# Parallel Programming using MPI

## Point to Point Communications

Claudio Gheller, Andreas Jocksch
CSCS

cgheller@cscs.ch, jocksch@cscs.ch

# Agenda

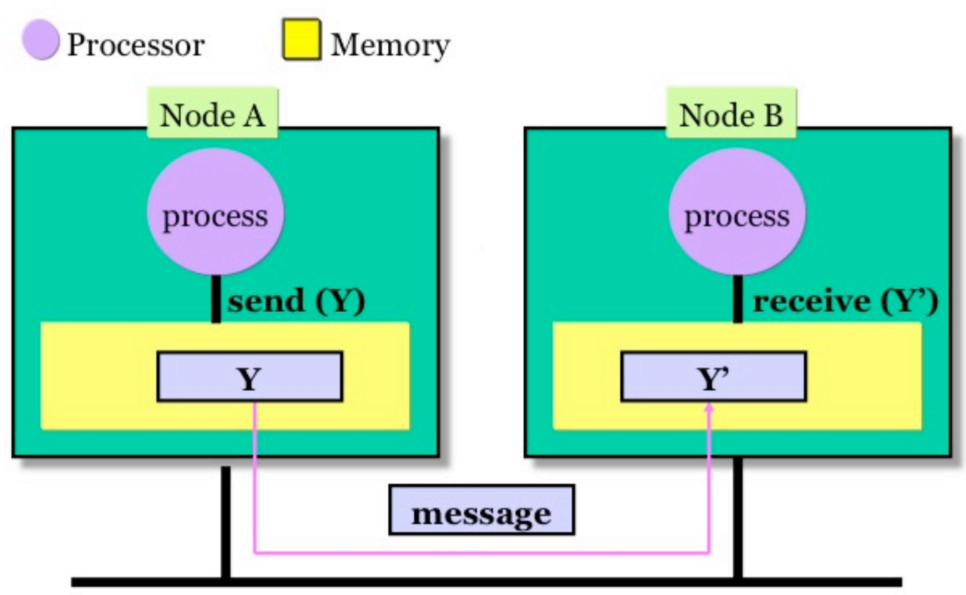
- Message Passing Programming Paradigm
- Introduction to MPI
- Point-to-point Communication
- Completion and Synchronization
- Collective Communication
- Communicators and Topologies
- Derived Data Types

# Message Passing Paradigm

- Resources are Local (differently from shared memory model)
- Each process runs in a "isolated" environment. Interactions requires Messages Exchange
- Messages can be: instructions, data, synchronization
- Message Passing works also in a Shared Memory system
- Time to exchange messages is much larger than accessing local memory

Message Passing is a COOPERATIVE approach, based on THREE basic operation

- SEND (a message)
- RECEIVE (a message)
- SYNCRONIZE



# Advantages and Drawbacks

- Advantages
  - Communications is the most important part of high-performance parallel computing: it can be highly optimized
  - Message-passing paradigm is portable
  - Many current applications/libraries use message-passing.
  - (Message passing can be used for distributed processing.)
- Drawbacks
  - Explicit nature of message-passing is error-prone. . .
  - and discourages frequent communications. . .
  - and hard to do MIMD programming. . .

# Message Passing Interface - MPI

- MPI is standard defined in a set of documents compiled by a consortium of organizations: http://www.mpi-forum.org/
- In particular the MPI documents define the APIs (application interfaces) for C, C++, FORTRAN77 and FORTRAN 90.
- Bindings available for Perl, Python, Java…
- The actual implementation of the standard is demanded to the software developers of the different systems
- In all systems MPI is implemented as a library of subroutines over the network drivers and primitives

# Goals of the MPI standard

MPI's prime goals are:

- To allow efficient implementation
- To provide source-code portability

MPI also offers:

- A great deal of functionality
- Support for heterogeneous parallel architectures

MPI2 further extends the library power (parallel I/O, Remote Memory Access, Multi Threads, Object Oriented programming)

# Basic Features of MPI Programs

- An MPI program consists of multiple instances of a serial program that communicate by library calls.
- Calls may be roughly divided into four classes:

1. Calls used to initialize, manage, and terminate communications
2. Calls used to communicate between pairs of processors. (point to point communication)
3. Calls used to communicate among groups of processors. (collective communication)
4. Calls to create data types.

# Header files

All Subprogram that contains calls to MPI subroutine must include the MPI header file

C:

```
#include <mpi.h>
```

Fortran:

```
include 'mpif.h'
```

Fortran 90:

```
USE MPI
```

The header file contains definitions of MPI constants, MPI types and functions

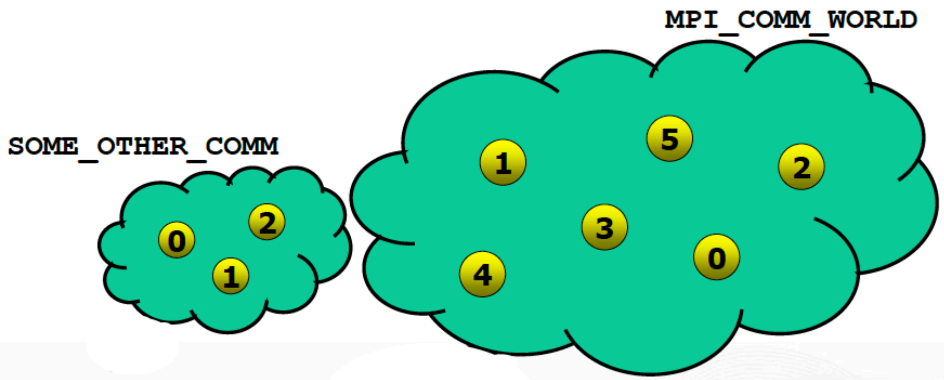
# MPI Communicator

Communicator objects connect groups of processes in the MPI session. Each communicator gives each contained process an identifier and arranges its contained processes in an ordered topology.

- The Communicator is identified by a variable (HANDLE) identifying a group of processes that are allowed to communicate with each other;
- The communicator that includes all processes is called: MPI_COMM_WORLD
- MPI_COMM_WORLD is the default communicator (automatically defined):

All MPI communication subroutines have a communicator argument.

Multiple communicators can be defined and used at the same time



# MPI functions appearance

C:

```
int error = MPI_Xxxxx(parameter,...);
MPI_Xxxxx(parameter,...);
```

Fortran:

```
CALL MPI_XXXXX(parameter, IERROR)
INTEGER IERROR
```

# Initializing MPI

C:

```
int MPI-Init(int*argc, char***argv)
```

Fortran:

```
MPI_INIT(IERROR)
INTEGER IERROR
```

This is the first MPI call: initializes the message passing environment

# Communicator Size

- How many processors are associated with a communicator?

- C:

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

- Fortran:

```
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
INTEGER COMM, SIZE, IERR
OUTPUT:  SIZE
```

# Process Rank

How can you identify different processes?
What is the ID of a processor in a group?

The MPI_COMM_RANK function is used to find the rank (the identifier) of a process in a given communicator

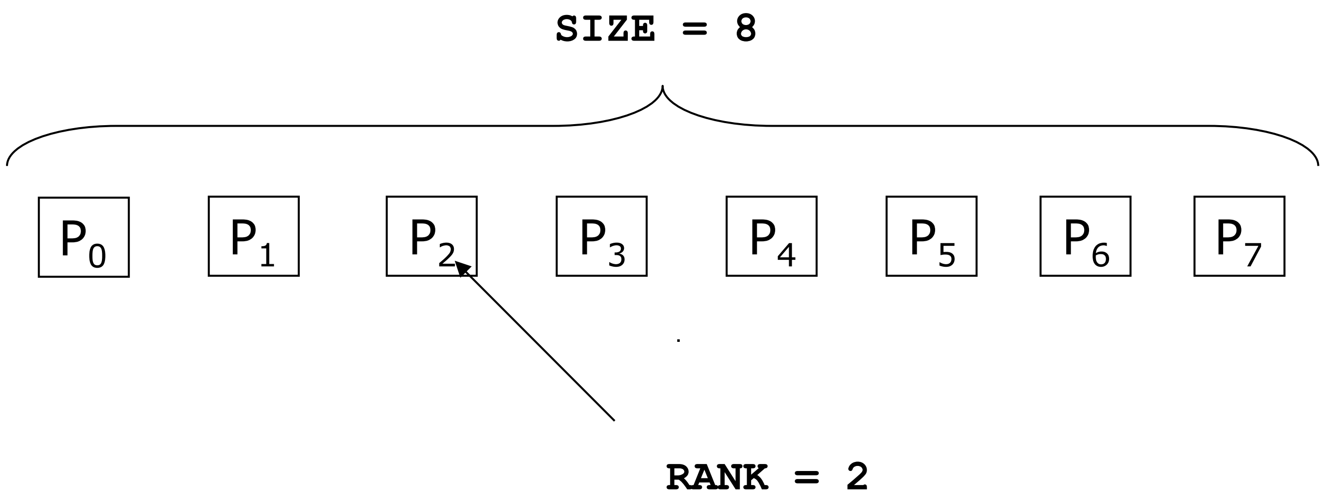
C:

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

Fortran:

```
CALL MPI_COMM_RANK(COMM, RANK, IERR)
INTEGER COMM, RANK, IERR
OUTPUT:  RANK
```

# Communicator Size and Process Rank




**`Size`** is the number of processors associated to the communicator

**`rank`** is the index of the process within a group associated to a communicator (**`rank`** = 0,1,...,N-1). The rank is used to identify the source and destination process in a communication

# Exiting MPI

Finalizing MPI environment

C:

```
int MPI_Finalize()
```

Fortran:

```
INTEGER IERR
CALL MPI_FINALIZE(IERR)
```

This two subprograms should be called by all processes, and no other MPI calls are allowed before **`mpi_init`** and after **`mpi_finalize`**. However the program can proceed serial.

# MPI_ABORT

- Usage
  - int MPI_Abort( MPI_Comm comm, /* in */
    int errorcode ); /* in */
- Description
  - Terminates all MPI processes associated with the communicator comm; in most systems (all to date), terminates *all* processes.

# Exercise 1

**Parallel Hello World**

**Initialize**

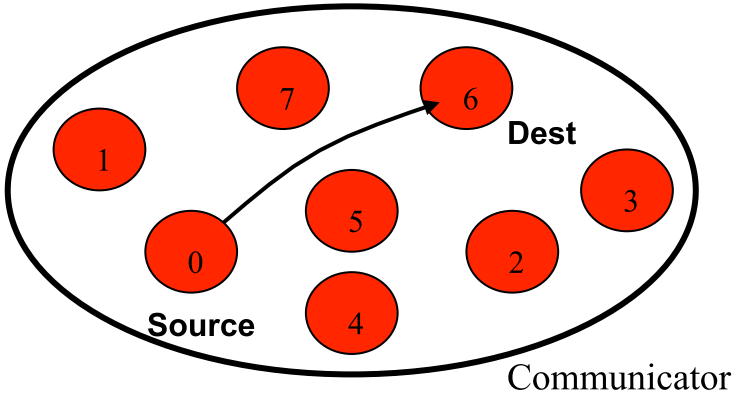
**Get rank**

**Get size**

**Print them**

# Point to Point Communication

- It is the fundamental communication facility provided by MPI library. Communication between 2 processes
- It is conceptually simple: source process A sends a **message** to destination process B, B receive the message from A.
- Communication take places within a communicator
- Source and Destination are identified by their rank in the communicator



# The Message

- Data is exchanged in the buffer, an array of count elements of some particular MPI data type
- One argument that usually must be given to MPI routines is the *type* of the data being passed.
- This allows MPI programs to run automatically in heterogeneous environments

Messages are identified by their envelopes. A message could be exchanged only if the sender and receiver specify the correct envelope

## Message Structure

| envelope | | | | body | | |
|---|---|---|---|---|---|---|
| source | destination | communicator | tag | buffer | count | datatype |

# Data Types

Programmer declares variables to have "standard" C/Fortran type, but uses MPI datatypes as arguments in MPI routines

- Basic types
- Derived types

• Handle type conversion in a heterogeneous collection of machines: implemented so that the MPI datatype is the same as the corresponding elementary datatype on the host machine (e.g. MPI_REAL is usually a four-byte floating-point number on a 32-bits system and it is an eight-byte floating-point number a 64-bits one).

• if one system sends 100 MPI_REALs to a system having a different architecture, the receiving system will still receive 100 MPI_REALs in its own format

• General rule: MPI datatype specified in a receive must match the MPI datatype specified in the send

MPI defines '*handles*' to allow programmers to refer to data types

# Fortran - MPI Intrinsic Datatypes

| MPI Data type | Fortran Data type |
|---|---|
| MPI_INTEGER | INTEGER |
| MPI_REAL | REAL |
| MPI_DOUBLE_PRECISION | DOUBLE PRECISION |
| MPI_COMPLEX | COMPLEX |
| MPI_DOUBLE_COMPLEX | DOUBLE COMPLEX |
| MPI_LOGICAL | LOGICAL |
| MPI_CHARACTER | CHARACTER(1) |
| MPI_PACKED | |
| MPI_BYTE | |

# C - MPI Intrinsic Datatypes

| MPI Data type | C Data type |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | Signed log 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 | |

# Standard Send and Receive

Basic point-to-point communication routine in MPI.

## Fortran:

```
MPI_SEND(buf, count, type, dest, tag, comm, ierr)
MPI_RECV(buf, count, type, dest, tag, comm, status, ierr)
```

$\underbrace{\texttt{buf, count, type}}_{\text{Message body}}$ , $\underbrace{\texttt{dest, tag, comm, status}}_{\text{Message envelope}}$

**buf** array of type **type** see table.

**count** (INTEGER) number of element of **buf** to be sent

**type** (INTEGER) MPI type of **buf**

**dest** (INTEGER) rank of the destination process

**tag** (INTEGER) number identifying the message

**comm** (INTEGER) communicator of the sender and receiver

**status** (INTEGER) array of size **MPI_STATUS_SIZE** containing communication status information (Orig Rank, Tag, Number of elements received)

**ierr** (INTEGER) error code (if **ierr=0** no error occurs)

# Standard Send and Receive

C:

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


int MPI_Recv (void *buf, int count, MPI_Datatype type,
    int dest, int tag, MPI_Comm comm, MPI_Status
    *status);
```

# MPI_status

MPI_Status structures are used by the message receiving functions to return data about a message. It is an INTEGER array of MPI_STATUS_SIZE elements in Fortran

The array contains the following info:

- MPI_SOURCE - id of processor sending the message
- MPI_TAG - the message tag
- MPI_ERROR - error status

There may also be other fields in the structure, but these are reserved for the implementation.

# Wildcards

Both in Fortran and C **MPI_RECV** accepts wildcards:

- To receive from any source: MPI_ANY_SOURCE
- To receive with any tag: MPI_ANY_TAG
- Actual source and tag are returned in the receiver's status parameter.

# Exercise 2

- Standard Send – recv communication
- Do with 2 processors
- What happens with more than 2 processors?

# Completion & Sycronization

In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is actually never the case.

Two important concepts:

- **Syncronization:** all tasks are at the same execution stage
- **Completion** of the communication: memory locations used in the message transfer can be safely accessed
  - Send: variable sent can be reused after completion
  - Receive: variable received can be used after completion

# Blocking communications

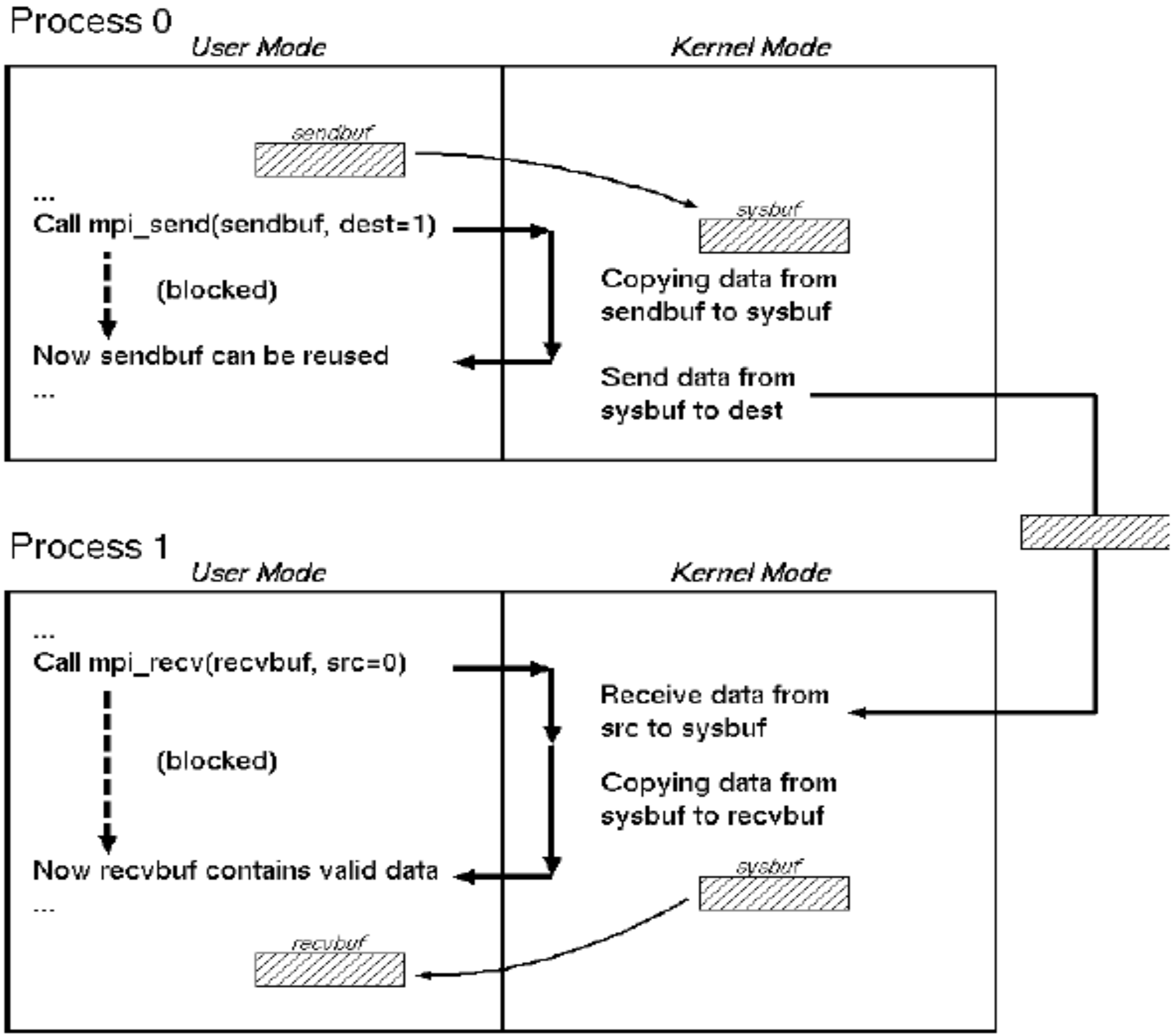
Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

**Blocking mode:**

- A blocking send returns after it is safe to modify the application buffer (your send data) for reuse. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.

- A blocking receive only "returns" after the data has arrived and is ready for use by the program.

# Blocking Communications

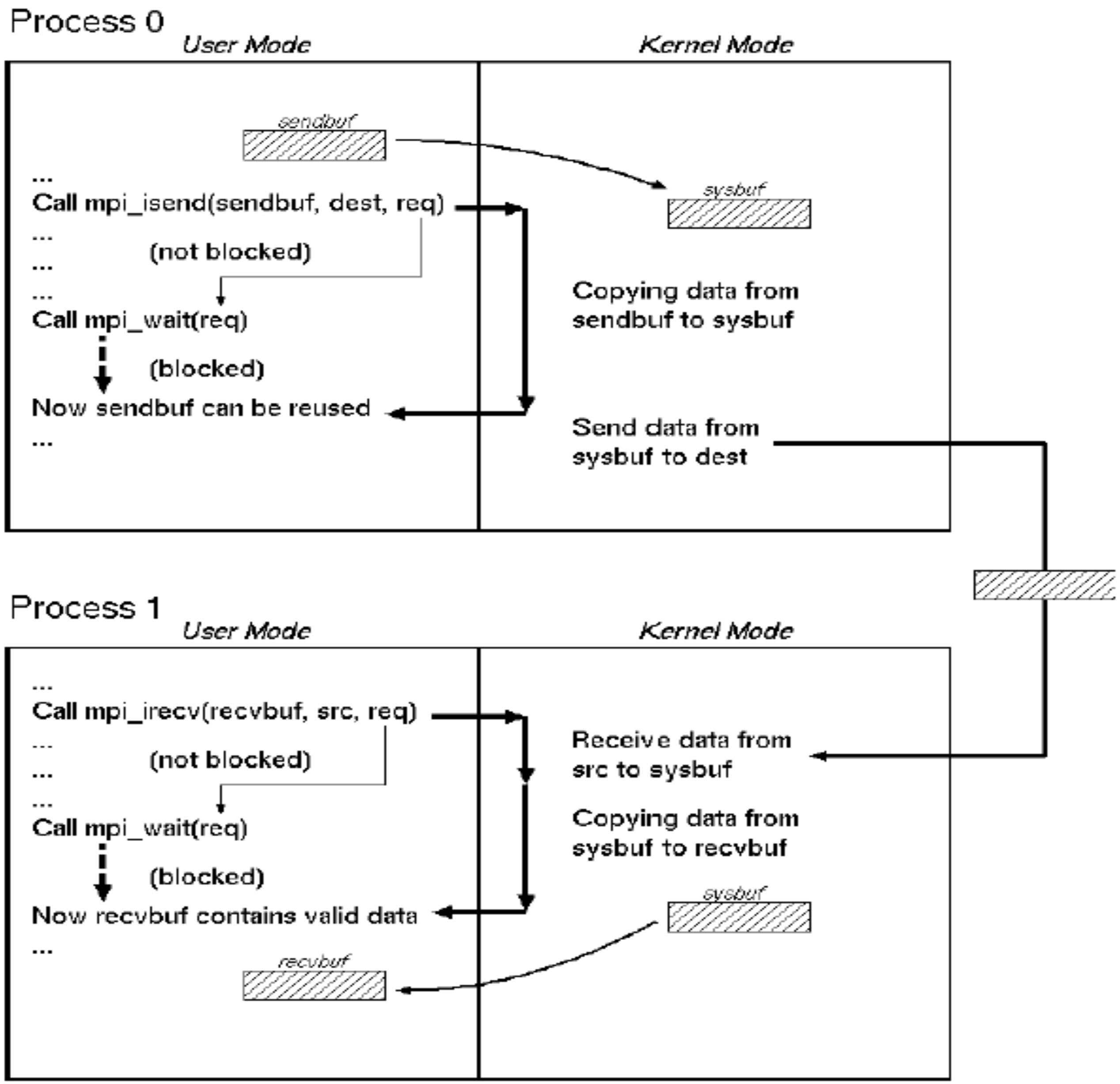


# Non Blocking communications

**Non-blocking mode:**

- Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete
- Non-blocking operations simply "request" the MPI library to perform the operation when it can. The user cannot predict when this will happen.
- It is unsafe to modify the application buffer until communication is actually completed. Completion is ensured by the wait functions.
- Non-blocking communications are primarily used to overlap computation with communication.

# Non-Blocking Communications



# Communication Modes

Five different communication modes are supported for the Send:

- Standard Mode
- Synchronous Mode
- Buffered Mode
- Ready Mode

Then we have the

- Receive

All of them can be Blocking or Non-Blocking

# Communication Modes in MPI

- Standard send
  - A send operation can be started whether or not a matching receive has started
  - Can be buffered or synchronous. It is up to implementation (and not MPI standard) to decide whether outgoing messages will be buffered
  - may complete before a matching receive is posted
  - Non-local operation (in general)

- Standard Receive
  - Receive a message. If blocking, blocks until the requested data is available in the application buffer of the receiving task.

# Communication Modes in MPI

- Synchronous send
  - A send operation can be started whether or not a matching receive has started
  - The send will complete successfully only if a matching receive was posted and the receive operation has reached a certain point in its execution
  - The completion of a synchronous send not only indicates that the send buffer can be reused but also indicates that the receiver has reached a certain point in its execution (usually it has received all data)
  - Non-local operation

# Communication Modes in MPI

- Buffered (asynchronous) mode
  - A send operation can be started whether or not a matching receive has been posted
  - It completes whether or not a matching receive has been posted (independent from the receive)
  - Buffer space is allocated on demand (overcoming default settings – MPI_Buffer_Attach function)
  - Local

# Communication Modes in MPI

- Ready send
  - A send operation may be started only if the matching receive is already started
  - The completion of the send operation does not depend on the status of a matching receive and merely indicates the send buffer can be reused
  - Used for performance reasons
  - Non-local

# Communication Modes

| Mode | Completion Condition | Blocking subroutine | Non-blocking subroutine |
|---|---|---|---|
| Standard send | Message sent (receive state unknown) | `MPI_SEND` | `MPI_ISEND` |
| receive | Completes when a matching message has arrived | `MPI_RECV` | `MPI_IRECV` |
| Synchronous send | Only completes after a matching recv() is posted and the receive operation is at some stages. | `MPI_SSEND` | `MPI_ISSEND` |
| Buffered send | Always completes, irrespective of receiver. Guarantees the message being buffered | `MPI_BSEND` | `MPI_IBSEND` |
| Ready send | Always completes, irrespective of whether the receive has completed | `MPI_RSEND` | `MPI_IRSEND` |

# Communication Modes: final remarks

- Blocking send and recv
  - Does not mean that the process is stopped during communication.
  - It means that, at return, it is safe to use the variables involved in communication.
- Non Blocking send and recv
  - Cannot use variables involved in communication until completion functions are called.
- Communication modes
  - Define the behaviour of the various function for point to point communication. The behaviour can be implementation dependent.

# Non-Blocking Send and Receive

Fortran:

```
MPI_ISEND(buf, count, type, dest, tag, comm, req, ierr)
MPI_IRECV(buf, count, type, dest, tag, comm, req, ierr)
```

| | | |
|---|---|---|
| **buf** | | array of type **type** see table. |
| **count** | (INTEGER) | number of element of **buf** to be sent |
| **type** | (INTEGER) | MPI type of **buf** |
| **dest** | (INTEGER) | rank of the destination process |
| **tag** | (INTEGER) | number identifying the message |
| **comm** | (INTEGER) | communicator of the sender and receiver |
| **req** | (INTEGER) | output, identifier of the communications handle |
| **ierr** | (INTEGER) | output, error code (if **ierr=0** no error occurs) |

# Non-Blocking Send and Receive

C:

```
int MPI_Isend(void *buf, int count, MPI_Datatype
   type, int dest, int tag, MPI_Comm comm,
   MPI_Request *req);

int MPI_Irecv (void *buf, int count, MPI_Datatype
   type, int dest, int tag, MPI_Comm comm,
   MPI_Request *req);
```

# Waiting for Completion

## Fortran:

**MPI_WAIT(req, status, ierr)**

**MPI_WAITALL (count,array_of_requests,array_of_statuses, ierr)**

A call to this subroutine cause the code to wait until the communication pointed by req is complete.

**req** (INTEGER) :input/output, identifier associated to a communications event (initiated by **MPI_ISEND** or **MPI_IRECV**).

**Status** (INTEGER) array of size **MPI_STATUS_SIZE**, if **req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

**ierr** (INTEGER) output, error code (if **ierr=0** no error occours).

## C:

**int MPI_Wait(MPI_Request *req, MPI_Status *status)**

**Int MPI_Waitall (count,&array_of_requests,&array_of_statuses)**

# Testing Completion

## Fortran:

```
MPI_TEST(req, flag, status, ierr)
MPI_TESTALL (count,array_of_requests,flag,array_of_statuses,ierr)
```

A call to this subroutine sets **flag** to **.true.** if the communication pointed by **req** is complete, sets **flag** to **.false.** otherwise.

**Req** (INTEGER) input/output, identifier associated to a communications event (initiated by **MPI_ISEND** or **MPI_IRECV**).

**Flag** (LOGICAL) output, **.true.** if communication **req** has completed **.false.** otherwise

**Status** (INTEGER) array of size **MPI_STATUS_SIZE**, if **req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

**Ierr** (INTEGER) output, error code (if **ierr=0** no error occours).

## C:

```
int MPI_Test (&request,&flag,&status)
Int MPI_Testall (count,&array_of_requests,&flag,&array_of_statuses)
```

# SendRecv

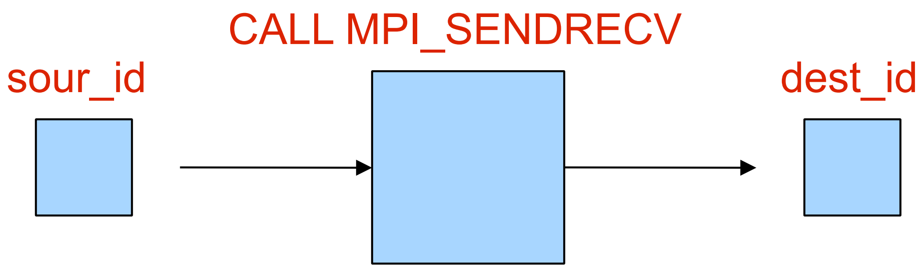
The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes.

Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.

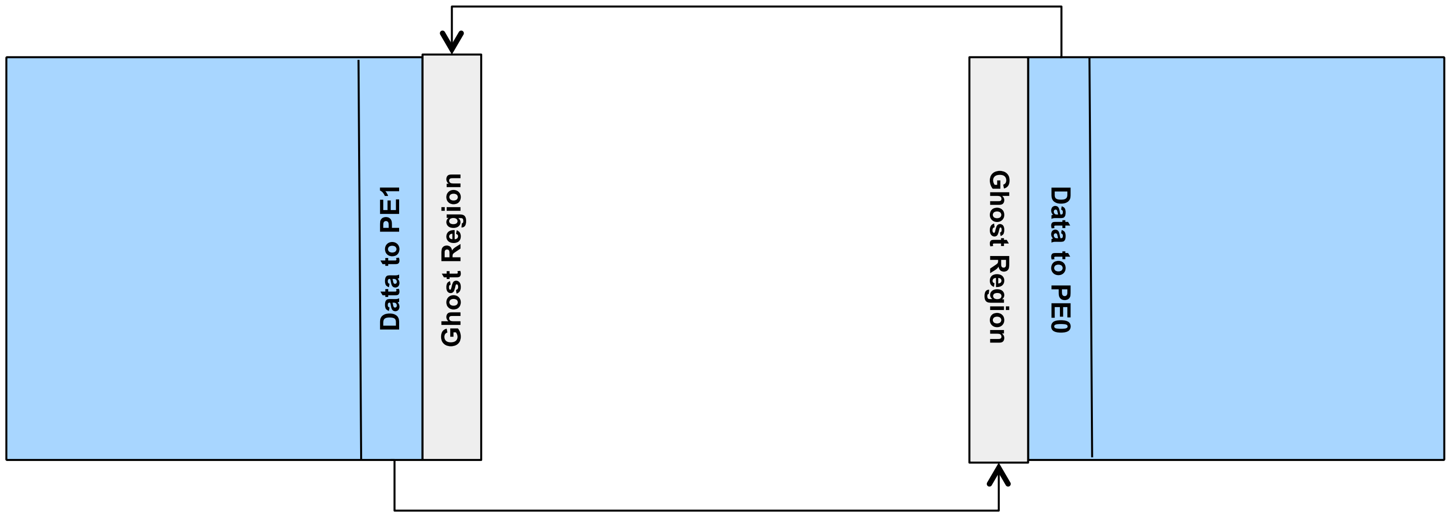
**Sender side**

```
CALL MPI_SENDRECV(sndbuf, snd_size, snd_type, destid, tag,
rcvbuf, rcv_size, rcv_type, sourid, tag, comm, status, ierr)
```

**Receiver side**



# SendRecv (cont.ed)




PE0
Send(data to PE1, 1)
Recv(data to PE0, 1)

PE1
Send(data to PE0, 0)
Recv(data to PE1, 0)

DEADLOCK

PE0 send recv
SendRecv(data to PE1, 1, data to PE0, 1)

PE1 send recv
SendRecv(data to PE0, 0, data to PE1, 0)

# Exercise 3

- Send-recv → deadlock
- Isend-irecv
- sendrecv