# Introduction to the Message Passing Interface (MPI)

# MPI (Message Passing Interface)?

- Standardized message passing library specification (IEEE)
  - for parallel computers, clusters and heterogeneous networks
  - not a specific product, compiler specification etc.
  - many implementations, MPICH, LAM, OpenMPI …

- Portable, with Fortran and C/C++ interfaces.
- Many functions
- Real parallel programming
- Notoriously difficult to debug

# Information about MPI

- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July 18,1997)
- **MPI: The Complete Reference**, Marc Snir and William Gropp et al, The MIT Press, 1998 (2-volume set)
- **Using MPI: Portable Parallel Programming With the Message-Passing Interface** and **Using MPI-2: Advanced Features of the Message-Passing Interface.** William Gropp, Ewing Lusk and Rajeev Thakur, MIT Press, 1999 – also available in a single volume *ISBN 026257134X.*
- **Parallel Programming with MPI**, Peter S. Pacheco, Morgen Kaufmann Publishers, 1997 - *very good introduction.*
- **Parallel Programming with MPI**, Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti. Training handbook from EPCC which can be used together with these slides -

**http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.pdf**

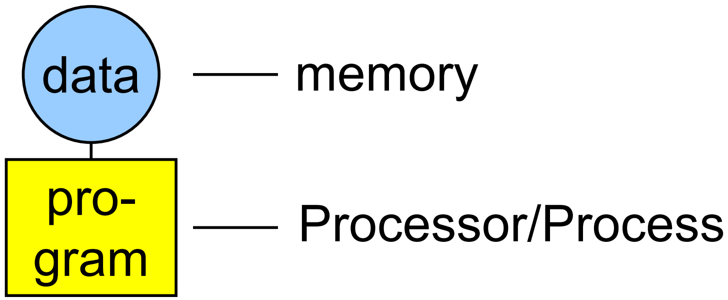
# Compilation and Parallel Start

- Compilation in C: **mpicc -o** *prog prog***.c**
- Compilation in C++: **mpiCC -o** prpg prog**.c** (Bull)
  **mpicxx -o** prog prog**.cpp** (IBM cluster)
- Compilation in Fortran: **mpif77 -o** prog prog**.f**
  **mpif90 -o** prog prog**.f90**
- Executing program with *num* processes:
  **mprun** –n *num* ./pra (Bull)
  **mpiexec -n** *num* ./prg (Standard MPI-2)
- Examples ~course00/MPI-I/examples

*Note: The examples of a chapter are only readable after the end of the practical of that chapter.*

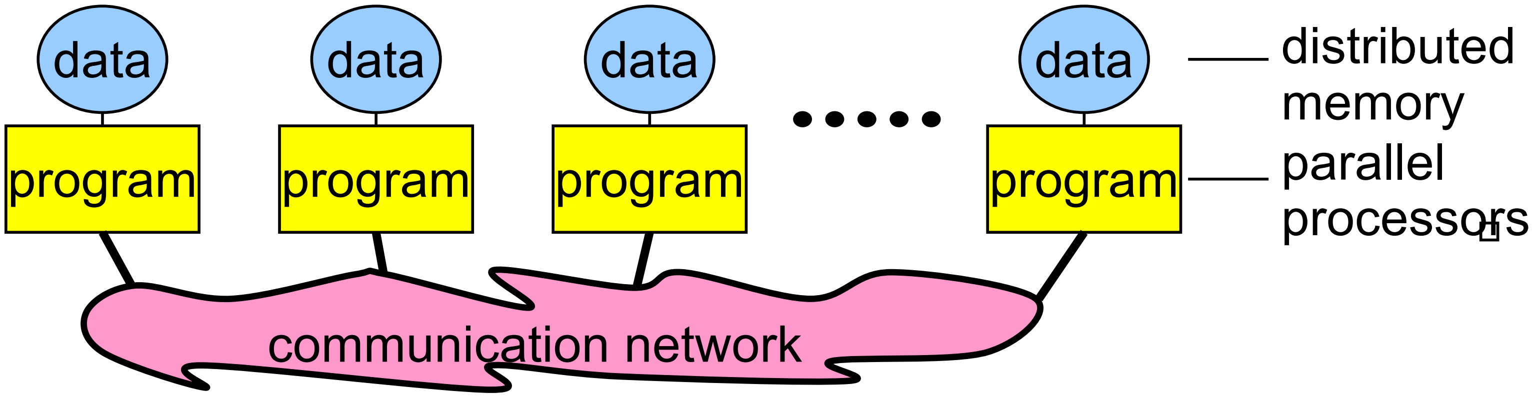
# The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**



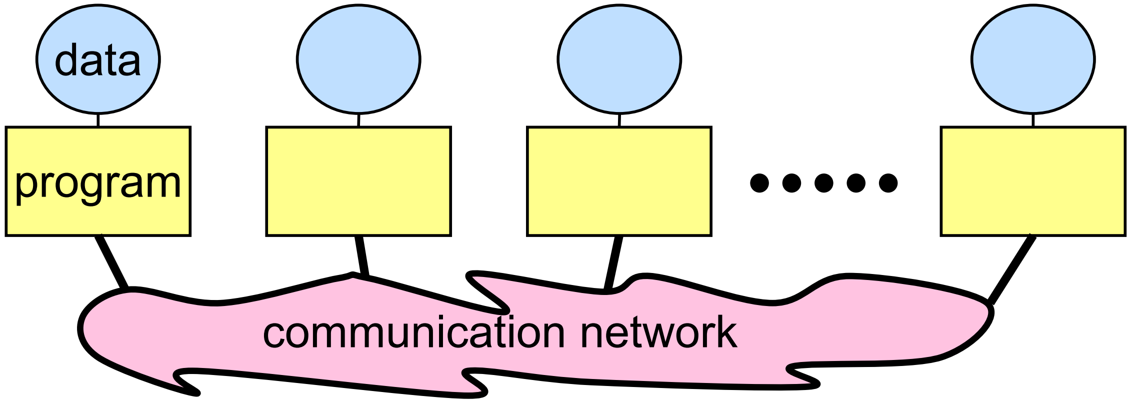

A processor may
run many processes

- **Message-Passing Programming Paradigm**



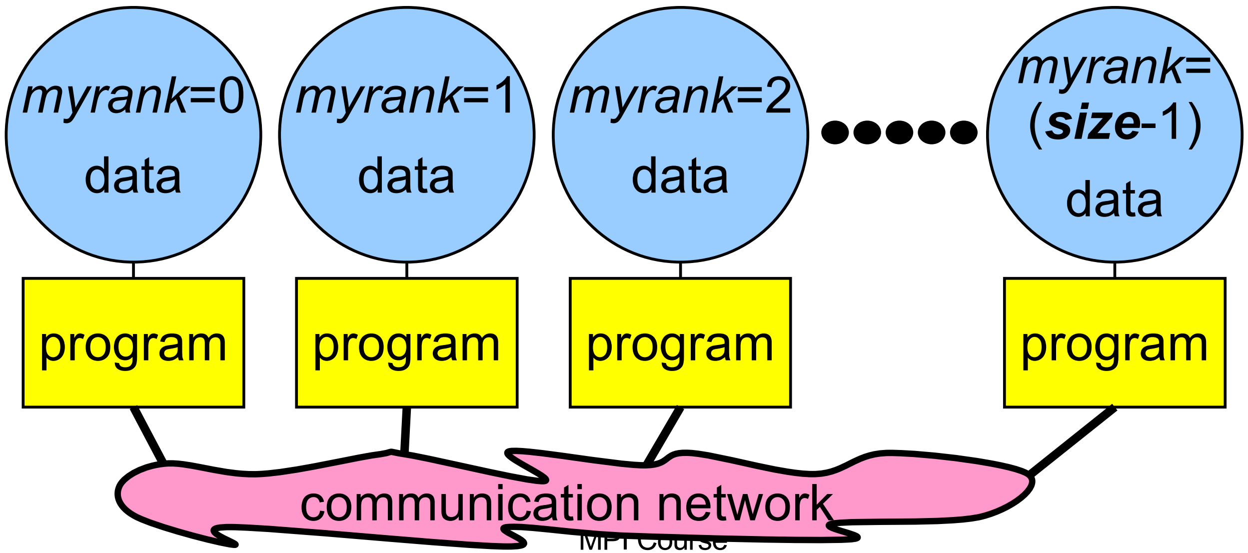
MPI

- A **process** is a program performing a task on a **processor**
- Each processor/process in a message passing program runs a instance/copy of a ***program:***
  - written in a conventional sequential language, e.g., C or Fortran,
  - typically a single program operating of multiple dataset
  - the variables of each sub-program have
    - the same name
    - but different locations (distributed memory) and different data!
    - i.e., all variables are local to a process
  - communicate via special send & receive routines (***message passing***)



# Data and Work Distribution

- To communicate together mpi-processes need identifiers: **rank = identifying number**
- all distribution decisions are based on the ***rank***
  - i.e., which process works on which data



# What is SPMD

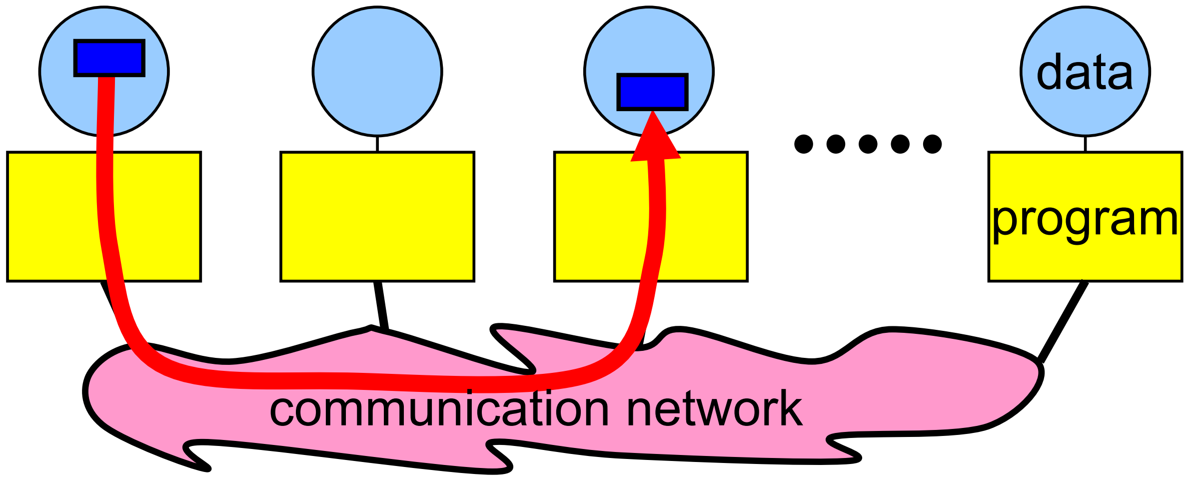
- **S**ingle **P**rogram, **M**ultiple **D**ata
- Same (sub-)program runs on each processor
- MPI allows also MPMD, i.e., **Multiple** Program, ...
  - but some vendors may be restricted to SPMD
  - MPMD can be emulated with SPMD

# Emulation of MPMD

- ```
  main(int argc, char **argv){
      if (myrank < .... /* process should run the ocean model */){
          ocean( /* arguments */ );
      }else{
          weather( /* arguments */ );
      }
  }
  ```
  ```

# Message passing

- Messages are packets of data moving between sub-programs
- 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

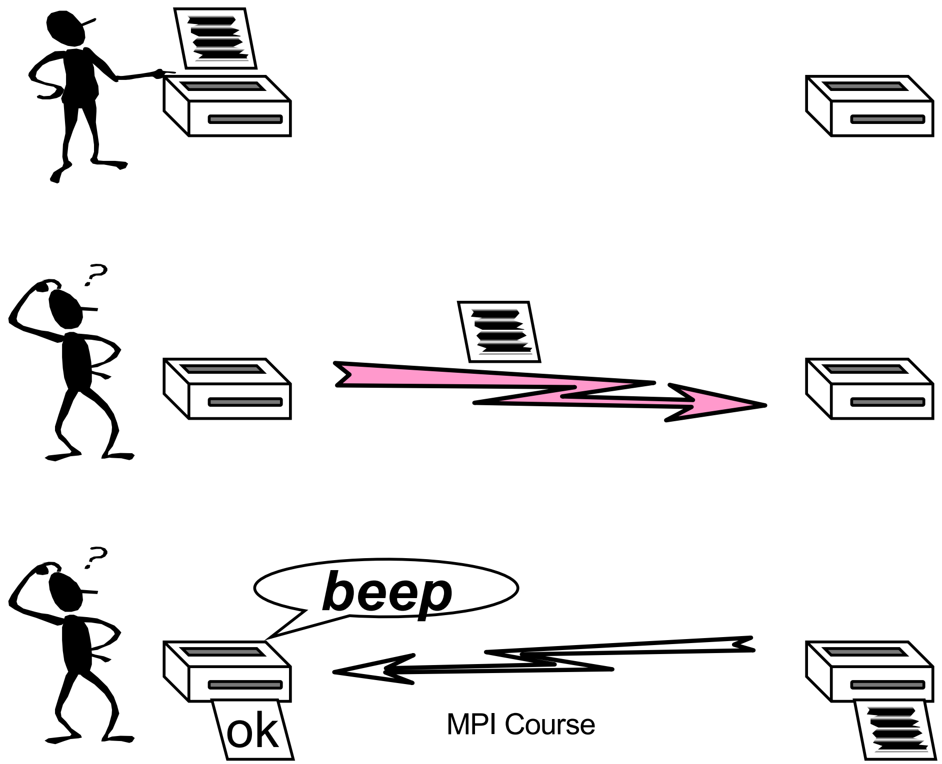


# Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communication:
  - synchronous send
  - buffered = asynchronous send

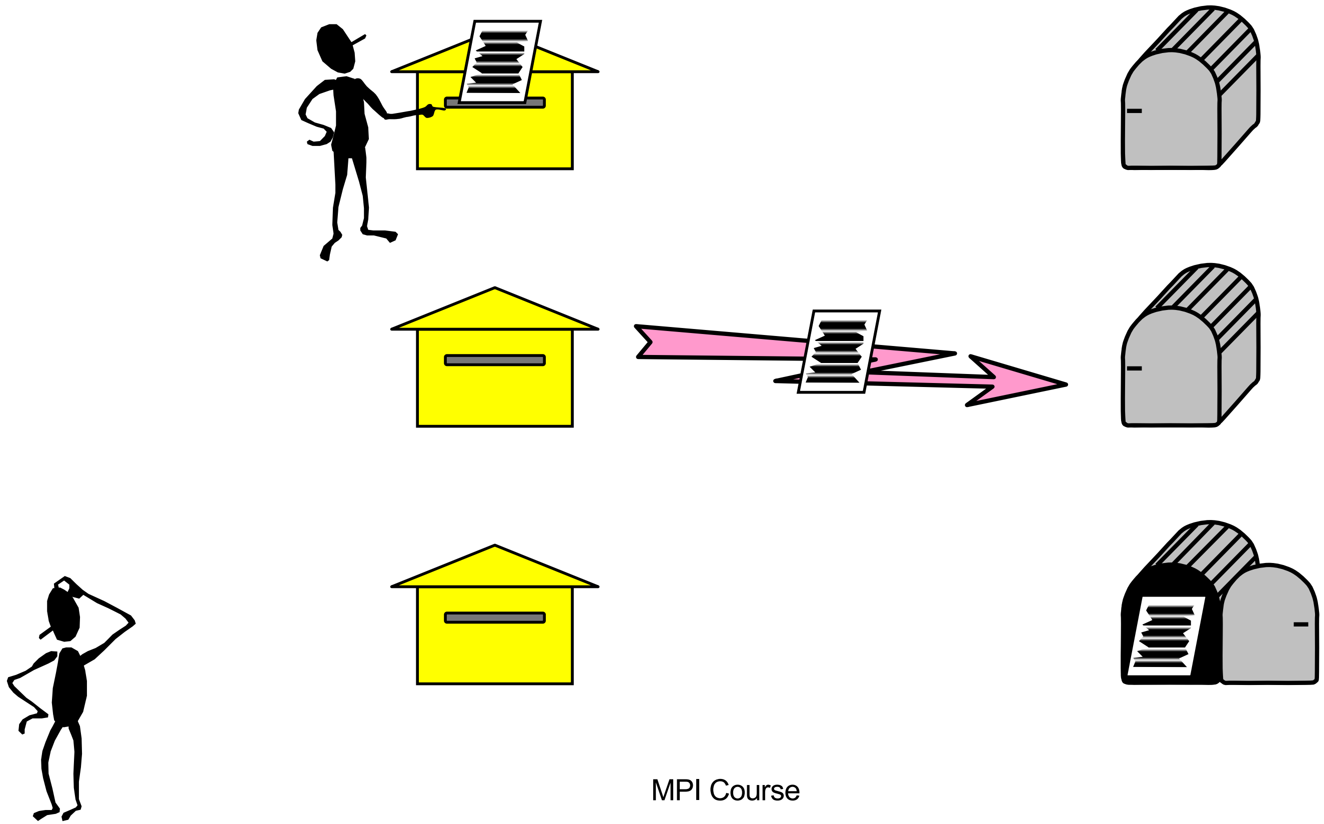
# Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.



# Buffered = Asynchronous Sends

- Only know when the message has left.

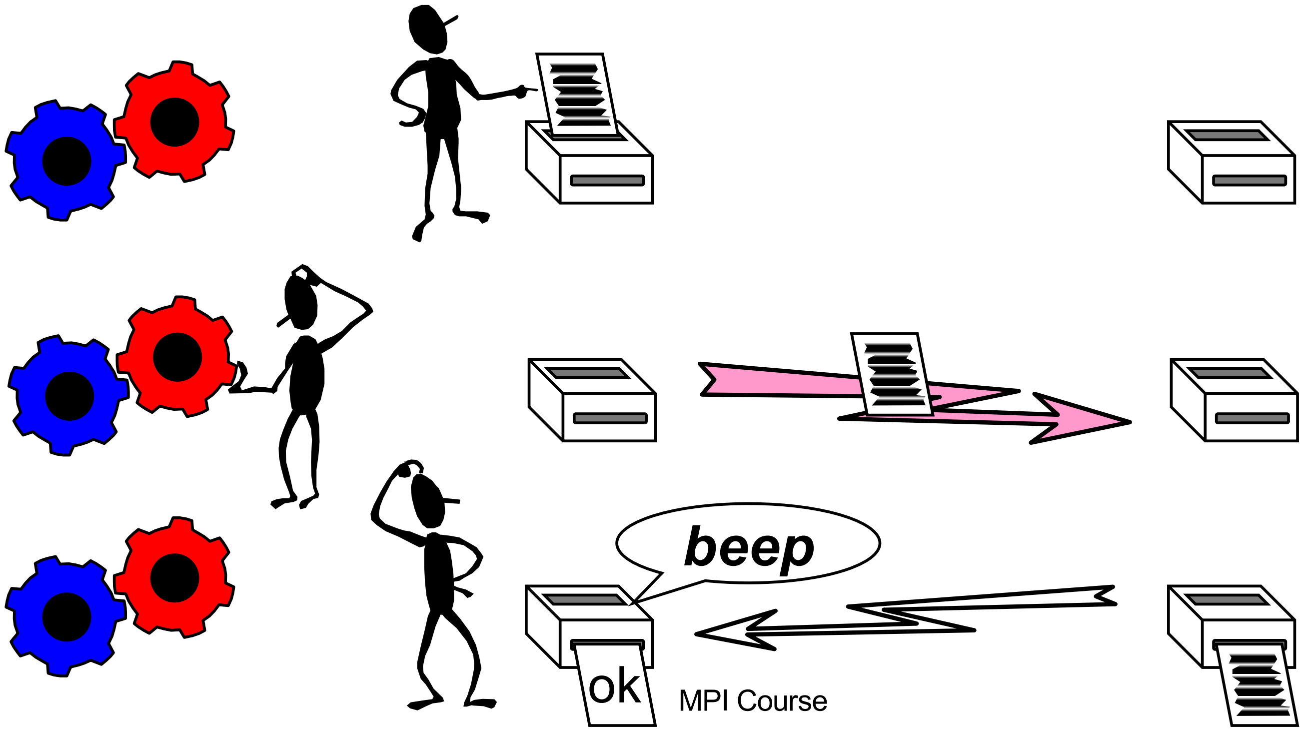


# Blocking Operations

- Some sends/receives may **block** until another process acts:
  - synchronous send operation **blocks until** receive is issued;
  - receive operation **blocks until** message is sent.
- Blocking subroutine returns only when the operation has completed.

# Non-Blocking Operations

- Non-blocking operations return immediately and allow the sub-program to perform other work.

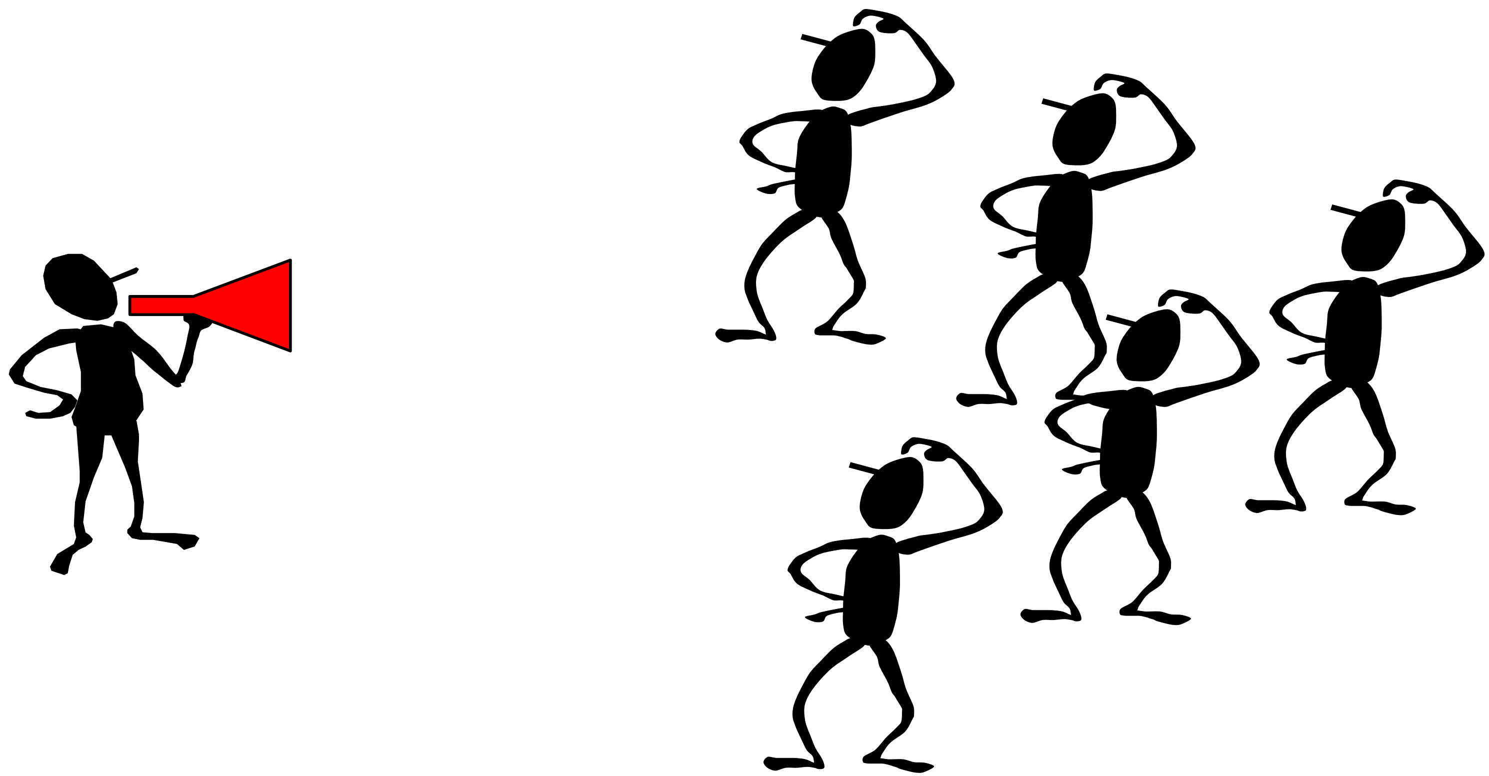


# Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow **optimized internal** implementations, e.g., tree based algorithms

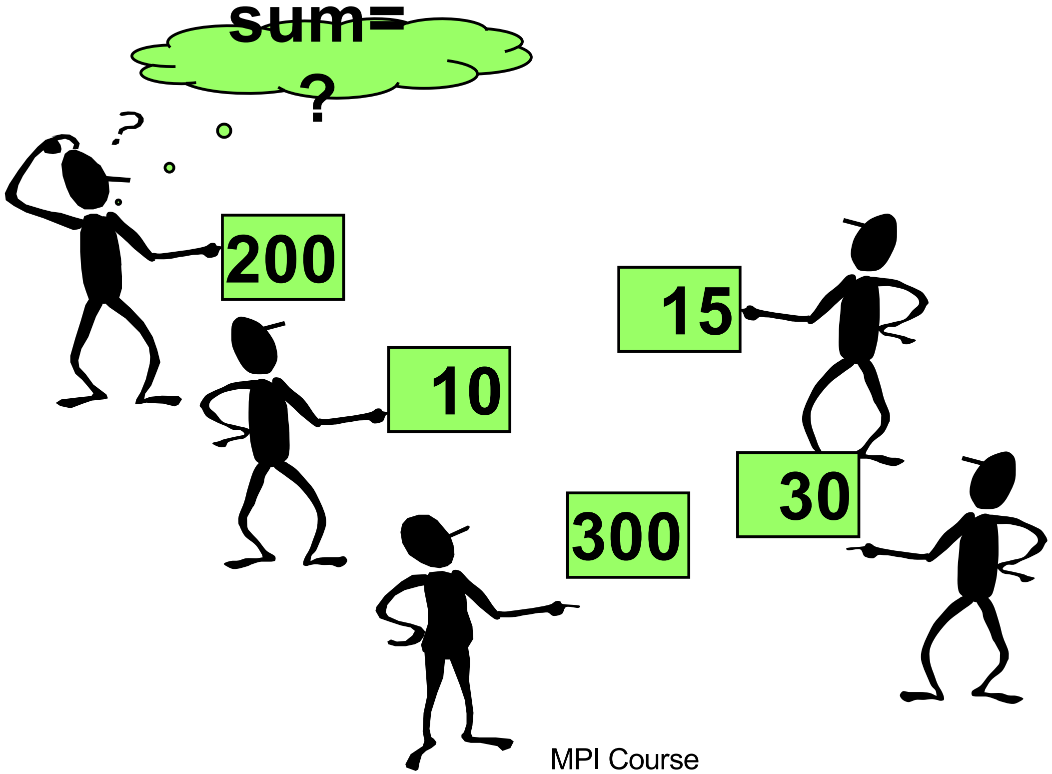
# Broadcast

- A one-to-many communication.



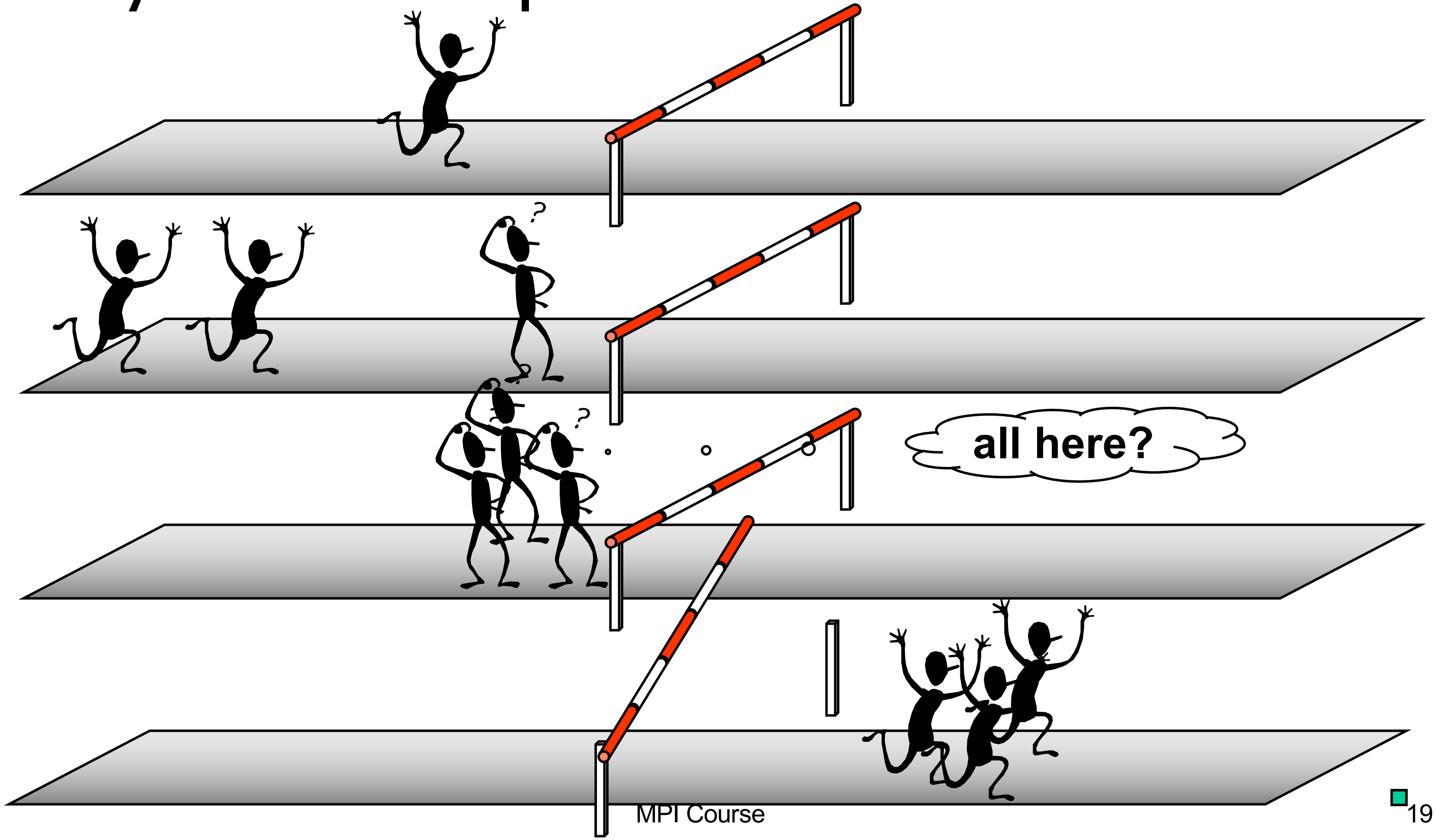
# Reduction Operations

- Combine data from several processes to produce a single result.



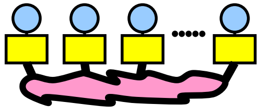
# Barriers

- Synchronize processes.



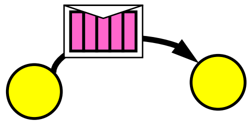
# Chap.2 Process Model and Language Bindings

1. MPI Overview

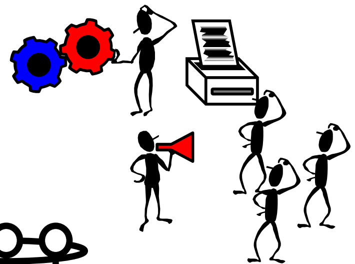


MPI_Init()
MPI_Comm_rank()

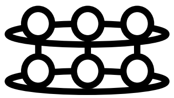
2. **Process model and language bindings**
   - **starting several MPI processes**
3. Messages and point-to-point communication



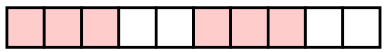
4. Non-blocking communication



5. Collective communication
6. Virtual topologies



7. Derived datatypes



8. Case study

# Header files

- #include <mpi.h>

# MPI Function Format

error = MPI_Xxxxxx(parameter, ...);

MPI_Xxxxxx( parameter, ... );

# Initializing MPI

- C: int MPI_Init( int *argc, char ***argv)

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

- Must be first MPI routine that is called.

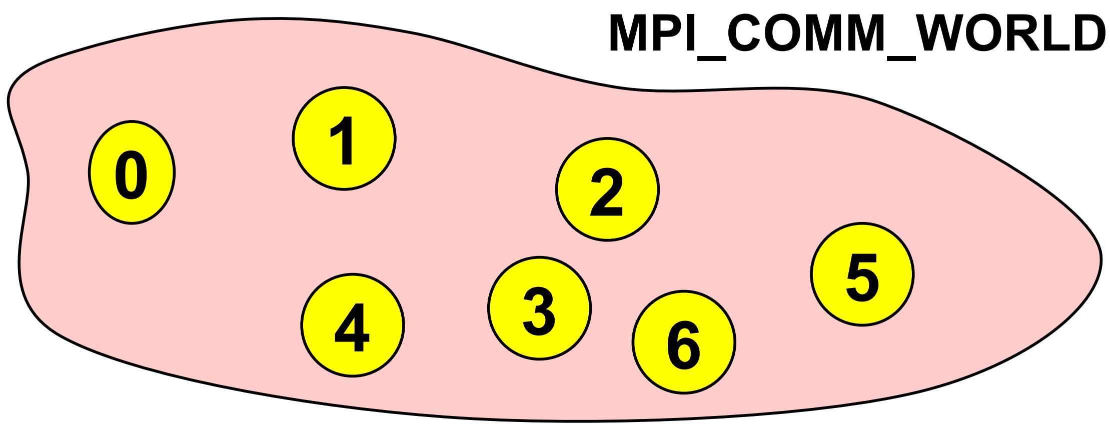
# Starting the MPI Program

- Start mechanism is implementation dependent
  - Most implementations provide mpirun:
    mpirun –np ***number_of_processes*** ./***executable***
  - MPI-2 standard defines mpiexec:
    mpiexec –n ***number_of_processes*** ./***executable***

- The parallel MPI processes exist at least after MPI_Init was called.

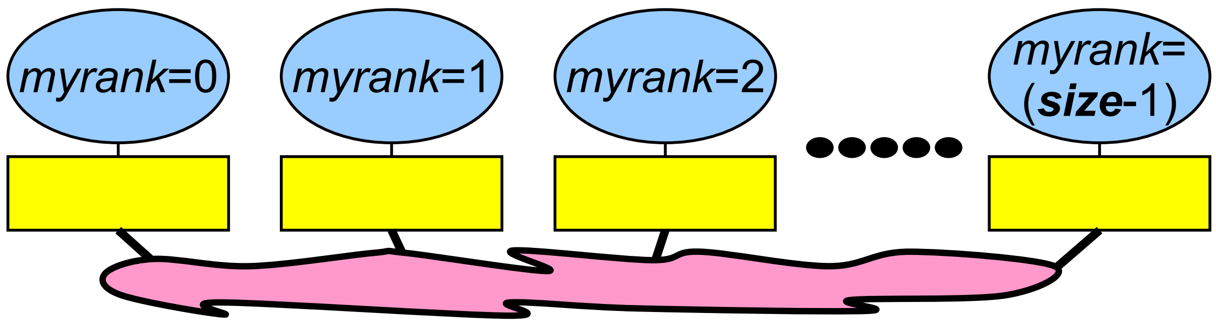
# Communicator MPI_COMM_WORLD

- All processes of an MPI program are members of the default **communicator MPI_COMM_WORLD**.
- MPI_COMM_WORLD is a predefined **handle** in mpi.h
- Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)



# Rank & Size

- The *rank* identifies different processes within a communicator
- The rank is the basis for any work and data distribution.
  - int MPI_Comm_rank( MPI_Comm comm, int **rank*)




- Size is How many processes are contained within a communicator?
  - int MPI_Comm_size( MPI_Comm comm, int **size*)

# Exiting MPI

- C: int MPI_Finalize()

- **<u>Must</u>** be called last by all processes.
- After MPI_Finalize:
  - Further MPI-calls are forbidden
  - Especially re-initialization with MPI_Init is forbidden

# Example: Hello World

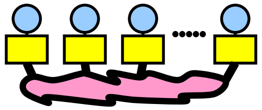
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  int rank, size;

  MPI_Init(&argc, &argv);

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

  printf("Hello from process %d/%d\n", rank, size);

  MPI_Finalize();
  return 0;
}
```

Terminal

```
bash-3.2$ /usr/local/bin/mpirun -n 8 ./hello
Hello from process 2/8
Hello from process 5/8
Hello from process 6/8
Hello from process 1/8
Hello from process 0/8
Hello from process 3/8
Hello from process 4/8
Hello from process 7/8
bash-3.2$
bash-3.2$
```

# Chap.3 Messages and Point-to-Point Communication

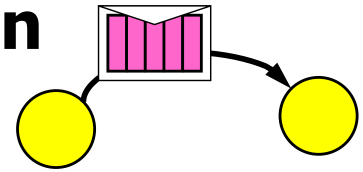
1. MPI Overview



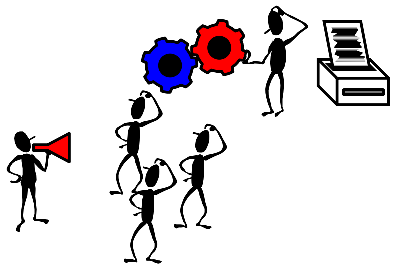
2. Process model and language bindings

```
MPI_Init()
MPI_Comm_rank()
```

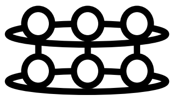
3. **Messages and point-to-point communication**
   - **the MPI processes can communicate**



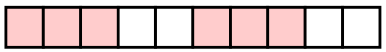
4. Non-blocking communication



5. Collective communication
6. Virtual topologies



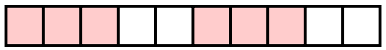
7. Derived datatypes



8. Case study

# Messages

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
  - Basic datatype.
  - Derived datatypes



- Datatype handles are used to describe the type of the data in the memory.

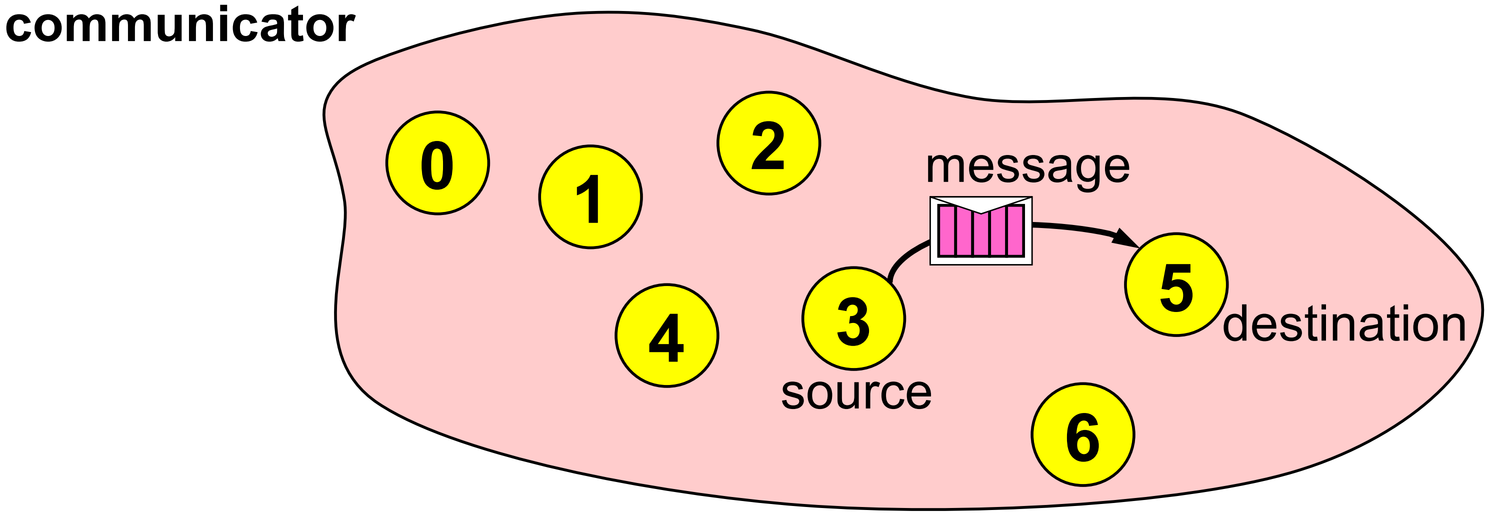
Example: message with 5 integers

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

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

# Point-to-Point Communication

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



# Sending a Message

- C: int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- buf is the starting point of the message with count elements, each described with datatype.
- dest is the rank of the destination process within the communicator comm.
- tag is an additional nonnegative integer piggyback information, additionally transferred with the message.
- The tag can be used by the program to distinguish different types of messages.

# Receiving a Message

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

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

# Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message datatypes must match.
- Receiver's buffer must be large enough.

# Wildcards

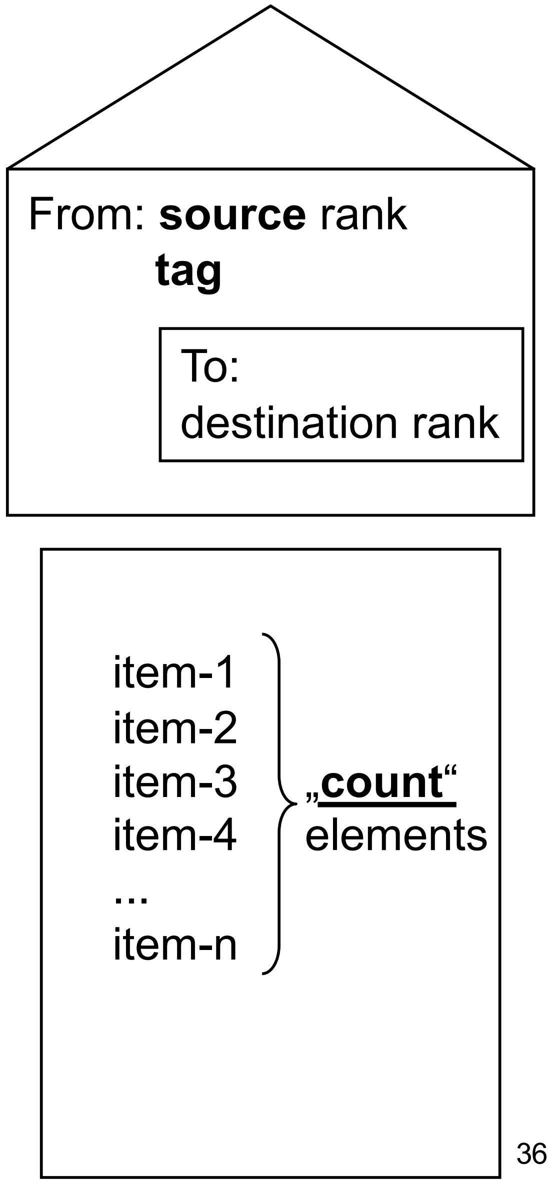
- Receiver can wildcard.
- To receive from any source — source = MPI_ANY_SOURCE
- To receive from any tag — tag = MPI_ANY_TAG
- Actual source and tag are returned in the receiver's *status* parameter.

# Communication Envelope

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

status.MPI_SOURCE
status.MPI_TAG
<u>count</u> via MPI_Get_count()

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count*)



# Communication Modes

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

# Communication Modes — Definitions

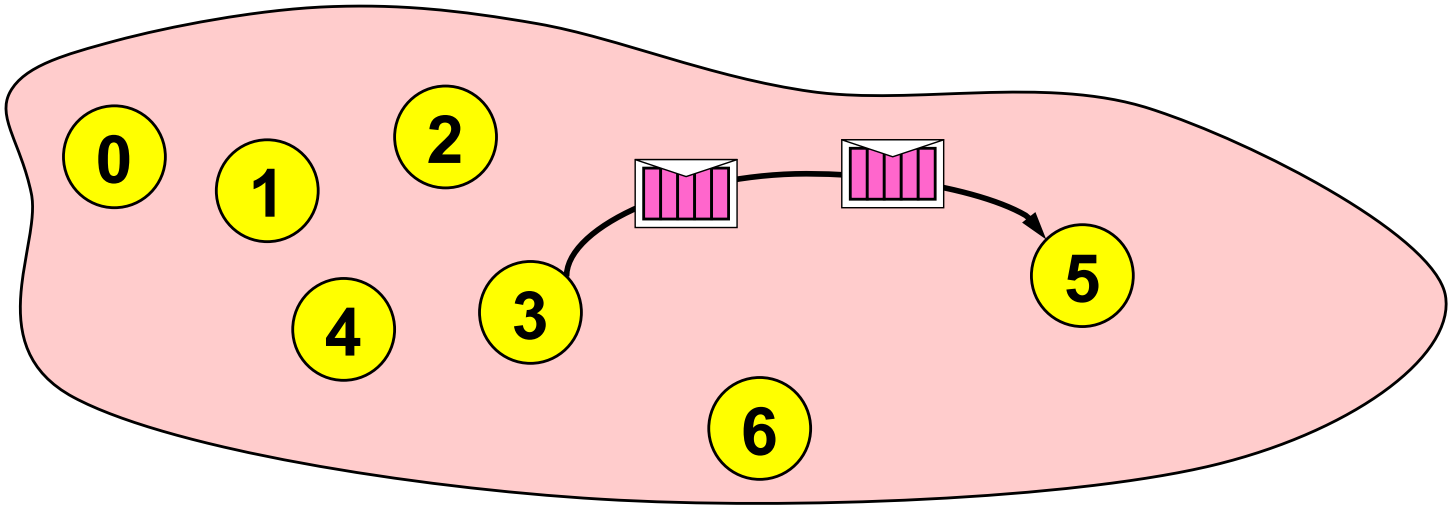
| Sender modes | Definition | Notes |
|---|---|---|
| Synchronous send<br>**MPI_SSEND** | Only completes when the receive has started | |
| Buffered send<br>**MPI_BSEND** | Always completes<br>(unless an error occurs), irrespective of receiver | needs application-defined buffer to be declared with<br>MPI_BUFFER_ATTACH |
| Synchronous<br>**MPI_SEND** | Standard send. Either uses an internal buffer or buffered | |
| Ready send<br>**MPI_RSEND** | same as MPI_Send, but, it expects a **ready destination** to receive the message. This can increase the MPI performance if the programmer is sure there is a receive function waiting for this. If no receive posted before, it is erroneous. | highly dangerous! |
| Non-blocking send<br>**MPI_ISEND** | returns immediately, data must not be modified unless MPI_Test and MPI_Wait confirm MPI_Isend is completed | |
| Receive<br>**MPI_RECV** | Completes when a the message (data) has arrived | |

# Rules for the communication modes

- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode
  - **—>** risks with synchronous send
- Synchronous send (**MPI_SSEND**)
  - risk of deadlock
  - risk of serialization
  - risk of waiting **—>** idle time
  - high latency / best bandwidth
- Buffered send (**MPI_BSEND**)
  - low latency / bad bandwidth
- Ready send (**MPI_RSEND**)
  - use **never**, except you have a 200% guarantee that Recv is already called in the current version and all future versions of your code

# Message Order Preservation

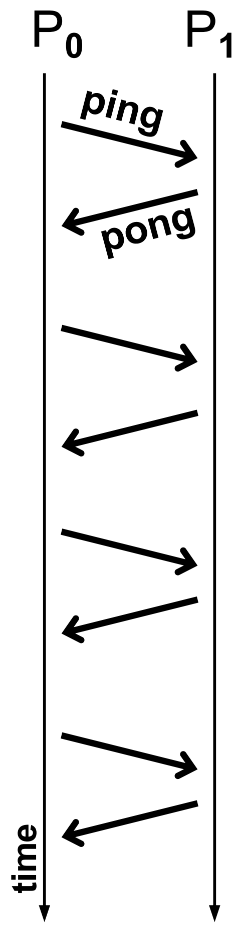
- Rule for messages on the same connection, i.e., same communicator, source, and destination rank:
- **Messages do not overtake each other.**
- This is true even for non-synchronous sends.



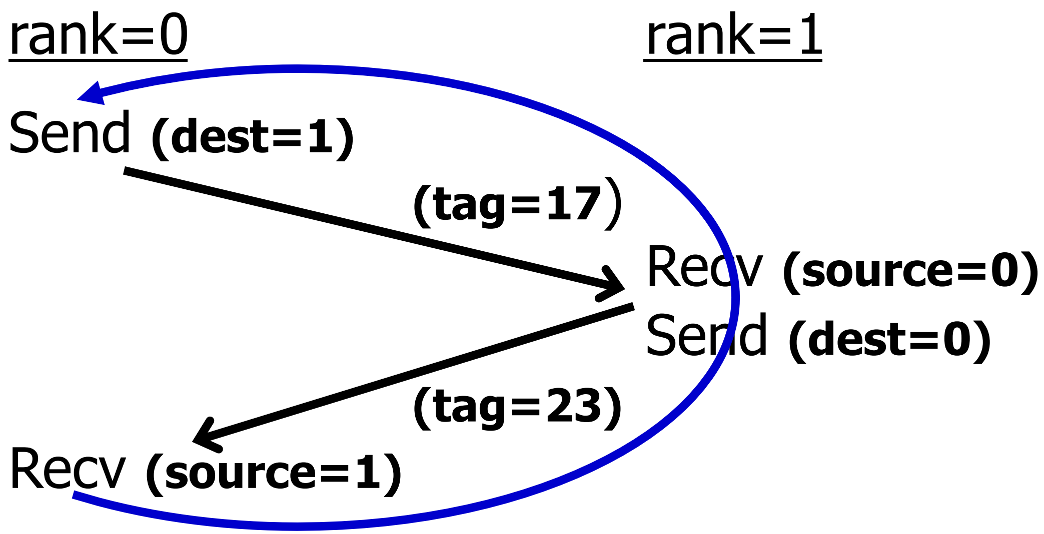

- If both receives match both messages, then the order is preserved.

# Exercise — Ping pong

- Write a program according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message,
    process 1 sends a message back to process 0 (pong)
- Repeat this ping-pong with a loop of length 50
- Add timing calls before and after the loop:
- C: *double MPI_Wtime*(void);
- MPI_WTIME returns a wall-clock time in seconds.
- At process 0, print out the transfer time of **one** message
  - in seconds
  - in μs.



# Exercise — Ping pong




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

# Advanced Exercise - Measure latency and bandwidth

- latency = transfer time for zero length messages
- bandwidth = message size (in bytes) / transfer time

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