EE 542
Lecture 14: Message Passing Interface

Internet and Cloud Computing

Young Cho
Department of Electrical Engineering
University of Southern California

based on the MPI course developed by Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart

Parallel Processing

- Parallel Hardware Accelerators
  - GPU (CUDA, OpenCL, TensorFlow, OpenAcc)
  - Tensor Processing Unit (TensorFlow)
  - FPGA (TensorFlow?!, OpenAcc?!, custom)
- Multicore Processors
  - GPP (OpenMP)
  - GPP+HW accelerators (OpenAcc)
- Multicomputers
  - NOW/Clusters (*MPI)
  - Cloud (Hadoop, Spark, MPI?!)

MPI (Message Passing Interface)?

- Messages
  - More than just the data
  - Generalized framework for parallel programming
  - Data types and other defined parameters
- Active Messages (On top of existing MPI)
  - Carry the actual code or calls to code also along with MPI contents
  - Can be of higher scalability
- Standardized message passing library spec. (IEEE)
  - for parallel computers, clusters and heterogeneous networks
  - not a specific product, compiler specification etc.
  - many implementations, MPICH, LAM, OpenMPI, AM, and …
- Real Parallel Programming
  - Portable with C/C++ (and Fortran, obsolete)
  - Support many common parallel functions
  - Difficult to develop and notoriously difficult to debug

History of MPI

- MPI-1 Forum
  - First message-passing interface standard.
  - Sixty people from forty different organizations.
  - Users and vendors represented, from US and Europe.
  - Two-year process of proposals, meetings and review.
  - *Message-Passing Interface* document produced.
  - MPI 1.0 — June, 1994.
  - MPI 1.1 — June 12, 1995.
- MPI-2 Forum July 18, 1997

Goals and Scope of MPI

- Goals
  - To provide a parallel programing interface.
  - To provide source-code portability.
  - To allow efficient implementations.
- Result
  - A great deal of functionality for parallel programming
  - Support for heterogeneous parallel architectures
  - MPI-2: additional functions and backward compatible

Information about MPI

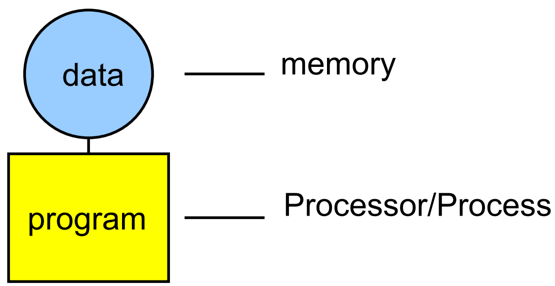
- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July 18,1997)
- **MPI: The Complete Reference**, Marc Snir and William Gropp et al, The MIT Press, 1998 (2-volume set)
- **Using MPI: Portable Parallel Programming With the Message-Passing 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.

Example of Compilation

- Compilation in C
  - mpicc -o prog prog.c
- Compilation in C++:
  - mpiCC -o prpg prog.c
  - mpicxx -o prog prog.cpp
- Executing program with num processes
  - mprun –n num ./pra
  - mpiexec -n num ./prg

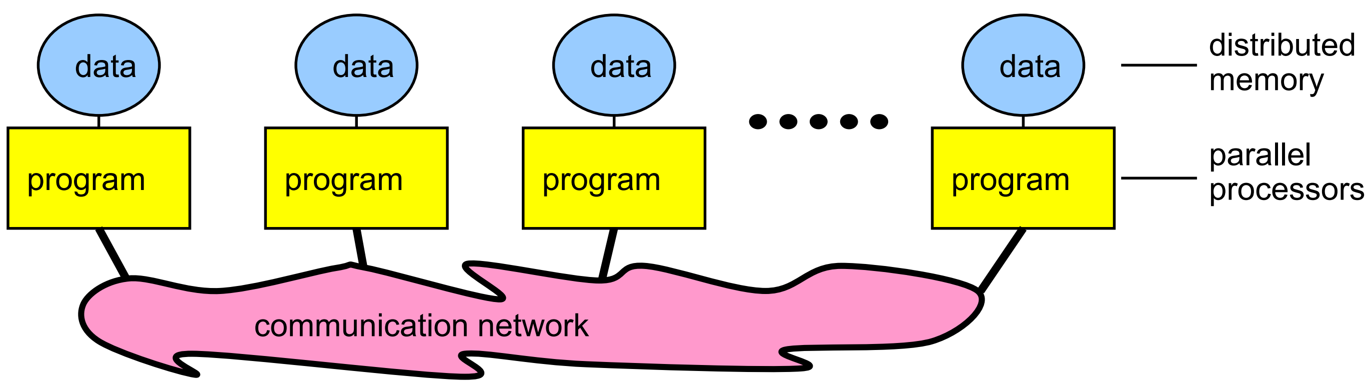
The Message-Passing Programming Paradigm

- Sequential Programming Paradigm



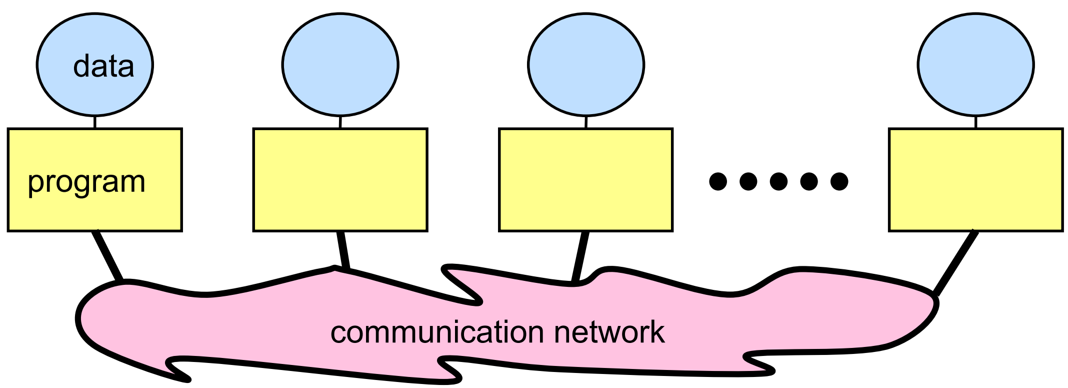
A processor may run many processes

- Message Passing Programming Paradigm



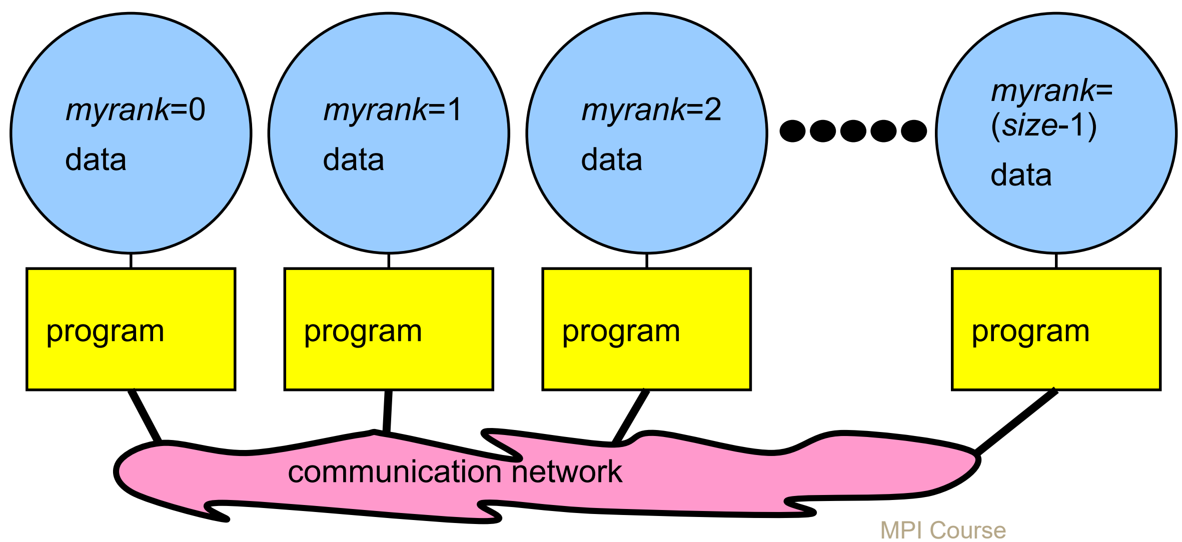
The Message-Passing Programming Paradigm

- 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 C++
  - 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



MPI Programming Execution

- Same (sub-)program may run on each processor
  - **S**ingle **P**rogram, **M**ultiple **D**ata
  - Easier to develop and run
- MPI allows also MPMD
  - **Multiple P**rogram, **M**ultiple **D**ata
  - Easier to emulate MPMD with SPMD
  - Decision tree based on rank

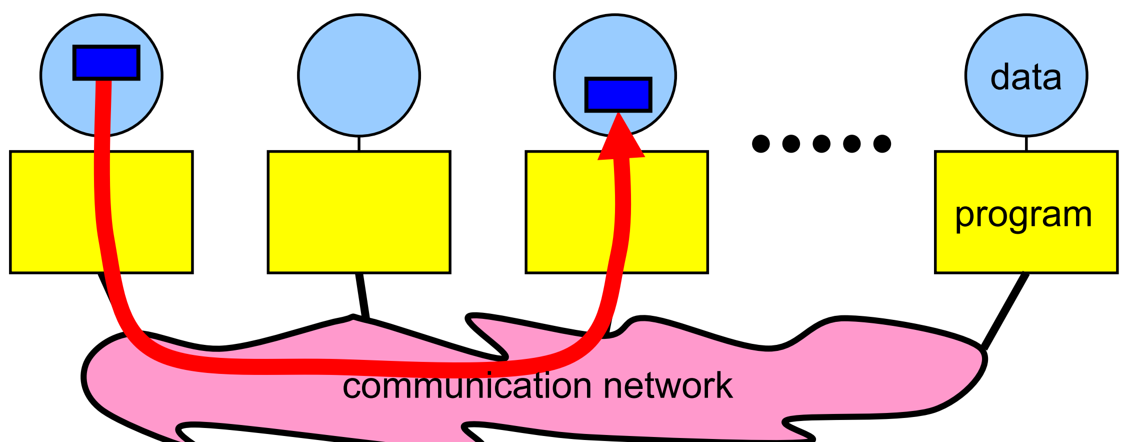
Emulation of MPMD

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

- ```
  PROGRAM
  IF (myrank < ... ) THEN    !! process should run the ocean model
          CALL  ocean ( some arguments )
  ELSE
          CALL  weather ( some arguments )
  ENDIF
  END
  ```

Message passing

- Messages are packets of 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



Access

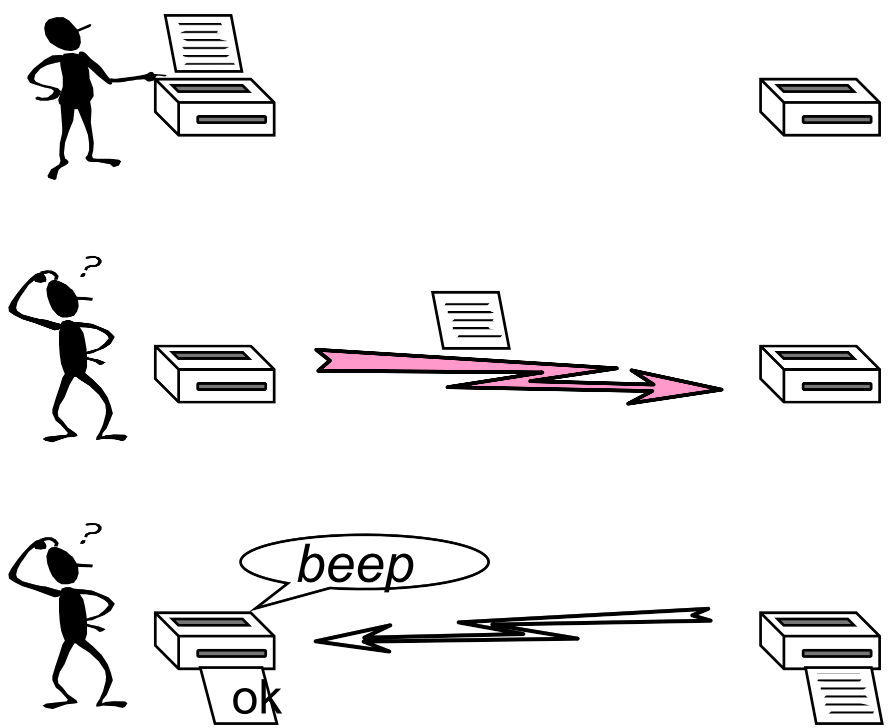
- Program must be linked with an MPI library
- Program must be started with the MPI startup
- A sub-program needs to be connected to a message passing system

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

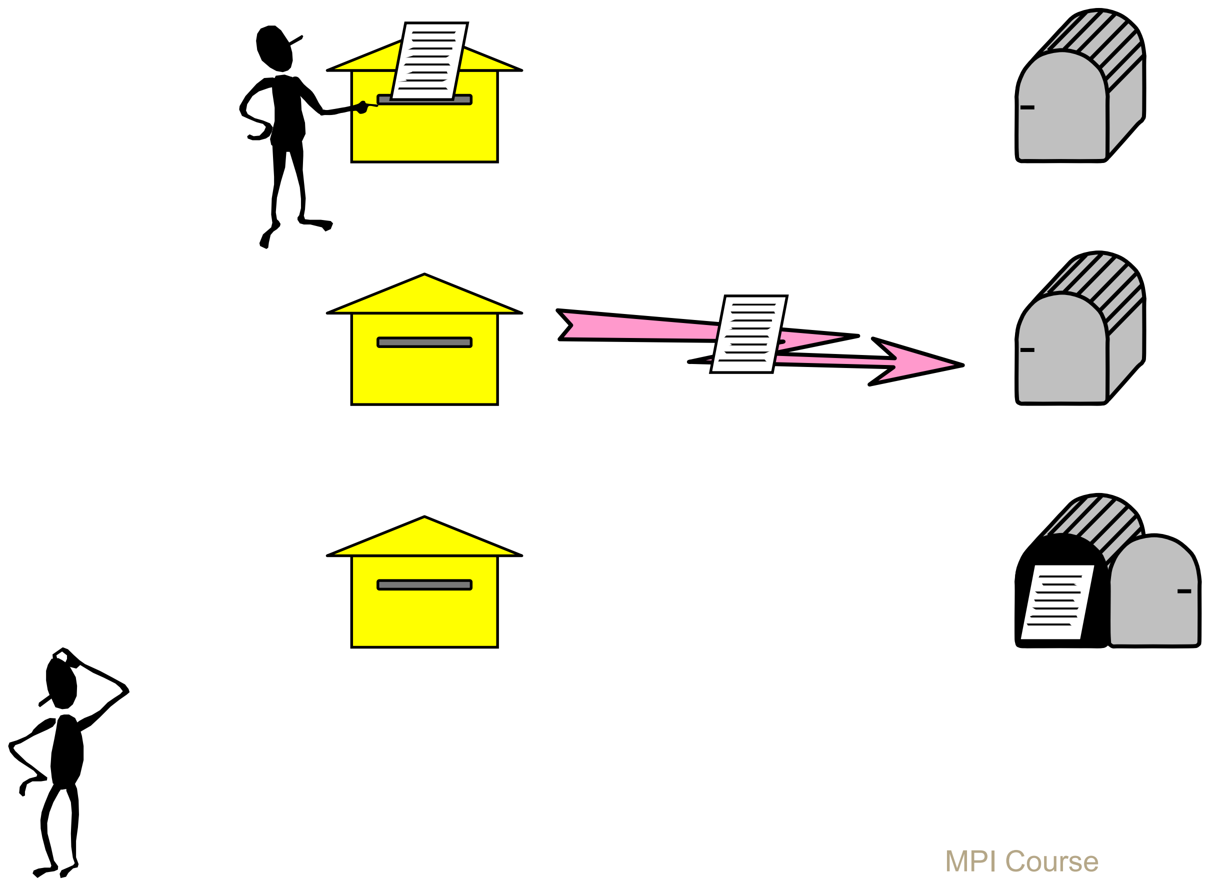
Synchronous Sends

- 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.

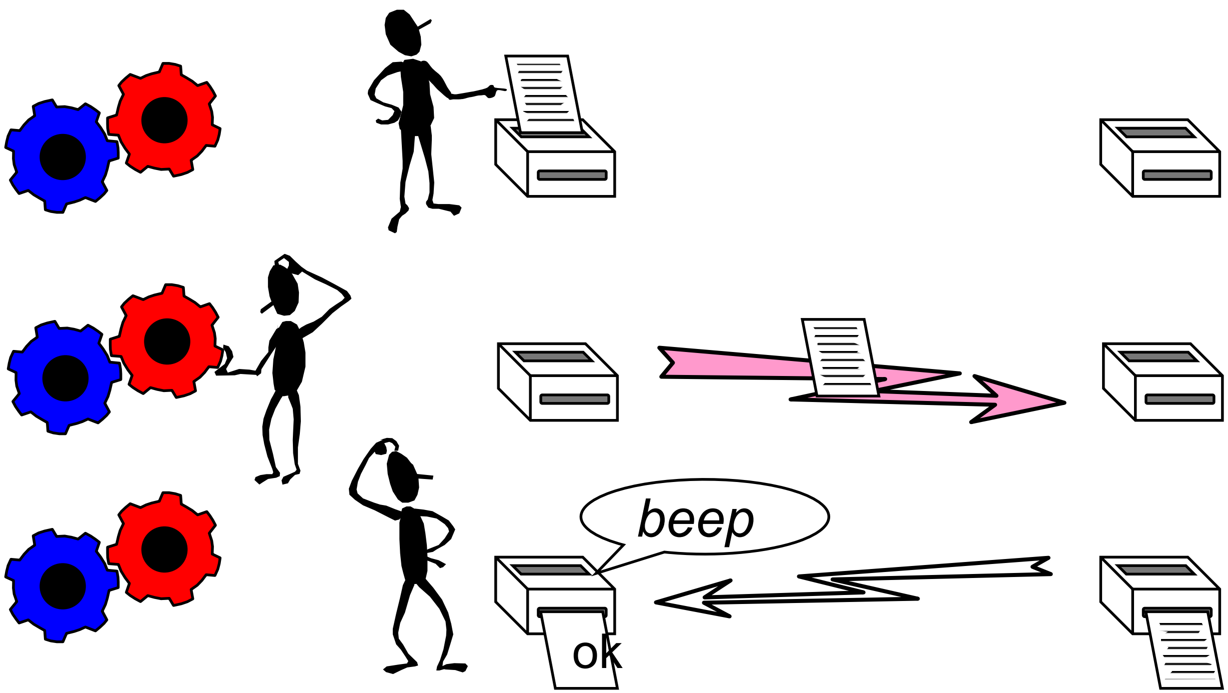


Blocking Operations

- 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.

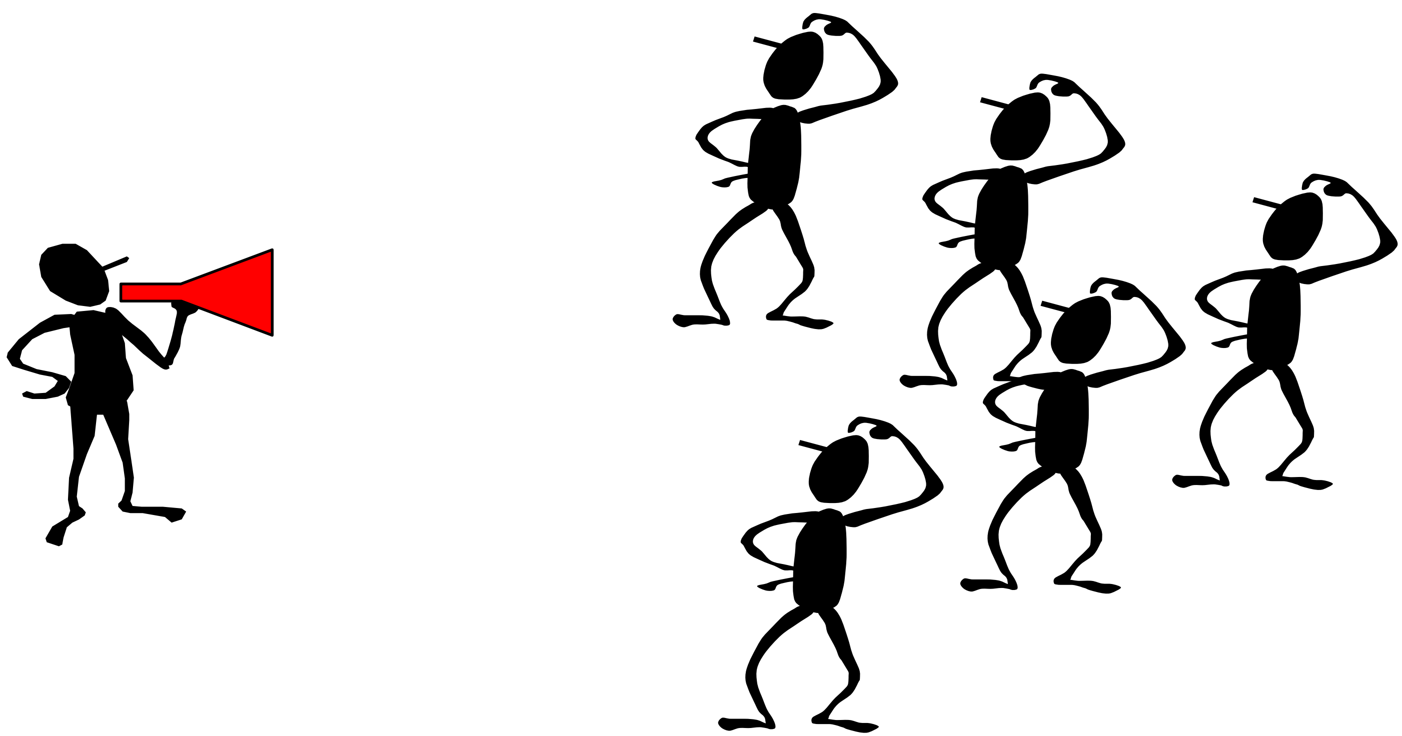


Collective Communications

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

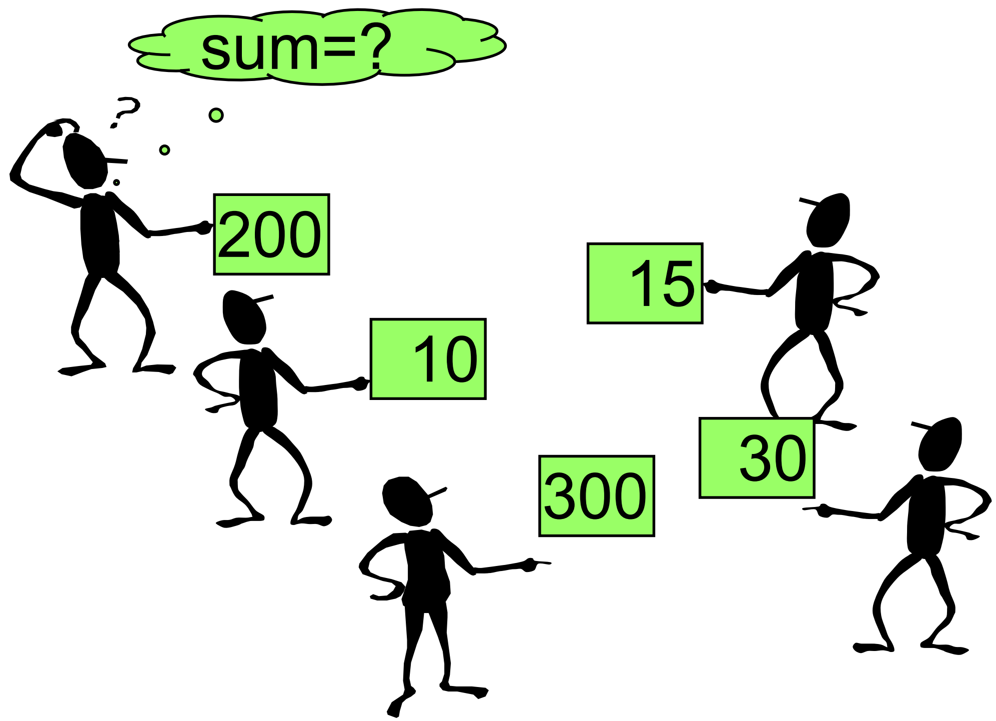
Broadcast

- A one-to-many communication.



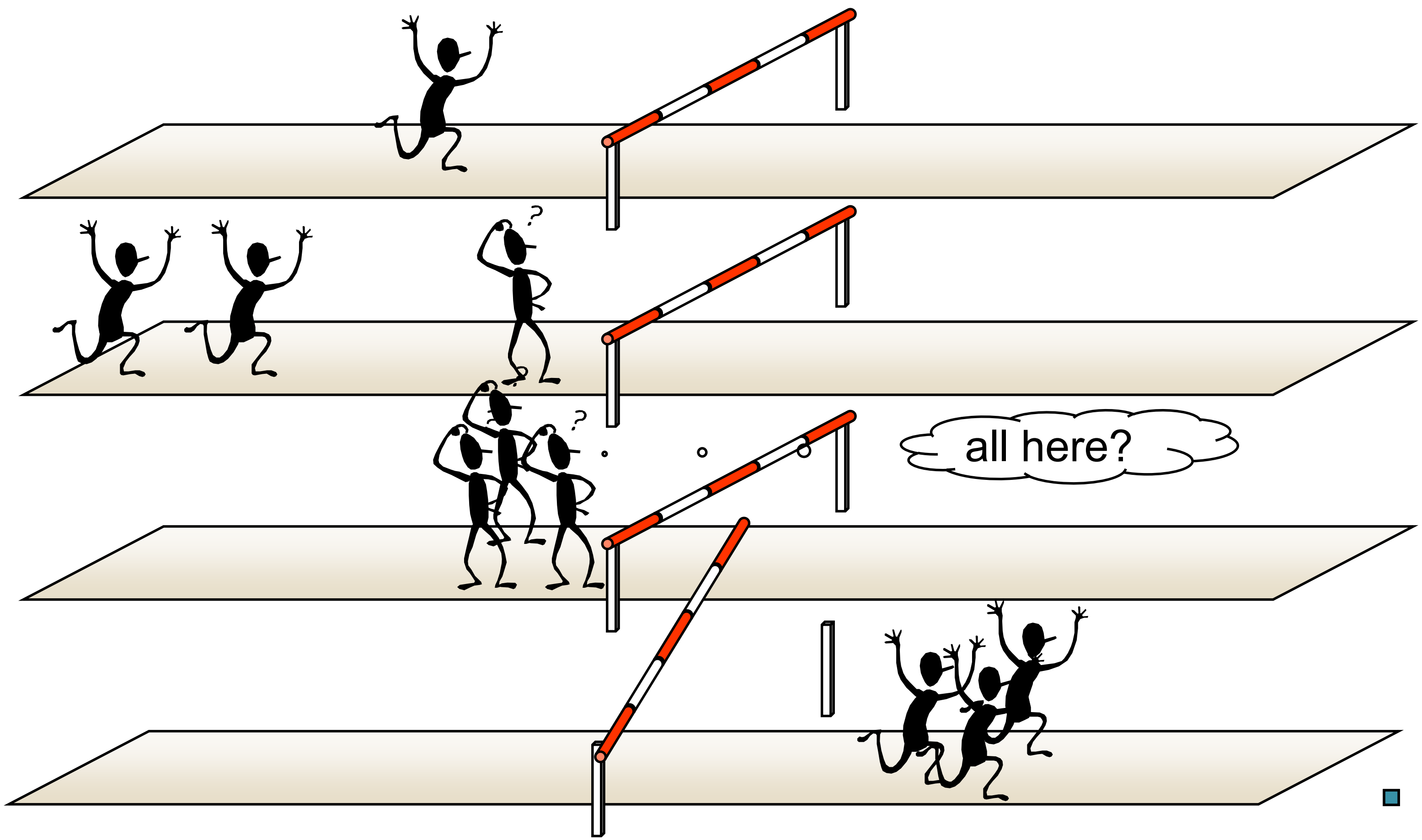
Reduction Operations

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



Barriers

- Synchronize processes.



MPI Function Format

- Include file for C
  - #include <mpi.h>
- Interface in C might look like
  - error = MPI_Xxxxxx(parameter, ...);
  - MPI_...... namespace is reserved for MPI constants and routines
  - i.e. application routines and variable names must NOT begin with MPI_ .
- Example arguments in C
  - Definition standard

```
MPI_Comm_rank( ...., int *rank)
MPI_Recv(..., MPI_Status *status)
```

  - Usage

```
main...
{  int myrank;
   MPI_Status rcv_status;
   MPI_Comm_rank(..., &myrank);
   MPI_Recv(..., &rcv_status); }
```

MPI Programming and Running

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

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

  - MPI_Init must be the first MPI routine that is called
  - The parallel MPI processes exist at least after MPI_Init
- Start mechanism is implementation dependent
  - Most implementations provide mpirun:

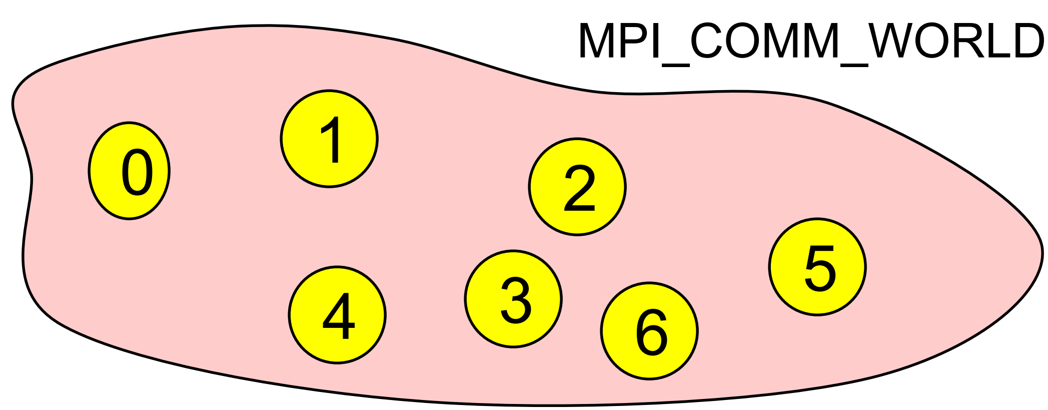
    mpirun –np ***number_of_processes ./executable***

    mprun –n ***number_of_processes ./executable***
  - MPI-2 standard defines mpiexec:

    mpiexec –n ***number_of_processes ./executable***

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 and mpif.h.
- Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)

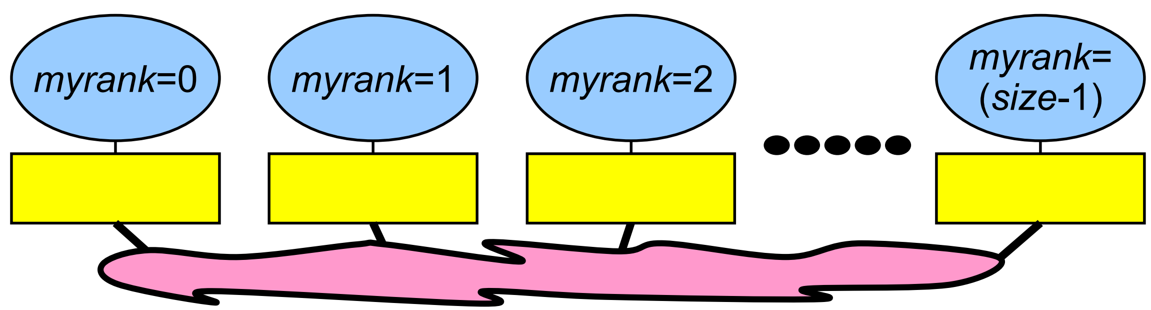


Handles

- Handles identify MPI objects.
- For the programmer, handles are
  - predefined constants in mpi.h or mpif.h
    - **example: MPI_COMM_WORLD**
    - **predefined values exist only** after MPI_Init **was called**
  - values returned by some MPI routines,
    to be stored in variables, that are defined as
    - **in C: special MPI typedefs**
- Handles refer to internal MPI data structures

Rank and Size

- The *rank* identifies different processes within a communicator
- The rank is the basis for any work and data distribution.
    - C: int MPI_Comm_rank(MPI_Comm comm, int **rank*)
- How many processes are contained within a communicator?
    - C: int MPI_Comm_size(MPI_Comm comm, int *size)



CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierror)

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

Hello World

- Example ***hello world*** by each MPI process.
  - You can compile and run it on a single processor.
  - You can run it on several processors in parallel.
  - Every process knows 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
```

- The sequence of the output non-deterministic?

MPI Data Types

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	

MPI Datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_ LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

2345	654	96574	-12	7676

count=5
datatype=MPI_INTEGER

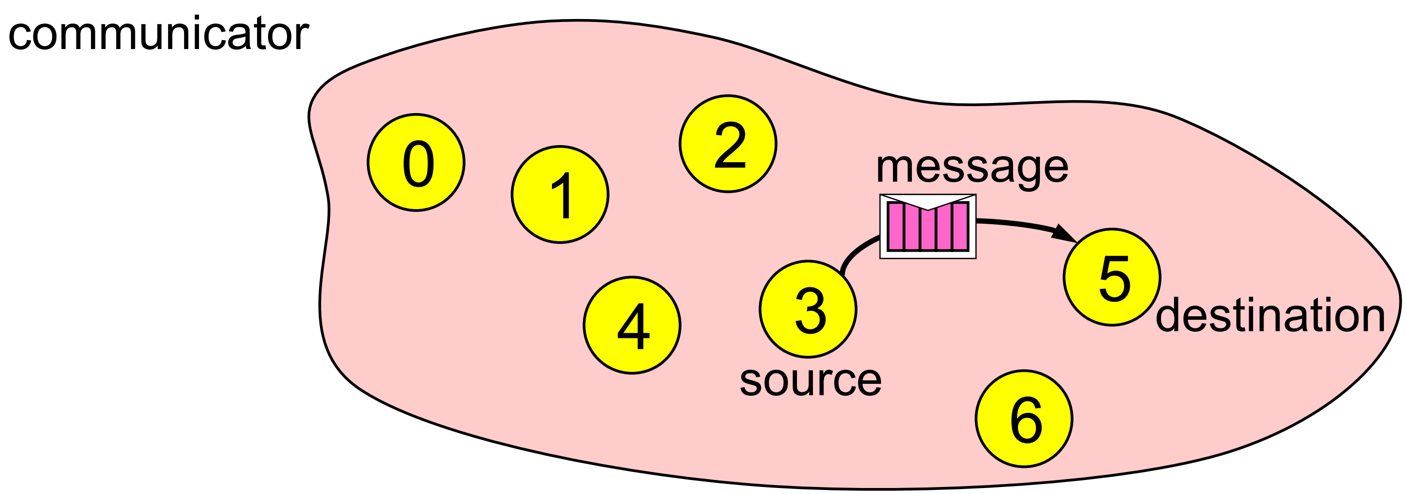
INTEGER arr(5)

MPI Derived Data Types

```
static const int blocklen[] = {1, 1, 1, 1};
static const MPI_Aint disp[] = {
    offsetof(struct B, a) + offsetof(struct A, f),
    offsetof(struct B, a) + offsetof(struct A, p),
    offsetof(struct B, pp),
    offsetof(struct B, vp)
};
static MPI_Datatype type[] = {MPI_INT, MPI_SHORT, MPI_INT,
MPI_INT};
MPI_Datatype newtype;
MPI_Type_create_struct(sizeof(type) / sizeof(*type), blocklen,
disp, type, &newtype);
MPI_Type_commit(&newtype);
```

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.

Point-to-Point Communications

- 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.

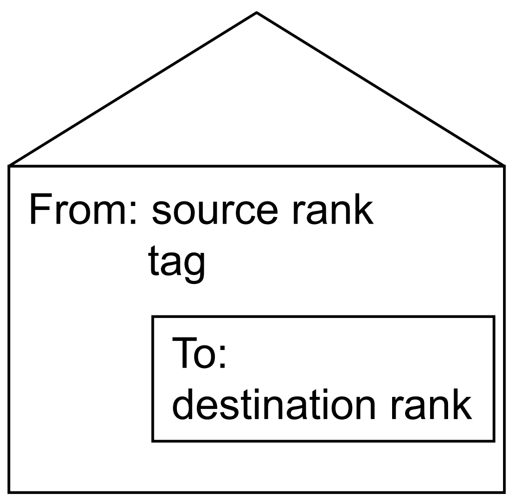
Wildcards

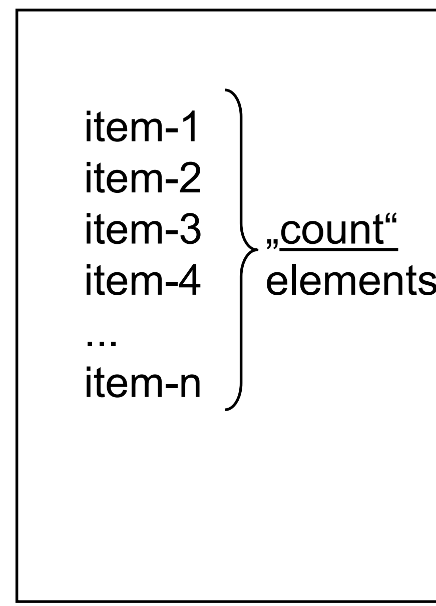
- Receiver can wildcard.
- To receive from any source — <u>source</u> = MPI_ANY_SOURCE
- To receive from any tag — <u>tag</u> = MPI_ANY_TAG
- Actual source and tag are returned in the receiver's *<u>status</u>* parameter.

Communication Envelope

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

- C: status.MPI_SOURCE
  status.MPI_TAG
  <u>count</u> via MPI_Get_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

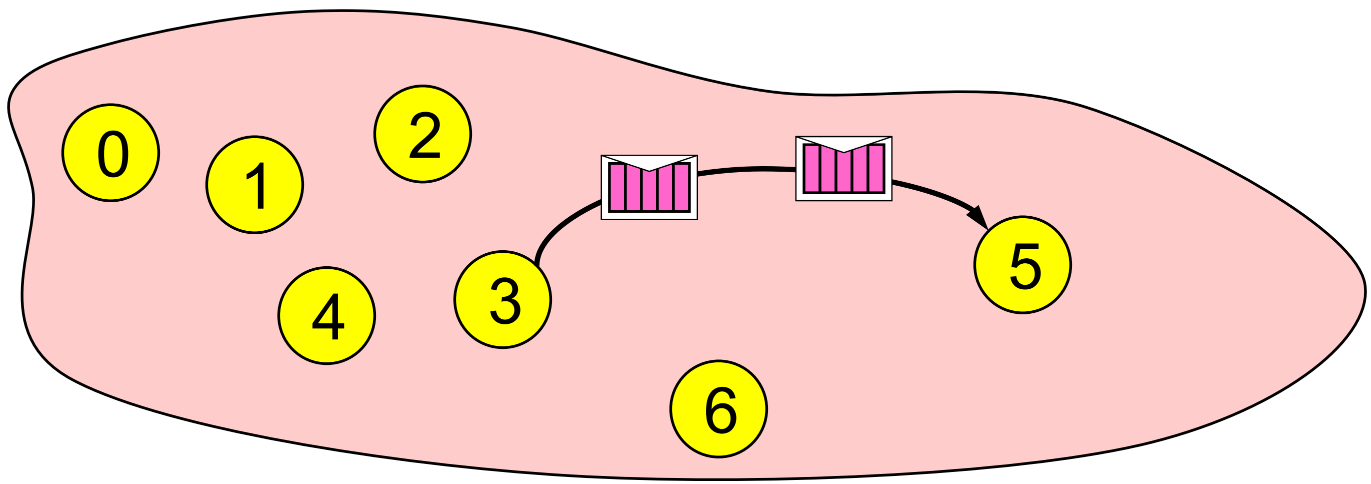
Sender modes	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Synchronous **MPI_SEND**	Standard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

Rules for the communication modes

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

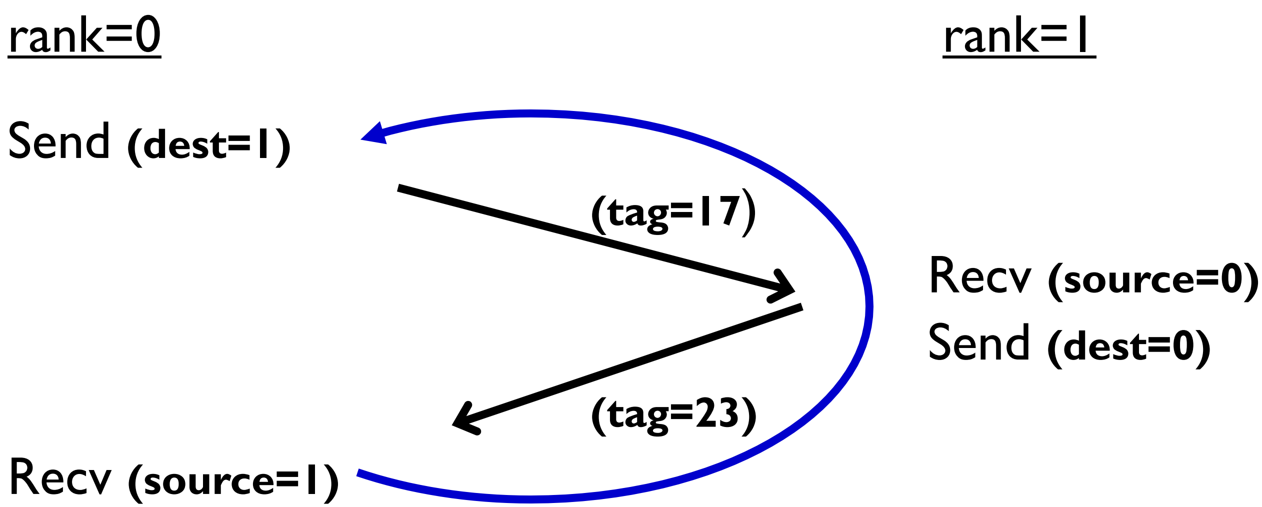
Message Order Preservation

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



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

Ping pong



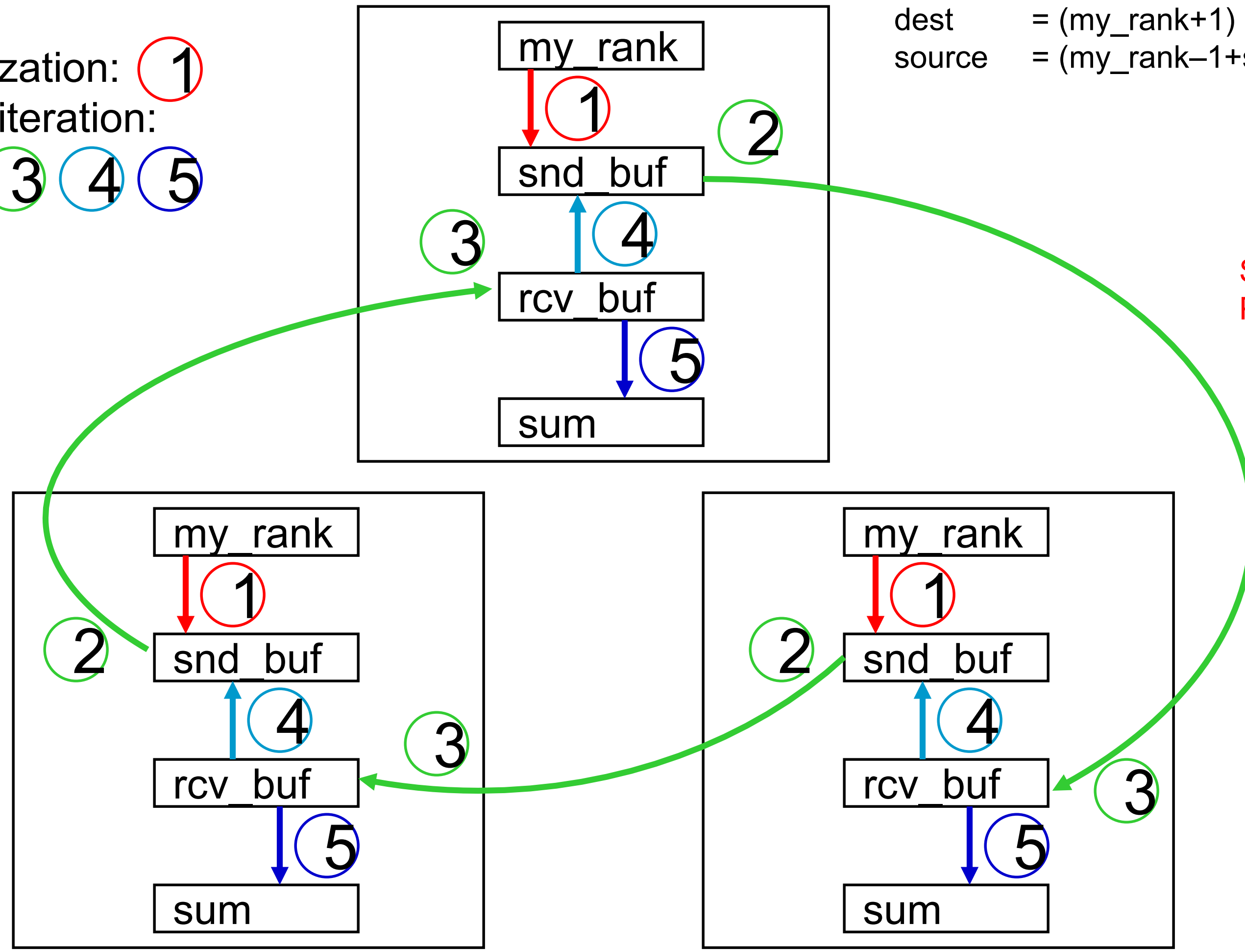

```
if (my_rank==0)              /* i.e., emulated multiple program */
    MPI_Send( ... dest=1 ...)
    MPI_Recv( ... source=1 ...)
else
    MPI_Recv( ... source=0 ...)
    MPI_Send( ... dest=0 ...)
fi
```

Initialization: 1

Each iteration: 2 3 4 5

dest = (my_rank+1) % size;

source = (my_rank–1+size) % size;



Single Program !!!

Collective Communication

- Communications involving a group of processes.
- Must be called by all processes in a communicator.
- Examples:
  - Barrier synchronization.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

Characteristics of Collective Communication

- Optimised Communication routines involving a group of processes
- Collective action over a communicator, i.e. all processes must call the collective routine.
- Synchronization may or may not occur.
- All collective operations are blocking.
- No tags.
- Receive buffers must have exactly the same size as send buffers.

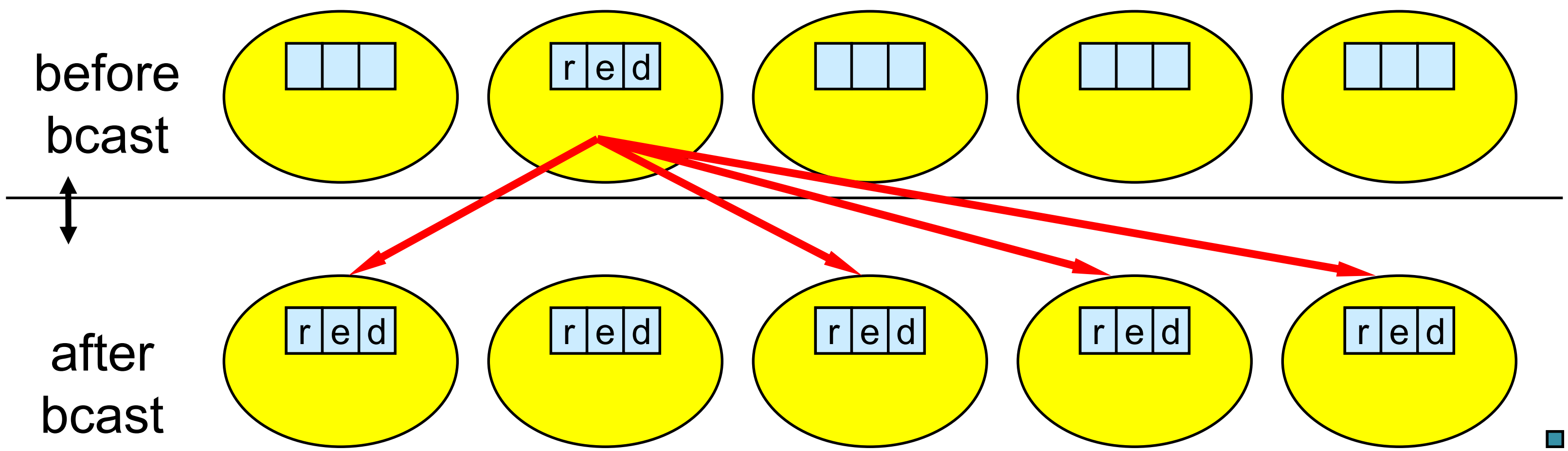
Barrier Synchronization

- C: int MPI_Barrier(MPI_Comm comm)
- Fortran: MPI_BARRIER(COMM, *IERROR*)

  INTEGER COMM, IERROR
- MPI_Barrier is normally never needed:
  - all synchronization is done automatically by the data communication:
    - a process cannot continue before it has the data that it needs.
  - if used for debugging:
    - please guarantee, that it is removed in production.

Broadcast

C: int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)



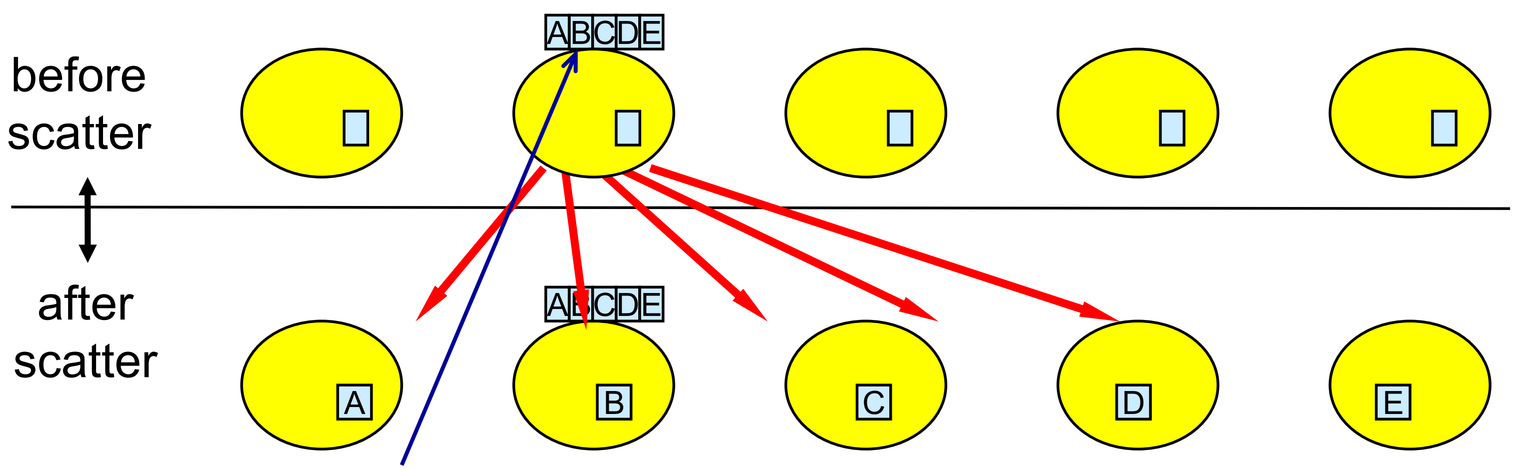
e.g., root=1

- rank of the sending process (i.e., root process)
  - must be given identically by all processes

Scatter

- C: int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
  void ***recvbuf**, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)

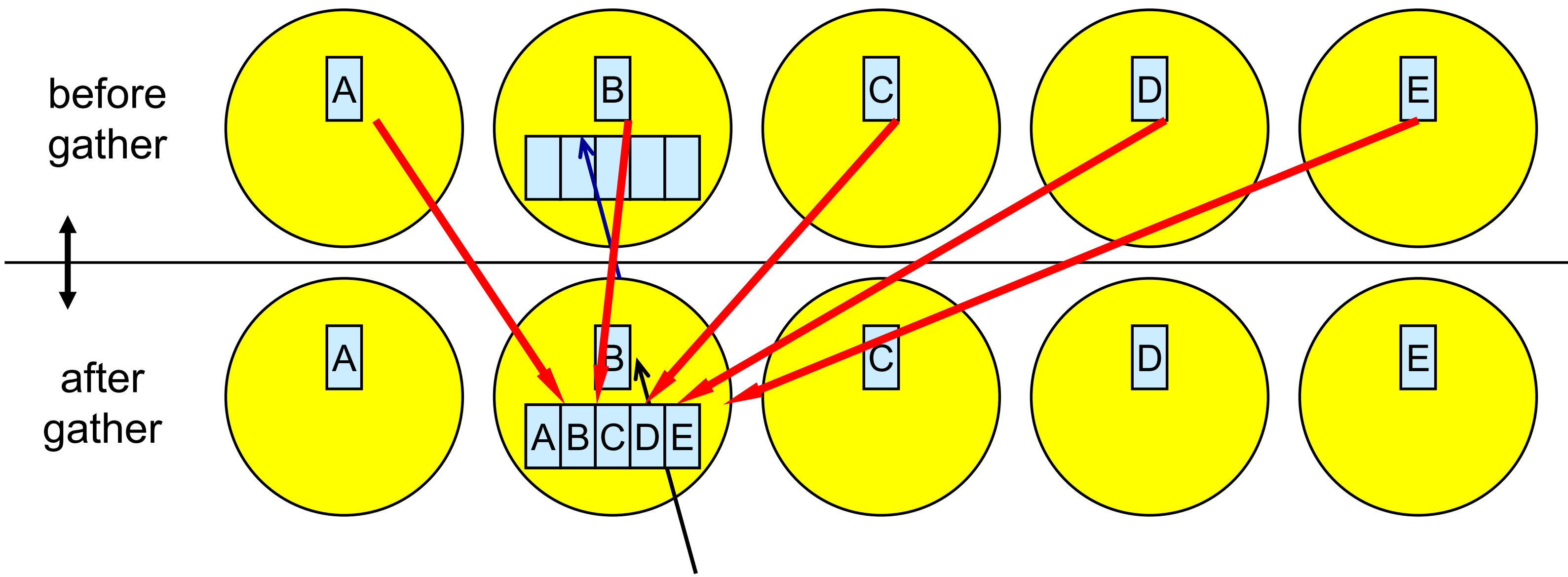
e.g., root=1



Gather

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

e.g., root=1



Global Reduction Operations

- To perform a global reduce 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
  - Example:
    - global sum or product
    - global maximum or minimum
    - global user-defined operation
- floating point rounding may depend on usage of associative law:
  - $[(d_0 \mathbf{o} d_1) \mathbf{o} (d_2 \mathbf{o} d_3)] \mathbf{o} [\ldots \mathbf{o} (d_{s-2} \mathbf{o} d_{s-1})]$
  - $((((((d_0 \mathbf{o} d_1) \mathbf{o} d_2) \mathbf{o} d_3) \mathbf{o} \ldots) \mathbf{o} d_{s-2}) \mathbf{o} d_{s-1})$

Example of Global Reduction

- Global integer sum.
- Sum of all inbuf values should be returned in *resultbuf.*
- C: root=0;
  MPI_Reduce(&inbuf, &*resultbuf*, 1, MPI_INT,
  MPI_SUM,root, MPI_COMM_WORLD);
- Fortran: root=0
  MPI_REDUCE(inbuf, *resultbuf*, 1, MPI_INTEGER,
  MPI_SUM, root, MPI_COMM_WORLD, *IERROR*)
- The result is only placed in *resultbuf* at the root process.

Predefined Reduction Operation Handles

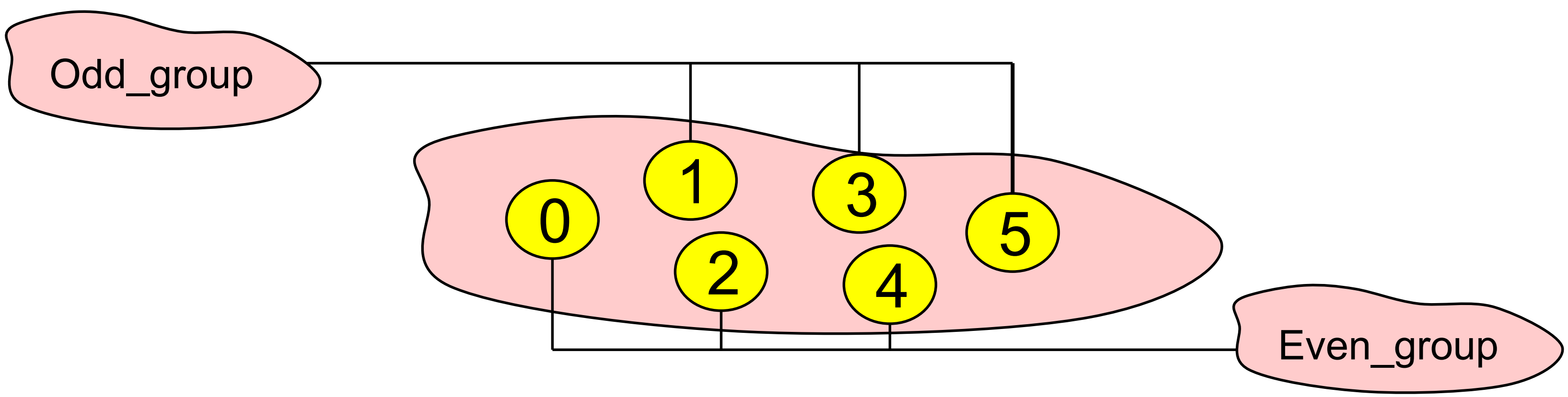
Predefined operation handle	**Function**
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum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

User-Defined Reduction Operations

- Operator handles
  - predefined – see table above
  - user-defined
- User-defined operation ■:
  - associative
  - user-defined function must perform the operation vector_A ■ vector_B
  - syntax of the user-defined function → MPI-1 standard
- Registering a user-defined reduction function:
  - C: MPI_Op_create(MPI_User_function *func, int commute, MPI_Op **op*)
- COMMUTE tells the MPI library whether FUNC is commutative.

Working with groups

- Select processes ranks to create groups
- Associate to these groups *new* communicators
- Use these new communicators as usual
- MPI_Comm_group(comm, group) returns in *group* the group associated to the communicator *comm*



Virtual Topologies

- Convenient process naming.
- Simplifies writing of code.
- Can allow MPI to optimize communications.