SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Introduction to the Message Passing Interface (MPI)

Claudia Blaas-Schenner

VSC Research Center, TU Wien

06/2021

Univerza v Ljubljani

TECHNISCHE UNIVERSITÄT WIEN

CINECA consorzio interuniversitario

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.

hands-on labs MPI

- `cp -a ~cblass/MPI .` → copy the MPI exercises
- `cd ~/MPI` → go to the folder

- `module load foss/2019a` → @viz.hpc.fs.uni-lj.si
  → GCC 8.2.0 & OpenMPI/3.1.3

- `mpicc program.c`

- **SLURM** queuing system → MPI exercises can also be run interactively (error messages can be ignored)
- `sbatch job.sh` → submit (`mpirun -n # ./a.out`)
- `squeue -u $USER` → check
- `scancel JOB_ID` → cancel
- `slurm-*.out` → output

acknowledgements

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**

goals and scope of MPI

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- MPI‘s prime goals
  - provide a message-passing interface
  - provide source-code portability
  - allow efficient implementations
- MPI also offers
  - a great deal of functionality
  - support for heterogeneous parallel architectures
- MPI-2.0, 2.1, 2.2, 3.0, 3.1
  - important additional functionality, fit on new hardware principles
  - deprecated MPI routines – with MPI-3.0 some deprecated features removed

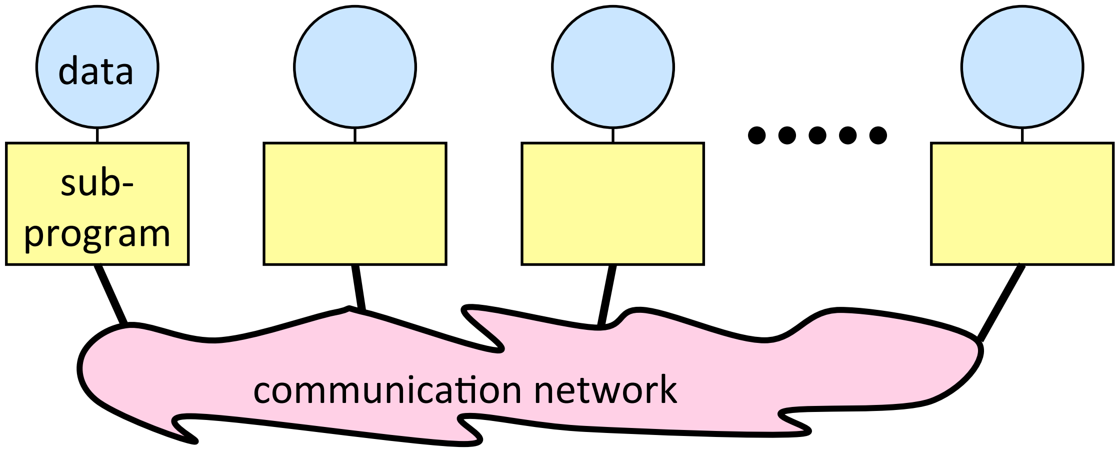
current version (June 9, 2021)

MPI-4.0

available libraries are for MPI-3.1

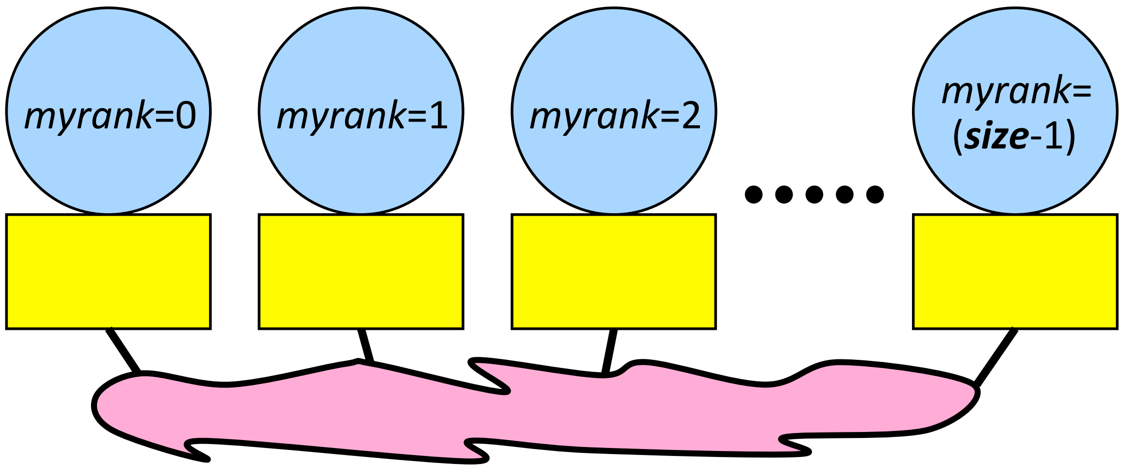
Each processor in a message passing program runs a ***sub-program***:

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



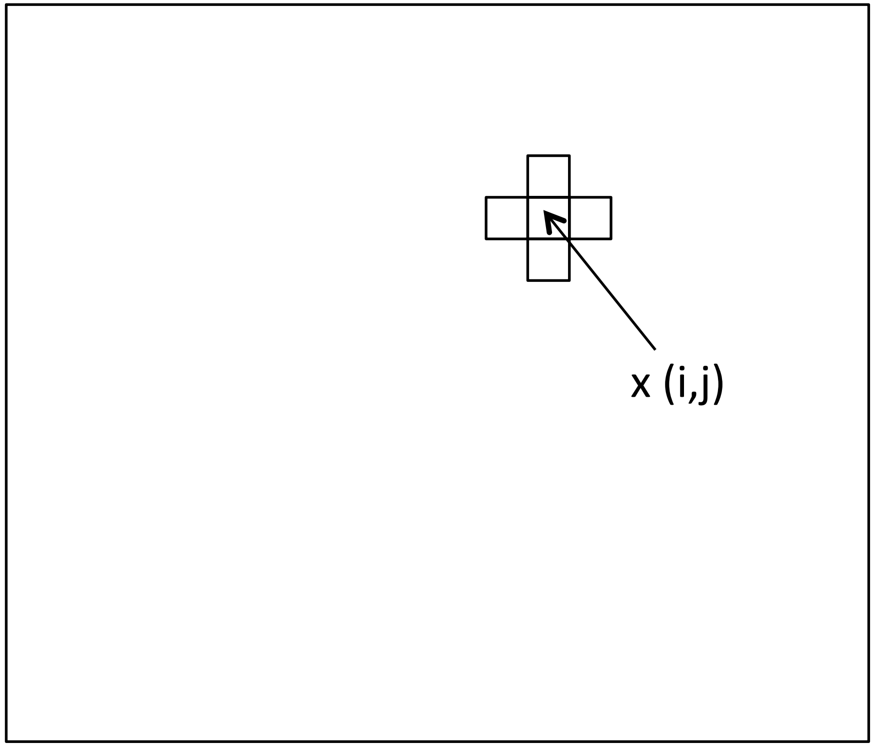
data & work distribution

- 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***



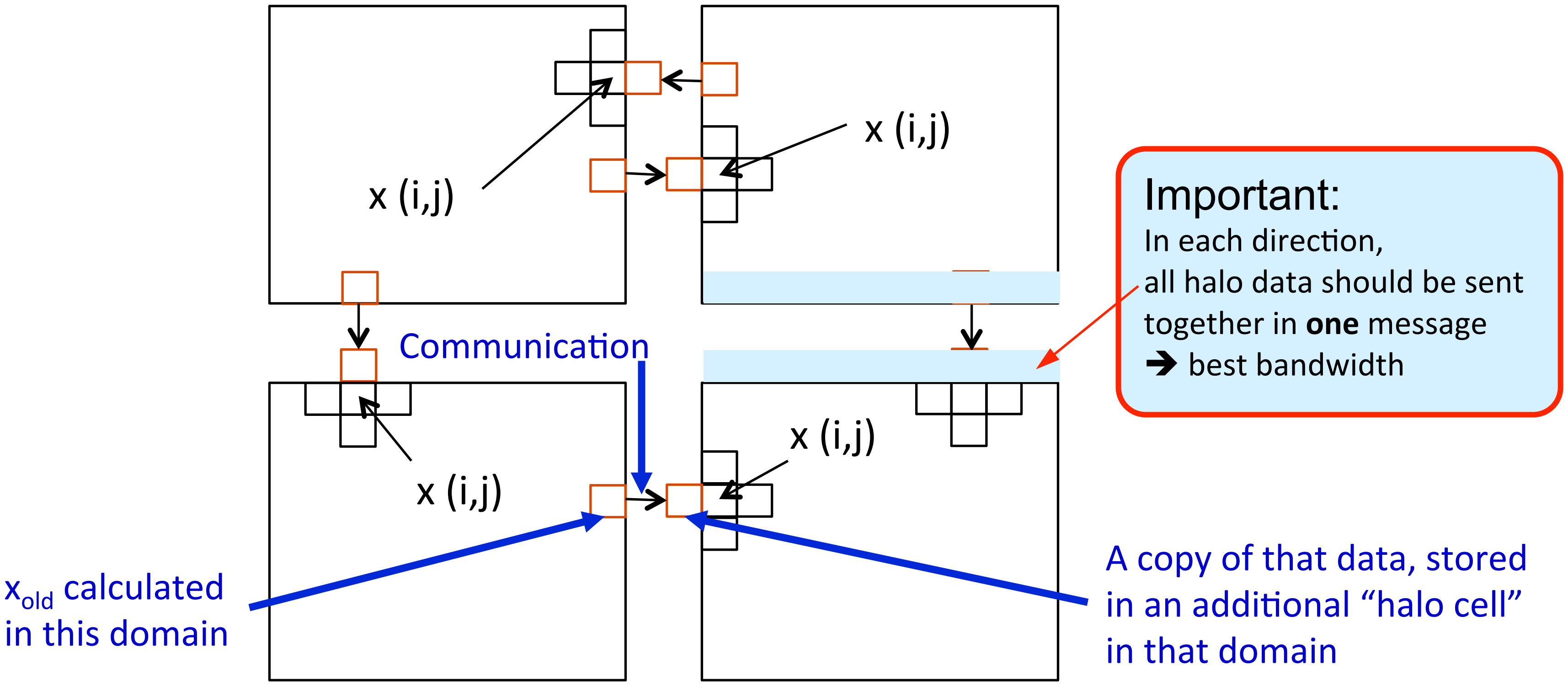
domain decomposition – serial

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



domain decomposition – parallel

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



MPI process model

- 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

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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

C/C++

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

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

Fortran

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

python

! not part of the MPI standard !

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

initializing & finalizing MPI

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

C/C++

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

Fortran

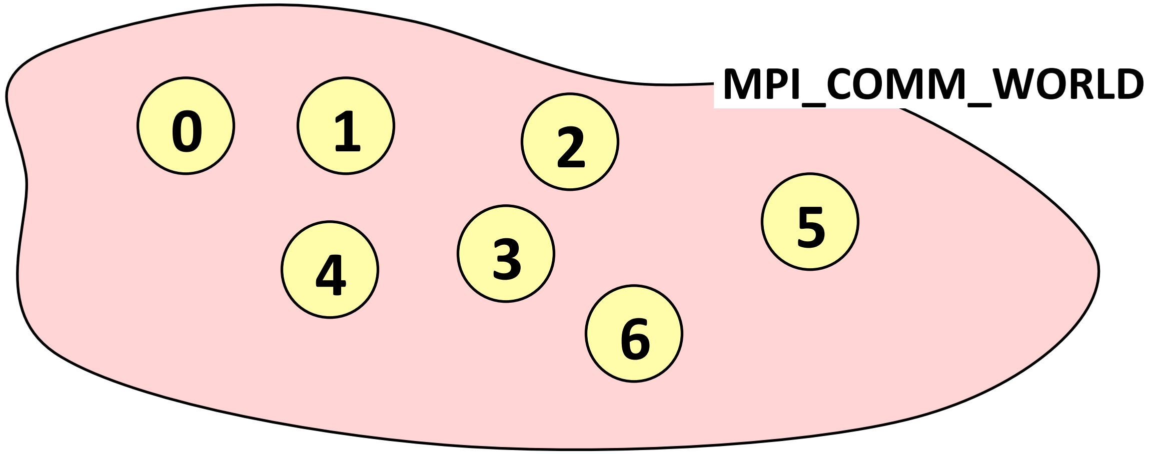
```
program xxxxx
use mpi_f08
implicit none

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

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 & size

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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

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

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

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

exercise: Hello world!

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- 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
- 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”
- why is the sequence of the output non-deterministic?

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

solution: Hello world! C/C++

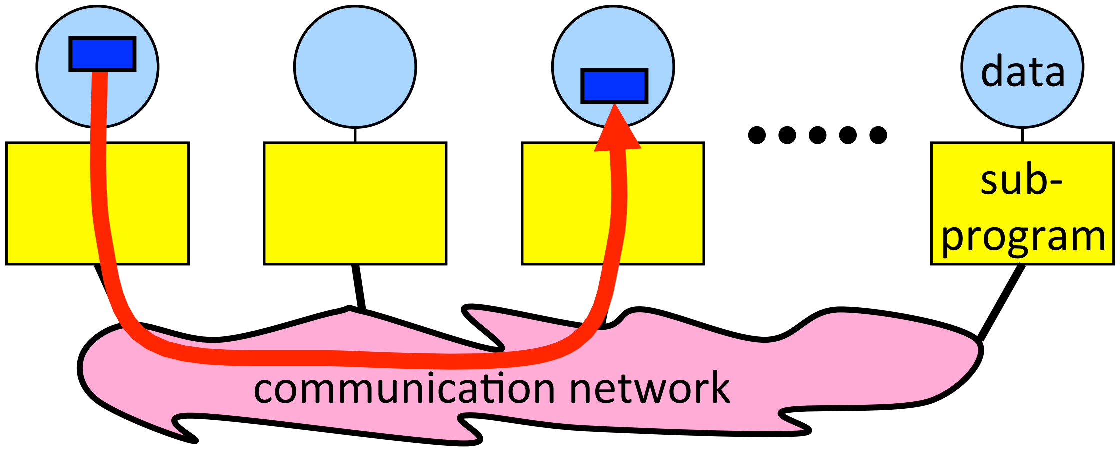
SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
#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();
    return 0;
}
```

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

example: message with 5 integers

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

MPI basic datatypes C/C++

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

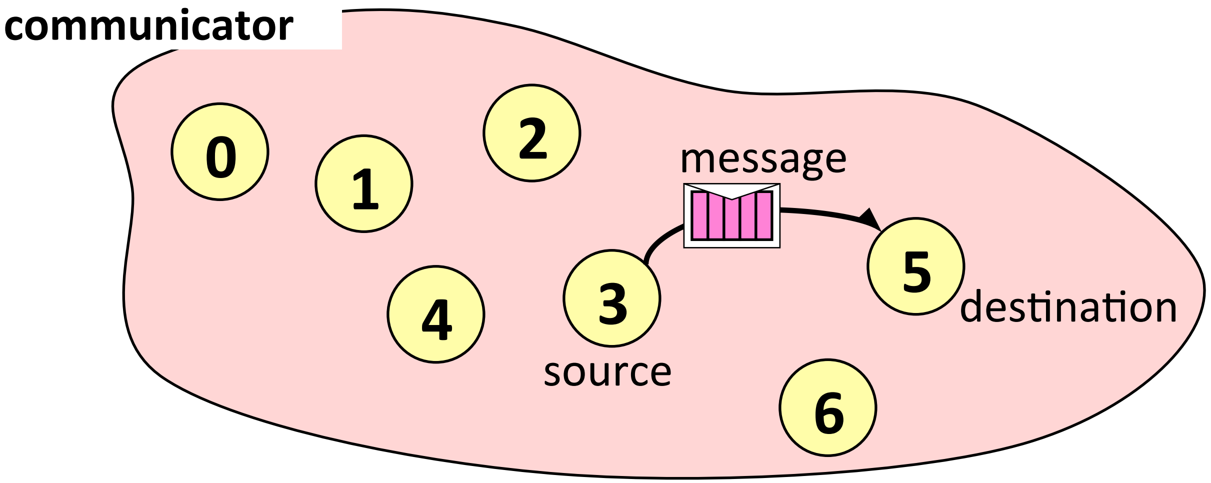
datatype=MPI_INT

declaration of the buffers

int arr[5];

point-to-point communication

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



sending & receiving a message

- sending:

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

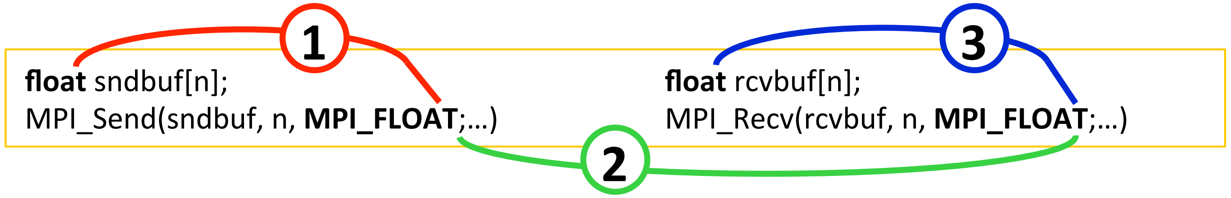
- receiving:

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

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

requirements for p2p communication

- 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

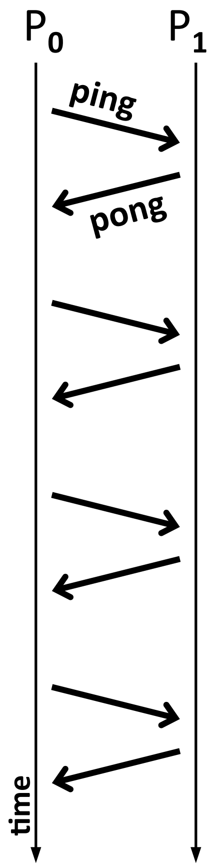
Sender mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Standard send **MPI_SEND**	Either 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

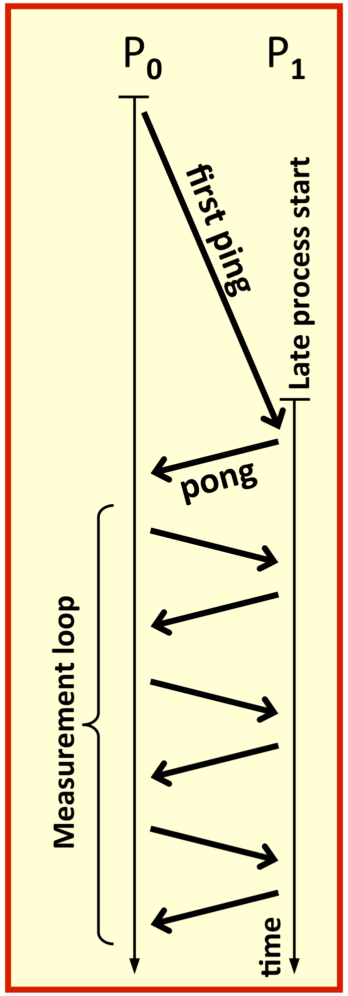
← **debuging**

← **production**

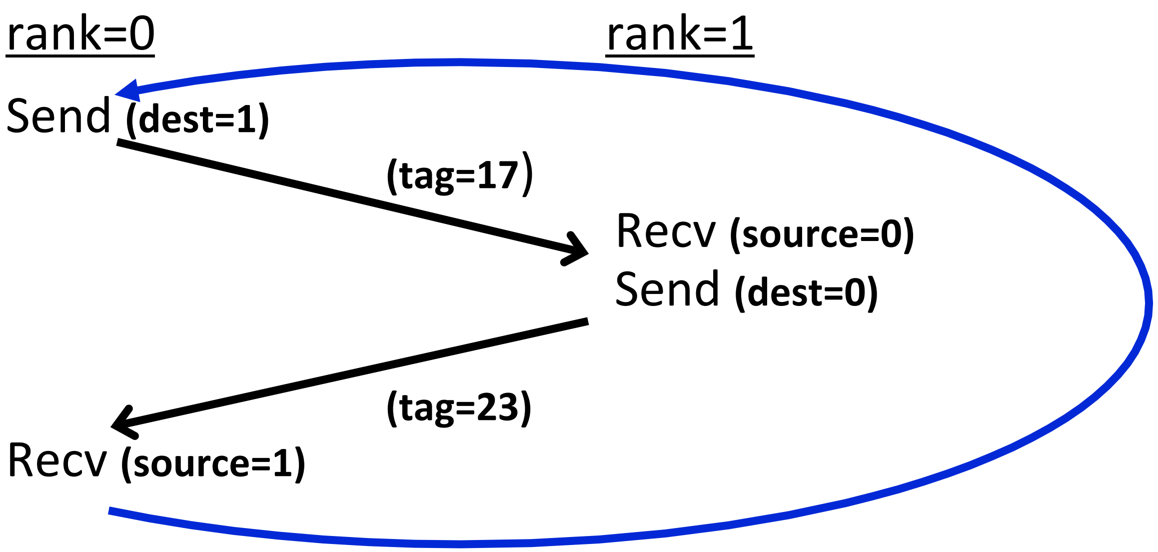
exercise: ping pong

- write a program according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message,
    process 1 sends a message back to process 0 (pong)
- repeat this ping-pong with a loop of length 50
- add timing calls before and after the loop:
- C/C++: *double MPI_Wtime*(void);
- 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





exercise: ping pong




```
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

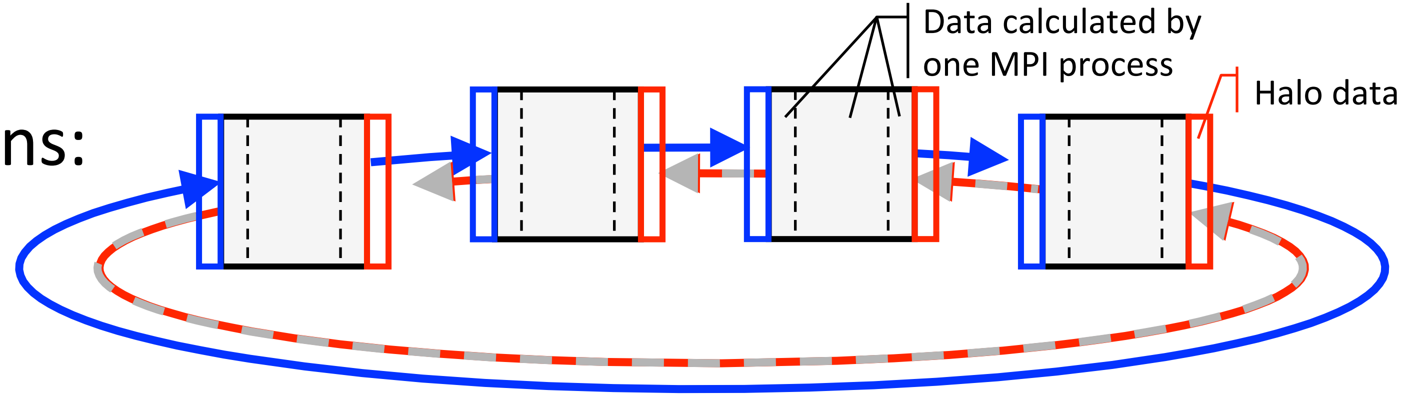
```
start = MPI_Wtime();

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 );
```

nonblocking communication

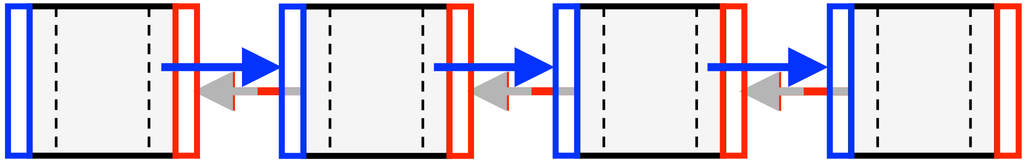
→ **to avoid idle times, serializations and deadlocks**

→ **halo communication**

cyclic boundary conditions:



non-cyclic:



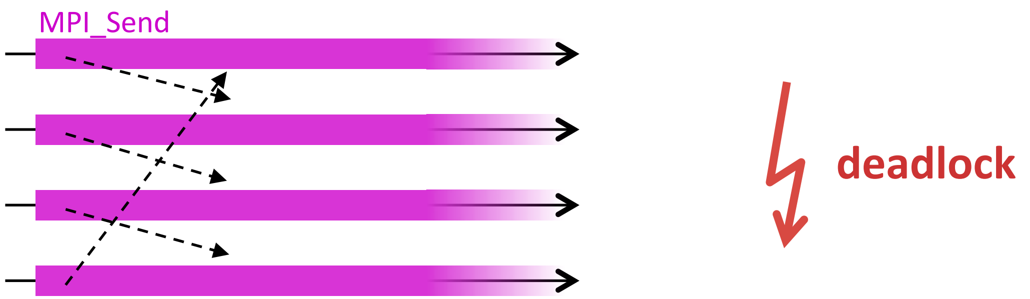
blocking → risk deadlocks & serializations

if the MPI library chooses the synchronous protocol

timelines of all processes

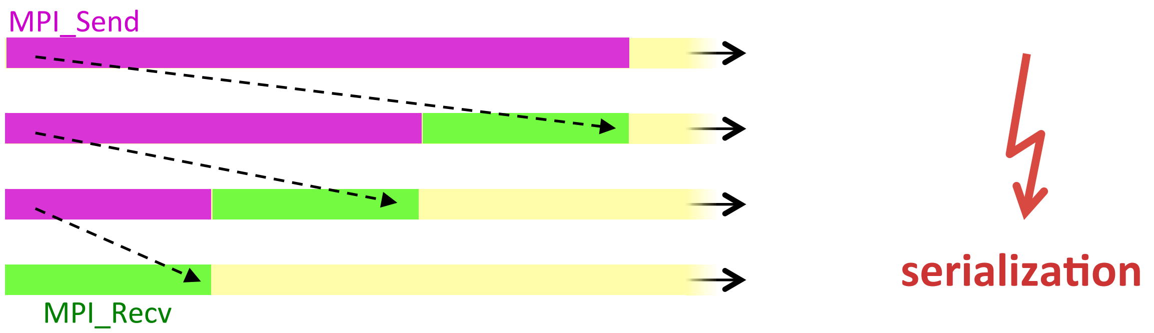
cyclic boundary:

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



non-cyclic boundary:

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



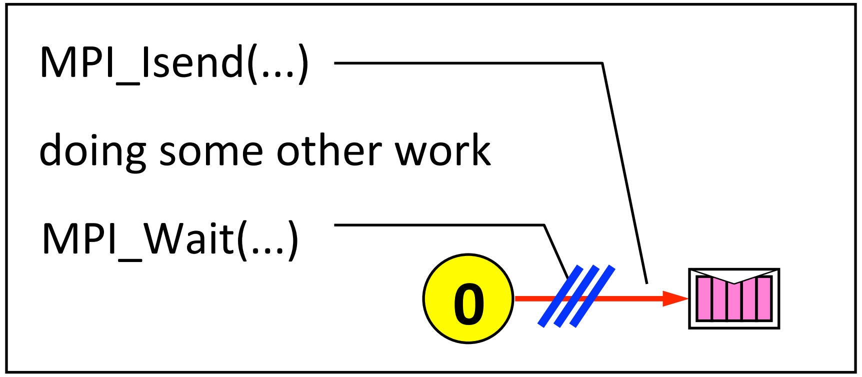
nonblocking communication

separate communication into **three phases**:

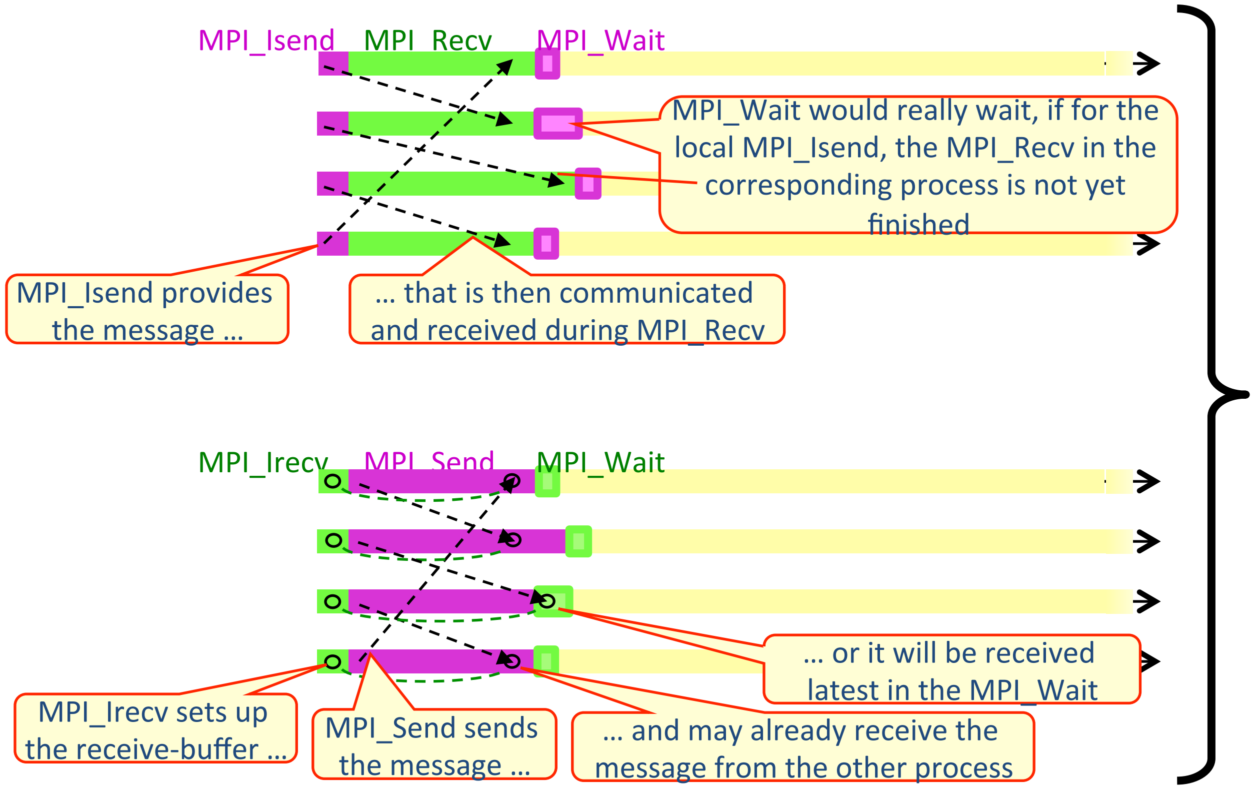
- initiate nonblocking communication
  - routine name starting with MPI_**I**…
  - incomplete
  - → it is local, returns immediately,
    returns independently of any other process' activity

→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



nonblocking timelines



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, in C/C++: MPI_Request, Fortran: TYPE(MPI_Request)
- the value (INTEGER)
  - **is generated** by a nonblocking communication routine
  - **is used** (and freed) in the MPI_WAIT routine

nonblocking synchronous send

→ ss for debugging only

→ s for production code

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

- 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)

nonblocking receive

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

MPI_Wait[INOUT] &request_handle, &status)
```

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

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

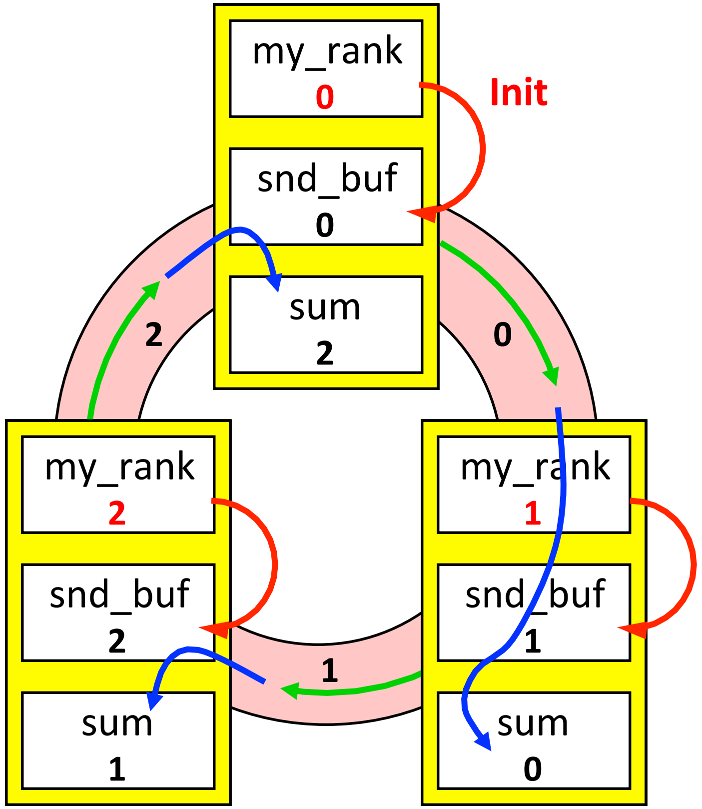
completion


```
MPI_Wait( &request_handle, &status);
MPI_Test( &request_handle, &flag, &status);
```

- one must
  - WAIT or
  - loop with TEST until request is completed, i.e., flag == 1 or .TRUE.
- multiple nonblocking communications (several request handles): MPI_[Wait|Test]any, MPI_[Wait|Test]all, MPI_[Wait|Test]some

exercise: ring

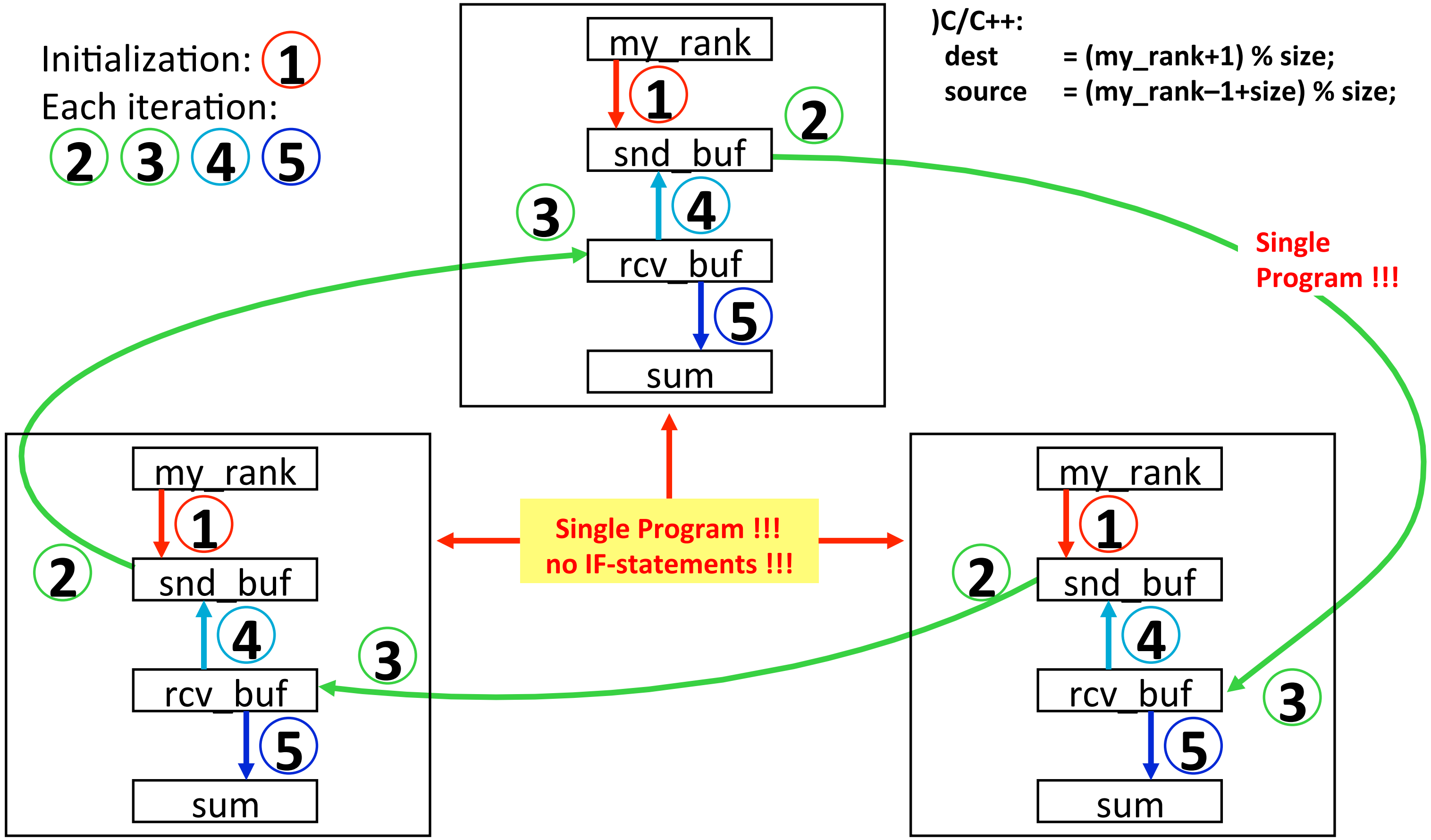
- a set of processes are arranged in a ring
- (1) each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*
- (2) each process passes this on to its neighbor on the right
- (3) (4) each processor calculates the sum of all values
- (5) repeat (2) - (5) with “size” iterations (size = number of processes), i.e.
- each process calculates sum of all ranks
- use nonblocking MPI_Issend
  - to avoid deadlocks
  - to verify the correctness, because blocking synchronous send will cause a deadlock



exercise: ring

Initialization: 1
Each iteration: 2 3 4 5

)C/C++:
dest = (my_rank+1) % size;
source = (my_rank–1+size) % size;



solution: ring

C/C++


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

Thank you for your attention!

http://sctrain.eu/

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.