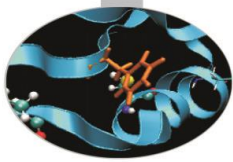
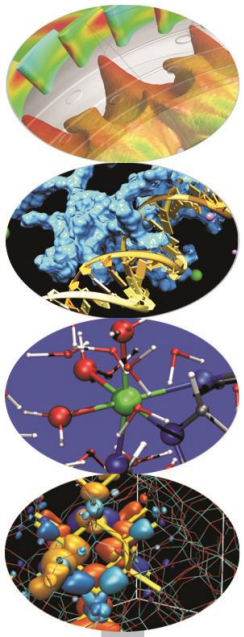
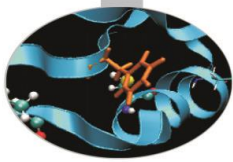


MPI introduction



MPI (Message Passing Interface)

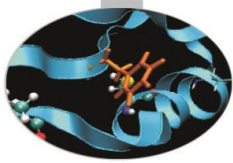


MPI origin: 1992, "Workshop on Standards for Message Passing in a Distributed Memory Environment“

- **MPI-1.0**: June 1994;
- **MPI-1.1**: June 1995;
- **MPI-1.2** e **MPI-2**: June 1997

60 experts from more than 40 organisations (IBM T. J. Watson Research Center, Intel’s NX/2, Express, nCUBE’s Vertex, p4, PARMACS, Zipcode, Chimp, PVM, Chameleon, PICL, …).

Many of them coming from the most important constructors of parallel computers or researchers from University, government and private research centres.



MPI versions

Some of the public domain most used MPI libraries:

MPICH : Argonne National Laboratory

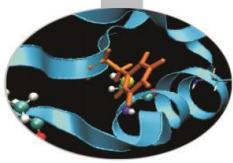
Open MPI : "open source" implementation of MPI-2

CHIMP/MPI : Edinburgh University

LAM : Ohio Supercomputer Center

To realize a (simple) parallel program *only six MPI functions* are needed.

But if the program is a complex one and the best performances are sought for, the whole MPI library may be used, with more than a hundred functions.



MPI introduction

What we will learn in this lesson on MPI library:

Compiling and executing MPI programs

C and Fortran calling syntax

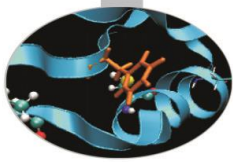
Environment

Point to point communications

Collective communications

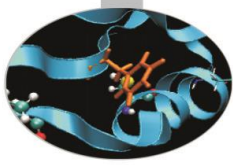
Synchronization

Hello world! (Fortran)

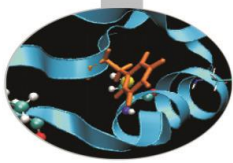


```
program greetings
    include 'mpif.h'
    integer my_rank
    integer p
    integer source
    integer dest
    integer tag
    character*100 message
    character*10 digit_string
    integer size
    integer status(MPI_STATUS_SIZE)
    integer ierr      call MPI_Init(ierr)

    call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
    if (my_rank  .NE.  0) then
        write(digit_string,FMT="(I3)") my_rank
        message = 'Greetings from process ' // trim(digit_string) // ' !'
        dest = 0
        tag = 0
        call MPI_Send(message, len_trim(message), MPI_CHARACTER, dest, tag, MPI_COMM_WORLD, ierr)
    else
        do source = 1, p-1
            tag = 0
            call MPI_Recv(message, 100, MPI_CHARACTER, source, tag, MPI_COMM_WORLD, status, ierr)
            write(6,FMT="(A)") message
        enddo
     endif
    call MPI_Finalize(ierr)
end program greetings
```

Hello world! (C/C++)



```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[])
{
    int my_rank, numprocs;
    char message[100];
    int dest, tag, source;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

    if (my_rank != 0)
        {
            sprintf(message,"Greetings from process %d !\0",my_rank);
            dest = 0;
            tag = 0;
            MPI_Send(message, sizeof(message),
                MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        } else {
            for (source = 1; source <= (numprocs-1); source++)
               {
                  MPI_Recv(message, 100, MPI_CHAR,
                         source, tag, MPI_COMM_WORLD, &status);
                  printf("%s\n",message);
            }
        }

    MPI_Finalize();

    return 0;
}
```



Hello world! (output)

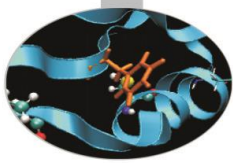
If the program is executed with two processes the output is:

```
Greetings from process 1!
```

If the program is executed with four processes the output is:

```
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
```

Compiling notes



To compile programs that make use of MPI library:

```
mpif90/mpicc/mpiCC -o <executable> <file 1> <file 2> … <file n>
```

Where: `<file n>` - program source files

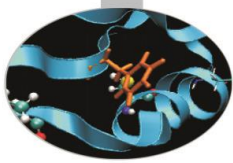
`<executable>` - executable file

To start parallel execution on one node only:

```
mpirun -np <processor_number> <executable> <exe_params>
```

To start parallel execution on many nodes:

```
mpirun -np <processor_number> -machinefile <node_list_file> \
       <executable> <exe_params>
```



MPI syntax

An MPI program should contain the directives:

```
INCLUDE   'mpif.h'                                    fortran
```

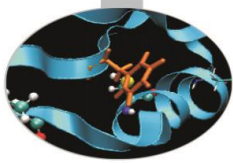
```
#include   "mpi.h"                                    C/C++
```

The above include files contain the proper definition of MPI function prototypes and parameters.

Every Fortran subroutine in the MPI library returns, as the last argument, an INTEGER type error code.

Every MPI C function returns an int value representing the error code.

Whenever a MPI function has exited without errors, the error code should have the value `MPI_SUCCESS`.

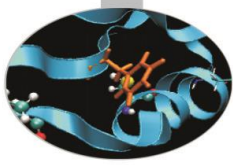


MPI syntax

```
include "mpif.h"
    integer :: ierror
    ....
    call mpi_send (..., ierror)
    if (ierror .ne. mpi_success) then
        write (*,*)"SEND operation failed"
        stop 777
    end if
```

Error code values different from `MPI_SUCCESS` are implementation dependent.

MPI syntax



Generally speaking, MPI functions have the following prototype:

```
call  MPI_name  ( parameter, ..., ierror )
```
fortran

```
rt = MPI_Name  (parameter, ...)
```
C/C++

To inizialize the MPI environment the `MPI_Init` function must be called:

```
call  MPI_INIT ( ierror )
```
fortran

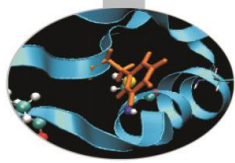
```
rt = MPI_Init(int *argc, char ***argv)
```
C/C++

On ending parallel computations the `MPI_Finalize` function should be called, otherwise processes could remain alive on local or remote computing units:

```
call MPI_FINALIZE  ( ierror )
```
fortran

```
rt = MPI_Finalize();
```
C/C++

Groups of MPI processes



A **group** is an ordered set of processes.

All MPI processes are organized in groups; each process belongs to one or more groups.

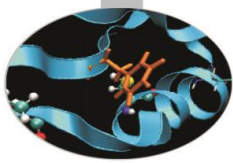
Processes are sequentially ordered in a unambiguous way. In each group every process has its own identifying number or **rank**.

Process identifying numbers are integer numbers in the range 0 – *N-1* where *N* is the group size.

On initializing MPI a default group is created containing all the processes: this group is associated to the defaul communicator `MPI_COMM_WORLD`.

Normally, if the processes are not many the default group is sufficient. Otherwise it may be convenient to create new groups defined as subsets, either disjoined or not, of the default group or formerly created groups.

MPI processes



The following function returns the extension of the group associated to a communicator, i.e. the number of processes belonging to the group:

```
call  MPI_COMM_SIZE  ( comm, size, ierr )                    fortran
```

```
ierr = int MPI_Comm_size ( MPI_Comm comm, int *size )        C/C++
```

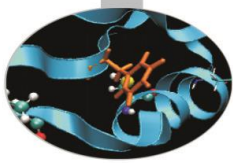
The following function returns the rank of the calling process:

```
call  MPI_COMM_RANK ( comm, rank, ierror )                   fortran
```

```
ierr = MPI_Comm_rank ( MPI_Comm comm, int *rank )            C/C++
```

Where `comm` = communicator handle (at start there is only: MPI_COMM_WORLD)
`size` = number of processes
`rank` = process rank (a number in the range 0 - size-1)
`ierr` = error code

Communication domains (communicators)



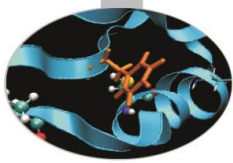
On MPI initialization the default communicator `MPI_COMM_WORLD` is generated. It allows all the activated processes to communicate each other.

Sometimes it is necessary to generate new communicators either by duplicating existing ones, or by associating it to a newly created group of processes.

A new communicator should be created every time a new group of processes is generated. A new group is always generated by choosing processes from a wider existing group.

The creation of a group of processes is a local operation, it is realized at process level. On the contrary, the creation of a new communicator is a global operation and involves (hidden) communications among all the processes of the group.

Point to point communications



Point to point communications realize connections between two processes.

From the programmer point of view communications depend on a *communicator* and are identified by a *handle* and a *tag*.

The communicator defines the processes that can be involved.

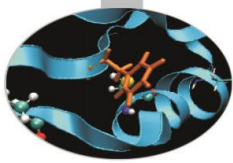
The tag is used to differentiate messages.

The handle may be useful whenever it is necessary to control the completion of the communicating operation.

A communication is said to be locally completed if the process has terminated the operation.

A communication is said to be globally completed when all the involved processes have terminated the operation.

Point to point communications



Communication calls may be **blocking** or **nonblocking**.

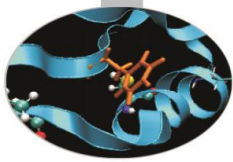
The functions relevant to **blocking calls** do not return control unless data in the message can be safely modified without changing the message data.

These functions (MPI_Send, MPI_Recv) are **very reliable** but the program execution may be slowed down because the processes are blocked until the message has been received.

The functions relevant to **nonblocking calls** are faster but **care must be taken** that the sent data are actually received and are not corrupted.

Therefore data sent by nonblocking calls can not be modified unless it is safe to do so. To check this the functions `MPI_Wait` or `MPI_Test` should be called.

Point to point communications

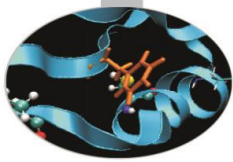


There are 4 modes of sending data in MPI:

- **Buffered** – Data are copied in a memory area explicitly allocated in the program. Either blocking or nonblocking calls are available, but non blocking calls may lead to problems if the buffer is not large enough to keep all the messages waiting to be sent.

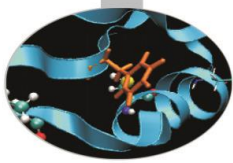
- **Synchronous** – Send operation is considered completed only if receiving operation has been started, i.e. the receiving processes have provided the memory space needed to copy the sent data. Therefore memory allocation is not an issue because memory buffers are always made available by the sender and the receiver. The problem is that if sending and receiving processes are not synchronized the execution may be slowed down.

Point to point communications



- **Standard** – The operation is automatically managed by the MPI system. If buffered communications are used, memory space is automatically allocated. Again this may lead to memory problems if data sent are too large.

- **Ready** – This mode should be used with care because when the sender starts operation the receiving process must be ready to receive the message. If this is not the case, errors and undefined results are produced. Anyhow, if synchronization is granted, this may be the fastest communication mode.



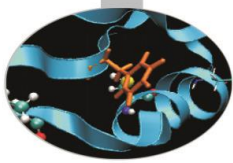
Point to point communications

Receiving calls can be blocking or nonblocking and do not differentiate sending modes.

Summary table

SEND	Blocking	Nonblocking
Standard	mpi_send	mpi_isend
Ready	mpi_rsend	mpi_irsend
Synchronous	mpi_ssend	mpi_issend
Buffered	mpi_bsend	mpi_ibsend

RECEIVE	Blocking	Nonblocking
Standard	mpi_recv	mpi_irecv



Messages

The receiving process may receive messages in random order if they are sent by different processes.

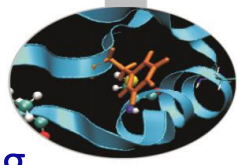
Care must be taken to insure the correct receiving order of the messages.

The following rules are always true:

- Messages with the same tag sent by the same process will be received in the sending sequence.

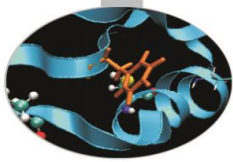
- Messages sent by nonblocking calls will be received in the sending order. This is important because otherwise large messages could be received after smaller ones sent later.

Basic data types



MPI messages are sent as arrays of data homogeneous in type. In sending and receiving calls only one data type can and shall be specified. The allowed data types may be either basic or derived. Derived types shall be defined and registered by the MPI system.

Basic types in Fortran	Basic types in C
MPI_INTEGER	MPI_CHAR
MPI_REAL	MPI_SHORT
MPI_DOUBLE_PRECISION	MPI_INT
MPI_COMPLEX	MPI_LONG
MPI_DOUBLE_COMPLEX	MPI_UNSIGNED_CHAR
MPI_LOGICAL	MPI_UNSIGNED_SHORT
MPI_CHARACTER	MPI_UNSIGNED
MPI_BYTE	MPI_UNSIGNED_LONG
MPI_PACKED	MPI_FLOAT
	MPI_DOUBLE
	MPI_LONG_DOUBLE
	MPI_BYTE
	MPI_PACKED



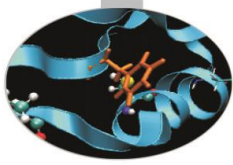
Data types

The MPI_BYTE type is not related to a Fortran or C data type. MPI_BYTE differs from MPI_CHARACTER/MPI_CHAR because MPI_BYTE messages are never translated, i.e the bit order is always maintained. Character data instead may be represented in a slightly different manner on diverse computing platforms.

The MPI_PACKED type does not have a corresponding type neither in Fortran nor in C because is used for bundled data.

In MPI communications the data type of the receiving message must always match the one of the sending call.

Sending calls



The prototype of a sending function is:

```fortran
type :: buf(count)
integer :: count, datatype, dest, tag, comm, ierror
call  MPI_sending(buf, count, datatype, dest, tag, &
       & comm, ierror )
```

fortran

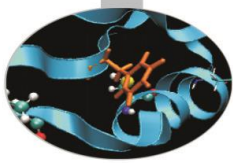
```c
ierror = MPI_Sending( void *buf, int count, MPI_Datatype
datatype,int dest, int tag, MPI_Comm comm);
```

C/C++

where:

`buf` = array of data to be sent
`count` = how many elements are sent
`datatype` = data type
`dest` = rank of the receiving process
`tag` = identifier of the message
`comm` = communicator connecting sending and receiving processes
`ierror` = error code

The starting position of the array to be sent must be passed to the sending call.



Receiving calls

The prototype of a receiving call is:

```
integer :: source, status(MPI_STATUS_SIZE)
call  MPI_receiving( buf, count, datatype, source, &
       & tag, comm, status, ierror )
```

fortran

```
ierror =   MPI_Receiving( void *buf, int count,
      MPI_Datatype datatype, int source, int tag,
      MPI_Comm comm, MPI_Status *status );
```

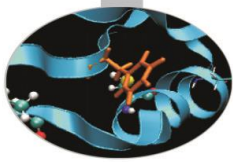
C/C++

where:

`source` = rank of the sending process

`status` = message info

`ierror` = error code



Notes on communications

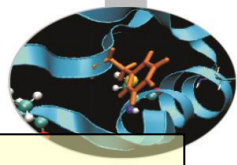
- A blocking receiving call returns only when the receiving buffer has been completed.

- Message tags and sending processes may be wildcarded using the constant values `MPI_ANY_TAG` and `MPI_ANY_SOURCE` respectively. These may be used to enhance parallel efficiency.

- On exiting the `status` array will contain useful informations. The array size is `MPI_STATUS_SIZE` and two of the most used infos are:
  - `status(MPI_SOURCE)` = rank of the sender. It may be particularly useful when the sender is `MPI_ANY_SOURCE`.
  - `status(MPI_TAG)` = message tag. It may be particularly useful when tag message is `MPI_ANY_TAG`.

The nonblocking call `MPI_Irecv` can not return a message status but a message handle `MPI_Request *request`. It can be later used by the function `MPI_Wait` to check for communication completion or by the function `MPI_Test` to wait for completion.

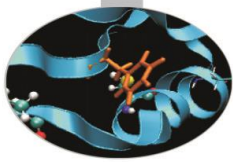
An example



```
integer, dimension (2000) :: box
integer :: error_code, msg_tag=5432, sender=2
integer, dimension (mpi_status_size) :: status
....
call mpi_recv (box(1), 1500, mpi_integer, sender, &
       & msg_tag, mpi_comm_world, status, error_code)
....
call mpi_recv (box(1501), 500, mpi_integer, &
       & mpi_any_source, mpi_any_tag, mpi_comm_world, &
       &  status, error_code)
```

In this example the first 1500 elements of the array box are received from the process with rank 2; the remaining elements are received from whichever the sending process is, without even specifying the message tag.

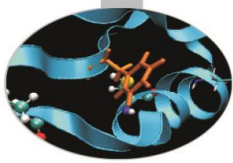
Another example



Sending and receiving the second half of an array from process 0 to process 1.

```
real :: vector(100)
integer :: status(MPI_STATUS_SIZE)
integer :: my_rank, ierr, tag, count, dest, source
      . . .
if (my_rank == 0) then
     tag = 47
     count = 50
     dest = 1
     call MPI_SEND (vector(51), count, MPI_REAL, dest, tag, &
                    & MPI_COMM_WORLD, ierr)
  else
     tag = 47
     count = 50
     source = 0
     call MPI_RECV (vector(51), count, MPI_REAL, source, tag, &
                    & MPI_COMM_WORLD, status, ierr)
 endif
```

Communications



Sending and receiving may be accomplished by one call only:

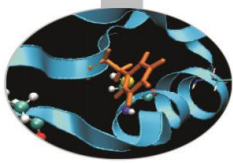
```
type :: SENDBUF, RECVBUF
integer :: SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, &
& RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR

call MPI_SENDRECV( SENDBUF, SENDCOUNT, SENDTYPE, &
        & DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, &
        & SOURCE, RECVTAG, COMM, STATUS, IERROR )
```

fortran

```
ierror = 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)
```

C/C++

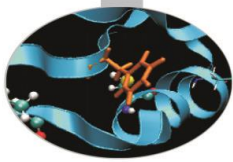
Communications



where:

- `SENDBUF` = data buffer to be sent
- `SENDCOUNT` = how many sent elements
- `SENDTYPE` = sent data type
- `DEST` = rank of the receiving process
- `SENDTAG` = sent message tag
- `RECVBUF` = receiving data buffer
- `RECVCOUNT` = how many receiving elements
- `RECVTYPE` = receiving data type
- `SOURCE` = rank of the sending process
- `RECVTAG` = receiving message tag
- `COMM` = communicator
- `STATUS` = message info
- `IERROR` = error code

Communications



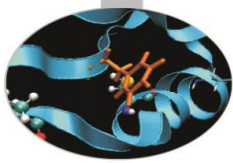
A similar function is `MPI_Sendrecv_replace`, with a simpler prototype because the sent and receiving data do share the same memory space:

```
type :: BUF
integer :: COUNT, TYPE, DEST, SENDTAG, &
                & SOURCE, RECVTAG, COMM, STATUS, IERROR

call MPI_SENDRECV_REPLACE(BUF, COUNT, TYPE, &
       & DEST, SENDTAG, SOURCE, RECVTAG, COMM, &
      & STATUS, IERROR )
```

fortran

```
ierror = MPI_Sendrecv_replace(void *buf, int count,
             MPI_Datatype type,
             int dest, int sendtag,
             int source, int recvtag,
             MPI_Comm comm, MPI_Status *status)
```

C/C++



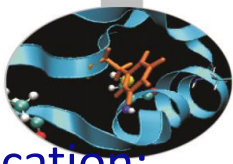
Communications

The following function returns how many elements have been received. The number of bytes received is dependent on the received data type.

```
call mpi_get_count (status, datatype, count, ierr)
```
fortran

```
ierror = mpi_get_count (MPI_Status *status, MPI_Datatype
       datatype, int *count )
```
C/C++

Communications



The following functions may be used to check the completion of a communication:

fortran

```
call mpi_wait (request_id, return_status, ierr)
call mpi_test (request_id, flag, return_status, ierr)
```

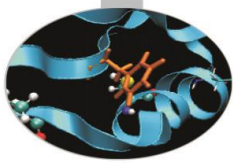
C/C++

```
ierr = MPI_Wait (MPI_Request *request_id, MPI_Status
      *return_status)
ierr = MPI_Test (MPI_Request *request_id, int *flag,
      MPI_Status *return_status)
```

The wait function blocks execution until the operation has been completed. The test function returns `FLAG=.TRUE.` if the communication is locally completed.

On completion of the operations both functions return the array containing informations about the message.

Communications



Whenever is necessary to control completion of a lot of communication operations, the following functions may be used instead.

MPI_Waitall does block execution until the operations in LIST_REQUEST are all completed.

MPI_Testall checks if all the operations in LIST_REQUEST are completed (FLAG=.TRUE.).

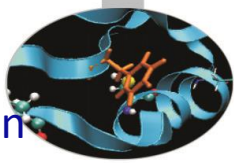
fortran

```
call mpi_waitall(count, list_requests, list_status,
ierr)
call mpi_testall(count, list_requests, flag, list_status,
ierr)
```

C/C++

```
ierr = MPI_Waitall (int count, MPI_Request
list_requests[], MPI_Status list_status[] )
ierr = MPI_Testall (int count, MPI_Request
list_requests[], int *flag, MPI_Status list_status[])
```

Communications



The function **MPI_WAITANY** blocks execution until at least one of the operations in LIST_REQUEST is locally completed.

The function **MPI_TESTANY** checks if at least one of the operations in LIST_REQUEST is locally completed. On output INDEX is the position in LIST_REQUESTS of the completed operation and RETURN_STATUS contains infos about it.

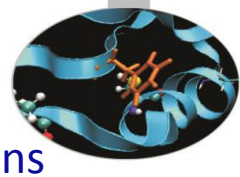
fortran

```
call mpi_waitany(count, list_requests, index,
      return_status, ierr)
call mpi_testany(count, list_requests, index, flag,
      return_status, ierr)
```

C/C++

```
ierr = MPI_Waitany (int count, MPI_Request
      list_requests[], int *index, MPI_Status
      *return_status )
ierr = MPI_Testany(int count, MPI_Request
      list_requests[], int *index, int *flag, MPI_Status
      *return_status )
```

Communications



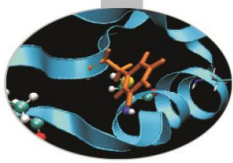
The functions **MPI_WAITSOME** e **MPI_TESTSOME** checks if some of the operations in LIST_REQUEST have been locally completed:

fortran

```
call mpi_waitsome (count, list_requests, count_done,
      list_index, list_status, ierr)
call mpi_testsome(count, list_requests, count_done,
      list_index, list_status, ierr)
```

C/C++

```
ierr = MPI_Waitsome (int incount, MPI_Request
      list_requests[], int *outcount, int list_index[],
      MPI_Status list_status[] )
ierr = MPI_Testsome(int incount, MPI_Request
      array_of_requests[], int *outcount, int
      array_of_indices[],MPI_Status array_of_statuses[])
```



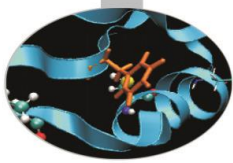
Exercise

ping-pong is perhaps the simplest example of point-to-point communication.

In a two-process ping-pong process 0 sends a message to process 1 and this sends it back to process 0. Try writing a program that realizes this operations.

Examples:

- ping-pong.c.txt
- ping-pong.f.txt



Collective communications

Communications is a very important issue in MPI programs, therefore their optimization is a mandatory effort.

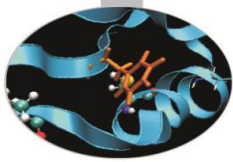
In many cases communications involve a lot of processors and realizing them by point-to-point communications become inefficient and an error prone exertion. For this reason MPI library contains functions optimized to accomplish collective communications. Therefore using collective communications in such cases is much more effective than using point-to-point communications.

Collective communications do not need tags for messages.

All collective communications are blocking.

Collective communication calls carry out both sending and receiving operations.

The calls to collective communication functions should be issued by all the processes of a given communicator.



Collective communications

Collective communications may be of two types: **data transfer** and **global computations**.

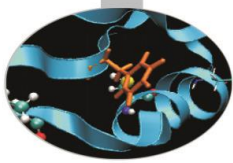
Data transfer functions can be:

- **broadcast** - data are shared among all the processes
- **gather** - data are collected from every process
- **scatter** - data are distributed to the processes

Global computation functions can be:

- **reduction** - the result is a computed value
- **scanning** – partial reduction results

Collective communications



The following function may be used to send the same data to all the processes belonging to a communicator. A loop performing point-to-point communications to all the processes gives the same results but is much less efficient.

fortran

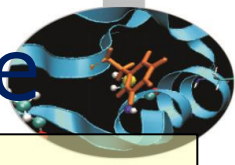
```
type :: array
integer :: count, datatype, root, comm, ierror
call  MPI_BCAST( array, count, datatype, root, comm, ierror )
```

C/C++

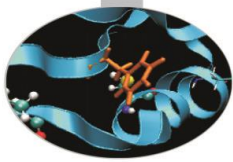
```
ierror = MPI_Bcast( void *buffer, int count, MPI_Datatype
                                  datatype, int root, MPI_Comm comm )
```

where: `array` = data to be sent
`count` = how many elements
`datatype` = data type of the elements
`root` = process owing data to be sent
`comm` = communicator
`ierror` = error code

Collective communications example



```
subroutine GetData (a, b, n, my_rank)

real :: a, b
integer :: n, my_rank, ierr
include 'mpif.h'

if (my_rank == 0) then
                print *, 'Enter a, b, and n'
         read *, a, b, n
endif

call MPI_BCAST (a, 1, MPI_REAL , 0, MPI_COMM_WORLD, ierr )
call MPI_BCAST (b, 1, MPI_REAL , 0, MPI_COMM_WORLD, ierr )
call MPI_BCAST (n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr )

end subroutine GetData
```



Collective communications

The following function may be used whenever data dispersed among the processes have to be collected in one ROOT process

where:

SEND_COUNT - how many elements are sent

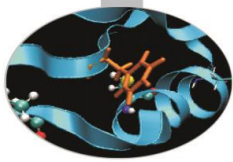
RECV_COUNT - how many elements have to be received

```
type :: SEND_BUF(*), RECV_BUF(*)
integer :: SEND_COUNT, SEND_TYPE, RECV_COUNT, RECV_TYPE, ROOT, &
       & COMM, IERROR, DISP(comm_size)
call MPI_Gather ( SEND_BUF, SEND_COUNT, SEND_TYPE, RECV_BUF, &
       & RECV_COUNT, RECV_TYPE, ROOT, COMM, IERROR )
```

fortran

```
ierror = MPI_Gather ( void *send_buf, int send_count, MPI_Datatype
       sendtype, void *recv_buf, int recv_count, MPI_Datatype
       recv_type, int root, MPI_Comm comm )
```

C/C++



Collective communications

The following function may be used to collect data dispersed among the processes if each process owns a different number of elements.

In the function variants with an ending "v"“ the array `RECV_COUNT(:)` specify how many elements are stored in each process.

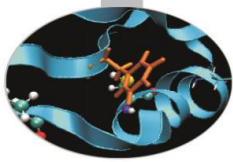
The array `DISP(:)` specifies the position in the receiving buffer where data coming from i^{th} process must be copied.

fortran

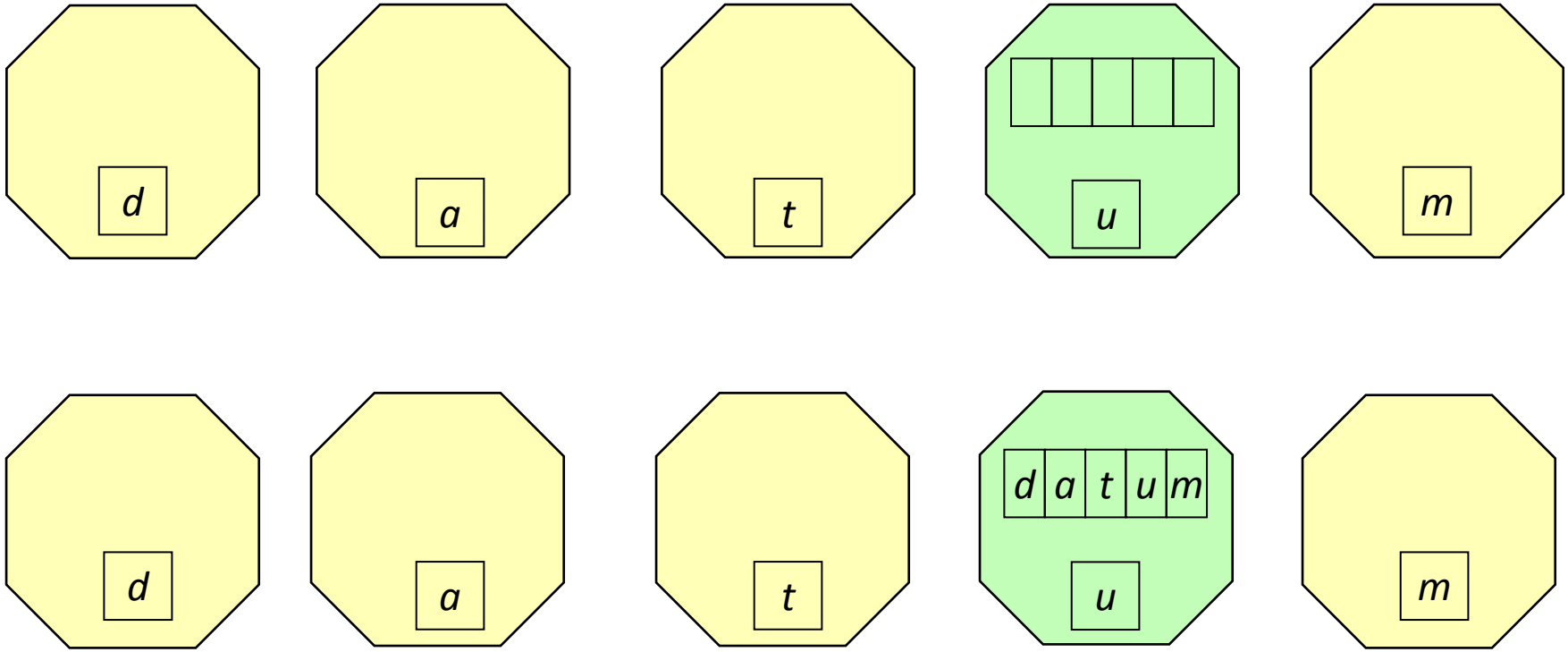
```
call MPI_Gatherv ( SEND_BUF, SEND_COUNT, SEND_TYPE, RECV_BUF, &
       & RECV_COUNT, DISP, RECV_TYPE, ROOT, COMM, IERROR )
```

C/C++

```
ierro = MPI_Gatherv ( void *send_buf, int send_count, MPI_Datatype
       send_type, void *recv_buf, int *recv_count,
       int *disp, MPI_Datatype recv_type, int root, MPI_Comm comm )
```

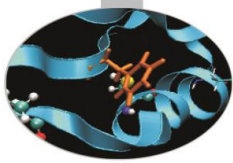


Collective communications



Every process in the communicator send the content of `send_buf` to the *root* process that receives data and orders them in the `recv_buf` array according to the rank of the sending processes.

Collective communications

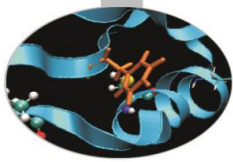


fortran

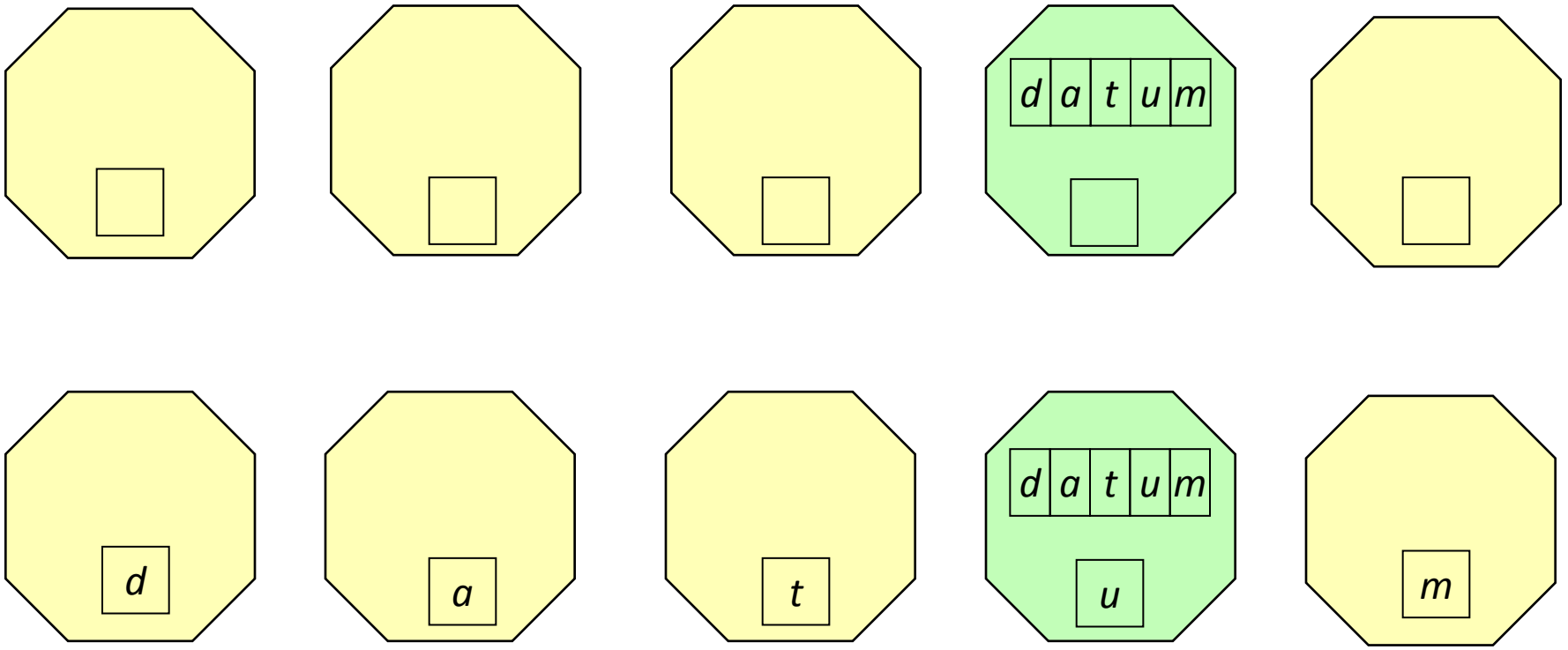
```
type :: SEND_BUF(*), RECV_BUF(*)
integer :: SEND_COUNT, SEND_TYPE, RECV_COUNT, &
                & RECV_TYPE, ROOT, COMM, IERROR


call MPI_Scatter ( SEND_BUF, SEND_COUNT, SEND_TYPE, &
                RECV_BUF, RECV_COUNT, RECV_TYPE, &
                ROOT, COMM, IERROR )
```

```
ierror = MPI_Scatter ( void *send_buf, int send_count,
      MPI_Datatype send_type,
      void *recv_buf, int recv_count, MPI_Datatype
      recv_type, int root, MPI_Comm comm )
```

C/C++



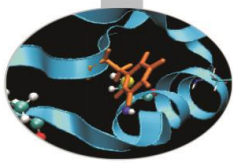
Collective communications



The `root` process disperses the content of `send_buf` array to the other processes of the communicator group.

Data are scattered according to the order of the processes.

Collective communications



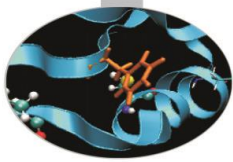
fortran

```
type :: SEND_BUF(*), RECV_BUF(*)
integer :: SEND_COUNT, SEND_TYPE, RECV_COUNT,  RECV_TYPE,
COMM, IERROR

call MPI_Allgather ( SEND_BUF, SEND_COUNT, SEND_TYPE,
       RECV_BUF, RECV_COUNT, RECV_TYPE, COMM, IERROR )
```

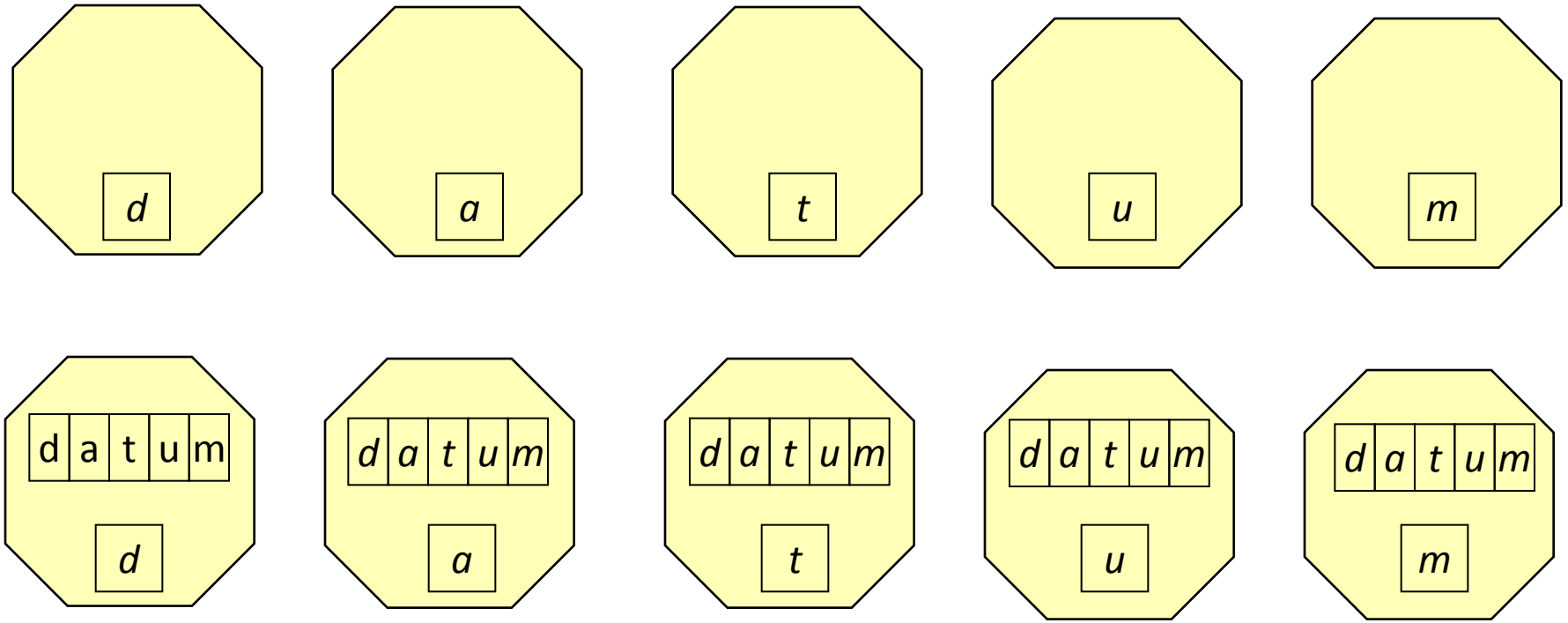
C/C++

```
ierror = MPI_Allgather ( void *send_buf, int send_count,
MPI_Datatype send_type,

void *recv_buf, int recv_count, MPI_Datatype recv_type,
MPI_Comm comm )
```

The above function may be used to collect data from all the processes to all the processes. It is equivalent to a sequence of calls to `MPI_Gather` in which each call identifies a different process as *root*.

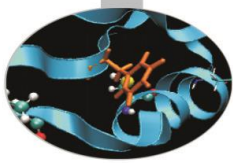


Collective communications



Every process in the communicator send the content of `send_buf` to *all the other* processes that receive data and order them in the `recv_buf` array according to the rank of the sender.

Collective communications



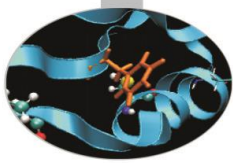
Collective communications include global computations, of a reduction type.

The result of the computations may be:

- stored in one process only
- broadcasted to all the processes
- scattered to all the processes

Three functions are available:

- MPI_Reduce
- MPI_Allreduce
- MPI_Reduce_scatter



Collective communications

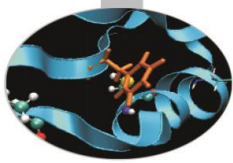
The following function may be used to compute reduction operations such as sum, product, logical, min/max and others. It must be called by all the processes of the communicator `comm`. The result is stored in the process identified as `root`. If `count > 1` then `send_buf` and `recv_buf` are arrays and the computation is executed element by element.

fortran

```
type :: send_buf, recv_buf
integer :: count, datatype, op, root, comm, ierror
call MPI_REDUCE ( send_buf, recv_buf, count, &
                  datatype, op, root, comm, ierror )
```

C/C++

```
ierror = MPI_Reduce ( void *send_buf, void *recv_buf,
                  int count, MPI_Datatype datatype,
                  MPI_Op op, int root, MPI_Comm comm )
```



Collective communications

`send_buf` = data to be used for computation, operands

`recv_buf` = result buffer (received by root process only)

`count` = buffer size

`datatype` = type of the elements

`op` = reduction operation (es.: MPI_SUM, MPI_MAX, …)

`root` = which process stores the results

`comm` = communicator

`ierror` = error code

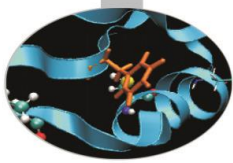
As an example, if `count=3` and `op=MPI_SUM`, then:

Recv_buf(0) = $send\_buf_{proc0}(0) + \ldots + send\_buf_{procN-1}(0)$

Recv_buf(1) = $send\_buf_{proc0}(1) + \ldots + send\_buf_{procN-1}(1)$

Recv_buf(2) = $send\_buf_{proc0}(2) + \ldots + send\_buf_{procN-1}(2)$

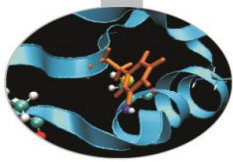
Collective communications



The available operations are:

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical AND
MPI_BAND	bit-wise AND
MPI_LOR	logical OR
MPI_BOR	bit-wise OR
MPI_LXOR	logical XOR
MPI_BXOR	bit-wise XOR
MPI_MAXLOC	maximum value and location
MPI_MINLOC	minimum value and location

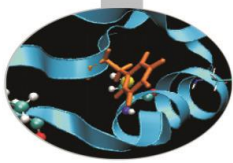
Collective communications example



The following portion of code shows how to use MPI_MAXLOC reduction operation that requires, together with MPI_MINLOC, a struct to be defined:

```
    .   .   .
struct
{
   double value;
   int rank;
} array1[len], array2[len];
    .   .   .
MPI_Reduce(array1, array2, len, MPI_DOUBLE_INT,
           MPI_MAXLOC, 0, MPI_COMM_WORLD );
```

See the example program in *Es-maxloc.c.txt*.



Collective communications

fortran

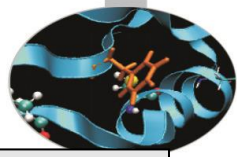
```
type :: OPERAND(*), RESULT(*)
integer :: COUNT, DATATYPE, OP ,COMM, IERROR
call MPI_AllReduce ( OPERAND, RESULT, COUNT, DATATYPE, OP,
      COMM, IERROR )
```

C/C++

```
ierror = MPI_Allreduce ( void *operand, void *result, int
      count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

This function differs from the previous one because the operation result is stored in all the processes of the communicator `comm`.

Collective communications



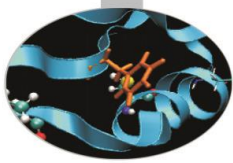
```
<type>, IN :: SENDBUF(*)
<type>, OUT :: RECVBUF(*)
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR


call MPI_REDUCE_SCATTER (SENDBUF, RECVBUF, RECVCOUNTS,
   DATATYPE, OP, COMM, IERROR)
```

fortran

```
ierror = MPI_Reduce_scatter ( void *sendbuf, void *recvbuf,
      int *recvcounts, MPI_Datatype datatype, MPI_Op op,
      MPI_Comm comm )
```

C/C++

Using the function `MPI_Reduce_scatter`, the reduction result is first computed element by element, then the obtained vector is split into disjoined segments and dispersed to all the processes. The array `recvcounts(:)` is used to specify how many elements each process will store.



Process synchronization

Whenever it is necessary that all the processes get to a determined point at the same time, then sychronization barriers must be used. To avoid heavy loss of performances barriers should be used with care and only if it is unavoidable, i.e. the implemented algorithm requires it.

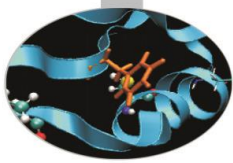
The following function can be used to define a synchronising point:

```
integer :: comm, ierror
call MPI_BARRIER ( comm, ierror )
```
fortran

```
ierror = MPI_Barrier ( MPI_Comm comm )
```
C/C++

Where: `comm` – communicator whose processes must be synchronized
`ierror` – error code.

This function returns only after all the processes have called it.



Performance evaluation

It is often useful to measure computing time of portions of the program. The following functions may be used. Both functions return a double floating point value.

fortran

```
REAL(8) :: t1, t2, elapsed
t1 = MPI_WTIME ( )
…
t2 = MPI_WTIME ( )
elapsed = t2 - t1
```

C/C++

```
double t1, dt
t1 = MPI_Wtime()     /* elapsed time in seconds */
dt = MPI_Wtick()     /* time resolution in seconds */
```

Time values are process dependent unless MPI_WTIME_IS_GLOBAL is defined and its value is .TRUE..