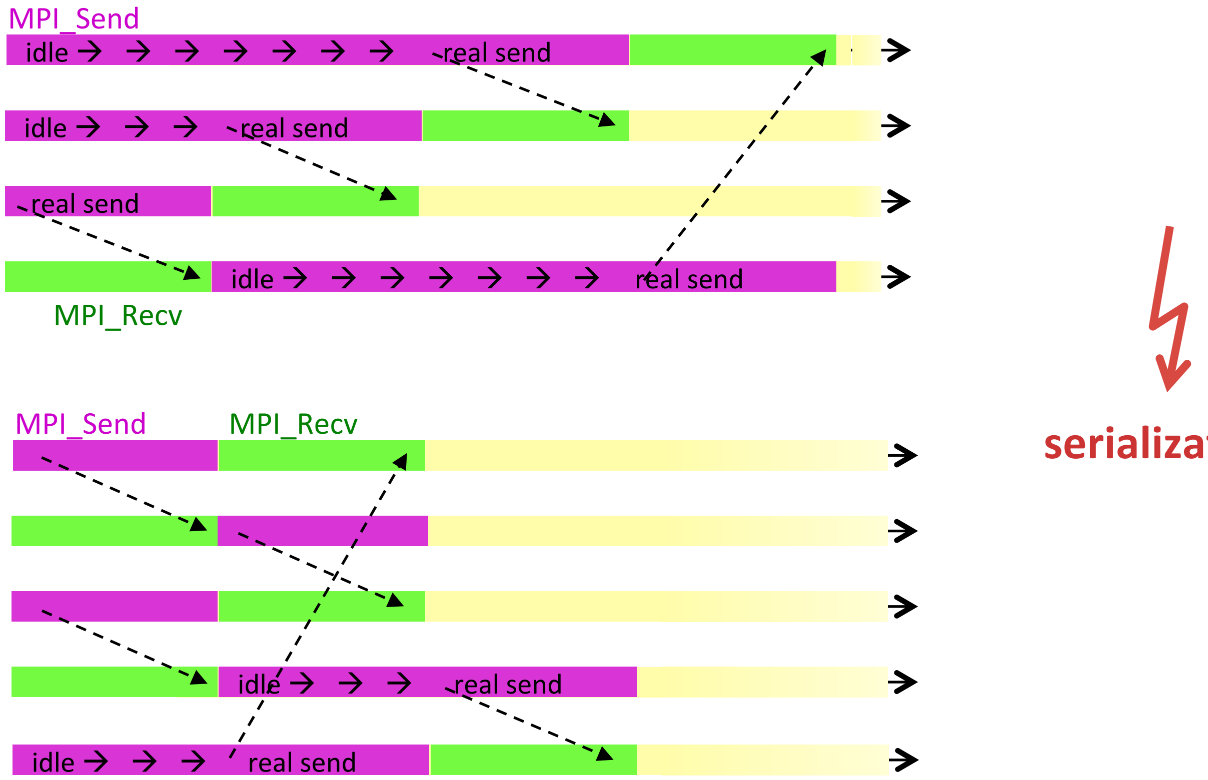
SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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

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

if the MPI library chooses the synchronous protocol

timelines of all processes



serialization

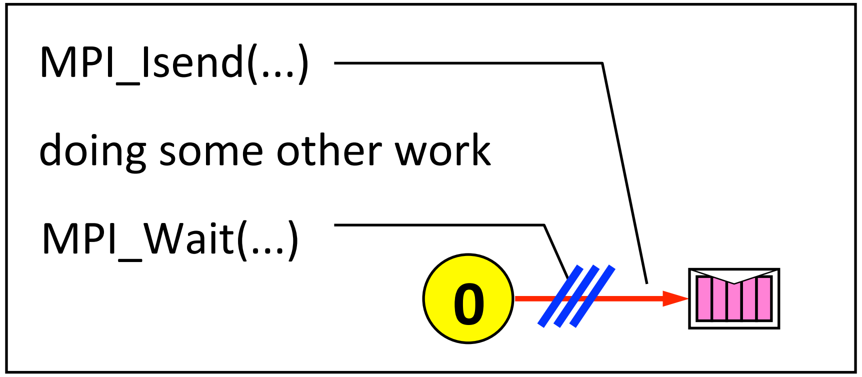
nonblocking communication

separate communication into **three phases**:

- initiate nonblocking communication
  - routine name starting with MPI_**I**…
  - incomplete
  - local, returns immediately,
    returns independently of any other process’ activity

→ do some work (perhaps involving other communications?)

- wait for nonblocking communication to **complete**
  - the send buffer is read out, or
  - the receive buffer is filled in



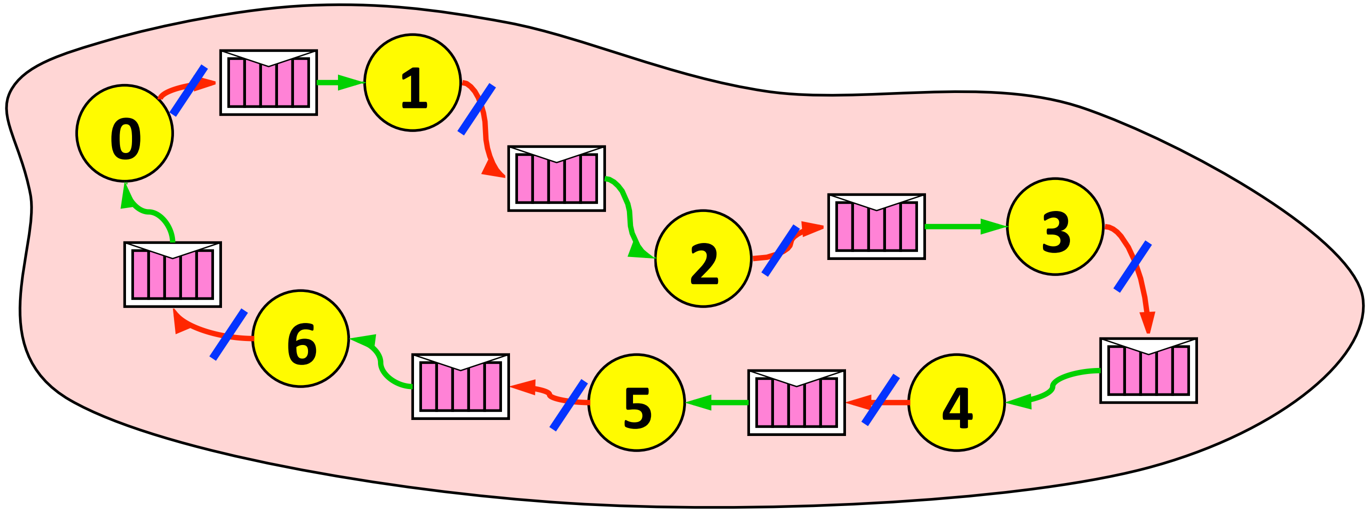
the definition of nonblocking is clarified in

MPI-4.0

reading: MPI-4.0/2.4 & MPI-4.0/3.7

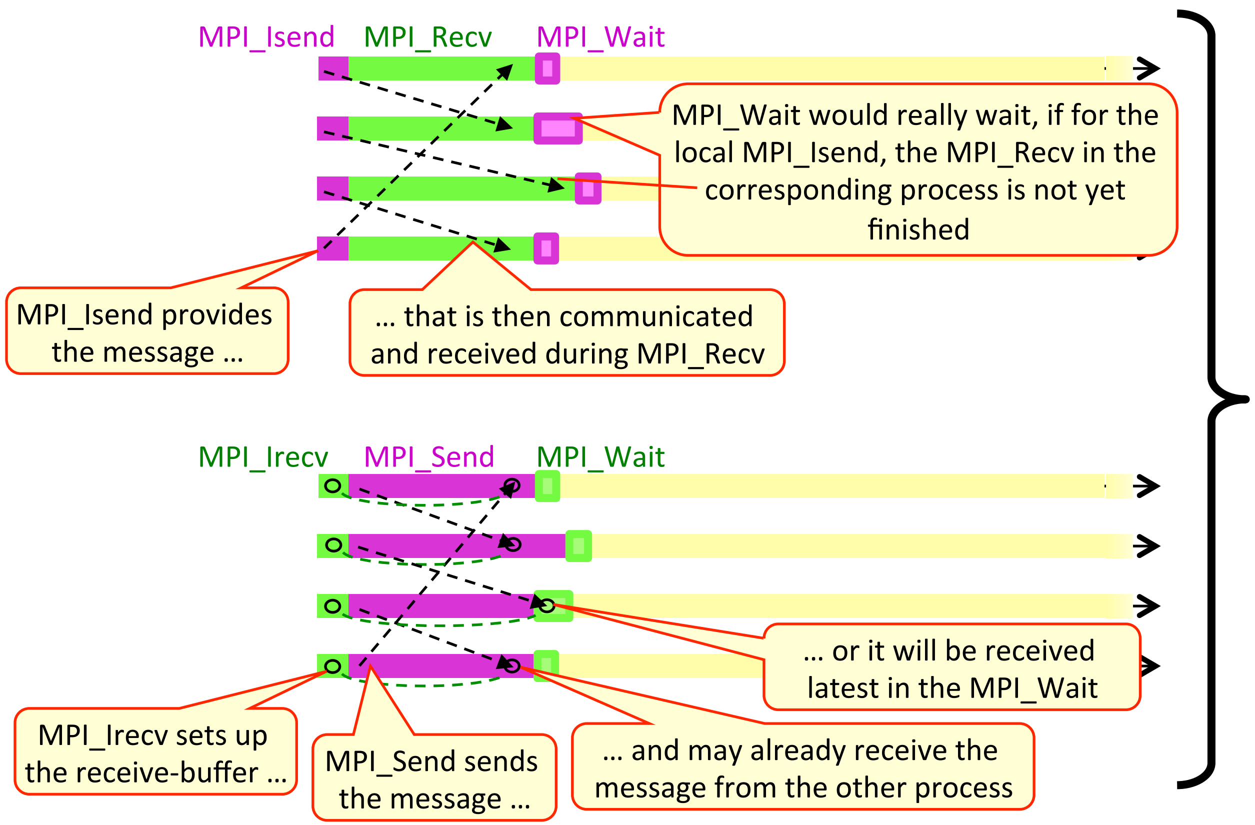
nonblocking send

- 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



nonblocking timelines

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP



no serialization
no deadlock

request handles

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- 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, e.g., in C: MPI_Datatype, MPI_Comm
- **request handles**
- are used for nonblocking communication
- **must** be stored in local variables →

C/C++:   MPI_Request
Fortran: TYPE(MPI_Request) / INTEGER
python: automatically

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

nonblocking synchronous send

for debugging only

```
MPI_Issend(&buf, count, datatype, dest, tag, comm,
              [OUT] &request_handle);
                    ⬇
MPI_Wait([INOUT] &request_handle, &status)
```

C/C++

```
..., 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 )
```

Fortran

```
request = comm_world.Issend(…)
status = MPI.Status(); request.Wait(status)
```

python

- buf must not be modified between Issend and Wait
- nothing returned in status (because send operations have no status)
- "Issend + Wait directly after Issend" is equivalent to blocking call (Ssend)

→ **ss for debugging only**

→ **s for production code**

nonblocking receive

C/C++

```
MPI_Irecv (buf, count, datatype, source, tag, comm,
           [OUT] &request_handle);
                  ⬇
MPI_Wait[INOUT] &request_handle, &status)
```

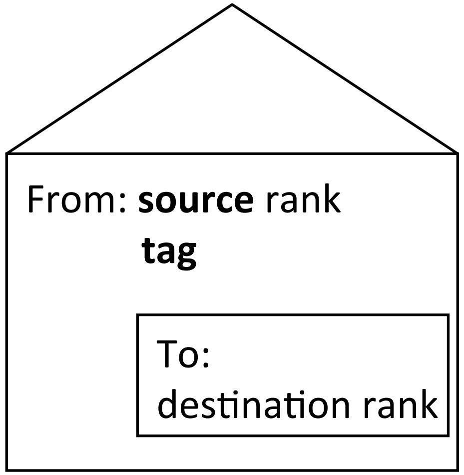
Fortran

```
..., ASYNCHRONOUS :: buf
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)CALL MPI_F_SYNC_REG( buf )
```

python

```
request = comm_world.Irecv(…)
status = MPI.Status(); request.Wait(status)
```

- <u>buf</u> must not be used between <u>Irecv</u> and <u>Wait</u>
- message <u>status</u> is returned in <u>Wait</u>
- "<u>Irecv</u> + <u>Wait</u> directly after Irecv" is equivalent to blocking call (<u>Recv</u>)



blocking and nonblocking

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- 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….

- A nonblocking operation immediately followed by a matching wait is equivalent to the blocking operation

completion

C/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)
```

python

```
status = MPI.Status(); request.Wait(status)
status = MPI.Status(); flag = request.Test(status)
```

- one must
  - WAIT or
  - loop with TEST until request is completed, i.e., flag == non-zero or .TRUE. or True
- multiple nonblocking communications (several request handles)
  - MPI_[Wait|Test]any, MPI_[Wait|Test]all, MPI_[Wait|Test]some

use cases for nonblocking comm.

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

→ **to avoid idle times, serializations and deadlocks**
(as if overlapping of communication with other communication)

→ **real overlapping of**

- several communications
- communication and computation

→ **other MPI features: Send-Receive in one routine**

- MPI_Sendrecv & MPI_Sendrecv_replace (blocking → prevent serializations & deadlocks)
- combines the triple "MPI_Irecv + Send + Wait" into one routine
- MPI_Isendrecv & MPI_Isendrecv_replace (nonblocking → minimize idle times) ← **new MPI 4.0**

exercise: ring

init (1)

iterations (2) (3) (4) (5)

sum = (size) * (size-1) / 2

- a set of processes 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
- preparation of next iteration
- each processor 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

- use nonblocking MPI_Issend
- keep the blocking MPI_Recv

single program
no if statements

hint – neigbor ranks:

C/C++:

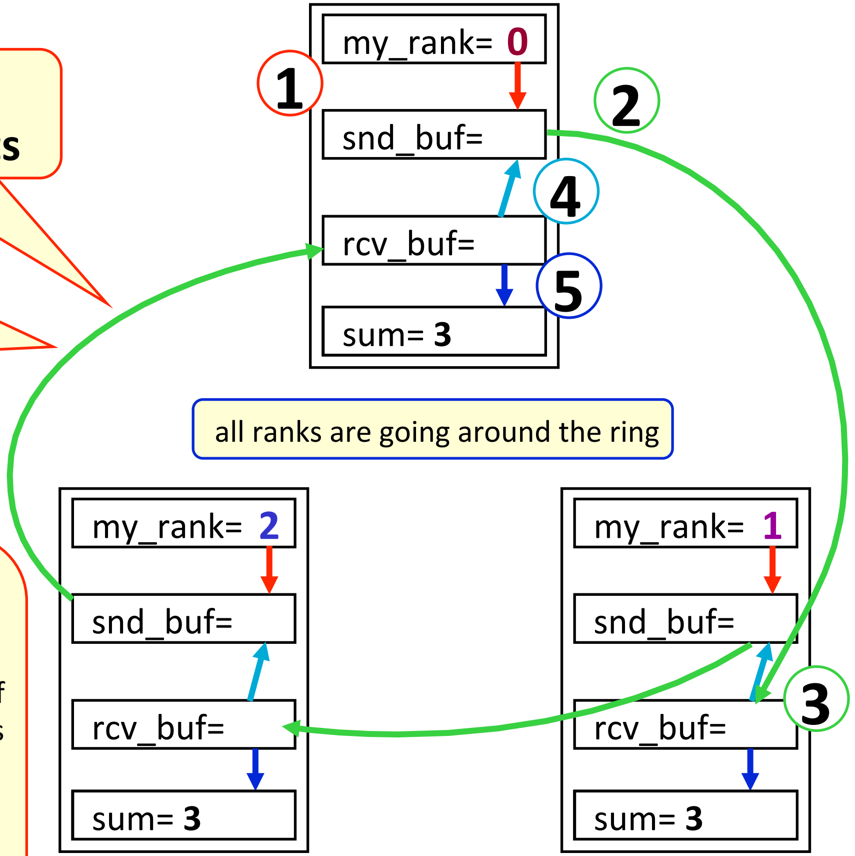
```
dest        = (my_rank+1) % size;
source      = (my_rank–1+size) % size;
```

Fortran:

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

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

A real application would use *standard* **Isend() !!!**
Never ***synchronous*** **Issend() !!!**



all ranks are going around the ring

final figure shows data at the end of all iterations

cd ~/##/MPI/**C**/3_ring/
cd ~/##/MPI/**F**/3_ring/
cd ~/##/MPI/**P**/3_ring/

ring-skel* } see: solutions/

try to see deadlock (SS) !!!!!
try also: IRECV – ISSEND !!!!!
try also: SENDRECV (no sol.)

solution: ring

C/C++

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
int snd_buf, rcv_buf;
int right, left;
int sum, my_rank, size, i;
MPI_Status  status;
MPI_Request request;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

right = (my_rank+1)      % size;
left  = (my_rank-1+size) % size;
sum = 0;
snd_buf = my_rank;                                    1
for( i = 0; i < size; i++)
{
  MPI_Issend(&snd_buf, 1, MPI_INT, right, 17, MPI_COMM_WORLD, &request);   2
  MPI_Recv ( &rcv_buf, 1, MPI_INT, left,  17, MPI_COMM_WORLD, &status);    3
  MPI_Wait(&request, &status);
  snd_buf = rcv_buf;                                  4
  sum += rcv_buf;                                     5
}
printf ("PE%i:\tSum = %i\n", my_rank, sum);
MPI_Finalize();
```

In C, normally such helper variables should be declared only within the scope where needed, here the loop.
For our exercises, they are all declared at the beginning, mainly to keep C and Fortran solutions identical.

Synchronous **send** (**Issend**) instead of standard send (**Isend**) is used only to demonstrate the use of the nonblocking routine resolves the deadlock (or serialization) problem.
A real application would use standard **Isend()**.

solution: ring

Fortran

```
2      INTEGER, ASYNCHRONOUS :: snd_buf
       INTEGER :: rcv_buf, sum, i, my_rank, size
       TYPE(MPI_Status)  :: status
       TYPE(MPI_Request) :: request
       INTEGER(KIND=MPI_ADDRESS_KIND) :: iadummy
       CALL MPI_Init()
       CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
       CALL MPI_Comm_size(MPI_COMM_WORLD, size)
       right = mod(my_rank+1,      size)
       left  = mod(my_rank-1+size, size)
       sum = 0
1      snd_buf = my_rank
       DO i = 1, size
2 3       CALL MPI_Issend(snd_buf,1,MPI_INTEGER,right,17,MPI_COMM_WORLD, request)
          CALL MPI_Recv ( rcv_buf,1,MPI_INTEGER,left, 17,MPI_COMM_WORLD, status)
          CALL MPI_Wait(request, status)
          IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
4         snd_buf = rcv_buf
5         sum = sum + rcv_buf
       END DO
       WRITE(*,*) 'PE', my_rank, ': Sum =', sum
       CALL MPI_Finalize()
```

Synchronous **send** (**Issend**) instead of standard send (**Isend**) is used only to demonstrate the use of the nonblocking routine resolves the deadlock (or serialization) problem. A real appl. would use Isend.

solution: ring

python

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
#!/usr/bin/env python3
from mpi4py import MPI
import numpy as np

rcv_buf = np.empty((), dtype=np.intc)
status = MPI.Status()

comm_world = MPI.COMM_WORLD
my_rank = comm_world.Get_rank()
size = comm_world.Get_size()
right = (my_rank+1)        % size
left  = (my_rank-1+size) % size

sum = 0
snd_buf = np.array(my_rank, dtype=np.intc)
for i in range(size):
    request = comm_world.Issend((snd_buf, 1, MPI.INT), dest=right, tag=17)
    comm_world.Recv((rcv_buf, 1, MPI.INT), source=left, tag=17, status=status)
    request.Wait(status)
    np.copyto(snd_buf, rcv_buf) # We make a copy here.
    sum += rcv_buf
print(f"PE{my_rank}:\tSum = {sum}")
```

1 2 3 4 5

Synchronous **send** (**Issend**) instead of standard send (**Isend**) is used only to demonstrate the use of the nonblocking routine resolves the deadlock (or serialization) problem. A real appl. would use Isend.

MPI & **Fortran**

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- **Fortran** compiler is an optimizing compiler → tell it NOT to do certain optimizations

- MPI-1 → mpif.h → inconsistent with Fortan 90 (several routines substituted / deprecated)
  used INTEGER instead of INTEGER(KIND=MPI_ADDRESS_KIND)

- MPI-2 → mpi → aware of the Fortran issues → proposed a work-around

- MPI-3 → mpi_f08 → solves the previous inconsistencies with Fortran, needs TS 29113

nonblocking & **Fortran**

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- **Fortran source code**

```
CALL MPI_IRECV( buf, ..., request_handle, ierror)
CALL MPI_WAIT( request_handle, status, ierror)
write (*,*) buf
```

buf is not part of the argument list

data may be received in buf during MPI_Wait

- **may be compiled as**

```
CALL MPI_IRECV( buf, ..., request_handle, ierror)
registerA = buf
CALL MPI_WAIT( request_handle, status, ierror)
write (*,*) registerA
```

therefore old data may be printed instead of received data

- **solution**

```
<type>, ASYNCHRONOUS :: buf
CALL MPI_IRECV ( buf, ..., request_handle, ierror)
CALL MPI_WAIT( request_handle, status, ierror)
IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG( buf )
write (*,*) buf
```

with a Fortran 2018 or TS 29113 compiler
code movements with buf across subroutine calls are prohibited
scope includes MPI_Wait and the subsequent use of buf

buf is not part of the argument list

needed for **non-TS 29113** compiler directly after CALL MPI_Wait

work-around in older MPI versions:
CALL **MPI_GET_ADDRESS**(buf, *iaddrdummy*, *ierror*)
with INTEGER(KIND=MPI_ADDRESS_KIND) iaddrdummy

with a TS 29113 compiler, this will be removed at compile time
MPI_ASYNC_PROTECTS_NONBLOCKING == .TRUE.

strided subarrays & **Fortran**

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- **Fortran**

CALL MPI_ISEND (*buf(7,:,:)*, ..., *request_handle*, *ierror*)

- the content of this non-contiguous sub-array is stored in a temporary array
- then MPI_ISEND is called
- on return, the temporary array is **released**

other work

- The data may be transferred while other work is done, …
- … or inside of MPI_Wait, but the **data** in the temporary array **is already lost**!

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

Fortran source code

```
real, dimension(m,n) :: arr
...
CALL MPI_ISEND (arr(1,1:n),n,…)
```

will be compiled
(without TS 29113 compiler & mpi_f08) **as**

```
allocate( scratch_buf(n) )
scratch_buf(1:n) = array(1,1:n)
CALL MPI_ISEND(scratch_buf,n,…)
array(1,1:n) = scratch_buf(1:n)
deallocate(scratch_buf)
```

- **since MPI-3.0: works if MPI_SUBARRAYS_SUPPORTED == .TRUE.** (requires Fortran 200x + TS29113 or Fortran 2018 compiler)
- **still do not use non-contiguous sub-arrays in nonblocking calls!!!**
- contiguous array sections: pass the starting element (array(1,1)) instead of (array(1:m,1))
- non-contiguous sections: do an explicit copy to a contiguous temporary buffer (kept after Wait)
  or define an appropriate vector derived data type

ierror & **Fortran**

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- unused ierror

  INCLUDE 'mpif.h' or USE mpi
  ! wrong call, because **with mpi & mpif.h ierror is mandatory → NEVER FORGET!**
  CALL MPI_SEND(…., MPI_COMM_WORLD)
  ! → terrible implications because ierror=0 is written somewhere to the memory

- with the mpi_f08 module

  USE mpi_f08
  ! correct call, because **with mpi_f08 ierror is OPTIONAL**
  CALL MPI_SEND(…., MPI_COMM_WORLD)

- **solution →** switch to the **mpi_f08** module

mpi_f08 module for **Fortran**

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS 1) :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Wait(request, status, ierror) BIND(C)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- DIMENSION(..): to solve the strided-array problem
- TYPE(*), DIMENSION(..), ASYNCHRONOUS: Fortran compatible buffer declaration allows correct compiler optimizations
- TYPE(MPI_Datatype), TYPE(MPI_Comm), TYPE(MPI_Request): unique handle types allow best compile-time argument checking
- INTENT → compiler-based optimizations & checking
- TYPE(MPI_Status): status is now a Fortran structure, i.e., a Fortran derived type
- OPTIONAL ierror: MPI routine can be called without ierror argument

[1] ASYNCHRONOUS: only in nonblocking routines, not in MPI_Recv

keyword-based & **Fortran**

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

positional and **keyword-based** argument lists

- CALL MPI_SEND(sndbuf, 5, MPI_REAL, right, 33, MPI_COMM_WORLD)
- CALL MPI_SEND(**buf**=sndbuf, **count**=5, **datatype**=MPI_REAL,
  dest=right, **tag**=33, **comm**=MPI_COMM_WORLD)
  - keywords are defined in the language bindings for mpi_f08 & mpi
  - some keywords have changed, do not use outdated documents!
  - some MPI libraries show version numbers 3.0 or higher although they do not correctly implement keyword based argument lists (see version test routines)

Fortran – best practice

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Switch to the new **mpi_f08** module to be consistent with Fortran standard

```
program xxxxx
implicit none
include 'mpif.h'
```

(1) change ! →

```
program xxxxx
use mpi
implicit none
```

handle types (2) →

```
program xxxxx
use mpi_f08
implicit none
```

(1) Compile with a library that provides compile-time argument checking

(2) INTEGER rq, comm, datatype, status(MPI_STATUS_SIZE)

→ TYPE(MPI_Request) :: rq
TYPE(MPI_Comm) :: comm
TYPE(MPI_Datatype) :: datatype
TYPE(MPI_Status) :: status

!

full consistency requires Fortran 2003/2008 **+ TS 29113** or **Fortran 2018**

non-contiguous subarrays: do NOT use in nonblocking routines ! (workaround: see before)

buffers in nonblocking routines or together with MPI_BOTTOM or in 1-sided communication:

```
<type>, ASYNCHRONOUS :: buffer
IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(buffer)
```

after MPI_Wait or before **and** after blocking calls with MPI_BOTTOM
or before a nonblocking routine with MPI_BOTTOM **and** after final MPI_Wait / …
or in 1-sided communication before the 1st **and** after 2nd CALL MPI_Win_fence

in older MPI versions or if the MPI-3.0 Fortran support methods are incomplete:

```
INTEGER(KIND=MPI_ADDRESS_KIND) :: iadummy
CALL MPI_GET_ADDRESS(buffer, iadummy, ierror)
```

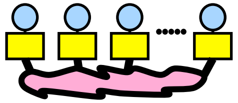
Fortran – further reading

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

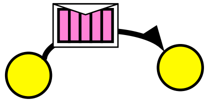
MPI 4.0 Sections

- 19.1.8 Additional Support for Fortran Register-Memory-Synchronization
- 19.1.10 Problems With Fortran Bindings for MPI
- 19.1.11 Problems Due to Strong Typing
- 19.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets
- 19.1.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts
- 19.1.14 Special Constants
- 19.1.15 Fortran Derived Types
- 19.1.16 Optimization Problems, an Overview
- 19.1.17 Problems with Code Movement and Register Optimization
  - Nonblocking Operations
  - One-sided Communication
  - MPI_BOTTOM and Combining Independent Variables in Datatypes
  - Solutions
  - The Fortran ASYNCHRONOUS Attribute
  - Calling MPI_F_SYNC_REG (new routine, defined in Section 19.1.7)
  - A User Defined Routine Instead of MPI_F_SYNC_REG
  - Module Variables and COMMON Blocks
  - The (Poorly Performing) Fortran VOLATILE Attribute
  - The Fortran TARGET Attribute
- 19.1.18 Temporary Data Movement and Temporary Memory Modication
- 19.1.19 Permanent Data Movement
- 19.1.20 Comparison with C

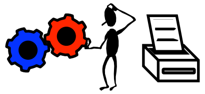
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



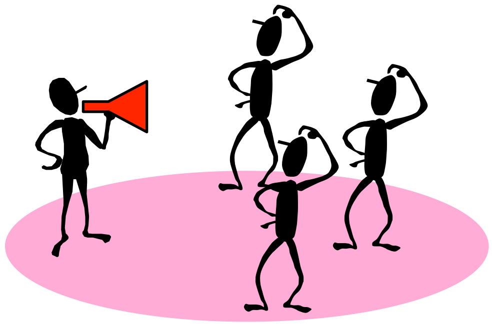
- **messages and point-to-point communication**
  - the MPI processes can communicate



- **non-blocking communication**
  - to avoid idle times, serializations, and deadlocks



- **collective communication**
  - e.g. broadcast, reduction, …



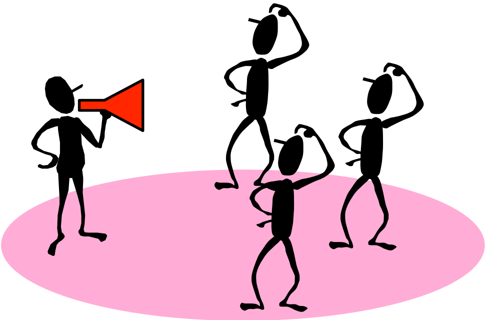
- **MPI basics – summary**

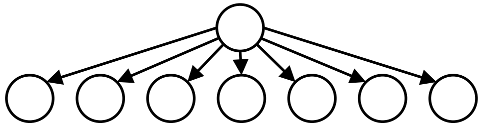
collective communication

- **all processes in a communicator** processes are involved
- can be built out of point-to-point communications, but …
- allow **optimized** internal implementations (by MPI libraries)
- examples:
  - **broadcast**, scatter, gather
  - reduction operations (global sum, maximum, etc.)
  - barrier synchronization (do NOT use in production code!)
  - neighbor communication in a virtual process grid

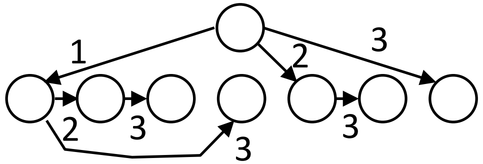
You need not to care about it !
It is the job of the MPI library !!!

Should be faster than any programming with point-to-point messages!





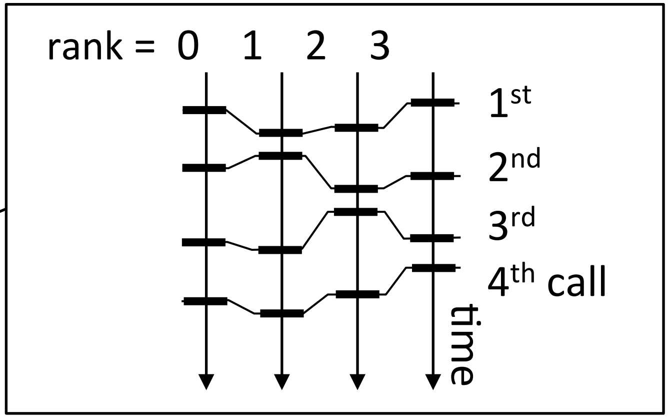
Sequential algorithm
O(# processes)



Tree based algorithm
$O(\log_2(\#\ \text{processes}))$

characteristics of collectives

- **collective** action over a communicator
- **all process of the communicator** must communicate, i.e., must call the collective routine
- synchronization may or may not occur, therefore all processes must be able to start the collective routine
- on a given communicator, the n-th collective call must match on all processes of the communicator
- available as blocking and nonblocking versions
- no tags



For each message, the amount of data sent **must exactly match** the amount of data specified by the receiver!
→ It is forbidden to provide receive buffer count arguments that are too long (and also too short, of course).

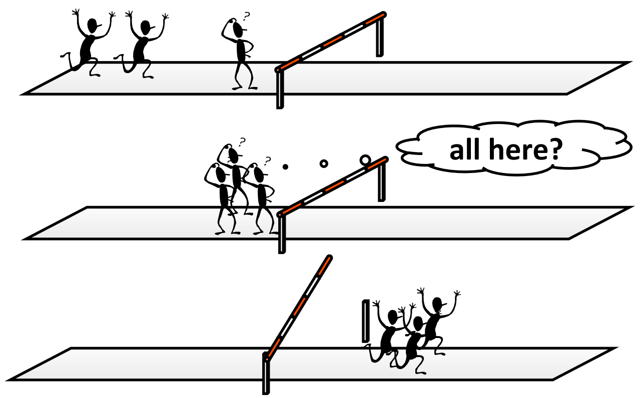
barrier synchronization

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
int MPI_Barrier(MPI_Comm comm)
```
C/C++

```
MPI_BARRIER(comm, ierror)
```
Fortran

```
comm.Barrier()
comm.barrier()
```
python



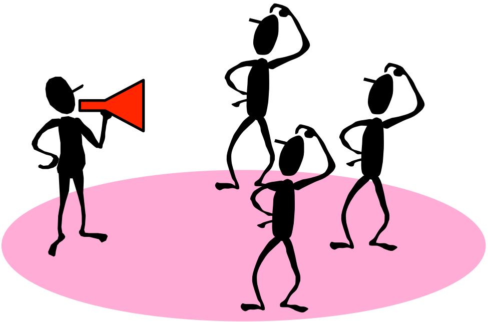
- MPI_Barrier is **never needed** in a production code
  → all synchronization is done implicitly by the data communication
  (a process cannot continue before it has the data it needs)
- **if used for profiling / debugging**
  → please guarantee that it is removed in production version of code
- **for profiling → to separate time measurement of**
  load imbalance of computation [MPI_Wtime(); MPI_Barrier(); MPI_Wtime()]
  communication epochs [MPI_Wtime(); MPI_Allreduce(); …; MPI_Wtime()]

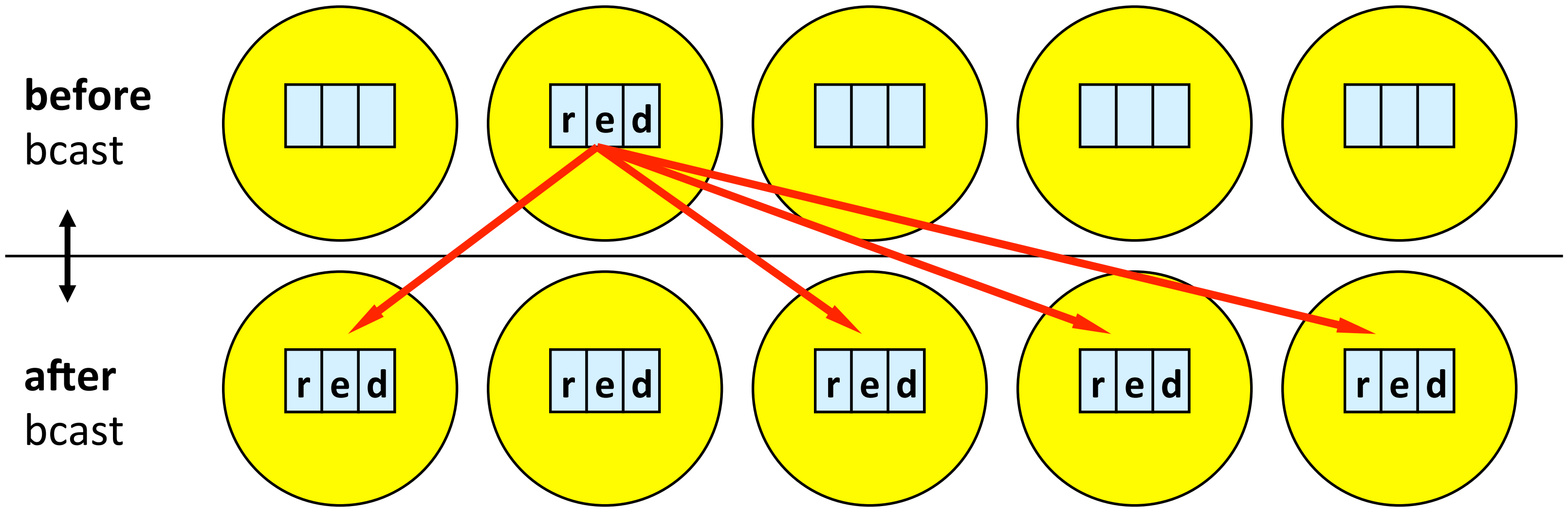
broadcast

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```
C/C++

```
MPI_BCAST(buf, count, datatype, root, comm, ierror)
```
Fortran

```
comm.Bcast(buf, int root=0)
comm.bcast(obj, int root=0)
```
python





e.g., root=1

root = rank of the sending/root process
must be given identically by all processes

MPI_Bcast (buf, 3, MPI_CHAR, 1, MPI_COMM_WORLD)

scatter

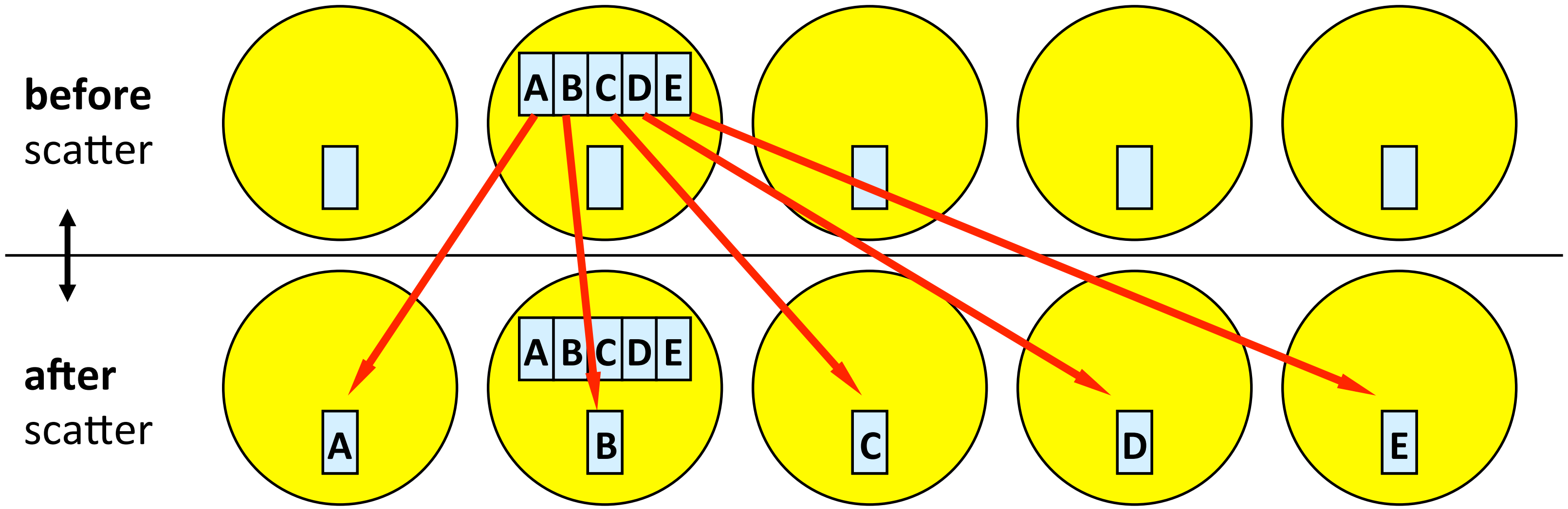
C/C++
```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

Fortran
```
MPI_SCATTER(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm,ierror)
```

python
```
comm.Scatter(sendbuf or None, recvbuf, int root=0)
recvobj = comm.scatter(sendobj or None, int root=0)
```

sendbuf, sendcount, sendtype needed only by root process (ignored at all other processes)

sendcount for only one message



e.g., root=1

MPI_Scatter (sbuf, 1, MPI_CHAR, rbuf, 1, MPI_CHAR, 1, MPI_COMM_WORLD)

gather

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

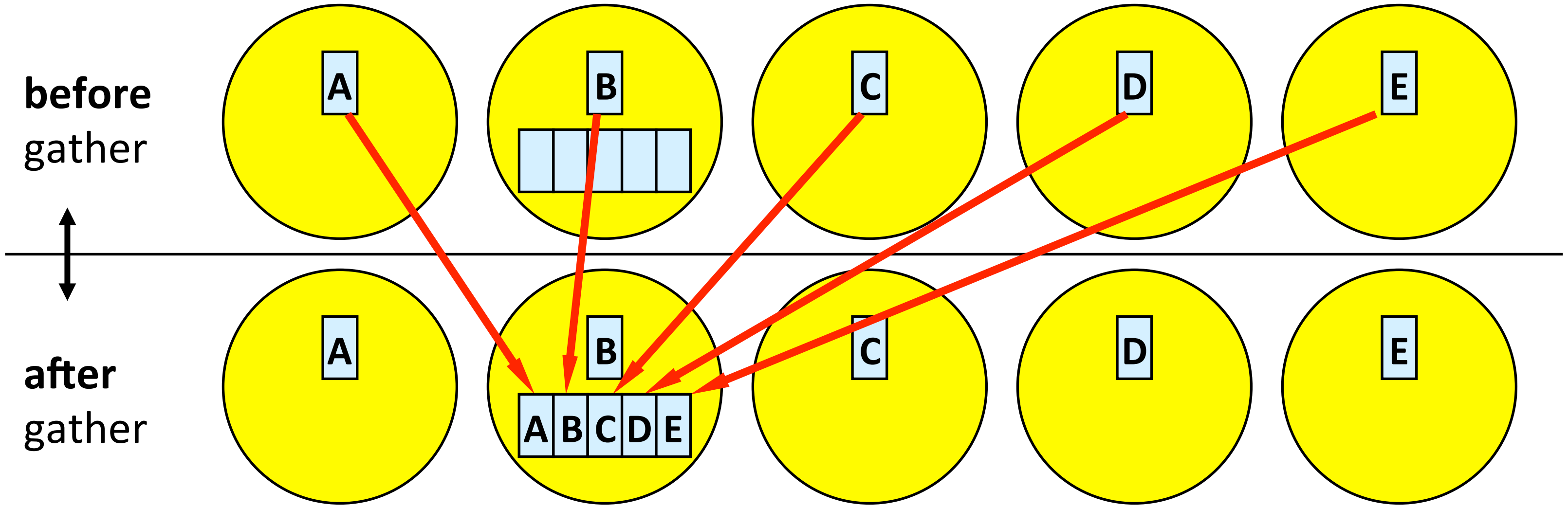
C/C++
```
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

Fortran
```
MPI_GATHER (sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm,ierror)
```

python
```
comm.Gather (sendbuf, recvbuf or None, int root=0)
recvobj = comm.gather (sendobj, int root=0)
```

recvcount for only one message

recvbuf, recvcount, recvtype needed only by root process (ignored at all other processes)



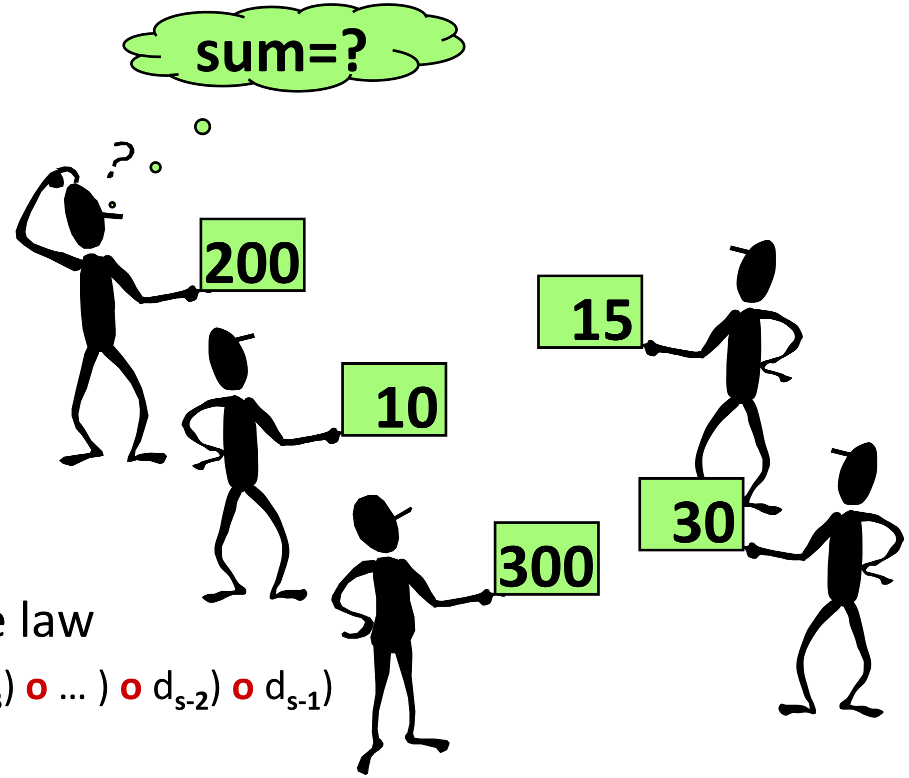
e.g., root=1

MPI_Gather (sbuf, 1, MPI_CHAR, rbuf, 1, MPI_CHAR, 1, MPI_COMM_WORLD)

global reduction operations

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- perform a **global reduction operation** across all members of a group
- d_0 **o** d_1 **o** d_2 **o** d_3 **o** … **o** d_{s-2} **o** d_{s-1}
  - d_i = data in process rank i
    - single variable or
    - vector
  - **o** = associative operation
  - examples:
    - global **sum** or product
    - global maximum or minimum
    - global user-defined operation
- floating point rounding may depend on usage of associative law
  - [(d_0 **o** d_1) **o** (d_2 **o** d_3)] **o** [… **o** (d_{s-2} **o** d_{s-1})] versus (((((d_0 **o** d_1) **o** d_2) **o** d_3) **o** …) **o** d_{s-2}) **o** d_{s-1})
  - partial sums in each process



predefined & user-defined reductions

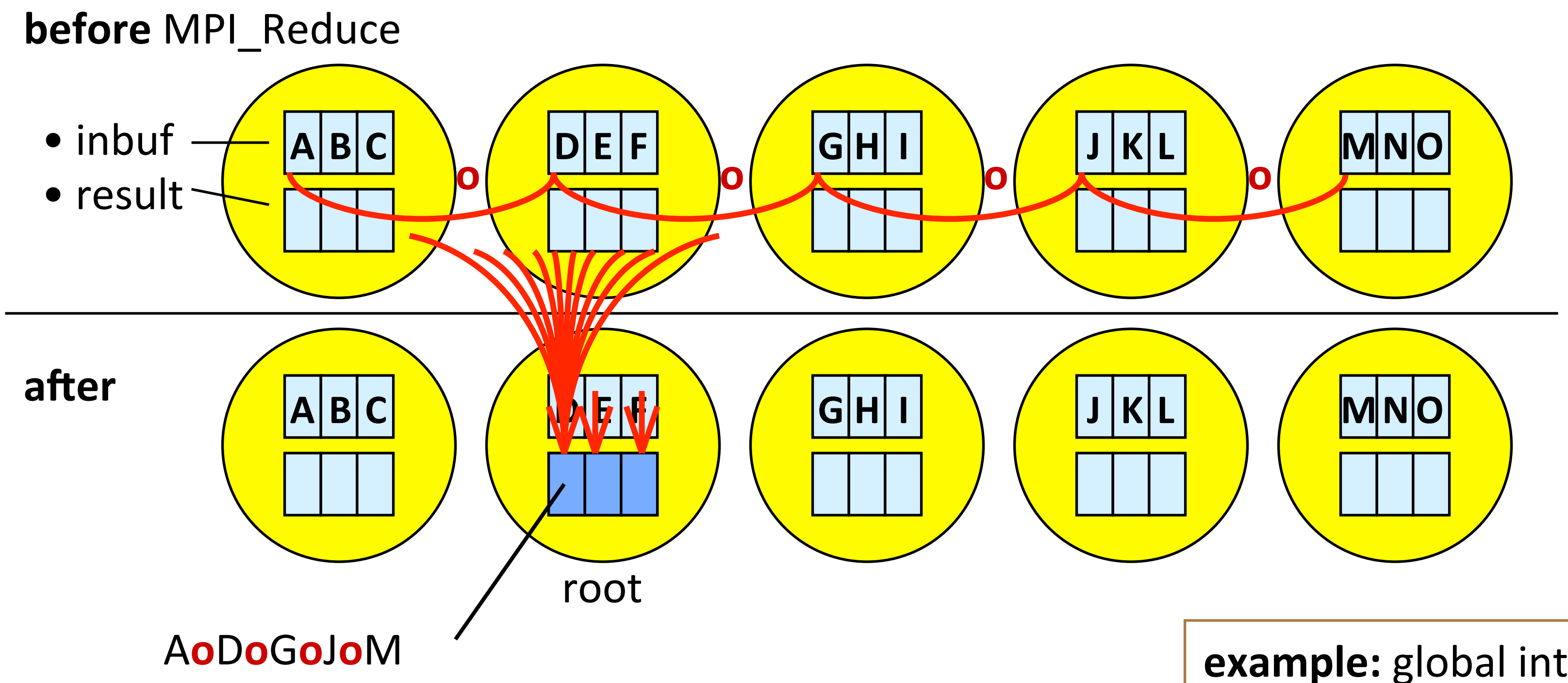
SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

- **reduction operations**
  - predefined (see table)
  - user-defined

- **user-defined operation**
  - associative
  - performs the operation:
    vector_A **o** vector_B
  - syntax: → MPI standard
  - registering:
    MPI_OP_CREATE
    (FUNC, COMMUTE,OP)
  - COMMUTE tells whether FUNC
    is commutative or not

MPI_Reduce



result is only placed
in the resultbuf
at the root process

example: global integer sum at root = 0
sum of all inbuf values should be returned in resultbuf

```
MPI_Reduce(&inbuf, &resultbuf, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
```
C/C++

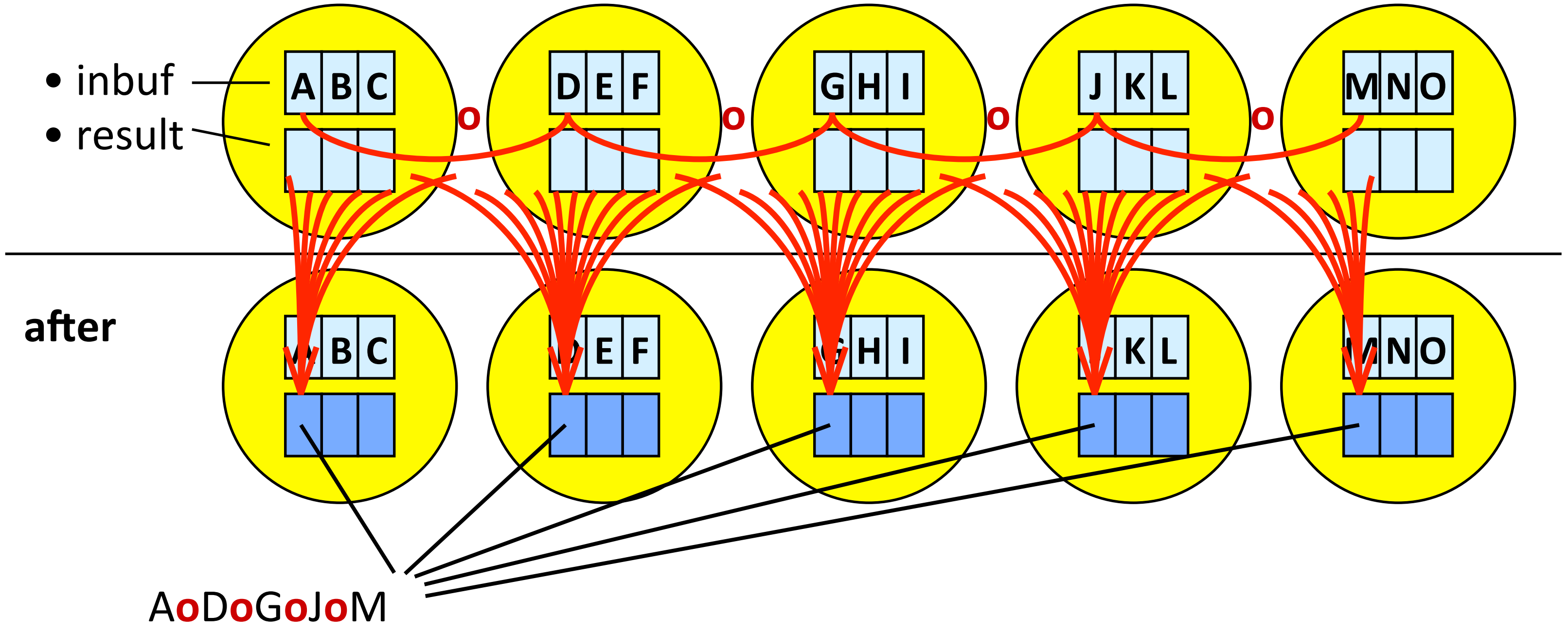
```
CALL MPI_REDUCE(inbuf,resultbuf,1,MPI_INTEGER,MPI_SUM,root,MPI_COMM_WORLD,ierror
```
Fortran

python
```
comm_world = MPI.COMM_WORLD
snd_buf = np.array(value, dtype=np.intc)
resultbuf = np.empty((), dtype=np.intc)
comm_world.Reduce(snd_buf,resultbuf,op=MPI.SUM)
```

op=MPI.SUM and root=0 are defaults

variants: MPI_Allreduce

before MPI_Allreduce

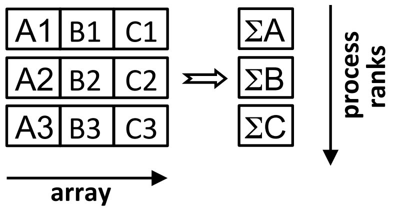


MPI_Allreduce

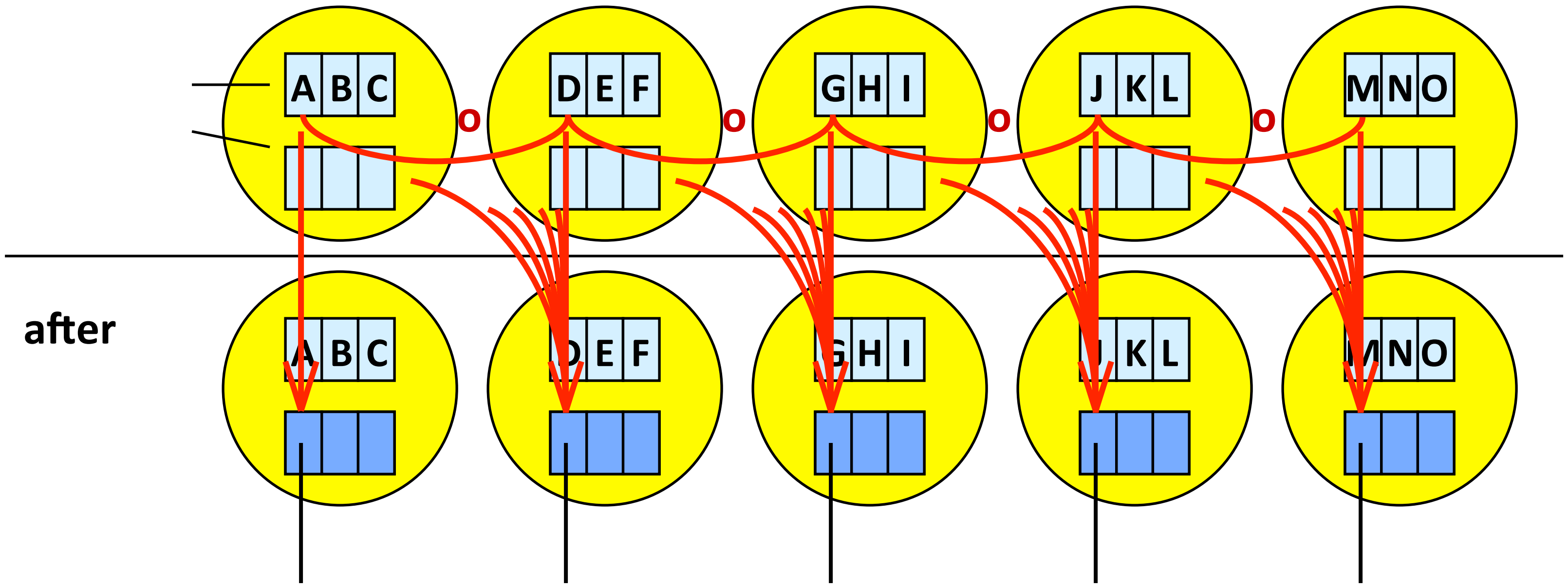
- no root
- **result in all processes**

MPI_Reduce_scatter_block and **MPI_Reduce_scatter**

- result vector of the reduction operation is scattered to the processes into the result buffers



MPI_Scan & MPI_Exscan



MPI_Scan:	A	AoD	AoDoG	AoDoGoJ	AoDoGoJoM
MPI_Exscan:	-	A	AoD	AoDoG	AoDoGoJ

done in parallel

prefix reduction

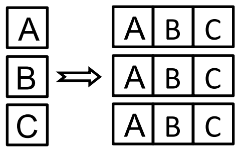
result at process with rank i :=

reduction from rank 0 to rank i

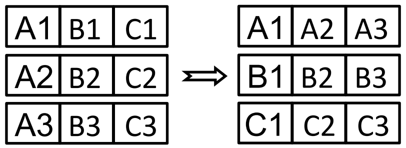
reduction from rank 0 to rank i-1

other collective routines

- **MPI_Allgather** → similar to MPI_Gather, but all processes receive the result vector



- **MPI_Alltoall** → each process sends messages to all processes



- **MPI_.........v** (Gather**v**, Scatter**v**, Allgather**v**, Alltoall**v**, Alltoall**w**)

  → each message has a different count and displacement

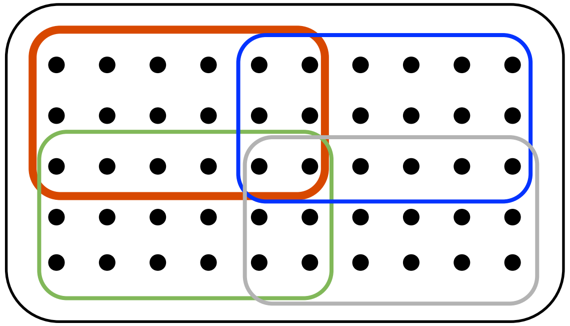
  → array of counts and array of displs (Alltoall**w**: also array of types)

  → **does not scale** to thousands of MPI processes

  → **recommendation** → **try to use data structures with the same communication size on all ranks**

nonblocking collective routines

- **MPI_I……… nonblocking** variants of all collective communication routines
  MPI_Ibarrier, MPI_Ibcast, …
- nonblocking collective operations do not match with blocking collective operations
- collective initiation and completion are separated
- **MPI_I…** calls are **local** (i.e., not synchronizing),
  whereas the **corresponding MPI_Wait** collectively **synchronizes**
  in same way as corresponding blocking collective procedure
- may have multiple outstanding collective communications on same communicator
- ordered initialization on each communicator
- **opportunities with nonblocking collectives**
  → several collective communications on several overlapping communicators
  → overlap computation and communication (for this, progress is needed)



exercise: allreduce

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- rewrite the ring program
  use the MPI global reduction to get the global sum of all ranks of the processes in the ring
  print it from all processes

- the pass-around the ring communication loop must be substituted
  by one call to the MPI collective reduction routine

- please look into the MPI standard to see the argument list of MPI_Allreduce
  - go to the end of the MPI standard, i.e., MPI standard – function index
  - click on the underlined reference: MPI_Allreduce …… 239 …… (in MPI-4.0)
  - python see e.g., mpi4py.MPI.Comm - mpi4py.MPI.Comm.Allreduce

new feature in
MPI-4.0
large count variants: **_c**

cd ~/##/MPI/**C**/4_allreduce/
cd ~/##/MPI/**F**/4_allreduce/
cd ~/##/MPI/**P**/4_allreduce/

allreduce-skel* } see: solutions/

additional exercise:
rewrite with MPI_SCAN (partial sums)
mpirun –n 4 ./a.out | sort -n

solution: allreduce C/C++

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
#include <stdio.h>
#include <mpi.h>


int main (int argc, char *argv[])
{
  int my_rank, size;
  int sum;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  /* Compute sum of all ranks. */
  MPI_Allreduce (&my_rank, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

  printf ("PE%i:\tSum = %i\n", my_rank, sum);

  MPI_Finalize();
}
```

solution: allreduce **Fortran**

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
PROGRAM allreduce

  USE mpi_f08

  IMPLICIT NONE

  INTEGER :: my_rank, size
  INTEGER :: sum

  CALL MPI_Init()
  CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
  CALL MPI_Comm_size(MPI_COMM_WORLD, size)

  ! Compute sum of all ranks.
  CALL MPI_Allreduce(my_rank, sum, 1, MPI_INTEGER, MPI_SUM, MPI_COMM_WORLD)

  WRITE(*,*) "PE", my_rank, ": Sum =", sum

  CALL MPI_Finalize()

END PROGRAM
```

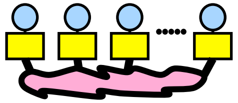
solution: allreduce python

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

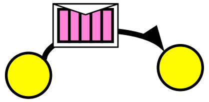
```
#!/usr/bin/env python3

from mpi4py import MPI
import numpy as np

comm_world = MPI.COMM_WORLD
my_rank = comm_world.Get_rank()
size = comm_world.Get_size()

snd_buf = np.array(my_rank, dtype=np.intc)
sum = np.empty((), dtype=np.intc)

# Compute sum of all ranks.
comm_world.Allreduce(snd_buf, (sum,1,MPI.INT), op=MPI.SUM )

# Also possible
# comm_world.Allreduce((snd_buf,1,MPI.INT), (sum,1,MPI.INT), op=MPI.SUM)
# Shortest version in python is
# comm_world.Allreduce(snd_buf, sum)

print(f"PE{my_rank}:\tSum = {sum}")
```

collective communication

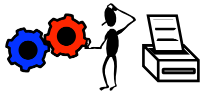
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



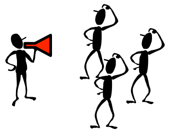
- **messages and point-to-point communication**
  - the MPI processes can communicate



- **non-blocking communication**
  - to avoid idle times, serializations, and deadlocks



- **collective communication**
  - e.g. broadcast, reduction, …



- **MPI basics – summary**

Thank you for your attention!

http://sctrain.eu/

Univerza v Ljubljani

TECHNISCHE UNIVERSITÄT WIEN

CINECA

VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Co-funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.