CEMRACS - MPI

D. Lecas

24 juillet 2012

Sommaire I

Sommaire II

1 – Introduction
1.1 – Definitions

➊ The sequential programming model :

- ☞ the program is executed by one and only one process ;
- ☞ all the variables and constants of the program are allocated in the memory of the process ;
- ☞ a process is executed on a physical processor of the machine.

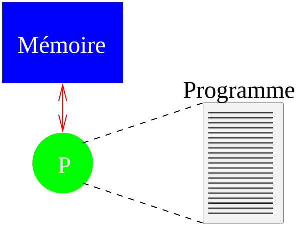


FIGURE 1 – Sequential programming model

❷ In the message passing programming model :

- ☞ the program is written in a classic language (`Fortran`, `C`, `C++`, etc.) ;
- ☞ each process may executes different parts of a program ;
- ☞ all the variables of the program are private and reside in the local memory of each process ;
- ☞ a variable is exchanged between two or many processes via a call to subroutines.

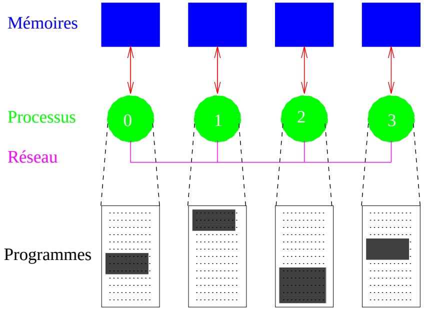


FIGURE 2 – Message-Passing Programming Model

2 – Environnement

2.1 – Description

- Every program unit calling MPI subroutines has to include a header file. In Fortran, we must use the **mpi** module introduced in MPI-2 (in MPI-1, it was the **mpif.h** file), and in C/C++ the **mpi.h** file.
- The MPI_INIT() subroutine initializes the necessary environment :

```
integer, intent(out)  : code

call MPI_INIT(code)
```

- The MPI_FINALIZE() subroutine disables this environment :

```
integer, intent(out)  : code

call MPI_FINALIZE(code)
```

☞ All the operations made by MPI are related to communicators. The default communicator is `MPI_COMM_WORLD` which includes all the active processes.

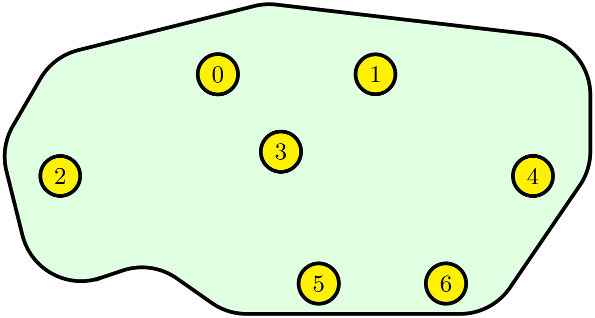


FIGURE 3 – MPI_COMM_WORLD Communicator

☞ At any moment, we can know the number of processes managed by a given communicator by the MPI_COMM_SIZE() subroutine :

```
integer, intent(out)  : nb_procs,code

call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
```

☞ Similarly, the MPI_COMM_RANK() subroutine allows to obtain the process rank (i.e. its instance number, which is a number between 0 and the value sent by MPI_COMM_SIZE() – 1) :

```
integer, intent(out)  : rank,code

call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
```

2 – Environnement

2.2 – Exemple

```fortran
program who_am_I
  use mpi
  implicit none
  integer  : nb_procs,rank,code

  call MPI_INIT(code)

  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)

  print *,'I am the process ',rank,' among ',nb_procs

  call MPI_FINALIZE(code)
end program who_am_I
```

```
> mpiexec -n 7 who_am_I

I am the process 3 among 7
I am the process 0 among 7
I am the process 4 among 7
I am the process 1 among 7
I am the process 5 among 7
I am the process 2 among 7
I am the process 6 among 7
```

3 – Point to point communications

3.1 – General concepts

☞ A point to point communication occurs between two processes, one names the sender process and the other one the receiver process

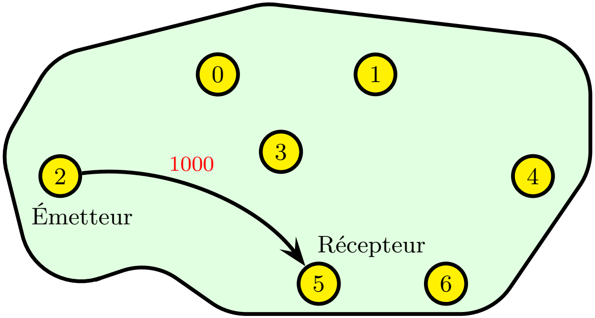


Figure 4 – Point to point communications

- ☞ The sender and the receiver are identified by their rank in the communicator.
- ☞ The so-called message envelope is composed of :
  1. the rank of the send process ;
  2. the rank of the receive process ;
  3. the tag of the message ;
  4. the name of the communicator which will define the operation communication context.
- ☞ The exchanged data are predefined (integer, real, etc.) or personal derived datatypes.
- ☞ There are in each case many communication modes, calling different protocols.

```fortran
program point_to_point
  use mpi
  implicit none

  integer, dimension(MPI_STATUS_SIZE)  : status
  integer, parameter                   : tag=100
  integer                              : rank,value,code

  call MPI_INIT(code)

  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)

  if (rank == 2) then
     valeur=1000
     call MPI_SEND(value,1,MPI_INTEGER,5,tag,MPI_COMM_WORLD,code)
  elseif (rank == 5) then
     call MPI_RECV(value,1,MPI_INTEGER,2,tag,MPI_COMM_WORLD,status,code)
     print *,'Myself, process 5, I have received ',value,' from the process 2.'
  end if

  call MPI_FINALIZE(code)

end program point_to_point
```

```
> mpiexec -n 7 point_to_point

Myself, process 5, I have received 1000 from the process 2
```

3 – Point to point communications

3.2 – Predefined MPI Datatypes

TABLE 1 – Predefined MPI Datatypes (Fortran)

Type MPI	Type Fortran
MPI_INTEGER	`INTEGER`
MPI_REAL	`REAL`
MPI_DOUBLE_PRECISION	`DOUBLE PRECISION`
MPI_COMPLEX	`COMPLEX`
MPI_LOGICAL	`LOGICAL`
MPI_CHARACTER	`CHARACTER`

3 – Point to point communications
3.3 – Other possibilities

- ☞ On the reception of a message, the process rank and the tag can be wild card, `MPI_ANY_SOURCE` and `MPI_ANY_TAG` respectively.
- ☞ A communication with the dummy process of rank `MPI_PROC_NULL` has no effect.
- ☞ `MPI_STATUS_IGNORE` is a predefined constant that can be used instead of status variable.
- ☞ `MPI_SUCCESS` is a predefined constant which allows testing the return code of an MPI function.
- ☞ There are syntaxic variants, `MPI_SENDRECV()` and `MPI_SENDRECV_REPLACE()`, which launch simultaneously a send and a receive (in the first case, the receive buffer must be necessarily different of the send buffer).
- ☞ We can create more complex data structures.

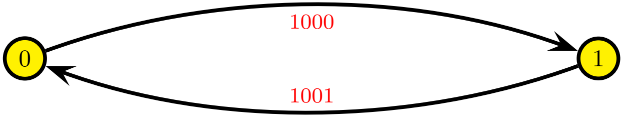


FIGURE 5 – Communication between the processes 0 and 1

```fortran
program sendrecv
  use mpi
  implicit none
  integer                                    : rank,value,num_proc,code
  integer,parameter                          : tag=110

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)

  ! We suppose that we have exactly 2 processes
  num_proc=mod(rank+1,2)

  call MPI_SENDRECV(rank+1000,1,MPI_INTEGER,num_proc,tag,value,1,MPI_INTEGER, &
                    num_proc,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE,code)
  ! Test of the return code of the MPI_SENDRECV subroutine
  if (code /= MPI_SUCCESS) call MPI_ABORT(MPI_COMM_WORLD,2,code)

  print *,'Myself, process',rank,', I have received',value,'from the process ',num_proc

  call MPI_FINALIZE(code)
end program sendrecv
```

```
> mpiexec -n 2 sendrecv

Myself, process 1, I have received 1000 from the process 0
Myself, process 0, I have received 1001 from the process 1
```

Warning ! It must be noticed that if the MPI_SEND() subprogram is implemented in a synchronous way in the implementation used of the MPI library, the previous code would be in a deadlock situation if, rather than to use the MPI_SENDRECV() subprogram we used the MPI_SEND() subprogram followed by the MPI_RECV() one. In this case, each of the two subprograms would wait a receipt command which will never happened, because the two sends would stay suspended. So, for portability reasons, it is absolutely necessary to avoid such situations.

```
call MPI_SEND(rank+1000,1,MPI_INTEGER,num_proc,tag,MPI_COMM_WORLD,code)
call MPI_RECV(value,1,MPI_INTEGER,num_proc,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE,code)
```

4 – Collective communications

4.1 – General concepts

☞ The collective communications allow to make a series of point-to-point communications in one single call.

☞ A collective communication always concerns all the processes of the indicated communicator.

☞ For each process, the call ends when its participation in the collective call is completed, in the sense of point-to-point communications (when the concerned memory area can be changed).

☞ It is useless to add a global synchronization (barrier) after a collective call.

☞ The management of tags in these communications is transparent and system-dependent. Therefore, they are never explicitly defined during the calling of these subroutines. This has among other advantages that the collective communications never interfere with point-to-point communications.

☞ There are three types of subroutines :

❶ the one which ensures the global synchronizations : `MPI_BARRIER()`.

❷ the ones which only transfer data :

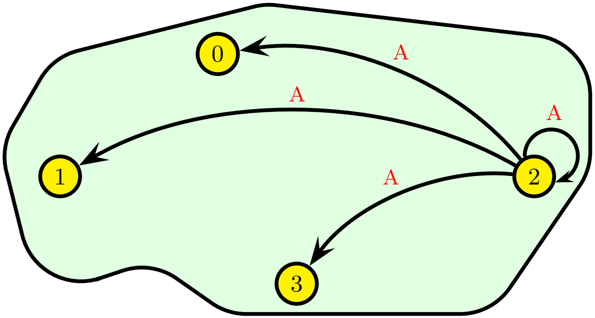
- ❏ global distribution of data : `MPI_BCAST()` ;
- ❏ selective distribution of data : `MPI_SCATTER()` ;
- ❏ collection of distributed data : `MPI_GATHER()` ;
- ❏ collection by all the processes of distributed data : `MPI_ALLGATHER()` ;
- ❏ selective distribution, by all the processes, of distributed data : `MPI_ALLTOALL()`.

❸ the ones which, in addition to the communications management, carry out operations on the transferred data :

- ❏ reduction operations (sum, product, maximum, minimum, etc.) whether they are of a predefined or personal type : `MPI_REDUCE()` ;
- ❏ reduction operations with broadcasting of the result (it is in fact equivalent to an `MPI_REDUCE()` followed by an `MPI_BCAST()`) : `MPI_ALLREDUCE()`.

4 – Collective communications

4.2 – Broadcast : MPI_BCAST()



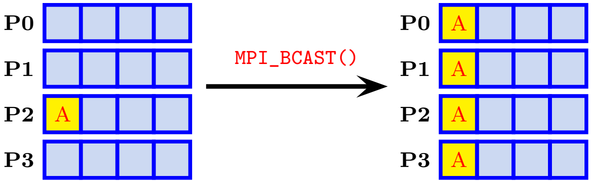


Figure 6 – Broadcast : MPI_BCAST()

```
program bcast
  use mpi
  implicit none

  integer  : rank,value,code

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)

  if (rank == 2) value=rank+1000

  call MPI_BCAST(value,1,MPI_INTEGER,2,MPI_COMM_WORLD,code)

  print *,'I, process ',rank,' I have received ',value,' of the process 2'

  call MPI_FINALIZE(code)

end program bcast
```

```
> mpiexec -n 4 bcast

I, process 2 I have received 1002 of the process 2
I, process 0 I have received 1002 of the process 2
I, process 1 I have received 1002 of the process 2
I, process 3 I have received 1002 of the process 2
```

4 – Collective communications
4.3 – Scatter : MPI_SCATTER()

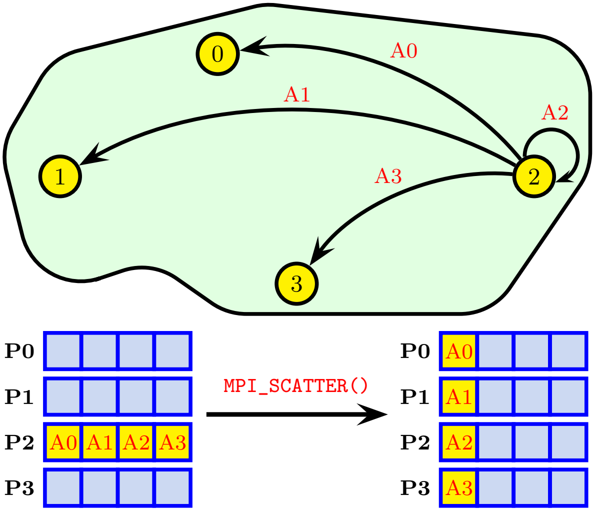


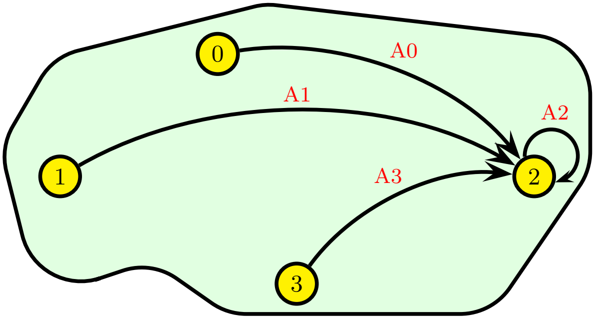
FIGURE 7 – Scatter : MPI_SCATTER()

```fortran
program scatter
  use mpi
  implicit none

  integer, parameter                  : nb_values=8
  integer                             : nb_procs,rank,block_length,i,code
  real, allocatable, dimension( :)    : values,data

  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
  block_length=nb_values/nb_procs
  allocate(data(block_length))

  if (rank == 2) then
     allocate(values(nb_values))
     values( :)=(/(1000.+i,i=1,nb_values)/)
     print *,'I, process ',rank,'send my values  array  :',&
             values(1 :nb_values)
  end if

  call MPI_SCATTER(values,block_length,MPI_REAL,data,block_length, &
                   MPI_REAL,2,MPI_COMM_WORLD,code)
  print *,'I, process ',rank,', I have received ', data(1 :block_length), &
        ' of the process 2'
  call MPI_FINALIZE(code)

end program scatter
```

```
> mpiexec -n 4 scatter
I, process 2 send my values array  :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.

I, process 0, I have received 1001. 1002. of the process 2
I, process 1, I have received 1003. 1004. of the process 2
I, process 3, I have received 1007. 1008. of the process 2
I, process 2, I have received 1005. 1006. of the process 2
```

4 – Collective communications

4.4 – Gather : MPI_GATHER()



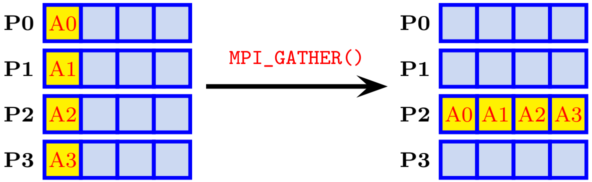


FIGURE 8 – Gather : MPI_GATHER()

```fortran
program gather
  use mpi
  implicit none
  integer, parameter                  : nb_values=8
  integer                             : nb_procs,rank,block_length,i,code
  real, dimension(nb_values)          : data
  real, allocatable, dimension( :)    : values

  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)

  block_length=nb_values/nb_procs

  allocate(values(block_length))

  values( :)=(/(1000.+rank*block_length+i,i=1,block_length)/)
  print *,'I, process ',rank,'send my values array  :',&
            values(1 :block_length)

  call MPI_GATHER(values,block_length,MPI_REAL,data,block_length, &
                  MPI_REAL,2,MPI_COMM_WORLD,code)

  if (rank == 2) print *,'I, process 2', ' have received ',data(1 :nb_values)

  call MPI_FINALIZE(code)

end program gather
```

```
> mpiexec -n 4 gather
I, process 1 send my values array  1003. 1004.
I, process 0 send my values array  1001. 1002.
I, process 2 send my values array  1005. 1006.
I, process 3 send my values array  1007. 1008.

I, process 2 have received 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
```

4 – Collective communications
4.5 – Global reduction

- ☞ A reduction is an operation applied to a set of elements in order to obtain one single value. Classical examples are the sum of the elements of a vector (`SUM(A(:))`) or the search of the maximum value element in a vector (`MAX(V(:))`).
- ☞ MPI proposes high-level subroutines in order to operate reductions on distributed data on a group of processes. The result is obtained on one process (`MPI_REDUCE()`) or on all (`MPI_ALLREDUCE()`, which is in fact equivalent to an `MPI_REDUCE()` followed by an `MPI_BCAST()`).
- ☞ If many elements are implied by process, the reduction function is applied to each one of them.
- ☞ The `MPI_SCAN()` subroutine allows also to make partial reductions by considering, for each process, the previous processes of the group and itself.
- ☞ The `MPI_OP_CREATE()` and `MPI_OP_FREE()` subroutines allow personal reduction operations.

TABLE 2 – Main Predefined Reduction Operations (there are also other logical operations)

Name	**Opération**
MPI_SUM	Sum of elements
MPI_PROD	Product of elements
MPI_MAX	Maximum of elements
MPI_MIN	Minimum of elements
MPI_MAXLOC	Maximum of elements and location
MPI_MINLOC	Minimum of elements and location
MPI_LAND	Logical AND
MPI_LOR	Logical OR
MPI_LXOR	Logical exclusive OR

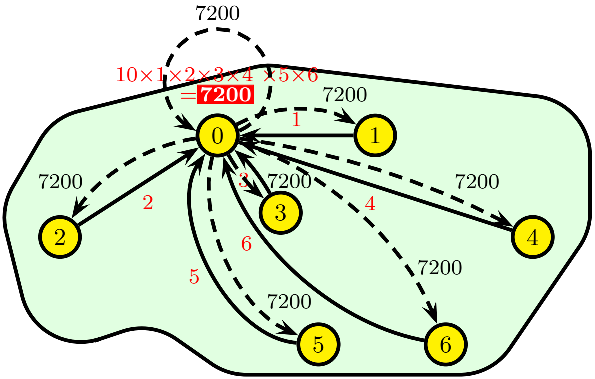


Figure 9 – Distributed reduction (product) with broadcast of the result

```fortran
program allreduce

  use mpi
  implicit none

  integer  : nb_procs,rank,value,product,code

  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)

  if (rank == 0) then
     value=10
  else
     value=rank
  endif

  call MPI_ALLREDUCE(value,product,1,MPI_INTEGER,MPI_PROD,MPI_COMM_WORLD,code)

  print *,'I,process ',rank,',I have received the value of the global product ',product

  call MPI_FINALIZE(code)

end program allreduce
```

```
> mpiexec -n 7 allreduce

I, process 6, I have received the value of the global product 7200
I, process 2, I have received the value of the global product 7200
I, process 0, I have received the value of the global product 7200
I, process 4, I have received the value of the global product 7200
I, process 5, I have received the value of the global product 7200
I, process 3, I have received the value of the global product 7200
I, process 1, I have received the value of the global product 7200
```

5 – One-sided Communication

5.1 – Introduction

There are various approaches to transfer data between two different processes. Among the most commonly used are :

➊ Point-to-point communications by message-passing (`MPI`, etc.) ;

➋ One-sided communications (direct access to the memory of a distant process). Also called RMA for Remote Memory Access , it is one of the major contributions of `MPI`.

5 – One-sided Communication

5.1 – Introduction

5.1.1 – Reminder : The Concept of Message-Passing

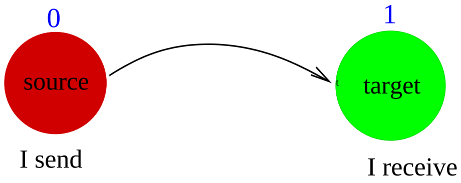


Figure 10 – Message-Passing

In message-passing, a sender (origin) sends a message to a destination process (target) which will make all what is necessary to receive this message. This requires that the sender as well as the receiver be involved in the communication. This can be restrictive and difficult to implement in some algorithms (for example when it is necessary to manage a global counter).

5 – One-sided Communication
5.1 – Introduction
5.1.2 – The Concept of One-sided Communication

The concept of one-sided communication is not new, MPI having simply unified the already existing constructors' solutions (such as shmem (CRAY), lapi (IBM), ...) by offering its own RMA primitives. Through these subroutines, a process has a direct access (in read, write or update) to the memory of another remote process. In this approach, the remote process does not have to participate in the data-transfer process.

The principle advantages are the following :

- ☞ enhanced performances when the hardware allows it,
- ☞ a simpler programming for some algorithms.

5 – One-sided Communication
5.1 – Introduction
5.1.3 – RMA Approach of MPI

The use of MPI RMA is done in three steps :

❶ definition on each process of a memory area (local memory window) visible and eventually accessible to remote processes ;

❷ start of the data transfer directly from the memory of a process to the memory of another process. It is therefore necessary to specify the type, the number and the initial and final localization of data.

❸ completion of current transfers by a step of synchronization, the data are then available.

5 – One-sided Communication

5.2 – Memory Window

All the processes participating in an one-sided communication have to specify which part of their memory will be available to the other processes ; it is the notion of memory window.

☞ More precisely, the `MPI_WIN_CREATE()` collective operation allows the creation of an MPI window object. This object is composed, for each process, of a specific memory area called local memory window. For each process, a local memory window is characterized by its initial address, its size in bytes (which can be zero) and the displacement unit size inside this window (in bytes). These characteristics can be different on each process.

6 – Derived datatypes
6.1 – Introduction

- ☞ In the communications, the exchanged data have datatypes : `MPI_INTEGER`, `MPI_REAL`, `MPI_COMPLEX`, etc.
- ☞ We can create more complex data structures by using subroutines such as `MPI_TYPE_CONTIGUOUS()`, `MPI_TYPE_VECTOR()`, `MPI_TYPE_CREATE_HVECTOR()`
- ☞ Each time that we use a datatype, it is mandatory to validate it by using the `MPI_TYPE_COMMIT()` subroutine.
- ☞ If we wish to reuse the same name to define another derived datatype, we have to free it first with the `MPI_TYPE_FREE()` subroutine.

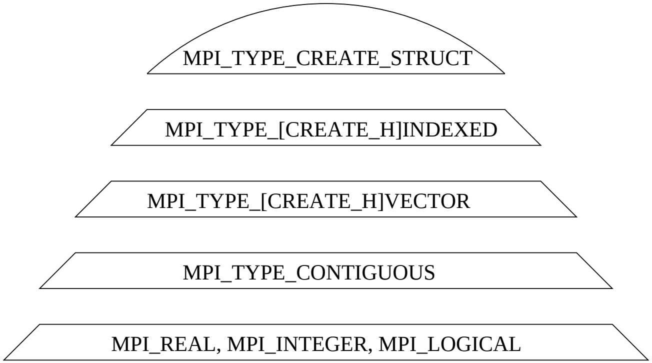

MPI_TYPE_CREATE_STRUCT
MPI_TYPE_[CREATE_H]INDEXED
MPI_TYPE_[CREATE_H]VECTOR
MPI_TYPE_CONTIGUOUS
MPI_REAL, MPI_INTEGER, MPI_LOGICAL

FIGURE 11 – Hierarchy of the MPI constructors

6 – Derived datatypes

6.2 – Contiguous datatypes

☞ MPI_TYPE_CONTIGUOUS() creates a data structure from a homogenous set of existing datatypes contiguous in memory.

1.	6.	11.	16.	21.	26.
2.	7.	12.	17.	22.	27.
3.	8.	13.	18.	23.	28.
4.	9.	14.	19.	24.	29.
5.	10.	15.	20.	25.	30.

```
call MPI_TYPE_CONTIGUOUS(5,MPI_REAL,new_type,code)
```

FIGURE 12 – MPI_TYPE_CONTIGUOUS subroutine

```
integer, intent(in)   : count, old_type
integer, intent(out)  : new_type,code

call MPI_TYPE_CONTIGUOUS(count,old_type,new_type,code)
```

6 – Derived datatypes

6.3 – Constant stride

☞ MPI_TYPE_VECTOR() creates a data structure from a homogenous set of existing data separated by a constant stride in memory. The stride is given by the number of **elements**.

1.	6.	11.	16.	21.	26.
2.	7.	12.	17.	22.	27.
3.	8.	13.	18.	23.	28.
4.	9.	14.	19.	24.	29.
5.	10.	15.	20.	25.	30.

```
call MPI_TYPE_VECTOR(6,1,5,MPI_REAL,new_type,code)
```

FIGURE 13 – MPI_TYPE_VECTOR subroutine

```
integer, intent(in)   : count,block_length
integer, intent(in)   : stride ! given in elements
integer, intent(in)   : old_type
integer, intent(out)  : new_type,code

call MPI_TYPE_VECTOR(count,block_length,stride,old_type,new_type,code)
```

6 – Derived datatypes

6.4 – Other subroutines

☞ Before using a new derived datatype, it is necessary to validate it by the MPI_TYPE_COMMIT() subroutine.

```
integer, intent(inout)  : new_type
integer, intent(out)    : code

call MPI_TYPE_COMMIT(new_type,code)
```

☞ The freeing of a derived datatype is made by using the MPI_TYPE_FREE() subroutine.

```
integer, intent(inout)  : new_type
integer, intent(out)    : code

call MPI_TYPE_FREE(new_type,code)
```

6 – Derived datatypes

6.5 – Homogenous datatypes of variable strides

☞ `MPI_TYPE_INDEXED()` allows to create a data structure composed of a sequence of blocks containing a variable number of elements separated by a variable stride in memory. The latter is given in **elements**.

☞ `MPI_TYPE_CREATE_HINDEXED()` has the same functionality as `MPI_TYPE_INDEXED()` except that the strides that separates two data blocks are given in **bytes**.
This subroutine is useful when the generic datatype is not an MPI base datatype(`MPI_INTEGER`, `MPI_REAL`, ...). We cannot therefore give the stride by the number of elements of the generic datatype.

☞ For `MPI_TYPE_CREATE_HINDEXED()`, as for `MPI_TYPE_CREATE_HVECTOR()`, use `MPI_TYPE_SIZE()` or `MPI_TYPE_GET_EXTENT()` in order to obtain in a portable way the size of the stride in bytes.

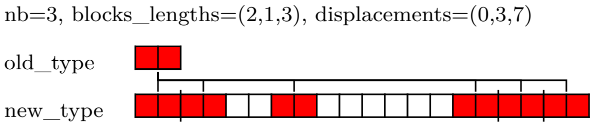


Figure 14 – The MPI_TYPE_INDEXED constructor

```
integer,intent(in)                  : nb
integer,intent(in),dimension(nb)    : :block_lengths
! Attention the displacements are given in elements
integer,intent(in),dimension(nb)    : displacements
integer,intent(in)                  : old_type

integer,intent(out)                 : new_type,code

call MPI_TYPE_INDEXED(nb,block_lengths,displacements,old_type,new_type,code)
```

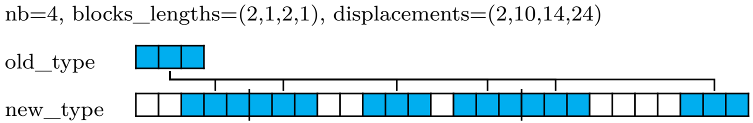


Figure 15 – The MPI_TYPE_CREATE_HINDEXED constructor

```
integer,intent(in)                                          : nb
integer,intent(in),dimension(nb)                            : :block_lengths
! Attention the displacements are given in bytes
integer(kind=MPI_ADDRESS_KIND),intent(in),dimension(nb)      : displacements
integer,intent(in)                                          : old_type

integer,intent(out)                                         : new_type,code

call MPI_TYPE_CREATE_HINDEXED(nb, block_lengths,displacements,
                              old_type,new_type,code)
```

6 – Derived datatypes

6.6 – Subarray Datatype Constructor

☞ The MPI_TYPE_CREATE_SUBARRAY() subroutine allows to create a subarray from an array.

```
integer,intent(in)                      : nb_dims
integer,dimension(ndims),intent(in)     : shape_array,shape_sub_array,coord_start
integer,intent(in)                      : order,old_type
integer,intent(out)                     : new_type,code
call MPI_TYPE_CREATE_SUBARRAY(nb_dims,shape_array,shape_sub_array,coord_start,
                              order,old_type,new_type,code)
```

Reminder of the vocabulary relative to the arrays in Fortran 95

- The rank of an array is its number of dimensions.
- The extent of an array is its number of elements in a dimension.
- The shape of an array is a vector whose each dimension is the extent of the array in the corresponding dimension.

For example the `T(10,0:5,-10:10)` array. Its rank is 3, its extent in the first dimension is 10, in the second 6 and in the third 21, its shape is the (10,6,21) vector.

- nb_dims : rank of the array
- shape_array : shape of the array from which a subarray will be extracted
- shape_sub_array : shape of the subarray
- coord_start : start coordinates if the indices of the array start at 0. For example, if we want that the start coordinates of the subarray be **`array(2,3)`**, we must have **`coord_start(:)=(/ 1,2 /)`**
- order : storage order of elements
  1. **`MPI_ORDER_FORTRAN`** for the ordering used by Fortran arrays (column-major order)
  2. **`MPI_ORDER_C`** for the ordering used by C arrays (row-major order)

7 – Optimisation

Point-to-Point Send Modes

Mode	Blocking	Non-blocking
Standard send	`MPI_Send`	`MPI_Isend`
Synchronous send	`MPI_Ssend`	`MPI_Issend`
Buffered send	`MPI_Bsend`	`MPI_Ibsend`
Ready send	`MPI_Rsend`	`MPI_Irsend`
Receive	`MPI_Recv`	`MPI_Irecv`

7 – Optimisation

Key Terms

- **Blocking call :** a call is blocking if the memory space used for the communication can be reused immediately after the exit of the call. The data that have been or will be sent are the data that were in this space at the moment of the call. If it is a receive, the data must have already been received in this space (if the return code is `MPI_SUCCESS`).
- **Non-blocking call :** a non-blocking call returns very quickly, but it does not authorize the immediate re-use of the memory space used in the communication. It is necessary to make sure that the communication is fully completed (with `MPI_Wait` for example) before using it again.
- **Synchronous send :** a synchronous send involves a synchronization between the involved processes. There can be no communication before the two processes are ready to communicate. A send cannot start until its receive is posted.
- **Buffered send :** a buffered send implies the copying of data in an intermediate memory space. There is then no coupling between the two processes of communication. So the output of this type of send does not mean that the receive occurred.

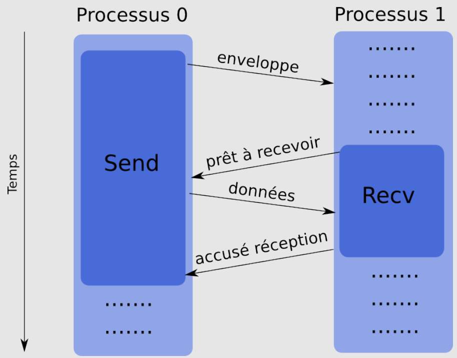
7 – Optimisation

Synchronous Sends

A synchronous send is made by calling the `MPI_Ssend` or `MPI_Issend` subroutine.

Rendezvous Protocol

The rendezvous protocol is generally the protocol used for synchronous sends (implementation-dependent). The return receipt is optional.



7 – Optimisation

Advantages

- Less use of resources (no buffer)
- Faster if the receiver is ready (no copying in a buffer)
- Guarantee of receive through synchronization

Disadvantages

- Waiting time if the receiver is not there/not ready
- Risks of deadlocks

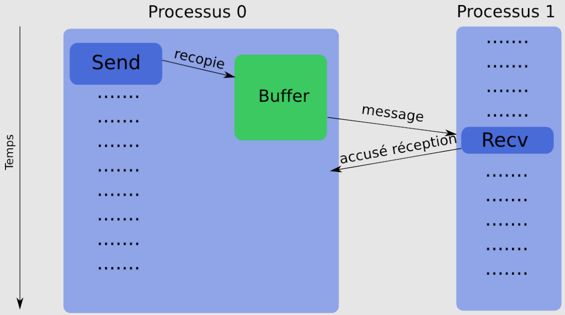
7 – Optimisation

Buffered Sends

A buffered send is made by calling the **`MPI_Bsend`** or **`MPI_Ibsend`** subroutine. The buffers have to be managed manually (with calls to **`MPI_Attach`** and **`MPI_Detach`**). They have to be allocated by taking into account the header size of messages (by adding the constant **`MPI_BSEND_OVERHEAD`** for each message instance).

Protocol with User Buffer on the Sender Side

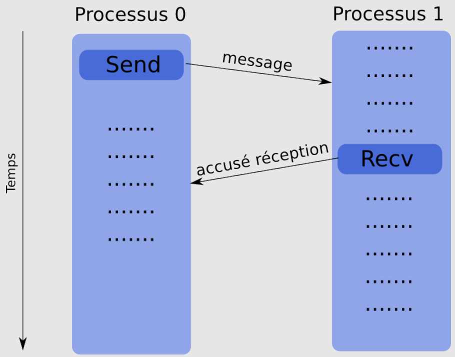
This approach is the one generally used for the **`MPI_Bsend`** or **`MPI_Ibsend`**. In this approach, the buffer is on the sender side and is managed explicitly by the application. A buffer managed by MPI can exist on the receiver side. Many variants are possible. The return receipt is optional.



7 – Optimisation

Eager Protocol

The eager protocol is often used for standard sends of small-size messages. It can also be used for sends with `MPI_Bsend` with small messages (implementation-dependent) and by bypassing the user buffer on the sender side. In this approach, the buffer is on the receiver side. The return receipt is optional.



7 – Optimisation

Advantages

- No need to wait for the receiver (copying in a buffer)
- No risks of deadlocks

Disadvantages

- Use of more resources (memory use by buffers with saturation risks)
- The used send buffers in the `MPI_Bsend` or `MPI_Ibsend` calls have to be managed manually (often hard to choose a suitable size)
- A little bit slower than the synchronous sends if the receiver is ready
- There is no guarantee of good receive (send-receive decoupling)
- Risk of wasted memory space if the buffers are too oversized
- There is often also hidden buffers managed by the MPI implementation on the sender side and/or on the receiver side (and using memory resources)

7 – Optimisation

Standard Sends

A standard send is made by calling the `MPI_Send` or `MPI_Isend` subroutine. In most implementations, this mode switches from a buffered mode to a synchronous mode when the size of messages grows.

Advantages

- Often the most efficient (because the constructor chose the best parameters and algorithms)
- The most portable for the performances

Disadvantages

- Little control over the really used mode (often accessible via environment variables)
- Risk of deadlock according to the actual mode
- Behavior that can vary according to the architecture and the problem size

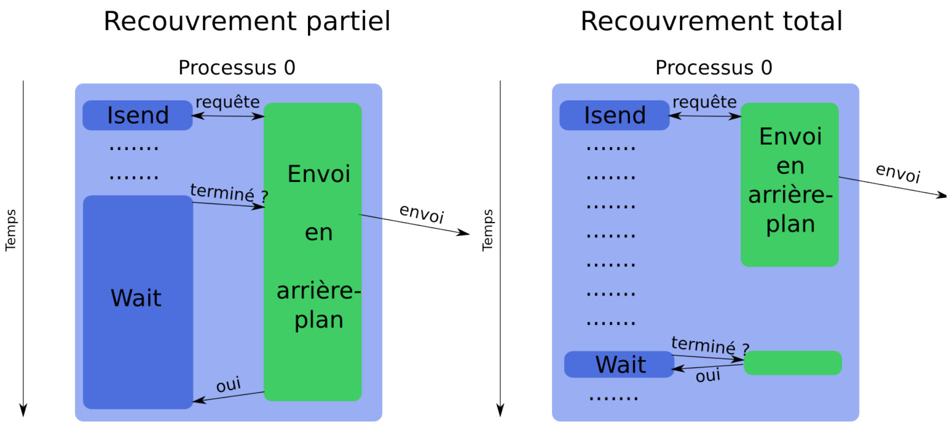
7 – Optimisation

Presentation

The overlap of communications by computations is a method which allows to execute communications operations in background while the program continues to operate.

- It is thus possible, if the hardware and software architecture allows it, to hide all or part of communications costs.
- The computation-communication overlap can be seen as an additional level of parallelism.
- This approach is used in MPI by the use of non-blocking subroutines (i.e. `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`).

7 – Optimisation



7 – Optimisation

Advantages

- Possibility of hiding all or part of communications costs (if the architecture allows it)
- No risks of deadlock

Disadvantages

- Greater additional costs (several calls for one single send or receive, management of requests)
- Higher complexity and more complicated maintenance
- Less efficient on some machines (for example with transfer starting only at the `MPI_Wait` call)
- Performance-loss risk on the computational kernels (for example differentiated management between the area near the border of a domain and the interior area resulting in less efficient use of memory caches)
- Limited to point-to-point communications (it will be extended to collective communications in MPI 3.0)

7 – Optimisation

Use

The message send is made in two steps :

- Initiate the send or the receive by a call to a subroutine beginning with `MPI_Isend` or `MPI_Irecv` (or one of their variants)
- Wait the end of the local contribution by a call to `MPI_Wait` (or one of its variants).

The communications overlap with all the operations that occur between these two steps. The access to data being in receive is not permitted before the end of the `MPI_Wait` (the access to data being in send is also not permitted for the MPI implementations previous to the 2.2 version).

7 – Optimisation

Example

```
do i=1,niter
 ! Initialize communications
  call MPI_Irecv(data_ext,  sz,MPI_REAL,dest,tag,comm, &
                 req(1),ierr)
  call MPI_Isend(data_bound,sz,MPI_REAL,dest,tag,comm, &
                 req(2),ierr)

 ! Compute the interior domain (data_ext and data_bound
 ! are unused) during communications
  call compute_interior_domain(data_int)

 ! Wait for the end of communications
  call MPI_Waitall(2,req,MPI_STATUSES_IGNORE,ierr)

 ! Compute the exterior domain
  call compute_exterior_domain(data_int,data_bound,data_ext)
end do
```

8 – Communicators
8.1 – Introduction

Communicators usage consists of partitioning a group of processes in order to create subgroups on which we can carry out operations such as collective or point-to-point communications. Each created subgroup will have its own communication space.

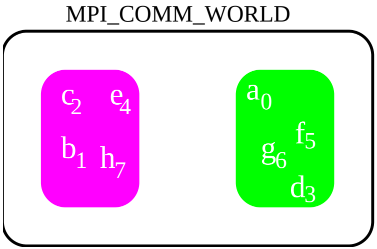


FIGURE 16 – Communicator partitioning

8 – Communicators

8.2 – Example

In the following example, we will :

☞ put together on one hand the even-ranked processes and on the other hand the odd-ranked processes ;

☞ broadcast a collective message only to even-ranked processes and another only to odd-ranked processes.

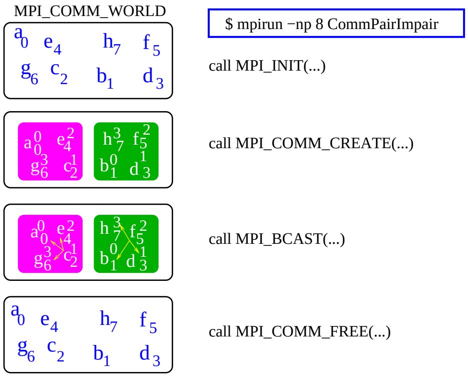


FIGURE 17 – Communicator creation/destruction

8 – Communicators

8.3 – Groups and Communicators

☞ A communicator consists :

① of a group, which is an ordered group of processes ;

② of a communication context made at the calling of the communicator construction subroutine, which allows to define the communication space.

☞ The communication contexts are managed by MPI (the programmer has no action on them : it is an opaque attribute).
In pratice, in order to build a communicator, there are two ways to do this :

① through a group of processes ;

② directly from another communicator.

The MPI_COMM_SPLIT() subroutine allows to partition a given communicator in as many communicators as we want...

```
integer, intent(in)   : comm, color, key
integer, intent(out)  : new_comm, code
call MPI_COMM_SPLIT(comm,color,key,new_comm,code)
```

process	a	b	c	d	e	f	g	h
rank_world	0	1	2	3	4	5	6	7
color	0	2	3	0	3	0	2	3
key	2	15	0	0	1	3	11	1
rank_new_com	1	1	0	0	1	2	0	2

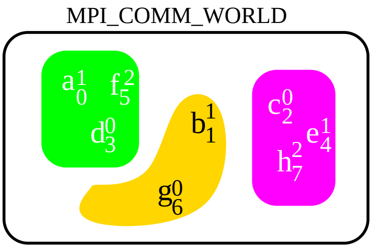


FIGURE 18 – Construction of communicators with MPI_COMM_SPLIT()

A process that is assigned a color equal to the MPI_UNDEFINED value will belong only to its initial communicator.

```fortran
program EvenOdd
  use mpi
  implicit none

  integer, parameter   : m=16
  integer              : key,CommEvenOdd
  integer              : rank_in_world,code
  real, dimension(m)   : a

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank_in_world,code)

   ! Initialization of the A vector
  a( :)=0.
  if(rank_in_world == 2) a( :)=2.
  if(rank_in_world == 5) a( :)=5.

  key = rank_in_world
  if (rank_in_world == 2 .OR. rank_in_world == 5 ) then
    key=-1
  end if

  ! Creation of even and odd communicators by giving them the same name
 call MPI_COMM_SPLIT(MPI_COMM_WORLD,mod(rank_in_world,2),key,CommEvenOdd,code)

  ! Broadcast of the message by the rank process 0 of each communicator to the processes
  ! of its group
  call MPI_BCAST(a,m,MPI_REAL,0,CommEvenOdd,code)

  ! Destruction of the communicators
 call MPI_COMM_FREE(CommEvenOdd,code)
 call MPI_FINALIZE(code)
end program EvenOdd
```

8 – Communicators
8.4 – Topologies

- In most applications, especially in domain decomposition methods where we match the calculation domain to the grid of processes, it is interesting to be able to arrange the processes according to a regular topology.
- MPI allows to define cartesian or graph virtual topologies.
  - Cartesian topologies :
    - each process is defined in a grid ;
    - the grid can be periodic or not ;
    - the processes are identified by their coordinates in the grid.
  - Graph Topologies :
    - generalization to more complex topologies.

8 – Communicators
8.4 – Topologies
8.4.1 – Cartesian topologies

☞ A cartesian topology is defined when a group of processes belonging to a given communicator comm_old calls the MPI_CART_CREATE() subroutine.

```
integer, intent(in)                      : comm_old, ndims
integer, dimension(ndims),intent(in)     : dims
logical, dimension(ndims),intent(in)     : periods
logical, intent(in)                      : reorganization

integer, intent(out)                     : comm_new, code

call MPI_CART_CREATE(comm_old, ndims,dims,periods,reorganization,comm_new,code)
```

☞ Example on a grid having 4 domains along x and 2 along y, periodic in y.

```
use mpi
integer                      : comm_2D, code
integer, parameter           : ndims = 2
integer, dimension(ndims)    : dims
logical, dimension(ndims)    : periods
logical                      : reorganization

...............................................

dims(1) = 4
dims(2) = 2
periods(1) = .false.
periods(2) = .true.
reorganization = .false.

call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganization,comm_2D,code)
```

☞ If `reorganization = .false.` then the rank of the processes in the new communicator (comm_2D) is the same as in the old communicator (`MPI_COMM_WORLD`). If `reorganization = .true.`, the MPI implementation chooses the order of the processes.

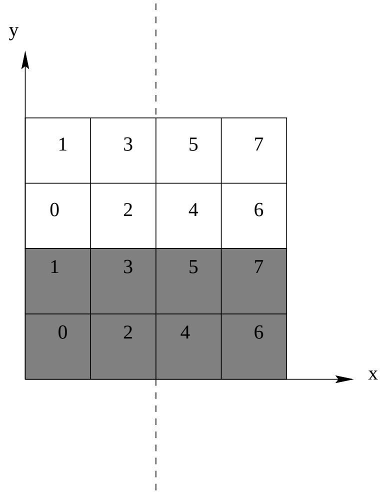


FIGURE 19 – 2D periodic cartesian topology in y

☞ The MPI_DIMS_CREATE() subroutine returns the number of processes in each dimension of the grid according to the total number of processes.

```
integer, intent(in)                        : nb_procs, ndims
integer, dimension(ndims),intent(inout)    : dims
integer, intent(out)                       : code

call MPI_DIMS_CREATE(nb_procs,ndims,dims,code)
```

☞ Remark : if the values of dims in entry are all 0, this means that we leave to MPI the choice of the number of processes in each direction according to their total number.

dims in entry	call MPI_DIMS_CREATE	dims en exit
(0,0)	(8,2,dims,code)	(4,2)
(0,0,0)	(16,3,dims,code)	(4,2,2)
(0,4,0)	(16,3,dims,code)	(2,4,2)
(0,3,0)	(16,3,dims,code)	error

☞ In a cartesian topology, a process that calls the MPI_CART_SHIFT() subroutine can get the rank of its neighboring processes in a given direction.

```
integer, intent(in)   : comm_new, direction, step

integer, intent(out)  : rank_previous,rank_next
integer, intent(out)  : code

call MPI_CART_SHIFT(comm_new, direction, step, rank_previous, rank_next, code)
```

☞ The **direction** parameter corresponds to the displacement axis (xyz).

☞ The **step** parameter corresponds to the displacement step.

☞ Program Example :

```
program decomposition
  use mpi
  implicit none

  integer                         : rank_in_topo,nb_procs
  integer                         : code,comm_2D
  integer, dimension(4)           : neighbor
  integer, parameter              : N=1,E=2,S=3,W=4
  integer, parameter              : ndims = 2
  integer, dimension (ndims)      : dims,coords
  logical, dimension (ndims)      : periods
  logical                         : reorganization

  call MPI_INIT(code)

  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)

  ! Know the number of processes along x and y
  dims( :) = 0

  call MPI_DIMS_CREATE(nb_procs,ndims,dims,code)
  ! 2D y-periodic grid creation
  periods(1) = .false.
  periods(2) = .true.
  reorganization = .false.

  call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganization,comm_2D,code)

  ! Know my coordinates in the topology
  call MPI_COMM_RANK(comm_2D,rank_in_topo,code)
  call MPI_CART_COORDS(comm_2D,rank_in_topo,ndims,coords,code)

  ! Initialization of the neigboring array to the MPI_PROC_NULL value
  neighbor( :) = MPI_PROC_NULL

  ! Search of my West and East neigbors
  call MPI_CART_SHIFT(comm_2D,0,1,neighbor(W),neighbor(E),code)

  ! Search of my South and North neighbors
  call MPI_CART_SHIFT(comm_2D,1,1,neighbor(S),neighbor(N),code)

  call MPI_FINALIZE(code)

end program decomposition
```

9 – MPI-IO
9.1 – Introduction
9.1.1 – Presentation

☞ Very logically, the applications that make large calculations also handle large amounts of data, and generate therefore a significant number of I/O.

☞ Thus, their effective treatment sometimes affects very strongly the global performances of applications.

- The I/O optimization of parallel codes is made by the combination :
  - of their parallelization, in order to avoid creating a bottleneck due to their serialization ;
  - of explicitly implemented techniques at the level of programming (nonblocking reads / writes) ;
  - of specific operations supported by the operating system (grouping of requests, buffer management of I/O, etc.).
- The goals of MPI-IO, via the high-level interface that it proposes, are to provide simplicity, expressivity and flexibility, while authorizing performing implementations that take into account the software and hardware specificities of I/O devices of the target machines.
- MPI-IO provides an interface modeled on the one used for message passing. The definition of data accessed according to the processes is made by the use of (basic or derived) datatypes. As for the notions of nonblocking and collective operations, they are managed similarly to what MPI proposes for the messages.
- MPI-IO authorises both sequential and random accesses.

9 – MPI-IO
9.2 – File management

- ☞ The file management tasks are collective operations made by all the processes of the indicated communicator.
- ☞ We are only describing here the principal subroutines (opening, closing) but others are available (deletion, etc.).
- ☞ The attributes (describing the access rights, the opening mode, the possible destruction at the closing, etc.) must be precised by sum on predefined constants.
- ☞ All the processes of the communicator inside of which a file is open will participate in the later collective operations of data access.
- ☞ The opening of a file returns a file handle, which will be later used in all the operations relative to this file.
- ☞ The available information via the `MPI_FILE_SET_INFO()` subroutine varies from one implementation to another.

TABLE 3 – Attributes that can be positioned during the opening of files

Attribut	Meaning
MPI_MODE_RDONLY	read only
MPI_MODE_RDWR	reading and writing
MPI_MODE_WRONLY	write only
MPI_MODE_CREATE	create the file if it does not exist
MPI_MODE_EXCL	error if the file exists
MPI_MODE_UNIQUE_OPEN	error if the file is already open by another application
MPI_MODE_SEQUENTIAL	sequential access
MPI_MODE_APPEND	pointers at the end of file (add mode)
MPI_MODE_DELETE_ON_CLOSE	delete after the closing

```
program open01

  use mpi
  implicit none

  integer  : fh,code

  call MPI_INIT(code)

  call MPI_FILE_OPEN(MPI_COMM_WORLD,"file.data", &
                    MPI_MODE_RDWR + MPI_MODE_CREATE,MPI_INFO_NULL,fh,code)

  call MPI_FILE_CLOSE(fh,code)
  call MPI_FINALIZE(code)

end program open01
```

```
> ls -l file.data

-rw-------   1 name       grp    0 Feb 08 12 :13 file.data
```

9 – MPI-IO

9.3 – Reads/Writes : general concepts

- The data transfers between files and memory areas of processes are made via explicit calls to read and write subroutines.
- We distinguish three aspects to file access :
  - the positioning, which can be explicit (by specifying for example the desired number of bytes from the beginning of the file) or implicit, via pointers managed by the system (these pointers can be of two types : either individual to each process, or shared by all the processes) ;
  - the synchronism, the accesses can be blocking or nonblocking ;
  - the coordination, the accesses can be collective (that is to say made by all the processes of the communicator inside of which the file is opened) or specific only to one or many processes.
- There are many available variants : we will describe some of them.

TABLE 4 – Summary of possible access types

Positioning	**Synchronism**	**Coordination**	
		individual	*collective*
explicit offsets	blocking	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL
		MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL
	nonblocking	MPI_FILE_IREAD_AT	MPI_FILE_READ_AT_ALL_BEGIN
			MPI_FILE_READ_AT_ALL_END
		MPI_FILE_IWRITE_AT	MPI_FILE_WRITE_AT_ALL_BEGIN
			MPI_FILE_WRITE_AT_ALL_END

see next page

Positioning	**Synchronism**	**Coordination** *individual*	**Coordination** *collective*
individual file pointers	blocking	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	nonblocking	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
shared file pointers	blocking	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	nonblocking	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

- It is possible to mix the access types performed at the same file inside an application.
- The accessed memory areas are described by three quantities :
  - the initial address of the concerned area ;
  - the number of elements ;
  - the datatype, which must match a sequence of contiguous copies of the etype of the current "view".

9 – MPI-IO

9.4 – Definition of views

- The views are a flexible and powerful mechanism for describing the accessed areas in the files.
- The views are constructed by the help of MPI derived datatypes.
- Each process has its own view (or its own views) of a file, defined by three variables : a displacement, an etype and a filetype. A view is defined as a repetition of the filetype, once the initial positioning is made.
- It is possible to define holes in a view, by not taking into account some data parts.
- Different processes can perfectly have different views of the file, in order to access complementary parts of it.
- A given process can define and use many different views of the same file.
- A shared pointer may be used with a view only if all the processes have the same view.

- If the file is open for writing, the described areas by the etypes and the filetypes cannot overlap, even partially.
- The default view consists of a simple sequence of bytes (zero initial displacement, etype and filetype equal to `MPI_BYTE`).

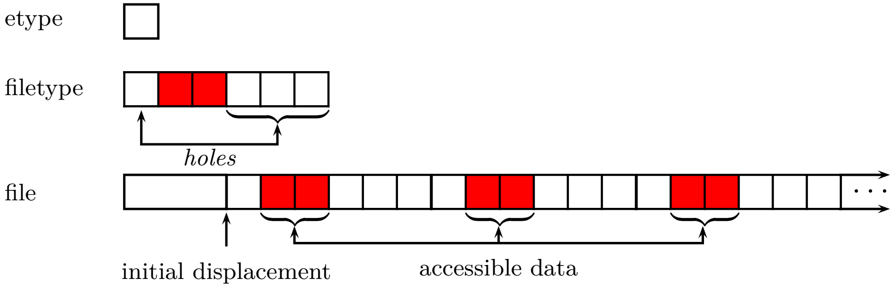


Figure 20 – etype and filetype

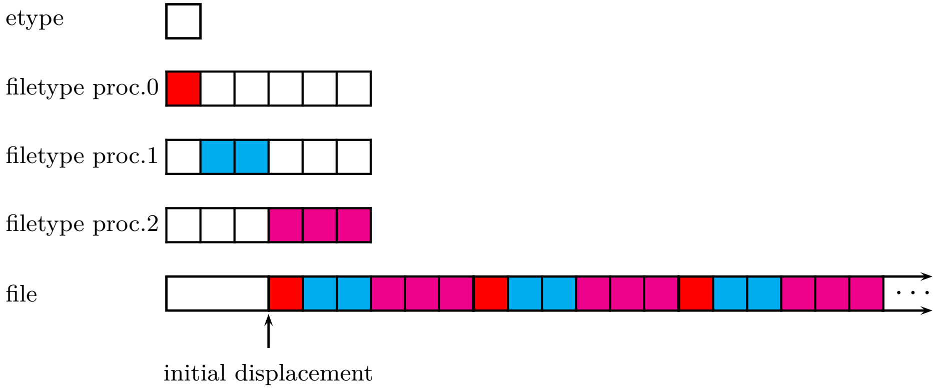


FIGURE 21 – Example of definition of different filetypes according to the processes

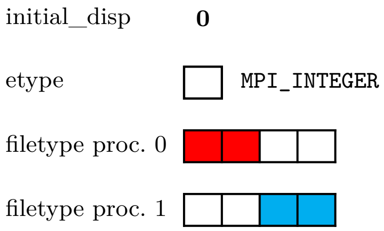


FIGURE 22 – Filetype used in example 2 of `MPI_FILE_SET_VIEW()`

```
program read_view02

  use mpi
  implicit none

  integer, parameter                       : nb_values=10
  integer                                  : rank,fh,coord,filetype,code
  integer(kind=MPI_OFFSET_KIND)             : initial_displacement
  integer, dimension(nb_values)            : values
  integer, dimension(MPI_STATUS_SIZE)       : status
  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)

  call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
                     fh,code)

  if (rank == 0) then
     coord=1
  else
     coord=3
  end if

  call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/coord - 1/), &
                                MPI_ORDER_FORTRAN,MPI_INTEGER,filetype,code)
  call MPI_TYPE_COMMIT(filetype,code)

  initial_displacement=0
  call MPI_FILE_SET_VIEW(fh,initial_displacement,MPI_INTEGER,filetype, &
                         "native",MPI_INFO_NULL,code)

  call MPI_FILE_READ(fh,values,nb_values,MPI_INTEGER,status,code)

  print *, "Read process",rank," :",values( :)

  call MPI_FILE_CLOSE(fh,code)
  call MPI_FINALIZE(code)

end program read_view02
```

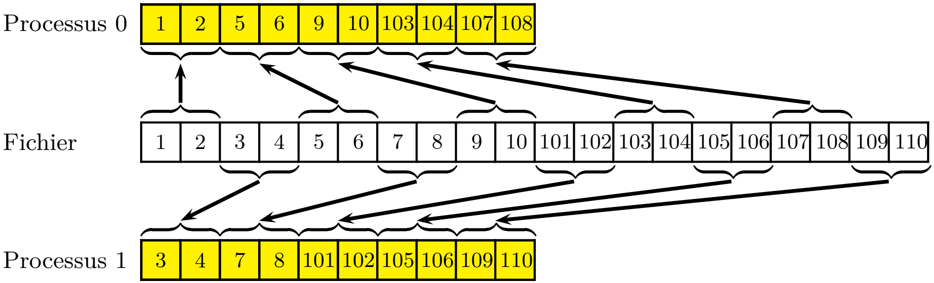


FIGURE 23 – Example 2 of `MPI_FILE_SET_VIEW()`

```
mpiexec -n 2 read_view02

Read process 1  :3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0  :1, 2, 5, 6,   9,  10, 103, 104, 107, 108
```

9 – MPI-IO

9.5 – NonBlocking Reads/Writes

- ☞ The nonblocking I/O are implemented according to the model used for the nonblocking communications.
- ☞ A nonblocking access must later lead to an explicit test of completeness or to a standby (via `MPI_TEST()`, `MPI_WAIT()`, etc.), in a way similar to the management of nonblocking messages.
- ☞ The advantage is to make an overlap between the computations and the I/O.

10 – Conclusion

- ☞ Use blocking point-to-point communications, this before going to nonblocking communications. It will be necessary then to try to make computations/communications overlap.
- ☞ Use the blocking I/O functions, this before going to nonblocking I/O. Similarly, it will be necessary then to make I/O-computations overlap.
- ☞ Write the communications as if the sendings were synchronous (`MPI_SSEND()`).
- ☞ Avoid the synchronization barriers (`MPI_BARRIER()`), especially on the blocking collective functions.
- ☞ The MPI/OpenMP hybrid programming can bring gains of scalability, in order for this approach to function well, it is obviously necessary to have good OpenMP performances inside each MPI process. A course is given at IDRIS (`https://cours.idris.fr/`).