

Programming Techniques for Supercomputers:

Distributed memory parallel processing with MPI

Introduction

MPI in a nutshell

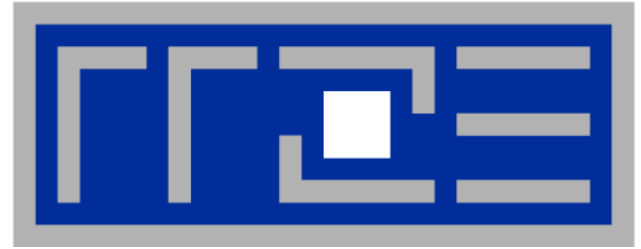
Communication schemes

Prof. Dr. G. Wellein^(a,b) , Dr. G. Hager^(a)

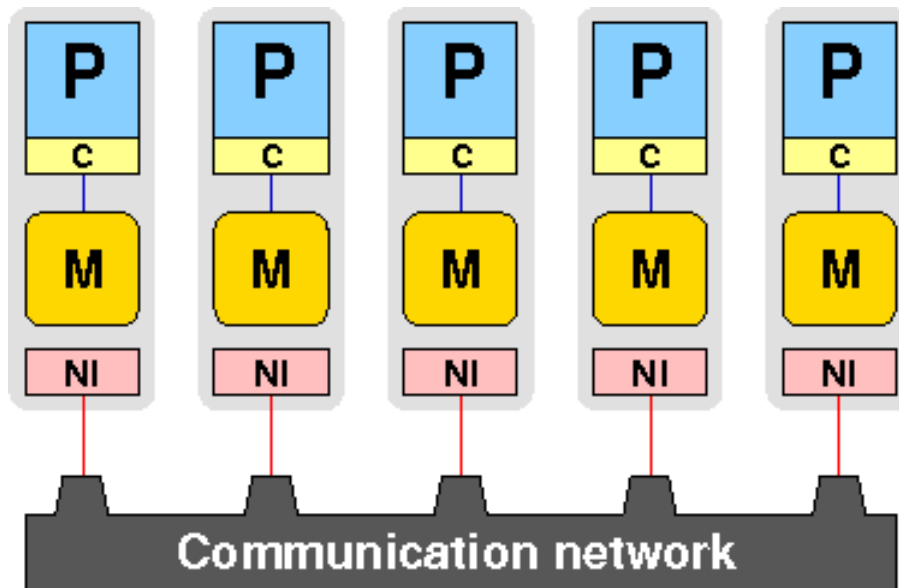
^(a)HPC Services – Regionales Rechenzentrum Erlangen

^(b)Department für Informatik

University Erlangen-Nürnberg, Sommersemester 2018



Introduction to the Message Passing Interface (MPI)



The message passing paradigm

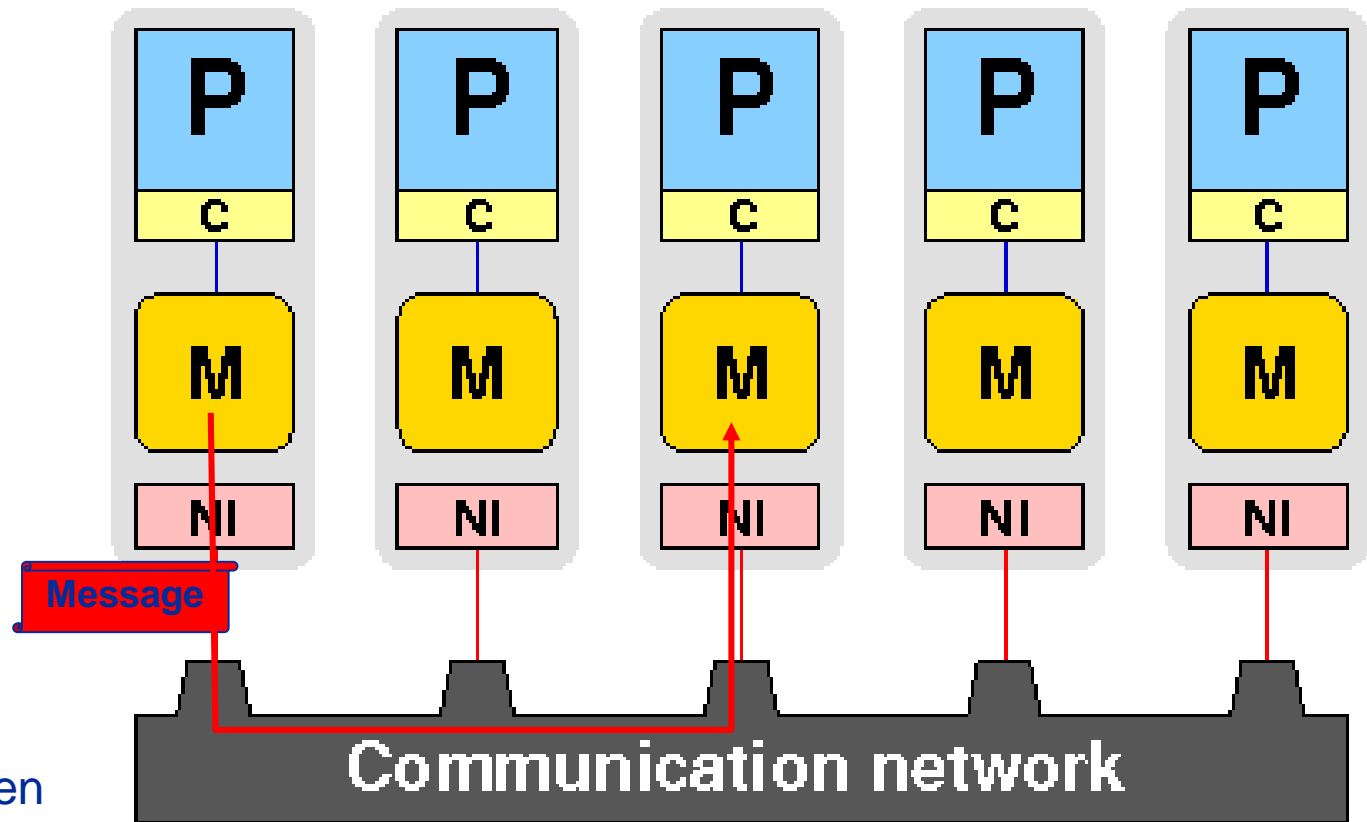


Distributed memory architecture:

Each process(or) can only access its **dedicated address space**.

No global shared address space

Data exchange and communication between processes is done by **explicitly passing messages** through a communication network



Message passing library:

- Should be flexible, efficient and portable
- Hide communication hardware and software layers from application programmer



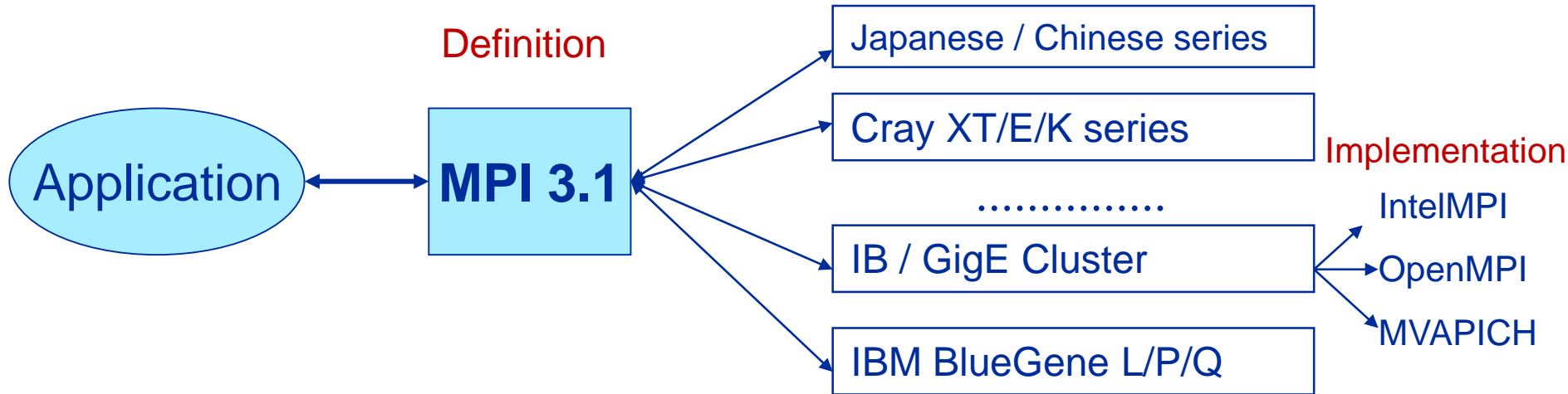
- Widely accepted standard in HPC / numerical simulation: Message Passing Interface (MPI)
- **Process** based approach: All **variables** are **local!**
- Same program on each processor/machine (SPMD)
 - No restriction of the general MP model, because processes can be distinguished by their rank (see later)
- The program is written in a sequential language (FORTRAN/C[++]
- **Data exchange** between processes: Send/receive messages **via MPI library calls**
 - This is usually the most tedious but also the most flexible way of parallelization
- MPI is standard for explicit message passing today:
 - In the early days Parallel Virtual Machine (actually MPMD) did compete



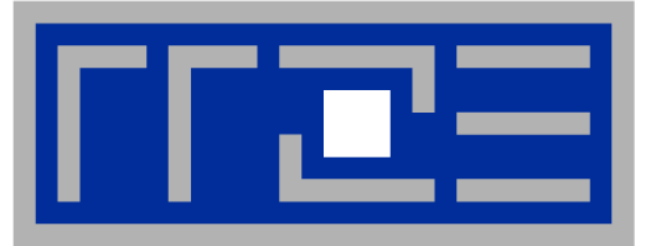
- MPI forum – defines MPI standard / library subroutine interfaces
- Beginning: April 1992 – Before: vendor specific libraries
- Latest standard: MPI 3.1 (2015) – MPI 4.0 under development
- Members (approx. 60) of MPI standard forum
 - Application programmers
 - Research institutes & Computing centres
 - Manufacturers of supercomputers & software designers
- Successful free implementation: MPICH, OpenMPI + many others + vendor libraries (Intel, IBM, CRAY)
- All documents and more pointers available at:
<http://www.mpi-forum.org/>
- MPI defines more than 500 subroutines – **typically 10-30 are used**



- **PORTABILITY:** Architecture and hardware independent code



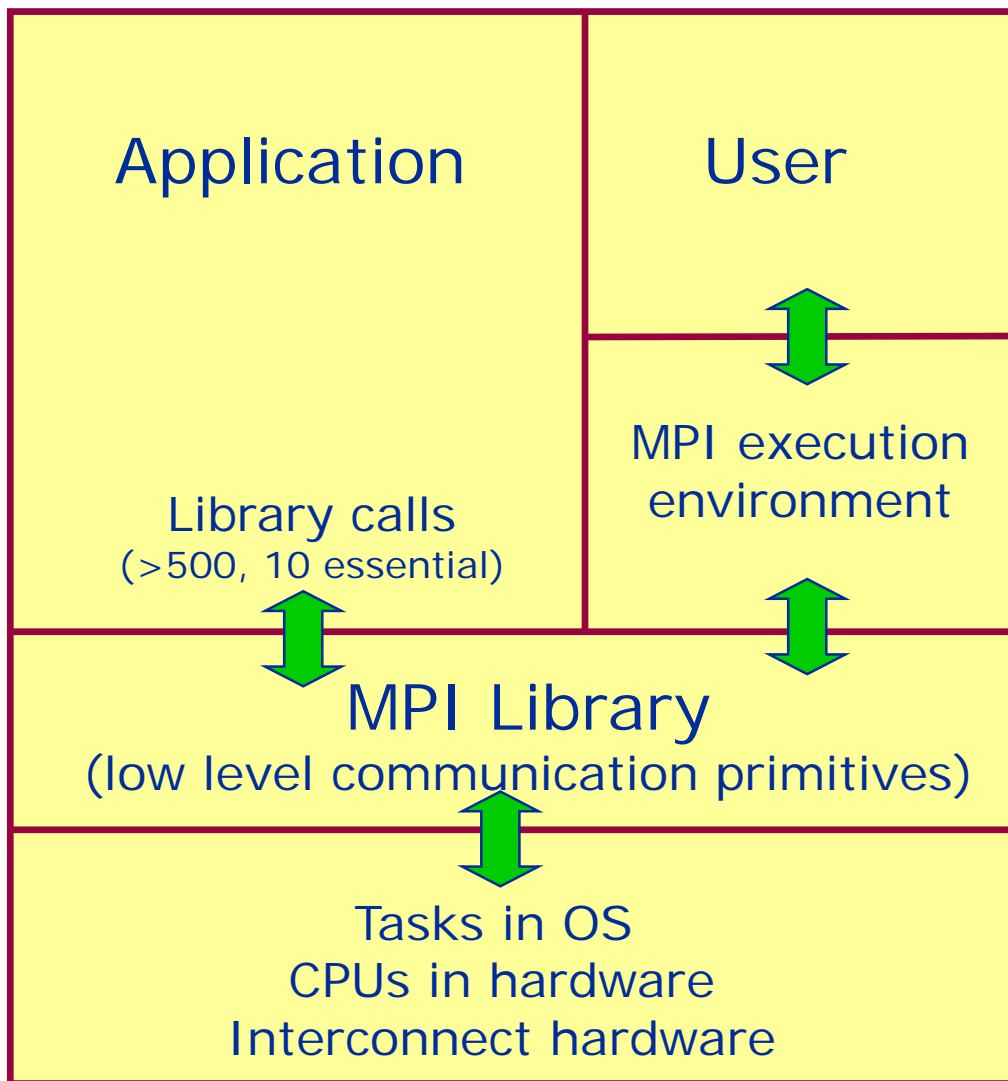
- FORTRAN, C & C++ Interface
- Provides 'well-defined' and 'safe' data transfer
- Enables development of parallel libraries
- Support heterogeneous environment (e.g. clusters with heterogeneous compute nodes)



MPI in a nutshell

MPI in a nutshell

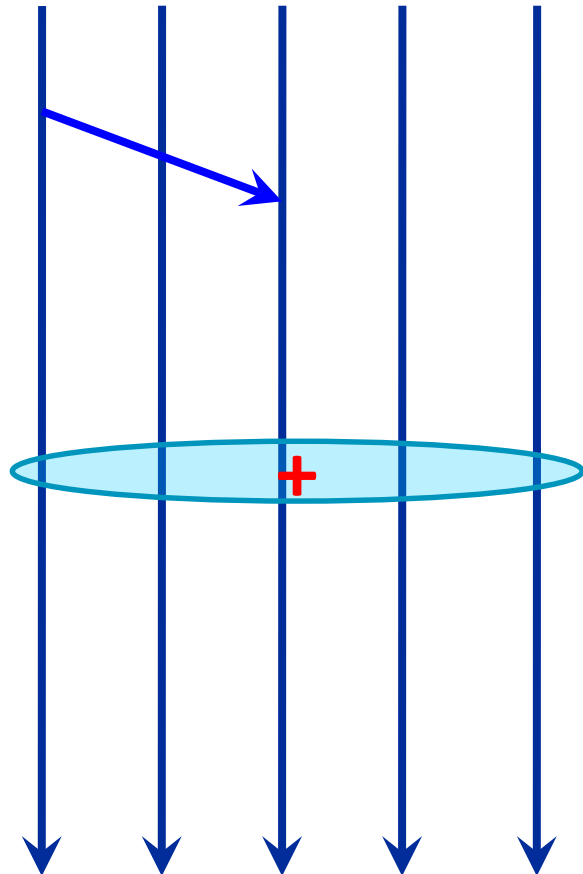
The software architecture



- **Operating system view:**
 - parallel work done by tasks/processes
- **Programmer's view:** Library routines for
 - coordination
 - communication
 - synchronization
- **User's view:** MPI execution environment provides
 - resource allocation
 - startup method
 - and other (implementation-dependent) behavior



- Tasks run throughout program execution: **All variables are local**



- Startup phase: **MPI**
 - launches tasks
 - establishes communication context („communicator“) among all tasks
- MPI Point-to-point data transfer:**
 - usually between **pairs of tasks**
 - usually coordinated
 - may be **blocking** or **non-blocking**
 - explicit **synchronization** is needed for non-blocking
- MPI Collective communication:**
 - between **all tasks** or a subgroup of tasks
 - barrier, reductions, scatter/gather
 - scalability – implementation dependent
- Clean shutdown by **MPI**



- **Required header files:**

- C: `#include <mpi.h>`
- Fortran: `include 'mpif.h'`
- Fortran90: `USE MPI`

- **Bindings:**

- C: `error = MPI_Xxxx(parameter,.....);`
- Fortran: `call MPI_XXXX(argument,...,ierror)`
- MPI constants (global/common): All upper case in C

- **Arrays:**

- C: indexed from **0**
- Fortran: indexed from **1**

- **Here: concentrate on Fortran interface!**

- **Most frequent source of errors in C: call by reference with return values!**



- **C MPI routines**
 - return an `int` — may be ignored
- **Fortran MPI routines**
 - `ierror` argument — cannot be omitted!
- **Return value `MPI_SUCCESS`**
 - indicates that all went ok
- **Default: Abort parallel computation in case of other return values**
 - but can also define error handlers (not covered in this lecture)



- **Each node must start/terminate at least one MPI process**
 - Usually handled automatically
 - More than one process per **core** is often, but not always possible
 - Number of MPI processes per node: Determined by MPI execution environment

- **First call in MPI program: initialization of parallel machine**

```
call MPI_INIT(ierror)
```

- **Last call: clean shutdown of parallel machine**

```
call MPI_FINALIZE(ierror)
```

- **Only process with rank 0 (see later) is guaranteed to return from `MPI_FINALIZE` (cf. standard)**
- **Stdout/stderr of each MPI process**
 - usually redirected to console where program was started
 - many options possible, depending on implementation



- Frequent source of errors: `MPI_Init()` in C

C binding:

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

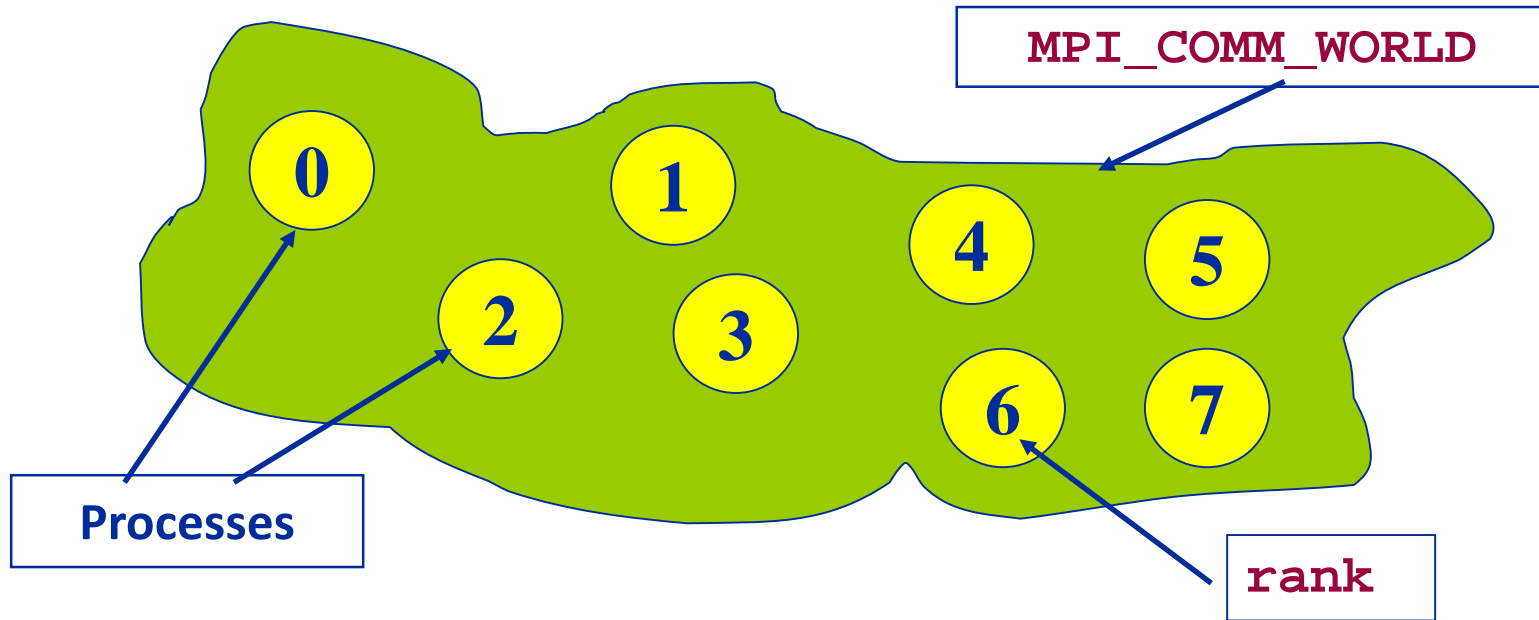
- If `MPI_Init()` is called in a function (bad idea anyway), this function must have pointers to the original data:

```
void init_all(int *argc, char ***argv) {  
    MPI_Init(argc, argv);  
    ...  
}  
...  
init_mpi(&argc, &argv);
```

- Depending on implementation, mistakes at this point might even go unnoticed until code is ported



- **MPI_INIT** defines "communicator" **MPI_COMM_WORLD**:



- **MPI_COMM_WORLD** defines the processes that belong to the parallel machine
- other communicators (subsets) are possible (cf. MPI communicators section)
- **rank** labels processes inside a communicator



- The **rank** identifies each process within a communicator (e.g. `MPI_COMM_WORLD`):

- obtain rank:

```
integer rank, ierror
```

```
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
```

- **rank** = 0,1,2,..., (number of MPI tasks – 1)

- Obtain number of MPI tasks in communicator:

```
integer size, ierror
```

```
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
```



- **MPI_COMM_WORLD** is
 - effectively an **MPI-global** variable and required as argument for nearly all MPI calls
- **rank**
 - is target label for MPI messages
 - can drive user-defined directives what each process should do:

```
if (rank == 0) then
    ... ! *** do work for rank 0 ***
else
    ... ! *** do work for other ranks ***
end if
```




```
program hello
  use mpi
  implicit none

  integer rank, size, ierror

  call MPI_INIT(ierror)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

  write(*,*) 'Hello World! I am ',rank,' of ',size

  call MPI_FINALIZE(ierror)
end program
```



■ Compile time:

- include files or module information file needed

■ Link time:

- MPI library required

■ Most implementations

- provide `mpif77`, `mpif90`, `mpicc` and `mpiCC` wrappers
- not standardized, so variations must be expected e.g., with Intel-MPI (`mpiifort`, `mpiicc` etc.)

• Startup facilities

- `mpirun` (legacy)
- `mpiexec`

■ Compile:

```
mpif90 -o hello hello.f90
```

■ Run on 4 processors:

```
mpirun -np 4 ./hello
```

■ Output:

```
Hello World! I am 3 of 4
Hello World! I am 1 of 4
Hello World! I am 0 of 4
Hello World! I am 2 of 4
```

order undefined

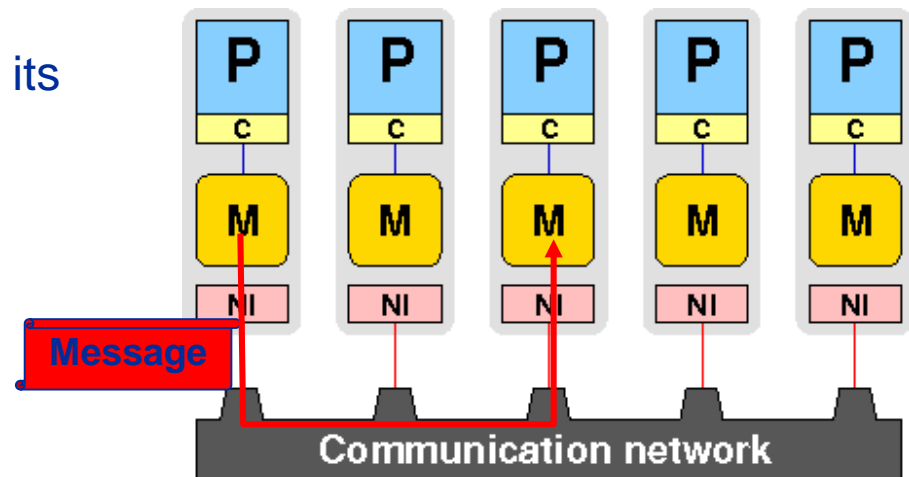


- **For successful and reliable data transfer MPI requires information:**

- Which processor is sending the message.
- Where is the data on the sending processor.
- What kind of data is being sent.
- How much data is there.
- Which processor(s) are receiving the message.
- Where should the data be left on the receiving processor.
- How much data is the receiving processor prepared to accept.

- Sender and receiver must pass its information to MPI separately

- Holds for point-to-point communication

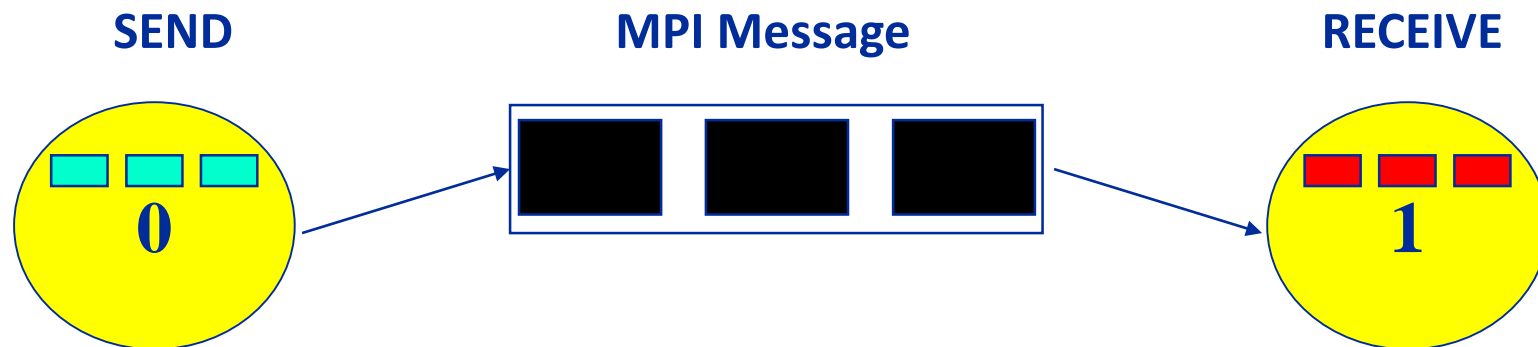




- Communication between two processes:
Sending / Receiving of MPI-Messages

- MPI-Message:

Array of elements of a particular MPI datatype



- MPI data types:
 - basic data types
 - derived data types



MPI datatype	FORTRAN datatype
<code>MPI_CHARACTER</code>	<code>CHARACTER(1)</code>
<code>MPI_INTEGER</code>	<code>INTEGER</code>
<code>MPI_REAL</code>	<code>REAL</code>
<code>MPI_DOUBLE_PRECISION</code>	<code>DOUBLE PRECISION</code>
<code>MPI_COMPLEX</code>	<code>COMPLEX</code>
<code>MPI_LOGICAL</code>	<code>LOGICAL</code>
<code>MPI_BYTE</code>	
<code>MPI_PACKED</code>	

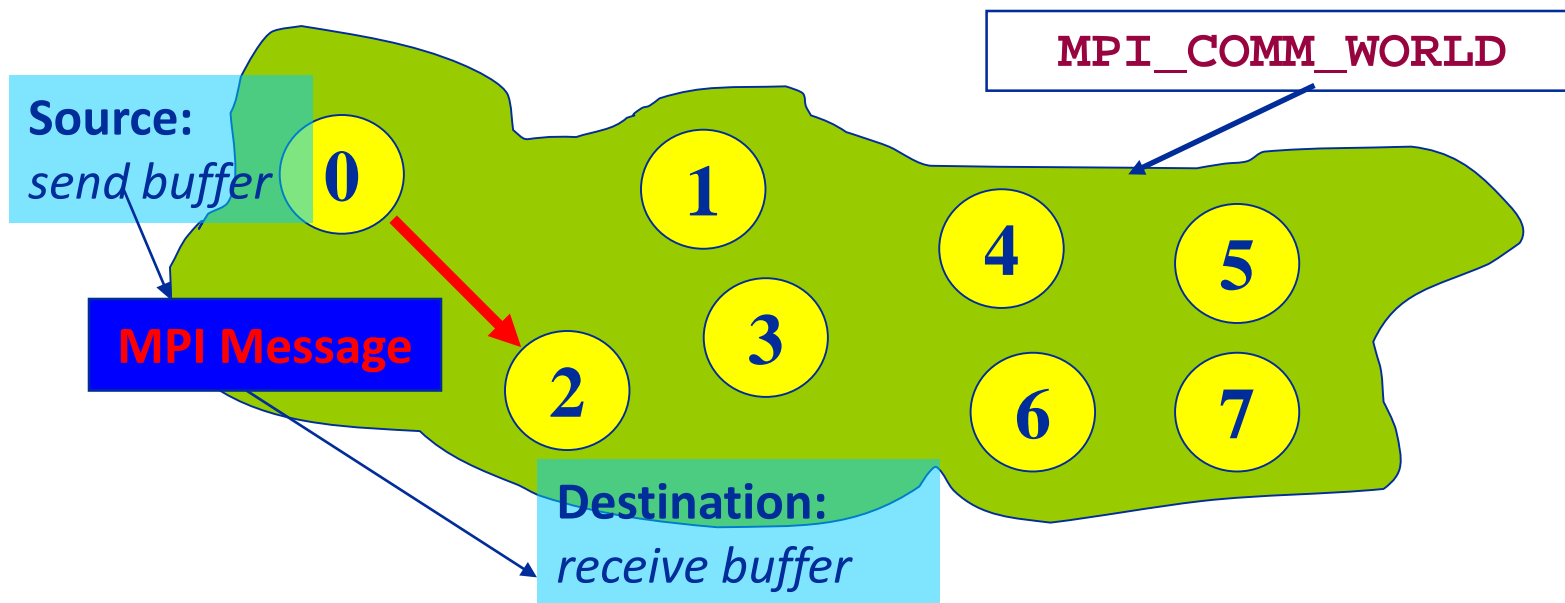
MPI datatype	C datatype
<code>MPI_CHAR</code> / <code>MPI_SHORT</code>	<code>signed char</code> / <code>short</code>
<code>MPI_INT</code> / <code>MPI_LONG</code>	<code>signed int</code> / <code>long</code>
<code>MPI_UNSIGNED_CHAR</code> / ...	<code>unsigned char</code> / ...
<code>MPI_FLOAT</code> / <code>MPI_DOUBLE</code>	<code>float</code> / <code>double</code>
<code>MPI_LONG_DOUBLE</code>	<code>long double</code>
<code>MPI_BYTE</code>	
<code>MPI_PACKED</code>	



- **MPI_BYTE**: Eight binary digits
 - hack value, do not use
 - **MPI_PACKED**: can implement new data types → however, it is more flexible to use ...
 - ... derived data types: Built at run time from basic data types (see later)
- **Data type matching is strictly required: Same MPI data type in SEND and RECEIVE call**
- type must match on both ends in order for the communication to take place
-
- Support for heterogeneous systems/clusters
 - implementation-dependent
 - automatic data type conversion between systems of differing architecture may be needed



- Communication between **exactly** two processes within the same communicator
- Identification of source and destination via the rank within the communicator!



- **Blocking communication:** After MPI call returns
 - Source process can safely modify send buffer
 - Receive buffer (destination process) contains message from source

MPI in a nutshell

Blocking Standard Send: MPI_Send



- Fortran syntax:

```
<type> buf(*)
integer :: count, datatype, dest, tag, comm, ierror
call MPI_Send(buf,          ! message buffer
               count,      ! # of items
               datatype,   ! MPI data type
               dest,       ! destination rank
               tag,        ! message tag (additional label)
               comm,       ! communicator
               ierror)    ! return value
```

- Completion of **MPI_SEND**:
 - Send buffer** may be reused after **MPI_SEND** returns
 - Status of **destination** process is not defined – message may or may not have been received after return!



- Fortran syntax:

```
<type> buf(*)
integer :: count, datatype, source, tag, comm,
integer :: status(MPI_STATUS_SIZE), ierror
call MPI_Recv(buf,           ! message buffer
              count,        ! maximum # of items
              datatype,     ! MPI data type
              source,       ! source rank
              tag,          ! message tag (additional label)
              comm,         ! communicator
              status,       ! status object (MPI_Status* in C)
              ierror)       ! return value
```

- Completion of **MPI_RECV**:
 - Message has been received successfully and completely
 - Size of **buf** needs to be at least of size of message to be receive
May be larger → **MPI_Get_count**



- Determine number of data items received via `MPI_Get_count`

```
integer :: status(MPI_STATUS_SIZE), datatype, count, ierror
call MPI_Get_count(status,      ! status object from MPI_Recv()
                  datatype,    ! MPI data type received
                  count,       ! count (output argument)
                  ierror)      ! return value
```

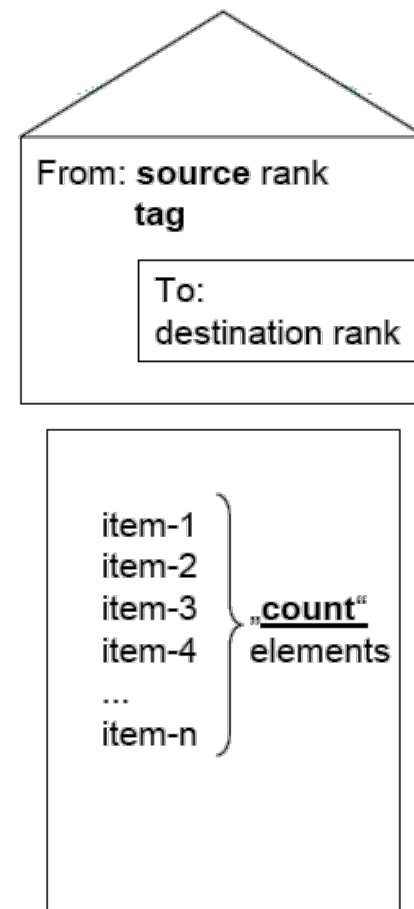
- `MPI_Recv` accepts wildcards for `source=MPI_ANY_SOURCE` and `tag=MPI_ANY_TAG`
- The actual parameters for the wildcard parameters can be determined via `status` object:

```
call MPI_RECV(field, count, MPI_REAL,
&             MPI_ANY_SOURCE, MPI_ANY_TAG,
&             MPI_COMM_WORLD, status, ierror)
write(*,*) 'Received from #', status(MPI_SOURCE),
&         ' with tag ',      status(MPI_TAG)
```



For a communication to succeed:

- sender must specify a valid destination.
- receiver must specify a valid source rank (or `MPI_ANY_SOURCE`).
- communicator must be the same (e.g., `MPI_COMM_WORLD`).
- tags must match (resp. `MPI_ANY_TAG` for receiver).
- message datatypes must match.
- receiver's buffer must be large enough.





- Beginner's MPI procedure toolbox:
 - `MPI_INIT` let's get going
 - `MPI_COMM_SIZE` how many are we?
 - `MPI_COMM_RANK` who am I?
 - `MPI_SEND` send data to someone else
 - `MPI_RECV` receive data from some-/anyone
 - `MPI_GET_COUNT` how many items have I received?
 - `MPI_FINALIZE` finish off
- Standard send/receive calls provide most simple way of point-to-point communication
- Send/receive buffer may safely be reused after the call has completed
- `MPI_SEND` must have a specific target/tag, `MPI_RECV` does not
- So far no explicit synchronization!!

MPI in a nutshell

Parallel integration in MPI

- Task: calculate $\int_a^b f(x)dx$ using (existing) function `integrate(x,y)`:
- Split up interval `[a,b]` into equal disjoint chunks
- Compute partial results in parallel
- Collect partial result on "root" process (`rank=0`) and sum up (receive `size-1` messages!)
- Every other process (`rank > 0`) needs to send its partial sum

```

integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya,myb)

! rank 0 collects partial results
if(rank.eq.0) then
  res=psum
  do i=1,size-1
    call MPI_Recv(tmp, & ! receive buffer
                  1, & ! array length
                  & ! data type
                  MPI_DOUBLE_PRECISION, &
                  i, & ! rank of source
                  0, & ! tag (unused here)
                  MPI_COMM_WORLD, & ! communicator
                  status, & ! status array (msg info)
                  ierror)

    res=res+tmp
  enddo
  write(*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
else
  call MPI_Send(psum, & ! send buffer
                1, & ! message length
                MPI_DOUBLE_PRECISION, &
                0, & ! rank of destination
                0, & ! tag (unused here)
                MPI_COMM_WORLD, ierror)
endif
endif

```



■ Remarks:

- **Gathering results from processes** is a very common task in MPI – there are more efficient and elegant ways to do this (see later).
- This is a **reduction operation** (summation). There are more efficient and elegant ways to do this (see later).
- The **'master' process waits** for one receive operation to be completed before the next one is initiated. There are more efficient ways... You guessed it!
- **'Master-worker' schemes** are quite common in MPI programming but scalability to high process counts may be limited
- **Error checking is rarely done** in MPI programs – debuggers are often more efficient if something goes wrong
- **Every process has its own `res` variable**, but only the master process actually uses it → it's typical for MPI codes to use more memory than actually needed



- Time measurement function (elapsed wallclock time)
`DOUBLE PRECISION FUNCTION MPI_WTIME()`
returns current timer value; determine resolution of timer with
`DOUBLE PRECISION FUNCTION MPI_WTICK()`
- `MPI_WTIME` has no defined starting point (always calculate time differences!)
- No `ierror` argument in Fortran!
- Example:

```
double precision start_time, total_time
```

```
start_time = MPI_WTIME( )
```

```
C ... do some work here
```

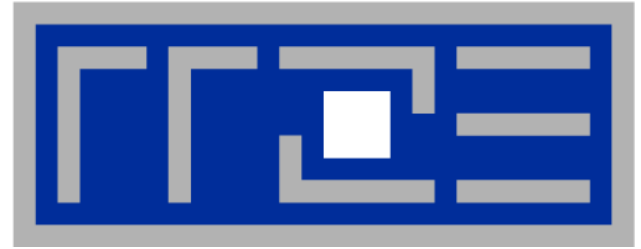
```
total_time = MPI_WTIME( ) - start_time
```



- MPI_ABORT forces an MPI program to terminate:

```
MPI_ABORT(comm, errorcode, ierror)  
integer errorcode
```

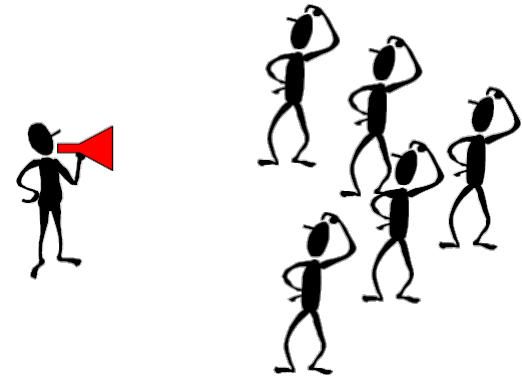
- Aborts all processes in communicator
- `errorcode` will be handed as exit value to calling environment
- Safe and well-defined way of terminating an MPI program (if implemented correctly)
- In general, if something unexpected happens, try to shut down your MPI program the standard way (`MPI_FINALIZE`)



MPI communication schemes

**Point-to-point communication:
blocking vs. non-blocking**

Collective Communication





Blocking Point-to-Point Communication in MPI

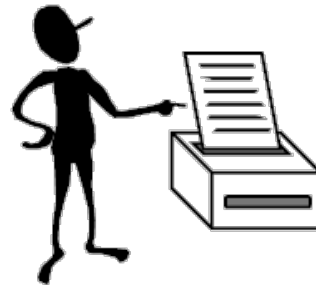


- **Point-to-Point communication:**
 - Simplest form of message passing
 - Two processes within a communicator exchange a message
 - Two classes of point-to-point communication:
 - Blocking (this section)
 - Non-blocking (next section)
 - Communication calls in both classes are available in two (three) flavors
 - Synchronous
 - Buffered
 - (Ready – never use it)
- **Reminder: “Blocking”** communication \leftrightarrow After MPI call returns
 - Source process can safely modify *send buffer*
 - *receive buffer* on destination process contains message from source
 - Potential implementation for send operation: synchronous or buffered (cf. next 2 slides)

Blocking Point-To-Point Communication: Synchronous Send



- The sender gets an information that the message is received.



- Analogue to the *beep* or *okay-sheet* of a fax.

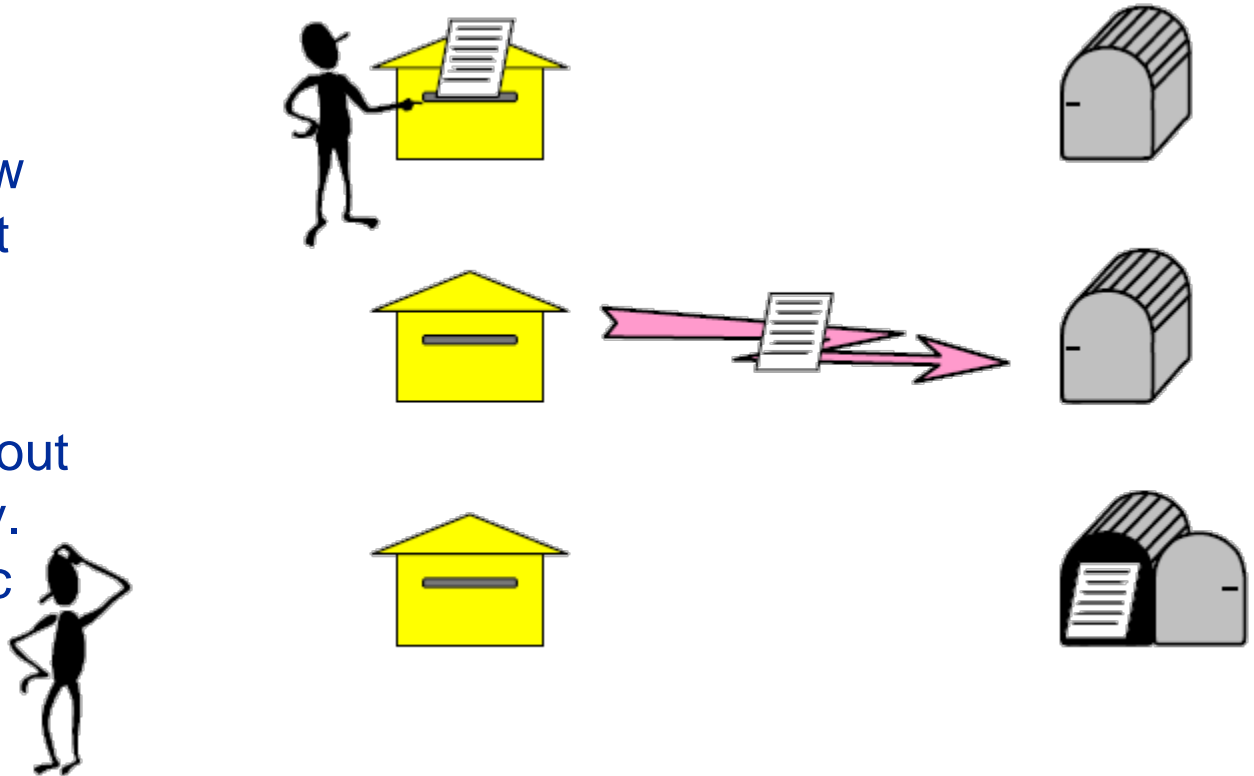


- Syncs the two processes





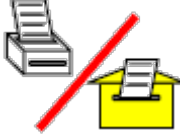



- One only knows when the message has left.
- We can write a new message and put it into the post box
- We do not care about the time of delivery.
→ No need to sync with other process





- Blocking communication:
Completion of send/receive ↔ buffer can safely be reused!

Communication mode	Completion condition	MPI Routine (Blocking)
 Synchronous Send	Only completes when the receive has started.	<code>MPI_SSEND</code>
 Buffered Send	Always completes, irrespective of the receive process.	<code>MPI_BSEND</code>
 Standard Send	Either synchronous or buffered.	<code>MPI_SEND</code>
 Ready Send	Always completes, irrespective whether the receive has completed.	<code>MPI_RSEND</code> DO NOT USE
Receive	Completes when a message has arrived.	<code>MPI_RECV</code>



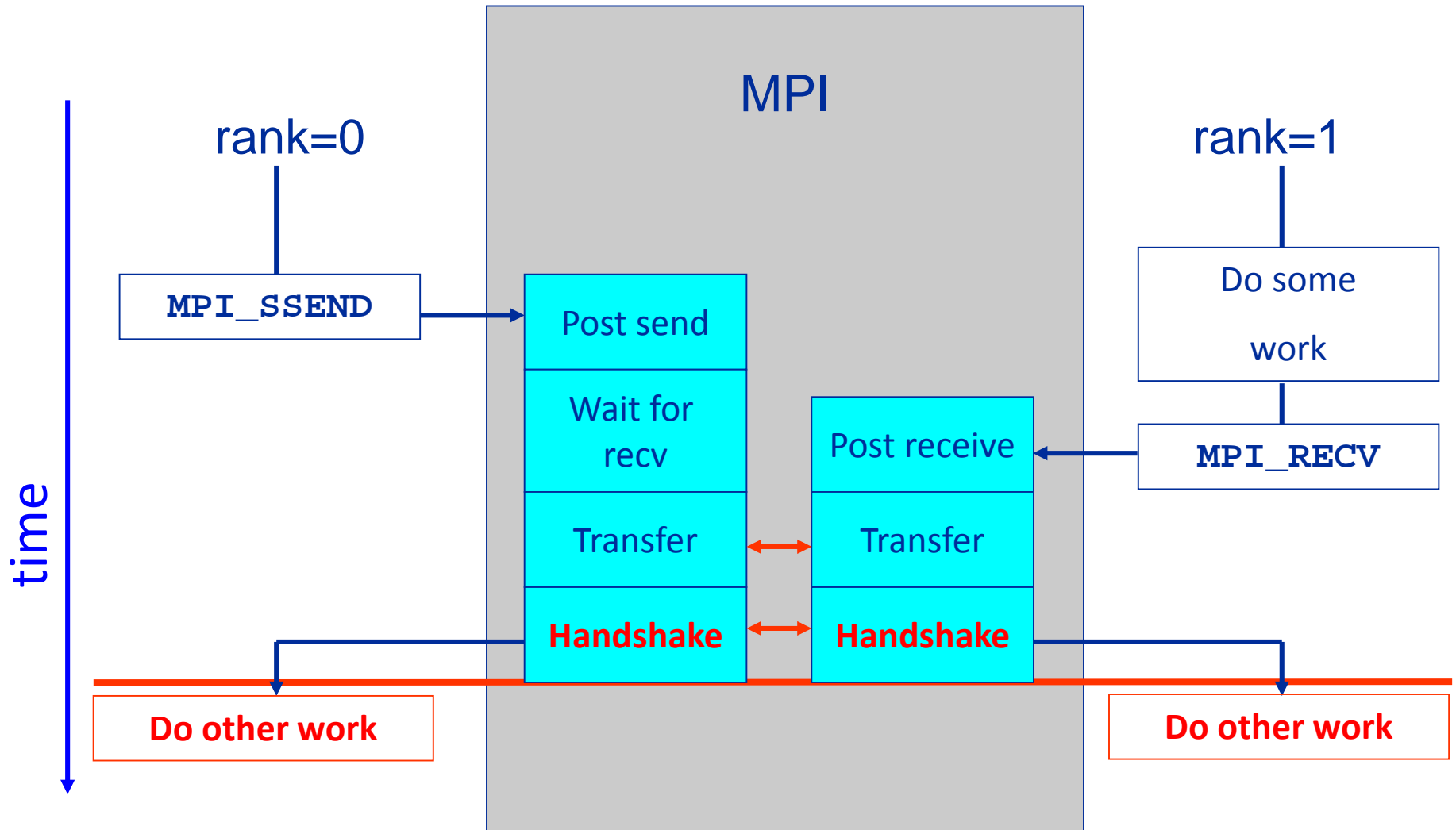
- **MPI_SSEND** completes **after** message has been accepted by the destination (“handshaking”).
- **Synchronization of source and destination!**
- **Predictable and safe behavior!**
- **MPI_SSEND** should be used for debugging purposes!

- **Problems:**
 - Performance (high latency, risk of serialization – best bandwidth)
 - Deadlock situations (see later)

- **Syntax (FORTRAN): like MPI_SEND**
MPI_SSEND(buf, count, datatype, dest, tag, comm, ierror)



- Diagrammatic view of synchronous communication:





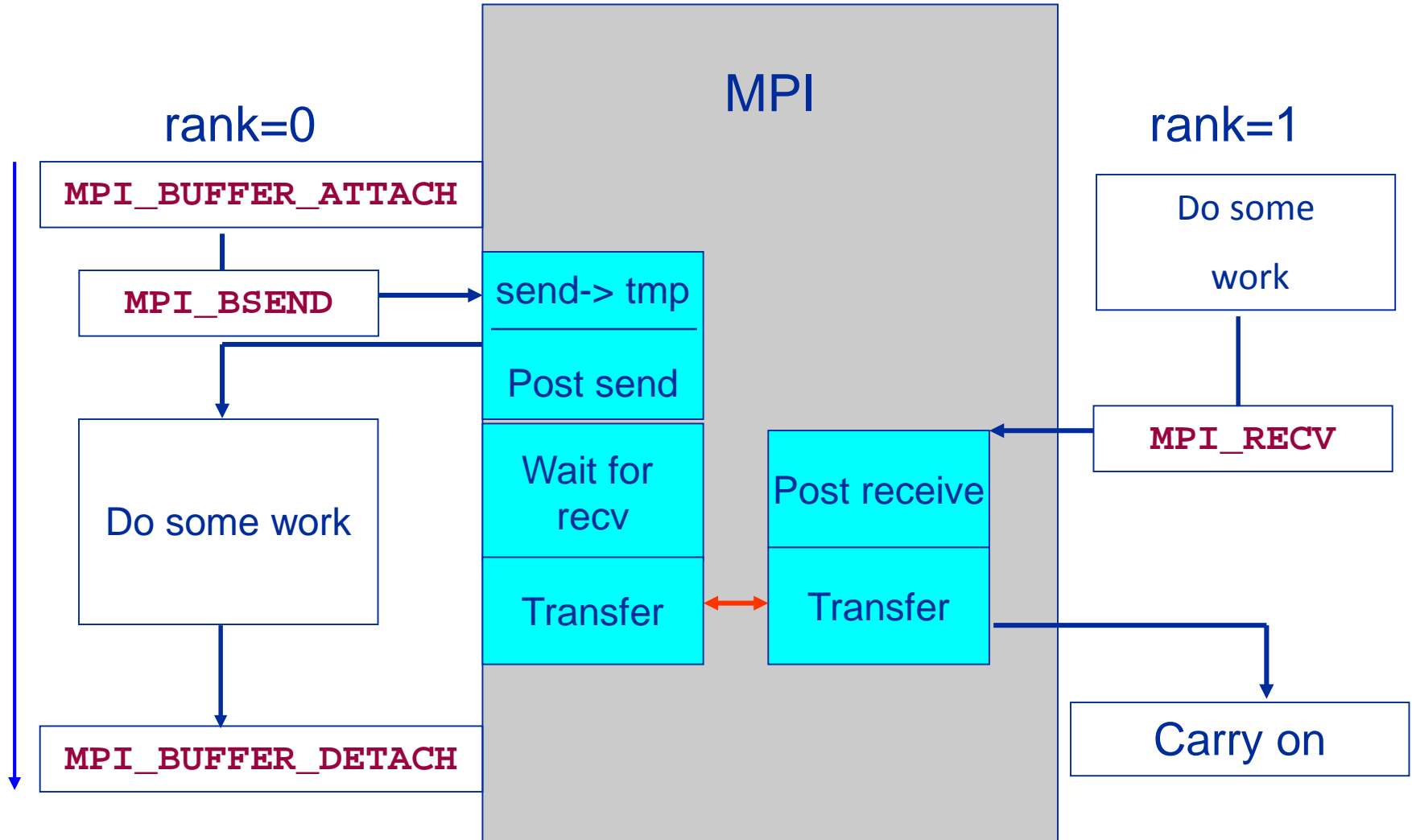
- **MPI_BSEND:**
 - 1) Copy message to a (system) buffer
 - 2) Complete
- **Completes immediately!**
- **Predictable behavior & no synchronization!**
- **Programmer has to attach *extra buffer* space (MPI_BUFFER_ATTACH / MPI_BUFFER_DETACH)**
- **Only one buffer can be attached per process at a time**
- **Additional copy operations!**
- **Syntax (FORTRAN): (cf. also syntax of MPI_SEND)**

```
MPI_BSEND(buf, count, datatype, dest, tag, comm,  
ierror)
```

buf: this is the 'normal' send buffer – *extra buffer* space elsewhere



- Diagrammatic view of blocking buffered communication:





- Syntax (FORTRAN):

```
MPI_BUFFER_ATTACH(tmpbuf, size, ierror)
```

tmpbuf: address of buffer space

size: buffer size in **bytes**

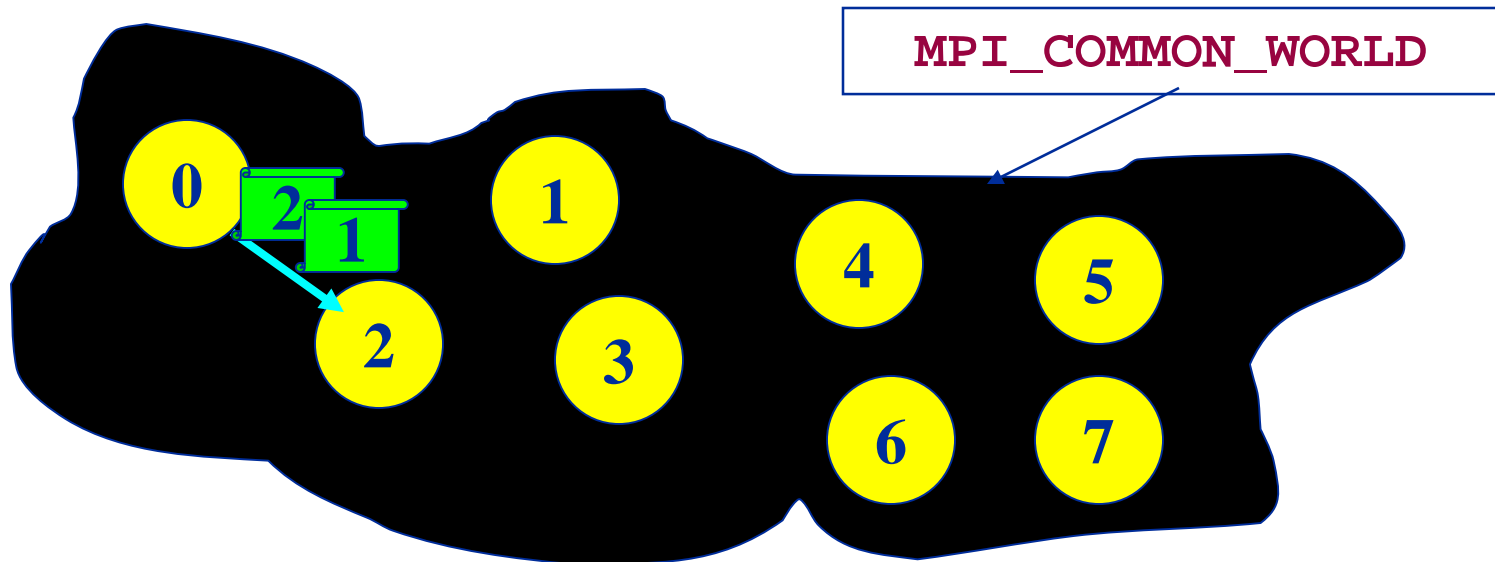
- Determine size of **tmpbuf**: (messagesize of all BSENDS) + (number of intended BSENDS * **MPI_BSEND_OVERHEAD**)
- Best way to get required size for **one** message:
call **MPI_PACK_SIZE(count, datatype, comm, s, ierror)**
size = s + MPI_BSEND_OVERHEAD
- Free buffer space after use:
MPI_BUFFER_DETACH(tmpbuf, size, ierror)

buf, size are output parameters here, but only **size** is set in FORTRAN



- **Semantics:** Rules, guaranteed by MPI implementations

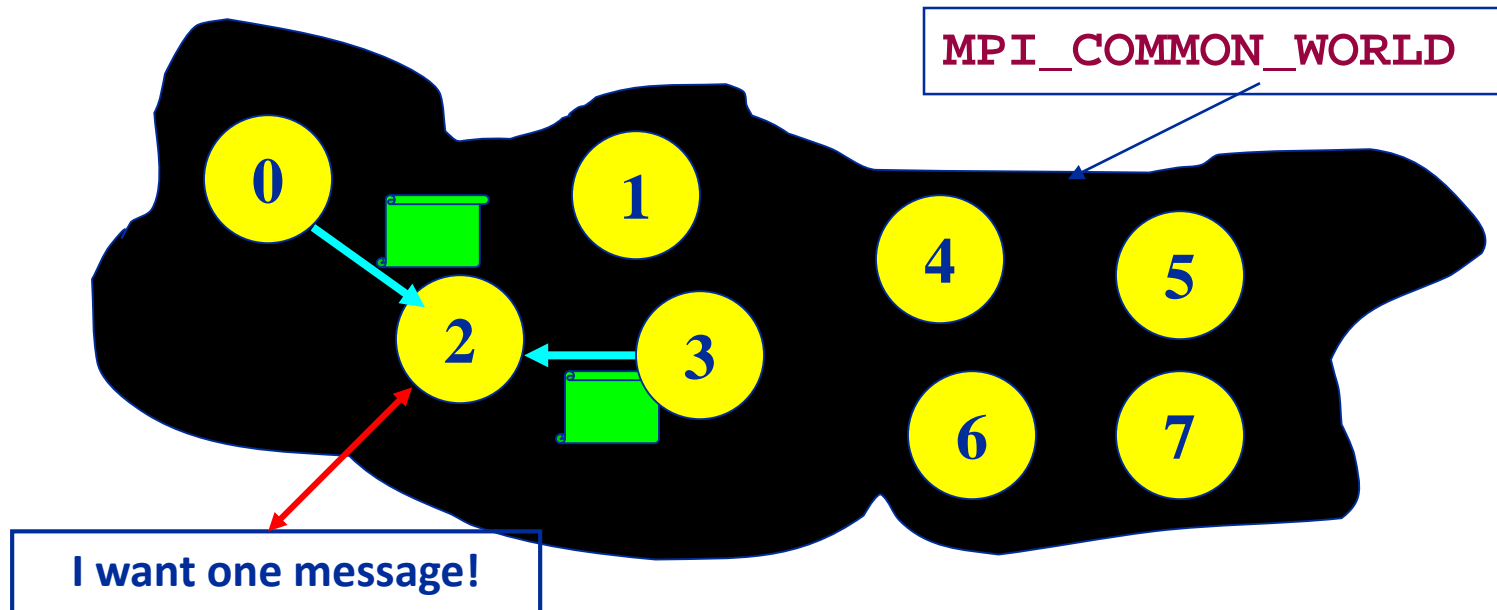
1) Message Order Preservation (within same communicator)



Point-to-Point Communication: Semantics: Guarantees progress



- Progress: It is not possible for a **matching** send and receive pair to remain permanently outstanding.
 - Matching** means: **data types**, **tags** and **receivers** match





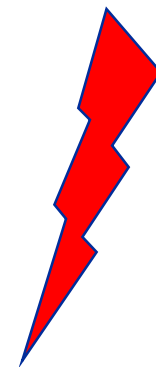
- Example with 2 processes, each sending a message to the other:

```
integer buf(200000)
if(rank.EQ.0) then
    dest=1
    source=1
else
    dest=0
    source=0
end if
```

This program will not work correctly on all systems!

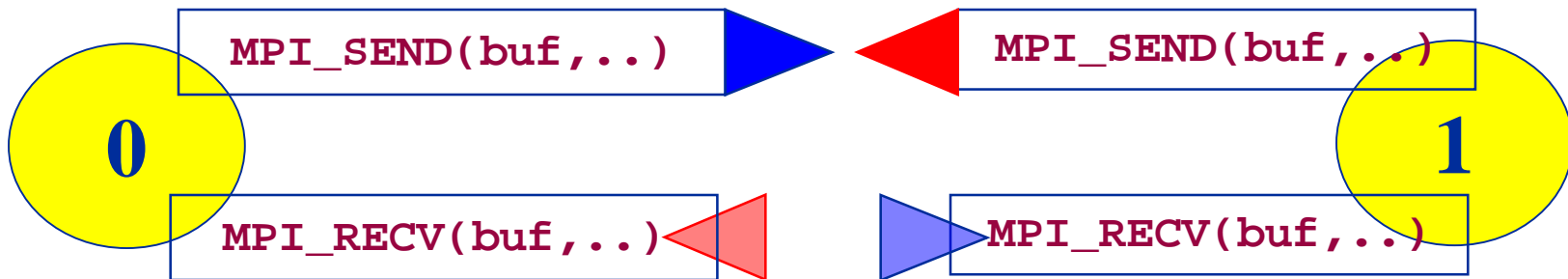
```
MPI_SEND(buf, 200000, MPI_INTEGER, dest, 0,
& MPI_COMM_WORLD, ierror)

MPI_RECV(buf, 200000, MPI_INTEGER, source, 0,
& MPI_COMM_WORLD, status, ierror)
```

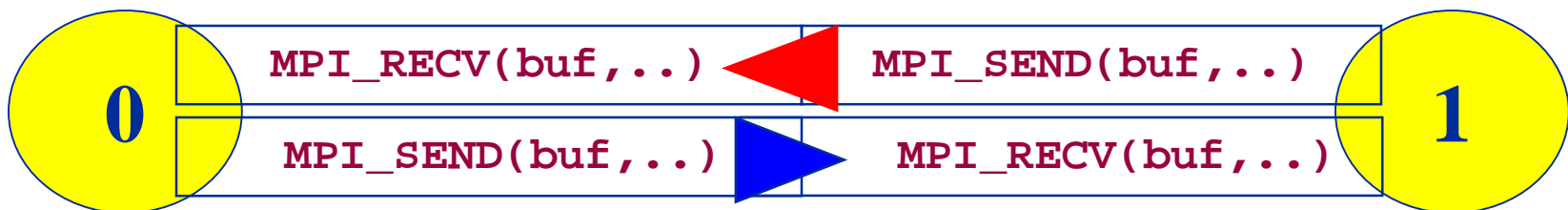




- **Deadlock:** Some of the outstanding blocking communication can never be completed (program stalls)
- Example: `MPI_SEND` is implemented as **synchronous send** for large messages!



- One remedy: reorder send/receive pair on one process (e.g. rank 0):



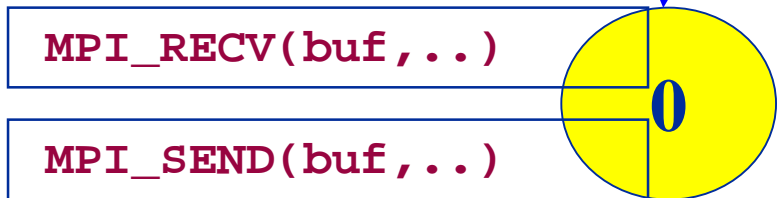


- **Other possibilities to avoid deadlocks:**
 - Use buffered sends (see above)
 - **MPI_SENDRECV** (see later)
 - Use non-blocking communication (see later)
- **Replacing MPI_SEND by MPI_SSEND in an MPI program**
 - Can often reveal deadlocks which would otherwise go unnoticed until the code is ported elsewhere
 - **Ideally, every MPI program should work with MPI_SSEND alone**
 - Probably with low efficiency, but this is for debugging only



Motivation:

- Start Send/Receive call at the same time
- Shift messages, Ring topologies, “Ring shift”
- Avoid deadlocks
- Example:



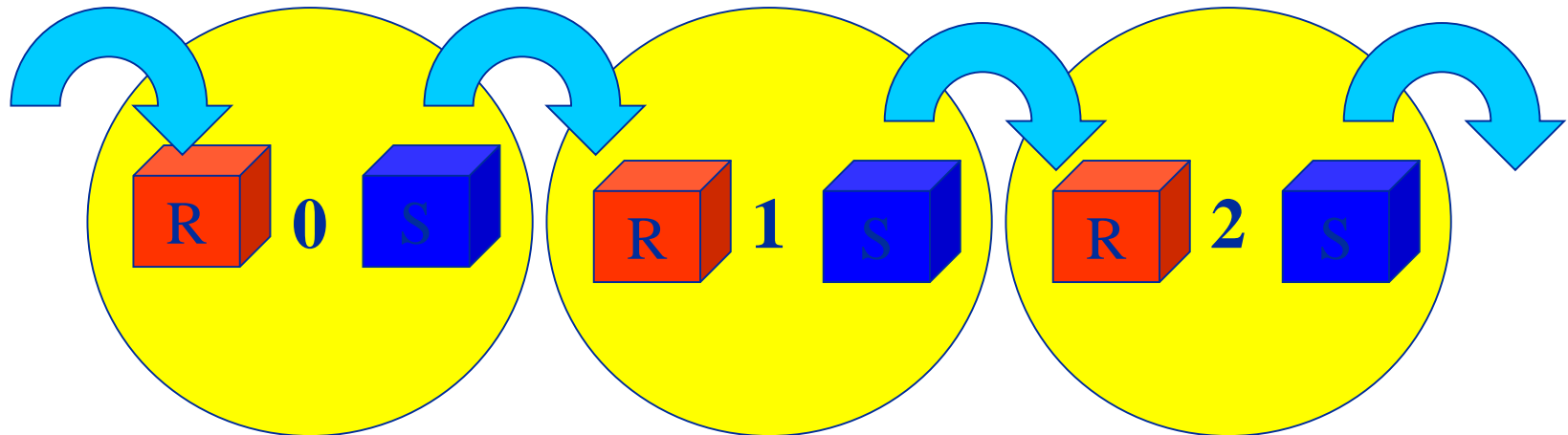
```
integer :: size, rank, left, right, ierror
integer, dimension(N) :: buf
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
left = rank+1           ! left and right neighbors
right = rank-1
if(right<0)      right=size-1    ! close the ring
if(left>=size) left=0
call MPI_Send(buf, N, MPI_INTEGER, left, 0, &
              MPI_COMM_WORLD, ierror)
call MPI_Recv(buf, N, MPI_INTEGER, right, 0, &
              MPI_COMM_WORLD, status, ierror)
```

Deadlock, if **MPI_Send** uses synchronous implementation



- **MPI_SENDRECV**: separate buffer for send and receive operation -
Syntax: simple combination of send and receive arguments

```
MPI_SENDRECV( sendbuf, sendcount, sendtype, dest, sendtag,  
recvbuf, recvcount, recvtype, source, recvtag,  
comm, status, ierror)
```

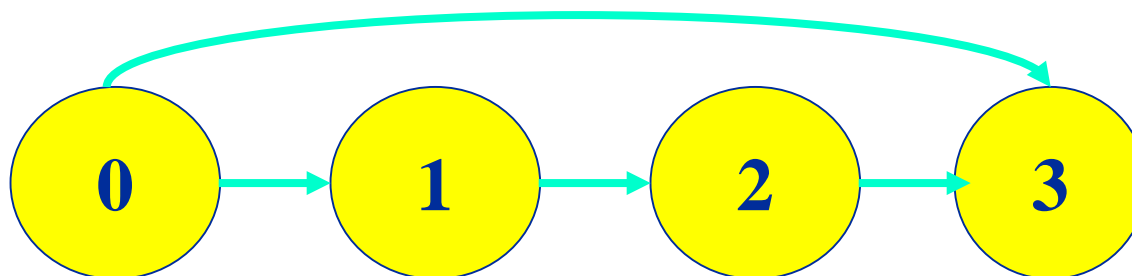




- Example (Guarantees progress – no deadlock)

```
! Determine rank of processor to the left
lrank = rank - 1
if(rank .eq. 0) lrank=size-1
! Determine rank of processor to the right
rrank = rank + 1
if(rank .eq. (size-1)) rrank=0
```

```
call MPI_SENDRECV(sendbuf, 5, MPI_INTEGER, rrank, 1,
  recvbuf, 5, MPI_INTEGER, lrank, 1,
  MPI_COMM_WORLD, status, ierror )
```

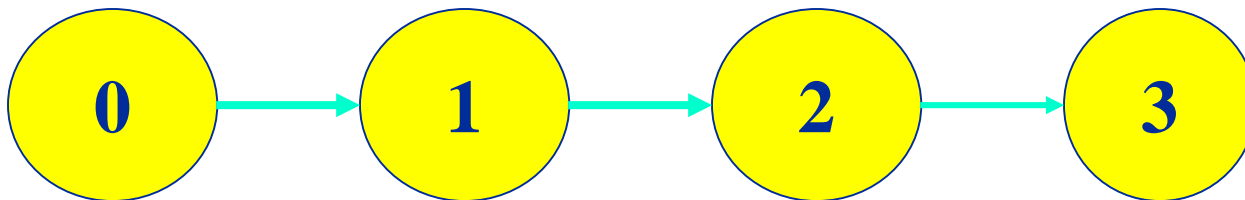




■ Notes:

- Completely compatible with point-to-point messages
 - **MPI_SENDRECV** matches a simple **xRECV** or **xSEND**
- Blocking call
- **MPI_PROC_NULL** can build an open chain:
source or destination can be specified as **MPI_PROC_NULL**
(c.f. example:

rank=0: specify **lrank= MPI_PROC_NULL**
rank=3: specify **rrank= MPI_PROC_NULL**)





- **Blocking MPI communication calls:**
 - MPI call has been completed → **send/receive buffer can safely be reused!**
 - No assumption about the matching receive call (destination) except for synchronous send!
 - May require synchronization of the processes (Performance!)

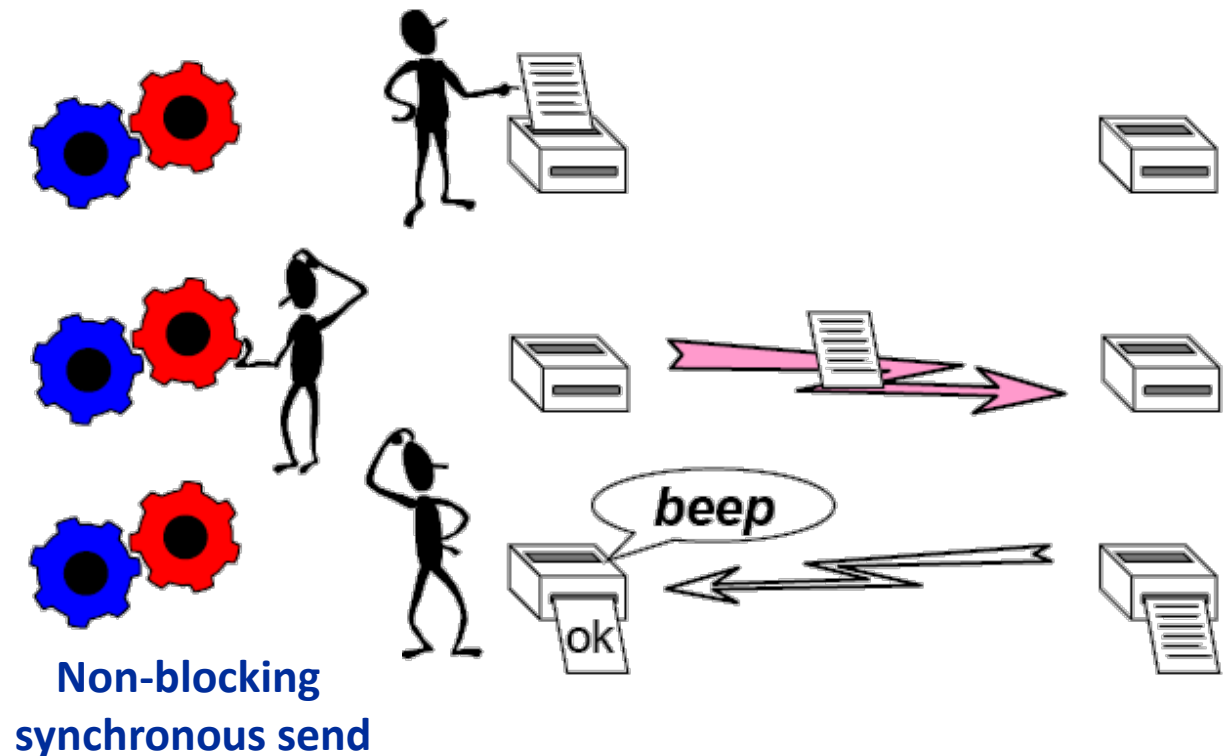
- **Available Send communication modes:**
 - Synchronous communication (performance drawbacks; deadlock dangers)
 - Buffered communication:
 - 1) Copy Send buffer to communication buffer
 - 2) MPI call completes
 - 3) MPI handles communication
 - Behavior of standard send can be synchronous or buffered or depending on message length – no guarantee about that!



Non-Blocking Point-to-Point Communication in MPI



- After initiating the communication one can return to perform other work.
- At some later time one must *test* or *wait* for the completion of the non-blocking operation.

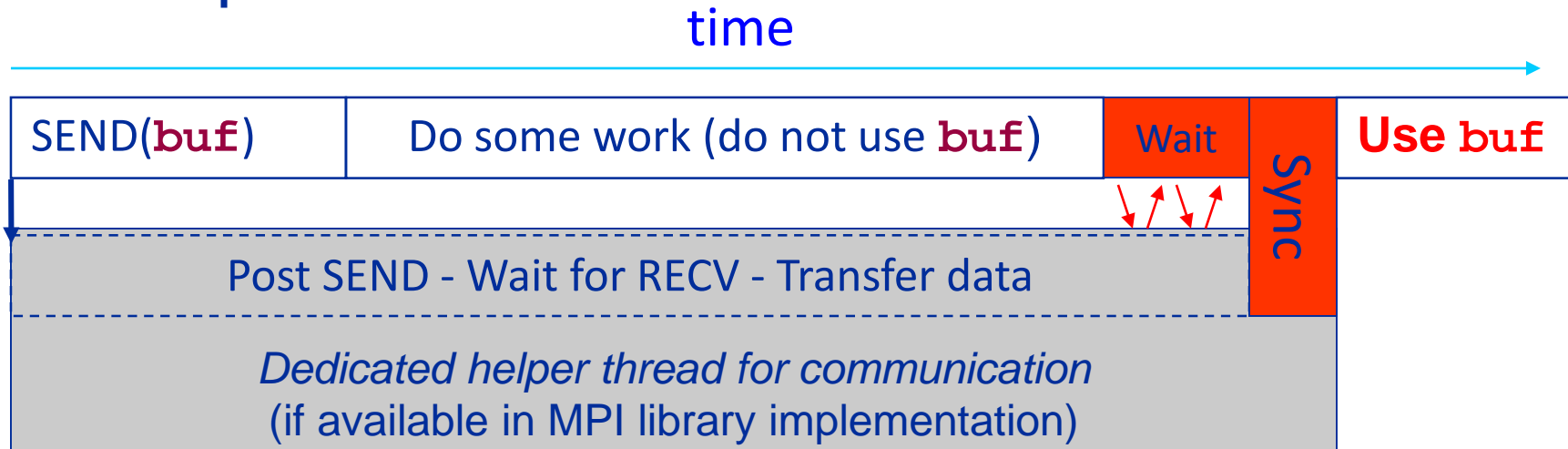




- **Motivation:**

- Avoid deadlocks
- Avoid idle processors
- Avoid useless synchronization
- Overlap communication and useful work (hide the ‘communication cost’; “overlap communication and computation”)
 - **However: This is not guaranteed by the standard!**

- **Principle:**



If no helper thread is available: Data transfer happens in SEND or WAIT step!



Detailed steps for non-blocking communication

- 1) Setup communication operation (MPI)
- 2) Build unique **request handle** (MPI)
- 3) Return **request handle** and control to user program (MPI)
- 4) User program continues while MPI library hopefully performs communication asynchronously
- 5) Status of a communication can be probed using its unique **request handle**

All non-blocking operations must have matching wait (or test) operations as some system or application resources can be freed only when the non-blocking operation is completed.



- The return of non-blocking communication call **does not imply completion** of the communication
- Check for completion: Use **request handle** !
- **Do not reuse buffer** until completion of communication has been checked !
- Data transfer can be overlapped with user program execution **(if supported by hardware and MPI implementation)**
 - Most “standard” implementations today do not support fully asynchronous transfers



- Communication models for non-blocking communication

Non-Blocking Operation	MPI call
Standard send	<code>MPI_ISEND()</code>
Synchronous send	<code>MPI_ISSEND()</code>
Buffered send	<code>MPI_IBSEND()</code>
Ready send	<code>MPI_IRESEND()</code> <i>DO NOT USE</i>
Receive	<code>MPI_Irecv()</code>



- A blocking send can be used with a non-blocking receive, and vice-versa.
- Non-blocking sends can use any mode
 - standard – `MPI_ISEND` (← focus on this in next slides)
 - 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...`.
- The non-blocking operation immediately followed by a matching wait is equivalent to the blocking operation, except some Fortran problems.



- **Standard non-blocking send**

```
<type> buf(*)
integer :: count, datatype, dest, tag, comm, request, ierror
call MPI_Isend(buf,           ! message buffer
               count,        ! # of items
               datatype,     ! MPI data type
               dest,         ! destination rank
               tag,          ! message tag
               comm,         ! communicator
               request,     ! request handle (MPI_Request* in C)
               ierror)       ! return value
```

- Do not use **buf** after **MPI_Isend** has returned!
- Successive test/wait for completion operation is required!
- **request** handle is unique identifier for each non-blocking communication

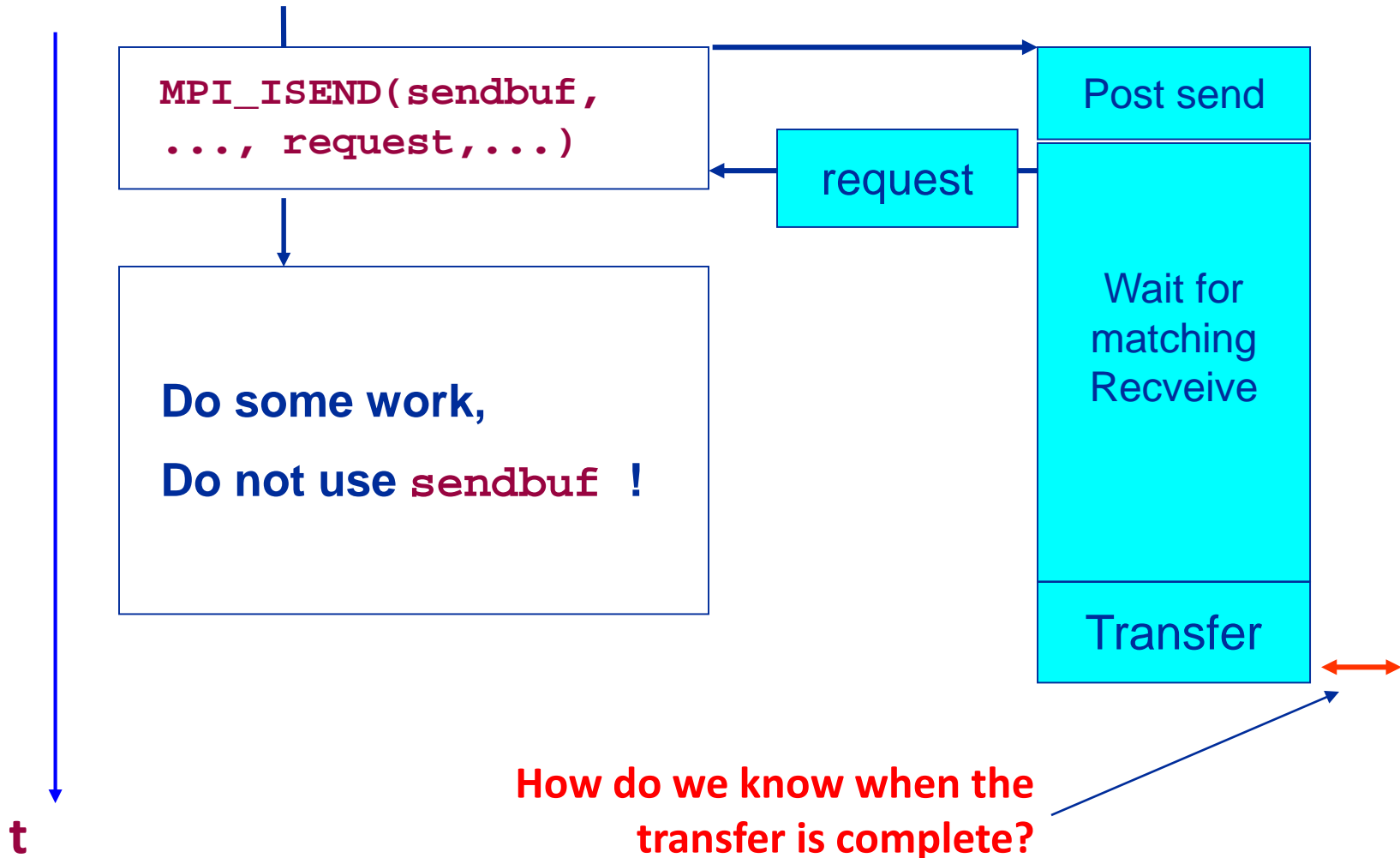


- **Standard non-blocking receive**

```
<type> buf(*)
integer :: count, datatype, source, tag, comm, request, ierror
call MPI_Irecv(buf,           ! message buffer
               count,        ! # of items
               datatype,     ! MPI data type
               source,       ! source rank
               tag,          ! message tag
               comm,         ! communicator
               request,     ! request handle
               ierror)      ! return value
```

- Do not use **buf** after **MPI_Irecv** has returned!
- Successive test/wait for completion operation is required!
- No **status** argument! Is provided at corresponding wait/test operation
- **MPI_Irecv** + immediately following wait/test operation: Potential Fortran pitfall (cf. later)

Non-Blocking Point-to-Point Communication: Standard Send/Receive MPI_ISEND/Irecv





- Ensure that communication has completed before the send/receive buffer is reused.
- MPI supports multiple concurrent non-blocking calls (“open communications”)
- MPI provides two test modes:
 - **WAIT** type: Wait until the communication has been completed and buffer can safely be reused: **Blocking**
 - **TEST** type: Return TRUE (FALSE) if the communication has (not) completed: **Non-blocking**
- **Check a single message request for completion:**

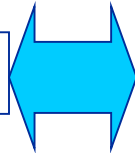
```
logical :: flag
integer :: request, status(MPI_STATUS_SIZE), ierror
call MPI_Test(request,      ! pending request handle
              flag,        ! true if request complete (int* in C)
              status,      ! status object
              ierror)      ! return value
call MPI_Wait(request,     ! pending request handle
              status,      ! status object
              ierror)      ! return value
```


Non-Blocking Point-to-Point Communication: Example: Wait for Completion



- Examples:

```
MPI_SEND(buf,...)
```



```
integer request  
.....  
call MPI_ISEND(buf,...,request,...)  
call MPI_WAIT(request,status,...)
```

```
integer request;  
.....  
call MPI_IRECV(buf,...,request,...);  
  
! DO SOME WORK  
! DO NOT USE buf  
  
call MPI_WAIT(request,status,ierror);  
buf(1) = 42.0
```

Non-Blocking Point-to-Point Communication: Testing for completion of multiple open communications



- Test **multiple** open communications for completion:
 - Test *all* / *any* / *some* of the communication requests

Test for completion	WAIT type (blocking)	TEST type (query only)
At least one; return exactly one	<code>MPI_WAITANY</code>	<code>MPI_TESTANY</code>
Every one	<code>MPI_WAITALL</code>	<code>MPI_TESTALL</code>
At least one; return all which completed	<code>MPI_WAITSSOME</code>	<code>MPI_TESTSSOME</code>

```
integer :: count, requests(*)
integer :: statuses(MPI_STATUS_SIZE,*), ierror
call MPI_Waitall(count,           ! number of requests
                 requests,       ! request handle array
                 statuses,       ! statuses array (MPI_Status* in C)
                 ierror)         ! return value
```

Non-Blocking Point-to-Point Com

Parallel integration

- Wait for completion of the open Irecv operations

```
integer, allocatable, dimension(:, :) :: statuses
integer, allocatable, dimension(:) :: requests
double precision, allocatable, dimension(:) :: tmp
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

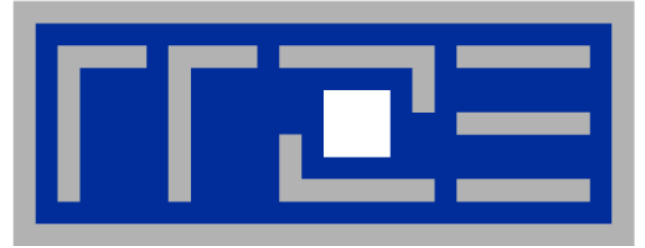
! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

if(rank.eq.0) then
  allocate(statuses(MPI_STATUS_SIZE, size-1))
  allocate(requests(size-1))
  allocate(tmp(size-1))
  ! pre-post nonblocking receives
  do i=1, size-1
    call MPI_Irecv(tmp(i), 1, MPI_DOUBLE_PRECISION, &
                  i, 0, MPI_COMM_WORLD, &
                  requests(i), ierror)
  enddo
endif

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya, myb)

! rank 0 collects partial results
if(rank.eq.0) then
  res=psum
  call MPI_Waitall(size-1, requests, statuses, ierror)
  do i=1, size-1
    res=res+tmp(i)
  enddo
  write (*, *) 'Result: ', res
  ! ranks != 0 send their results to rank 0
else
  call MPI_Send(psum, 1, &
               MPI_DOUBLE_PRECISION, 0, 0, &
               MPI_COMM_WORLD, ierror)
endif
endif
```



Collective Communication in MPI



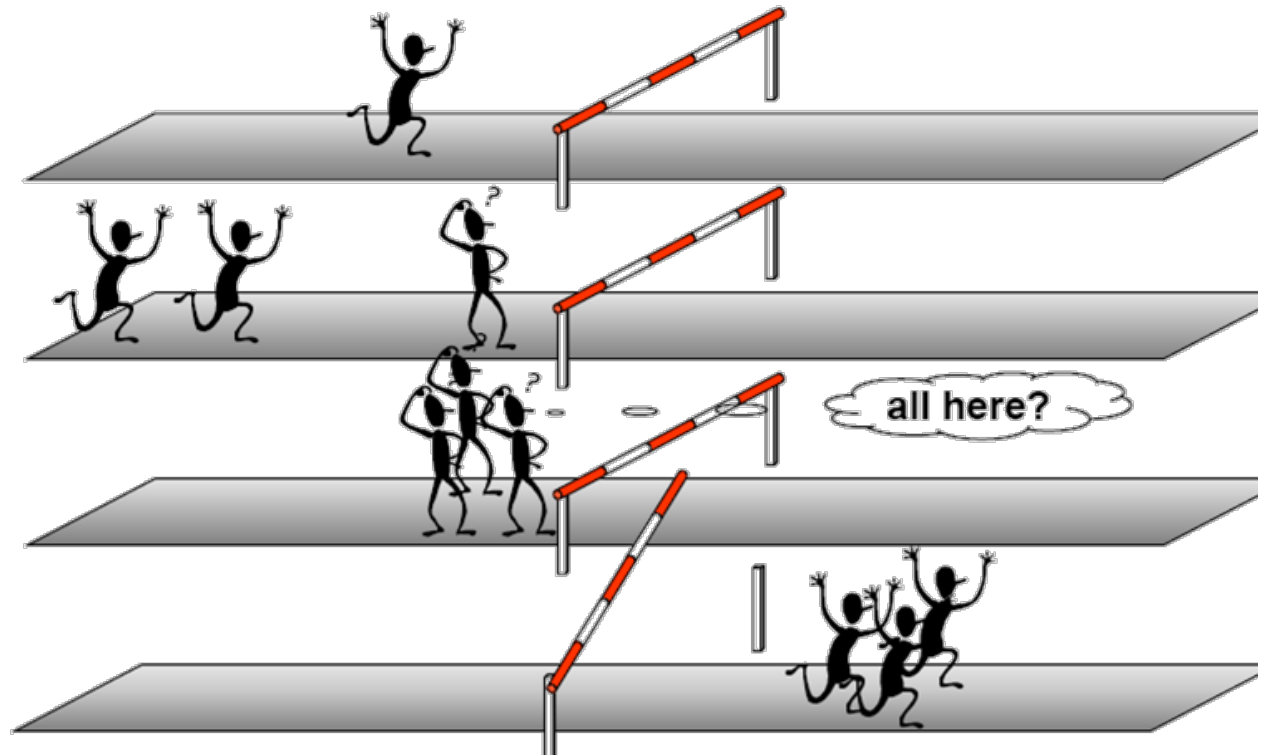
Collective communication always involves every process in the specified communicator

■ Features:

- All processes must call the subroutine
Remarks:
 - All processes must call the subroutine!
 - All processes must call the subroutine!!
 - Blocking and (since MPI 3.0) non-blocking variants
 - May or may not synchronize the processes
 - **Cannot interfere with point-to-point communication**
 - Datatype matching
 - No tags
 - Sent message **must** fill receive buffer (count is exact)
-
- Can be “built” out of point-to-point communications by hand, however, collective communication may allow optimized internal implementations, e.g., tree based algorithms



- Synchronize processes:

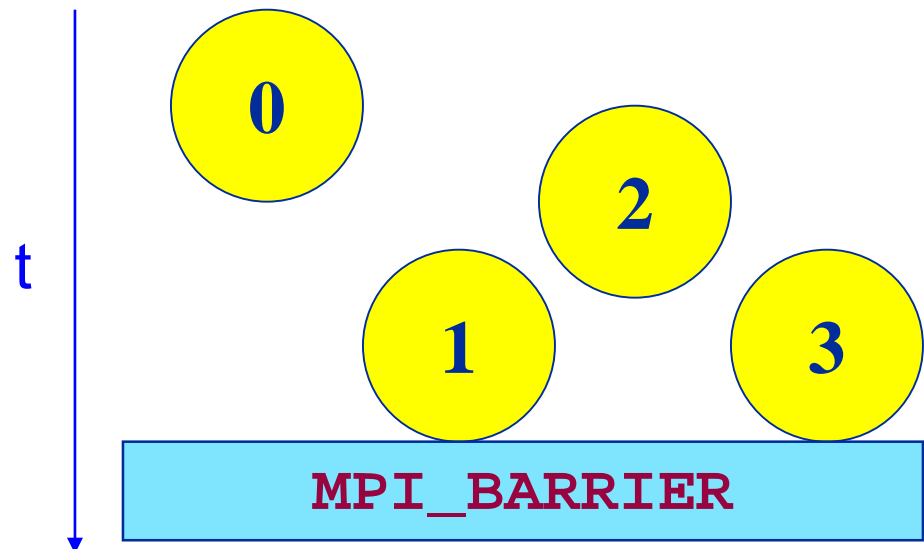




- **Syntax:**

```
MPI_BARRIER(comm, ierror)
```

- **MPI_BARRIER** blocks the calling process until all other group members (=processes) have called it.
- **MPI_BARRIER** is hardly ever needed – all synchronization is done automatically by the data communication – however: debugging, profiling, ...



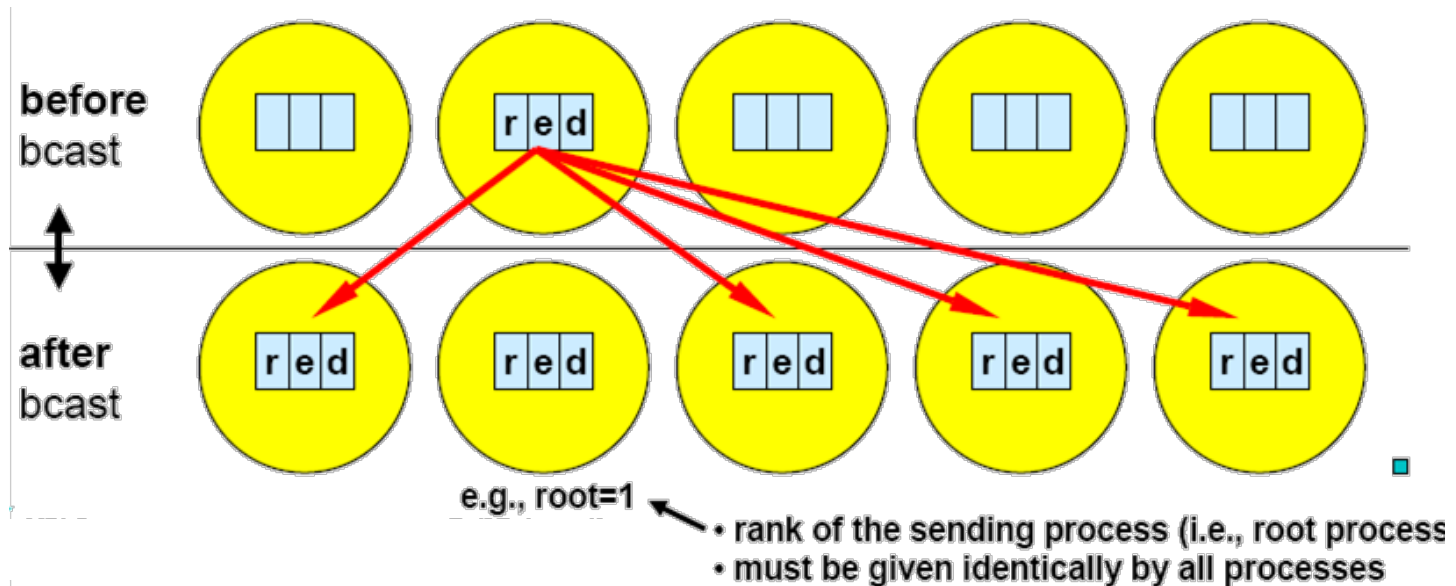


- A one-to-many communication.





- Every process receives one copy of the message from a root process



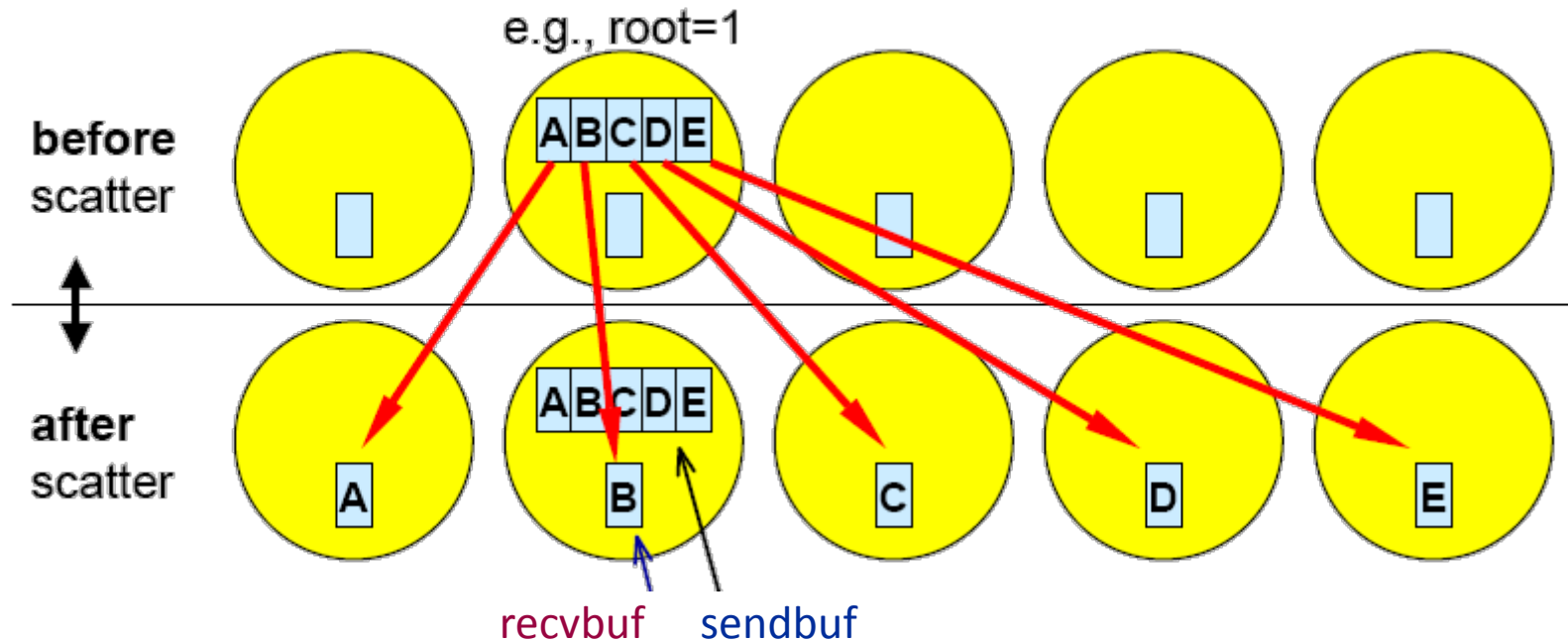
Syntax:

call `MPI_BCAST(buffer, count, datatype, root, comm, ierror)`

(root may be 0, but there is no "default" root process)



- Root process scatters data to all processes



- Specify root process (cf. example : `root=1`)
- send and receive details are different
- SCATTER: send-arguments significant only for root process



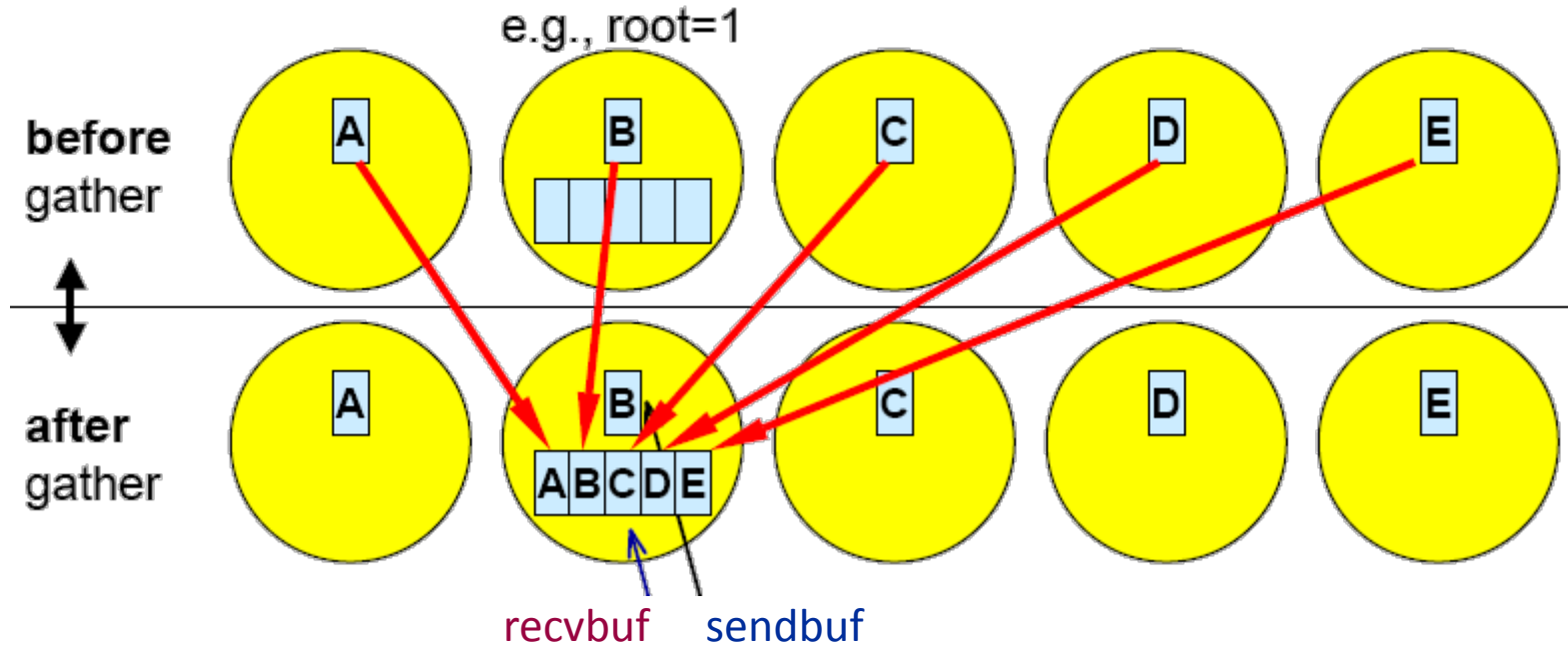
- `MPI_SCATTER`(`sendbuf`, `sendcount`, `sendtype`, `recvbuf`,
`recvcount`, `recvtype`, `root`, `comm`, `ierror`)
- root process sends the i-th. segment of `sendbuf` to the i-th process
- In general: `recvcount` = `sendcount`
- `sendbuf` is ignored for all non-root processes
- More flexible: Send segments of different sizes to different processes!
- `MPI_SCATTERV`(`sendbuf`, `sendcount`, `displ`,
`sendtype`, `recvbuf`, `recvcount`, `recvtype`,
`root`, `comm`, `ierror`)

`integer sendcount(*)`, `displ(*)`

Contains message count for each procs. “Pointer” to starting address in `sendbuf`



- Root process gathers data from all processes



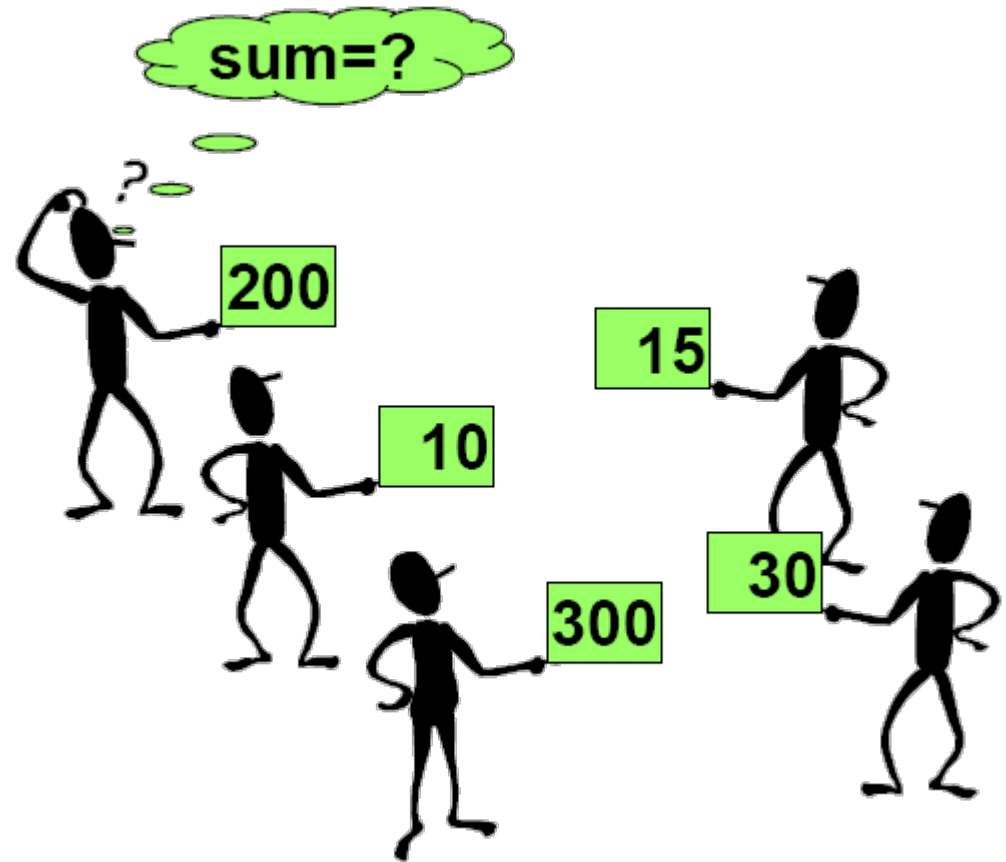
- Specify root process (cf. example : `root=1`)
- send and receive details are different
- GATHER: receive-arguments significant only for root process



- **Gather:** `MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)`
- Each process sends `sendbuf` to `root` process
- `root` process receives messages and stores them in rank order
- In general: `recvcount = sendcount`
- `recvbuf` is ignored for all non-root processes
- `MPI_GATHERV` is available also



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





Compute a result which involves **data distributed** across all processes of a group

- **Example: global maximum of a variable: $\max(\text{var}[\text{rank}])$**
- **MPI provides 12 predefined operations**
- **Definition of user defined operations:**

`MPI_OP_CREATE` & `MPI_OP_FREE`

- **MPI assumes that the operations are associative!**

(floating point operations may be not exactly associative because of rounding errors)



Name	Operation	Name	Operation
MPI_SUM	Sum	MPI_PROD	Product
MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_LAND	Logical AND	MPI_BAND	Bit-AND
MPI_LOR	Logical OR	MPI_BOR	Bit-OR
MPI_LXOR	Logical XOR	MPI_BXOR	Bit-XOR
MPI_MAXLOC	Maximum+ Position	MPI_MINLOC	Minimum+ Position



- Results stored on root process:

```
call MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, &  
               root, comm, ierror)
```

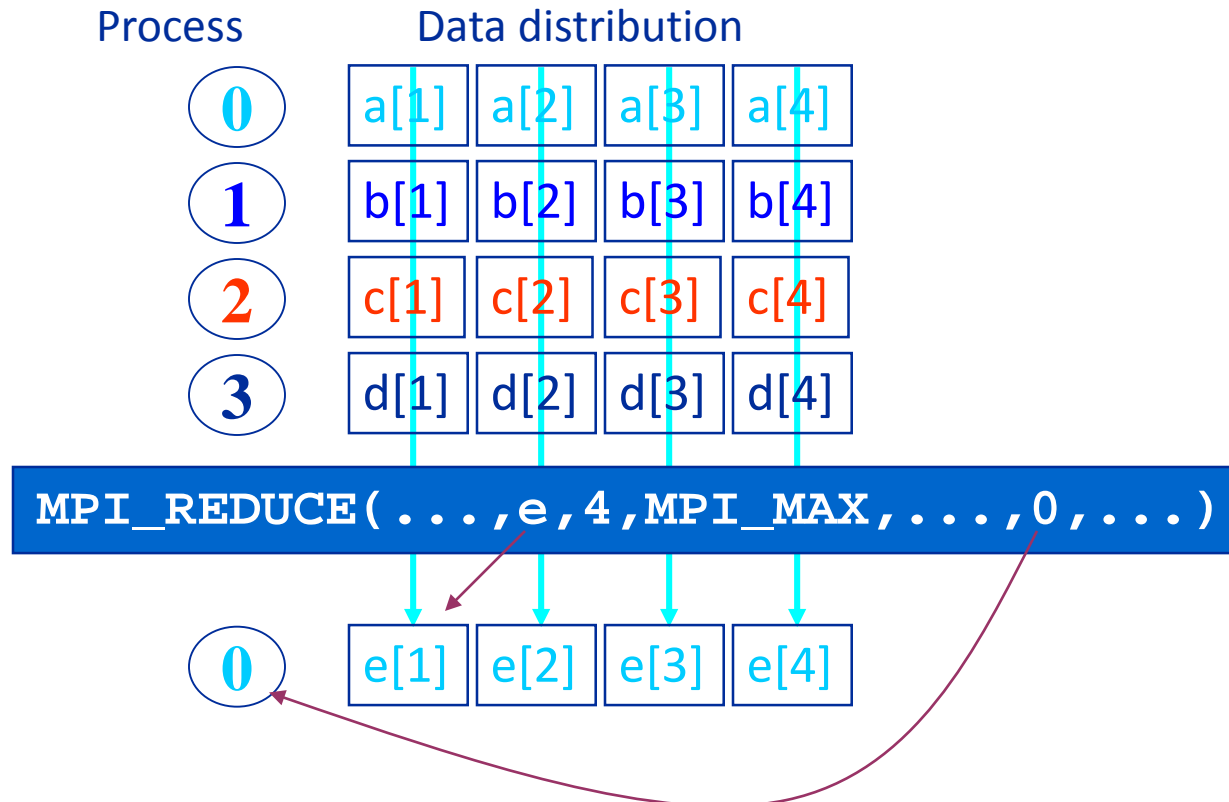
- Result in **recvbuf** on **root** process.
- Status of **recvbuf** on other processes is **undefined**.
- **count > 1**: Perform operations on all 'count' elements of an array

If results are desired to be stored on all processes:

- **MPI_ALLREDUCE**: No **root** argument
 - combination of **MPI_REDUCE** with successive of result from root process
- **MPI_BCAST**



Compute $e(i) = \max\{a(i), b(i), c(i), d(i)\}$
($i=1, 2, 3, 4$)



Global Reduction: Parallel integration



```
integer, allocatable, dimension(:,:) :: statuses
integer, allocatable, dimension(:) :: requests
double precision, allocatable, dimension(:) :: tmp
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

if(rank.eq.0) then
  allocate(statuses(MPI_STATUS_SIZE, size-1))
  allocate(requests(size-1))
  allocate(tmp(size-1))
! pre-post nonblocking receives
  do i=1,size-1
    call MPI_Irecv(tmp(i), 1, MPI_DOUBLE_PRECISION, &
                  i, 0, MPI_COMM_WORLD, &
                  requests(i), ierror)

  enddo
endif

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya,myb)

! rank 0 collects partial results
if(rank.eq.0) then
  res=psum
  call MPI_Waitall(size-1, requests, statuses, ierror)
  do i=1,size-1
    res=res+tmp(i)
  enddo
  write (*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
else
  call MPI_Send(psum, 1, &
               MPI_DOUBLE_PRECISION, 0, 0, &
               MPI_COMM_WORLD,ierror)
endif
```

```
call MPI_Comm_size(MPI_COMM_WORLD, &
                   size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, &
                   rank, ierror)
```

```
! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size
```

```
! integrate f(x) over my own chunk
psum = integrate(mya,myb)
```

```
call MPI_Reduce(psum, res, 1, &
               MPI_DOUBLE_PRECISION, MPI_SUM, &
               0, MPI_COMM_WORLD, ierror)
```

```
if(rank.eq.0) write(*,*) 'Result: ',res
```



Point-to-Point

Collective

Can be mixed, e.g. MPI_Send can be matched by appropriate MPI_Irecv(...)	Blocking	MPI_SEND(mybuf...) MPI_SSEND(mybuf...) MPI_BSEND(mybuf...) MPI_RECV(mybuf...) MPI_SENDRECV(mybuf...) (mybuf can be modified after call returns)	MPI_BARRIER(...) MPI_BCAST(...) MPI_ALLREDUCE(...) (All processes of the communicator must call the operation!)
	Non-blocking	MPI_ISEND(mybuf...) MPI_Irecv(mybuf...) MPI_WAIT/MPI_TEST MPI_WAITALL/ MPI_TESTALL (mybuf may only be modified after WAIT/TEST)	MPI 3.0 only



Efficient numerical simulation on multicore processors (MuCoSim)

- 5 ECTS: 2 hrs → Monday 16:00 – 17:30
- Implementing, optimizing and performance modelling of kernels and miniapps
- For a list of relevant topics see e.g (SS 2019 MUCoSim webpage → univis)

<https://moodle.rrze.uni-erlangen.de/course/view.php?id=395>

- Other topics are welcome
- Close interaction with tutor is expected