



**CSCS**

Centro Svizzero di Calcolo Scientifico  
Swiss National Supercomputing Centre

**ETH** zürich



# Message Passing Interface (MPI)

Summer School 2017 – Effective High Performance Computing

Tim Robinson, CSCS

July 19–20, 2017

# Previous course summary

- MPI, message passing paradigm
- Distributed memory (but also shared memory)
- Write a simple program, no communication

# Course Objectives

- The understanding of a point-to-point communications
- The understanding of their different flavours
- The importance of buffer availability and scope

# General Course Structure



- An introduction to MPI
- Point-to-point communications
- Collective communications
- Topology
- Datatypes

# General Course Structure



- An introduction to MPI
- Point-to-point communications
  - Communication
  - Message
  - Data types
  - Standard Send/Recv
  - Communication status
  - Wildcards
  - Synchronization
  - Blocking and non-blocking
  - Transfer modes
  - Summary
- Collective communications
- Topology
- Datatypes



**CSCS**

Centro Svizzero di Calcolo Scientifico  
Swiss National Supercomputing Centre

**ETH** zürich

# Point-to-point communications

---

# Point-to-Point communication

It is the fundamental communication facility provided by a MPI library.  
Communication between 2 processes:

- It is conceptually simple: source process A sends a message to destination process B, B receives the message from A.
- Communication takes place within a communicator
- Source and Destination are identified by their rank in the communicator

# Message

Data is exchanged in the buffer, an array of count elements of some particular MPI data type

- One argument that 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.

body			envelope			
buffer	count	datatype	source	destination	communicator	tag



# Message ordering

Each process has a FIFO receipt (queue), incoming messages never overtake each other.

If process A does multiple sends to process B those messages arrive in the same order.

# Data types

MPI provides its own data type for send and recv buffers.

- Handle type conversion in a heterogeneous collection of machines

## General rule

MPI datatype must match datatypes among pairs of send and recv

MPI defines "handles" to allow programmers to refer to data types.

## Some - MPI Intrinsic Datatypes

MPI Data type	Fortran/C Data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double

# Basic Send/Recv

## Pseudo-code

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

<b>buf</b>	array of type type see table
<b>count</b>	number of elements to be sent
<b>type</b>	MPI type of buf
<b>dest</b>	rank of the destination process
<b>source</b>	rank of the source process
<b>tag</b>	number identifying the message
<b>comm</b>	communicator of the sender and receiver
<b>status</b>	array containing communication status

# Basic Send/Recv

## Pseudo-code

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

<b>buf</b>	
<b>count</b>	BODY
<b>type</b>	
<hr/>	
<b>source/dest</b>	
<b>tag</b>	
<b>comm</b>	ENVELOPE
<b>status</b>	
<hr/>	

# MPI Status and tag

`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.

A tag is a user-defined identity for a message.

# 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

# Synchronization

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is actually never the case. A send resp. a receive can be triggered before or after its corresponding receive resp. send.
- Is the receiving rank prepared to receive the data?...

## Common MPI protocol

Internal to MPI not expose to the user, not defined by the standard

Pseudo-code Pseudo-code

```
if (size_to_send < CST ) {  
    /* EAGER protocol:  
    post a send request using a buffer  
    expecting the receive to store it */  
} else {  
    /* RENDEZ-VOUS protocol:  
    post an envelope to ask the receive to be prepared  
    wait for the receive to acknowledge it  
    then the receive is able to store the data  
    post a send request using a buffer */  
}
```



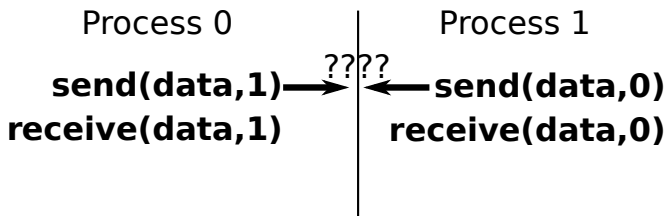
# Practicals

## Exercise: 02.MPI\_pt2pt

1. Standard Send/Recv communication
2. Ping-pong
3. Measuring bandwidth

# Blocking and non-blocking ?

Without non-blocking communications there is potential deadlocks



Also non-blocking communication are useful to overlap communication and computation

# Blocking and non-blocking communications

A blocking communication means. . .

That the buffer can be re-used after the function is returned.

A non-blocking communication means. . .

That the buffer can only be re-used after a wait function is returned.  
Until then, the buffer must not be overwritten (and even read).

**Blocking** and non-blocking do not mean. . .

That the communication is completed after the function as returned.

# Blocking and non-blocking communications

A blocking communication means...

That the buffer can be re-used after the function is returned.

A non-blocking communication means...

That the buffer can only be re-used after a wait function is returned.  
Until then, the buffer must not be overwritten (and even read).

**Blocking** and non-blocking do not mean...

That the communication is completed after the function as returned.

⇒ **It is all about buffer availability!**

# Blocking and non-blocking examples

## Blocking example:

Pseudo-code

```
int buf[120];
MPI_Send(buf, MPI_INT, 5, ...);
// after the function returns
// the communication is done
// the buffer can be re-used
buf[3]=5
```

## Non-blocking example:

Pseudo-code

```
int buf[120];
MPI_Isend(buf, MPI_INT, 5, ...);
// after the function returns
// the communication is not done
// do not overwrite buf
// buf[3]=5 illegal
MPI_Wait(...);
// here communication is done
buf[3]=5
```

Each rank does not necessary know if the communication has been completed by the time the MPI send function returns.

The **I** in **MPI\_Isend** means **immediate**, the function returns "immediately" after its call.

# Non-blocking MPI\_Isend and MPI\_Irecv

## Pseudo-code

```
MPI_Isend(buf, count, type, dest, tag, comm, MPI_Request req)  
MPI_Irecv(buf, count, type, source, tag, comm, MPI_Request req)
```

<b>buf</b>	array of type type see table
<b>count</b>	number of element of buf to be sent
<b>type</b>	MPI type of buf
<b>dest</b>	rank of the destination process
<b>source</b>	rank of the source process
<b>tag</b>	number identifying the message
<b>comm</b>	communicator of the sender and receiver
<b>req</b>	identifier of the communications handle

# Waiting for completion

## Pseudo-code

```
MPI_Wait(MPI_Request req, status)
```

```
MPI_Waitall(count, MPI_Request requests[], status[])
```

<b>req</b>	identifier of the communications handle
<b>status</b>	array containing communication status
<b>count</b>	number of element of in arrays

# Testing for completion

## Pseudo-code

```
MPI_Test(req, flag, status)
```

```
MPI_Testall(count, requests[], flag, status[])
```

<b>req</b>	identifier of the communications handle
<b>flag</b>	true if req has completed false otherwise
<b>status</b>	array containing communication status
<b>count</b>	number of element of in arrays



# Transfer modes

Four different communication modes are supported for the Send:

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

All of them can be Blocking or Non-Blocking.

# Transfer modes: Standard Mode

- A send operation can be started whether or not a matching receive has started
- Can be buffered or synchronous. It is up to the 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)

# Transfer modes: Synchronous Mode

- 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

# Transfer modes: Buffered 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)
- The original buffer can be read and overwritten, it has been copied to user-supplied buffer
- Buffer space is allocated on demand `MPI_Buffer_Attach`
- Local operation

# Transfer modes: Ready Mode

- A send operation may be started only if the matching receive is already started (error otherwise)
- 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 operation

# Transfer modes overview

Mode	Completion Condition	Blocking	Non-blocking
Standard send	Message sent (receive state unknown)	<code>MPI_Send</code>	<code>MPI_Isend</code>
Receive	Completes when a matching message has arrived	<code>MPI_Recv</code>	<code>MPI_Irecv</code>
Synchronous send	Only completes after a matching <code>recv()</code> is posted and the receive operation is at some stages.	<code>MPI_Ssend</code>	<code>MPI_Issend</code>
Buffered send	Always completes, irrespective of receiver. Guarantees the message to be buffered.	<code>MPI_Bsend</code>	<code>MPI_Ibsend</code>
Ready send	Always completes, irrespective of whether the receive has completed.	<code>MPI_Rsend</code>	<code>MPI_Irsend</code>

# Communication - summary

## 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 "wait" completion functions are called.

## Transfer modes

- Define the behaviour of the various function for point to point communication. The behaviour can be implementation dependent.

# Combined Send and Recv in one function

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 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.

Pseudo-code

```
MPI_Sendrecv(sndbuf, snd_size, snd_type, rcvid, tag,  
             rcvbuf, rcv_size, rcv_type, sndid, tag,  
             comm, status)
```



## Other functions

- Check for incoming message without actually receiving it:

`MPI_Probe, MPI_Iprobe`

- Buffer management:

`MPI_Buffer_attach, MPI_Buffer_detach`

- Wait functions:

`MPI_Waitany, MPI_Waitsome`

- Test functions:

`MPI_Testany, MPI_Testsome`

- Cancel a request:

`MPI_Cancel`

# Practicals

## Exercise: 02.MPI\_pt2pt

### 4. Parallel sum using a ring:

without using `if (rank == 0)... else ...`

	Rank 0	Rank 1	Rank 2
Send	0	1	2
Recv	2	0	1
Send	2	0	1
Recv	1	2	0
Send	1	2	0
Recv	0	1	2

All ranks obtain the sum.

### 5. Send and Recv neighbor ghost cells:

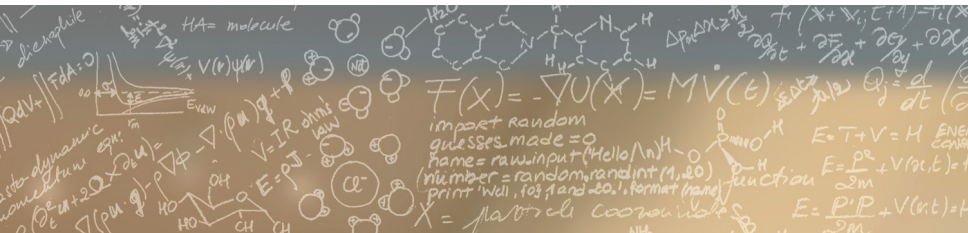
Top to bottom for C and right to left for Fortran



**CSCS**

Centro Svizzero di Calcolo Scientifico  
Swiss National Supercomputing Centre

**ETH** zürich



**Thank you for your attention.**