Message Passing Interface

Class 6

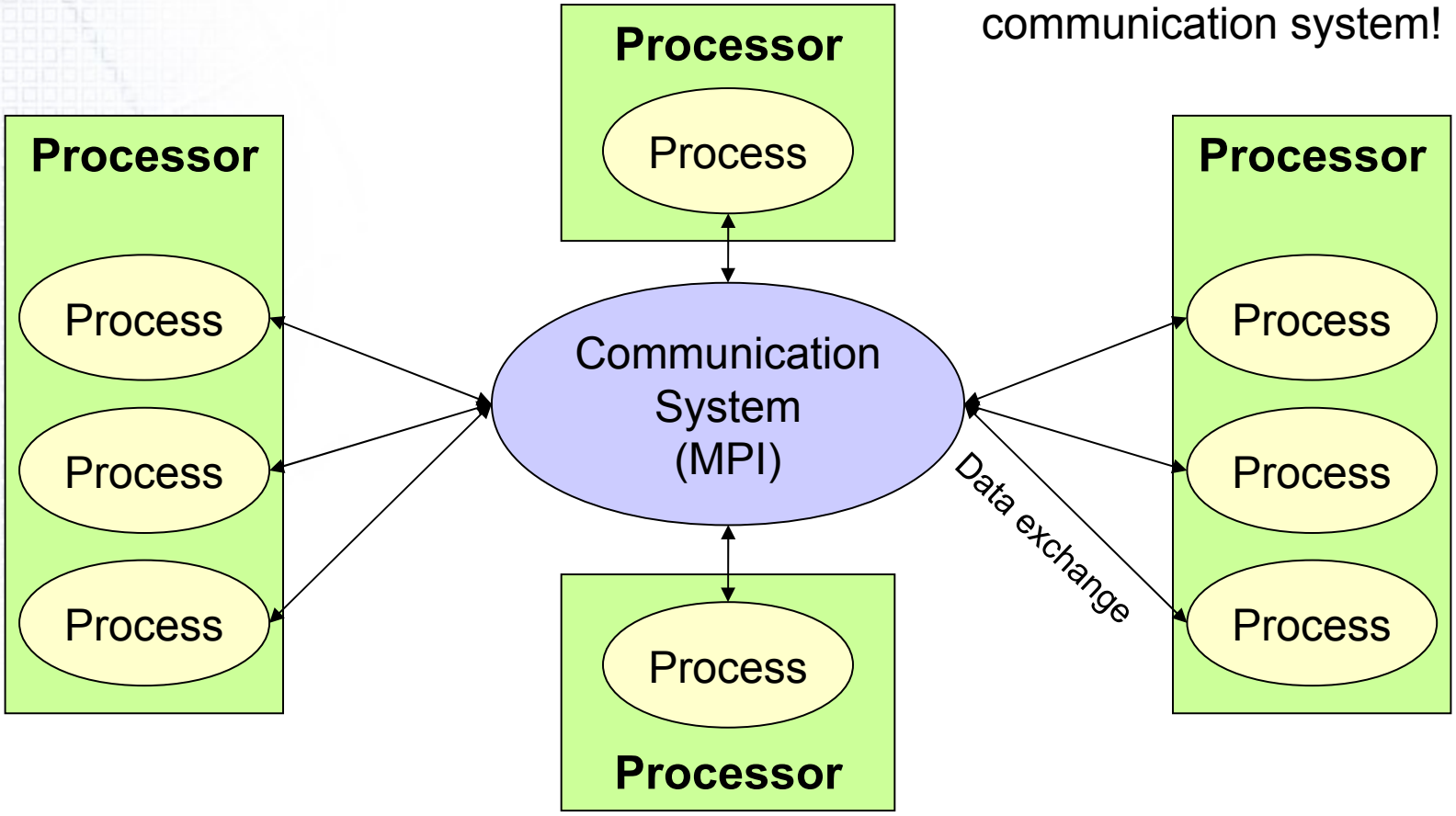
Message Passing Paradigm

The Underlying Principle

- A parallel program consists of p processes with different address spaces.
- Communication takes place via the explicit exchange of data or messages (realized via system calls like send(...) or receive(...) and others), only.
- Message consists of
  - header: target ID, message information (type, length, ...)
  - body: the data to be provided
- Need for a buffer mechanism: what to do if the receiver is not (yet) ready to receive?

The User's View

Library functions as the only interface to the communication system!

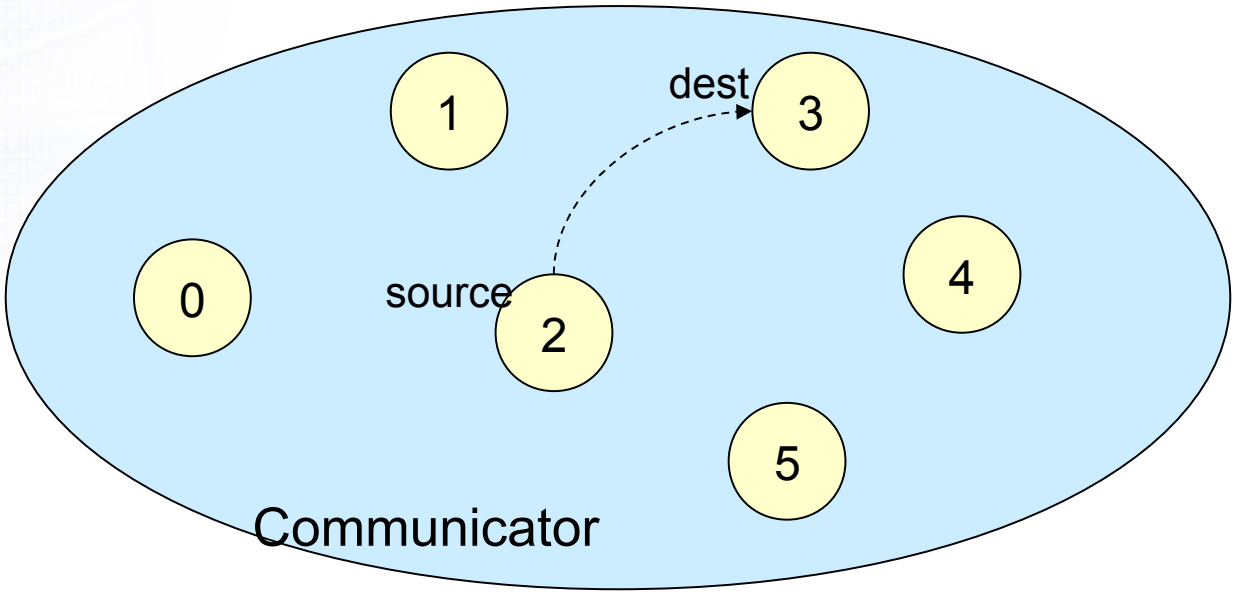


Message Buffers

- Typically (but not necessarily) connected parts of memory
  - homogeneous architectures (all processors of the same type):
    - buffer as a sequence of bytes, without any type information
  - heterogeneous architectures (different types of processors):
    - type information necessary for format conversion by message passing library
- Definition and allocation of message buffers:
  - send buffer: generally done by application program
  - receive buffer: either automatically by message passing library or manually by application program (eventually with check whether buffer length is sufficient)

Point-to-Point Communication

A point-to-point communication always involves exactly two processes. One process acts as the sender and the other acts as the receiver.



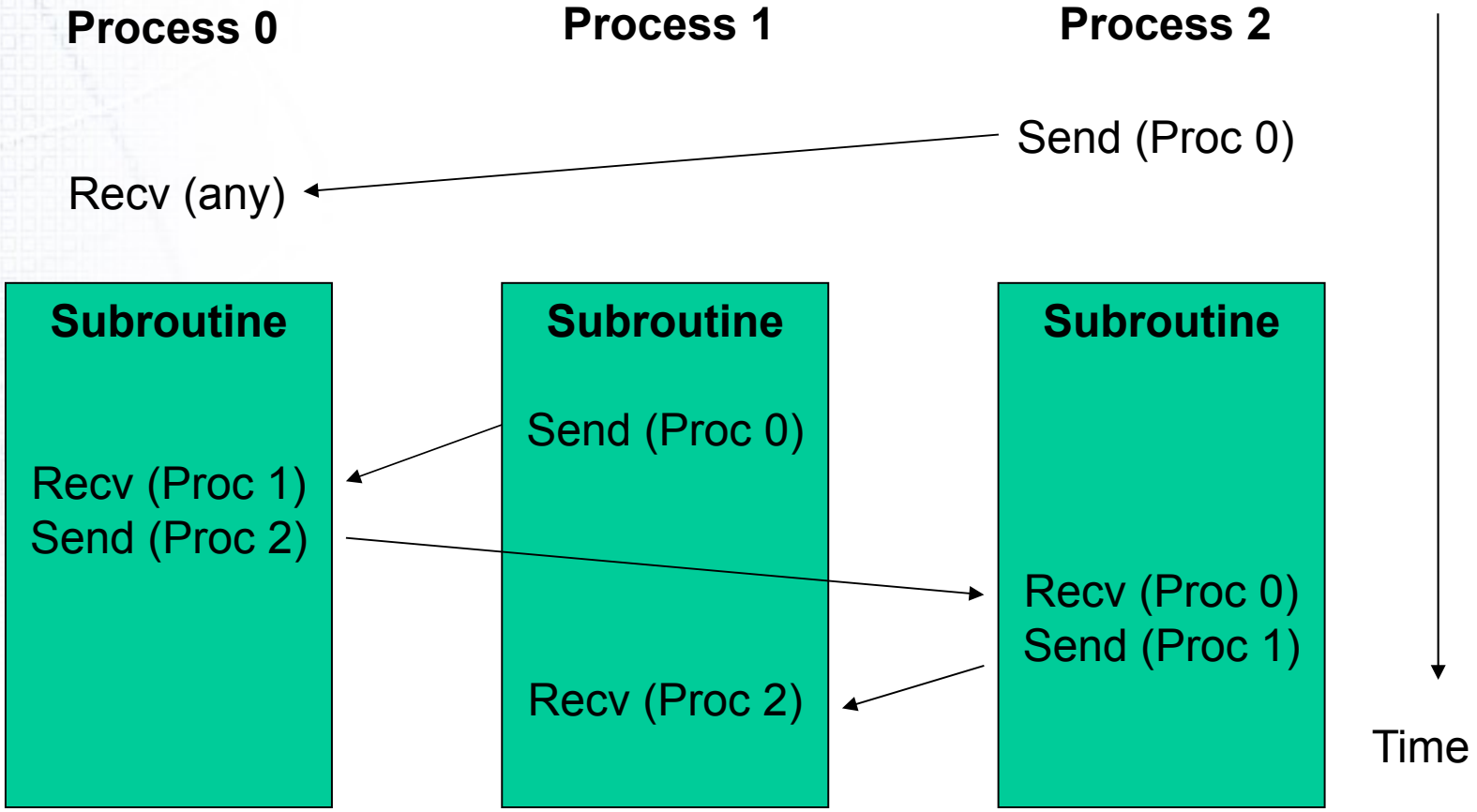
Point-to-Point Communication

- Send – required information:
  - receiver (who shall get the message?)
  - send buffer (where is the data provided?)
  - type of message (what kind of information is sent?)
  - communication context (context within which the message may be sent and received)
- Receive – required information:
  - sender (wild cards are possible, i.e. receive from any process)
  - receive buffer (where is the incoming message to be put?)
  - type of message
  - communication context

Communication Context

- Consider a scenario:
  - three processes, and all of them call a subroutine from a library
  - inter-process communication within the subroutines
  - communication context shall ensure this restriction to the subroutines
  - compare correct order (next slide) and error case

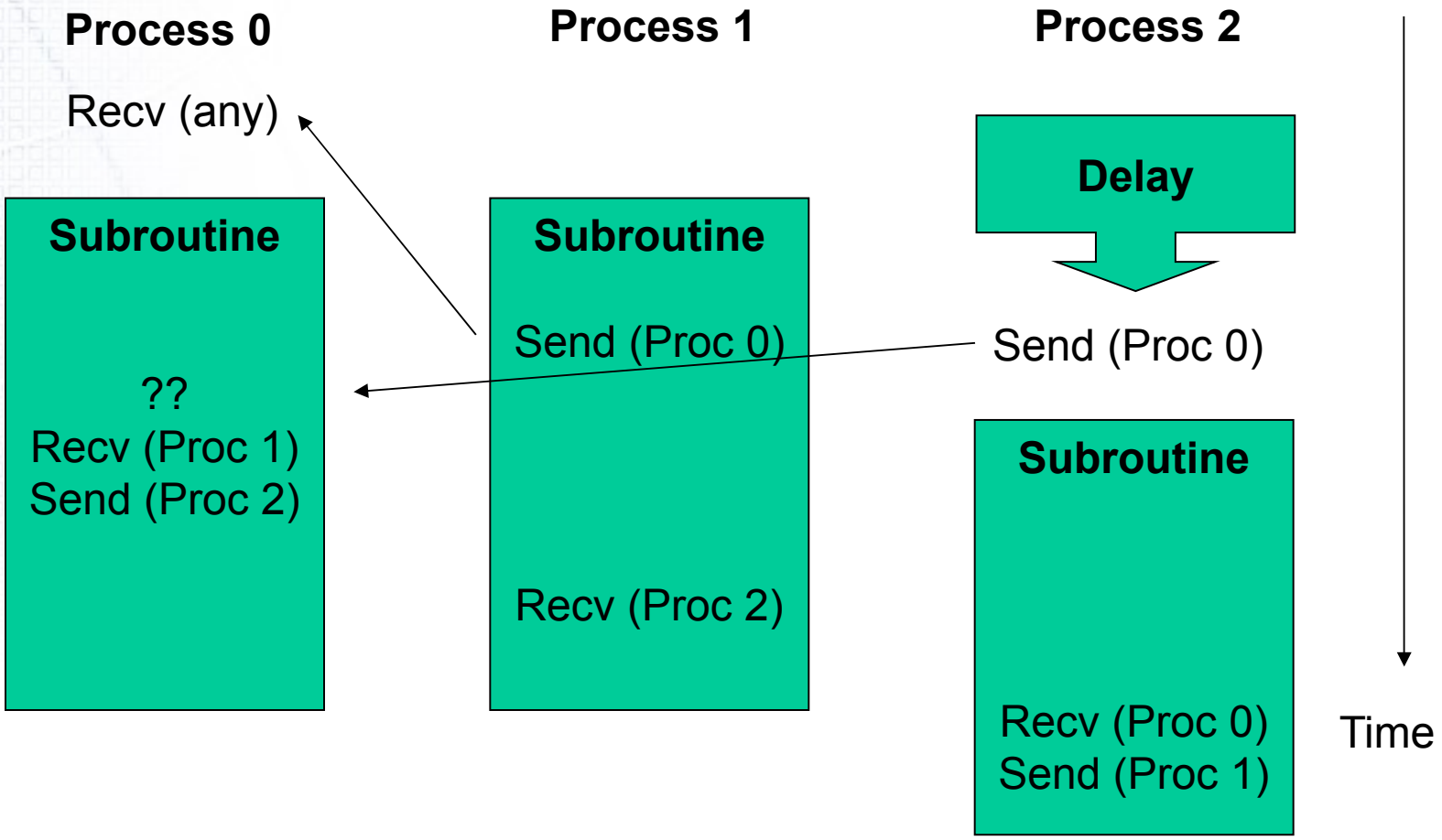
Communication Context



Communication Context

- Consider a scenario:
  - three processes, and all of them call a subroutine from a library
  - inter-process communication within the subroutines
  - communication context shall ensure this restriction to the subroutines
  - compare correct order (previous slide) and error case (next slide)

Communication Context



Why Buffers?

- P1:
- Compute something
- Store result in SBUF
- SendBlocking(P2,SBUF)
- ReceiveBlocking(P2,RBUF)
- Read data in RBUF
- Process RBUF

- P2:
- Compute something
- Store result in SBUF
- SendBlocking(P1,SBUF)
- ReceiveBlocking(P1,RBUF)
- Read data in RBUF
- Process RBUF

Case Study

- Using `which mpirun` to see whether you are using MPICH-1.2.5. If no, update the ~/.bashrc with the correct path.
- By using the `wget` command, download the sample program from
  - http://www.sci.hkbu.edu.hk/tdgc/tutorial/RHPCC/source/c/casestudy
- Compile and run the program with 2 processes
- Change the BUFSIZE with 32000 and then recompile and run the program
- Note the difference

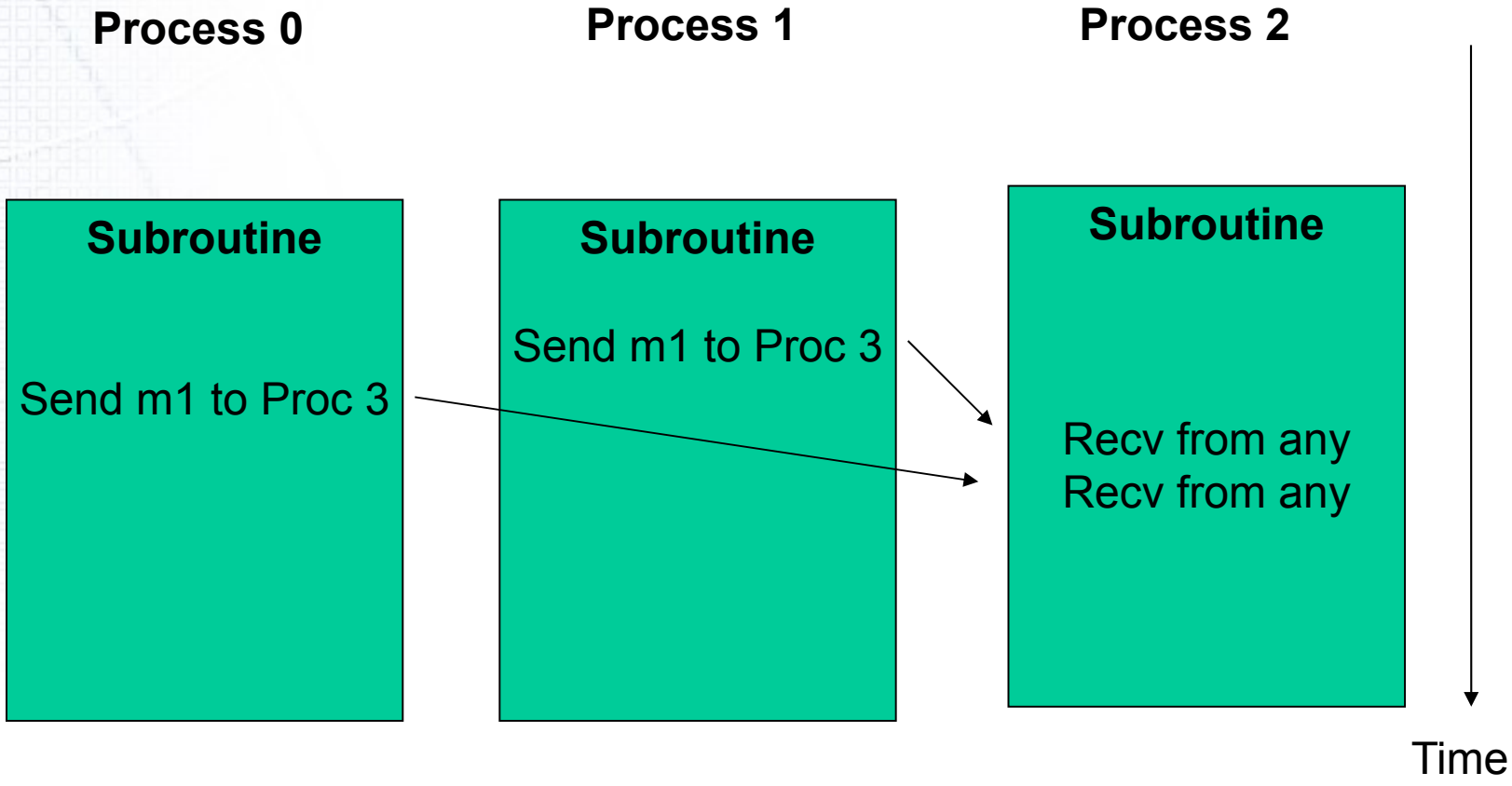
Why Buffers?

- Does this work?
  - YES, if the communication system buffers internally
  - NO, if the communication system does not use buffers (deadlock!)
- Hence: avoid this with non-blocking send operations or with an atomic sendreceive operation
- Typical buffering options:
  - nothing specified: buffering possible, but not mandatory (standard; users must not rely on buffering)
  - guaranteed buffering: problems if there is not enough memory
  - no buffering: efficient, if buffering is not necessary (due to the algorithm, for example)

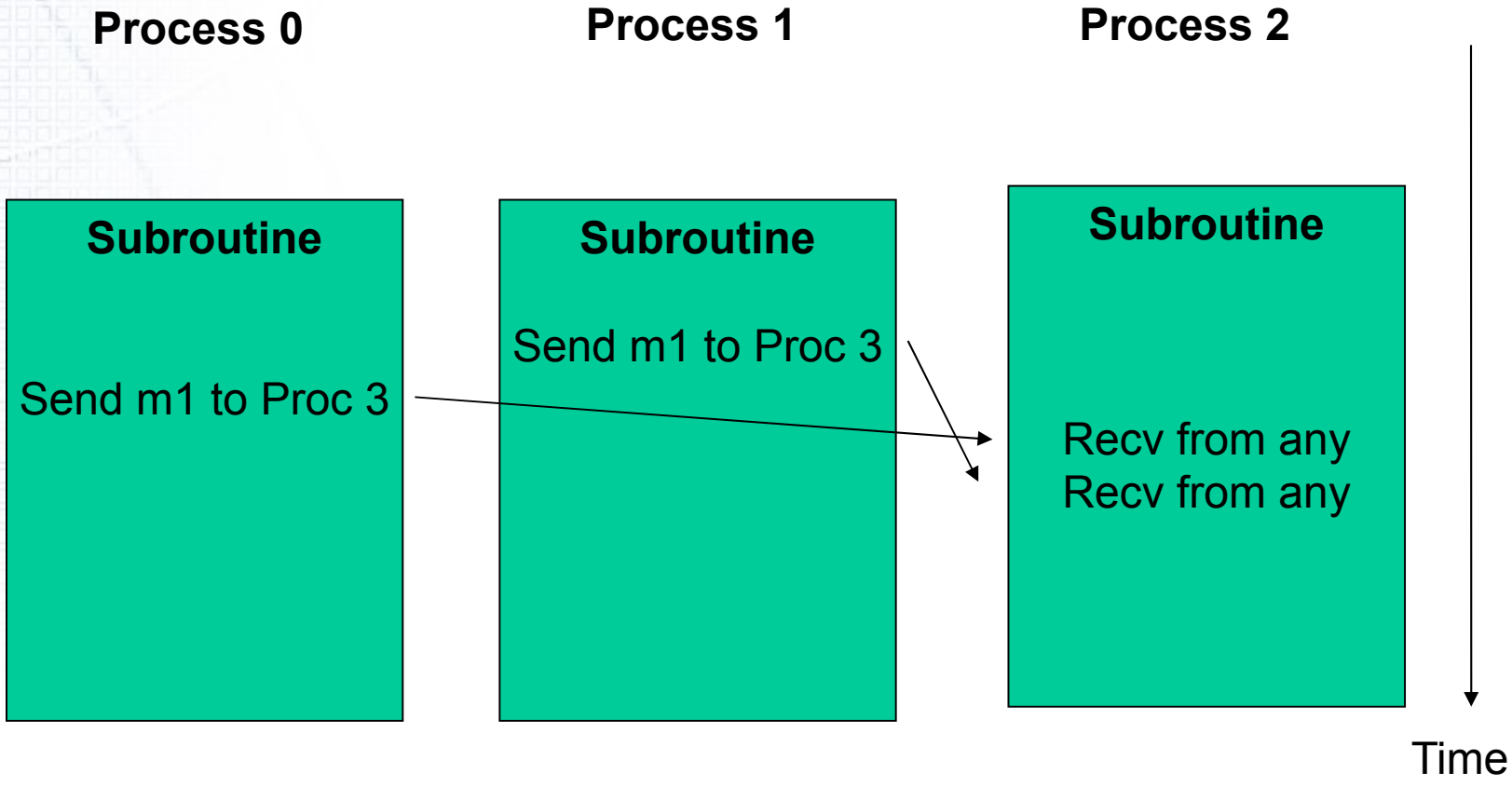
Keeping the Order

- Problem: there is no global time in a distributed system
- Consequence: there may be wrong send-receive assignments due to a changed order of occurrence
  - typically no problem for only one channel P1 ↔ P2
  - may be a problem if more processes communicate and if sender is specified via a wild card

Keeping the Order



Keeping the Order



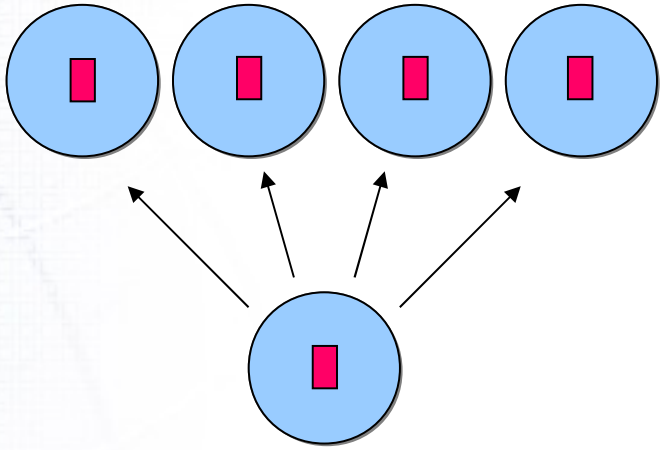
Collective Communication

- Many applications require not only a point-to-point communication, but also collective communication operations.

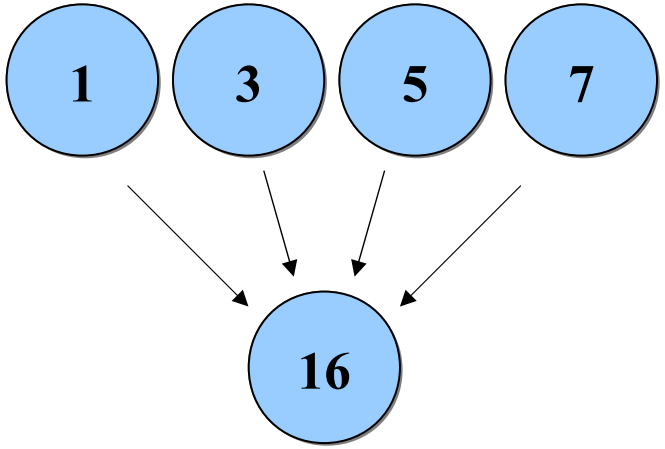
  A collective communication always involves data sharing in the specified communicator, which we mean every process in the group associated with the communicator.

e.g. broadcast, scatter, gather, etc.

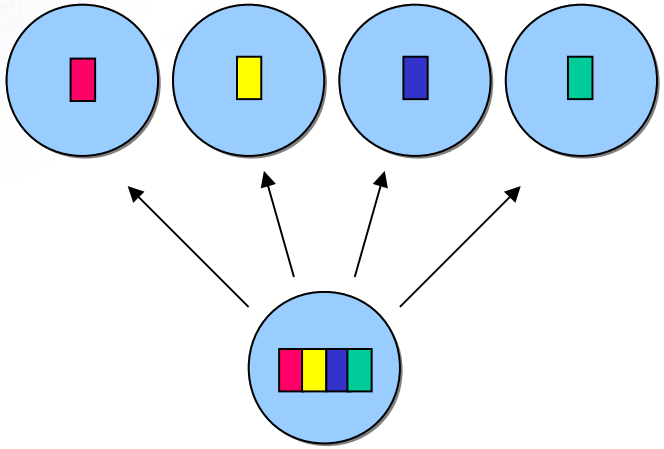
Collective Communication



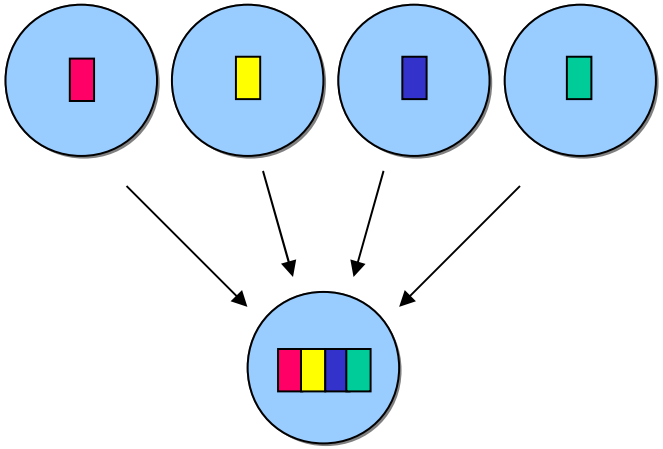
broadcast



reduction



scatter



gather

Message Types

- Data messages:
  - Meaning: data are exchanged in order to provide other processes' input for further computations
  - Example: interface values in a domain-decomposition parallelization of a PDE solution
- Control messages:
  - Meaning: data are exchanged in order to control the other processes' continuation
  - Examples: a global error estimator indicates whether a finite element mesh should be refined or not; a flag determines what to do next

Efficiency

- Avoid short messages: latency reduces the effective bandwidth

  $t_{comm} = t_{latency} + n/B$ (n: message size, B: bandwidth)

  $B_{eff} = n / t_{comm}$
- Computation should dominate communication!
- Typical conflict for numerical simulations:
  - overall runtime suggests large numbers of p processes
  - communication-computation ratio and message size suggest small p
- Try to find (machine- and problem-dependent) optimum number of processes
- Try to avoid communication points at all

MPI – The Message Passing Interface

MPI

- Objectives
  - Define an international long-term standard API for portable parallel applications and get all hardware vendors involved in implementations of this standard;
  - Define a target system for parallelizing compilers.
- The MPI Forum (http://www.mpi-forum.org/) brings together all contributing parties
- Most widespread implementations:
  - MPICH (Argonne Nat'l Lab, http://www-unix.mcs.anl.gov/mpi),
  - LAM (Indiana University, http://www.lam-mpi.org),...

Programming with MPI

- An MPI implementation consists of
  - a subroutine library with all MPI functions
  - include files for the calling application program
  - some startup script (usually called mpirun, but not standardized)
- Include the lib file mpi.h (or however called) into the source code
- Libraries available for all major imperative languages (C, C++, Fortran …)
- Initialize the MPI environment:

  ```
  MPI_Init(int *argc, **argv)
  ```

- Get your own process ID (rank):

  ```
  MPI_Comm_rank
  ```

- Get the number of processes (including oneself):

  ```
  MPI_Comm_size
  ```

Programming with MPI

- In error situations: terminate all processes of a process group

  `MPI_Abort`
- At the end of the program:

  `MPI_Finalize`
- After compilation: link the MPI library and (if necessary) lower communication libraries (depending on the concrete implementation)
- Program start:
  - mpirun
  - Use the program's name and the number of processes to be started as parameters

Point-to-Point Communication

Four types of point-to-point send operations, each of them available in a blocking and a non-blocking variant

	blocking	Non-blocking
Standard	`MPI_Send`	`MPI_Isend`
Buffered	`MPI_Bsend`	`MPI_Ibsend`
Synchronous	`MPI_Ssend`	`MPI_Issend`
Ready	`MPI_Rsend`	`MPI_Irsend`

Standard (regular) send: Asynchronous; the system decides whether or not to buffer messages to be sent

Buffered send: Asynchronous, but buffering of messages to be sent by the system is enforced

Synchronous send: Synchronous, i.e. the send operation is not completed before the receiver has started to receive the message

Ready send: Immediate send is enforced: if no corresponding receive operation is available, the result is undefined

Point-to-Point Communication

- Meaning of blocking or non-blocking communication (variants with 'I'):

  Blocking: the program will not return from the subroutine call until the copy to/from the system buffer has finished.

  Non-blocking: the program immediately returns from the subroutine call. It is not assured that the copy to/from the system buffer has completed so that user has to make sure of the completion of the copy.

Only one receive function:

- Blocking variant: **`MPI_Recv`**
  - Receive operation is completed when the message has been completely written into the receive buffer
- Non-blocking variant: **`MPI_Irecv`**
  - Continuation immediately after the receiving has begun
- Can be combined with each of the four send modes

- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

Point-to-Point Communication

- Syntax:

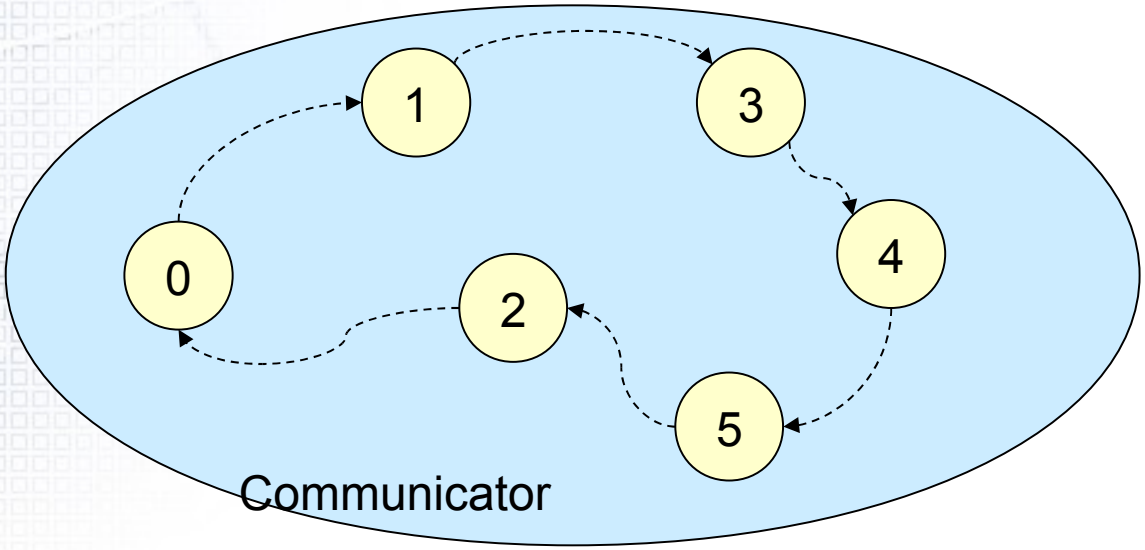
  `MPI_Send(buf,count,datatype,dest,tag,comm)`

  `MPI_Recv(buf,count,datatype,source,tag,comm,status)`

- where
  - int *buf — pointer to the buffer's begin
  - int count — number of data objects
  - int source — process ID of the sending process
  - int dest — process ID of the destination process
  - int tag — ID of the message
  - MPI_Datatype datatype — type of the data objects
  - MPI_Comm commcommunicator (see later)
  - MPI_Status *status — object containing message information
- In the non-blocking versions, there's one additional argument request for checking the completion of the communication.

Motivation for non-blocking communication

Blocking communication means that they do not return until the communication has completed.



Deadlock

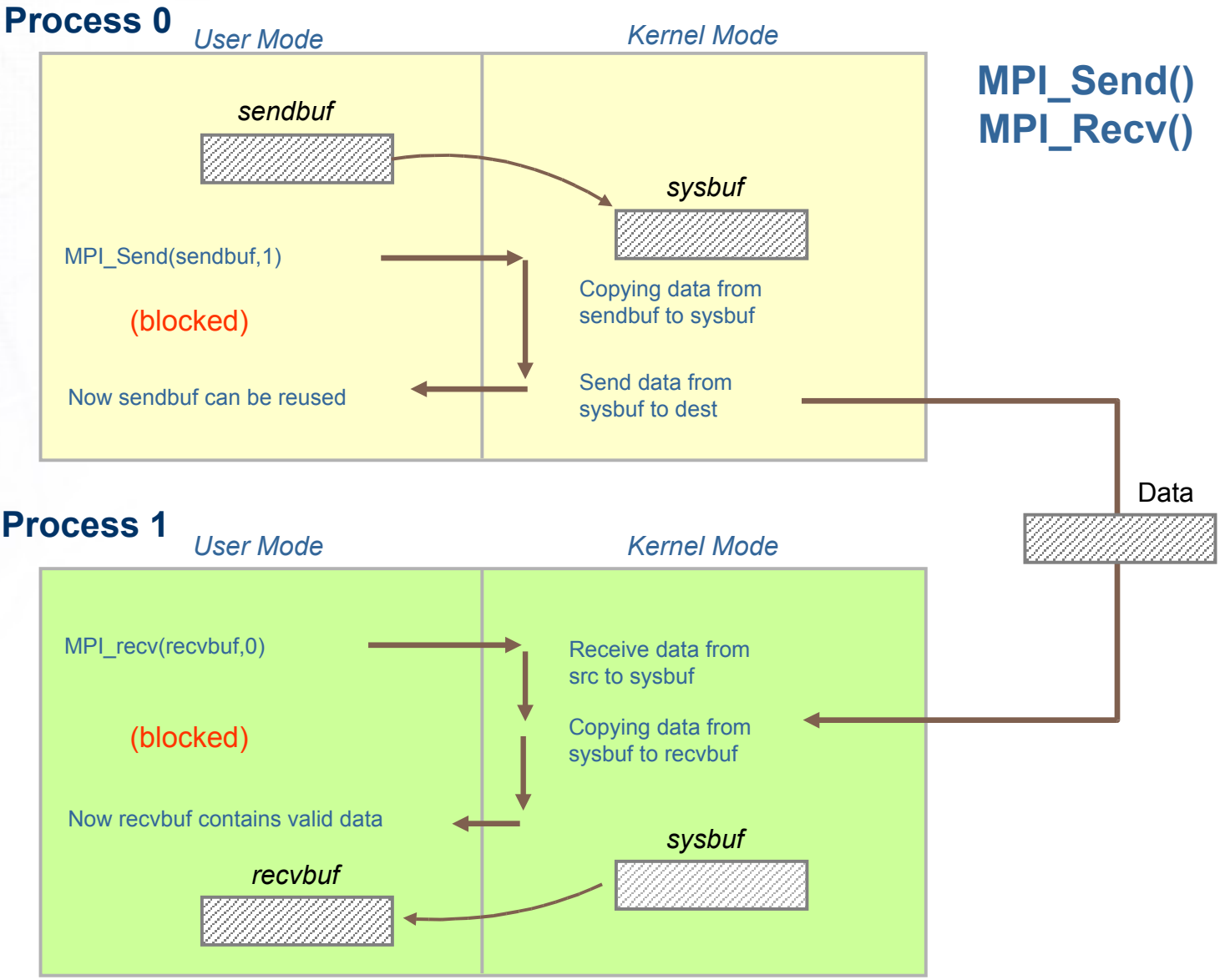
Two or more processes cannot proceed because they are both waiting for the other to release some resources (here is a response).

In case each process sends a message to another process using a standard send ,and then posts a receive.

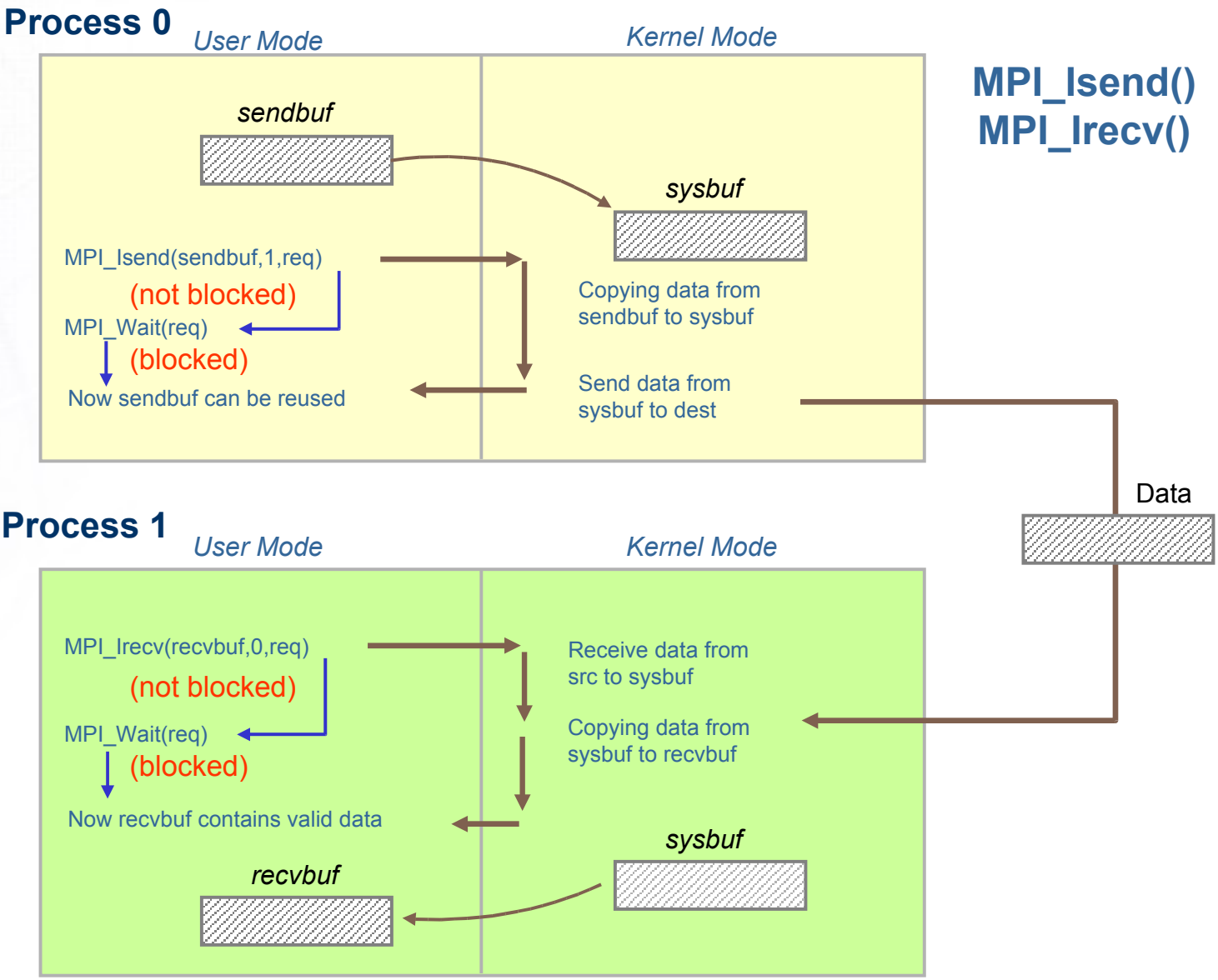
→ every process is sending and none is yet receiving,

→ *deadlock* can occur

MPI Send and Receive (blocking)



MPI Send and Receive (non-blocking)



Test Message Arrived

Used to check for non-blocking communication status.

`MPI_Buffer_attach(...)`:
lets MPI provide a buffer for sending

`MPI_Probe(...)`:
blocking test whether a message has arrived

`MPI_Iprobe(...)`:
non-blocking test whether a message has arrived

`MPI_Get_count(...)`:
provides the length of a message received

Used to check for completion of non-blocking communication.

`MPI_Test(...)`:
checks whether a send or receive operation is completed

`MPI_Wait(...)`:
causes the process to wait until a send or receive operation has been completed

Using Non-blocking Communication

- Method 1: MPI_Wait

```
MPI_Irecv(buf,…,req);
…do work not using buf
MPI_Wait(req,status);
…do work using buf
```

- Method 2: MPI_Test

```
MPI_Irecv(buf,…,req);
MPI_Test(req,flag,status);
while (flag != 0) {
    …do work not using buf
    MPI_Test(req,flag,status);
}
…do work using buf
```

Packing and Unpacking

- Elements of a complex data structure can be packed, sent, and unpacked again element by element: expensive and error-prone
- Faster alternative: send everything byte-wise, ignoring the structure; not applicable to heterogeneous clusters for lack of data format control
- Second alternative: extend the existing set of MPI data types and use standard commands like MPI_SEND or MPI_RECV afterwards
- MPI functions for explicit packing and unpacking:
  - **`MPI_Pack(...)`**:
    Packs data into a buffer
  - **`MPI_Unpack(...)`**:
    unpacks data from the buffer
  - **`MPI_Type_contiguous(...)`**:
    support for type conversion
  - **`MPI_Type_vector(...)`**:
    constructs an MPI array with element-to-element distance stride
  - **`MPI_Type_struct(...)`**:
    constructs an MPI record (complex data structure to be used as a standard MPI data type afterwards)

Standard MPI Datatypes

MPI datatype			
MPI Fortran		C	
MPI_CHARACTER	character(1)	MPI_CHAR	signed char
		MPI_SHORT	signed short int
MPI_INTEGER	integer	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_REAL	real	MPI_FLOAT	float
MPI_DOUBLE_PRECISION	double precision	MPI_DOUBLE	double
		MPI_LONG_DOUBLE	long double
MPI_COMPLEX	complex		

Simple Example

```
if (myrank == 0) {
    buf[0]=365;
    buf[1]=366;
    MPI_Send(buf,2,MPI_INT,1,10,MPI_COMM_WORLD);
}
else {

    MPI_Recv(buf,2,MPI_INT,0,10,MPI_COMM_WORLD,
    &status);
    MPI_Get_count(&status,MPI_INT,mess_length);
    mess_tag=status.MPI_TAG;
    mess_sender=status.MPI_SOURCE;
}
```

Process Groups and Communicators

- Messages are tagged for identification – message tag is message ID!
- Again: process groups for restricted message exchange and restricted collective communication
  - In MPI-1 static process groups only
  - Process groups are ordered sets of processes
  - Each process is locally uniquely identified via its local (group-related) process ID or rank
  - Ordering starts with zero, successive numbering
  - Global identification of a process via the pair (process group, rank)

Process Groups and Communicators

- MPI communicators: concept for working with contexts
  - Communicator = process group + message context
  - Message identification via the pair (context, tag)
  - Context may be a subroutine library
  - MPI offers intra-communicators for collective communication within a process group and inter-communicators for (point-to-point) communication between two process groups
  - Default (including all processes): **`MPI_COMM_WORLD`**
- MPI provides a lot of functions for working with process groups and communicators

Collective Communication

- Important application scenario:
  - distribute the elements of vectors or matrices among several processors
- Collective communication
- Some functions offered by MPI
  - **`MPI_Barrier(...)`**:
    synchronization barrier: process waits for the other group members; when all of them have reached the barrier, they can continue
  - **`MPI_Bcast(...)`**:
    sends the data to all members of the group given by a communicator (hence more a multicast than a broadcast)
  - **`MPI_Gather(...)`**:
    collects data from the group members

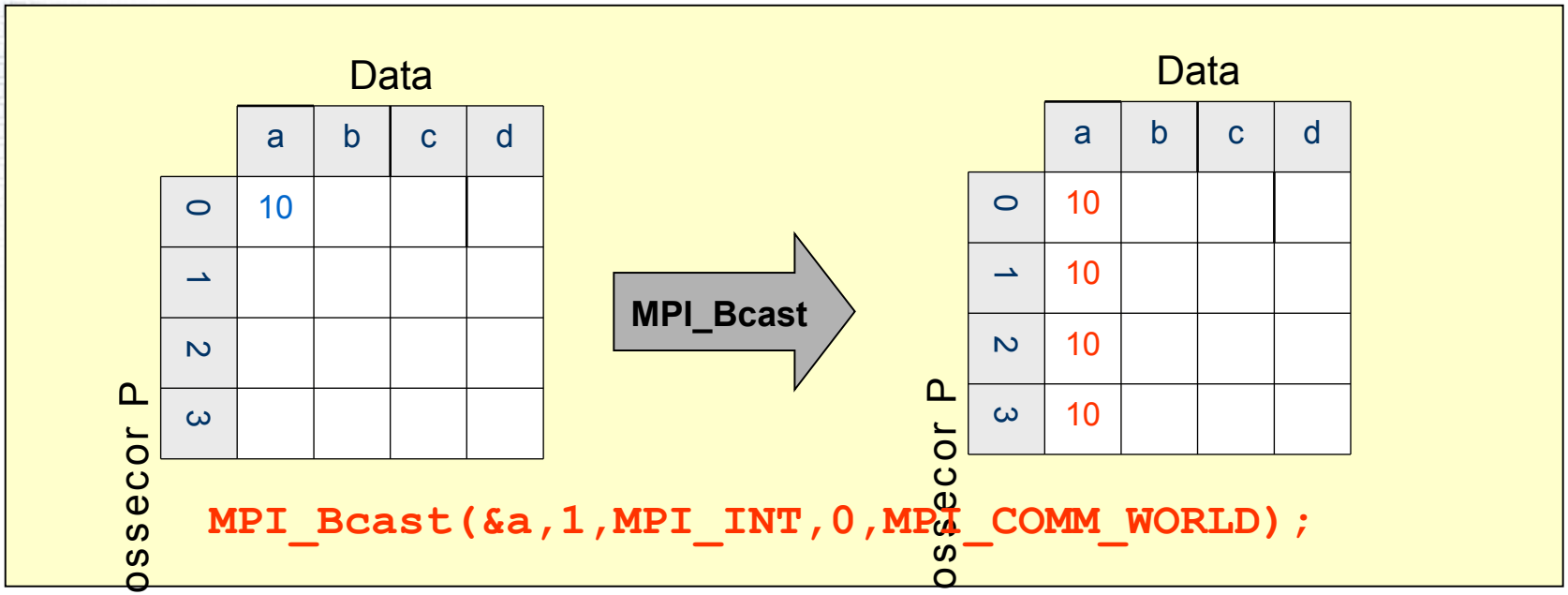
Collective Communication

- **`MPI_Allgather(...)`**:
  gather-to-all: data are collected from all processes, and all get the collection
- **`MPI_Scatter(...)`**:
  classical scatter operation: distribution of data among processes
- **`MPI_Reduce(...)`**:
  executes a reduce operation
- **`MPI_Allreduce(...)`**:
  executes a reduce operation where all processes get its result
- **`MPI_Op_create(...)`** and **`MPI_Op_free(...)`**:
  defines a new reduce operation or removes it, respectively

- Note that all of the functions above are with respect to a communicator (hence not necessarily a global communication)

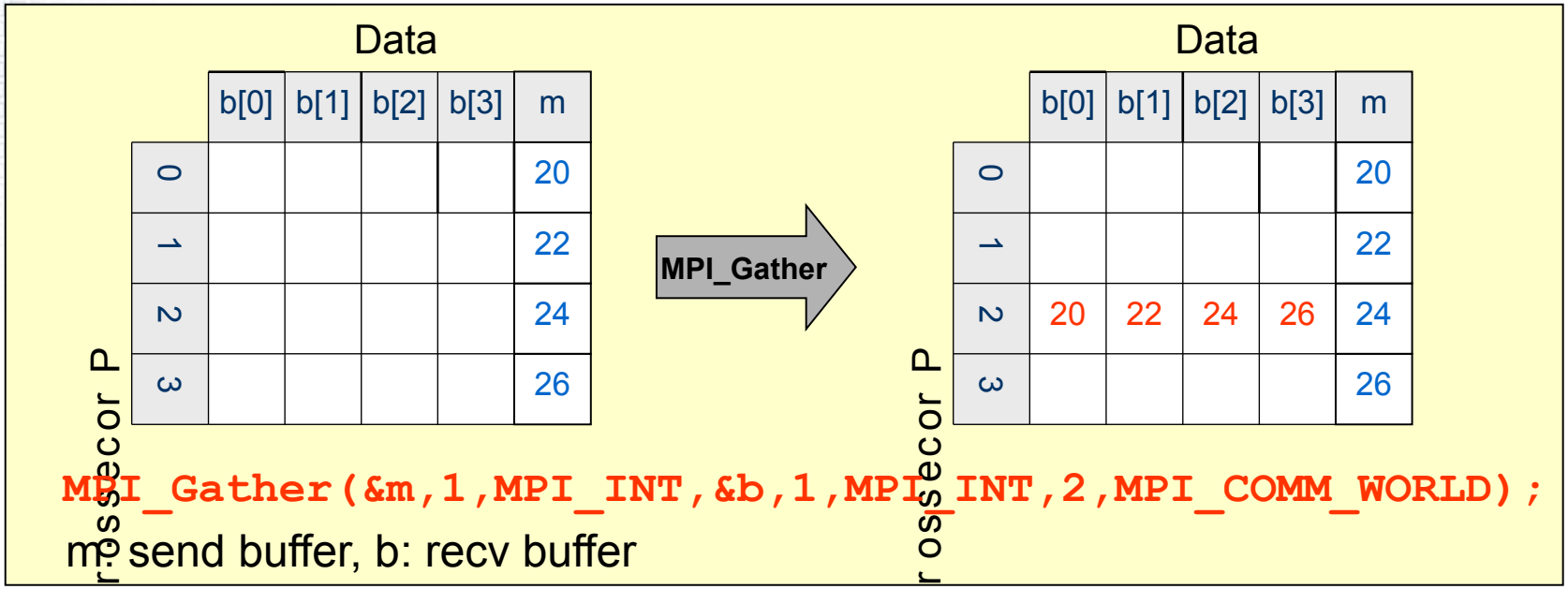
Broadcast

- Meaning: send the message to all participating processes
- Example: the first process that finds the solution in a competition informs everyone to stop



Gather

- Meaning: collect information from all participating processes
- Example: each process computes some part of the solution, which shall now be assembled by one process

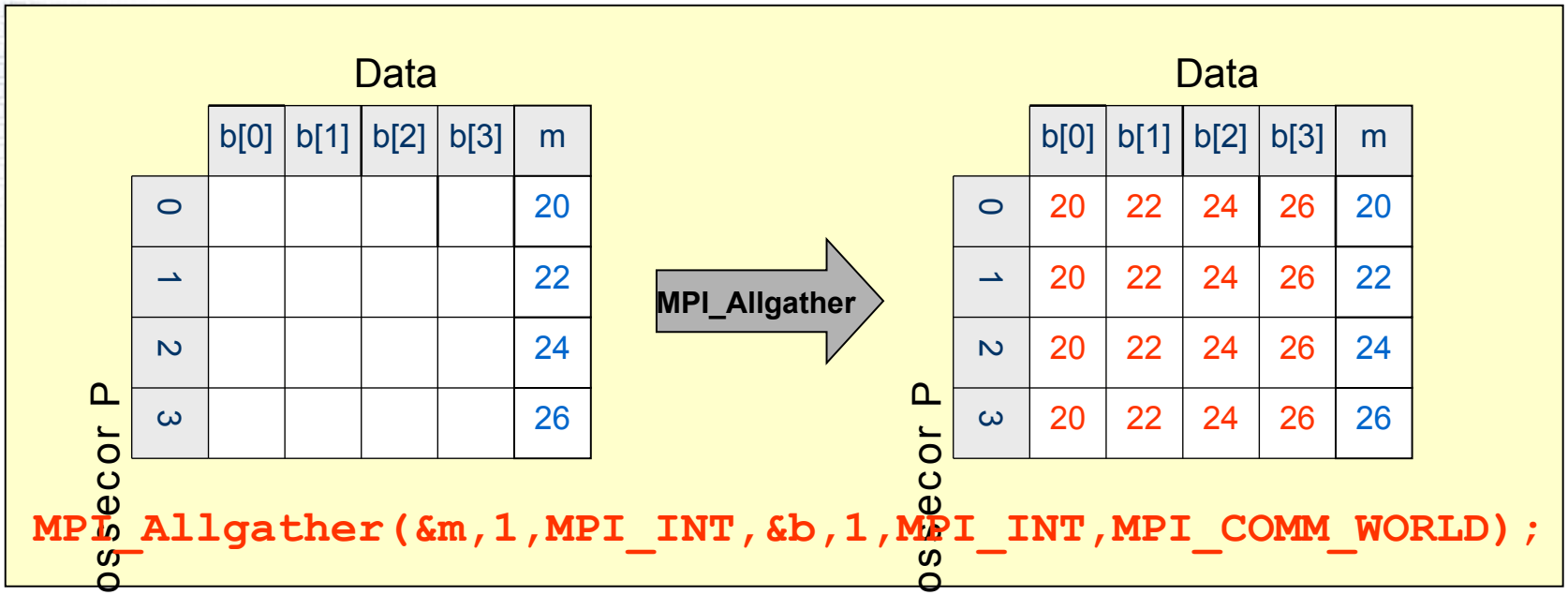



```
MPI_Gather(&m,1,MPI_INT,&b,1,MPI_INT,2,MPI_COMM_WORLD);
```

m: send buffer, b: recv buffer

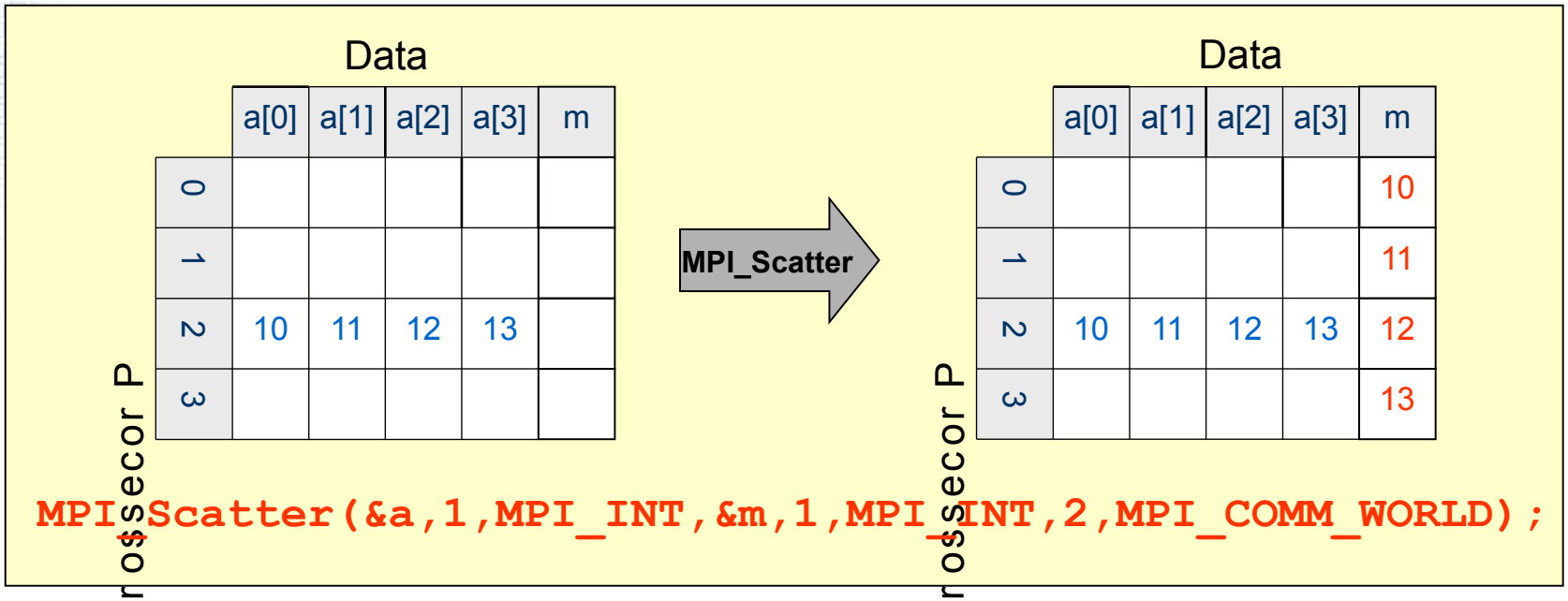
All Gather

- Meaning: like gather, but all participating processes assemble the collected information
- Example: as before, but now all processes need the complete solution for their continuation



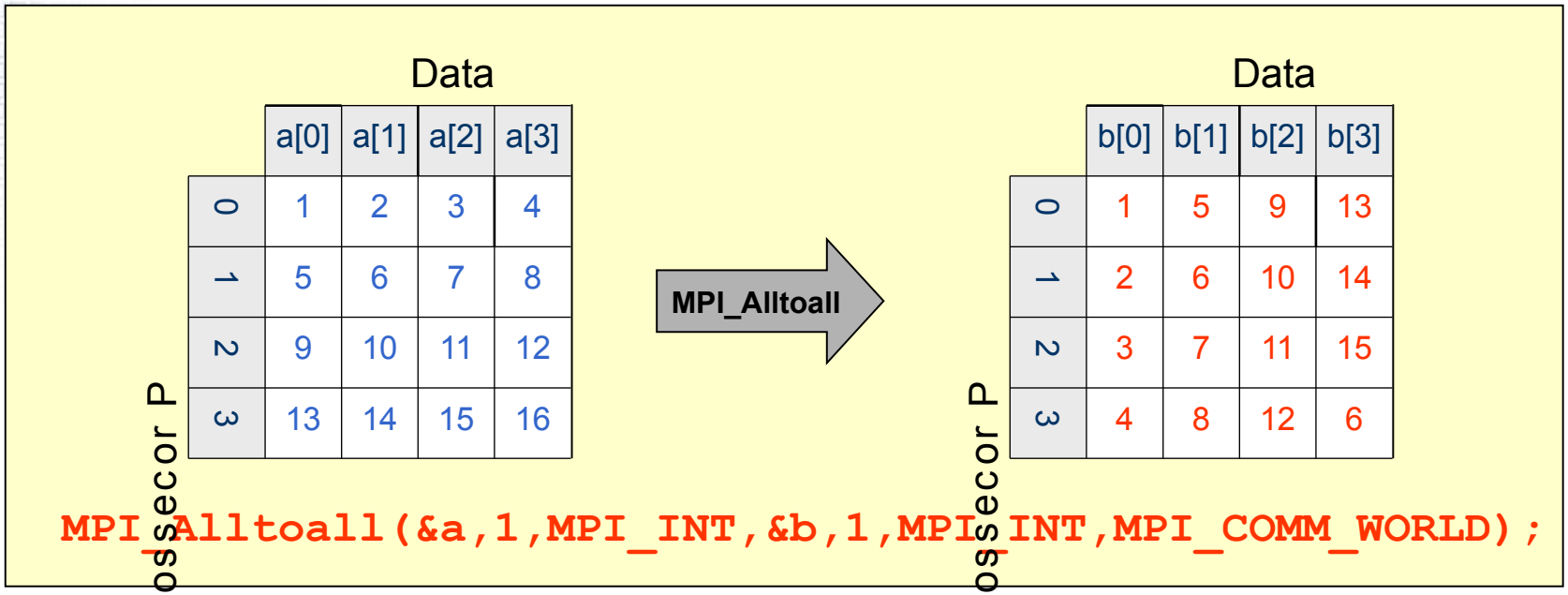
Scatter

- Meaning: distribute your data among the processes
- Example: two vectors are distributed in order to prepare a parallel computation of their scalar product



All to All

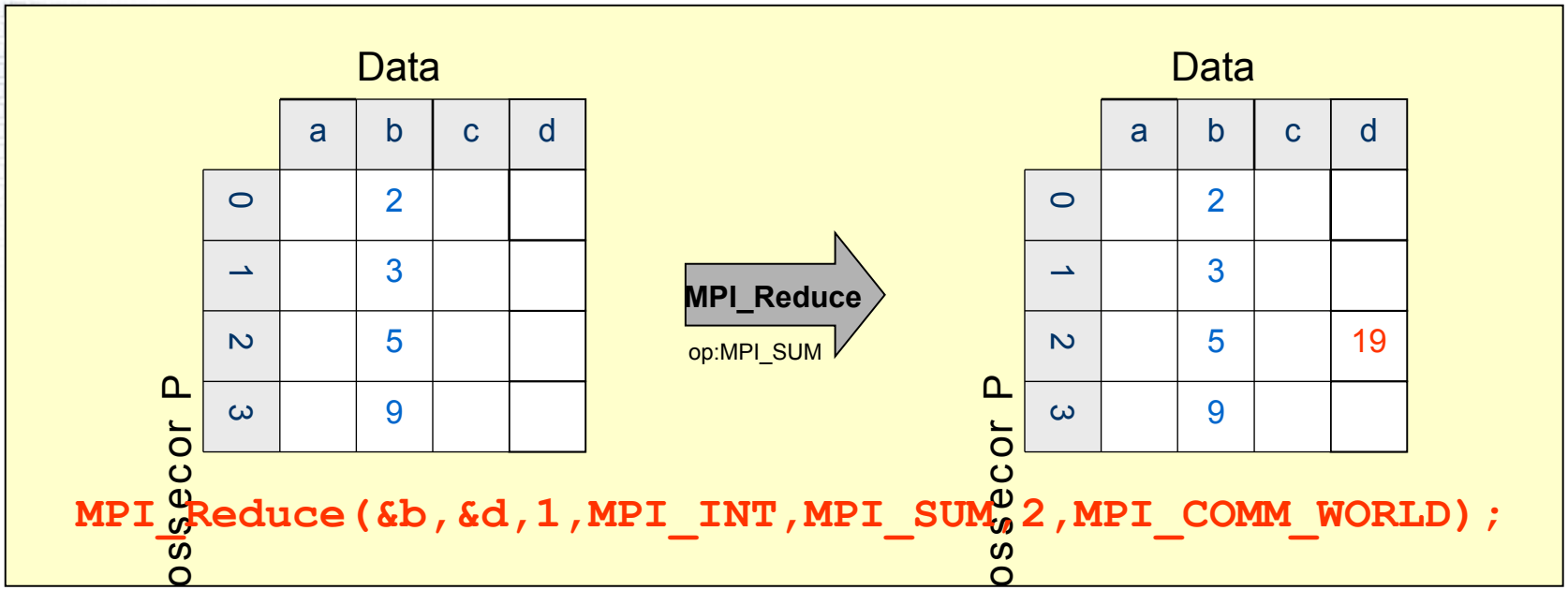
- Meaning: data of all processes are distributed among all processes




```
MPI_Alltoall(&a,1,MPI_INT,&b,1,MPI_INT,MPI_COMM_WORLD);
```

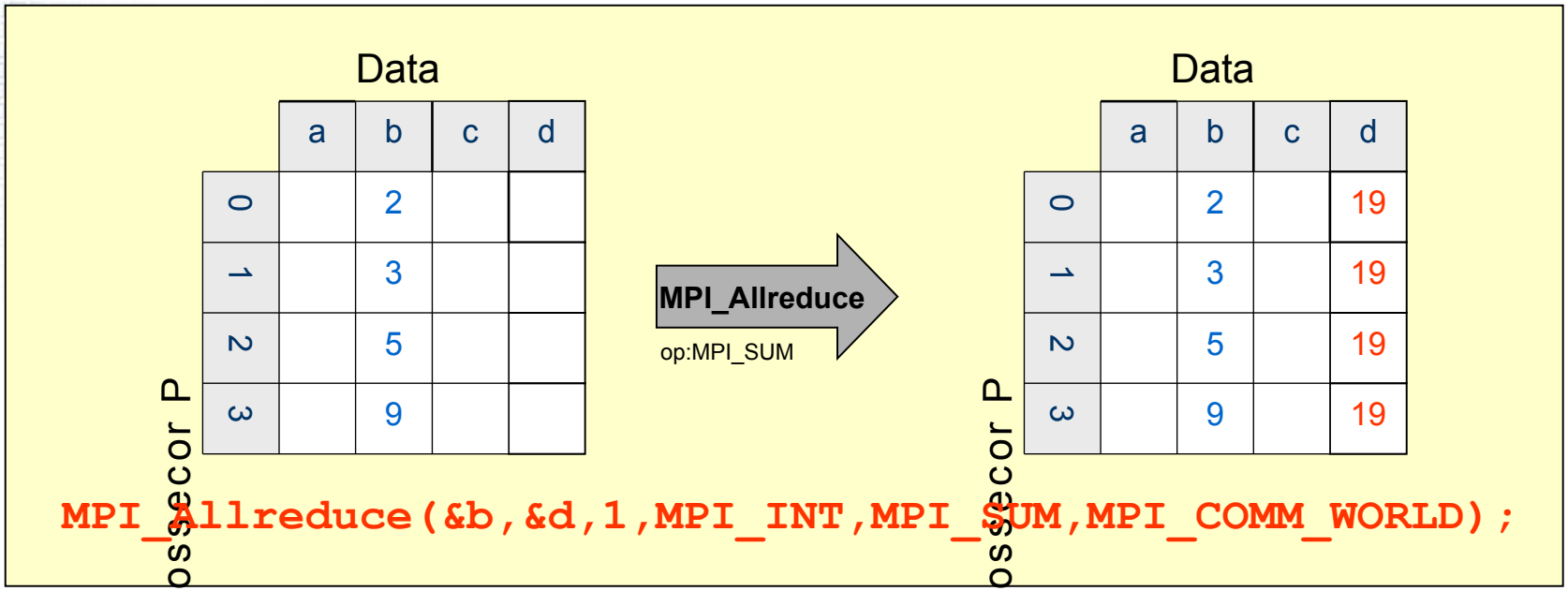
Reduce

- Meaning: information of all processes is used to provide a condensed result by/for one process
- Example: calculation of the global minimum of the variables kept by all processes, calculation of a global sum, etc.



All Reduce

- Meaning: like reduce, but condensed result is available for all processes
- Example: suppose the result is needed for the control of each process' continuation



Predefined Reduction Operations

MPI Name	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
MPI_MINLOC	Minimum and location

From MPI-1 to MPI-2

- Obvious drawbacks of MPI-1:
  - Restriction to SPMD program structure
  - No support of multithreading
  - No dynamic process creation nor management (like in PVM)
  - No standardization of functions for parallel I/O
  - Too many (hardly needed) functions
- Hence, MPI-2 provided improvements and extensions to MPI-1:
  - Now possible for dynamic creation and management of processes
  - Introduction of special communication functions for DSM systems
  - Extension of the collective communication features
  - Parallel I/O
  - C++ and FORTRAN 90 are supported, too

End