Parallel Programming with MPI and OpenMP

MPI

Hartmut Häfner, Steinbuch Centre for Computing (SCC)

STEINBUCH CENTRE FOR COMPUTING - SCC

Informations on MPI

Karlsruhe Institute of Technology

MPI-1: A Message-Passing Interface Standard (June,1995)
https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html

MPI-2: A Message-Passing Interface Standard (July, 1997)
https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

MPI-3: A Message-Passing Interface Standard (September, 2012)
https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

Marc Snir und William Gropp und andere:
MPI: The Complete Reference. (2-volume set). The MIT Press, 1998.
(MPI-1.2 und MPI-2 Standard in readable form)

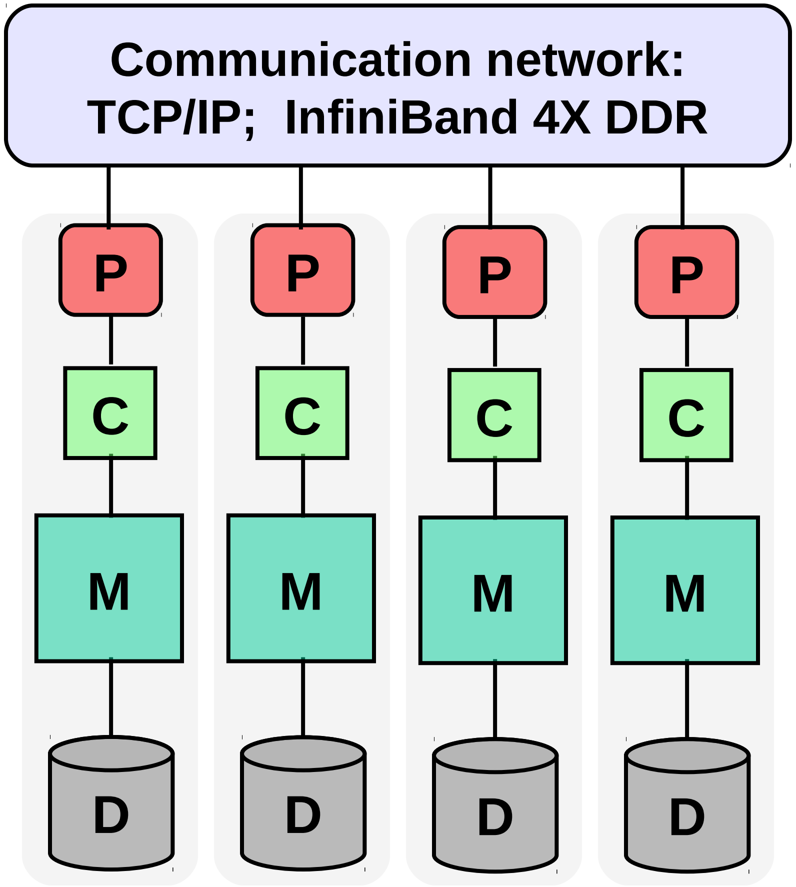
William Gropp, Ewing Lusk und Rajeev Thakur:
Using MPI, Third Edition: Portable Parallel Programming With the Message-Passing Interface, MIT Press, Nov. 2014, und
Using Advanced MPI: Advanced Features of the Message-Passing Interface.
MIT Press, Nov. 2014.

Peter S. Pacheco: Parallel Programming with MPI. Morgen Kaufmann Publishers, 1997

MPI-Tutorial vom Livermore Computing Center:
https://computing.llnl.gov/tutorials/mpi/

http://www.mpi-forum.org

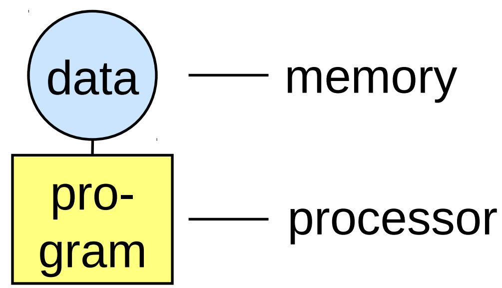
„Distributed Memory“ System



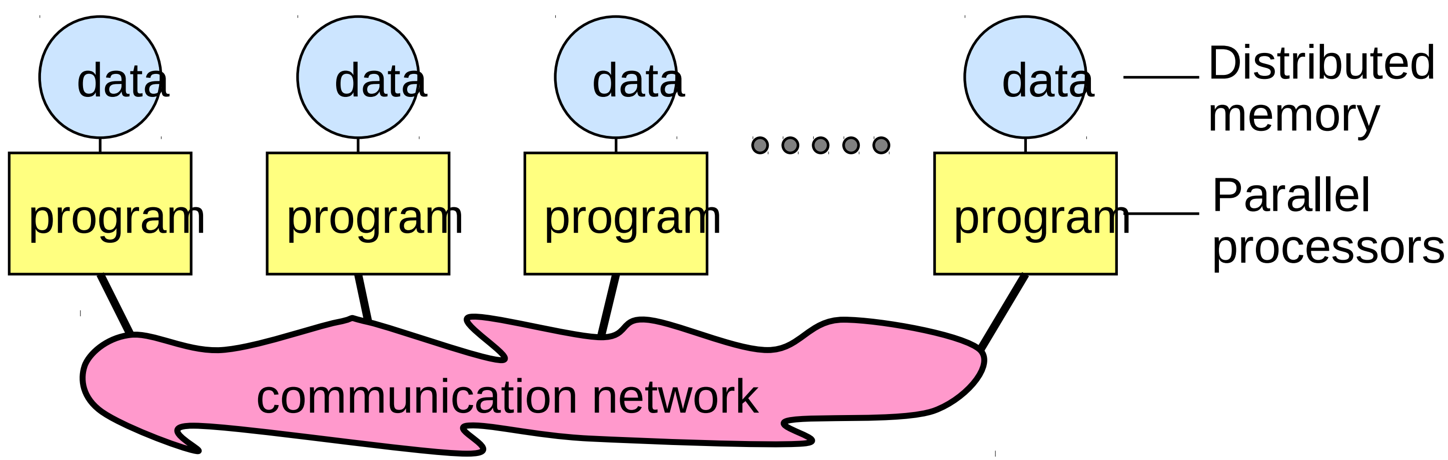
- Each node acts as independent computer system
- ONE copy of the operating system per node
- Each processor only accesses its own local memory
- Parallelization via “Message Passing Interface“
- Examples: HPC-systems at KIT, networked workstations

“Message Passing” Paradigm

Sequential paradigm

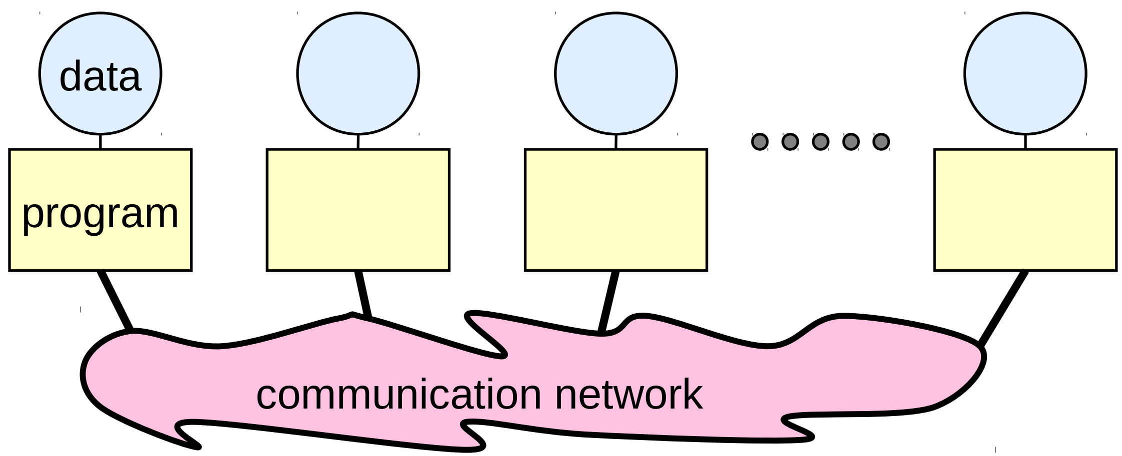


“Message-Passing” paradigm



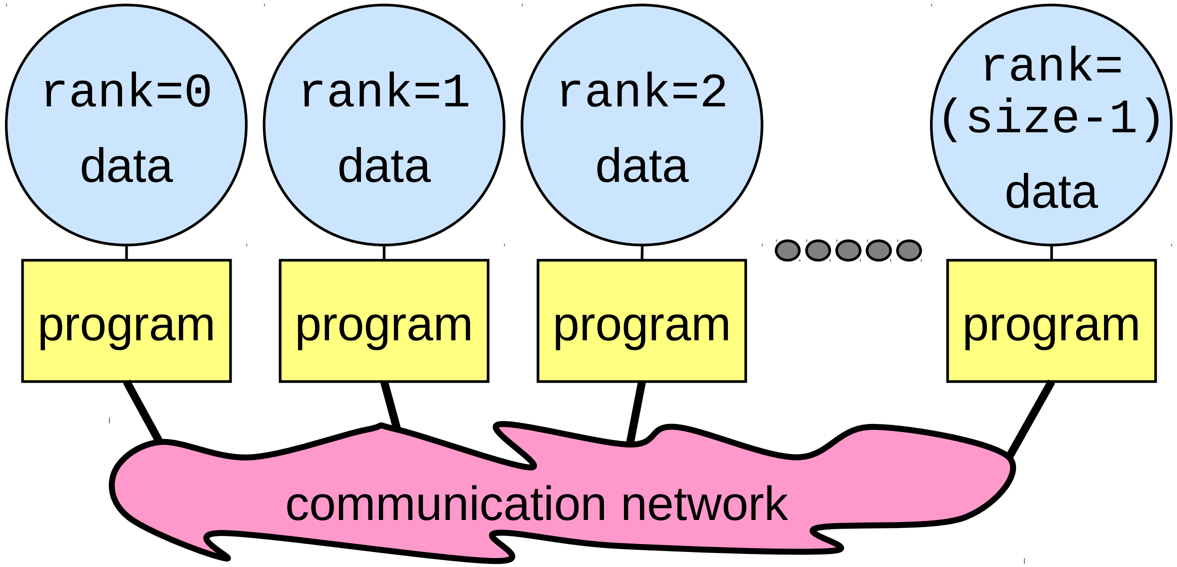
“Message Passing” Paradigm (2)

- On each processor runs one program (process) of a “message passing” program*:*
  - Written in a conventional programming language like e.g. C or Fortran
  - typically the same “Executable” on each processor (SPMD)
  - Variables of each program (process) have
    - the same name
    - But different locations (“distributed memory”) and different data
    - → all variables are private
  - Communicate via Send & Receive routines (“*Message Passing Interface”*)



Distribution of Work and Data

- **The value of the variable `rank` is determined by a MPI library routine**
- **All (`size`) processes are started by a MPI initialization-program (`mpirun` oder `mpiexec`)**
- **Which processes work on which data is based on the variable** rank



What is SPMD?

- Single Program, Multiple Data
- The same program runs on each processor, but on different datasets
- MPI also allows MPMD, i.e. Multiple Program, …
- MPMD can be emulated by SPMD

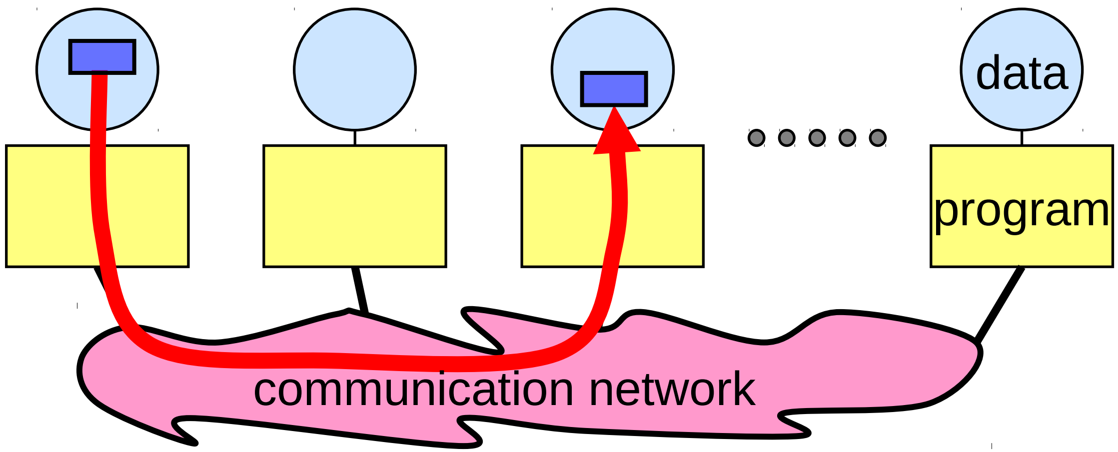
Emulation of MPMD by SPMD

Karlsruhe Institute of Technology

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

- ```
  program simulated_MPMD
  if (myrank < ... ) then
    call ocean( some arguments )
  else
    call weather( some arguments )
  endif
  end program simulated_MPMD
  ```

Messages



- Messages are data packages to be transferred from one process to another process
- Necessary Informations for the messages are:
  - Sending Process – Receiving process, i.e. their ranks
  - Location of the source – Location of the destination
  - Data type of the source – Data type of the destination
  - Data size of the source – Data size of the destination

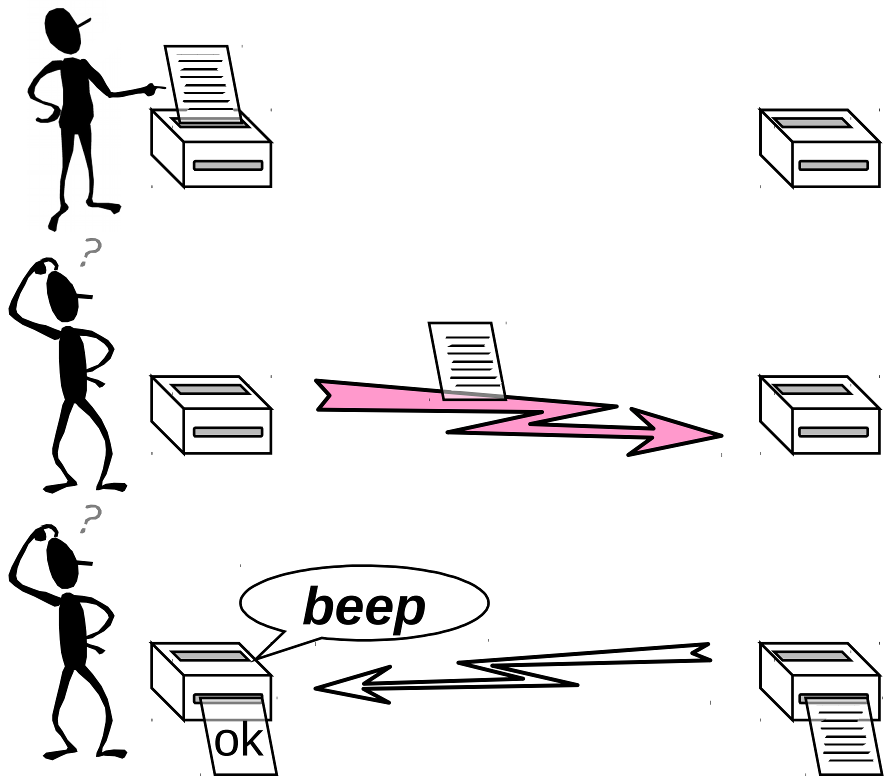
Point-to-Point (P2P) Communication

- Simplest form of “message passing”
- A process sends a message to another process
- Different typs of P2P communications
  - Synchronous Send
  - Asynchronous (buffered) Send

Synchronous Send

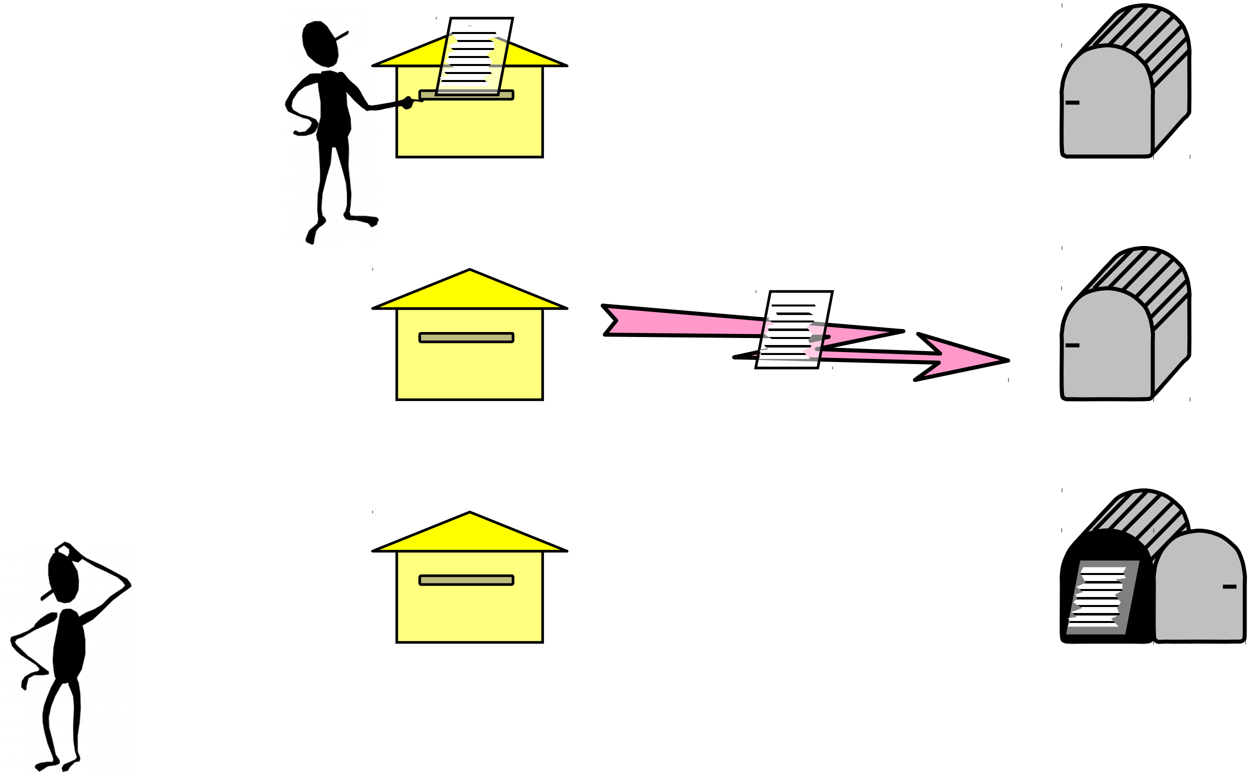
- Sending process gets information, that the message has been received

- Analogy to a Fax device.



Asynchronous (buffered) Send

- **One only knows, when the message has been sent**



Blocking Operations

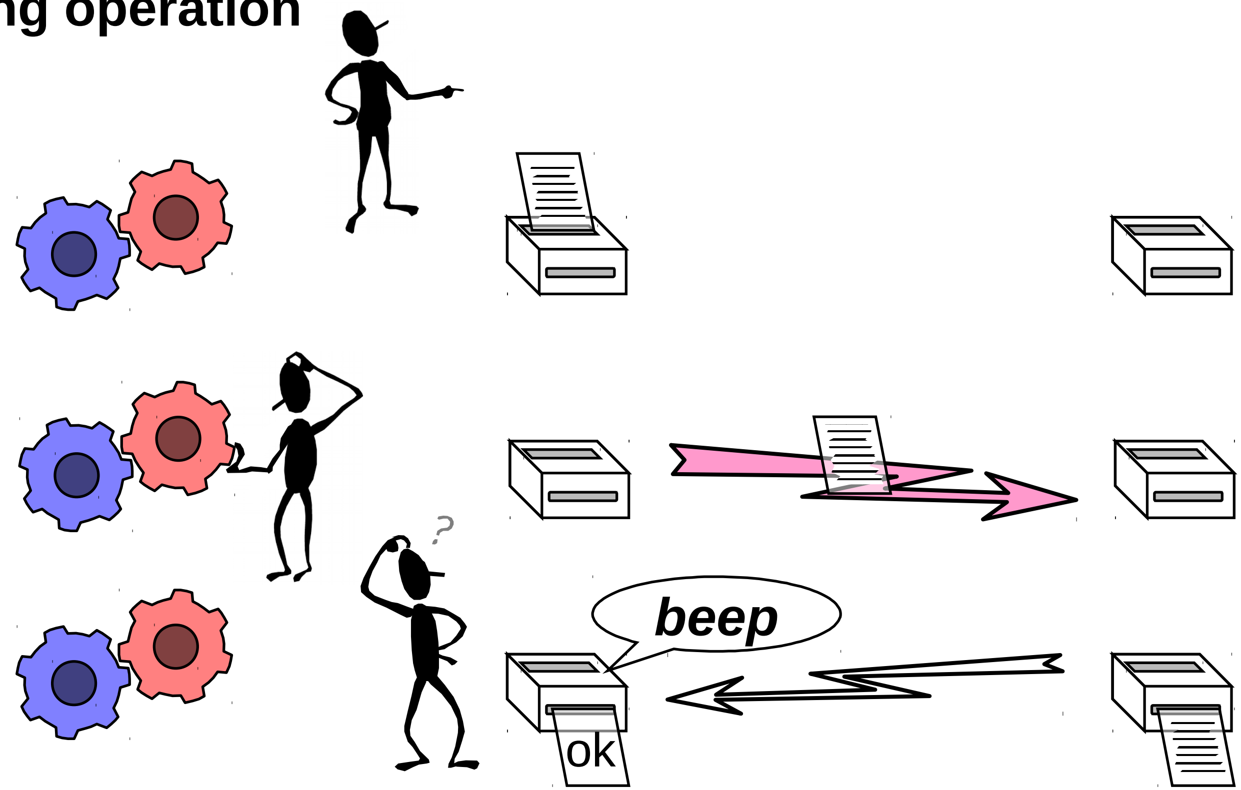
- Operations are local activities like sending oder receiving of a message
- Blocking Send- or Receive-subroutines will only be left, when the corresponding operation has been finished.
- *Synchronous Send*: Send-routine will only be left, when the message has been arrived at the receiving process

  A*synchronous Send*: Sending routine will only be left, when data have been completely sent

  Receiving: Receive-routine will only been left, when data have completely been stored in the storage of the application program

Non-blocking Operations

- Non-blocking operation: returns after initialization of communication (leaves routine) and allows the calling process to go on with executing code
- Before accessing the sent data the process must call a WAIT-routine and thus wait on the end of execution of the non-blocking operation



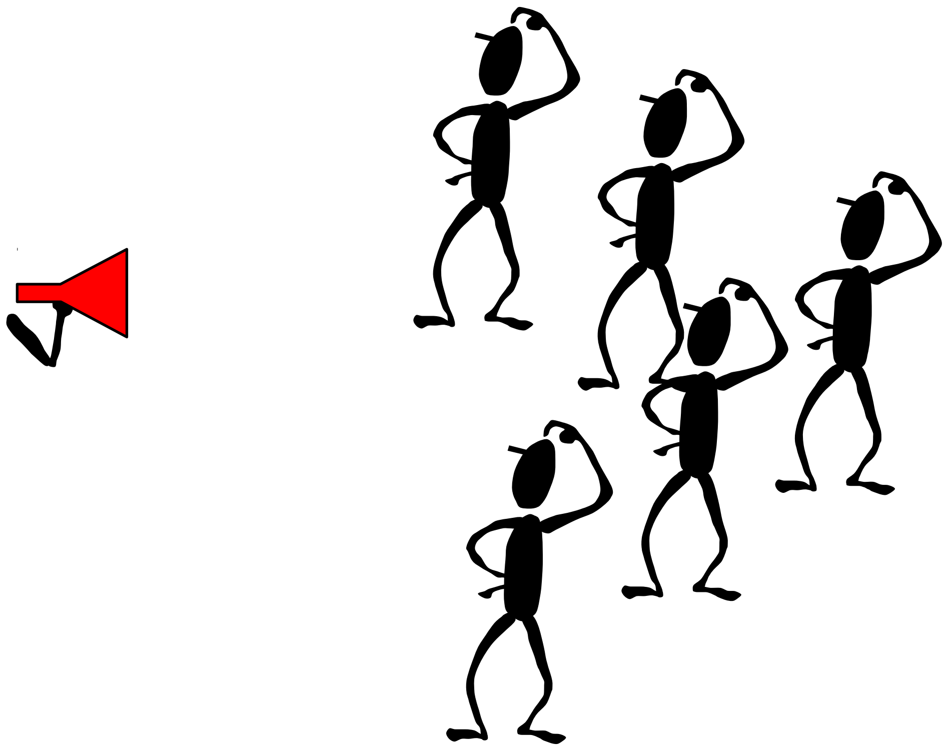
Collective Communications

Karlsruhe Institute of Technology

- Many processes are concurrently involved
- Usually optimized implementations from the MPI provider like e.g. “tree based” algorithms
- Can be implemented from P2P-routines

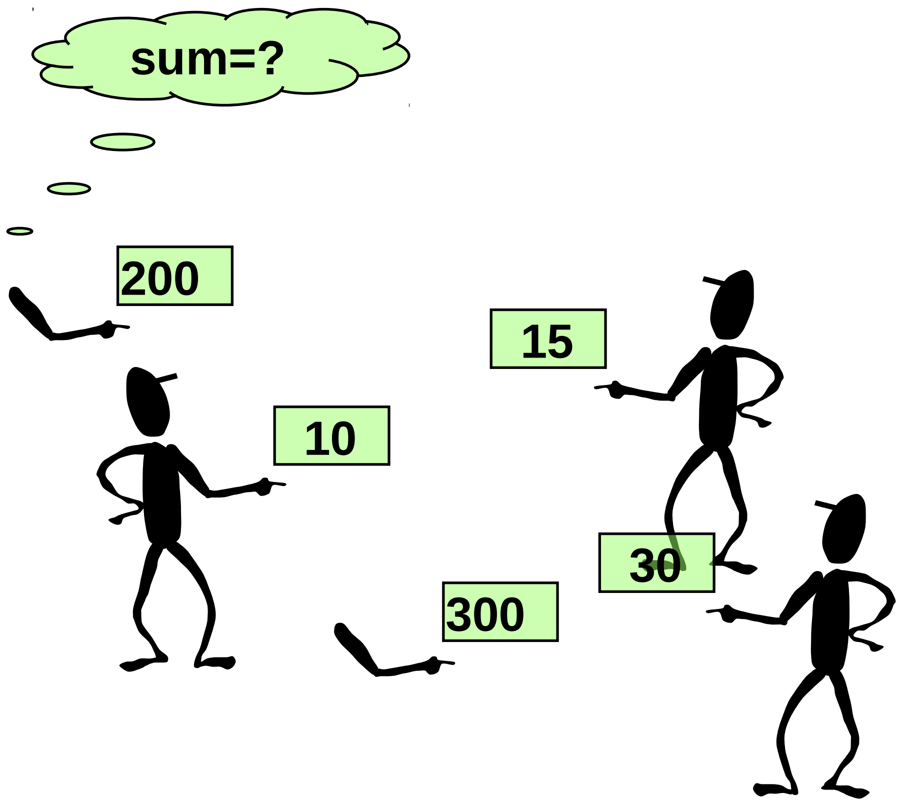
Broadcast

- "One-to-all" communication



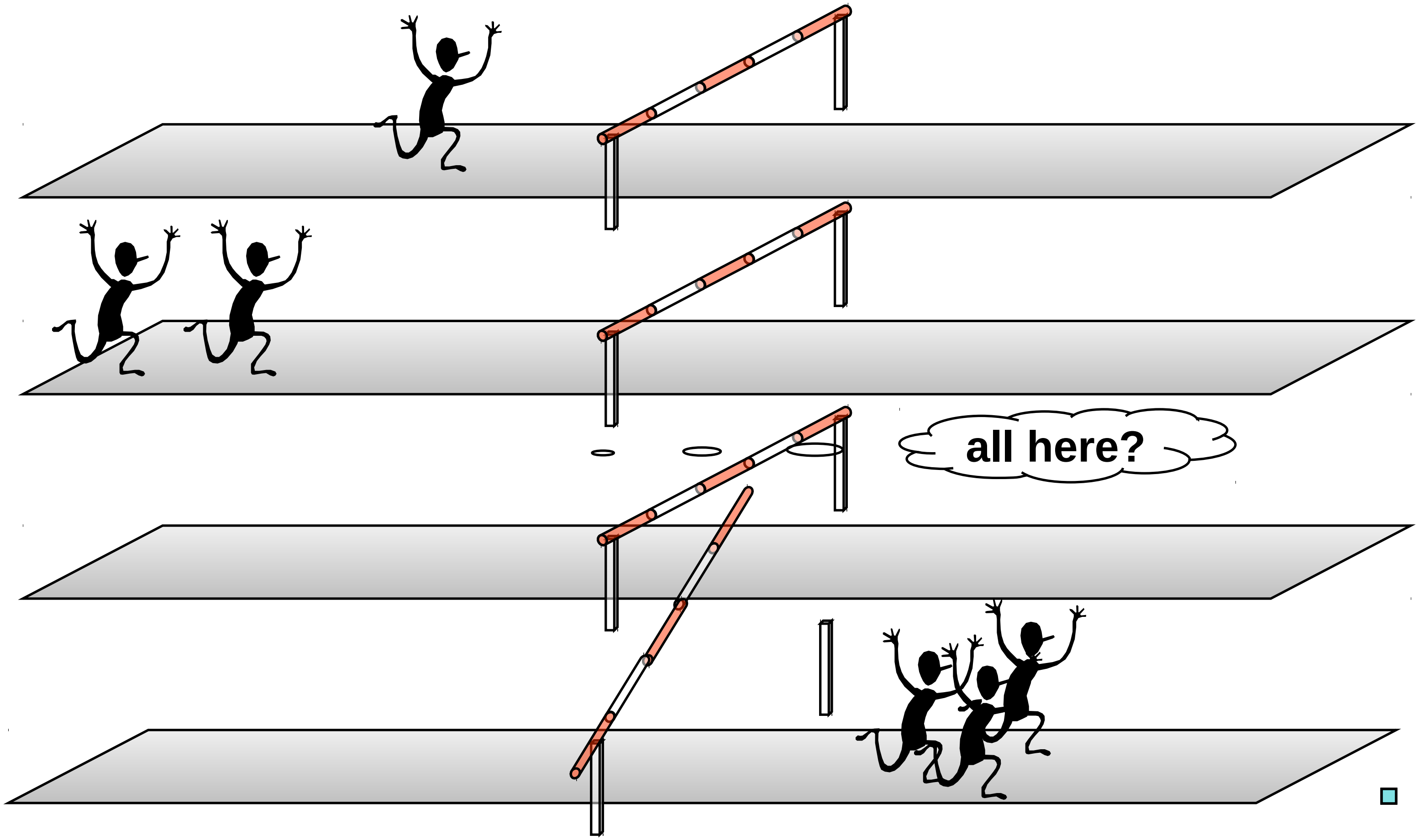
Reduction-Operations

- **Combine data from (all) processes to compute a single result**



Barriers

- Synchronize (all) processes



Initialization and Termination of MPI

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

```
int MPI_Init( int *argc, char ***argv)
 …
int MPI_finalize()
```

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

- Fortran:

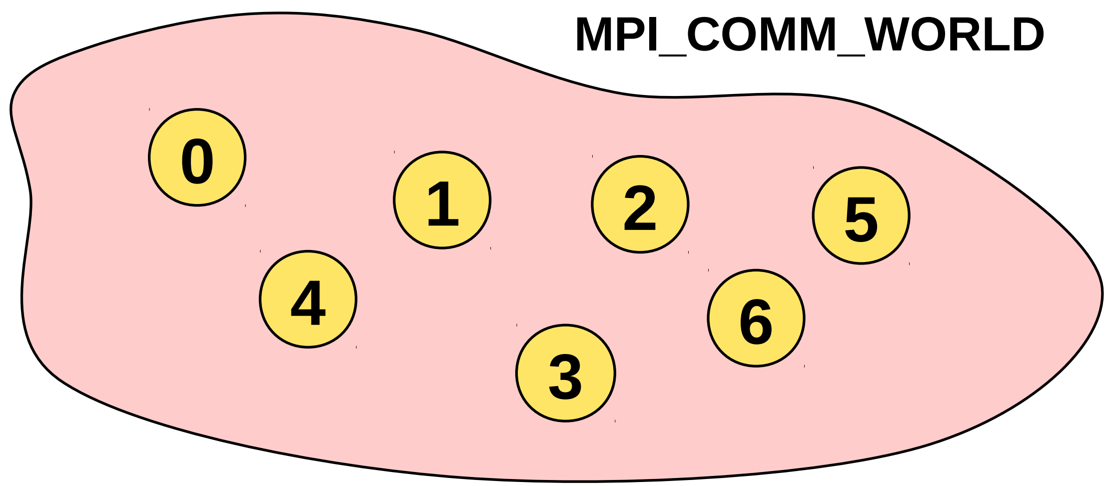
```
MPI_INIT( IERROR )
 INTEGER IERROR
 …
 MPI_FINALIZE(IERROR)
```

```
program xxxxx
implicit none
include ´mpif.h´
integer ierror
call MPI_Init(ierror)
....
call MPI_Finalize(ierror)
```

- **MPI_INIT must be the first MPI routine; after MPI_Finalize further MPI-commands or a re-initialization of MPI are forbidden!**

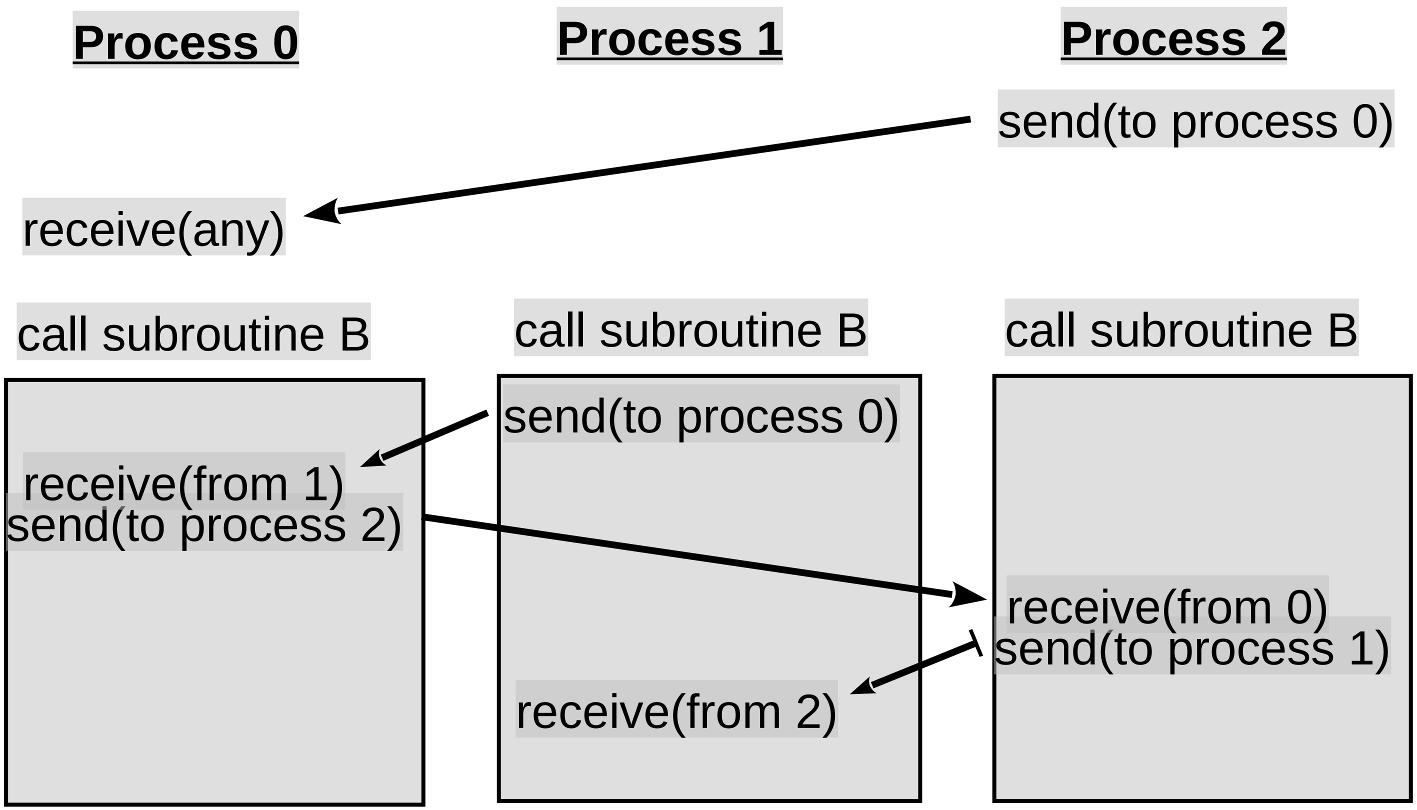
Communicator MPI_COMM_WORLD

- All processes of a MPI-program normally use the 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 with the numbers 0..(size-1)

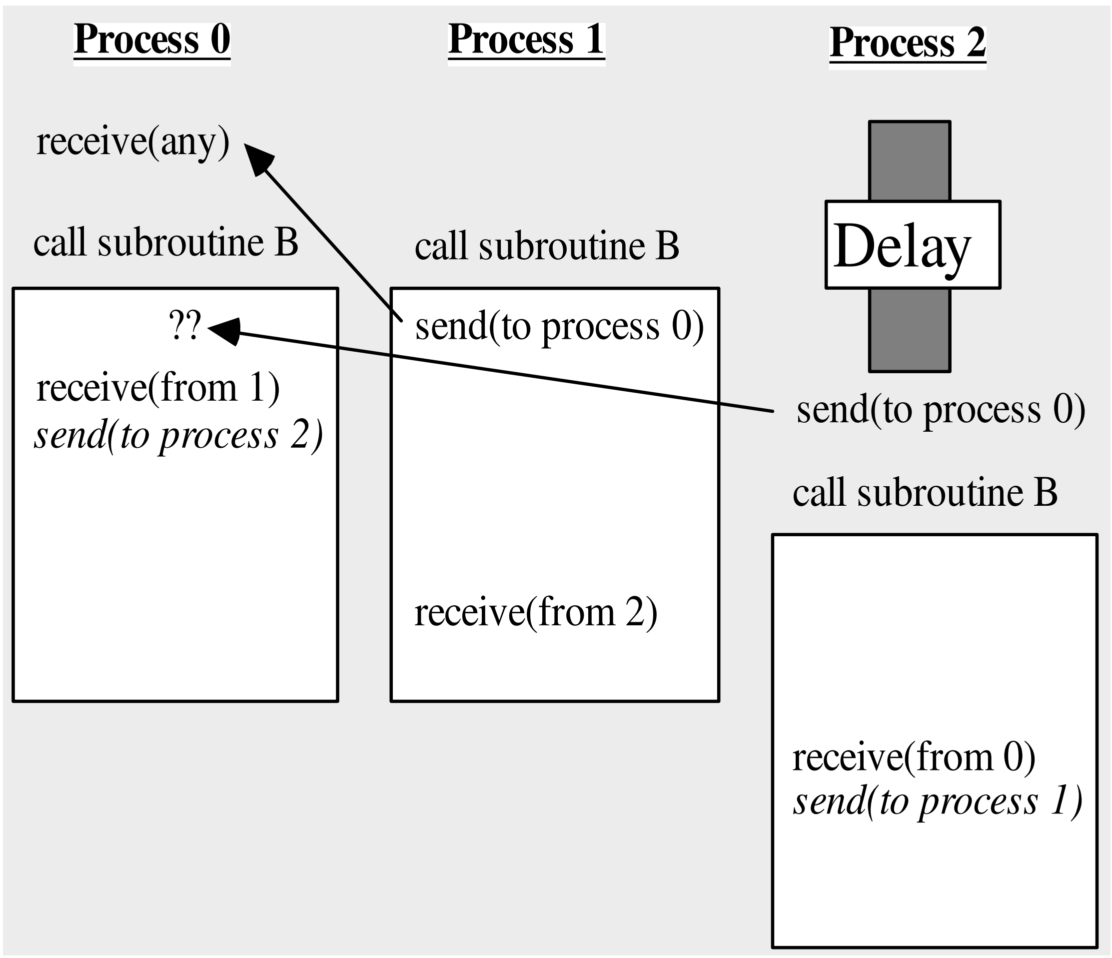


Why the context must be considered?

because of the usage of libraries!



MPI_COMM_WORLD: What can happen?!



In this example the communicator restricts the messages on their context!

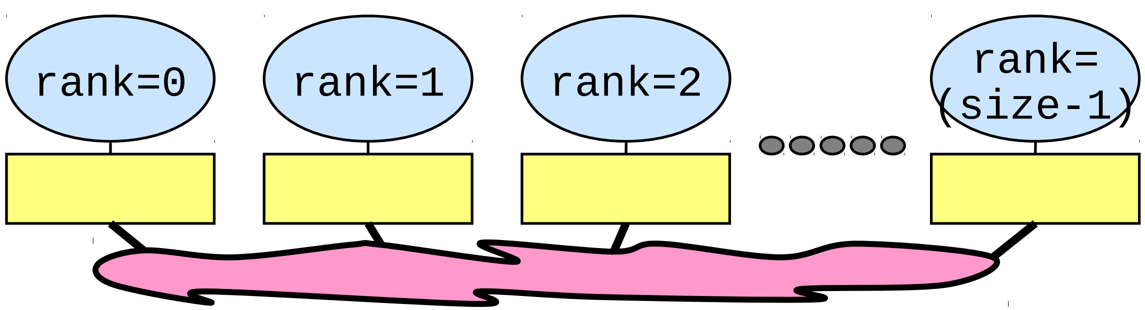
2 necessary MPI-commands

- The rank identifies different processes with values between `0` and `size-1`; the value of the variable `size` is returned by the MPI-system!
- The rank is the base for parallel code and the distribution of data
- C:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- Fortran:

```
INTEGER comm, rank, size, ierror
MPI_COMM_RANK( comm, rank, ierror)
MPI_COMM_SIZE( comm, size, ierror)
```

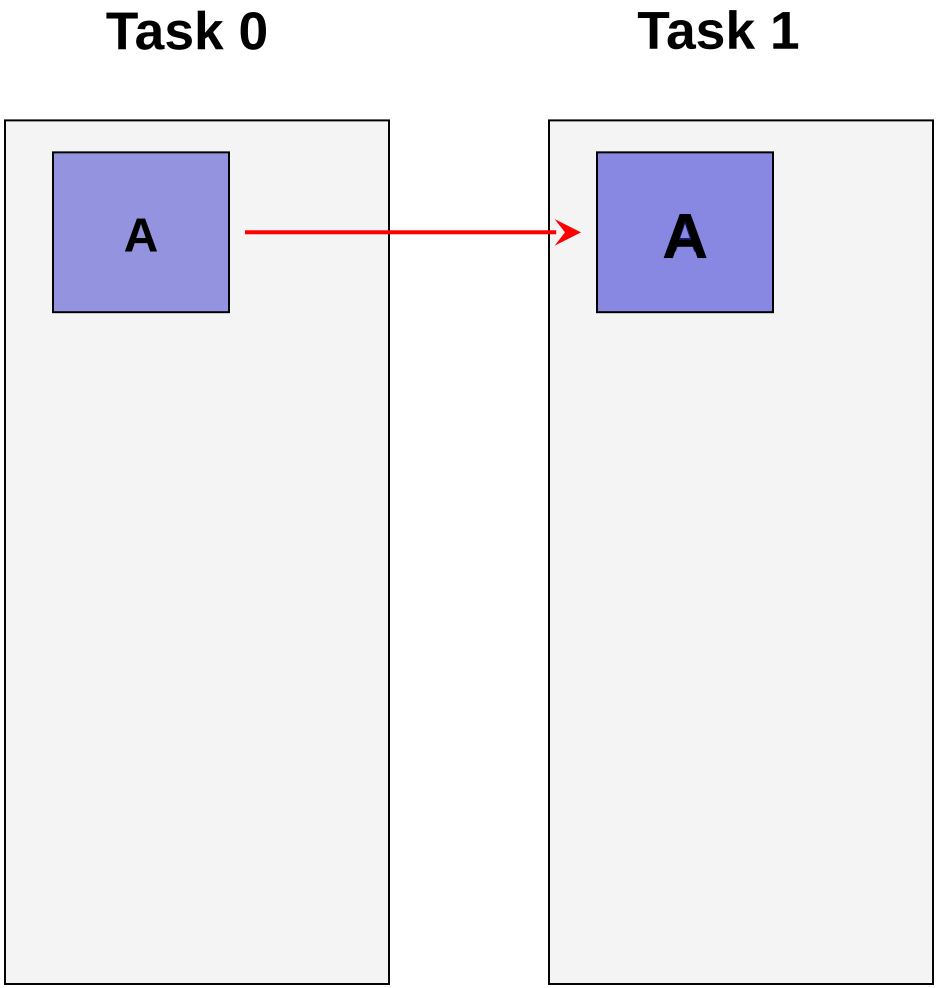


MPI Execution Model

```
PROGRAM main

REAL A(n,n)
INTEGER ierr
…
CALL MPI_Init(ierr)
CALL MPI_Comm_Size(...)
CALL MPI_Comm_Rank(...)
…
IF (rank == 0) THEN
 CALL MPI_Send(A, …)
ELSE
 CALL MPI_Recv(A, …)
ENDIF
…
CALL MPI_Finalize (...)

END PROGRAM main
```



MPI Basis Datatypes for C

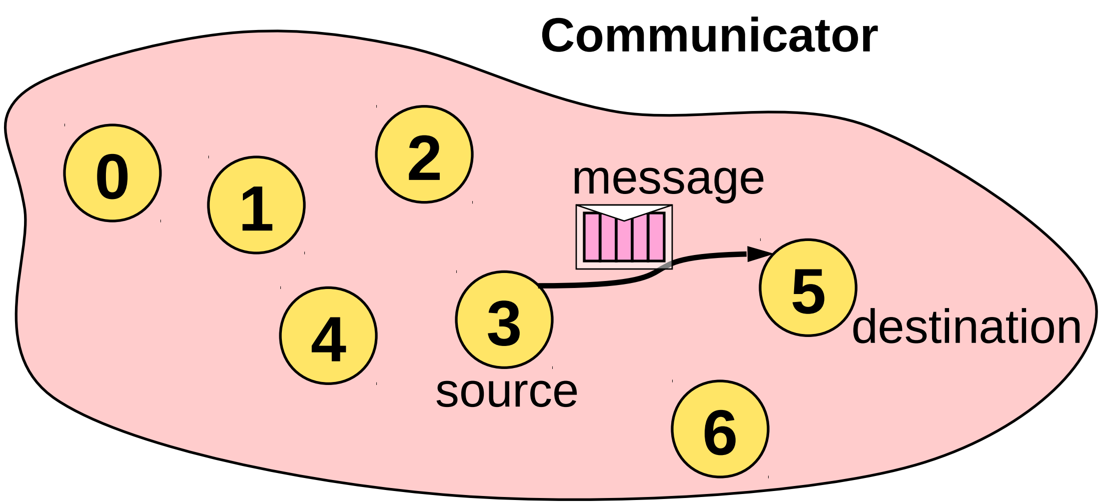
MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long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

MPI Basis Datatypes for Fortran

MPI Datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_INTEGER1	INTEGER (1 Byte)
MPI_INTEGER2	INTEGER (2 Byte)
MPI_INTEGER4	INTEGER (4 Byte)
MPI_REAL	REAL
MPI_REAL2	REAL (2 Byte)
MPI_REAL4	REAL (4 Byte)
MPI_REAL8	REAL (8 Byte)
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE PRECISION COMPLEX
MPI_ LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

P2P Communication

- Communication between 2 prozesses
- Process sends message to another process
- Communication takes place within a communicator, by default MPI_COMM_WORLD
- Processes are identified by their “ranks” within the communicator



Sending Messages

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

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

- Fortran:

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

- `buf` is the first element of a message; `count` values of type `datatype` are transferred
- `dest` is the rank of the destination process within the communicator `comm`
- `tag` is a additional non-negative information of typ integer
- `tag can be` used, to differentiate different messages

Receiving Messages

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

- **Fortran:**
```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
         STATUS, IERROR)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM
  INTEGER STATUS(MPI_STATUS_SIZE), IERROR
```

- **`buf/count/datatype` describe the buffer of a received message**
- **The message is received, that has been sent from the process with `rank` within the communicator `comm`**
- **Envelope information is stored in `status`**
- **Only messages with fitting `tag` are received**

Anforderungen an P2P Kommunikationen

For a successfull communication

- the sending routine must specify a valid rank
- the receiving process must specify a valid rank (wildcards allowed!)
- the communicator must be the same
- The tags must be the same (wildcards allowed!)
- the datatype of the message must be the same
- the receive buffer must be large enough

Usage of Wildcards

- Wildcards can be used when receiving a message
- For receiving from an arbitrary source
  `source = MPI_ANY_SOURCE`
- For receiving a message with an arbitrary tag
  `tag = MPI_ANY_TAG`
- Actual parameters - `source` and `tag` – are stored in the array `status` when calling MPI receive

Communication Types

Karlsruhe Institute of Technology

Communication type	Definition	Remark
Synchronous send `MPI_SSEND`	Exectution only ends, if the corresponding receive routine has been started	
Buffered send `MPI_BSEND`	Execution always ends	A buffer created by the user with `MPI_BUFFER_ATTACH` is necessary
Standard send `MPI_SEND`	Either synchronous or buffered	Uses an internal buffer
Ready send `MPI_RSEND`	Only starts, when corresponding receive routine already has been started	Risky
Receive `MPI_RECV`	Ends, when a message has been stored	The same routine for all communication types

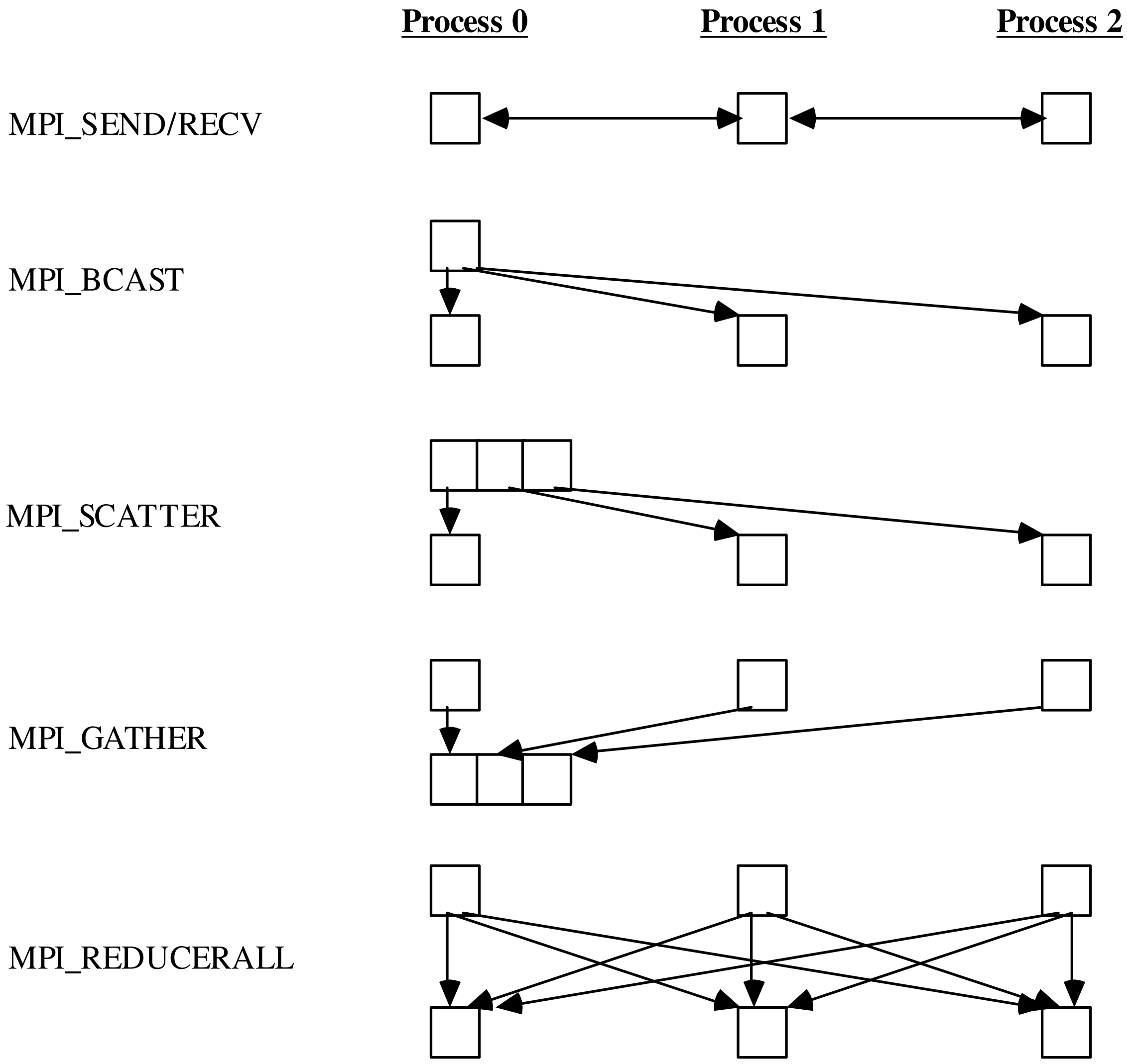
Rules for Communication Types

- Standard send (`MPI_SEND`)
  - Minimal transfer time
  - Can lead to a deadlock in „synchronous mode“
  - → risks for communication type „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`)
  - You shoud‘nt use it without a 200% garanty, that `MPI_Recv` already has been called when calling MPI_Send

(Non-)Blocking Communication

- **Send and Receive can be blocking or non-blocking**
- **A blocking Send can be used with a non-blocking Receive and vice versa**
- **Non-blocking Send can be used in each communication type**
  - **standard – `MPI_ISEND`**
  - **synchronous – `MPI_ISSEND`**
  - **buffered – `MPI_IBSEND`**
  - **ready – `MPI_IRSEND`**

Collective Operations



Collective Operations (2)

Datatypes

```
MPI_Comm comm
MPI_Datatype datatype, intype, outtype
MPI_Op op
MPI_Uop *function()
int count, root, incnt, outcnt, commute
void *buffer, *inbuf, *outbuf
```

Functions

Wait on all processes

```
MPI_Barrier(comm)
```

Broadcast: send `buffer` to all processes

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

Gather data from all processes

```
MPI_Gather(outbuf, outcnt, outtype, inbuf, incnt, intype, root,
comm)
```

MPI Provider

- Computer-companies (Intel, IBM Platform, …)

- MPICH2 – „public domain“ MPI-library of Argonne
  - for all UNIX platforms, for Linux and Windows
  - MVAPICH/MVAPICH2 for MPI via InfiniBand and iWARP

- OpenMPI www.open-mpi.org
  - Merging of FT-, LA- and LAM/MPI