

# MPI Communication Point-to-Point (P2P)

Luca Tornatore - I.N.A.F.



DATA SCIENCE &  
ARTIFICIAL INTELLIGENCE



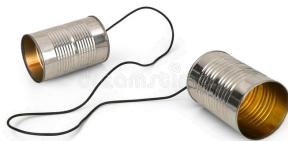
SCIENTIFIC &  
DATA-INTENSIVE COMPUTING

**2024-2025 @ Università di Trieste**

# Outline



Intro to  
MPI



Point-to-Point  
Communications



Collective  
Communications



few JEDI things

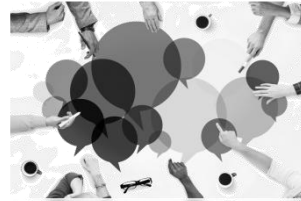
# Outline



Intro to  
MPI



Point-to-Point  
Communications



Collective  
Communications



few JEDI things



# Building blocks: Send and Receive

The very basic building block of a message-passing interface is, obviously, the capability of sending and receiving messages between two tasks:

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

The begin of the memory region to be sent

The size of the message

The rank of the receiver

Tag your messages to distinguish them

Specify the communicator



# Building blocks: Send and Receive

The very basic building block of a message-passing interface is, obviously, the capability of sending and receiving messages between two tasks:

```
int MPI_Recv( const void *,  
              int count,  
              MPI_Datatype, }  
              int source,   ← The rank of the receiver  
              int tag,      ← Tag your messages to distinguish them  
              MPI_Comm comm, ← Specify the communicator  
              MPI_Status &status)
```

← The begin of the memory region to be sent

← The **maximum** size of the message

← Return details about the message



# NOTICE

In the following slides we will describe many MPI calls, in their fundamental traits.

Please, refer to the man pages for a thorough description and for all the possibile output values in specific cases





# Building blocks: Send and Receive

```
int MPI_Send( const void *buf,    int MPI_Recv( const void *buf,
            int count,                int count,
            MPI_Datatype,              MPI_Datatype,
            int dest,                  int dest,
            int tag,                   int tag,
            MPI_Comm comm )            MPI_Comm comm,
                                      MPI_Status &status)
```

Then, the messages consists of a **body**

**buf**  
**count**  
**datatype**

and of an **envelop**

**dest**  
**tag**  
**comm**



# Building blocks: Send and Receive

## **MPI DataType**

MPI\_CHAR

MPI\_BYTE

MPI\_(UNSIGNED)\_SHORT

MPI\_(UNSIGNED)\_INT

MPI\_(UNSIGNED)\_LONG

MPI\_(UNSIGNED)\_LONG\_LONG

MPI\_FLOAT

MPI\_DOUBLE

MPI\_LONG\_DOUBLE

MPI\_PACKED

## **C DataType**

char

unsigned char

(unsigned) short int

(unsigned) int

(unsigned) long int

(unsigned) long long int

float

double

long double





# Building blocks: Send and Receive

## example

```
int N;  
if ( Myrank == 0 )  
    MPI_Send( &N, 1, MPI_INT, 1, 0, MPI_COMM_WORLD );  
  
else if ( Myrank == 1 ) {  
    MPI_Status status;  
    MPI_Recv( &N, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status ); }  

```

NOTE: **MPI\_STATUS\_IGNORE** is always valid instead of putting an MPI\_status variable's address as last argument of MPI\_Recv

```
else if ( Myrank == 1 )  
    MPI_Recv( &N, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
```



# Building blocks: Send and Receive

## example

How can I send more “things” of different type?

```
typedef struct {  
    int    i, j;  
    double d, f;  
    char    s[4]; } my_data;  
  
unsigned int length = N*sizeof(my_data);  
if ( Myrank == 0 )  
    MPI_Send( data, length, MPI_BYTE, 1, 0, MPI_COMM_WORLD );  
  
else if ( Myrank == 1 ) {  
    MPI_Recv( data, length, MPI_BYTE, 0, 0, MPI_COMM_WORLD,  
             MPI_STATUS_IGNORE );  
}
```



# Building blocks: Send and Receive

What are the tag useful for ?

It makes easier to distinguish between two similar messages and avoid errors

```
if ( Myrank == 0 ) {  
    MPI_Send( myname, 100, MPI_BYTE, 1, 0, MPI_COMM_WORLD );  
    MPI_Send( mysurnasme, 100, MPI_BYTE, 1, 1, MPI_COMM_WORLD ); }  
  
else if ( Myrank == 1 ) {  
    MPI_Recv( myname, 100, MPI_BYTE, 0, 0, MPI_COMM_WORLD,  
                                                     MPI_STATUS_IGNORE );  
    MPI_Recv( mysurname, 100, MPI_BYTE, 0, 1, MPI_COMM_WORLD,  
                                                     MPI_STATUS_IGNORE ); }
```

Note: We'll see more on how the messages are ordered



# Building blocks: Send and Receive

## Why the MPI\_Status in MPI\_Recv ?

- The `MPI_Recv`'s argument count provides the *maximum* number of elements that the call **expects to receive**. If the message exceeds that count, an error is thrown. Hence, you do not know whether `MPI_Recv` has got  $n \leq \text{count}$  elements.
- Valid values for the source and tag arguments are `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, so that the task could receive messages from anybody and with any tag.
- However, it may be that, once received, you need to know who was the sender, which was the tag and what is the size of the received message.  
You recover these data with

```
status.TAG, status.MPI_SOURCE
```

and by calling

```
MPI_Get_count ( &status, MPI_type_used, &count )
```

where you must use the same type than in the `Recv`, and you get how many data in `count`



# Building blocks: Send and Receive

## How to get the size of the received message with `MPI_Get_count`

Getting the size of the received message

**The `MPI_Recv`'s argument `count` provides the *maximum* number of elements that the call expects to receive.** If the message exceeds that count, an error is thrown.

Hence, in general you do not know whether `MPI_Recv` has got  $n \leq \text{count}$  elements.

```
if ( Me == 0 )
{
    MPI_Send( data, Ndata, MPI_BYTE, 1, TAG_DATA, MPI_COMM_WORLD );
    printf("Task %d has sent %d data: %d %d, ... %d!\n",
        Me, Ndata, data[0], data[1], data[Ndata-1] );
}
else
{
    MPI_Status status;
    int Nrecv;

    MPI_Recv( data, Ndata, MPI_BYTE, 0, TAG_DATA, MPI_COMM_WORLD, &status );

    MPI_Get_count ( &status, MPI_BYTE, &Nrecv );
    printf("Task %d has received %d data: %d %d, ... %d!\n",
        Me, Nrecv, data[0], data[1], data[Nrecv-1] );
}
```

get\_count.c



# Building blocks: Send and Receive

## How to check whether a message is arriving

You may want to know whether a message is arriving, from who and how large *before to actually get it*.

```
MPI_Status status;

// Probe for an incoming message from whatever process and whatever tag
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

source = status.MPI_SOURCE;
tag     = status.MPI_TAG;
MPI_Get_count(&status, MPI_BYTE, &number_amount);
char *buffer = (char*)malloc( number_amount );
MPI_Recv( buffer, number_amount, MPI_BYTE, source, tag, MPI_COMM_WORLD, &status)
...
// Probe for an incoming message from process zero with a precise tag

MPI_Probe(0, really_this_tag, MPI_COMM_WORLD, &status);
...
```

probe.c

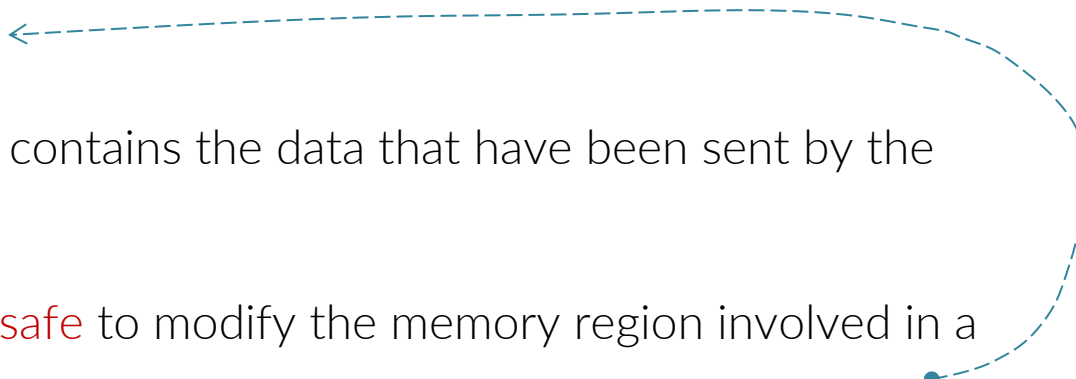


# Building blocks: Send and Receive

## Safety first

Let's consider `probe.c`

```
MPI_Send( data, N, MPI_INT, 1, TAG_DATA, MPI_COMM_WORLD );  
...  
modify( data );
```



`data` is the memory region that contains the data that have been sent by the `MPI_Send`.

At what point in the future it is **safe** to modify the memory region involved in a `MPI_Send` ?

In other words, **how can we be sure that the communication has ended and the data have all been received?**





# Building blocks: Send and Receive

Safety first

**how can we be sure that the communication has ended and the data have all been received?**

The MPI standard prescribes that **`MPI_Send`** returns when it is safe to modify the send buffer.

So, whenever `MPI_Send` returns, it is safe to act on the memory region that has been sent.

However, this leads us to a more general discussion.

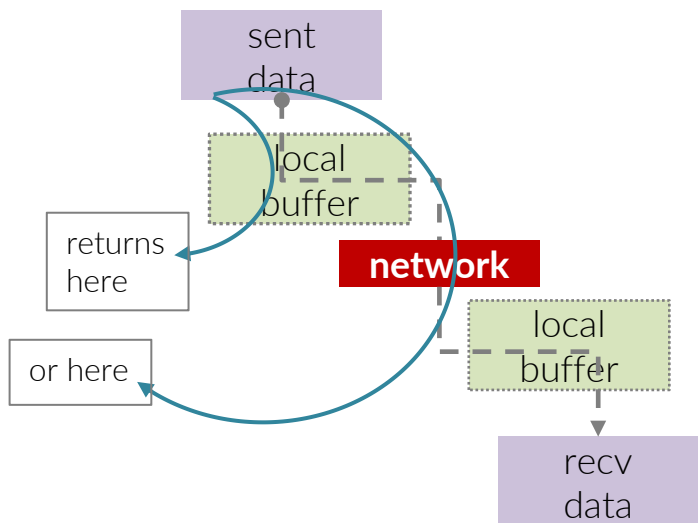


# MPI Communication protocols

## Eager protocol

Process 0

Process 1



MPI implementation may apply different communication protocols, depending on the message size.

In the eager protocol, `MPI_send` returns before that the data actually reach the destination. More correctly, it returns without *knowing* whether that happened already or not.

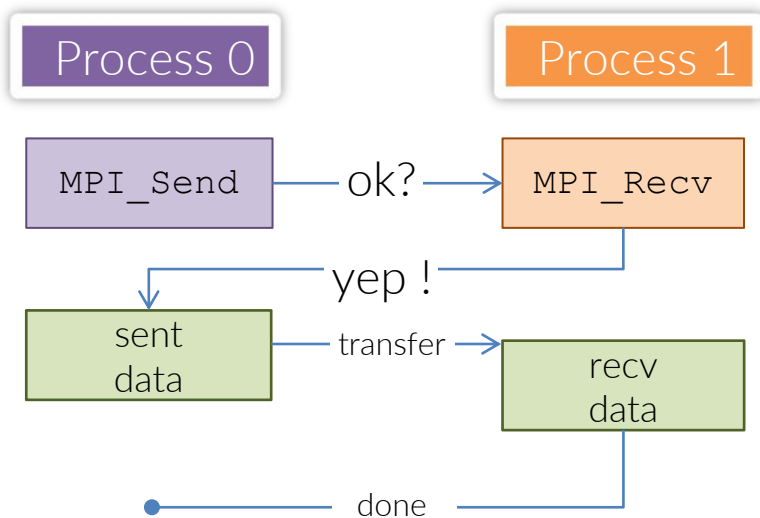
The MPI library copies the sent data on local buffer, either on the sender side or the receiver side. The actual transfer will complete afterwards and the `MPI_send` returns.

This protocol is normally applied for **small sizes**



# MPI Communication protocols

## Rendezvous protocol



In the rendezvous protocol, the sender first asks the agreement to the receiver.

Once it gets the acknowledgment, the data transfer starts.

At the end of it, the `MPI_Send` (and the `MPI_Recv`) returns.

This protocol is normally applied for large messages, for which the bufferization would require too much memory.



# Building blocks: Send and Receive

## Safety first

As a matter of fact, then, `MPI_Recv` does not complete until the buffer is full and `MPI_Send` does not complete until the buffer is empty.

As such, the completion of `MPI_Send`/`MPI_recv` depends on the size of the message and the size of the buffer provided by MPI.

This in general may lead to potentially dangerous situations, such as **deadlocks** or **unsafe code**, which appears to run smoothly just because of the system's bufferization.



# Unsafe code

```
if ( Me == 0 )
{
    MPI_Send( data_to_send, N, MPI_BYTE, !Me, DATA_FIRST, MPI_COMM_WORLD );
    MPI_Recv( data_to_recv, N, MPI_BYTE, !Me, DATA_SECND, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}
else
{
    MPI_Send( data_to_send, N, MPI_BYTE, !Me, DATA_FIRST, MPI_COMM_WORLD );
    MPI_Recv( data_to_recv, N, MPI_BYTE, !Me, DATA_SECND, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}
```

This exchange may run smoothly due to system's bufferization.

However, as the message size grows, it will incur in a deadlock.

```
char myname[10], myname2[10];

if ( Me == 0 )
{
    MPI_Send( myname, 100, MPI_BYTE, 1, NAME, MPI_COMM_WORLD );
    MPI_Send( myname2, 100, MPI_BYTE, SURNAME, 0, MPI_COMM_WORLD );
}
else
{
    MPI_Recv( myname2, 100, MPI_BYTE, 0, SURNAME, MPI_COMM_WORLD, MPI_STATUS_IGNORE );

    MPI_Recv( myname, 100, MPI_BYTE, 0, NAME, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}
```

This exchange may run smoothly due to system's bufferization.

However, using MPI\_Ssend instead of Send will surely result in a deadlock

( ... did you noticed that the Recv of the two messages are in opposite order than the Send ? )

potential\_deadlock.c



# Unsafe code

Code that rely on the system's bufferization to run correctly are called *unsafe*

Solutions to cure or, better, to avoid the unsafe situations are:

- designing more carefully the communication pattern
- checking the runnability by substituting `MPI_Send` with `MPI_Ssend`
- using `MPI_Sendrecv`
- supplying explicitly buffer with `MPI_Bsend`
- non-blocking operations



# Unsafe code & Deadlocks

The order of the Send / Recv must be crafted so that a Send is matched by Recv, with the same **source, tags and communicator**.

The mismatches lead to unsafe code or deadlocks, because the function will not return and the code hangs.

Due to system's bufferization in the *eager* protocol, some of these patterns may be run apparently smoothly, while hanging for larger data size or on different systems.

**Code that rely on the system's bufferization to run correctly are called *unsafe*.**



*a communication is made of the message data plus the envelop which assemble all the information needed to convey the message to the recipient.*





# Unsafe code & Deadlocks

deadlock

```
if (Rank == 0) {  
    Recv ( from 1 );  
    Send ( to 1 ); }  
if (Rank == 1) {  
    Recv ( from 0 );  
    Send ( to 0 ); }
```

```
if (Rank == 0) {  
    Recv ( from 1 );  
    Send ( to 2 ); }  
else if (Rank == 1) {  
    Recv ( from 2 );  
    Send ( to 1 ); }  
else if (Rank == 2 ) {  
    Recv ( from 0 );  
    Send ( to 0 ); }
```

unsafe

```
if (Rank == 0) {  
    Send ( from 1 );  
    Recv ( from 1 ); }  
if (Rank == 1) {  
    Send ( to 0 );  
    Recv ( from 0 ); }
```

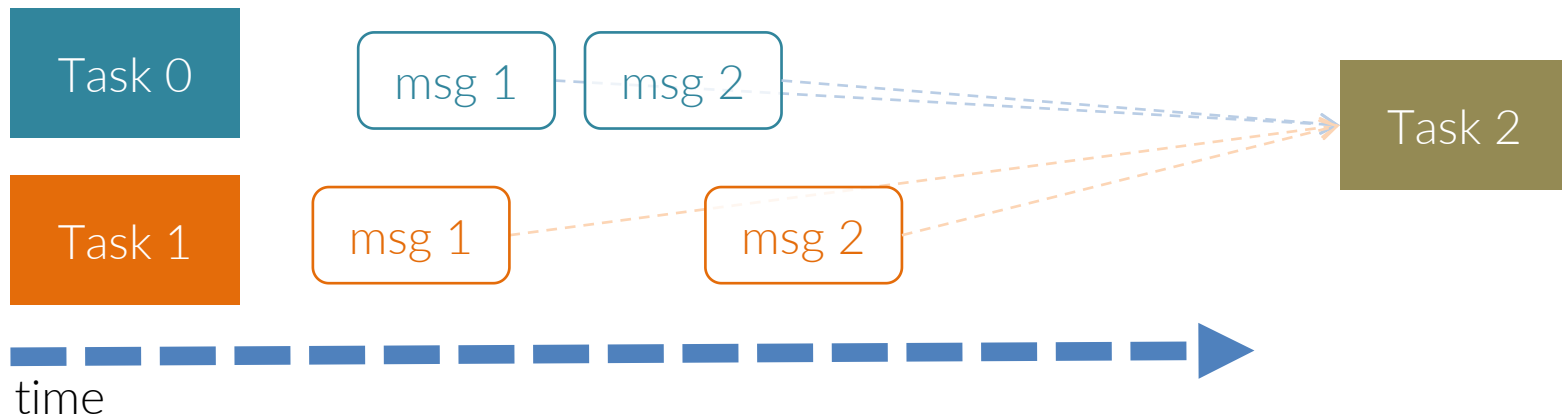
safe

```
if (Rank == 0) {  
    Recv ( from 1 );  
    Send ( to 1 ); }  
if (Rank == 1) {  
    Send ( to 0 );  
    Recv ( from 0 ); }
```



# Ordering of communications

MPI does not ensure any ordering of messages from different sources

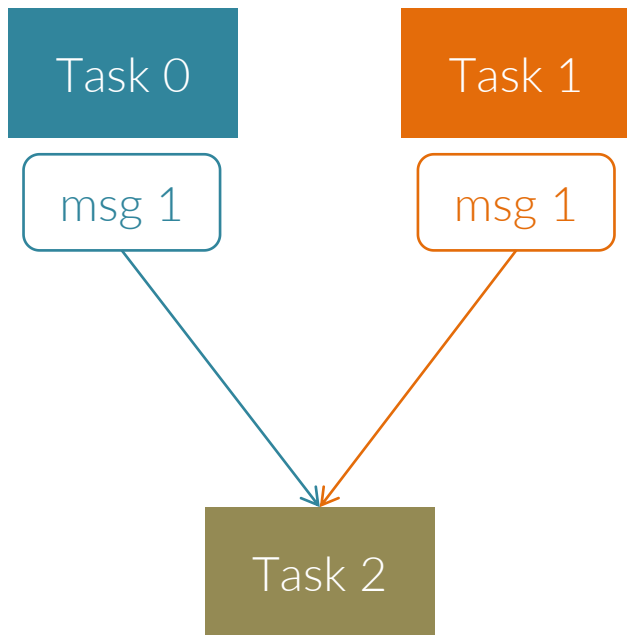


- Messages from the same source to the same target with same tags are ensured to arrive in issuing order (`msg1`, `msg2` and `msg 1`, `msg 2` respectively).
- Messages to the same target from different sources are *not* ensured to arrive in *any order* (not even if with the same tag).



# Fairness

MPI does not ensure any fairness in case of message starvation



If two messages from two sources match a `Recv` on a task ( two `Send` with same size, same tag, same comm; one `Recv` with that size, `ANY_SOURCE` and either the same tag or `ANY_TAG` ) then it is undefined which one will be received.



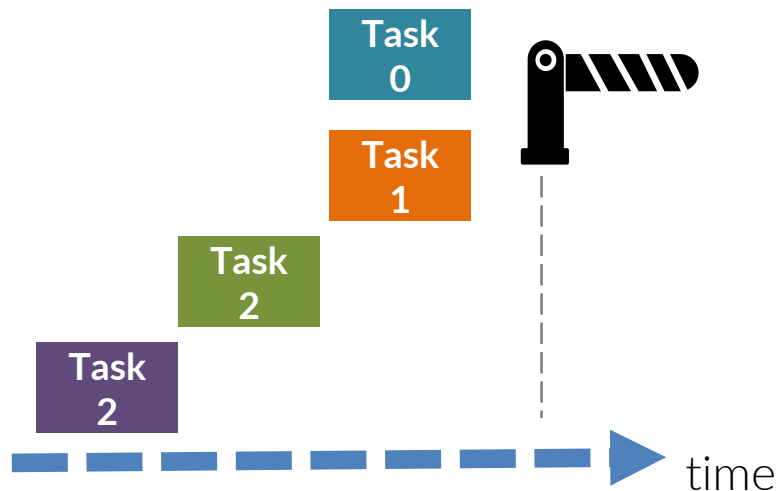
# The most violent sync: the barrier

The MPI\_Barrier ensures a synchronization among the MPI threads

```
int MPI_Barrier (MPI_Comm comm) ;
```

It is a **collective call**, we'll see more about that tomorrow.

The call completes when all the MPI ranks in the group have entered the barrier.





# First exercises: Ping-Pong test

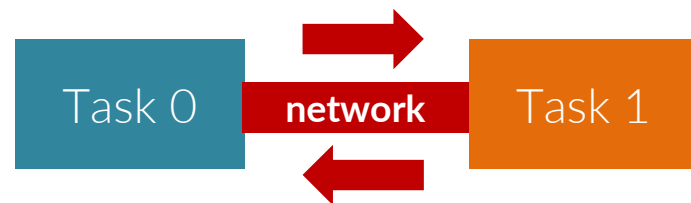
The aim is to estimate the latency time and the bandwidth of the network between two nodes

Modelling the communication time as

$$t_{\text{comm}} = \lambda + \text{size} / \text{bw}$$

where  $\lambda$  is the latency, bw is the bandwidth and  $\lambda$  size is the message size,

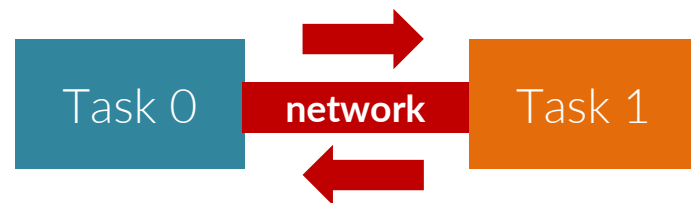
by exchanging messages of different size we should be able to infer both  $\lambda$  and bw.





# First exercises: Ping-Pong test

- 1) > Rank 0 sends a msg to Rank 1, with tag 01  
> Rank 1 waits in Recv
- 2) > Rank 1 sends a msg to Rank 0, with tag 10  
> Rank 0 waits in Recv
- 3) repeat N times ( $2 \times N$  messages), time each exchange and accumulate the timings.  
You may use either the timer that you know or `MPI_Wtime()`





# Different P2P communication modes

mode	routine	notes
standard	<code>MPI_Send</code>	Safe to modify data once returns. Equiv. to synchronous or asynchronous mode (uses sys buffers) depending on msg. size and implementation choices
synchronous	<code>MPI_Ssend</code>	Completes when the receive has started (it does <i>not</i> wait for the receive <i>completion</i> ). Unsafe communication patterns will deadlock ► a way to probe your code.
asynchronous or “buffered”	<code>MPI_Bsend</code>	Completes after the buffer has been copied. Needs an explicit buffer.
ready	<code>MPI_Rsend</code>	Mandatory that the matching receive has already been posted. May be the fastest solution, but it is quite problematic
<i>all</i>	<code>MPI_Recv</code>	One Recv serves ‘em all





# Ssend

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

MPI\_Ssend has the same signature of MPI\_Send.

The only difference is that MPI\_Ssend always apply the synchronous (i.e. the rendez-vous) protocol, and hence it returns when the actual data sending starts.

That is the reason why it can be used to spot unsafe communication patterns.



# Bsend

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

MPI\_Bsend has the same signature of MPI\_Send.

The difference is that MPI\_Bsend uses a buffer that must have been “attached” previously and is “detached” afterwards.



# Bsend

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

```
buffer_size = sizeof(int)*data_size + (Ntasks-1)*MPI_BSEND_OVERHEAD;
mybuffer     = (int*)malloc(buffer_size);
```

```
// attach the bufer
```

```
//
```

```
MPI_Buffer_attach( (void*)&mybuffer, buffer_size );
```

```
for ( int j = 0; j < Ntasks; j++ )
```

```
    if ( j != Myrank )
```

```
        MPI_Bsend( data, data_size, MPI_INT, j, j, myCOMM_WORLD );
```

```
// at this point, since Bsend has returned, the data have
```

```
// been copied into the buffer mybuffer
```

```
MPI_Buffer_detach( mybuffer, &buffer_size );
```

1) allocate room for the buffer. Clearly its size must be the max data size that will be sent.

Note that **MPI\_BSEND\_OVERHEAD** bytes must be added per every call posted

2) notify to MPI that that area is the buffer to be used. That is said “to attach”

3) when all the Bsend's issued that use that buffer have completed, the buffer can be “detached”

4) the detach will not return until the send has been completed

buffered\_send.c



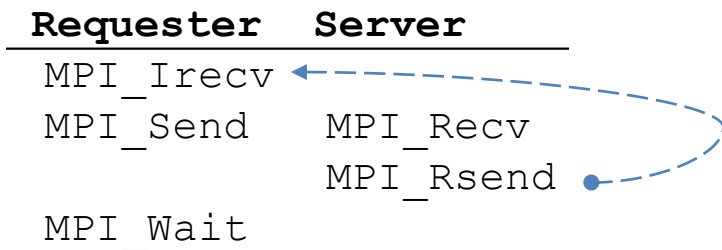
# Rsend

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

MPI\_Rsend has the same signature of MPI\_Send.

The difference is that MPI\_Rsend requires that the matching MPI\_Recv was already posted because it skips all the protocols and immediately starts the communication. It may be really performant but it must be used with **extreme caution**.

A typical pattern:



rsend.c



# Send & Receive

```
int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype,  
                  int dest, int sendtag,  
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,  
                  int source, int recvtag,  
                  MPI_Comm comm, MPI_Status &status);
```

```
int MPI_Sendrecv_replace (  
    void *buf, int count, MPI_Datatype type,  
    int dest, int sendtag,  
    MPI_Comm comm, MPI_Status &status);
```

Quite often there is a combined Send&Recv pattern between pairs of processes, for instance when performing domain decomposition when an all-to-all exchange takes place, or a “shift” on a chain of procs. MPI\_Sendrecv offers this combined call that executes exactly that pattern. With the `_replace` variant, the same amount of data is sent from a buffer / received on the same buffer.



# Non-Blocking communications

Until now we have considered p2p communication functions that do not return until some conditions are met (either the copy of the data into a buffer or the actual delivery of the data to the recipient, in case of the sender, or the actual arrival of the data into their local destination in case of the receiver).

As such, the caller is *blocked* into the call and can not perform any other operation. Those functions are consequently identified as **blocking functions**.

If what follows the Send/Recv depends on the fact that the operations mentioned above actually completed, then the usage of those functions reflects an actual dependency and there is little to be done.

However, if there are other instructions that could be executed while waiting that the data arrive at destination, by using blocking functions we are losing parallelism.



# Non-Blocking communications

To obviate to this issue, MPI offers the **non-blocking functions**, i.e. a set of functions that return immediately.

However, their return does not mean that the communication has completed but only that it has been posted on an internal queue system that will execute it at some point in the future.

To assess, at any moment, whether the communication has been executed, MPI provides dedicated routines:

```
MPI_Test ( MPI_Request *, int *flag, MPI_Status *)  
MPI_Wait ( MPI_Request *, MPI_Status *)  
MPI_Waitall (int count, MPI_Request array_of_req[],  
             MPI_Status array_of_st[] )
```





# Non-Blocking communications

## General non-blocking syntax

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

```
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype,  
              int source, int tag, MPI_Comm comm,  
              MPI_Request *request );
```

The request variable is used to handle the status of the `MPI_Isend` operation posted.  
At any point after the call, the status can be determined by the immediately-returning call



# Non-Blocking communications

## General non-blocking syntax

The `request` variable is used to handle the status of the `MPI_Isend` or `MPI_Irecv` operation posted.

At any point after the call, the status can be determined by the immediately-returning call

```
int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status );
```

A call to `MPI_Test` returns `flag = true` if the operation identified by `request` is complete. In such a case, the `status` object is set to contain information on the completed operation; if the communication object was created by a nonblocking send or receive, then it is deallocated and the request handle is set to `MPI_REQUEST_NULL`. The call returns `flag = false`, otherwise. In this case, the value of the `status` object is undefined. `MPI_Test` is a local operation.

which sets the `flag` variable to 0 (“not completed”) or 1 (“completed”)

In case the `status` variable is not needed, `MPI_STATUS_IGNORE` is a viable option (look in the man page how to inspect the `status` variable).



# Non-Blocking communications

## General non-blocking syntax

The `request` variable is used to handle the status of the `MPI_Isend` or `MPI_Recv` operation posted.

As well, the completion of a `Isen` call can be determined by the blocking call

```
int MPI_Wait( MPI_Request *request, MPI_Status *status );
```

A call to `MPI_Wait` returns when the operation identified by `request` is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to `MPI_Wait` and the request handle is set to `MPI_REQUEST_NULL`.

In case the `status` variable is not needed, `MPI_STATUS_IGNORE` is a viable option (look in the man page how to inspect the `status` variable).



# Non-Blocking communications

To assess, at any moment, whether the communication has been executed, MPI provides dedicated routines:

for one communication:

```
MPI_Test ( MPI_Request *, int *flag, MPI_Status *)
```

```
MPI_Wait ( MPI_Request *, MPI_Status *)
```

for many communications:

```
MPI_Testall (int count,
             MPI_Request array_of_req[],
             int flags[],
             MPI_Status array_of_st[] )
```

Annotations for `MPI_Testall`:

- how many communications we are testing (points to `count`)
- the count-long array of requests (points to `array_of_req`)
- the count-long array of flags to be returned (points to `flags`)
- the count-long array of status (may be IGNORE) (points to `array_of_st`)

```
MPI_Waitall (int count, MPI_Request array_of_req[], MPI_Status array_of_st[] )
```



# Non-Blocking communications

for many communications:

```
MPI_Testall / MPI_Waitall  
MPI_Testany / MPI_Waitany  
MPI_Testsome / MPI_Waitsome
```

```
MPI_Testany (int count, MPI_Request array_of_requests[],  
             int *index, int *flag, MPI_Status *status)
```

**flag** will be true if any of the communications succeeded; **index** will contain its index.

```
MPI_Testsome (int incount, MPI_Request array_of_requests[],  
              int *outcount, int array_of_idx[], MPI_Status array_of_st[])
```

**outcount** contains how many communications succeeded; **array\_of\_idx** contains the indices of the operations that completed



# Non-Blocking communications

for many communications:

```
MPI_Testall / MPI_Waitall  
MPI_Testany / MPI_Waitany  
MPI_Testsome / MPI_Waitsome
```

```
MPI_Waitany (int count, MPI_Request array_of_requests[],  
            int *index, MPI_Status *status)
```

Returns as soon as a communication in the pool completes; **index** will contain its index.

```
MPI_Waitsome (int incount, MPI_Request array_of_requests[],  
             int *outcount, int array_of_idx[], MPI_Status array_of_st[])
```

**outcount** contains how many communications succeeded; **array\_of\_idx** contains the indices of the operations that completed in the first outcount entries



# Different P2P communication modes

mode	Blocking routine	Non-blocking routine
standard	<code>MPI_Send</code>	<code>MPI_Isend</code>
synchronous	<code>MPI_Ssend</code>	<code>MPI_Issend</code>
asynchronous or “buffered”	<code>MPI_Bsend</code>	<code>MPI_Ibsend</code>
ready	<code>MPI_Rsend</code>	<code>MPI_Irsend</code>
<i>all</i>	<code>MPI_Recv</code>	<code>MPI_Irecv</code>

All the sending routines have a correspondent non-blocking version



# Non-Blocking communications

## Example of non-blocking communications usage: overlapping comm. & computation

```
MPI_Recv ( data_bunch, from_prev_proc );
int flag_send = 1;

while ( data_bunch != no_data )
{
    int          flag_recv;
    MPI_Request req_recv, req_send;

    MPI_Irecv ( next_data_bunch, &req_recv);
    process ( data_bunch );

    do {
        if ( flag_send )
            MPI_Isend ( data_bunch, to_next_proc, &req_send );
        else
            MPI_Test( &req_send, &flag, MPI_STATUS_IGNORE )
    } while ( flag == 0 )

    MPI_Test( req_recv, &flag, MPI_STATUS_IGNORE);
    while ( flag != true ) {
        do_something_else();
        MPI_Test( req_recv, flag); }
}
```

Non-blocking communications avoid to get stuck in non-returning communication when the involved processes are not synchronized.

If there are other independent tasks that the processes can perform, the non-blocking can be used instead. Quite often it is possible to re-design the workflow so that the communications are “pre-emptively” issued while some calculations are performed on a previous data bunch.



that's all, have fun

"So long  
and thanks  
for all the fish"