



Technical University of Denmark



High Performance Computing

FORTTRAN, OpenMP and MPI

41391



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Content – MPI: Day 2

- Message passing.
- Point-to-point communication.
- Collective communication.

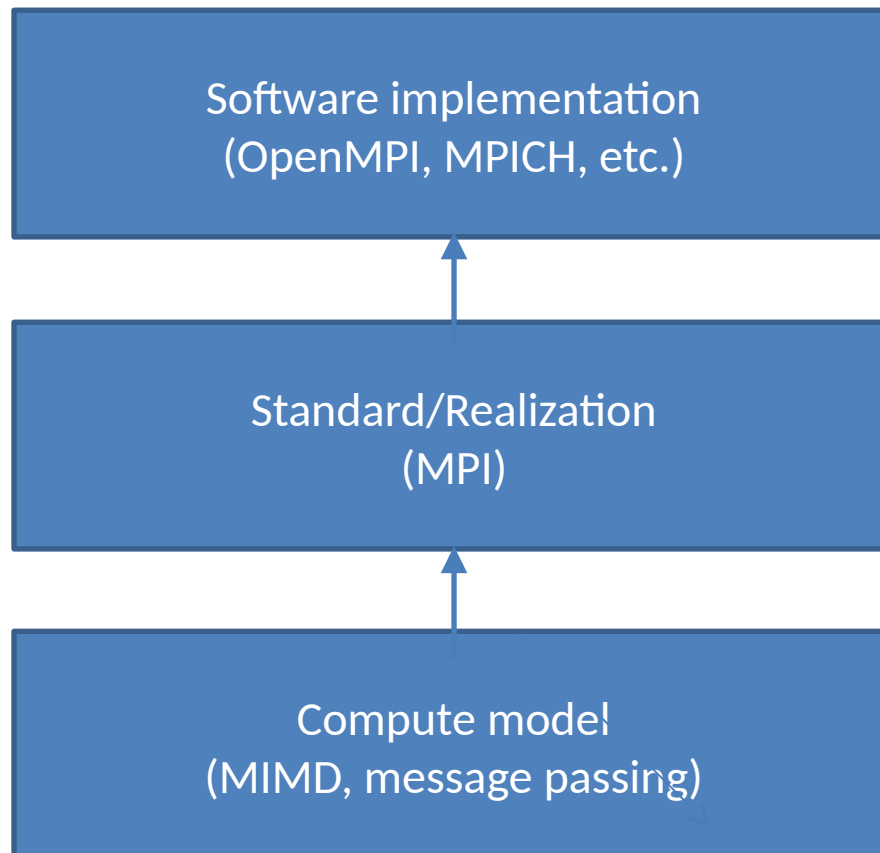
Message passing

What is MPI ?

- A message passing interface standard
 - MPI-1.0: 1993.
 - MPI-1.1: 1995.
 - MPI-2.0: 1997 (backward-compatible superset).
 - MPI-3.0: 2012 (MPI-3.1: 2015).
 - MPI-4.0: 2021.
- Abstraction from machine/OS-specific details.
- Vendor-specific implementations: free implementation: MPICH, LAM, OpenMPI.
- Portable to all shared- and distributed- (and hybrid-) memory machines.
- Efficiently implemented.
- Available everywhere.

Message passing

What is MPI ?



Message passing

What is MPI not ?

- It is not a programming *language* (rather a realization of a compute model).
- It is not a new way of parallel programming (rather a realization of the old message passing paradigm that was around before as POSIX sockets).
- It is not *automatically* parallelizing code (rather, the programmer gets full manual control over all communications).
- It is not dead (rather here to stay!)

Message passing

What can MPI **not** do?

- Give you control over the assignment of processes to physical processors.
 - If running on a heterogeneous cluster, run small test benchmarks to probe processor speeds and account for them in load distribution.
- Detect failing processes or recover from failure.
 - If one process fails, all other wishing to communicate with it will hang (wait forever).
- Detect deadlocks or communication timeouts.
 - Hanging processes will hang forever. There is no timeout.

Message passing

Literature:

- M. Snir, W. Gropp: “MPI: The Complete Reference” (2 vol.), MIT Press, 1998.
- W. Gropp, E. Lusk, and R. Thakur: “Using MPI: Portable Parallel Programming with the Message Passing Interface”, MIT Press, 1999.
- W. Gropp, E. Lusk, and R. Thakur: “Using MPI-2: Advanced Features of the Message Passing Interface”, MIT Press, 1999.
- P. S. Pacheco: “Parallel Programming with MPI”, Morgan Kaufmann Publishers, 1997 (very good introduction to MPI programming).
- www.mpi-forum.org/docs/mpi-3.1/index.htm
- www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

Message passing

Control structures and communication models:

- Multiple Instructions Multiple Data (MIMD).
- Single Program Multiple Data (SPMD: all processes run the same program; variables have the **same name**, but are stored in different locations and contain **different data** !)
- Message-passing (communication using explicit send/receive operations).
- While message passing is generally non-deterministic, MPI guarantees that the messages arrive in the **order send**: “Message do not overtake each other.”

Message passing

Data distribution:

- Processes have disjoint and separated memory address spaces.
- Each process is identified by its *rank* (returned by a special MPI routine): $rank=0, \dots, (size-1)$.
- All data and operation distribution decisions are based (by the programmer) on the value of *rank*.

```
PROGRAM main
INCLUDE 'mpif.h'
CALL MPI_Init(ierr)
CALL MPI_Comm_Rank(MPI_COMM_WORLD, irank, ierr)
CALL MPI_Comm_Size(MPI_COMM_WORLD, isize, ierr)
PRINT*, 'rank: ', irank, ' out of: ', isize
```

Message passing

MPI messages:

- Are packets of data moving between processes.
- All messages know:
 - Rank of the sending process and the receiving process.
 - Data type of the message.
 - Source data size (message size).
 - Receiver buffer size.
- Messages are addressed by the rank of the receiver.
- All messages must be received (no loss).

Message passing

Program compilation and start:

- Program must be compiled with the MPI header/include files.
- Program must be linked against the MPI library.
- Program must be started using the MPI startup tool (an executable/shell script that comes with the MPI library that takes the program to be ran as an argument):
 - `f90 -o prg -I<incpath>/ prg.f -lmpi # OR:`
 - `mpif90 -o prg prg.f`
 - `mpirun -np 8 ./prg`

Message passing

Pay attention to:

- Use the **same** compiler for the program as the one that was used for compiling the MPI library (using different compilers for the MPI library and the program will **not** work!).
- Using `mpif90` or `mpifort` to compile/link guarantees this!
- If the machine has no shared file system: make sure to manually copy the executable and all input files to the disks of all nodes before invoking `mpi run`.

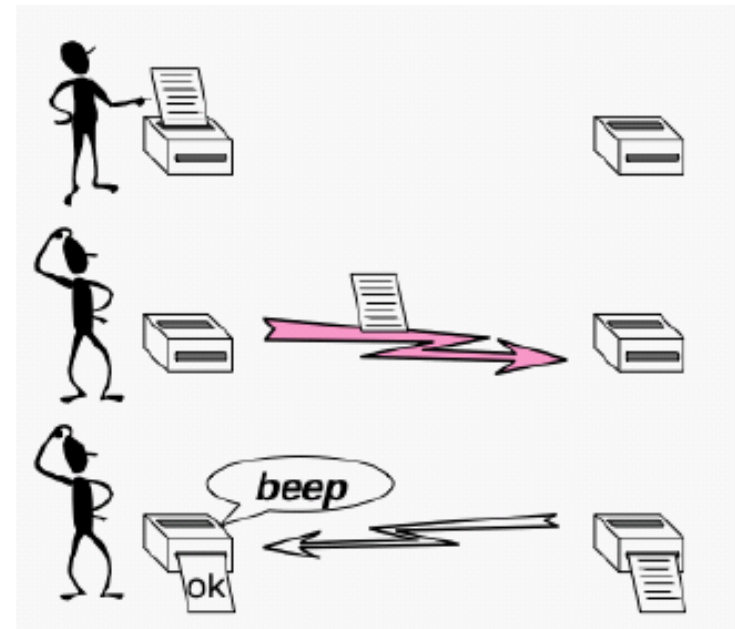
Point-to-point communication

- Simplest form of message passing.
- One process sends a message to another one.
- Like a fax machine.
- Different types:
 - Synchronous.
 - Asynchronous (buffered).

Point-to-point communication

Synchronous:

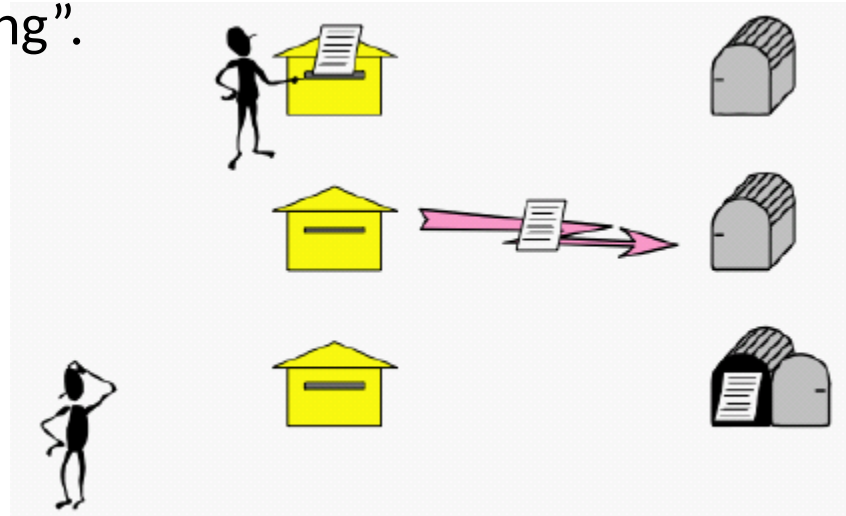
- Message arrives at the same time as it is sent.
- Sender has to wait if receiver is not ready.
- Sender gets confirmation of receipt.
- Analogy: fax machine.
- Synchronous → “blocking”.



Point-to-point communication

Asynchronous:

- Message arrives whenever receiver is ready.
- Sender does not have to wait.
- Sender only knows that the message has left.
- Analogy: surface mail.
- Asynchronous → “non-blocking”.



Point-to-point communication

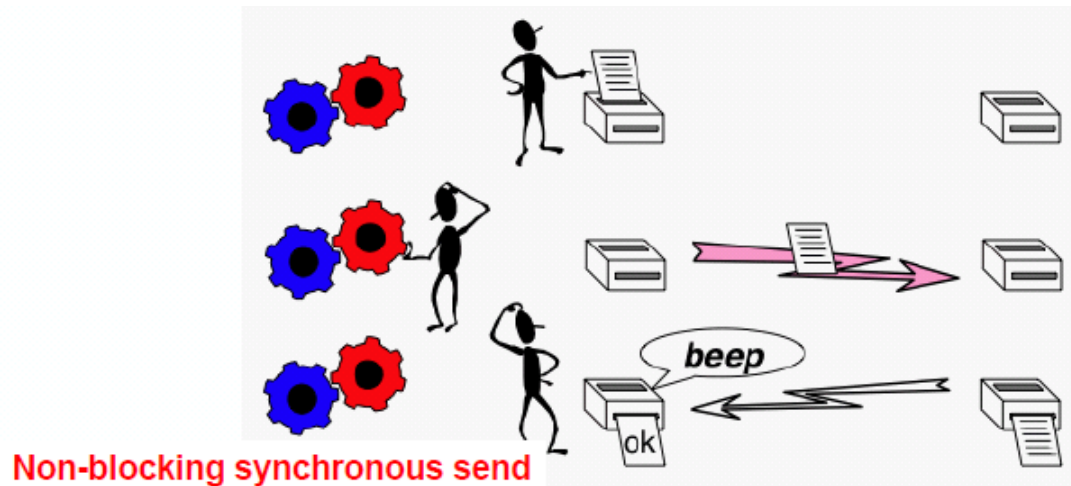
Blocking operations:

- Sending and receiving can be blocking.
- Blocking subroutines returns only after the operation has completed.

Point-to-point communication

Non-blocking operations:

- Non-blocking operations return immediately and allow the calling program to continue.
- At a later point, the program may *test* or *wait* for the completion of the non-blocking operations.



Point-to-point communication

Non-blocking operations:

- All non-blocking operations must have a matching **wait** operation (some system resources can only be freed after the operation has completed).
- A non-blocking operations that is immediately followed by a wait is equivalent to a blocking operation.

Point-to-point communication

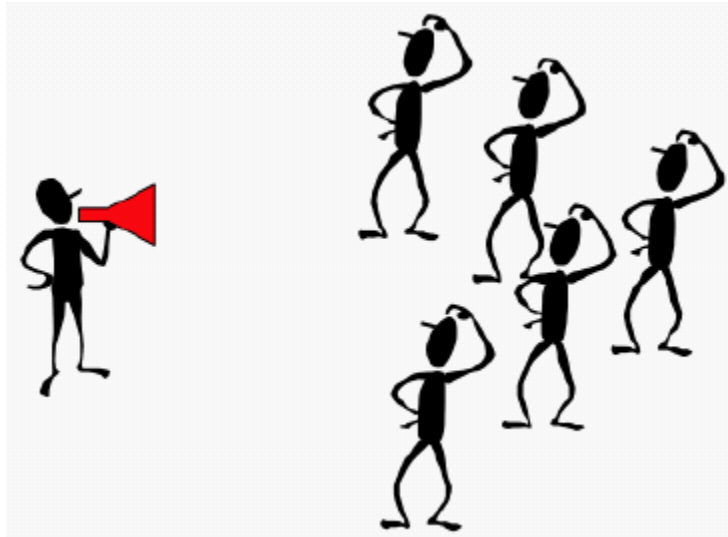
Collective operations:

- Collective communication operations are composed of several point-to-point operations.
- Optimized internal implementations (e.g., tree algorithms) are used in MPI (transparent to the user).
- More important examples:
 - Broadcast.
 - Reduction.
 - Barrier.

Point-to-point communication

Broadcast:

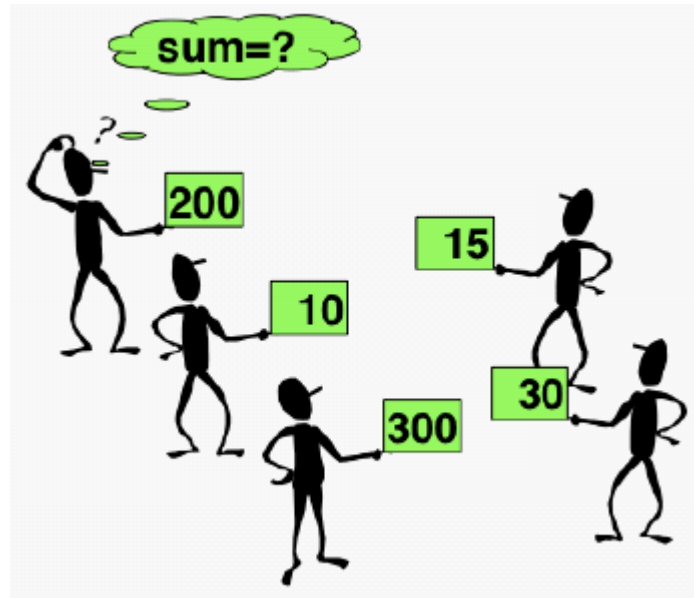
- A one-to-many communication.
- Analogy: speech, radio, TV, etc..



Point-to-point communication

Reduction:

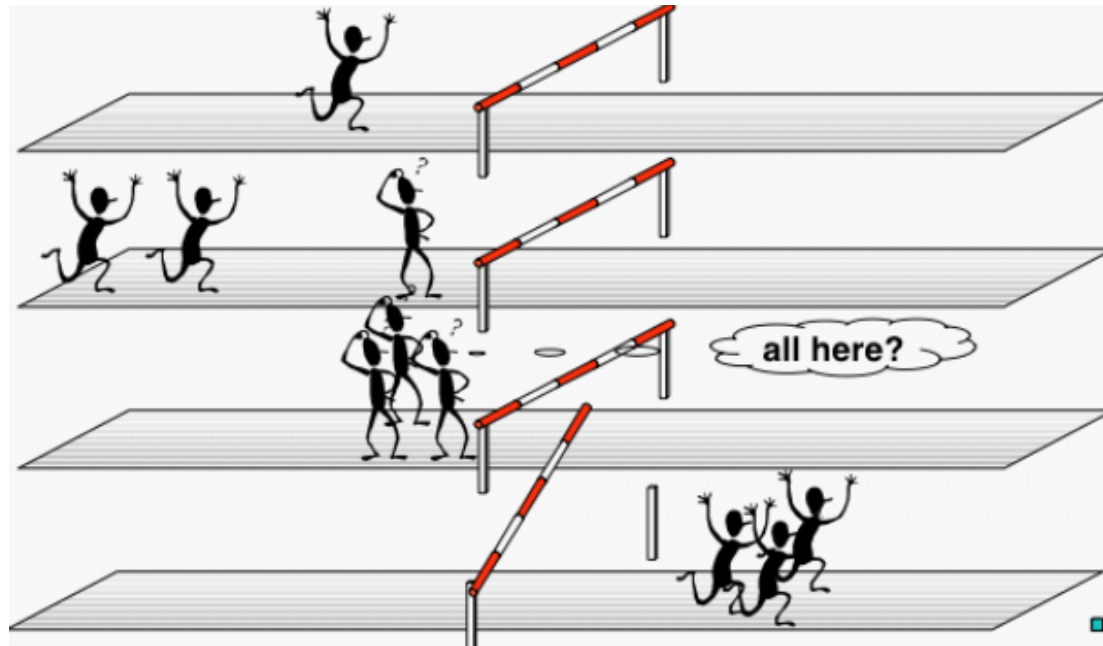
- Combine data from several processes into a single result (possibly followed by a broadcast of the results to all contributors).



Point-to-point communication

Barrier:

- Synchronize all processes:



Fortran and MPI

- Standard: Fortran, C, C++ (only since MPI-2).
- 3rd party: python, matlab, java, ...
- Fortran header file:
 - INCLUDE 'mpif.h' or module: mpi_f08 or the module mpi.
 - It is recommended to use the module solution as it provides implicit interfaces to all MPI calls and hence will check for missing arguments etc.

Fortran and MPI

MPI function format:

- `CALL MPI_xxxxx(parameter,...,ierror)`



Difference to C language binding;
Absolutely NEVER forget !

- MPI_ namespace is reserved for MPI constants and routines. Application routine and variables name must not begin with MPI_.

Fortran and MPI

A minimal set of MPI routines:

- `MPI_Init(ierr)`
- `MPI_Finalize(ierr)`
- `MPI_Comm_Rank(comm, rank, ierr)`
- `MPI_Comm_Size(comm, size, ierr)`
- `MPI_Send(...)`
- `MPI_Recv(...)`
- `MPI_SendRecv(...)`
- `MPI_BCast(...)`
- `MPI_Reduce(...)`

Fortran and MPI

Initializing MPI:

- `CALL MPI_Init(ierr)`

```
PROGRAM test
  IMPLICIT NONE
  INCLUDE 'mpif.h'
  INTEGER :: ierr
  CALL MPI_Init(ierr)
```

- Must always be the first MPI routine that is called in a program. Creates the parallel process.

Fortran and MPI

MPI communicator:

- All processes of one MPI program are combined in the global communicator `MPI_COMM_WORLD`.
- `MPI_COMM_WORLD` is a predefined handle in `mpif.h`
- Each process has a rank: $rank = 0, \dots, (size-1)$
- Custom communicators can be defined to group processes.

Fortran and MPI

MPI rank:

- Identifies a process.
- Is the basis for all work distribution (by the user).
- Each process can inquire its rank within the communicator using `MPI_Comm_Rank`

```
PROGRAM test
IMPLICIT NONE
INCLUDE 'mpif.h' ! defines MPI_COMM_WORLD
INTEGER :: rank,ierror
CALL MPI_Init(ierror)
CALL MPI_Comm_Rank(MPI_COMM_WORLD,rank,ierror)
```

Fortran and MPI

MPI rank:

- Identifies a process.
- Is the basis for all work distribution (by the user).
- Each process can inquire its rank within the communicator using `MPI_Comm_Rank`

```
PROGRAM test
USE mpi ! Defines interface and constants
IMPLICIT NONE
INTEGER :: rank,ierror
CALL MPI_Init(ierror)
CALL MPI_Comm_Rank(MPI_COMM_WORLD,rank,ierror)
```

Fortran and MPI

MPI size:

- Total number of processes in a communicator.
- Can be inquired using `MPI_Comm_Size`

```
PROGRAM test
IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER :: size,ierror
CALL MPI_Init(ierror)
CALL MPI_Comm_Size(MPI_COMM_WORLD,size,ierror)
```

Fortran and MPI

Get name of processor:

```
PROGRAM test
IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER :: ierror,ilen
CHARACTER(LEN=256) :: name
CALL MPI_Init(ierror)
CALL MPI_Get_Processor_Name(name,ilen,ierror)
WRITE(*,'(2A)') 'Processor name: ',name(1:ilen)
END PROGRAM test
```

Fortran and MPI

Terminating MPI

- CALL MPI_Finalize(ierr)

```
PROGRAM test
IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER :: ierr
...
CALL MPI_Finalize(ierr)
END PROGRAM test
```

- Must always be the last MPI routine that is called in a program. Must be called by all processes.
- After MPI_Finalize is called NO MPI routine can be called (even MPI_Init).

Point-to-point communication

MPI point-to-point communication:

- Communication between two processes in a communicator.
- **Source** process sends message to **destination** process.
- Processes are identified by their **rank** within the communicator.
- Messages are of a certain **length** and **data type**.

Point-to-point communication

Basic MPI data types

- Identified by handles via mpif.h
- Different in Fortran and C!

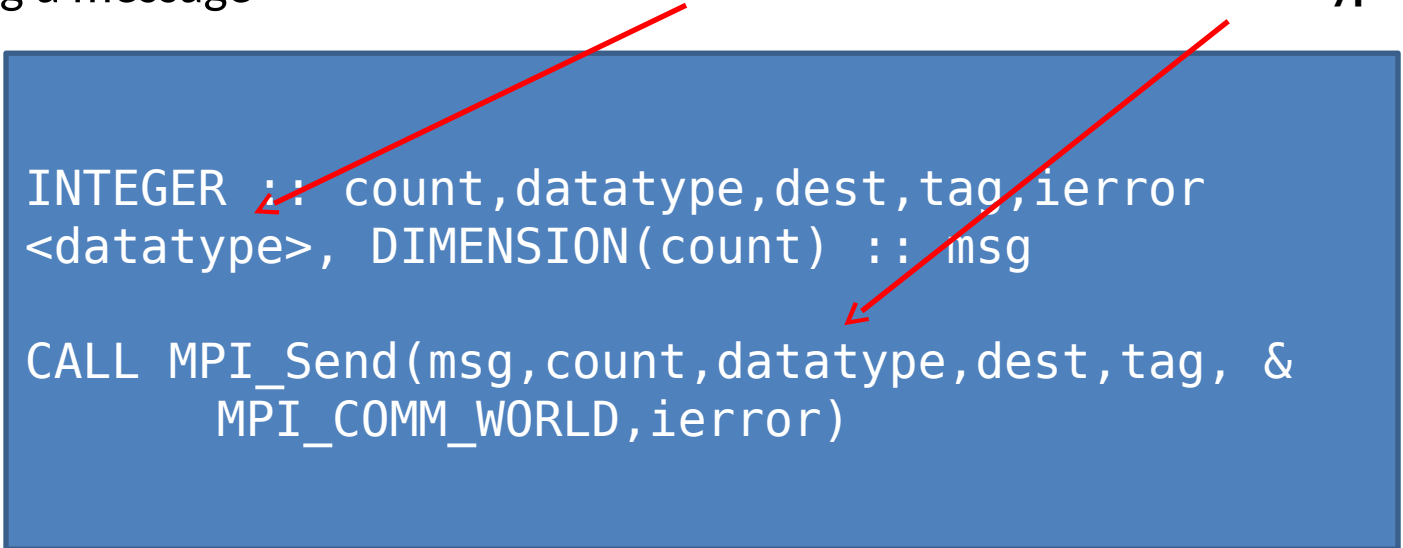
MPI data type handle	Fortran data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)

Point-to-point communication

Sending a message

Fortran data type

MPI data type



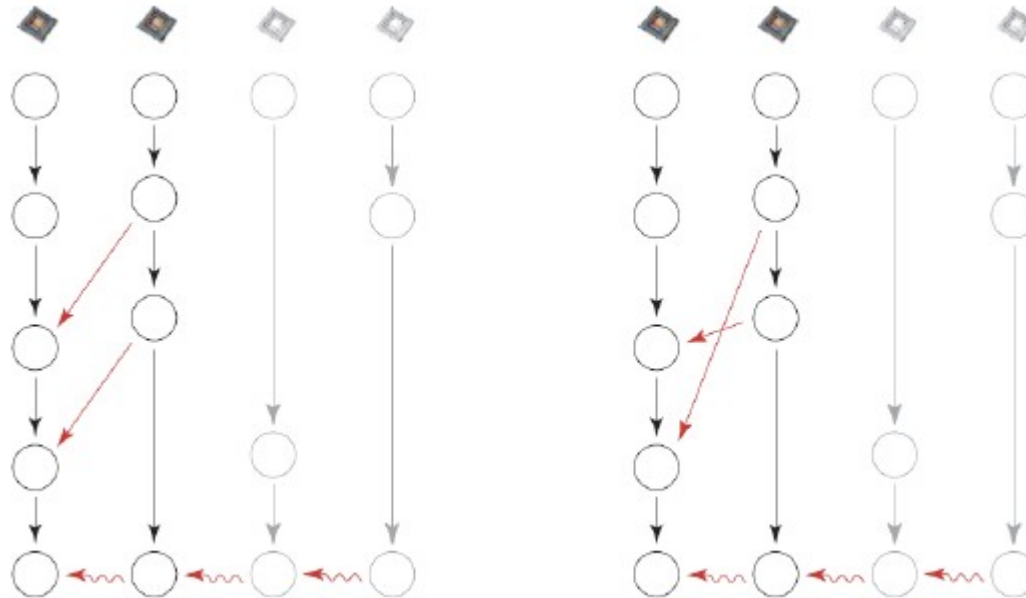
```
INTEGER :: count,datatype,dest,tag,ierror  
<datatype>, DIMENSION(count) :: msg  
  
CALL MPI_Send(msg,count,datatype,dest,tag, &  
             MPI_COMM_WORLD,ierror)
```

The diagram shows a blue rectangular box containing Fortran and MPI code. Above the box, three labels are positioned: 'Sending a message' on the left, 'Fortran data type' in the center, and 'MPI data type' on the right. Two red arrows originate from these labels. One arrow starts from 'Fortran data type' and points to the variable 'count' in the first line of the Fortran code. The other arrow starts from 'MPI data type' and points to the variable 'datatype' in the second line of the Fortran code.

- msg is the message (an array).
- dest is the rank of the destination process.
- tag can be used as message ID to distinguish different messages between the same pair.

Point-to-point communication

MPI tags



- Both are allowed.
- Tags resolve the ambiguity.

Point-to-point communication

Receiving a message

```
INTEGER :: count,datatype,src,tag,ierror  
<datatype>, DIMENSION(count) :: msg  
INTEGER :: status(MPI_STATUS_SIZE)  
  
CALL MPI_Recv(msg,count,datatype,src,tag, &  
             MPI_COMM_WORLD,status,ierror)
```

- msg is the message (an array of size count).
- src is the rank of the source/sender process.
- Only messages with matching tag are received.
- status returns extra information: status(MPI_SOURCE), status(MPI_TAG), status(MPI_ERROR); the size of the received message via routine MPI_Get_Count(status,datatype,count)
- Common **error** to forget the status in MPI_Recv.

Point-to-point communication

Requirements:

- Sender must specify a valid destination rank.
- Received must specify a valid source rank.
- The communicator must be the same.
- Message tags must match.
- Message data types must match.
- Receive buffer length must be $\geq \text{sizeof}(\text{message})$

Otherwise `ierror` returns a non-zero value – test it!

Point-to-point communication

Wildcards:

- Can only be used by the destination process!
 - To receive from any source:
`src = MPI_ANY_SOURCE.`
 - To receive messages with any tags:
`tag = MPI_ANY_TAG.`
- Actual source and tag are returned in `status`.

Point-to-point communication

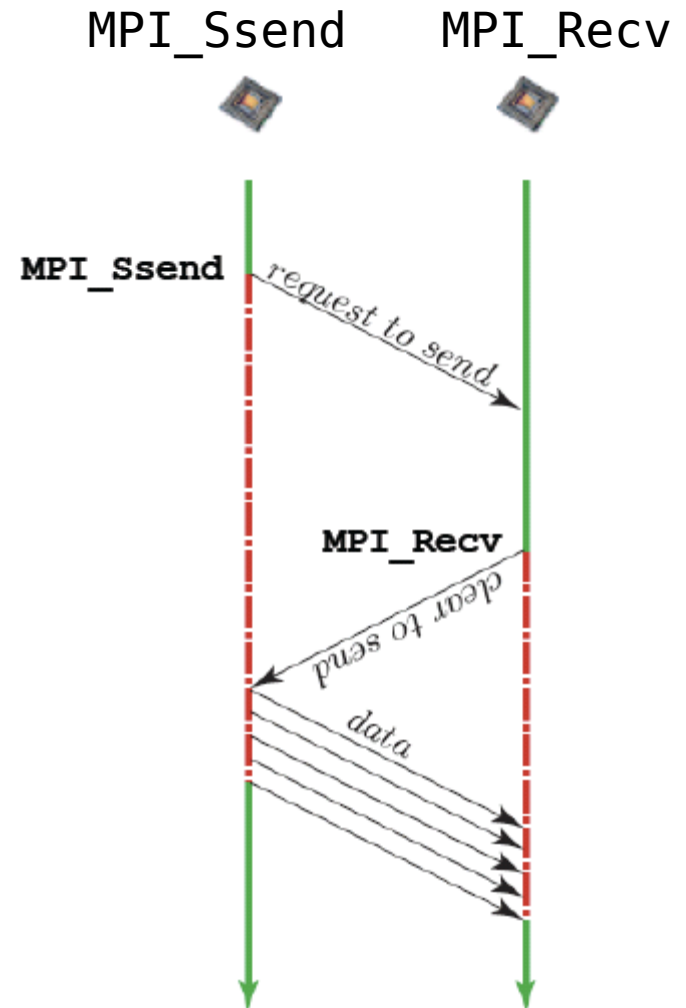
Blocking point-to-point communication modes

- Send modes:
 - Synchronous send (`MPI_Ssend`)
(completes when receive has started on destination process).
 - Buffered (asynchronous) send (`MPI_BSend`)
(completes when buffer left – needs additional buffer).
 - Standard send (`MPI_Send`)
(synchronous OR buffered; depends on implementation).
 - Ready send (`MPI_RSend`)
(can only start after receive has started – dangerous).
- Receiving all modes: (`MPI_Recv`)
(completes when a message has arrived).

Point-to-point communication

Blocking synchronous

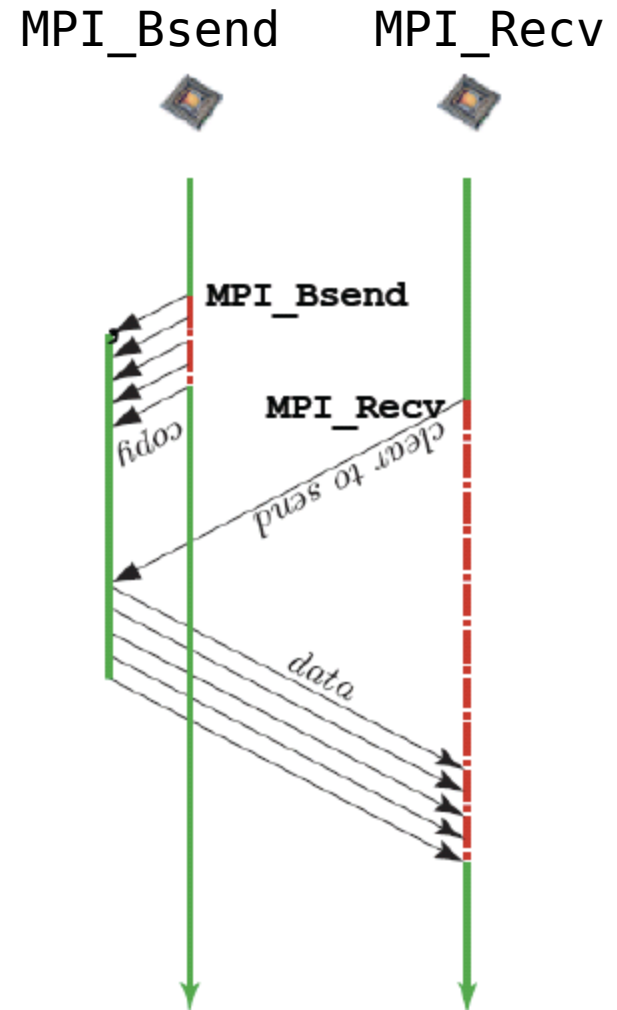
- Returns when message has been started to receive.
- Handshake and waiting.
- Most safe and portable.



Point-to-point communication

Blocking buffered

- Returns when message has been copied into buffer.
- Buffer must be provided by the user: `MPI_buffer_attach(msg, count, ierror)`.
- Fewer waits, but need for additional memory.



Point-to-point communication

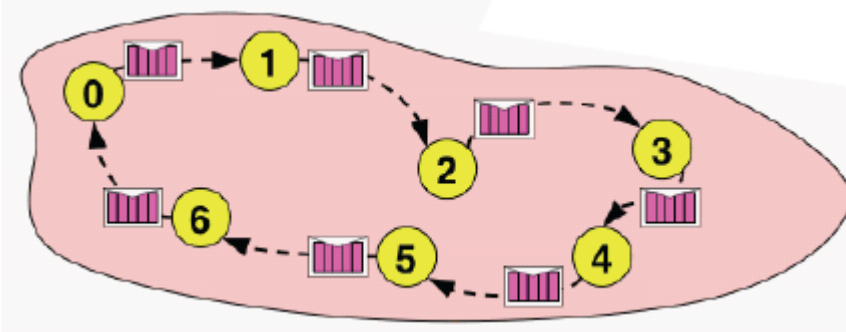
Properties of the communication modes:

- Standard send (`MPI_Send`):
 - Minimum transfer time.
 - All risks of synchronous send if implementation uses the synchronous protocol.
- Synchronous send (`MPI_Ssend`):
 - Risk for deadlock, serialization, waiting.
 - High latency, but best bandwidth.
- Buffered send (`MPI_Bsend`):
 - Low latency and low bandwidth.
 - High memory usage.
- Ready send (`MPI_Rsend`):
 - Low latency and high bandwidth.
 - Must be sure communication channel is already open!

Non-blocking communication

Deadlock

- Code in each MPI process:
 - `MPI_SSend(..., right_rank, ...)`
 - `MPI_Recv(..., left_rank, ...)`



Blocks forever because `MPI_Recv` cannot be called on right-most MPI process!

- Same problem with standard `MPI_Send` if MPI implementation chooses synchronous protocol!

Non-blocking communication

Communication is separated into three phases:

- Initiate communication:
 - Returns immediately.
 - MPI routines starting with MPI_I...
- Do something else (computations) while communication happens in the background.
- Wait for the communication to complete before you need the new data.

Non-blocking communication

Standard non-blocking send/receive:

- Non-blocking send:

```
CALL MPI_Isend(...,handle,...)  
! Do some other work  
CALL MPI_Wait(...,handle,...)
```

- Non-blocking receive:

```
CALL MPI_Irecv(...,handle,...)  
! Do some other work  
CALL MPI_Wait(...,handle,...)
```

Non-blocking communication

Completion:

```
CALL MPI_Wait(handle,status,info)
...
CALL MPI_Test(handle,flag,status,info)
```

- After any non-blocking operation, one must:
 - MPI_Wait OR
 - Loop with MPI_Test until flag = .TRUE. i.e., request is completed.

Non-blocking communication

Wildcards:

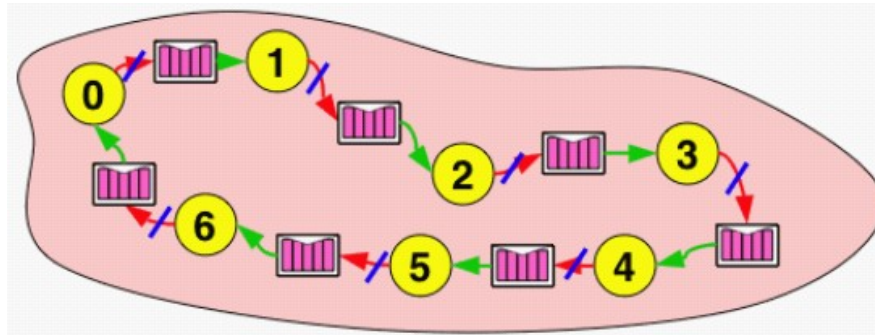
- With several pending request handles, one can:
 - Wait or test for completion of ONE message:
MPI_WaitAny / MPI_TestAny
 - Wait or test for completion for ALL messages:
MPI_WaitAll / MPI_TestAll
 - Wait or test for completion of AS MANY messages AS POSSIBLE:
MPI_WaitSome / MPI_TestSome

Non-blocking communication

No more deadlocks: non-blocking send:

(if you don't use MPI_Wait in the wrong place)

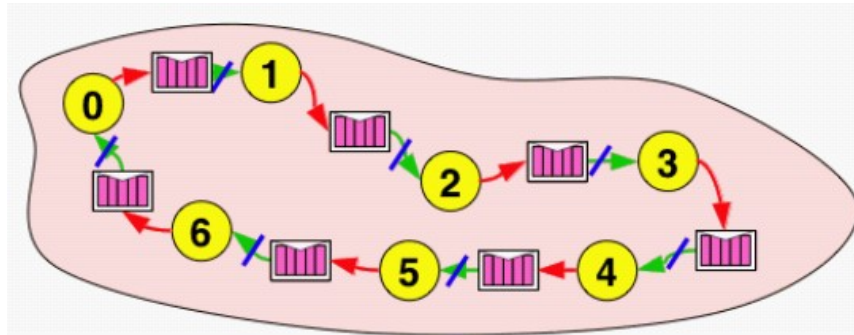
- Initiate non-blocking send:
—→ Ship off the message to the right neighbor
- Do some work.
—→ Receive the message from the left neighbor
- Message transfer can complete:
/ Wait for non-blocking send to complete.



Non-blocking communication

No more deadlocks: non-blocking receive:
(if you don't use MPI_Wait in the wrong place)

- Initiate non-blocking receive:
—> Open receive channel from left neighbor.
- Do some work.
- > Send message to the right neighbor
- Message transfer can complete:
/ Wait for non-blocking receive to complete.



Non-blocking communication

Other use of non-blocking communication:

- Overlap communication and computation in a numerical simulation.
- The internal nodes of a Poisson solve can, e.g., already be computed while the boundary conditions are communicated.
- This greatly improves the parallel efficiency of the program as, ideally, all communication can be “hidden”. (zero communication overhead!).
- **In practice** this requires hardware and OS that support concurrent communication. On most architectures today it is emulated using sequential memory accesses !

Non-blocking communication

Request handles:

- Each non-blocking communication operation generates a request handle of type INTEGER.
- This handle has to be used in test and wait statements to inquire about the status of the operation.

Non-blocking communication

Non-blocking send:

```
INTEGER :: count,datatype,dest,tag
INTEGER :: handle,info
<datatype>, DIMENSION(count) :: msg
INTEGER :: status(MPI_STATUS_SIZE)

CALL MPI_Isend(msg,count,datatype,dest,tag, &
               MPI_COMM_WORLD,handle,info)
CALL MPI_Wait(handle,status,info)
```

- msg is the message to be sent.
- dest is the rank of the receiver process.
- tag is the message ID.

Non-blocking communication

Non-blocking send:

- msg must NOT be used between ISend and Wait (MPI-1.1, page 40).
- Calling Wait directly after ISend is equivalent to a blocking call (SSend).
- See MPI-2, Chap. 10.2.2 (pp. 284-290) for open issues and problems.

Non-blocking communication

Non-blocking receive:

```
INTEGER :: count,datatype,src,tag
INTEGER :: handle,info
<datatype>, DIMENSION(count) :: msg
INTEGER :: status(MPI_STATUS_SIZE)

CALL MPI_IRecv(msg,count,datatype,src,tag, &
               MPI_COMM_WORLD,handle,info)
CALL MPI_Wait(handle,status,info)
```

- msg is the receive message.
- src is the rank of the send process.
- tag is the message ID.
- handle returns the request handle.

Non-blocking communication

Non-blocking receive:

- msg must NOT be used between IRecv and Wait (MPI-1.1, page 40).
- Compiler does not see modifications in msg before MPI_Wait. Call MPI_Address(msg,...) after MPI_Wait to get updated values.
- See MPI_2, Chap. 10.2.2 (pp. 284-290) for open issues and problems.

Non-blocking communication

Problem with small, **local** variables:

```
CALL MPI_IRecv(msg,...,handle,info)
CALL MPI_Wait(handle,status,info)
WRITE(*,*) msg
```

Old data are written instead of the newly received ones ! Why ?

Because the compiler might do:

```
CALL MPI_IRecv(msg,...,handle,info)
registerA = msg
! Receives data into msg, but this is not
! visible in the source code !
CALL MPI_Wait(handle,status,info)
WRITE(*,*) registerA
```

Non-blocking communication

Solution:

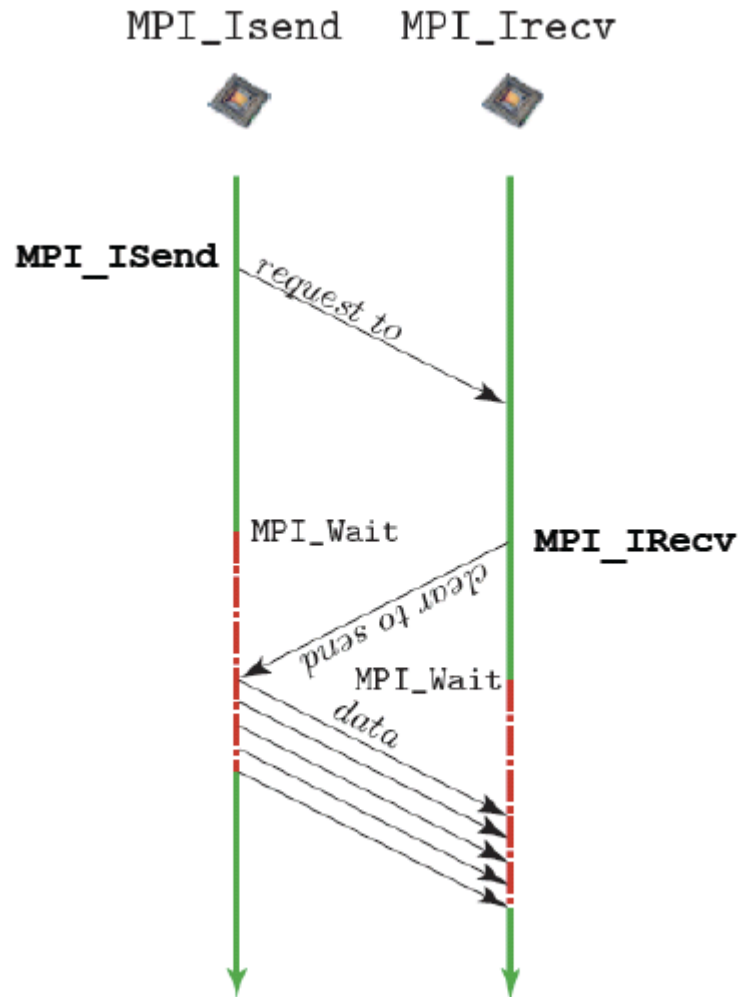
- Allocate msg in a module or a common block OR
- Call MPI_Address(msg,...) after MPI_Wait:

```
CALL MPI_IRecv(msg,...,handle,info)
CALL MPI_Wait(handle,status,info)
CALL MPI_Address(msg,address,info)
WRITE(*,*) msg
```

Now the compiler sees that msg could potentially change before WRITE.

Non-blocking communication

Example:



Non-blocking communication

```
MPI_Isend(msg(7, :, :), ..., handle, ierror)
```

- Content of non-contiguous sub-array is stored in a temporary array
- Then MPI_Isend is called
- On return the temporary array is DEALLOCATED

! Do some other work

- The data are communicated meanwhile...

```
MPI_Wait(handle, status, ierror)
```

- ...or in MPI_Wait, but the **data** in the temporary array **are already lost!!**

Non-blocking communication

Solution:

- Do NOT use non-contiguous sub-arrays in non-blocking calls!
- For contiguous sub-arrays use first sub-array element `msg(1,1,9)` instead of whole sub-array `buf(:, :, 9:13)`.
- Use compiler flags to prohibit call by in-and-out-copy.
- The same problematic also exists for POINTERS!

Non-blocking communication

Summary: blocking and non-blocking:

- Send and Receive can be blocking or non-blocking.
- A blocking send can be used with a non-blocking receive and vice versa.
- All sends can use any mode:

	non-blocking	blocking
– Standard	MPI_Isend	MPI_Send
– Synchronous	MPI_Issend	MPI_Ssend
– Buffered	MPI_IbSend	MPI_Bsend
– Ready	MPI_IrSend	MPI_Rsend

- There are two receives: MPI_Recv, MPI_IRecv.

Other point-to-point communication

Other point-to-point communication:

- `MPI_SendRecv`: send and receive at once: less risk for deadlocks; more efficient. It can be `src.NE.dest`.
- `MPI_SendRecv_Replace`: send and receive at once, replacing the sent data with the received data.
- `MPI_Probe`: check message length before calling a receive operation.
- `MPI_IProbe`: check if a message is available.

Other point-to-point communication

Recommendations:

- Use the standard `MPI_Send`, `MPI_Recv` unless you have a good reason for using any of the other.
- Use `MPI_RecvSend` wherever possible.
- Good reason for exceptions can be:
 - Using `MPI_BSend`: sends to resolve deadlocks (`MPI_Isend` is more dangerous as it relies on available resources for MPI-internal buffers).
 - Non-blocking communication to overlap with file I/O.

Collective communication

Definition:

- Communication involving a group of processes.
- Called by all processes within a communicator.
- Wrappers for several point-to-point operations.
- Examples:
 - Barrier.
 - Broadcast.
 - Scatter, Gather.
 - Reduce.

Collective communication

Properties

- All processes in the communicator must call the same collective routine and must communicate.
- All processes must be able to start the collective routine.
- All collective operations are blocking.
- No message tags allowed.
- Receive buffers on all processors must have exactly the same size.

Collective communication

Barrier synchronization:

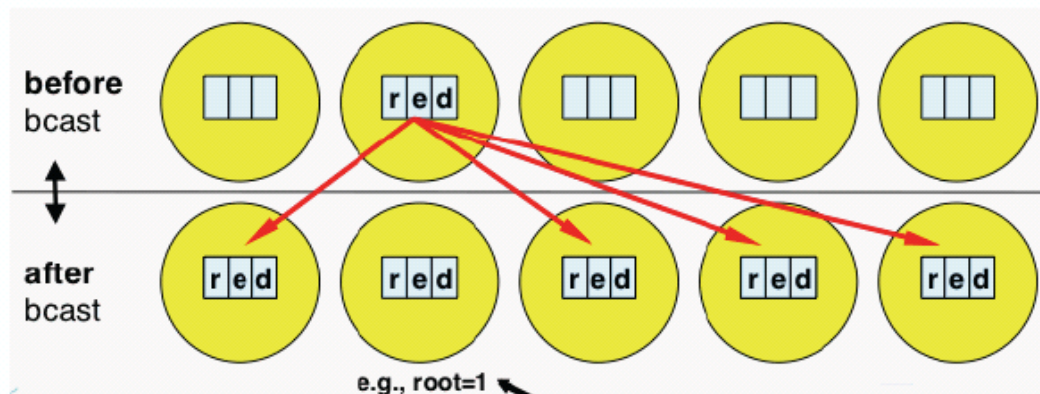
```
INTEGER :: info  
CALL MPI_Barrier(MPI_COMM_WORLD,info)
```

- Almost never used because synchronization is automatically ensured by data communication.
- Exceptions:
 - Debugging: 'make sure all make it this far!'
 - External I/O: wait for all processors to finish computations before writing a file.
 - Timing: before calls to MPI_Wtime when profiling the code.

Collective communication

Broadcast

```
INTEGER :: count,datatype,root,info  
<datatype>, DIMENSION(count) :: msg  
root = 1  
CALL MPI_BCast(msg,count,datatype,root, &  
MPI_COMM_WORLD,info)
```

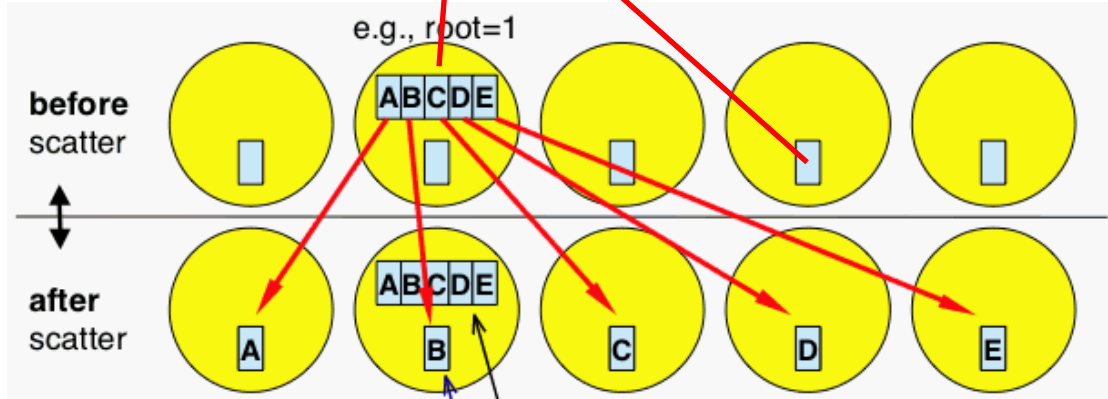


Rank of the sending process
Must be given identically on all processes!

Collective communication

Scatter:

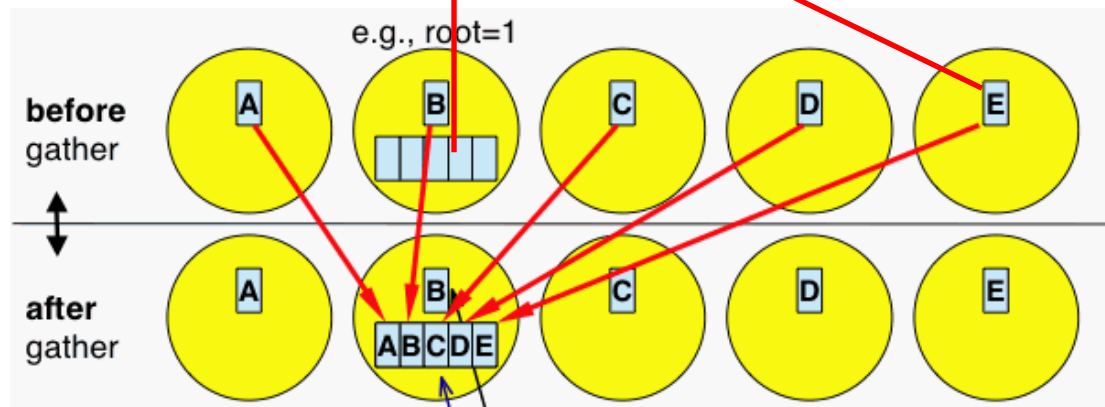
```
INTEGER :: sendcount, sendtype  
INTEGER :: rcvcount, rcvtype  
INTEGER :: root, info  
<sendtype>, DIMENSION(sendcount) :: sendmsg  
<rcvtype>, DIMENSION(rcvcount) :: rcvmsg  
CALL MPI_Scatter(sendmsg, sendcount, sendtype, &  
                rcvmsg, rcvcount, rcvtype, &  
                root, MPI_COMM_WORLD, info)
```



Collective communication

Gather:

```
INTEGER :: sendcount, sendtype  
INTEGER :: rcvcount, rcvtype  
INTEGER :: root, info  
<sendtype>, DIMENSION(sendcount) :: sendmsg  
<rcvtype>, DIMENSION(rcvcount) :: rcvmsg  
CALL MPI_Gather(sendmsg, sendcount, sendtype, &  
               rcvmsg, rcvcount, rcvtype, &  
               root, MPI_COMM_WORLD, info)
```



Collective communication

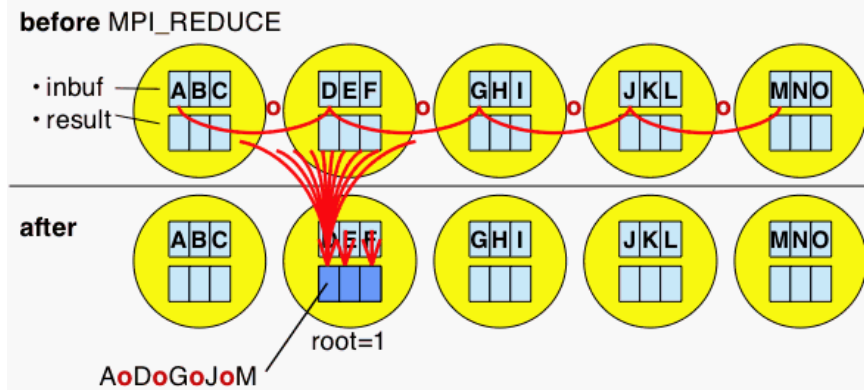
Global reduction operation:

- Perform global reduction operations across all processes within a communicator.
- $d_0 \circ d_1 \circ d_2 \circ \dots \circ d_{s-2} \circ d_{s-1}$
- d_i : data on process with rank i
- \circ : associative operation.
- Examples:
 - Sum or product.
 - Global maximum or minimum.

Collective communication

Reduce:

```
INTEGER :: count,datatype,operation
INTEGER :: root,info
<datatype>, DIMENSION(count) :: input
<datatype>, DIMENSION(count) :: output
CALL MPI_Reduce(input,output,count,datatype,&
               operation,root,MPI_COMM_WORLD,info)
```



Collective communication

MPI reduction operation types (handles):

Operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_LOR	Logical OR
MPI_BAND	Bitwise AND
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and its location
MPI_MINLOC	Minimum and its location

Collective communication

Variants of MPI_Reduce:

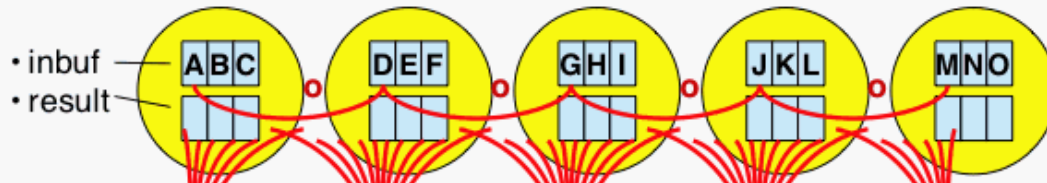
- MPI_AllReduce:
 - No root.
 - Result of reduction is returned to all processes.
- MPI_Reduce_Scatter:
 - Result vector of the reduction is directly scattered into the result buffers of the processes.
- MPI_Scan:
 - Prefix reduction.
 - The result of process i is the reduction of the values of all processes from rank 0 to rank i .

Collective communication

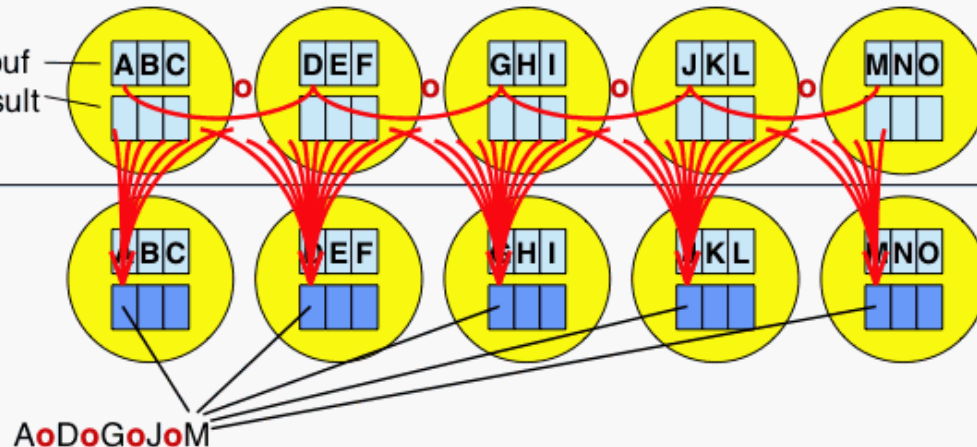
MPI_AllReduce:

```
INTEGER :: count,datatype,operation
INTEGER :: info
<datatype>, DIMENSION(count) :: input
<datatype>, DIMENSION(count) :: output
CALL MPI_AllReduce(input,output,count,datatype,&
    operation,MPI_COMM_WORLD,info)
```

before MPI_ALLREDUCE



after



Collective communication

Other collective communications:

- `MPI_AllGather`: gather data and send the gathered data to all processors.
- `MPI_AllToAll`: sends the data of all processors to all other processors.
- `MPI_Reduce_Scatter`: reduction operation followed by scattering the result vector among processes.
- `MPI_....v`: `Gatherv`, `ScatterV`, `AllToAllv` etc.. allow to explicitly specify the arrangement of the data in the result.