



# IDRIS

## MPI

Dimitri Lecas - Rémi Lacroix - Serge Van Criekingen - Myriam Peyrounette

*CNRS — IDRIS*

v5.3 March 13th 2025



# Plan I

## Introduction

- Availability and updating
- Introduction
- Concept of message passing
- Distributed memory
- History
- Library

## Environment

## Point-to-point Communications

- General Concepts
- Blocking send and receive
- Predefined MPI Datatypes
- Other Possibilities

## Collective communications

- General concepts
- Global synchronization : `MPI_Barrier()`
- Global distribution : `MPI_Bcast()`
- Selective distribution : `MPI_Scatter()`
- Collection : `MPI_Gather()`
- Gather-to-all : `MPI_Allgather()`
- Extended gather : `MPI_Gatherv()`

## Plan II

Collection and distribution : `MPI_Alltoall()`

Global reduction

Additions

## Communication Modes

Point-to-Point Send Modes

Blocking call

- Synchronous Sends

- Buffered sends

- Standard sends

Number of received elements

Nonblocking communication

One-Sided Communications

## Derived datatypes

Introduction

Contiguous datatypes

Constant stride

Commit derived datatypes

Examples

- The datatype "matrix row"

- The datatype "matrix line"

- The datatype "matrix block"

Homogenous datatypes of variable strides

## Plan III

- Size of MPI datatype
- Heterogenous datatypes
- Conclusion
- Memento

## Communicators

- Introduction
- Example
- Default communicator
- Groups and communicators
- Partitioning of a communicator
- Topologies
  - Cartesian topologies
  - Subdividing a Cartesian topology

## MPI-IO

- Introduction
- File Manipulation
- Data access : Concepts
- Noncollective data access
  - Data access with explicit offsets
  - Data access with individual file pointers
  - Data access with shared file pointers
- Collective data access

## Plan IV

- Data access with explicit offsets
- Data access with individual file pointers
- Data access with shared file pointers

### Positioning the file pointers

### Nonblocking Data Access

- Data Access with Explicit Offsets
- Data access with individual file pointers
- Nonblocking and collective data access routines

## MPI 4.x

### MPI-IO Views

- Definition
- Subarray datatype constructor
- Reading non-overlapping sequences of data segments in parallel
- Reading data using successive views
- Dealing with holes in datatypes

## Conclusion

# Introduction

# Introduction

## Availability and updating

This document is likely to be updated regularly. The most recent version is available on the Web server of IDRIS : <http://www.idris.fr/formations/mpi/>

- IDRIS  
Institut for Development and Resources in Intensive Scientific Computing  
Rue John Von Neumann  
Bâtiment 506  
BP 167  
91403 ORSAY CEDEX  
France  
<http://www.idris.fr>
- Translated with the help of Cynthia TAUPIN.

# Introduction

## Parallelism

The goal of parallel programming is to :

- Reduce elapsed time.
- Do larger computations.
- Exploit parallelism of modern processor architectures (multicore, multithreading).

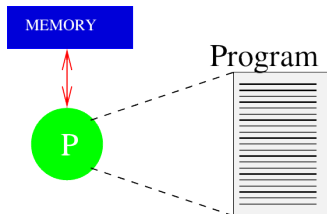
For group work, coordination is required. [MPI](#) is a library which allows process coordination by using a message-passing paradigm.



# Introduction

## 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.

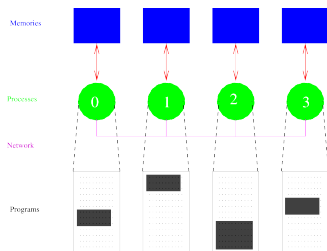


**Figure 1** – Sequential programming model

# Introduction

## Message passing programming model

- The program is written in a classic language (Fortran, C, C++, etc.).
- All the program variables are private and reside in the local memory of each process.
- Each process has the possibility of executing different parts of a program.
- A variable is exchanged between two or several processes via a programmed call to specific subroutines.

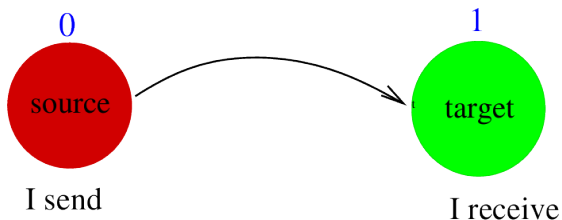


**Figure 2** – Message Passing Programming Model

# Introduction

## Message Passing concepts

If a message is sent to a process, the process must receive it.



**Figure 3** – Message Passing

# Introduction

## Message content

- A message consists of data chunks passing from the sending process to the receiving process/processes.
- In addition to the data (scalar variables, arrays, etc.) to be sent, a message must contain the following information :
  - The identifier of the sending process
  - The datatype
  - The length
  - The identifier of the receiving process

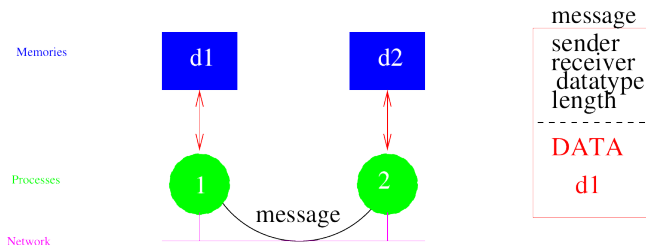


Figure 4 – Message Construction

# Introduction

## Environment

- The exchanged messages are interpreted and managed by an environment comparable to telephony, e-mail, postal mail, etc.
- The message is sent to a specified address.
- The receiving process must be able to classify and interpret the messages which are sent to it.
- The environment in question is MPI (Message Passing Interface). An MPI application is a group of autonomous processes, each executing its own code and communicating via calls to MPI library subroutines.

# Introduction

## Supercomputer architecture

Most supercomputers are distributed-memory computers. They are made up of many nodes and memory is shared within each node.

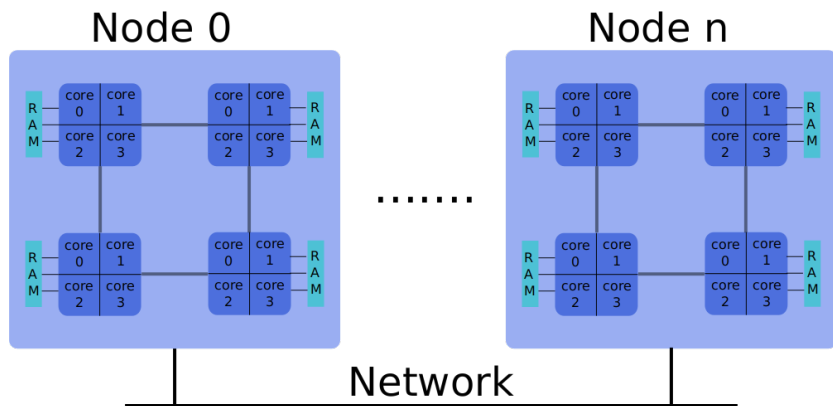


Figure 5 – Supercomputer architecture

# Introduction

## Jean Zay

- 1 116 nodes
- 2 Intel Cascade Lake processor (20 cores), 2,5 Ghz by node
- 4 GPU Nvidia V100 by node (on 391 nodes)
- 44 640 cores
- 214 TB (192 GB by node)



# Introduction

## MPI vs OpenMP

OpenMP uses a shared memory paradigm, while MPI uses a distributed memory paradigm.

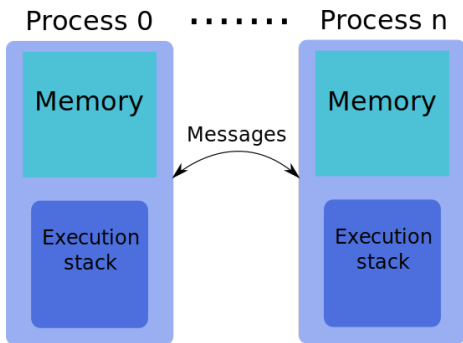


Figure 6 – MPI scheme

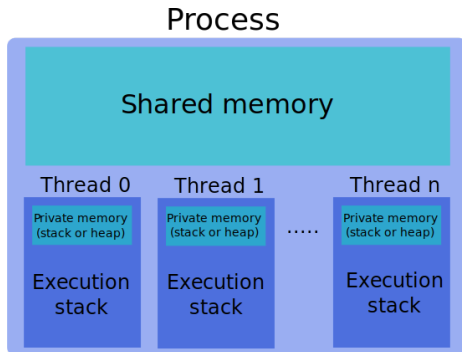


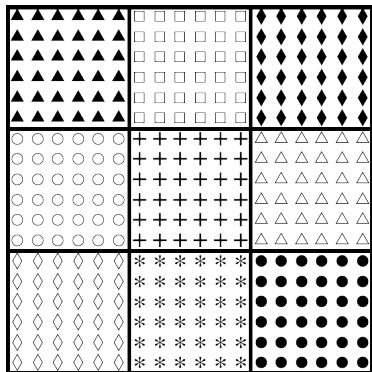
Figure 7 – OpenMP scheme



# Introduction

## Domain decomposition

A schema that we often see with **MPI** is domain decomposition. Each process controls a part of the global domain and mainly communicates with its neighbouring processes.



**Figure 8** – Decomposition in subdomains

# Introduction

## History

- **Version 1.0** : June 1994, the MPI (Message Passing Interface) Forum, with the participation of about forty organisations, developed the definition of a set of subroutines concerning the MPI library.
- **Version 1.1** : June 1995, only minor changes.
- **Version 1.2** : 1997, minor changes for more consistency in the names of some subroutines.
- **Version 1.3** : September 2008, with clarifications of the MPI 1.2 version which are consistent with clarifications made by MPI-2.1.
- **Version 2.0** : Released in July 1997, important additions which were intentionally not included in MPI 1.0 (process dynamic management, one-sided communications, parallel I/O, etc.).
- **Version 2.1** : June 2008, with clarifications of the MPI 2.0 version but without any changes.
- **Version 2.2** : September 2009, with only "small" additions.
- **Version 3.0** : September 2012, add nonblocking collective communications, new fortran bindings, etc.
- **Version 3.1** : June 2015 with clarifications and small additions.

# Introduction

## MPI 4.0

Version 4.0 : June 2021

- Large count
- Partitioned communication
- MPI Session

Version 4.1 : November 2023

# Introduction

## Library

- Website of MPI Forum <http://www.mpi-forum.org>
- Standard available in PDF on <http://www.mpi-forum.org/docs/>
- William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI, third edition Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 2014.
- William Gropp, Torsten Hoefler, Rajeev Thakur and Erwing Lusk : *Using Advanced MPI Modern Features of the Message-Passing Interface*, MIT Press, 2014.
- Victor Eijkhout : The Art of HPC <http://theartofhpc.com>

# Introduction

## Open source MPI implementations

These can be installed on a large number of architectures but their performance results are generally inferior to the implementations of the constructors.

- **MPICH** : <http://www.mpich.org>
- **Open MPI** : <http://www.open-mpi.org>

# Introduction

## Tools

- Debuggers
  - Totalview  
<https://totalview.io>
  - DDT  
<https://www.linaroforge.com/linaro-ddt>
- Performance measurement
  - **FPMPI** : *FPMPI*  
<http://www.mcs.anl.gov/research/projects/fpmapi/WWW/>
  - **Scalasca** : *Scalable Performance Analysis of Large-Scale Applications*  
<http://www.scalasca.org>
  - **MUST** : *MPI Runtime Correctness Analysis*  
<https://itc.rwth-aachen.de/must/>

# Introduction

## Open source parallel scientific libraries

- **ScaLAPACK** : Linear algebra problem solvers using direct methods.  
<http://www.netlib.org/scalapack/>
- **PETSc** : Linear and non-linear algebra problem solvers using iterative methods.  
<https://petsc.org/release/>
- **PaStiX** : Parallel sparse direct Solvers.  
<https://solverstack.gitlabpages.inria.fr/pastix/>
- **FFTW** : Fast Fourier Transform.  
<http://www.fftw.org>
- **HDF5** : Read and write on files.  
<https://www.hdfgroup.org/solutions/hdf5/>

# Environment



# Environment

## Description

- Every program unit calling MPI subroutines has to include a header file. In Fortran, we use the `mpi_f08` module introduced in MPI-3. Before in MPI-2, we used the *module* `mpi`, and in MPI-1, it was the `mpif.h` file).
- The `MPI_Init()` subroutine initializes the MPI environment :

```
MPI_INIT (code)
integer, optional, intent(out) :: code
```

- The `MPI_Finalize()` subroutine disables this environment :

```
MPI_FINALIZE (code)
integer, optional, intent(out) :: code
```

# Environment

## Differences between C/C++ and Fortran

In a C/C++ program :

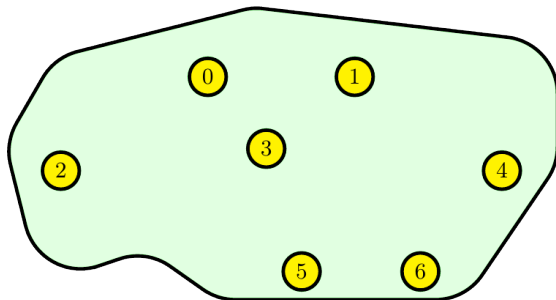
- you need to include the header file `mpi.h`;
- the `code` argument is the return value of MPI subroutines ;
- except for `MPI_Init()`, the function arguments are identical to Fortran ;
- the syntax of the subroutines changes : only the MPI prefix and the first following letter are in upper-case letters.

```
int MPI_Init(int *argc, char ***argv);  
int MPI_Finalize(void);
```

# Environment

## Communicators

- All the MPI operations occur in a defined set of processes, called **communicator**. The default communicator is `MPI_COMM_WORLD`, which includes all the active processes.



**Figure 9** – MPI\_COMM\_WORLD Communicator

# Environment

## Termination of a program

Sometimes, a program encounters some issue during its execution and has to stop prematurely. For example, we want the execution to stop if one of the processes cannot allocate the memory needed for its calculation. In this case, we call the `MPI_Abort()` subroutine instead of the Fortran instruction *stop* (Or *exit* in C).

```
MPI_ABORT(comm, erreur, code)

TYPE(MPI_Comm), intent(in)      :: comm
integer, intent(in)             :: error
integer, optional, intent(out)  :: code
```

- `comm` : the communicator of which all the processes will be stopped ; it is advised to use `MPI_COMM_WORLD` in general ;
- `error` : the error number returned to the UNIX environment.

## Code

It is not necessary to check the `code` value (return value in C) after calling MPI routines. By default, when MPI encounters a problem, the program is automatically stopped as in an implicit call to `MPI_Abort()` subroutine.

# Environment

## Rank and size

- At any moment, we have access to the number of processes managed by a given communicator by calling the `MPI_Comm_size()` subroutine :

```
MPI_COMM_SIZE(comm,nb_procs,code)

TYPE(MPI_Comm)           :: comm
integer, intent(out)     :: nb_procs
integer, optional, intent(out) :: code
```

- Similarly, the `MPI_Comm_rank()` subroutine allows us to obtain the rank of an active process (i.e. its instance number, between 0 and `MPI_Comm_size() - 1`) :

```
MPI_COMM_RANK(comm,rank,code)

TYPE(MPI_Comm), intent(in)  :: comm
integer, intent(out)       :: rank
integer, optional, intent(out) :: code
```

# Environment

## Example

```
1 program who_am_I
2   use mpi_f08
3   implicit none
4   integer :: nb_procs,rank
5
6   call MPI_INIT()
7
8   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
9   call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
10
11  print *,'I am the process ',rank,' among ',nb_procs
12
13  call MPI_FINALIZE()
14 end program who_am_I
```

```
> mpiexec -n 7 who_am_I
```

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

## Compilation and execution of an MPI code

- To **compile** an MPI code, we use a compiler *wrapper*, which makes the link with the chosen MPI library.
- This *wrapper* is different depending on the programming language, the compiler and the MPI library. For example : `mpif90`, `mpifort`, `mpicc`, ...

```
> mpif90 <options> -c source.f90  
> mpif90 source.o -o my_executable_file
```

- To **execute** an MPI code, we use an MPI launcher, which runs the execution on a given number of processes.
- The `mpiexec` launcher is defined by the MPI standard. There are also non-standard launchers, such as `mpirun`.

```
> mpiexec -n <number of processes> my_executable_file
```

## MPI Hands-On – Exercise 1 : MPI Environment

- Write an MPI program in such a way that each process prints a message, which indicates whether its rank is **odd** or **even**. For example :

```
> mpiexec -n 4 ./even_odd
I am process 0, my rank is even
I am process 2, my rank is even
I am process 3, my rank is odd
I am process 1, my rank is odd
```

- To test whether the rank is odd or even, the Fortran intrinsic function corresponding to the *modulo* operation is **mod** :

```
mod(a, b)
```

(use **%** symbol in C : `a%b`)

- To compile your program, use the command **make**
- To execute your program, use the command **make exe**
- For the program to be recognized by the Makefile, it must be named **even\_odd.f90** (or **even\_odd.c**)

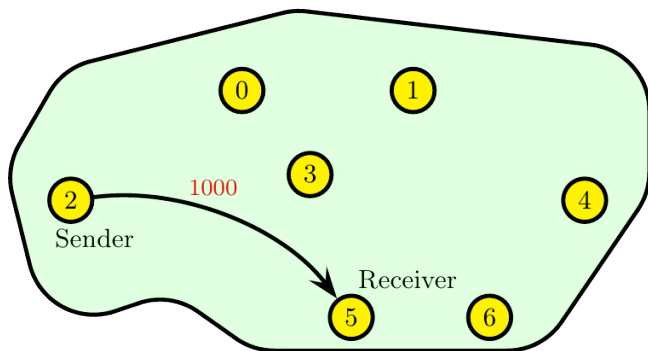


# Point-to-point Communications

# Point-to-point Communications

## General Concepts

A **point-to-point** communication occurs between two processes : the **sender** process and the **receiver** process.



**Figure 10** – Point-to-point communication

# Point-to-point Communications

## General Concepts

- The sender and the receiver are identified by their **ranks** in the communicator.
- The object communicated from one process to another is called **message**.
- A message is defined by its **envelope**, which is composed of :
  - the rank of the sender process
  - the rank of the receiver process
  - the message tag
  - the communicator in which the transfer occurs
- The exchanged data has a **datatype** (integer, real, etc, or individual derived datatypes).
- There are several transfer **modes**, which use different protocols.
- If two messages are sent with the same envelope, the order of receipt and sending are the same.

# Point-to-point Communications

## Blocking Send `MPI_Send`

```
MPI_SEND(buf, count, datatype, dest, tag, comm, code)

TYPE(*), dimension(..), intent(in) :: buf
integer, intent(in)                :: count, dest, tag
TYPE(MPI_Datatype), intent(in)     :: datatype
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

Sending, from the address `buf`, a message of `count` elements of type `datatype`, tagged `tag`, to the process of rank `dest` in the communicator `comm`.

### Remark :

This call is blocking : the execution remains blocked until the message can be re-written without risk of overwriting the value to be sent. In other words, the execution is blocked as long as the message has not been received.

# Point-to-point Communications

## Blocking Receive `MPI_Recv`

```
MPI_RECV(buf, count, datatype, source, tag, comm, status_msg, code)

TYPE(*), dimension(..), intent(in) :: buff
integer, intent(in)                :: count, source, tag
TYPE(MPI_Datatype), intent(in)     :: datatype
TYPE(MPI_Comm), intent(in)         :: comm
TYPE(MPI_Status)                   :: status_msg
integer, optional, intent(out)     :: code
```

Receiving, at the address `buf`, a message of `count` elements of type `datatype`, tagged `tag`, from the process of rank `source` in the communicator `comm`.

### Remarks :

- `status_msg` stores the state of a receive operation : source, tag, code, ... .
- An `MPI_Recv` can only be associated to an `MPI_Send` if these two calls have the same envelope (`source`, `dest`, `tag`, `comm`).
- This call is blocking : the execution remains blocked until the message content corresponds to the received message.

# Point-to-point Communications

## Example (see Fig. 10)

```
1 program point_to_point
2   use mpi_f08
3   implicit none
4
5   TYPE(MPI_Status)    :: status_msg
6   integer, parameter  :: tag=100
7   integer             :: rank,value
8
9   call MPI_INIT()
10
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
12
13  if (rank == 2) then
14    value=1000
15    call MPI_SEND(value,1,MPI_INTEGER,5,tag,MPI_COMM_WORLD)
16  elseif (rank == 5) then
17    call MPI_RECV(value,1,MPI_INTEGER,2,tag,MPI_COMM_WORLD,status_msg)
18    print *, 'I, process 5, I received ',value,' from the process 2.'
19  end if
20
21  call MPI_FINALIZE()
22
23  end program point_to_point
```

```
> mpiexec -n 7 point_to_point
```

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

# Point-to-point Communications

## Fortran MPI Datatypes

MPI Type	Fortran Type
<code>MPI_INTEGER</code>	INTEGER
<code>MPI_REAL</code>	REAL
<code>MPI_DOUBLE_PRECISION</code>	DOUBLE PRECISION
<code>MPI_COMPLEX</code>	COMPLEX
<code>MPI_LOGICAL</code>	LOGICAL
<code>MPI_CHARACTER</code>	CHARACTER
<code>MPI_BYTE</code>	

# Point-to-point Communications

## Fortran MPI Datatype

- The Fortran 95 language introduces two intrinsic functions `selected_int_kind()` and `selected_real_kind()` which make it possible to define precision and/or the range of an integer, real or complex number.
- MPI provides portability of these data types with `MPI_TYPE_CREATE_F90_INTEGER()`, `MPI_TYPE_CREATE_F90_REAL()` and `MPI_TYPE_CREATE_F90_COMPLEX()`

```
MPI_TYPE_CREATE_F90_INTEGER(r, newtype, code)
```

```
INTEGER, INTENT(IN)           :: r  
TYPE(MPI_Datatype), INTENT(OUT) :: newtype  
INTEGER, OPTIONAL, INTENT(OUT) :: code
```

```
MPI_TYPE_CREATE_F90_REAL(p, r, newtype, code)
```

```
INTEGER, INTENT(IN)           :: p, r  
TYPE(MPI_Datatype), INTENT(OUT) :: newtype  
INTEGER, OPTIONAL, INTENT(OUT) :: code
```

```
! Kind for double precision  
integer, parameter :: dp = selected_real_kind(15,307)  
! Kind for long int  
integer, parameter :: li = selected_int_kind(15)  
integer(kind=li)    :: nbbloc  
real(kind=dp)       :: width  
call MPI_TYPE_CREATE_F90_INTEGER(15,type1i)  
call MPI_TYPE_CREATE_F90_REAL(15,307,type1i)
```



# Point-to-point Communications

## Other possibilities

- When receiving a message, the rank of the sender process and the tag can be replaced by « *jokers* » : `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, respectively.
- A communication involving the dummy process of rank `MPI_PROC_NULL` has no effect.
- `MPI_STATUS_IGNORE` is a predefined constant, which can be used instead of the status variable.
- It is possible to send more complex data structures by creating **derived datatypes**.
- There are other operations, which carry out both send and receive operations **simultaneously** : `MPI_Sendrecv()` and `MPI_Sendrecv_replace()`.

# Point-to-point Communications

## Simultaneous send and receive `MPI_Sendrecv`

```
MPI_SENDRECV(sendbuf, sendcount, sendtype,
              dest, sendtag,
              recvbuf, recvcount, recvtype,
              source, recvtag, comm, status_msg, code)

TYPE(*), dimension(..), intent(in) :: sendbuf
TYPE(*), dimension(..)             :: recvbuf
integer, intent(in)                 :: sendcount, recvcount
integer, intent(in)                 :: source, dest, sendtag, recvtag
TYPE(MPI_Datatype), intent(in)      :: sendtype, recvtype
TYPE(MPI_Comm), intent(in)          :: comm
TYPE(MPI_Status)                    :: status_msg
integer, optional, intent(out)      :: code
```

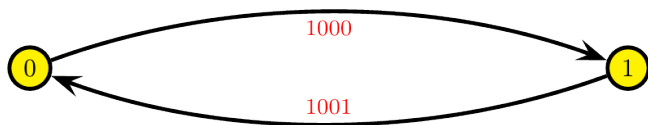
- Sending, from the address `sendbuf`, a message of `sendcount` elements of type `sendtype`, tagged `sendtag`, to the process `dest` in the communicator `comm` ;
- Receiving, at the address `recvbuf`, a message of `recvcount` elements of type `recvtype`, tagged `recvtag`, from the process `source` in the communicator `comm`.

### Remark :

- Here, the receiving zone `recvbuf` must be different from the sending zone `sendbuf`.

# Point-to-point Communications

Simultaneous send and receive `MPI_Sendrecv`



**Figure 11** – `sendrecv` Communication between the Processes 0 and 1

# Point-to-point Communications

## Example (see Fig. 11)

```
1 program sendrecv
2 use mpi_f08
3 implicit none
4 integer :: rank,value,num_proc
5 integer,parameter :: tag=110
6
7 call MPI_INIT()
8 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
9
10 ! We define the process we'll communicate with (we suppose that we have exactly 2 processes)
11 num_proc=mod(rank+1,2)
12
13 call MPI_SENDRECV(rank+1000,1,MPI_INTEGER,num_proc,tag,value,1,MPI_INTEGER, &
14                 num_proc,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE)
15
16 print *, 'I, process ',rank,', I received',value,' from process ',num_proc
17
18 call MPI_FINALIZE()
19 end program sendrecv
```

```
> mpiexec -n 2 sendrecv
```

```
I, process 1, I received 1000 from process 0
I, process 0, I received 1001 from process 1
```

# Point-to-point Communications

## Be careful !

In the case of a **synchronous** implementation of the `MPI_Send()` subroutine, if we replace the `MPI_Sendrecv()` subroutine in the example above by `MPI_Send()` followed by `MPI_Recv()`, the code will deadlock. Indeed, each of the two processes will wait for a receipt confirmation, which will never come because the two sending operations would stay suspended.

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

# Point-to-point Communications

## Simultaneous send and receive `MPI_Sendrecv_replace`

```
MPI_SENDRECV_REPLACE(buf, count, datatype,  
                    dest, sendtag,  
                    source, recvtag, comm, status_msg, code)  
  
TYPE(*), dimension(..)      :: buf  
integer, intent(in)         :: count, source, dest, sendtag, recvtag  
TYPE(MPI_Datatype), intent(in) :: datatype  
TYPE(MPI_Comm), intent(in)  :: comm  
TYPE(MPI_Status)            :: status_msg  
integer, optional, intent(out) :: code
```

- Sending, from the address `buf`, a message of `count` elements of type `datatype`, tagged `sendtag`, to the process `dest` in the communicator `comm` ;
- Receiving a message at the same address, with same count elements and same datatype, tagged `recvtag`, from the process `source` in the communicator `comm`.

### Remark :

- Contrary to the usage of `MPI_Sendrecv`, the receiving zone is the same here as the sending zone `buf`.

# Point-to-point Communications

## Example

```
1 program wilddcard
2   use mpi_f08
3   implicit none
4   integer, parameter      :: m=4,tag=11
5   integer, dimension(m,m) :: A
6   integer                 :: nb_procs,rank,i
7   TYPE(MPI_Status)       :: status_msg
8
9
10  call MPI_INIT()
11  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
12  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
13  A(:,:) = 0
14
15  if (rank == 0) then
16    ! Initialisation of the matrix A on the process 0
17    A(:,:) = reshape((/ (i,i=1,m*m) /), (/ m,m /))
18    ! Sending of 3 elements of the matrix A to the process 1
19    call MPI_SEND(A(1,1),3,MPI_INTEGER,1,tag1,MPI_COMM_WORLD)
20  else
21    ! We receive the message
22    call MPI_RECV(A(1,2),3, MPI_INTEGER ,MPI_ANY_SOURCE,MPI_ANY_TAG, &
23                 MPI_COMM_WORLD,status_msg)
24    print *,'I, process ',rank,', I received 3 elements from the process ', &
25            status_msg(MPI_SOURCE), 'with tag', status_msg( MPI_TAG ), &
26            " the elements are ", A(1:3,2)
27  end if
28  call MPI_FINALIZE()
29 end program wilddcard
```

# Point-to-point Communications

```
> mpiexec -n 2 wildcard
I, process      1, I received 3 elements from the process      0
with tag       11 the elements are      1      2      3
```



## MPI Hands-On – Exercise 2 : Ping-pong

- Point to point communications : *Ping-Pong* between two processes
- This exercise is composed of 3 steps :
  1. *Ping* : complete the script `ping_pong_1.f90` in such a way that the process 0 sends a message containing 1000 random reals to process 1.
  2. *Ping-Pong* : complete the script `ping_pong_2.f90` in such a way that the process 1 sends back the message to the process 0, and measure the communication duration with the `MPI_Wtime()` function.
  3. *Ping-Pong match* : complete the script `ping_pong_3.f90` in such a way that processes 0 and 1 perform 9 *Ping-Pong*, while varying the message size, and measure the communication duration each time. The corresponding bandwidths will be printed.

## MPI Hands-On – Exercise 2 : Ping-pong

### Remarks :

- To compile the first step : `make ping_pong_1`
- To execute the first step : `make exe1`
- To compile the second step : `make ping_pong_2`
- To execute the second step : `make exe2`
- To compile the last step : `make ping_pong_3`
- To execute the last step : `make exe3`
- The generation of random numbers uniformly distributed in the range  $[0,1[$  is made by calling the Fortran `random_number` subroutine :

```
call random_number(variable)
```

`variable` can be a scalar or an array

- The time duration measurements can be done like this :

```
time_begin=MPI_WTIME()  
.....  
time_end=MPI_WTIME()  
print ('"... in",f8.6," secondes. "') ,time_end-time_begin
```

## Collective communications

# Collective communications

## General concepts

- **Collective** communications allow making 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 (therefore, when the concerned memory area can be changed).
- The management of **tags** in these communications is transparent and system-dependent. Therefore, they are never explicitly defined during calls to subroutines. An advantage of this is that collective communications never interfere with point-to-point communications.

# Collective communications

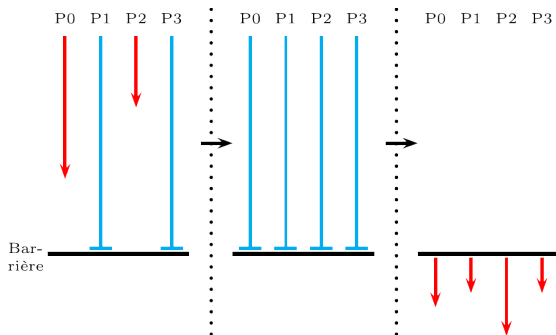
## Types of collective communications

There are three types of subroutines :

1. One which ensures global synchronizations : `MPI_Barrier()`
2. Ones which only transfer data :
  - Global distribution of data : `MPI_Bcast()`
  - Selective distribution of data : `MPI_Scatter()`
  - Collection of distributed data : `MPI_Gather()`
  - Collection of distributed data by all the processes : `MPI_Allgather()`
  - Collection and selective distribution by all the processes of distributed data : `MPI_Alltoall()`
3. Ones which, in addition to the communications management, carry out operations on the transferred data :
  - Reduction operations (sum, product, maximum, minimum, etc.), whether of a predefined or personal type : `MPI_Reduce()`
  - Reduction operations with distributing of the result (this is in fact equivalent to an `MPI_Reduce()` followed by an `MPI_Bcast()`) : `MPI_Allreduce()`

# Collective communications

## Global synchronization : `MPI_Barrier()`



**Figure 12** – Global Synchronization : `MPI_Barrier()`

```
MPI_BARRIER(comm, code)
```

```
TYPE(MPI_Comm), intent(in) :: comm
```

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

# Collective communications

Global distribution : `MPI_Bcast ()`

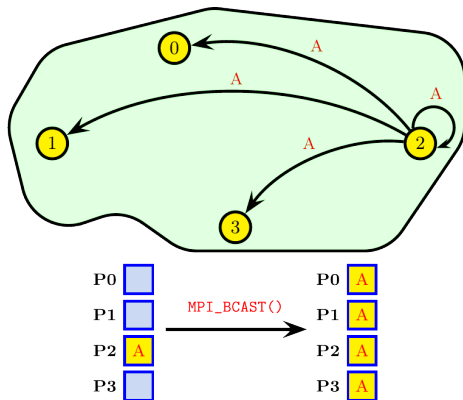


Figure 13 – Global distribution : `MPI_Bcast ()`

# Collective communications

## Global distribution : `MPI_Bcast()`

```
MPI_BCAST(buffer, count, datatype, root, comm, code)
```

```
TYPE(*), dimension(..)      :: buffer
integer, intent(in)         :: count, root
TYPE(MPI_Datatype), intent(in) :: datatype
TYPE(MPI_Comm), intent(in)  :: comm
integer, optional, intent(out) :: code
```

1. Send, starting at position `buffer`, a message of `count` element of type `datatype`, by the `root` process, to all the members of communicator `comm`.
2. Receive this message at position `buffer` for all the processes other than the `root`.



# Collective communications

## Example of `MPI_Bcast()`

```
1 program bcast
2   use mpi_f08
3   implicit none
4
5   integer :: rank,value
6
7   call MPI_INIT()
8   call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
9
10  if (rank == 2) value=rank+1000
11
12  call MPI_BCAST(value,1,MPI_INTEGER,2,MPI_COMM_WORLD)
13
14  print *, 'I, process ',rank,', received ',value,' of process 2'
15
16  call MPI_FINALIZE()
17
18 end program bcast
```

```
> mpiexec -n 4 bcast
```

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

# Collective communications

Selective distribution : `MPI_Scatter()`

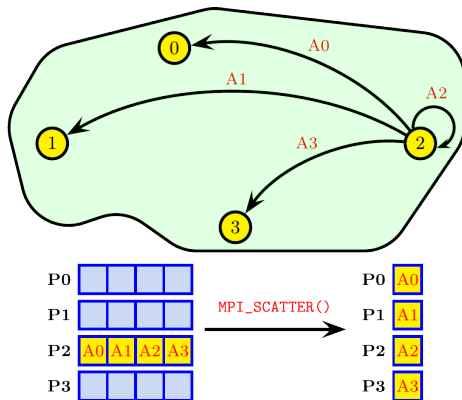


Figure 14 – Selective distribution : `MPI_Scatter()`

# Collective communications

## Selective distribution : `MPI_Scatter()`

```
MPI_SCATTER(sendbuf, sendcount, sendtype,  
            recvbuf, recvcount, recvtype, root, comm, code)  
  
TYPE(*), dimension(..)      :: sendbuf, recvbuf  
integer, intent(in)         :: sendcount, recvcount, root  
TYPE(MPI_Datatype), intent(in) :: sendtype, recvtype  
TYPE(MPI_Comm), intent(in)   :: comm  
integer, optional, intent(out) :: code
```

1. Scatter by process `root`, starting at position `sendbuf`, message `sendcount` element of type `sendtype`, to all the processes of communicator `comm`.
2. Receive this message at position `recvbuf`, of `recvcount` element of type `recvtype` for all processes of communicator `comm`.

### Remarks :

- The couples (`sendcount`, `sendtype`) and (`recvcount`, `recvtype`) must represent the same quantity of data.
- Data are scattered in chunks of same size ; a chunk consists of `sendcount` elements of type `sendtype`.
- The *i*-th chunk is sent to the *i*-th process.

# Collective communications

## Example of `MPI_Scatter()`

```
1 program scatter
2   use mpi_f08
3   implicit none
4   integer, parameter :: nb_values=8
5   integer :: nb_procs,rank,block_length,i,code
6   real, allocatable, dimension(:) :: values,recvdata
7
8   call MPI_INIT()
9   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
10  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
11  block_length=nb_values/nb_procs
12  allocate(recvdata(block_length))
13  if (rank == 2) then
14    allocate(values(nb_values))
15    values(:)=(/(1000.+i,i=1,nb_values)/)
16    print *, 'I, process ',rank,' send my values array : ',&
17            values(1:nb_values)
18  end if
19  call MPI_SCATTER(values,block_length,MPI_REAL,recvdata,block_length, &
20                MPI_REAL,2,MPI_COMM_WORLD)
21  print *, 'I, process ',rank,' received ', recvdata(1:block_length), &
22        ' of process 2'
23  call MPI_FINALIZE()
24
25 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, received 1001. 1002. of processus 2
I, process 1, received 1003. 1004. of processus 2
I, process 3, received 1007. 1008. of processus 2
I, process 2, received 1005. 1006. of processus 2
```

# Collective communications

Collection : `MPI_Gather()`

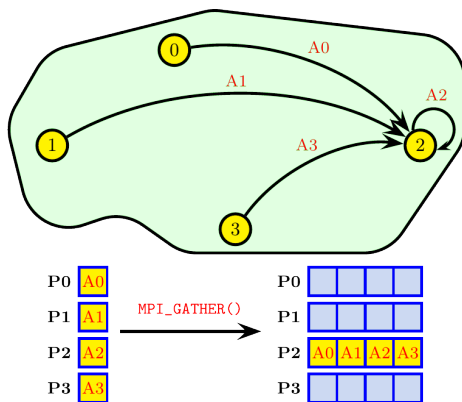


Figure 15 – Collection : `MPI_Gather()`

# Collective communications

## Collection : `MPI_Gather()`

```
MPI_GATHER(sendbuf, sendcount, sendtype,  
           recvbuf, recvcount, recvtype, root, comm, code)  
  
TYPE(*), dimension(..), intent(in) :: sendbuf  
TYPE(*), dimension(..)           :: recvbuf  
integer, intent(in)               :: sendcount, recvcount, root  
TYPE(MPI_Datatype), intent(in)    :: sendtype, recvtype  
TYPE(MPI_Comm), intent(in)        :: comm  
integer, optional, intent(out)    :: code
```

1. Send for each process of communicator `comm`, a message starting at position `sendbuf`, of `sendcount` element type `sendtype`.
2. Collect all these messages by the `root` process at position `recvbuf`, `recvcount` element of type `recvtype`.

### Remarks :

- The couples (`sendcount`, `sendtype`) and (`recvcount`, `recvtype`) must represent the same size of data.
- The data are collected in the order of the process ranks.

# Collective communications

## Collection : MPI\_Gather ()

```
1 program gather
2   use mpi_f08
3   implicit none
4   integer, parameter :: nb_values=8
5   integer :: nb_procs,rank,block_length,i
6   real, dimension(nb_values) :: recvdata
7   real, allocatable, dimension(:) :: values
8
9   call MPI_INIT()
10  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
12
13  block_length=nb_values/nb_procs
14
15  allocate(values(block_length))
16
17  values(:)=(/(1000.+rank*block_length+i,i=1,block_length)/)
18  print *, 'I, process ',rank,' sent my values array : ',&
19         values(1:block_length)
20
21  call MPI_GATHER(values,block_length,MPI_REAL,recvdata,block_length, &
22                MPI_REAL,2,MPI_COMM_WORLD)
23
24  if (rank == 2) print *, 'I, process 2', ' received ',recvdata(1:nb_values)
25
26  call MPI_FINALIZE()
27
28  end program gather
```

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

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

# Collective communications

Gather-to-all : `MPI_Allgather()`

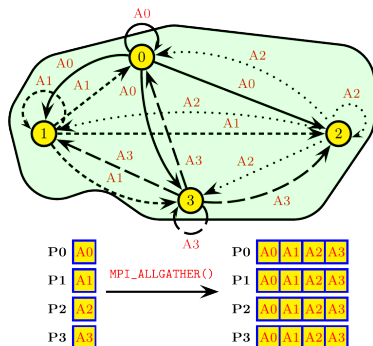


Figure 16 – Gather-to-all : `MPI_Allgather()`



# Collective communications

## Gather-to-all : `MPI_Allgather()`

Corresponds to an `MPI_Gather()` followed by an `MPI_Bcast()` :

```
MPI_ALLGATHER(sendbuf, sendcount, sendtype,
              recvbuf, recvcount, recvtype, comm, code)

TYPE(*), dimension(..), intent(in) :: sendbuf
TYPE(*), dimension(..)           :: recvbuf
integer, intent(in)               :: sendcount, recvcount
TYPE(MPI_Datatype), intent(in)    :: sendtype, recvtype
TYPE(MPI_Comm), intent(in)        :: comm
integer, optional, intent(out)    :: code
```

1. Send by each process of communicator `comm`, a message starting at position `sendbuf`, of `sendcount` element, type `sendtype`.
2. Collect all these messages, by all the processes, at position `recvbuf` of `recvcount` element type `recvtype`.

### Remarks :

- The couples (`sendcount`, `sendtype`) and (`recvcount`, `recvtype`) must represent the same data size.
- The data are gathered in the order of the process ranks.

# Collective communications

## Example of `MPI_Allgather()`

```
1 program allgather
2 use mpi_f08
3 implicit none
4
5 integer, parameter      :: nb_values=8
6 integer                :: nb_procs,rank,block_length,i
7 real, dimension(nb_values) :: recvdata
8 real, allocatable, dimension(:) :: values
9
10 call MPI_INIT()
11
12 call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
13 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
14
15 block_length=nb_values/nb_procs
16 allocate(values(block_length))
17
18 values(:)=(/(1000.+rank*block_length+i,i=1,block_length)/)
19
20 call MPI_ALLGATHER(values,block_length,MPI_REAL,recvdata,block_length, &
21 MPI_REAL,MPI_COMM_WORLD)
22
23 print *, 'I, process ',rank,', received ', recvdata(1:nb_values)
24
25 call MPI_FINALIZE()
26
27 end program allgather
```

```
> mpiexec -n 4 allgather
```

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

# Collective communications

## Extended gather : `MPI_Gatherv()`

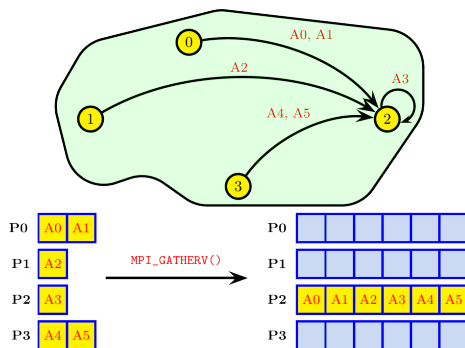


Figure 17 – Extended gather : `MPI_Gatherv()`

# Collective communications

## Extended Gather : `MPI_Gatherv()`

This is an `MPI_Gather()` where the size of messages can be different among processes :

```
MPI_GATHERV(sendbuf, sendcount, sendtype,
            recvbuf, recvcnts, displs, recvtype,
            root, comm, code)

TYPE(*), dimension(..), intent(in) :: sendbuf
TYPE(*), dimension(..)           :: recvbuf
integer, intent(in)               :: sendcount, root
TYPE(MPI_Datatype), intent(in)   :: sendtype, recvtype
integer, dimension(:), intent(in) :: recvcnts, displs
TYPE(MPI_Comm), intent(in)       :: comm
integer, optional, intent(out)    :: code
```

The  $i$ -th process of the communicator `comm` sends to process `root`, a message starting at position `sendbuf`, of `sendcount` element of type `sendtype`, and `root` receives at position `recvbuf`, of `recvcnts(i)` element of type `recvtype`, with a displacement of `displs(i)`.

### Remarks :

- The couples (`sendcount, sendtype`) of the  $i$ -th process and (`recvcnts(i), recvtype`) of process `root` must be such that the data size sent and received is the same.

# Collective communications

## Example of `MPI_Gatherv()`

```
1 program gatherv
2 use mpi_f08
3 implicit none
4 integer, parameter :: nb_values=10
5 integer :: nb_procs, rank, block_length, i
6 real, dimension(nb_values) :: recvddata, remainder
7 real, allocatable, dimension(:) :: values
8 integer, allocatable, dimension(:) :: nb_elements_received, displacement
9
10 call mpi_init()
11 call mpi_comm_size(mpi_comm_world, nb_procs)
12 call mpi_comm_rank(mpi_comm_world, rank)
13
14 block_length=nb_values/nb_procs
15 remainder=mod(nb_values,nb_procs)
16 if(rank < remainder) block_length = block_length + 1
17 allocate(values(block_length))
18 values(:) = (/ (1000.+(rank*(nb_values/nb_procs))+min(rank,remainder)+i, &
19              i=1,block_length)/)
20
21 print *, 'I, process ', rank, 'sent my values array : ', &
22         values(1:block_length)
23
24 if (rank == 2) then
25     allocate(nb_elements_received(nb_procs), displacement(nb_procs))
26     nb_elements_received(1) = nb_values/nb_procs
27     if (remainder > 0) nb_elements_received(1)=nb_elements_received(1)+1
28     displacement(1) = 0
29     do i=2, nb_procs
30         displacement(i) = displacement(i-1)+nb_elements_received(i-1)
31         nb_elements_received(i) = nb_values/nb_procs
32         if (i-1 < remainder) nb_elements_received(i)=nb_elements_received(i)+1
33     end do
34 end if
```

# Collective communications

## Example of `MPI_Gatherv()`

```
35 CALL MPI_GATHERV(values,block_length,MPI_REAL,recvdata,nb_elements_received,&
36                 displacement,MPI_REAL,2,MPI_COMM_WORLD)
37 IF (rank == 2) PRINT *, 'I, process 2, received ', recvdata (1:nb_values)
38 CALL MPI_FINALIZE()
39 end program gatherv
```

```
> mpiexec -n 4 gatherv
```

```
I, process 0 sent my values array : 1001. 1002. 1003.
I, process 2 sent my values array : 1007. 1008.
I, process 3 sent my values array : 1009. 1010.
I, process 1 sent my values array : 1004. 1005. 1006.

I, process 2 receives 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008. 1009. 1010.
```

# Collective communications

## Collection and distribution : `MPI_Alltoall()`

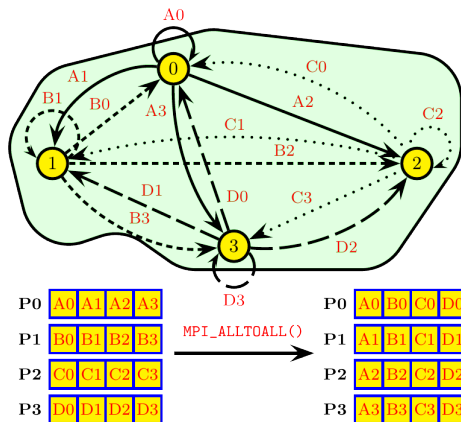


Figure 18 – Collection and distribution : `MPI_Alltoall()`

# Collective communications

## Collection and distribution : `MPI_Alltoall()`

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype,  
             recvbuf, recvcount, recvtype, comm, code)  
  
TYPE(*), dimension(..), intent(in) :: sendbuf  
TYPE(*), dimension(..)      :: recvbuf  
integer, intent(in)         :: sendcount, recvcount  
TYPE(MPI_Datatype), intent(in) :: sendtype, recvtype  
TYPE(MPI_Comm), intent(in)   :: comm  
integer, optional, intent(out) :: code
```

Here, the  $i$ -th process sends its  $j$ -th chunk to the  $j$ -th process which places it in its  $i$ -th chunk.

### Remark :

- The couples (`sendcount`, `sendtype`) and (`recvcount`, `recvtype`) must be such that they represent equal data sizes.



# Collective communications

## Example of `MPI_Alltoall()`

```
1 program alltoall
2   use mpi_f08
3   implicit none
4
5   integer, parameter      :: nb_values=8
6   integer                 :: nb_procs,rank,block_length,i
7   real, dimension(nb_values) :: values,recvdata
8
9   call MPI_INIT()
10  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
12
13  values(:)=(/(1000.+rank*nb_values+i,i=1,nb_values)/)
14  block_length=nb_values/nb_procs
15
16  print *, 'I, process ',rank,'sent my values array : ',&
17         values(1:nb_values)
18
19  call MPI_ALLTOALL(values,block_length,MPI_REAL,recvdata,block_length, &
20                 MPI_REAL,MPI_COMM_WORLD)
21
22  print *, 'I, process ',rank,', received ', recvdata(1:nb_values)
23
24  call MPI_FINALIZE()
25 end program alltoall
```

# Collective communications

## Example of `MPI_Alltoall()`

```
> mpiexec -n 4 alltoall
I, process 1 sent my values array :
1009. 1010. 1011. 1012. 1013. 1014. 1015. 1016.
I, process 0 sent my values array :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
I, process 2 sent my values array :
1017. 1018. 1019. 1020. 1021. 1022. 1023. 1024.
I, process 3 sent my values array :
1025. 1026. 1027. 1028. 1029. 1030. 1031. 1032.

I, process 0, received 1001. 1002. 1009. 1010. 1017. 1018. 1025. 1026.
I, process 2, received 1005. 1006. 1013. 1014. 1021. 1022. 1029. 1030.
I, process 1, received 1003. 1004. 1011. 1012. 1019. 1020. 1027. 1028.
I, process 3, received 1007. 1008. 1015. 1016. 1023. 1024. 1031. 1032.
```

# Collective communications

## Global reduction

- A **reduction** is an operation applied to a set of elements in order to obtain one single value. Typical examples are the sum of the elements of a vector ( $SUM(A(:))$ ) or the search for the maximum value element in a vector ( $MAX(V(:))$ ).
- MPI proposes high-level subroutines in order to operate reductions on data distributed on a group of processes. The result is obtained on only one process ( $MPI\_Reduce()$ ) or on all the processes ( $MPI\_Allreduce()$ ), which is in fact equivalent to an  $MPI\_Reduce()$  followed by an  $MPI\_Bcast()$ .
- If several elements are implied by process, the reduction function is applied to each one of them (for instance to each element of a vector).

# Collective communications

## Distributed reduction : MPI\_Reduce

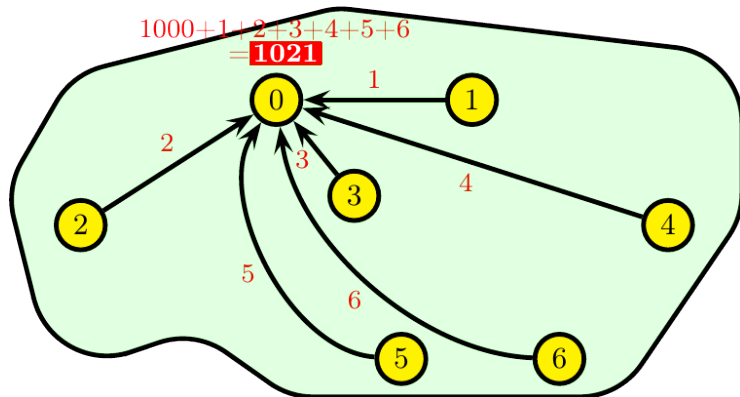


Figure 19 – Distributed reduction (sum)

# Collective communications

## Operations

Name	Operation
<code>MPI_SUM</code>	Sum of elements
<code>MPI_PROD</code>	Product of elements
<code>MPI_MAX</code>	Maximum of elements
<code>MPI_MIN</code>	Minimum of elements
<code>MPI_MAXLOC</code>	Maximum of elements and location
<code>MPI_MINLOC</code>	Minimum of elements and location
<code>MPI_LAND</code>	Logical AND
<code>MPI_LOR</code>	Logical OR
<code>MPI_LXOR</code>	Logical exclusive OR

# Collective communications

## Global reduction : `MPI_Reduce()`

```
MPI_REDUCE(sendbuf,recvbuf,count,datatype,op,root,comm,code)
```

```
TYPE(*), dimension(..), intent(in) :: sendbuf
TYPE(*), dimension(..)           :: recvbuf
integer, intent(in)               :: count, root
TYPE(MPI_Datatype), intent(in)   :: datatype
TYPE(MPI_Op), intent(in)         :: op
TYPE(MPI_Comm), intent(in)       :: comm
integer, optional, intent(out)   :: code
```

1. Distributed reduction of `count` elements of type `datatype`, starting at position `sendbuf`, with the operation `op` from each process of the communicator `comm`,
2. Return the result at position `recvbuf` in the process `root`.

# Collective communications

## Example of `MPI_Reduce()`

```
1 program reduce
2 use mpi_f08
3 implicit none
4 integer :: nb_procs, rank, value, sum
5
6 call MPI_INIT()
7 call MPI_COMM_SIZE(MPI_COMM_WORLD, nb_procs)
8 call MPI_COMM_RANK(MPI_COMM_WORLD, rank)
9
10 if (rank == 0) then
11     value=1000
12 else
13     value=rank
14 endif
15
16 call MPI_REDUCE(value, somme, 1, MPI_INTEGER, MPI_SUM, 0, MPI_COMM_WORLD)
17
18 if (rank == 0) then
19     print *, 'I, process 0, have the global sum value ', sum
20 end if
21
22 call MPI_FINALIZE()
23 end program reduce
```

```
> mpiexec -n 7 reduce
```

```
I, process 0, have the global sum value 1021
```

# Collective communications

Distributed reduction with distribution of the result : `MPI_Allreduce()`

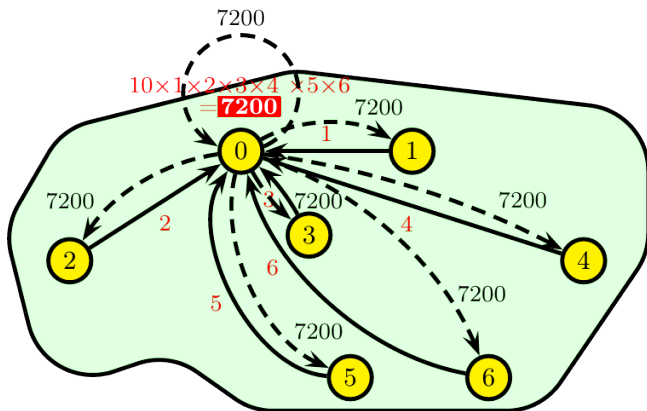


Figure 20 – Distributed reduction (product) with distribution of the result



# Collective communications

## Global all-reduction : `MPI_Allreduce()`

```
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, code)
```

```
TYPE(*), dimension(..), intent(in) :: sendbuf
TYPE(*), dimension(..)           :: recvbuf
integer, intent(in)               :: count
TYPE(MPI_Datatype), intent(in)   :: datatype
TYPE(MPI_Op), intent(in)         :: op
TYPE(MPI_Comm), intent(in)       :: comm
integer, optional, intent(out)   :: code
```

1. Distributed reduction of `count` elements of type `datatype` starting at position `sendbuf`, with the operation `op` from each process of the communicator `comm`,
2. Write the result at position `recvbuf` for all the processes of the communicator `comm`.

# Collective communications

## Example of `MPI_Allreduce()`

```
1 program allreduce
2
3 use mpi_f08
4 implicit none
5
6 integer :: nb_procs,rank,value,product
7
8 call MPI_INIT()
9 call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
10 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
11
12 if (rank == 0) then
13     value=10
14 else
15     value=rank
16 endif
17
18 call MPI_ALLREDUCE(value,product,1,MPI_INTEGER,MPI_PROD,MPI_COMM_WORLD)
19
20 print *, 'I,process ',rank,', received the value of the global product ', product
21
22 call MPI_FINALIZE()
23
24 end program allreduce
```

# Collective communications

## Example of `MPI_Allreduce()`

```
> mpiexec -n 7 allreduce
```

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

# Collective communications

## Additions

- The `MPI_Scan()` subroutine allows making partial reductions by considering, for each process, the previous processes of the communicator and itself.  
`MPI_Exscan()` is the *exclusive* version of `MPI_Scan()`, which is *inclusive*.
- The `MPI_Op_create()` and `MPI_Op_free()` subroutines allow personal reduction operations.
- For each reduction operation, the keyword `MPI_IN_PLACE` can be used in order to keep the result in the same place as the sending buffer (but only for the rank(s) that will receive results). Example : `call MPI_Allreduce(MPI_IN_PLACE, sendrecvbuf, ...)`.

# Collective communications

## Additions

- Similarly to what we have seen for `MPI_Gatherv()` with respect to `MPI_Gather()`, the `MPI_Scatterv()`, `MPI_Allgatherv()` and `MPI_Alltoallv()` subroutines extend `MPI_Scatter()`, `MPI_Allgather()` and `MPI_Alltoall()` to the cases where the processes have different numbers of elements to transmit or gather.
- `MPI_Alltoallw()` is the version of `MPI_Alltoallv()` which enables to deal with heterogeneous elements (by expressing the displacements in bytes and not in elements).

## MPI Hands-On – Exercise 3 : Collective communications and reductions

- The aim of this exercise is to compute  $\pi$  by numerical integration.  $\pi = \int_0^1 \frac{4}{1+x^2} dx$ .
- We use the rectangle method (mean point).
- Let  $f(x) = \frac{4}{1+x^2}$  be the function to integrate.
- $nblock$  is the number of points of discretization.
- $width = \frac{1}{nblock}$  the length of discretization and the width of all rectangles.
- Sequential version is available in the `pi.f90` source file.
- You have to do the parallel version with **MPI** in this file.

## Communication Modes

# Communication Modes

## Point-to-Point Send Modes

<i>Mode</i>	Blocking	Non-blocking
Standard send	<code>MPI_Send()</code>	<code>MPI_Isend()</code>
Synchronous send	<code>MPI_Ssend()</code>	<code>MPI_Issend()</code>
Buffered send	<code>MPI_Bsend()</code>	<code>MPI_Ibsend()</code>
Receive	<code>MPI_Recv()</code>	<code>MPI_Irecv()</code>



# Communication Modes

## 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 sent can be modified after the call.
- The data received can be read after the call.

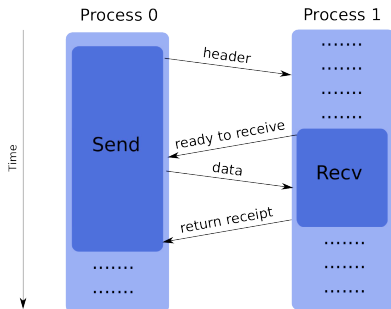
# Communication Modes

## Synchronous sends

A synchronous send involves a synchronization between the involved processes. A send cannot start until its receive is posted. There can be no communication before the two processes are ready to communicate.

## Rendezvous Protocol

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



# Communication Modes

## Interface of `MPI_Ssend()`

```
MPI_SSEND(values, count, msgtype, dest, tag, comm, code)

TYPE(*), dimension(..), intent(in) :: values
integer, intent(in)                :: count, dest, tag
TYPE(MPI_Datatype), intent(in)     :: msgtype
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

## Advantages of synchronous mode

- Low resource consumption (no buffer)
- Rapid if the receiver is ready (no copying in a buffer)
- Knowledge of receipt through synchronization

## Disadvantages of synchronous mode

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

# Communication Modes

## Deadlock example

In the following example, there is a deadlock because we are in synchronous mode. The two processes are blocked on the `MPI_Ssend()` call because they are waiting for the `MPI_Recv()` of the other process. However, the `MPI_Recv()` call can only be made after the unblocking of the `MPI_Ssend()` call.

```
1 program ssendrecv
2   use mpi_f08
3   implicit none
4   integer                               :: rank,value,num_proc
5   integer,parameter                    :: tag=110
6
7   call MPI_INIT()
8   call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
9
10  ! We run on 2 processes
11  num_proc=mod(rank+1,2)
12
13  call MPI_SSEND(rank+1000,1,MPI_INTEGER,num_proc,tag,MPI_COMM_WORLD)
14  call MPI_RECV(value,1,MPI_INTEGER,num_proc,tag,MPI_COMM_WORLD, &
15             MPI_STATUS_IGNORE)
16
17  print *, 'I, process',rank,', received',value,'from process',num_proc
18
19  call MPI_FINALIZE()
20 end program ssendrecv
```

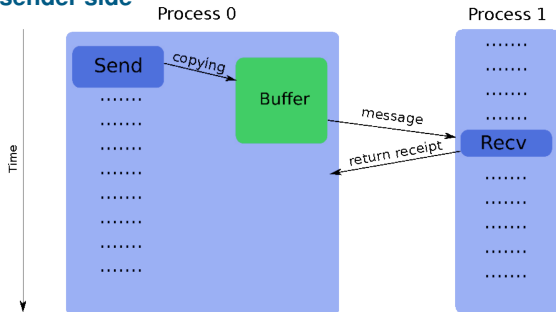
# Communication Modes

## Buffered sends

A buffered send implies the copying of data into an intermediate memory space. There is then no coupling between the two processes of communication. Therefore, the return of this type of send does not mean that the receive has occurred.

## Protocol with user buffer on the sender side

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.



# Communication Modes

## Buffered sends

The buffers have to be managed manually (with calls to `MPI_Buffer_attach()` and `MPI_Buffer_detach()`). Message header size needs to be taken into account when allocating buffers (by adding the constant `MPI_BSEND_OVERHEAD()` for each message occurrence).

## Interfaces

```
MPI_BUFFER_ATTACH (buf, typesize, code)
MPI_BUFFER_DETACH (buf_adr, typesize, code)

TYPE(*), dimension(..), asynchronous :: buf
TYPE(C_PTR), intent(out)              :: buf_adr
integer                                :: typesize
integer, optional, intent(out)        :: code

MPI_BSEND (values, count, msgtype, dest, tag, comm, code)

TYPE(*), dimension(..), intent(in)    :: values
integer, intent(in)                   :: count, dest, tag
TYPE(MPI_Datatype), intent(in)        :: msgtype
TYPE(MPI_Comm), intent(in)            :: comm
integer, optional, intent(out)        :: code
```

# Communication Modes

## Advantages of buffered mode

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

## Disadvantages of buffered mode

- Uses more resources (memory use by buffers with saturation risk)
- The send buffers in the `MPI_Bsend()` or `MPI_Ibsend()` calls have to be managed manually (often difficult to choose a suitable size)
- Slightly slower than the synchronous sends if the receiver is ready
- No knowledge of receipt (send-receive decoupling)
- Risk of wasted memory space if buffers are too oversized
- Application crashes if buffer is too small
- There are often hidden buffers managed by the MPI implementation on the sender side and/or on the receiver side (and consuming memory resources)

# Communication Modes

## No deadlocks

In the following example, we don't have a deadlock because we are in buffered mode. After the copy is made in the *buffer*, the `MPI_Bsend()` call returns and then the `MPI_Recv()` call is made.

```
1 program bsendrecv
2   use mpi_f08
3   use, INTRINSIC :: ISO_C_BINDING
4   implicit none
5   integer                               :: rank,value,num_proc,typesize,overhead,bufsize
6   integer,parameter                    :: tag=110, nb_elt=1, nb_msg=1
7   integer,dimension(:), allocatable    :: buffer
8   TYPE(C_PTR)                           :: p
9
10  call MPI_INIT()
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
12
13  call MPI_TYPE_SIZE(MPI_INTEGER,typesize)
14  ! Convert MPI_BSEND_OVERHEAD (bytes) in number of integer
15  overhead = int(1+(MPI_BSEND_OVERHEAD*1.)/typesize)
16  allocate(buffer(nb_msg*(nb_elt+overhead)))
17  bufsize = typesize*nb_msg*(nb_elt+overhead)
18  call MPI_BUFFER_ATTACH(buffer,bufsize)
19  ! We run on 2 processes
20  num_proc=mod(rank+1,2)
21  call MPI_BSEND(rank+1000,nb_elt,MPI_INTEGER,num_proc,tag,MPI_COMM_WORLD)
22  call MPI_RECV(value,nb_elt,MPI_INTEGER, num_proc,tag,MPI_COMM_WORLD, &
23              MPI_STATUS_IGNORE)
24
25  print *, 'I, process', rank, ', received', value, 'from process', num_proc
26  call MPI_BUFFER_DETACH(p,bufsize)
27  call MPI_FINALIZE()
28 end program bsendrecv
```



# Communication Modes

## Standard sends

A standard send is made by calling the `MPI_Send()` subroutine. In most implementations, the mode is buffered (*eager*) for small messages but is synchronous for larger messages.

## Interfaces

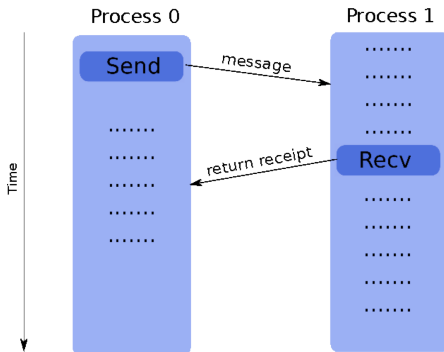
```
MPI_SEND(values, count, msgtype, dest, tag, comm, code)

TYPE(*), dimension(..), intent(in) :: values
integer, intent(in)                :: count, dest, tag
TYPE(MPI_Datatype), intent(in)     :: msgtype
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

# Communication Modes

## The 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()` for 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.



# Communication Modes

## Advantages of standard mode

- Often the most efficient (because the constructor chose the best parameters and algorithms)

## Disadvantages of standard mode

- Little control over the mode actually used (often accessible via environment variables)
- Risk of deadlocks depending on the mode used
- Behavior can vary according to the architecture and problem size

# Communication Modes

## Number of received elements

```
MPI_RECV(buf, count, datatype, source, tag, comm, msgstatus, code)
```

```
TYPE(*), dimension(..)      :: buf
integer                      :: count, source, tag
TYPE(MPI_Datatype), intent(in) :: datatype
TYPE(MPI_Comm), intent(in)   :: comm
TYPE(MPI_Status)             :: msgstatus
integer, optional, intent(out) :: code
```

- In `MPI_Recv()` call, the `count` argument in the standard is the number of elements in the buffer `buf`.
- This number must be greater than the number of elements to be received.
- When it is possible, for increased clarity, it is advised to put the number of elements to be received.
- We can obtain the number of elements received with `MPI_Get_count()` and the `msgstatus` argument returned by the `MPI_Recv()` call.

```
MPI_GET_COUNT(msgstatus, msgtype, count, code)
```

```
TYPE(MPI_Status), intent(in)  :: msgstatus
TYPE(MPI_Datatype), intent(in) :: msgtype
integer, intent(out)          :: count
integer, optional, intent(out) :: code
```

# Communication Modes

## Number of received elements

`MPI_Probe` allow incoming messages to be checked for, without actually receiving them.

```
MPI_PROBE(source, tag, comm, statut, code)

integer, intent(in)           :: source, tag
TYPE(MPI_Comm), intent(in)   :: comm
TYPE(MPI_Status)             :: statut
integer, optional, intent(out) :: code
```

A common use of `MPI_Probe` is to allocate space for a message before receiving it.

```
call MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, statut)
call MPI_Get_count(statut, MPI_INTEGER, msgsize)
allocate(buf(msgsize))
call MPI_Recv(buf, msgsize, MPI_INTEGER, statut%MPI_SOURCE,
             statut%MPI_TAG, comm, MPI_STATUS_IGNORE)
```

# Communication Modes

## Presentation

The overlap of communications by computations is a method which allows executing communications operations in the background while the program continues to operate. On Jean Zay, the latency of a communication internode is  $1.5 \mu\text{s}$ , or 2500 processor cycles.

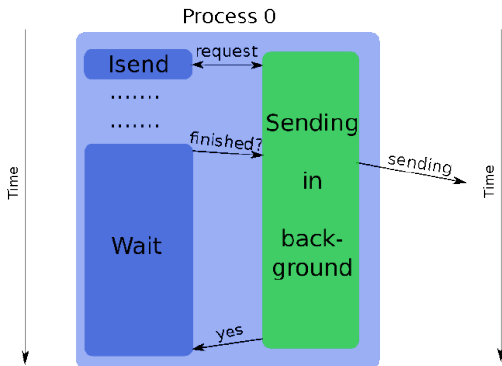
- It is thus possible, if the hardware and software architecture allows it, to hide all or part of communications costs.
- The computation-communication overlap can be seen as an additional level of parallelism.
- This approach is used in MPI by using nonblocking subroutines (i.e. `MPI_Isend()`, `MPI_Irecv()`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

## Non blocking communication

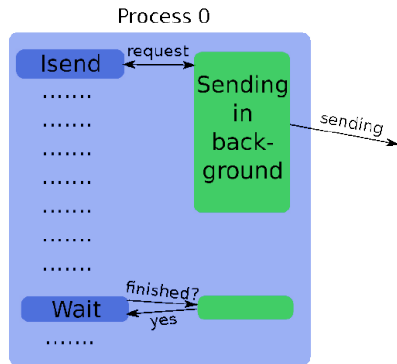
A nonblocking call returns very quickly but it does not authorize the immediate re-use of the memory space which was used in the communication. It is necessary to make sure that the communication is fully completed (with `MPI_Wait()`, for example) before using it again.

# Communication Modes

## Partial overlap



## Full overlap



# Communication Modes

## Advantages of non blocking call

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

## Disadvantages of non blocking call

- Greater additional costs (several calls for one single send or receive, request management)
- Higher complexity and more complicated maintenance
- Less efficient on some machines (for example with transfer starting only at the `MPI_Wait()` call)
- Risk of performance loss 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 is extended to collective communications in MPI 3.0)



# Communication Modes

## Interfaces

`MPI_Isend()` `MPI_Issend()` and `MPI_IbSEND()` for nonblocking send

```
MPI_ISEND(values, count, datatype, dest, tag, comm, req, code)
MPI_ISSEND(values, count, datatype, dest, tag, comm, req, code)
MPI_IBSEND(values, count, datatype, dest, tag, comm, req, code)

TYPE(*), dimension(..), intent(in), asynchronous :: values
integer, intent(in) :: count, dest, tag
TYPE(MPI_Datatype), intent(in) :: datatype
TYPE(MPI_Comm), intent(in) :: comm
TYPE(MPI_Request), intent(out) :: req
integer, optional, intent(out) :: code
```

`MPI_Irecv()` for nonblocking receive.

```
MPI_IRECV(values, count, msgtype, source, tag, comm, req, code)

TYPE(*), dimension(..), intent(in), asynchronous :: values
integer, intent(in) :: count, source, tag
TYPE(MPI_Datatype), intent(in) :: msgtype
TYPE(MPI_Comm), intent(in) :: comm
TYPE(MPI_Request), intent(out) :: req
integer, optional, intent(out) :: code
```

# Communication Modes

## Interfaces

`MPI_Wait()` wait for the end of a communication, `MPI_Test()` is the nonblocking version.

```
MPI_WAIT(req, statut, code)
MPI_TEST(req, flag, statut, code)

TYPE(MPI_Request), intent(inout) :: req
logical, intent(out)           :: flag
TYPE(MPI_Status)              :: statut
integer, optional, intent(out) :: code
```

`MPI_Waitall()` (`MPI_Testall()`) await the end of all communications.

```
MPI_WAITALL(count, reqs, statuts, code)
MPI_TESTALL(count, reqs, flag, statuts, code)

integer, intent(in)           :: count
TYPE(MPI_Request), dimension(taille) :: reqs
logical, intent(out)         :: flag
TYPE(MPI_Status), dimension(taille) :: statuts
integer, optional, intent(out) :: code
```

# Communication Modes

## Interfaces

`MPI_Waitany()` wait for the end of one communication, `MPI_Testany()` is the nonblocking version.

```
MPI_WAITANY(count, reqs, index, msgstatus, code)
MPI_TESTANY(count, reqs, index, flag, msgstatus, code)

integer, intent(in) :: count
TYPE(MPI_Request), dimension(count), intent(inout) :: reqs
integer, intent(out) :: index
logical, intent(out) :: flag
TYPE(MPI_Status) :: msgstatus
integer, optional, intent(out) :: code
```

`MPI_Waitsome()` wait for the end of at least one communication, `MPI_Testsome()` is the nonblocking version.

```
MPI_WAITSSOME(count, reqs, outcount, indices, statuses, code)
MPI_TESTSSOME(count, reqs, outcount, indices, statuses, code)

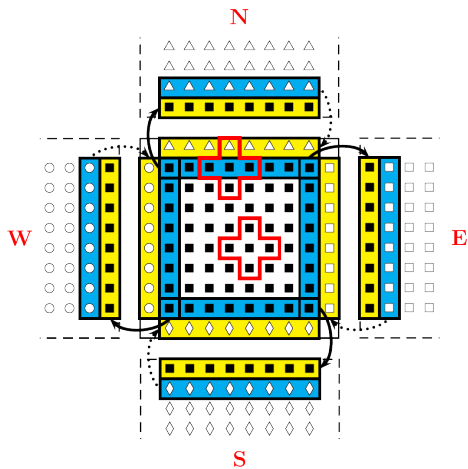
integer, intent(in) :: count
TYPE(MPI_Request), dimension(count), intent(inout) :: reqs
integer, intent(out) :: outcount
integer, dimension(count), intent(out) :: indices
TYPE(MPI_Status), dimension(count), intent(out) :: statuses
integer, optional, intent(out) :: code
```

# Communication Modes

## Request management

- After a call to a blocking wait function (`MPI_Wait()`, `MPI_Waitall()`, ...), the request argument is set to `MPI_REQUEST_NULL`.
- The same for a nonblocking wait when the *flag* is set to true.
- A wait call with a `MPI_REQUEST_NULL` request does nothing.

# Communication Modes



# Communication Modes

```
1  SUBROUTINE start_communication(u)
2      ! Send to the North and receive from the South
3      CALL MPI_Irecv( u(:,), 1, rowtype, neighbor(S), &
4          tag, comm2d, request(1))
5      CALL MPI_Isend( u(:,), 1, rowtype, neighbor(N), &
6          tag, comm2d, request(2))
7
8      ! Send to the South and receive from the North
9      CALL MPI_Irecv( u(:,), 1, rowtype, neighbor(N), &
10         tag, comm2d, request(3))
11     CALL MPI_Isend( u(:,), 1, rowtype, neighbor(S), &
12         tag, comm2d, request(4))
13
14     ! Send to the West and receive from the East
15     CALL MPI_Irecv( u(:,), 1, columntype, neighbor(E), &
16         tag, comm2d, request(5))
17     CALL MPI_Isend( u(:,), 1, columntype, neighbor(W), &
18         tag, comm2d, request(6))
19
20     ! Send to the East and receive from the West
21     CALL MPI_Irecv( u(:,), 1, columntype, neighbor(W), &
22         tag, comm2d, request(7))
23     CALL MPI_Isend( u(:,), 1, columntype, neighbor(E), &
24         tag, comm2d, request(8))
25 END SUBROUTINE start_communication
26 SUBROUTINE end_communication(u)
27     CALL MPI_WAITALL(2*NB_NEIGHBORS, request, tab_status, code)
28     if (.not. MPI_ASYNC_PROTECTS_NONBLOCKING) call MPI_F_SYNC_REG(u)
29 END SUBROUTINE end_communication
```

# Communication Modes

```
1 DO WHILE ((.NOT. convergence) .AND. (it < it_max))
2   it = it + 1
3   u(sx:ex,sy:ey) = u_new(sx:ex,sy:ey)
4
5   ! Exchange value on the interfaces
6   CALL start_communication( u )
7
8   ! Compute u
9   CALL calcul( u, u_new, sx+1, ex-1, sy+1, ey-1)
10
11  CALL end_communication( u )
12
13  ! North
14  CALL calcul( u, u_new, sx, sx, sy, ey)
15  ! South
16  CALL calcul( u, u_new, ex, ex, sy, ey)
17  ! West
18  CALL calcul( u, u_new, sx, ex, sy, sy)
19  ! East
20  CALL calcul( u, u_new, sx, ex, ey, ey)
21
22  ! Compute global error
23  diffnorm = global_error (u, u_new)
24
25  convergence = ( diffnorm < eps )
26
27  END DO
```

# Communication Modes

## Overlap levels on different machines

<i>Machine</i>	<i>Level</i>
Zay(IntelMPI)	43%
Zay(IntelMPI) I_MPI_ASYNC_PROGRESS=yes	95%

Measurements taken by overlapping a compute kernel with a communication kernel which have the same execution times.

An overlap of 0% means that the total execution time is twice the time of a compute (or a communication) kernel.

An overlap of 100% means that the total execution time is the same as the time of a compute (or a communication) kernel.



## Nonblocking collectives

- Nonblocking version of collective communications
- With an I (**immediate**) before : `MPI_Ireduce()` , `MPI_Ibcast()` , ...
- Wait with `MPI_Wait()` , `MPI_Test()` calls and all their variants
- No match between blocking and nonblocking
- The *status* argument retrieved by `MPI_Wait()` has an undefined value for `MPI_SOURCE` and `MPI_TAG`
- For a given communicator, the call order must be the same

```
MPI_IBARRIER(comm, request, code)
```

```
TYPE(MPI_Comm), intent(in)      :: comm,  
TYPE(MPI_Request), intent(out)  :: request  
integer, optional, intent(out) :: code
```

# Modèles de communication

## Usage example of `MPI_Ibarrier`

How to manage communications when we don't know at each iteration if our neighbors will send a message.

```
logical isAllFinish=.false.
logical isMySendFinish=.false.
do i=1,m
  ! Synchronous send
  call MPI_ISSEND(sbuf(i), ssize(i), datatype, dst(i), tag, comm, reqs(i))
end do

do while (.not. isAllFinish)
  ! Do we have a message ready to be received
  call MPI_IPROBE(MPI_ANY_SOURCE, tag, comm, flag, astat)
  if (flag) then
    ! Receive the message
    call MPI_RECV(rbuf, rsize, datatype, astat%MPI_SOURCE, tag, comm, rstat)
  end if
  if (.not. isMySendFinish) then
    ! Check if all our ssend are finished
    call MPI_TESTALL(m, reqs, flag, MPI_STATUSES_IGNORE)
    if (flag) then
      ! If yes we start the ibarrier
      call MPI_IBARRIER(comm, reqb)
      isMySendFinish=.true.
    end if
  else
    ! Test if everybody have started the ibarrier
    call MPI_TEST(reqb, isAllFinish, MPI_STATUS_IGNORE)
  end if
end do
```

# Communication Modes

## MPI-3 functionalities added

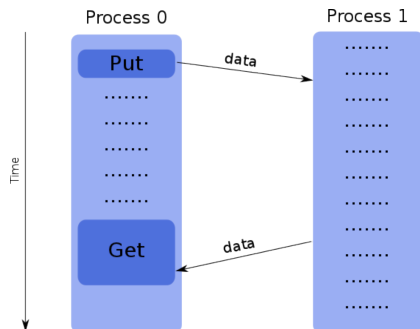
- If `MPI_SUBARRAYS_SUPPORTED` is set to *true*, it's possible to use Fortran subarrays in nonblocking calls.
- If `MPI_ASYNC_PROTECTS_NONBLOCKING` is set to *true*, the send and/or receive arguments are *asynchronous* in nonblocking interfaces.

```
call MPI_ISEND(buf, ..., req)
...
call MPI_WAIT(req, ...)
if (.not. MPI_ASYNC_PROTECTS_NONBLOCKING) call MPI_F_SYNC_REG(buf)
buf = val2
```

# Communication Modes

## One-Sided Communications

One-sided communications (Remote Memory Access or RMA) consists of accessing the memory of a distant process in *read* or *write* without the distant process having to manage this access explicitly. The target process does not intervene during the transfer.



# Communication Modes

## General approach

- Creation of a memory window with `MPI_Win_create()` to authorize RMA transfers in this zone.
- Remote access in *read* or *write* by calling `MPI_Put()`, `MPI_Get()`, `MPI_Accumulate()`, `MPI_Fetch_and_op()`, `MPI_Get_accumulate()` and `MPI_Compare_and_swap()`.
- Free the memory window with `MPI_Win_free()`.

## Synchronization methods

In order to ensure the correct functioning of the application, it is necessary to execute some synchronizations. Three methods are available :

- Active target communication with global synchronization (`MPI_Win_fence()`)
- Active target communication with synchronization by pair (`MPI_Win_start()` and `MPI_Win_complete()` for the origin process ; `MPI_Win_post()` and `MPI_Win_wait()` for the target process)
- Passive target communication without target intervention (`MPI_Win_lock()` and `MPI_Win_unlock()`)

# Communication Modes

```
1 program ex_fence
2   use mpi_f08
3   implicit none
4
5   integer, parameter :: assert=0
6   integer :: code, rank, realsize, i, nbelts, targetrank, m=4, n=4
7   TYPE(MPI_Win) :: win
8   integer (kind=MPI_ADDRESS_KIND) :: displacement, dim_win
9   real(kind=kind(1.d0)), dimension(:), allocatable :: win_local, tab
10
11  call MPI_INIT()
12  call MPI_COMM_RANK(MPI_COMM_WORLD, rank)
13  call MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION, realsize)
14
15  if (rank==0) then
16     n=0
17     allocate(tab(m))
18  endif
19
20  allocate(win_local(n))
21  dim_win = realsize*n
22
23  call MPI_WIN_CREATE(win_local, dim_win, realsize, MPI_INFO_NULL, &
24                     MPI_COMM_WORLD, win)
```

# Communication Modes

```
25  if (rank==0) then
26      tab(:) = (/ (i, i=1,m) /)
27  else
28      win_local(:) = 0.0
29  end if
30
31  call MPI_WIN_FENCE(assert,win)
32  if (rank==0) then
33      targetrank = 1; nbelts = 2; displacement = 1
34      call MPI_PUT(tab, nbelts, MPI_DOUBLE_PRECISION, targetrank, displacement, &
35                  nbelts, MPI_DOUBLE_PRECISION, win)
36  end if
37
38  call MPI_WIN_FENCE(assert,win)
39  if (rank==0) then
40      tab(m) = sum(tab(1:m-1))
41  else
42      win_local(n) = sum(win_local(1:n-1))
43  endif
44
45  call MPI_WIN_FENCE(assert,win)
46  if (rank==0) then
47      nbelts = 1; displacement = m-1
48      call MPI_GET(tab, nbelts, MPI_DOUBLE_PRECISION, targetrank, displacement, &
49                  nbelts, MPI_DOUBLE_PRECISION, win)
50  end if
```

# Communication Modes

## Advantages of One-Sided Communications

- Certain algorithms can be written more easily.
- More efficient than point-to-point communications on certain machines (use of specialized hardware such as a DMA engine, coprocessor, specialized memory, ...).
- The implementation can group together several operations.

## Disadvantages of One-Sided Communications

- Synchronization management is tricky.
- Complexity and high risk of error.
- Less efficient than point-to-point communications on certain machines.



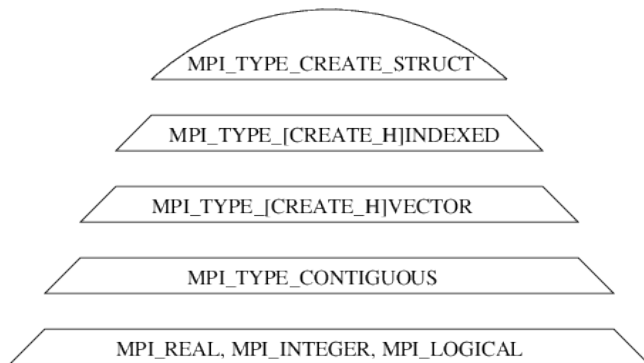
## Derived datatypes

# Derived datatypes

## Introduction

- In communications, exchanged data have different 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_indexed()` or `MPI_Type_create_struct()`
- Derived datatypes allow exchanging non-contiguous or non-homogenous data in the memory and limiting the number of calls to communications subroutines.

## Derived datatypes



**Figure 21** – Hierarchy of the MPI constructors

# Derived datatypes

## 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 22 – `MPI_Type_contiguous` subroutine

```
MPI_TYPE_CONTIGUOUS(count,old_type,new_type,code)  
  
integer, intent(in)           :: count  
TYPE(MPI_Datatype), intent(int) :: old_type  
TYPE(MPI_Datatype), intent(out) :: new_type  
integer, optional, intent(out) :: code
```

# Derived datatypes

## Constant stride

- `MPI_Type_vector()` creates a data structure from a homogenous set of existing datatypes **separated by a constant stride** in memory. The stride is given in 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 23** – `MPI_Type_vector` subroutine

```
MPI_TYPE_VECTOR(count,block_length,stride,old_type,new_type,code)  
  
integer, intent(in)           :: count,block_length  
integer, intent(in)           :: stride ! donne en elements  
TYPE(MPI_Datatype), intent(in) :: old_type  
TYPE(MPI_Datatype), intent(out) :: new_type  
integer, optional, intent(out) :: code
```

# Derived datatypes

## Constant stride

- `MPI_Type_create_hvector()` creates a data structure from a **homogenous** set of existing datatype **separated by a constant stride** in memory. The stride is given in **bytes**.
- This call is useful when the old type is no longer a base datatype (`MPI_INTEGER`, `MPI_REAL`,...) but a more complex datatype constructed by using MPI subroutines, because in this case the stride can no longer be given in number of elements.

```
MPI_Type_create_hvector(count,block_length,stride,old_type,new_type,code)

integer, intent(in)                :: count,block_length
integer(kind=MPI_ADDRESS_KIND), intent(in) :: stride ! given in bytes
TYPE(MPI_Datatype), intent(in)      :: old_type
TYPE(MPI_Datatype), intent(out)     :: new_type
integer, optional, intent(out)      :: code
```

# Derived datatypes

## Commit derived datatypes

- Before using a new derived datatype, it is necessary to validate it with the `MPI_Type_commit()` subroutine.

```
MPI_TYPE_COMMIT(new_type, code)

TYPE(MPI_Datatype), intent(inout) :: new_type
integer, optional, intent(out)    :: code
```

- The freeing of a derived datatype is made by using the `MPI_Type_free()` subroutine.

```
MPI_TYPE_FREE(new_type, code)

TYPE(MPI_Datatype), intent(inout) :: new_type
integer, optional, intent(out)    :: code
```

# Derived datatypes

```
1 program column
2   use mpi_f08
3   implicit none
4
5   integer, parameter           :: nb_lines=5, nb_columns=6
6   integer, parameter          :: tag=100
7   real, dimension(nb_lines, nb_columns) :: a
8   TYPE(MPI_Status)           :: msgstatus
9   integer                     :: rank
10  TYPE(MPI_Datatype)          :: type_column
11
12  call MPI_INIT()
13  call MPI_COMM_RANK(MPI_COMM_WORLD, rank)
14
15  ! Initialization of the matrix on each process
16  a(:, :) = real(rank)
17
18  ! Definition of the type_column datatype
19  call MPI_TYPE_CONTIGUOUS(nb_lines, MPI_REAL, type_column)
20
21  ! Validation of the type_column datatype
22  call MPI_TYPE_COMMIT(type_column)
```



# Derived datatypes

```
23  ! Sending of the first column
24  if ( rank == 0 ) then
25      call MPI_SEND(a(1,1),1,type_column,1,tag,MPI_COMM_WORLD)
26
27  ! Reception in the last column
28  elseif ( rank == 1 ) then
29      call MPI_RECV(a(1,nb_columns),nb_lines,MPI_REAL,0,tag,&
30                  MPI_COMM_WORLD,msgstatus)
31  end if
32
33  ! Free the datatype
34  call MPI_TYPE_FREE(type_column)
35
36  call MPI_FINALIZE()
37
38  end program column
```

# Derived datatypes

```
1 program line
2 use mpi_f08
3 implicit none
4
5 integer, parameter          :: nb_lines=5,nb_columns=6
6 integer, parameter        :: tag=100
7 real, dimension(nb_lines,nb_columns) :: a
8 TYPE(MPI_Status)          :: msgstatus
9 integer                    :: rank
10 TYPE(MPI_Datatype)        :: type_line
11
12 call MPI_INIT()
13 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
14
15 ! Initialization of the matrix on each process
16 a(:, :) = real(rank)
17
18 ! Definition of the datatype type_line
19 call MPI_TYPE_VECTOR(nb_columns,1,nb_lines,MPI_REAL,type_line)
20
21 ! Validation of the datatype type_line
22 call MPI_TYPE_COMMIT(type_line)
```

# Derived datatypes

```
23  ! Sending of the second line
24  if ( rank == 0 ) then
25      call MPI_SEND(a(2,1),nb_columns,MPI_REAL,1,tag,MPI_COMM_WORLD)
26
27  ! Reception in the next to last line
28  elseif ( rank == 1 ) then
29      call MPI_RECV(a(nb_lines-1,1),1,type_line,0,tag,&
30                  MPI_COMM_WORLD,msgstatus)
31  end if
32
33  ! Free the datatype type_line
34  call MPI_TYPE_FREE(type_line)
35
36  call MPI_FINALIZE()
37
38  end program line
```

# Derived datatypes

```
1 program block
2   use mpi_f08
3   implicit none
4
5   integer, parameter           :: nb_lines=5,nb_columns=6
6   integer, parameter           :: tag=100
7   integer, parameter           :: nb_lines_block=2,nb_columns_block=3
8   real, dimension(nb_lines,nb_columns) :: a
9   TYPE(MPI_Status)             :: msgstatus
10  integer                       :: rank
11  TYPE(MPI_Datatype)            :: type_block
12
13  call MPI_INIT()
14  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
15
16  ! Initialization of the matrix on each process
17  a(:, :) = real(rank)
18
19  ! Creation of the datatype type_bloc
20  call MPI_TYPE_VECTOR(nb_columns_block,nb_lines_block,nb_lines,&
21                     MPI_REAL,type_block)
22
23  ! Validation of the datatype type_block
24  call MPI_TYPE_COMMIT(type_block)
```

# Derived datatypes

```
25  ! Sending of a block
26  if ( rank == 0 ) then
27      call MPI_SEND(a(1,1),1,type_block,1,tag,MPI_COMM_WORLD)
28
29  ! Reception of the block
30  elseif ( rank == 1 ) then
31      call MPI_RECV(a(nb_lines-1,nb_columns-2),1,type_block,0,tag,&
32                  MPI_COMM_WORLD,msgstatus)
33  end if
34
35  ! Freeing of the datatype type_block
36  call MPI_TYPE_FREE(type_block)
37
38  call MPI_FINALIZE()
39
40  end program block
```

# Derived datatypes

## Homogenous datatypes of variable strides

- `MPI_Type_indexed()` allows creating a data structure composed of a sequence of blocks containing a variable number of elements separated by a variable stride in memory. The stride is given in number of `elements`.
- `MPI_Type_create_hindexed()` has the same functionality as `MPI_Type_indexed()` except that the strides separating two data blocks are given in `bytes`.  
This subroutine is useful when the old datatype is not an MPI base datatype (`MPI_INTEGER`, `MPI_REAL`, ...). We cannot therefore give the stride in number of elements of the old 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.

## Derived datatypes

nb=3, blocks\_lengths=(2,1,3), displacements=(0,3,7)

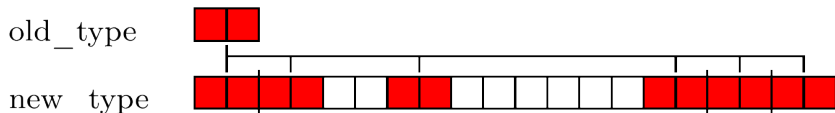


Figure 24 – The `MPI_Type_indexed` constructor

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

integer,intent(in)           :: nb
integer,intent(in),dimension(nb) :: block_lengths
! Attention the displacements are given in elements
integer,intent(in),dimension(nb) :: displacements
TYPE(MPI_Datatype), intent(in)  :: old_type
TYPE(MPI_Datatype), intent(out)  :: new_type
integer, optional, intent(out)  :: code
```

## Derived datatypes

nb=4, blocks\_lengths=(2,1,2,1), displacements=(2,10,14,24)



Figure 25 – The `MPI_Type_create_hindexed` constructor

```
MPI_TYPE_CREATE_HINDEXED(nb, block_lengths,displacements,  
                          old_type,new_type,code)  
  
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  
TYPE (MPI_Datatype), intent (in)    :: old_type  
TYPE (MPI_Datatype), intent (out)   :: new_type  
integer, optional, intent (out)     :: code
```



# Derived datatypes

## Example : triangular matrix

In the following example, each of the two processes :

1. Initializes its matrix (positive growing numbers on process 0 and negative decreasing numbers on process 1).
2. Constructs its datatype : triangular matrix (superior for the process 0 and inferior for the process 1).
3. Sends its triangular matrix to the other process and receives back a triangular matrix which it stores in the same place which was occupied by the sent matrix. This is done with the `MPI_Sendrecv_replace()` subroutine.
4. Frees its resources and exits MPI.

# Derived datatypes

BEFORE

Processus 0

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

AFTER

1	-2	-3	-5	-8	-14	-22	-32
2	10	-4	-6	-11	-15	-23	-38
3	11	19	-7	-12	-16	-24	-39
4	12	20	28	-13	-20	-29	-40
5	13	21	29	37	-21	-30	-47
6	14	22	30	38	46	-31	-48
7	15	23	31	39	47	55	-56
8	16	24	32	40	48	56	64

Processus 1

-1	-9	-17	-25	-33	-41	-49	-57
-2	-10	-18	-26	-34	-42	-50	-58
-3	-11	-19	-27	-35	-43	-51	-59
-4	-12	-20	-28	-36	-44	-52	-60
-5	-13	-21	-29	-37	-45	-53	-61
-6	-14	-22	-30	-38	-46	-54	-62
-7	-15	-23	-31	-39	-47	-55	-63
-8	-16	-24	-32	-40	-48	-56	-64

-1	-9	-17	-25	-33	-41	-49	-57
9	-10	-18	-26	-34	-42	-50	-58
17	34	-19	-27	-35	-43	-51	-59
18	35	44	-28	-36	-44	-52	-60
25	36	45	52	-37	-45	-53	-61
26	41	49	53	58	-46	-54	-62
27	42	50	54	59	61	-55	-63
33	43	51	57	60	62	63	-64

# Derived datatypes

```
1 program triangle
2 use mpi_f08
3 implicit none
4 integer, parameter :: n=8, tag=100
5 real, dimension(n,n) :: a
6 TYPE(MPI_Status) :: msgstatus
7 integer :: i
8 integer :: rank
9 TYPE(MPI_Datatype) :: type_triangle
10 integer, dimension(n) :: block_lengths, displacements
11
12 call MPI_INIT()
13 call MPI_COMM_RANK(MPI_COMM_WORLD, rank)
14
15 ! Initialization of the matrix on each process
16 a(:, :) = reshape( (/ sign(i, -rank), i=1, n*n) /), (/n, n/)
17
18 ! Creation of the triangular matrix datatype sup for process 0
19 ! and of the inferior triangular matrix datatype for process 1
20 if (rank == 0) then
21   block_lengths(:) = (/ (i-1, i=1, n) /)
22   displacements(:) = (/ (n*(i-1), i=1, n) /)
23 else
24   block_lengths(:) = (/ (n-i, i=1, n) /)
25   displacements(:) = (/ (n*(i-1)+i, i=1, n) /)
26 endif
27
28 call MPI_TYPE_INDEXED(n, block_lengths, displacements, MPI_REAL, type_triangle)
29 call MPI_TYPE_COMMIT(type_triangle)
30
31 ! Permutation of the inferior and superior triangular matrices
32 call MPI_SENDRECV_REPLACE(a, 1, type_triangle, mod(rank+1, 2), tag, mod(rank+1, 2), &
33   tag, MPI_COMM_WORLD, msgstatus)
34
35 ! Freeing of the triangle datatype
36 call MPI_TYPE_FREE(type_triangle)
37 call MPI_FINALIZE()
38 end program triangle
```

# Derived datatypes

## Size of datatype

- `MPI_Type_size()` returns the number of bytes needed to send a datatype. This value ignores any holes present in the datatype.

```
MPI_TYPE_SIZE(datatype, typesize, code)

TYPE(MPI_Datatype), intent(in) :: datatype
integer, intent(out)           :: typesize
integer, optional, intent(out) :: code
```

- The extent of a datatype is the memory space occupied by this datatype (in bytes). This value is used to calculate the position of the next datatype element (i.e. the stride between two successive datatype elements).

```
MPI_TYPE_GET_EXTENT(datatype, lb, extent, code)

TYPE(MPI_Datatype), intent(in)           :: datatype
integer(kind=MPI_ADDRESS_KIND), intent(out) :: lb, extent
integer, optional, intent(out)           :: code
```

## Derived datatypes

**Example 1** : `MPI_Type_indexed(2, (/2,1/), (/1,4/), MPI_INTEGER, type, code)`

Derived datatype :



Two successive elements : 

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

size = 12 (3 integers); lb = 4 (1 integer); extent = 16 (4 integers)

**Example 2** : `MPI_Type_vector(3,1,nb_lines,MPI_INTEGER,type_half_line,code)`

2D View :

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

1D View : 

1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	----	----

size = 12 (3 integers); lb = 0; extent = 44 (11 integers)

# Derived datatypes

## Modify the extent

- The extent is a datatype parameter. By default, it's the space in memory between the first and last component of a datatype (bounds included and with alignment considerations). We can modify the extent to create a new datatype by adapting the preceding one using `MPI_Type_create_resized()`. This provides a way to choose the stride between two successive datatype elements.

```
MPI_TYPE_CREATE_RESIZED (old, lb, extent, new, code)

TYPE(MPI_Datatype), intent(in)           :: old
integer(kind=MPI_ADDRESS_KIND), intent(in) :: lb, extent
TYPE(MPI_Datatype), intent(out)          :: new
integer, optional, intent(out)           :: code
```

# Derived datatypes

```
1 program half_line
2   USE mpi_f08
3   IMPLICIT NONE
4   INTEGER, PARAMETER :: nb_lines=5, nb_columns=6, &
5                       half_line_size=nb_columns/2, tag=1000
6   INTEGER, DIMENSION(nb_lines, nb_columns) :: A
7   TYPE(MPI_Datatype) :: typeHalfLine, typeHalfLine2
8   INTEGER :: size_integer, rank, i
9   INTEGER(kind=MPI_ADDRESS_KIND) :: lb=0, extent, sizeDisplacement
10  TYPE(MPI_Status) :: msgstatus
11
12  CALL MPI_INIT()
13  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank)
14
15  !Initialization of the A matrix on each process
16  A(:, :) = RESHAPE( (/ (SIGN(i, -rank), i=1, nb_lines*nb_columns) /), &
17                  (/ nb_lines, nb_columns /) )
18
19  !Construction of the derived datatype typeHalfLine
20  CALL MPI_TYPE_VECTOR(half_line_size, 1, nb_lines, MPI_INTEGER, typeHalfLine)
21
22  !Know the size of the datatype MPI_INTEGER
23  CALL MPI_TYPE_SIZE(MPI_INTEGER, size_integer)
24
25  ! Information on type typeHalfLine
26  call MPI_TYPE_GET_EXTENT(typeHalfLine, lb, extent)
27  if (rank == 0) print *, "typeHalfLine: lb=", lb, ", extent=", extent
28
29  !Construction of the derived datatype typeHalfLine2
30  sizeDisplacement = size_integer
31  CALL MPI_TYPE_CREATE_RESIZED(typeHalfLine, lb, sizeDisplacement, &
32                              typeHalfLine2)
```

# Derived datatypes

```
33  ! Information on type typeHalfLine2
34  call MPI_TYPE_GET_EXTENT(typeHalfLine2,lb,extent)
35  if (rank == 0) print *, "typeHalfLine2: lb=",lb," , extent=",extent
36
37  !Validation of the datatype typeHalfLine2
38  CALL MPI_TYPE_COMMIT(typeHalfLine2)
39
40  IF (rank == 0) THEN
41    !Sending of the A matrix to the process 1 with the derived datatype typeHalfLine2
42    CALL MPI_SEND(A(1,1), 2, typeHalfLine2, 1, tag, &
43                MPI_COMM_WORLD)
44  ELSE
45    !Reception for the process 1 in the A matrix
46    CALL MPI_RECV(A(1,nb_columns-1), 6, MPI_INTEGER, 0, tag,&
47                MPI_COMM_WORLD,msgstatus)
48    PRINT *, 'A matrix on the process 1'
49    DO i=1,nb_lines
50      PRINT *,A(i,:)
51    END DO
52  END IF
53
54  CALL MPI_FINALIZE()
55  END PROGRAM half_line
```

```
> mpiexec -n 2 half_line
typeHalfLine: lb=0, extent=44
typeHalfLine2: lb=0, extent=4
```

```
A matrix on the process 1
-1 -6 -11 -16 1 12
-2 -7 -12 -17 6 -27
-3 -8 -13 -18 11 -28
-4 -9 -14 -19 2 -29
-5 -10 -15 -20 7 -30
```

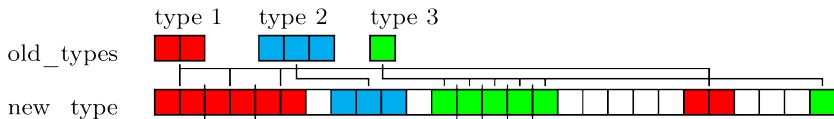


# Derived datatypes

## Heterogenous datatype

- `MPI_Type_create_struct()` call allows creating a set of data blocks indicating the `type`, the `count` and the `displacement` of each block.
- It is the most general datatype constructor. It further generalizes `MPI_Type_indexed()` by allowing a different datatype for each block.

`nb=5`, `blocks_lengths=(3,1,5,1,1)`, `displacements=(0,7,11,21,26)`,  
`old_types=(type1,type2,type3,type1,type3)`



```
MPI_TYPE_CREATE_STRUCT(nb,blocks_lengths,displacements,  
old_types,new_type,code)
```

```
integer,intent(in) :: nb  
integer,intent(in), dimension(nb) :: blocks_lengths  
integer(kind=MPI_ADDRESS_KIND), intent(in), dimension(nb) :: displacements  
TYPE(MPI_Datatype), intent(in), dimension(nb) :: old_types  
TYPE(MPI_Datatype), intent(out) :: new_type  
integer, optional, intent(out) :: code
```

# Derived datatypes

## Compute displacements

- `MPI_Type_create_struct()` is useful for creating MPI datatypes corresponding to Fortran derived datatypes or to C structures.
- The memory alignment of heterogeneous data structures is different for each architecture and each compiler.
- Warning, you have to check the extent of the MPI datatypes obtained.
- `MPI_Get_address()` provides the address of a variable. It's equivalent of `&` operator in C.
- Warning, even in C, it is better to use this subroutine for portability reasons.
- It's advised to use `MPI_Aint_add()` and `MPI_Aint_diff()` to add or subtract addresses.

```
MPI_GET_ADDRESS(variable,address_variable,code)

TYPE(*), dimension(..), asynchronous      :: variable
integer(kind=MPI_ADDRESS_KIND), intent(out) :: address_variable
integer, optional, intent(out)            :: code

integer(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(addr1, addr2)
integer(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(addr1, addr2)

integer(KIND=MPI_ADDRESS_KIND)           :: addr1, addr2
```

# Derived datatypes

```
1 program Interaction_Particles
2
3 use mpi_f08
4 implicit none
5
6 integer, parameter :: n=1000,tag=100
7 integer, dimension(MPI_STATUS_SIZE) :: msgstatus
8 integer :: rank,i
9 TYPE(MPI_Datatype) :: type_particle
10 TYPE(MPI_Datatype), dimension(4) :: types
11 integer, dimension(4) :: blocks_lengths
12 integer(kind=MPI_ADDRESS_KIND), dimension(5) :: displacements,addresses
13 integer(kind=MPI_ADDRESS_KIND) :: lb,extent
14
15 type Particule
16 character(len=5) :: category
17 integer :: mass
18 real, dimension(3) :: coords
19 logical :: class
20 end type Particule
21 type(Particule), dimension(n) :: p,temp_p
22
23 call MPI_INIT()
24 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
25
26 ! Construction of the datatype
27 types = (/MPI_CHARACTER,MPI_INTEGER,MPI_REAL,MPI_LOGICAL/)
28 blocks_lengths= (/5,1,3,1/)
```

# Derived datatypes

```
29  call MPI_GET_ADDRESS (p(1), addresses(1))
30  call MPI_GET_ADDRESS (p(1)%category, addresses(2))
31  call MPI_GET_ADDRESS (p(1)%mass, addresses(3))
32  call MPI_GET_ADDRESS (p(1)%coords, addresses(4))
33  call MPI_GET_ADDRESS (p(1)%class, addresses(5))
34  ! Calculation of displacements relative to the start address
35  do i=1,4
36      displacements(i)=MPI_AINT_DIFF (addresses(i+1), addresses(1))
37  end do
38  call MPI_TYPE_CREATE_STRUCT (4, blocks_lengths, displacements, types, temp)
39  call MPI_GET_ADDRESS (p(2), addresses(2))
40  lb = 0
41  extent = MPI_AINT_DIFF (addresses(2), addresses(1))
42  call MPI_TYPE_CREATE_RESIZED (temp, lb, extent, type_particle)
43  ! Validation of the structured datatype
44  call MPI_TYPE_COMMIT (type_particle)
45  ! Initialization of particles for each process
46  ....
47  ! Sending of particles from 0 towards 1
48  if (rank == 0) then
49      call MPI_SEND (p(1), n, type_particle, 1, tag, MPI_COMM_WORLD)
50  else
51      call MPI_RECV (temp_p(1), n, type_particle, 0, tag, MPI_COMM_WORLD, &
52                  msgstatus)
53  endif
54
55  ! Freeing of the datatype
56  call MPI_TYPE_FREE (type_particle)
57  call MPI_FINALIZE ()
58  end program Interaction_Particles
```

# Derived datatypes

## Conclusion

- The MPI derived datatypes are powerful data description portable mechanisms.
- When they are combined with subroutines like `MPI_Sendrecv()`, they allow simplifying the writing of interprocess exchanges.
- The combination of derived datatypes and topologies (described in one of the next chapters) makes MPI the ideal tool for all domain decomposition problems with both regular or irregular meshes.

# Derived datatypes

## Memento

Subroutines	blocks_lengths	strides	old_types
<code>MPI_Type_Contiguous()</code>	constant*	constant*	constant
<code>MPI_Type_[Create_H]Vector()</code>	constant	constant	constant
<code>MPI_Type_[Create_H]Indexed()</code>	<i>variable</i>	<i>variable</i>	constant
<code>MPI_Type_Create_Struct()</code>	<i>variable</i>	<i>variable</i>	<i>variable</i>

(\*) hidden parameter, equal to 1

## MPI Hands-On – Exercise 4 : Matrix transpose

- The goal of this exercise is to practice with the derived datatypes.
- $A$  is a matrix with 5 lines and 4 columns defined on the process 0.
- Process 0 sends its  $A$  matrix to process 1 and transposes this matrix during the send.

1.	6.	11.	16.
2.	7.	12.	17.
3.	8.	13.	18.
4.	9.	14.	19.
5.	10.	15.	20.

Processus 0



1.	2.	3.	4.	5.
6.	7.	8.	9.	10.
11.	12.	13.	14.	15.
16.	17.	18.	19.	20.

Processus 1

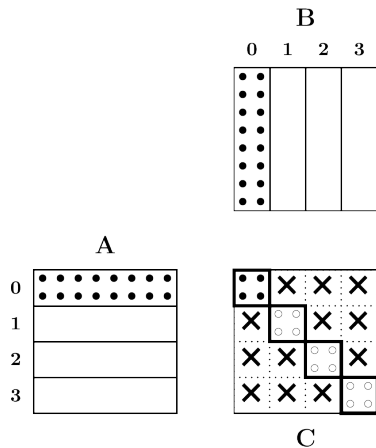
- To do this, we need to create two derived datatypes, a derived datatype `type_line` and a derived datatype `type_transpose`.

## MPI Hands-On – Exercise 5 : Matrix-matrix product

- Collective communications : matrix-matrix product  $C = A \times B$ 
  - The matrixes are square and their sizes are a multiple of the number of processes.
  - The matrixes  $A$  and  $B$  are defined on process 0. Process 0 sends a horizontal slice of matrix  $A$  and a vertical slice of matrix  $B$  to each process. Each process then calculates its diagonal block of matrix  $C$ .
  - To calculate the non-diagonal blocks, each process sends to the other processes its own slice of  $A$ .
  - At the end, process 0 gathers and verifies the results.



## MPI Hands-On – Exercise 5 : Matrix-matrix product



**Figure 26** – Distributed matrix product

## MPI Hands-On – Exercise 5 : Matrix-matrix product

- The algorithm that may seem the most immediate and the easiest to program, consisting of each process sending its slice of its matrix  $A$  to each of the others, does not perform well because the communication algorithm is not well-balanced. It is easy to see this when doing performance measurements and graphically representing the collected traces.

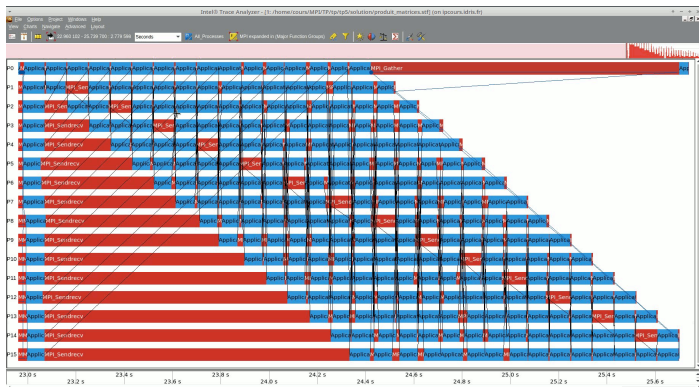


Figure 27 – Parallel matrix product on 16 processes, for a matrix size of 1024 (first algorithm)

## MPI Hands-On – Exercise 5 : Matrix-matrix product

- Changing the algorithm in order to *shift* slices from process to process, we obtain a perfect balance between calculations and communications and have a speedup of 2 compared to the naive algorithm.

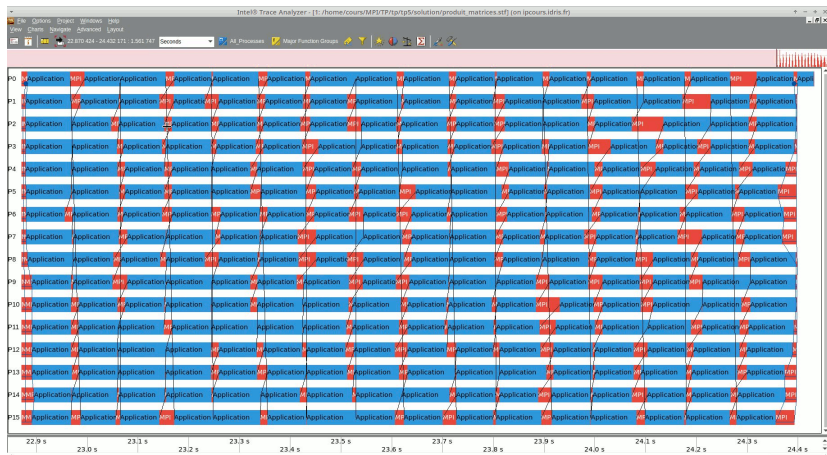


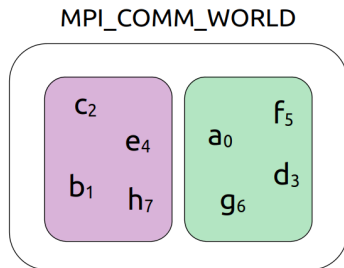
Figure 28 – Parallel matrix product on 16 processes, for a matrix size of 1024 (second algorithm)

# Communicators

# Communicators

## Introduction

The purpose of communicators is to create subgroups on which we can carry out operations such as collective or point-to-point communications. Each subgroup will have its own communication space.



**Figure 29** – Communicator partitioning

# Communicators

## Example

For example, we want to broadcast a collective message to even-ranked processes and another message to odd-ranked processes.

- Looping on *send/recv* can be very detrimental especially if the number of processes is high. Also a test inside the loop would be compulsory in order to know if the sending process must send the message to an even or odd process rank.
- A solution is to create a communicator containing the even-ranked processes, another containing the odd-ranked processes, and initiate the collective communications inside these groups.

# Communicators

## Default communicator

- A communicator can only be created from another communicator. The first one will be created from the `MPI_COMM_WORLD`.
- After the `MPI_Init()` call, a communicator is created for the duration of the program execution.
- Its identifier `MPI_COMM_WORLD` is a variable defined in the header files.
- This communicator can only be destroyed via a call to `MPI_Finalize()`.
- By default, therefore, it sets **the scope** of collective and point-to-point communications **to include all the processes** of the application.

# Communicators

## Groups and communicators

- A communicator consists of :
  - A **group**, which is an ordered group of processes.
  - A communication **context** put in place by calling one of the communicator construction subroutines, which allows determination of the communication space.
- The communication contexts are managed by MPI (the programmer has no action on them : It is a hidden attribute).
- In the MPI library, the following subroutines exist for the purpose of building communicators : `MPI_Comm_create()` , `MPI_Comm_dup()` ,  
`MPI_Comm_split()`
- The **communicator constructors** are **collective calls**.
- Communicators created by the programmer can be destroyed by using the `MPI_Comm_free()` subroutine.

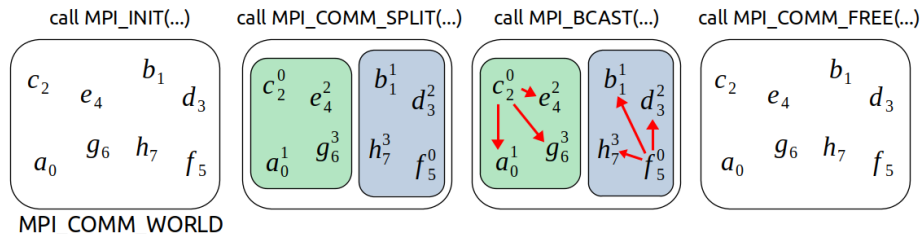


# Communicators

## Partitioning of a communicator

In order to solve the problem example :

- Partition the communicator into odd-ranked and even-ranked processes.
- Broadcast a message inside the odd-ranked processes and another message inside the even-ranked processes.



**Figure 30** – Communicator creation/destruction

# Communicators

## Partitioning of a communicator with `MPI_Comm_split()`

The `MPI_Comm_split()` subroutine allows :

- Partitioning a given communicator into as many communicators as we want.
- Giving the same name to all these communicators : The process value will be the value of its communicator.
- Method :
  1. Define a colour value for each process, associated with its communicator number.
  2. Define a key value for ordering the processes in each communicator
  3. Create the partition where each communicator is called `new_comm`

```
MPI_COMM_SPLIT(comm,color,key,new_comm,code)
```

```
TYPE(MPI_Comm), intent(in)      :: comm  
integer, intent(in)             :: color, key  
TYPE(MPI_Comm), intent(out)     :: new_comm  
integer, optional, intent(out)  :: code
```

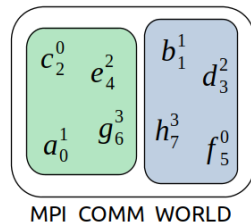
A process which assigns a color value equal to `MPI_UNDEFINED` will have the invalid communicator `MPI_COMM_NULL` for `new_com`.

# Communicators

## Example

Let's look at how to proceed in order to build the communicator which will subdivide the communication space into odd-ranked and even-ranked processes via the `MPI_Comm_split()` constructor.

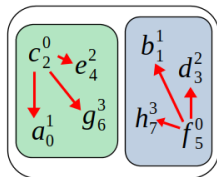
process	a	b	c	d	e	f	g	h
rank_world	0	1	2	3	4	5	6	7
color	0	1	0	1	0	1	0	1
key	0	1	-1	3	4	-1	6	7
rank_even_odd	1	1	0	2	2	0	3	3



**Figure 31** – Construction of the `ComEvenOdd` communicator with `MPI_Comm_split()`

# Communicators

```
1 program EvenOdd
2   use mpi_f08
3   implicit none
4
5   integer, parameter :: m=16
6   integer           :: key
7   TYPE(MPI_Comm)   :: CommEvenOdd
8   integer           :: rank_in_world
9   real, dimension(m) :: a
10
11  call MPI_INIT()
12  call MPI_COMM_RANK(MPI_COMM_WORLD,rank_in_world)
13
14  ! Initialization of the A vector
15  a(:)=0.
16  if(rank_in_world == 2) a(:)=2.
17  if(rank_in_world == 5) a(:)=5.
18
19  key = rank_in_world
20  if (rank_in_world == 2 .OR. rank_in_world == 5 ) then
21    key=-1
22  end if
23
24  ! Creation of even and odd communicators by giving them the same name
25  call MPI_COMM_SPLIT(MPI_COMM_WORLD,mod(rank_in_world,2),key,CommEvenOdd)
26
27  ! Broadcast of the message by the rank process 0 of each communicator to the processes
28  ! of its group
29  call MPI_BCAST(a,m,MPI_REAL,0,CommEvenOdd)
30
31  ! Destruction of the communicators
32  call MPI_COMM_FREE(CommEvenOdd)
33  call MPI_FINALIZE()
34  end program EvenOdd
```

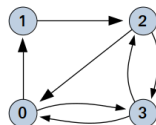


# Communicators

## Topologies

- In most applications, especially in domain decomposition methods where we match the calculation domain to the process grid, it is helpful to be able to arrange the processes according to a regular topology.
- MPI allows defining virtual cartesian or graph topologies.
  - Cartesian topologies :
    - ▶ Each process is defined in a grid.
    - ▶ Each process has a neighbour in the grid.
    - ▶ The grid can be periodic or not.
    - ▶ The processes are identified by their coordinates in the grid.
  - Graph topologies :
    - ▶ Can be used in more complex topologies.

1	3	5	7
0	2	4	6



**Figure 32** – A 2D Cartesian topology (left) and a Graph topology (right)

## Cartesian topologies

- A Cartesian topology is defined from a given communicator named `comm_old`, calling the `MPI_Cart_create()` subroutine.
- We define :
  - An integer `ndims` representing the number of grid dimensions.
  - An integer array `dims` of dimension `ndims` showing the number of processes in each dimension.
  - An array of `ndims` `logicals` which shows the periodicity of each dimension.
  - A logical `reorder` which shows if the process numbering can be changed by MPI.

```
MPI_CART_CREATE(comm_old, ndims,dims,periods,reorder,comm_new,code)
```

```
TYPE(MPI_Comm), intent(in)           :: comm_old  
integer, intent(in)                  :: ndims  
integer, dimension(ndims), intent(in) :: dims  
logical, dimension(ndims), intent(in) :: periods  
logical, intent(in)                   :: reorder  
TYPE(MPI_Comm), intent(out)           :: comm_new  
integer, optional, intent(out)        :: code
```

# Communicators

## Example

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

```
use mpi_f08
integer                :: comm_2D
integer, parameter    :: ndims = 2
integer, dimension(ndims) :: dims
logical, dimension(ndims) :: periods
logical                :: reorder

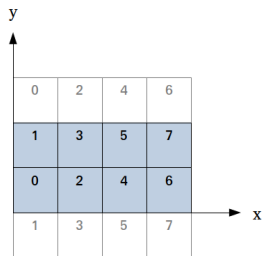
.....

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

call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, dims, periods, reorder, comm_2D)
```

If `reorder = .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 `reorder = .true.`, the MPI implementation chooses the order of the processes.



**Figure 33** – A 2D periodic Cartesian topology in y



# Communicators

## 3D Example

Example on a 3D grid having 4 domains along x, 2 along y and 2 along z, non periodic.

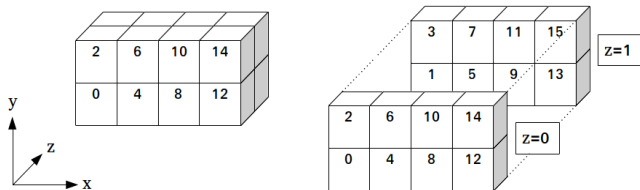
```
use mpi_f08
TYPE(MPI_Comm)      :: comm_3D
integer, parameter  :: ndims = 3
integer, dimension(ndims) :: dims
logical, dimension(ndims) :: periods
logical              :: reorder

.....

dims(1) = 4
dims(2) = 2
dims(3) = 2
periods(:) = .false.
reorder = .false.

call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, dims, periods, reorder, comm_3D)
```

# Communicators



**Figure 34** – A 3D non-periodic Cartesian topology

# Communicators

## Process distribution

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

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

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

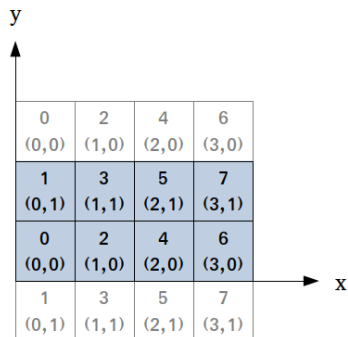
Remark : If the values of `dims` in entry are all 0, then we leave to MPI the choice of the number of processes in each direction according to the total number of processes.

dims in entry	call <code>MPI_Dims_create</code>	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

# Communicateurs

## Rank and coordinates of a process

In a Cartesian topology, the rank of each process is associated with its coordinates in the grid.



**Figure 35** – A 2D periodic Cartesian topology in y

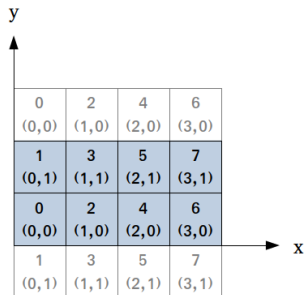
## Rank of a process

In a Cartesian topology, the `MPI_Cart_rank()` subroutine returns the rank of the associated process to the coordinates in the grid.

```
MPI_CART_RANK(comm, coords, rank, code)

TYPE(MPI_Comm), intent(in)           :: comm
integer, dimension(ndims), intent(in) :: coords
integer, intent(out)                 :: rank
integer, optional, intent(out)       :: code
```

# Communicators



**Figure 36** – A 2D periodic Cartesian topology in y

```
coords(1)=dims(1)-1
do i=0,dims(2)-1
  coords(2) = i
  call MPI_CART_RANK(comm_2D,coords,rank(i))
end do
.....
i=0,in entry coords=(3,0),in exit rank(0)=6.
i=1,in entry coords=(3,1),in exit rank(1)=7.
```

# Communicators

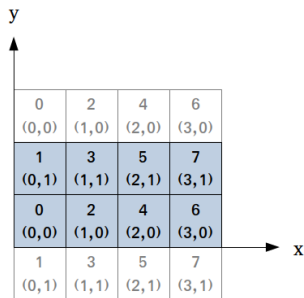
## Coordinates of a process

In a cartesian topology, the `MPI_Cart_coords()` subroutine returns the coordinates of a process of a given rank in the grid.

```
MPI_CART_COORDS(comm, rank, ndims, coords, code)

TYPE(MPI_Comm), intent(in)           :: comm
integer, intent(in)                  :: rank, ndims
integer, dimension(ndims), intent(out) :: coords
integer, optional, intent(out)       :: code
```

# Communicators



**Figure 37** – A 2D periodic Cartesian topology in y

```
if (mod(rank,2) == 0) then
  call MPI_CART_COORDS(comm_2D,rank,2,coords)
end if
.....
In entry, the rank values are : 0,2,4,6.
In exit, the coords values are :
(0,0), (1,0), (2,0), (3,0)
```



# Communicators

## Rank of neighbours

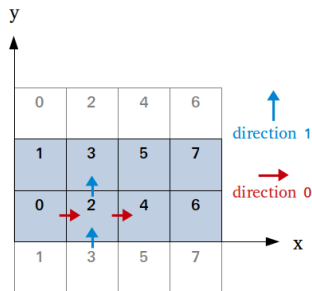
In a Cartesian topology, a process that calls the `MPI_Cart_shift()` subroutine can obtain the rank of a neighboring process in a given direction.

```
MPI_CART_SHIFT(comm, direction, step, rank_previous, rank_next, code)

TYPE(MPI_Comm), intent(in)      :: comm
integer, intent(in)             :: direction, step
integer, intent(out)            :: rank_previous, rank_next
integer, optional, intent(out)  :: code
```

- The `direction` parameter corresponds to the displacement axis (xyz).
- The `step` parameter corresponds to the displacement step.
- If a rank does not have a neighbor before (or after) in the requested direction, then the value of the previous (or following) rank will be `MPI_PROC_NULL`.

# Communicators



**Figure 38** – Call of the `MPI_Cart_shift()` subroutine

```
call MPI_CART_SHIFT(comm_2D,0,1,rank_left,rank_right)
```

```
.....  
For the process 2, rank_left=0, rank_right=4
```

```
call MPI_CART_SHIFT(comm_2D,1,1,rank_low,rank_high)
```

```
.....  
For the process 2, rank_low=3, rank_high=3
```

# Communicators

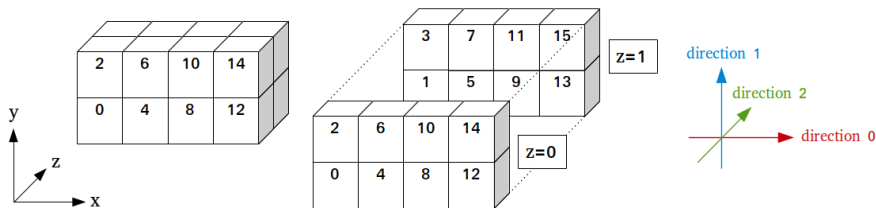


Figure 39 – Call of the MPI\_Cart\_shift() subroutine

```
call MPI_CART_SHIFT(comm_3D,0,1,rank_left,rank_right)
.....
For the process 0, rank_left=-1, rank_right=4
```

```
call MPI_CART_SHIFT(comm_3D,1,1,rank_low,rank_high)
.....
For the process 0, rank_low=-1, rank_high=2
```

```
call MPI_CART_SHIFT(comm_3D,2,1,rank_ahead,rank_before)
.....
For the process 0, rank_ahead=-1, rank_before=1
```

# Communicators

## Example

- create a 2D Cartesian grid periodic in y
- get coordinates of each process
- get neighbours ranks for each process

```
1 program decomposition
2 use mpi
3 implicit none
4
5 integer                :: rank_in_topo,nb_procs
6 TYPE(MPI_Comm)        :: comm_2D
7 integer, dimension(4)  :: neighbor
8 integer, parameter     :: N=1,E=2,S=3,W=4
9 integer, parameter     :: ndims = 2
10 integer, dimension (ndims) :: dims,coords
11 logical, dimension (ndims) :: periods
12 logical               :: reorder
13
14 call MPI_INIT()
15
16 call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
17
18 ! Know the number of processes along x and y
19 dims(:) = 0
20
21 call MPI_DIMS_CREATE(nb_procs,ndims,dims)
```

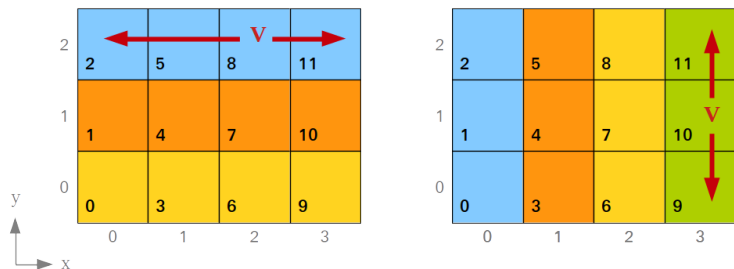
# Communicators

```
22      ! 2D y-periodic grid creation
23      periods(1) = .false.
24      periods(2) = .true.
25      reorder = .false.
26
27      call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, dims, periods, reorder, comm_2D)
28
29      ! Know my coordinates in the topology
30      call MPI_COMM_RANK(comm_2D, rank_in_topo)
31      call MPI_CART_COORDS(comm_2D, rank_in_topo, ndims, coords)
32
33      ! Search of my West and East neighbors
34      call MPI_CART_SHIFT(comm_2D, 0, 1, neighbor(W), neighbor(E))
35
36      ! Search of my South and North neighbors
37      call MPI_CART_SHIFT(comm_2D, 1, 1, voisin(S), voisin(N))
38
39      call MPI_FINALIZE()
40
41 end program decomposition
```

## Subdividing a Cartesian topology

- The goal, by example, is to degenerate a 2D or 3D cartesian topology into, respectively, a 1D or 2D Cartesian topology.
- For MPI, degenerating a 2D Cartesian topology creates as many communicators as there are rows or columns in the initial Cartesian grid. For a 3D Cartesian topology, there will be as many communicators as there are planes.
- The major advantage is to be able to carry out collective operations limited to a subgroup of processes belonging to :
  - the same row (or column), if the initial topology is 2D ;
  - the same plane, if the initial topology is 3D.

# Communicators



**Figure 40** – Two examples of data distribution in a degenerated 2D topology

# Communicators

## Subdividing a Cartesian topology

There are two ways to degenerate a topology :

- By using the `MPI_Comm_split()` general subroutine
- By using the `MPI_Cart_sub()` subroutine designed for this purpose

```
MPI_CART_SUB(CommCart,remain_dims,CommCartD,code)

logical, intent(in), dimension(NDim) :: remain_dims
TYPE(MPI_Comm), intent(in)           :: CommCart
TYPE(MPI_Comm), intent(out)          :: CommCartD
integer, optional, intent(out)       :: code
```

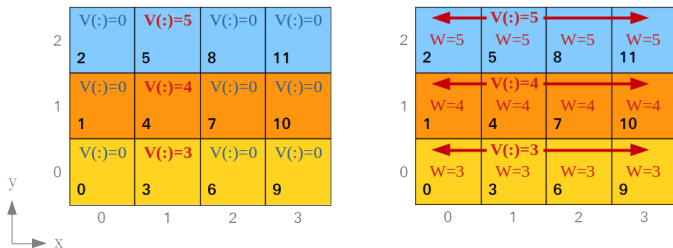


Figure 41 – Broadcast of a  $V$  array in the degenerated 2D grid.



# Communicators

```
1 program CommCartSub
2   use mpi_f08
3   implicit none
4
5   TYPE(MPI_Comm)           :: Comm2D,Comm1D
6   integer                  :: rank
7   integer,parameter        :: NDim2D=2
8   integer,dimension(NDim2D) :: Dim2D,Coord2D
9   logical,dimension(NDim2D) :: Period,remain_dims
10  logical                   :: Reorder
11  integer,parameter         :: m=4
12  real,dimension(m)         :: V=0.
13  real                       :: W=0.
```

# Communicators

```
14  call MPI_INIT()
15
16  ! Creation of the initial 2D grid
17  Dim2D(1) = 4
18  Dim2D(2) = 3
19  Period(:) = .false.
20  ReOrder = .false.
21  call MPI_CART_CREATE(MPI_COMM_WORLD, NDim2D, Dim2D, Period, ReOrder, Comm2D)
22  call MPI_COMM_RANK(Comm2D, rank)
23  call MPI_CART_COORDS(Comm2D, rank, NDim2D, Coord2D)
24
25  ! Initialization of the V vector
26  if (Coord2D(1) == 1) V(:)=real(rank)
27
28  ! Every row of the grid must be a 1D cartesian topology
29  remain_dims(1) = .true.
30  remain_dims(2) = .false.
31  ! Subdivision of the 2D cartesian grid
32  call MPI_CART_SUB(Comm2D, remain_dims, Comm1D)
33
34  ! The processes of column 2 distribute the V vector to the processes of their row
35  call MPI_SCATTER(V, 1, MPI_REAL, W, 1, MPI_REAL, 1, Comm1D)
36
37  print ' ("Rank : ", I2, " ; Coordinates : (" , I1, " , " , I1, " ) ; W = ", F2.0)', &
38      rank, Coord2D(1), Coord2D(2), W
39
40  call MPI_FINALIZE()
41  end program CommCartSub
```

# Communicators

```
> mpiexec -n 12 CommCartSub
Rank : 0 ; Coordinates : (0,0) ; W = 3.
Rank : 1 ; Coordinates : (0,1) ; W = 4.
Rank : 3 ; Coordinates : (1,0) ; W = 3.
Rank : 8 ; Coordinates : (2,2) ; W = 5.
Rank : 4 ; Coordinates : (1,1) ; W = 4.
Rank : 5 ; Coordinates : (1,2) ; W = 5.
Rank : 6 ; Coordinates : (2,0) ; W = 3.
Rank : 10 ; Coordinates : (3,1) ; W = 4.
Rank : 11 ; Coordinates : (3,2) ; W = 5.
Rank : 9 ; Coordinates : (3,0) ; W = 3.
Rank : 2 ; Coordinates : (0,2) ; W = 5.
Rank : 7 ; Coordinates : (2,1) ; W = 4.
```

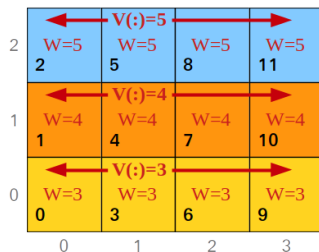
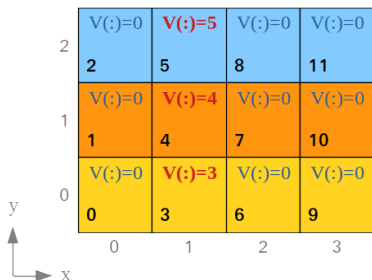
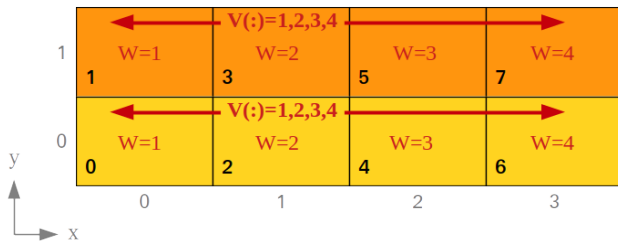


Figure 42 – Broadcast of a  $V$  array in the degenerated 2D grid.

## MPI Hands-On – Exercise 6 : Communicators

- Using the Cartesian topology defined below, subdivide in 2 communicators following the lines by calling `MPI_Comm_split()`. Then, have the processes of the second column scatter the `V` array to the processes of their line.



**Figure 43** – Subdivision of a 2D topology and communication using the obtained 1D topology

- Constraint : define the color of each process without using the *modulo* operation.

# MPI-IO

## Input/Output Optimisation

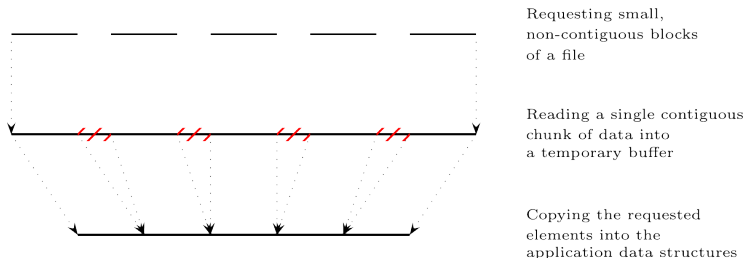
- Applications which perform large calculations also tend to handle large amounts of data and generate a significant number of I/O requests.
- Effective treatment of I/O can highly improve the global performances of applications.
- I/O tuning of parallel codes involves :
  - **Parallelizing** I/O access of the program in order to avoid serial bottlenecks and to take advantage of parallel file systems
  - Implementing **efficient** data access algorithms (nonblocking I/O)
  - Leveraging mechanisms implemented by the **operating system** (request grouping methods, I/O buffers, etc.).
- Libraries make I/O optimisations of parallel codes easier by providing ready-to-use capabilities.

## The MPI-IO interface

- The MPI-2 norm defines a set of functions designed to manage parallel I/O.
- The I/O functions use well-known MPI concepts. For instance, **collectives** and **nonblocking operations** on files and between MPI processes are similar. Files can also be accessed in a patterned way using the existing **derived datatype** functionality.
- Other concepts come from native I/O interfaces (file descriptors, attributes, ...).

## Example of a sequential optimisation implemented by I/O libraries

- I/O performance suffers considerably when making many small I/O requests.
- Access on small, non-contiguous regions of data can be optimized by grouping requests and using temporary buffers.
- Such optimisation is performed automatically by MPI-IO libraries.



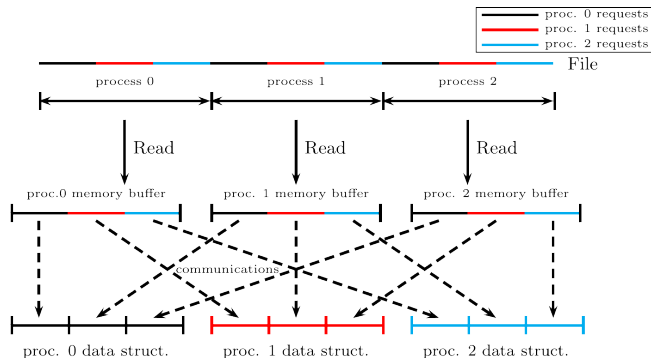
**Figure 44** – Data sieving mechanism improving I/O access on small, non-contiguous data set.



# MPI-IO

## Example of a parallel optimisation

Collective I/O access can be optimised by rebalancing the I/O operations in contiguous chunks and performing inter-process communications.



**Figure 45** – Read operation performed in two steps by a group of processes

## Open a file

```
MPI_File_open(comm, filename, amode, info, fh, code)

TYPE(MPI_Comm), intent(in)      :: comm
character(len=*), intent(in)    :: filename
integer, intent(in)            :: amode
TYPE(MPI_Info), intent(in)      :: info
TYPE(MPI_File), intent(out)     :: fh
integer, optional, intent(out)  :: code
```

- Open the file `filename` with access modes `amode` ;
- `fh` is an opaque representation of the opened file ;
- Opening is a `collective operation` ;
- `filename` and `amode` must be the same for all ranks in the `comm` communicator ;
- `MPI_Info` object are key-value database usefull for optimisation. `MPI_INFO_NULL` could be used as a default value.

## Mode

Mode	Meaning
<code>MPI_MODE_RDONLY</code>	Read only
<code>MPI_MODE_RDWR</code>	Reading and writing
<code>MPI_MODE_WRONLY</code>	Write only
<code>MPI_MODE_CREATE</code>	Create the file if it does not exist
<code>MPI_MODE_EXCL</code>	Error if creating file that already exists
<code>MPI_MODE_UNIQUE_OPEN</code>	File will not be concurrently opened elsewhere
<code>MPI_MODE_SEQUENTIAL</code>	File will only be accessed sequentially
<code>MPI_MODE_APPEND</code>	Set initial position of all file pointers to end of file
<code>MPI_MODE_DELETE_ON_CLOSE</code>	Delete file on close

Modes can be combined with IOR function or addition if each term are only used once.

## Closing a file

```
MPI_File_close(fh,code)

TYPE(MPI_File), intent(out)    :: fh
integer, optional, intent(out) :: code
```

- Close the file ;
- Closing is a *collective* operation.

# MPI-IO

```
1 program open01
2   use mpi_f08
3   implicit none
4   character (len=MPI_MAX_ERROR_STRING) :: error_text
5   TYPE(MPI_File)                        :: fh
6   integer                               :: code, error_len
7
8   call MPI_INIT()
9
10
11  call MPI_FILE_OPEN(MPI_COMM_WORLD,"file.data", &
12                    MPI_MODE_RDWR + MPI_MODE_CREATE,MPI_INFO_NULL,fh,code)
13  IF (code /= MPI_SUCCESS) THEN
14    CALL MPI_ERROR_STRING(code,error_text,error_len)
15    PRINT *, error_text(1:error_len)
16    CALL MPI_ABORT(MPI_COMM_WORLD, 42)
17  END IF
18
19  call MPI_FILE_CLOSE(fh,code)
20  IF (code /= MPI_SUCCESS) THEN
21    PRINT *, 'Error in closing file'
22    CALL MPI_ABORT(MPI_COMM_WORLD, 2)
23  END IF
24
25  call MPI_FINALIZE()
26
27 end program open01
```

```
> ls -l file.data
```

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

## Error handling

- The behavior concerning code argument is different for the IO part of MPI.
- It's necessary to check the value of this argument.
- It's possible to change this behaviour with `MPI_File_set_errhandler()`.
- Two error handlers are available : `MPI_ERRORS_ARE_FATAL` and `MPI_ERRORS_RETURN`.
- `MPI_Comm_set_errhandler()` provides a way to change the error handler for the communications.

```
MPI_FILE_SET_ERRHANDLER(fh, errhandler, code)

TYPE(MPI_File), intent(inout)  :: fh
TYPE(MPI_Errhandler), intent(in) :: errhandler
integer, optional, intent(out)  :: code
```

The default behaviour can be changed with `MPI_FILE_NULL` as file handler.

## Data access routines

- MPI-IO proposes a broad range of subroutines for transferring data between files and memory.
- Subroutines can be distinguished through several properties :
  - The **position** in the file can be specified using an **explicit offset** (ie. an absolute position relative to the beginning of the file) or using **individual** or **shared** file pointers (ie. the offset is defined by the current value of pointers).
  - Data access can be **blocking** or **nonblocking**.
  - Sending and receiving messages can be **collective** (in the communicator group) or **noncollective**.
- Different access methods may be mixed within the same program.

Positioning	Synchronism	noncollective	collective
explicit offsets	blocking	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	nonblocking	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_iread_at_all MPI_File_iwrite_at_all
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_iread_all MPI_File_iwrite_all
shared file pointer	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



## File Views

- By default, files are treated as a sequence of bytes but access patterns can also be expressed using predefined or derived MPI datatypes.
- This mechanism is called **file views** and is described in further detail later.
- For now, we only need to know that the **views** rely on an **elementary data type** and that the default type is `MPI_BYTE`.

## Explicit Offsets

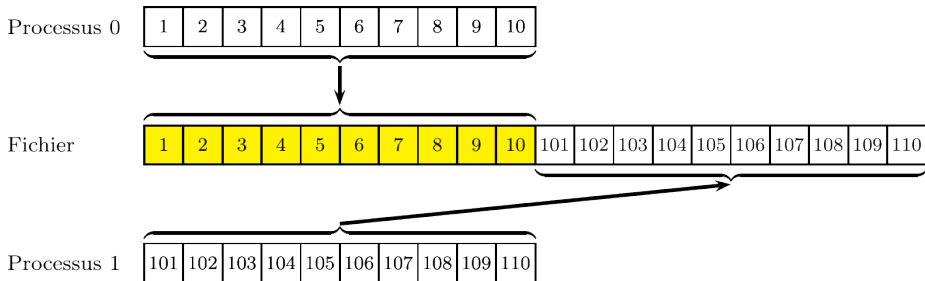
```
MPI_File_read_at(fh, offset, buf, count, datatype, iostatus, code)
MPI_File_write_at(fh, offset, buf, count, datatype, iostatus, code)

TYPE (MPI_File), INTENT(IN)           :: fh
INTEGER (KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
TYPE(*), DIMENSION(..)               :: buf
INTEGER, INTENT(IN)                   :: count
TYPE (MPI_Datatype), INTENT(IN)      :: datatype
TYPE (MPI_Status)                     :: iostatus
INTEGER, OPTIONAL, INTENT(OUT)        :: code
```

- Write/read at offset `offset` in the file `fh`, `count` element of type `datatype` from address `buf`;
- The `offset` is expressed as a multiple of the `elementary data type` of the current view (therefore, the default offset unit is bytes).
- The `datatype` size must be a multiple of the elementary datatype.

```
1 program write_at
2   use mpi_f08
3   implicit none
4   integer, parameter           :: nb_values=10
5   integer                     :: i,rank,code,bytes_in_integer
6   TYPE(MPI_File)              :: fh
7   integer(kind=MPI_OFFSET_KIND) :: offset
8   integer, dimension(nb_values) :: values
9   TYPE(MPI_Status)           :: iostatus
10
11 call MPI_INIT()
12 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
13 values(:)= (/ (i+rank*100,i=1,nb_values)/)
14 print *, "process",rank, ":",values(:)
15
16 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_WRONLY + MPI_MODE_CREATE, &
17                   MPI_INFO_NULL,fh,code)
18 IF (code /= MPI_SUCCESS) THEN
19   PRINT *, 'Error in opening file'
20   CALL MPI_ABORT(MPI_COMM_WORLD, 42)
21 END IF
22 call MPI_TYPE_SIZE(MPI_INTEGER,bytes_in_integer)
23 offset=rank*nb_values*bytes_in_integer
24
25 call MPI_FILE_SET_ERRHANDLER(fh,MPI_ERRORS_ARE_FATAL)
26 call MPI_FILE_WRITE_AT(fh,offset,values,nb_values,MPI_INTEGER, &
27                       iostatus)
28
29 call MPI_FILE_CLOSE(fh)
30 call MPI_FINALIZE()
31 end program write_at
```

# MPI-IO



**Figure 46** – MPI\_File\_write\_at()

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

```
1 program read_at
2
3 use mpi_f08
4 implicit none
5
6 integer, parameter :: nb_values=10
7 integer :: rank,code,bytes_in_integer
8 TYPE(MPI_File) :: fh
9 integer(kind=MPI_OFFSET_KIND) :: offset
10 integer, dimension(nb_values) :: values
11 TYPE(MPI_Status) :: iostatus
12
13 call MPI_INIT()
14 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
15 call MPI_FILE_SET_ERRHANDLER(MPI_FILE_NULL,MPI_ERRORS_ARE_FATAL);
16 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
17 fh,code)
18
19 call MPI_TYPE_SIZE(MPI_INTEGER,bytes_in_integer)
20
21 offset=rank*nb_values*bytes_in_integer
22 call MPI_FILE_READ_AT(fh,offset,values,nb_values,MPI_INTEGER, &
23 iostatus)
24 print *, "process",rank,":",values(:)
25
26 call MPI_FILE_CLOSE(fh)
27 call MPI_FINALIZE()
28
29 end program read_at
```

# MPI-IO

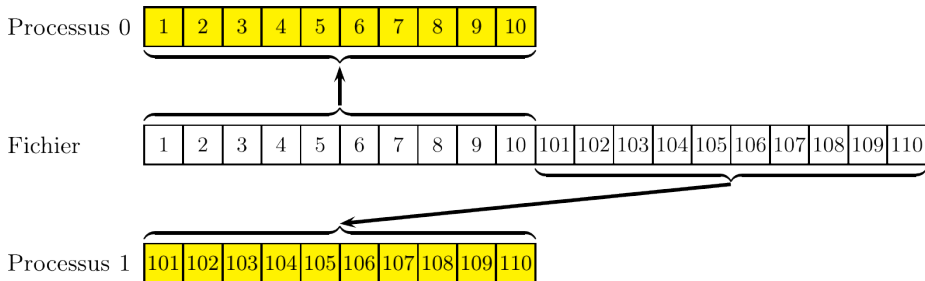


Figure 47 – MPI\_File\_read\_at()

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

## Individual file pointers

```
MPI_File_read(descriptor, buf, count, datatype, statut, code)
MPI_File_write(descriptor, buf, count, datatype, statut, code)

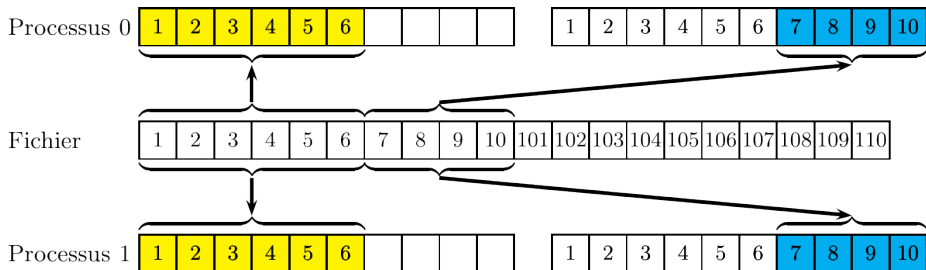
TYPE(MPI_File), INTENT(IN)           :: descriptor
TYPE(*), DIMENSION(..)              :: buf
INTEGER, INTENT(IN)                 :: count
TYPE(MPI_Datatype), INTENT(IN)      :: datatype
TYPE(MPI_Status)                    :: statut
INTEGER, OPTIONAL, INTENT(OUT)      :: code
```

- Write/read in the file `fh`, `count` element of type `datatype` from address `buf` ;
- MPI maintains `one` individual file pointer per `process` per `file handle`.
- After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next data item.
- The shared file pointer is neither used nor updated.

```
1 program read01
2
3 use mpi_f08
4 implicit none
5
6 integer, parameter :: nb_values=10
7 integer :: rank,code
8 TYPE(MPI_File) :: fh
9 integer, dimension(nb_values) :: values
10 TYPE(MPI_Status) :: iostatus
11
12 call MPI_INIT()
13 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
14
15 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
16 fh,code)
17
18 call MPI_FILE_READ(fh,values,6,MPI_INTEGER,iostatus)
19 call MPI_FILE_READ(fh,values(7),4,MPI_INTEGER,iostatus)
20
21 print *, "process",rank,":",values(:)
22
23 call MPI_FILE_CLOSE(fh)
24 call MPI_FINALIZE()
25
26 end program read01
```



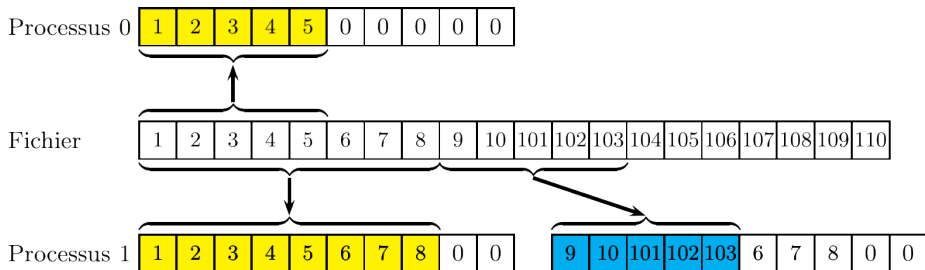
# MPI-IO



**Figure 48** – Example 1 of `MPI_File_read()`

```
> mpiexec -n 2 read01
process 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
1 program read02
2 use mpi
3 implicit none
4
5 integer, parameter          :: nb_values=10
6 integer                    :: rank,code
7 TYPE(MPI_File)             :: fh
8 integer, dimension(nb_values) :: values=0
9 TYPE(MPI_Status)           :: iostatus
10
11 call MPI_INIT()
12 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
13
14 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                   fh,code)
16
17 if (rank == 0) then
18   call MPI_FILE_READ(fh,values,5,MPI_INTEGER,iostatus)
19 else
20   call MPI_FILE_READ(fh,values,8,MPI_INTEGER,iostatus)
21   call MPI_FILE_READ(fh,values,5,MPI_INTEGER,iostatus)
22 end if
23
24 print *, "process",rank,":",values(1:8)
25
26 call MPI_FILE_CLOSE(fh)
27 call MPI_FINALIZE()
28 end program read02
```



**Figure 49** – Example 2 of `MPI_File_read()`

```
> mpiexec -n 2 read02
```

```
process 0 : 1, 2, 3, 4, 5, 0, 0, 0
process 1 : 9, 10, 101, 102, 103, 6, 7, 8
```

## Shared file pointers

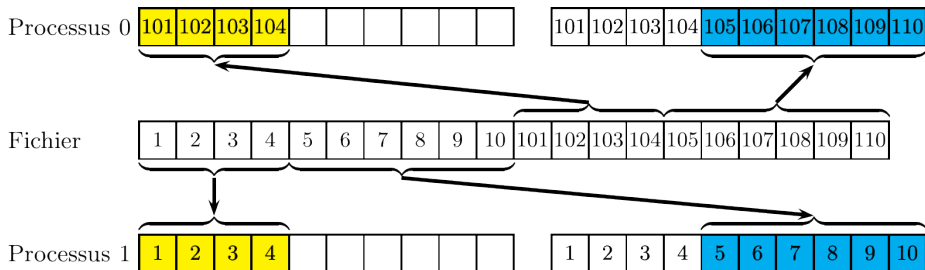
```
MPI_File_read_shared(fh, buf, count, datatype, iostatus, code)
MPI_File_write_shared(fh, buf, count, datatype, iostatus, code)

TYPE(MPI_File), INTENT(IN)           :: descripteur
TYPE(*), DIMENSION(..)              :: buf
INTEGER, INTENT(IN)                  :: count
TYPE(MPI_Datatype), INTENT(IN)      :: datatype
TYPE(MPI_Status)                     :: statut
INTEGER, OPTIONAL, INTENT(OUT)       :: code
```

- Write/read in the file `fh`, `count` element of type `datatype` from address `buf` ;
- MPI maintains only one shared file pointer per file (shared among processes in the communicator group).
- All processes must use the [same file view](#).
- For the noncollective shared file pointer routines, the order [is not deterministic](#). To enforce a specific order, the user needs to use other synchronisation means or use collective variants.
- After a shared file pointer operation, the shared file pointer is updated to point to the next data item, that is, just after the last one accessed by the operation.
- The individual file pointers are neither used nor updated.

```
1 program read_shared01
2
3 use mpi_f08
4 implicit none
5
6 integer                               :: rank,code
7 TYPE(MPI_File)                        :: fh
8 integer, parameter                    :: nb_values=10
9 integer, dimension(nb_values)         :: values
10 TYPE(MPI_Status)                      :: iostatus
11
12 call MPI_INIT()
13 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
14
15 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
16                   fh,code)
17
18 call MPI_FILE_READ_SHARED(fh,values,4,MPI_INTEGER,iostatus)
19 call MPI_FILE_READ_SHARED(fh,values(5),6,MPI_INTEGER,iostatus)
20
21 print *, "process",rank,":",values(:)
22
23 call MPI_FILE_CLOSE(fh)
24 call MPI_FINALIZE()
25
26 end program read_shared01
```

# MPI-IO



**Figure 50** – Example of `MPI_File_read_shared()`

```
> mpiexec -n 2 read_shared01
```

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

## Collective data access

- Collective operations require the participation of all the processes within the **communicator** group associated with the file handle.
- Collective operations may **perform much better** than their noncollective counterparts, as global data accesses have significant potential for automatic optimisation.
- For the collective shared file pointer routines, the accesses to the file will be **in the order** determined by the ranks of the processes within the group. The ordering is therefore **deterministic**.

## Interfaces

```
MPI_File_read_at_all(fh, offset, buf, count, datatype, iostatus, code)
MPI_File_write_at_all(fh, offset, buf, count, datatype, iostatus, code)

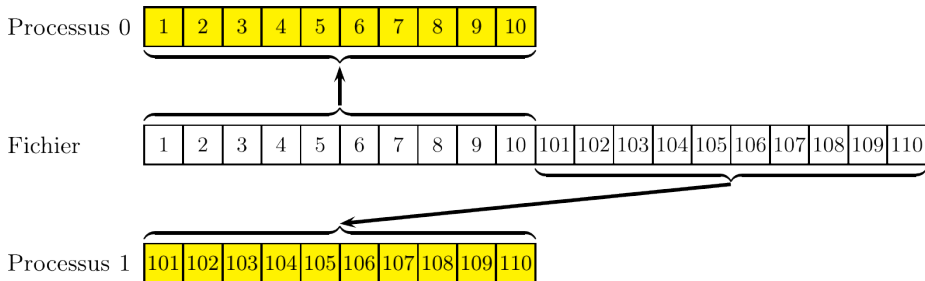
MPI_File_read_all(fh, buf, count, datatype, iostatus, code)
MPI_File_write_all(fh, buf, count, datatype, iostatus, code)

MPI_File_read_ordered(fh, buf, count, datatype, iostatus, code)
MPI_File_write_ordered(fh, buf, count, datatype, iostatus, code)

TYPE(MPI_File), INTENT(IN)           :: fh
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
TYPE(*), DIMENSION(..)             :: buf
INTEGER, INTENT(IN)                 :: count
TYPE(MPI_Datatype), INTENT(IN)     :: datatype
TYPE(MPI_Status)                     :: iostatus
INTEGER, OPTIONAL, INTENT(OUT)    :: code
```



```
1 program read_at_all
2   use mpi_f08
3   implicit none
4
5   integer, parameter           :: nb_values=10
6   integer                     :: rank,code,bytes_in_integer
7   TYPE(MPI_File)              :: fh
8   integer(kind=MPI_OFFSET_KIND) :: offset_file
9   integer, dimension(nb_values) :: values
10  TYPE(MPI_Status)            :: iostatus
11
12  call MPI_INIT()
13  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
14
15  call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
16                    fh)
17
18  call MPI_TYPE_SIZE(MPI_INTEGER,bytes_in_integer)
19  offset_file=rank*nb_values*bytes_in_integer
20  call MPI_FILE_READ_AT_ALL(fh,offset_file,values,nb_values, &
21                            MPI_INTEGER,iostatus)
22  print *, "process",rank,":",values(:)
23
24  call MPI_FILE_CLOSE(fh)
25  call MPI_FINALIZE()
26  end program read_at_all
```



**Figure 51** – Example of `MPI_File_read_at_all()`

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

```
1 program read_all01
2   use mpi_f08
3   implicit none
4
5   integer                               :: rank,code
6   TYPE(MPI_File)                        :: fh
7   integer, parameter                    :: nb_values=10
8   integer, dimension(nb_values)        :: values
9   TYPE(MPI_Status)                      :: iostatus
10
11 call MPI_INIT()
12 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
13
14 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                   fh)
16
17 call MPI_FILE_READ_ALL(fh,values,4,MPI_INTEGER,iostatus)
18 call MPI_FILE_READ_ALL(fh,values(5),6,MPI_INTEGER,iostatus)
19
20 print *, "process ",rank, ":",values(:)
21
22 call MPI_FILE_CLOSE(fh)
23 call MPI_FINALIZE()
24 end program read_all01
```

# MPI-IO

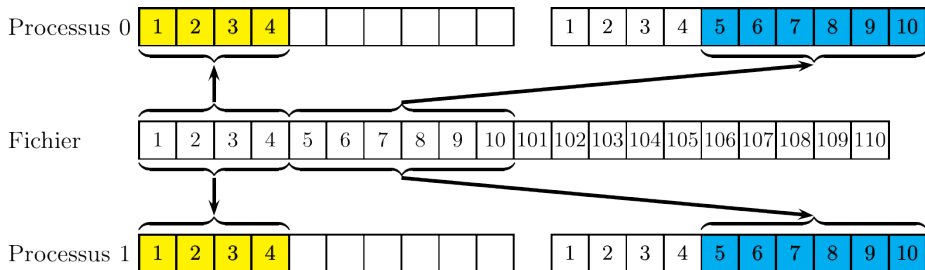


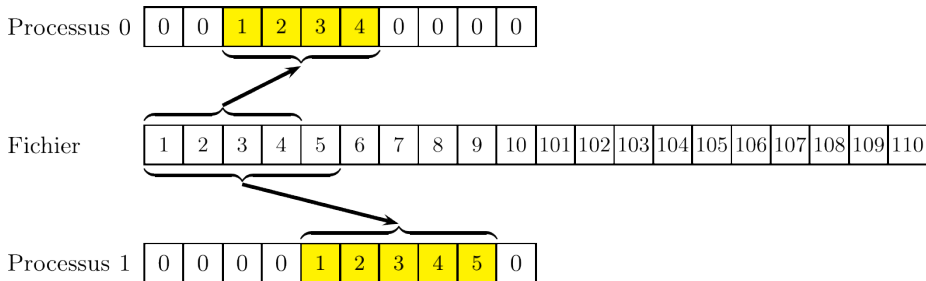
Figure 52 – Example 1 of MPI\_File\_read\_all()

```
> mpiexec -n 2 read_all01
```

```
process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10  
process 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
1 program read_all02
2   use mpi_f08
3   implicit none
4
5   integer, parameter           :: nb_values=10
6   integer                     :: rank, index1, index2, code
7   TYPE(MPI_File)              :: fh
8   integer, dimension(nb_values) :: values=0
9   TYPE(MPI_Status)            :: iostatus
10
11  call MPI_INIT()
12  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
13  call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                    fh)
15
16  if (rank == 0) then
17    index1=3
18    index2=6
19  else
20    index1=5
21    index2=9
22  end if
23
24  call MPI_FILE_READ_ALL(fh,values(index1),index2-index1+1, &
25                        MPI_INTEGER,iostatus)
26  print *, "process",rank,":",values(:)
27
28  call MPI_FILE_CLOSE(fh)
29  call MPI_FINALIZE()
30 end program read_all02
```

# MPI-IO



**Figure 53** – Example 2 of `MPI_File_read_all()`

```
> mpiexec -n 2 read_all02  
process 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0  
process 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```

```
1 program read_all03
2   use mpi_f08
3   implicit none
4
5   integer, parameter                :: nb_values=10
6   integer                           :: rank,code
7   TYPE(MPI_File)                    :: fh
8   integer, dimension(nb_values)     :: values=0
9   TYPE(MPI_Status)                  :: iostatus
10
11  call MPI_INIT()
12  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
13
14  call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                    fh)
16
17  if (rank == 0) then
18    call MPI_FILE_READ_ALL(fh,values(3),4,MPI_INTEGER,iostatus)
19  else
20    call MPI_FILE_READ_ALL(fh,values(5),5,MPI_INTEGER,iostatus)
21  end if
22
23  print *, "process",rank,":",values(:)
24
25  call MPI_FILE_CLOSE(fh)
26  call MPI_FINALIZE()
27 end program read_all03
```

# MPI-IO

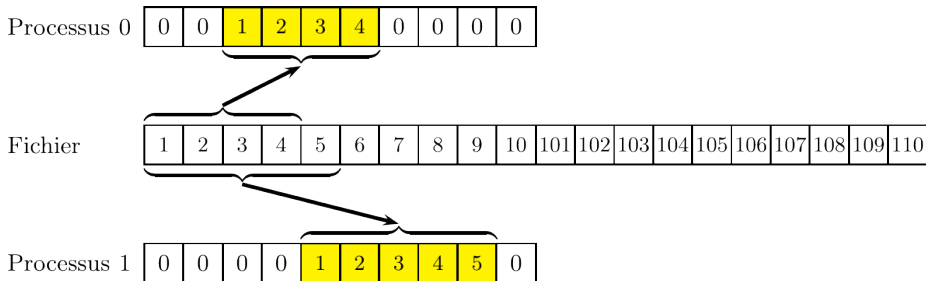


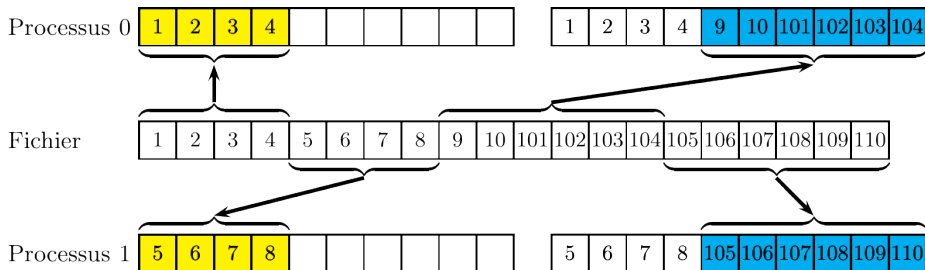
Figure 54 – Example 3 of `MPI_File_read_all()`

```
> mpiexec -n 2 read_all103  
process 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0  
process 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```



```
1 program read_ordered
2   use mpi_f08
3   implicit none
4
5   integer                               :: rank,code
6   TYPE(MPI_File)                        :: fh
7   integer, parameter                    :: nb_values=10
8   integer, dimension(nb_values)         :: values
9   TYPE(MPI_Status)                      :: iostatus
10
11 call MPI_INIT()
12 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
13
14 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                   fh)
16
17 call MPI_FILE_READ_ORDERED(fh,values,4,MPI_INTEGER,iostatus)
18 call MPI_FILE_READ_ORDERED(fh,values(5),6,MPI_INTEGER,iostatus)
19
20 print *, "process",rank,":",values(:)
21
22 call MPI_FILE_CLOSE(fh)
23 call MPI_FINALIZE()
24 end program read_ordered
```

# MPI-IO



**Figure 55** – Example of MPI\_File\_ordered()

```
> mpiexec -n 2 read_ordered
```

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

## Positioning the file pointers

```
MPI_File_seek(fh, offset, whence, code)
MPI_File_seek_shared(fh, offset, whence, code)

TYPE(MPI_File), INTENT(IN)           :: fh
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
INTEGER, INTENT(IN)                 :: whence
INTEGER, OPTIONAL, INTENT(OUT)      :: code
```

- `MPI_File_seek()` and `MPI_File_seek_shared()` updates the file pointer values by using the following possible modes :
  - `MPI_SEEK_SET` : The pointer is set to offset.
  - `MPI_SEEK_CUR` : The pointer is set to the current pointer position plus offset.
  - `MPI_SEEK_END` : The pointer is set to the end of file plus offset.
- With `MPI_SEEK_CUR` and `MPI_SEEK_END`, the `offset` can be negative, which allows seeking backwards.

```
1 program seek
2 use mpi_f08
3 implicit none
4 integer, parameter :: nb_values=10
5 integer :: rank, bytes_in_integer, code
6 TYPE(MPI_File) :: fh
7 integer(kind=MPI_OFFSET_KIND) :: offset
8 integer, dimension(nb_values) :: values
9 TYPE(MPI_Status) :: iostatus
10
11 call MPI_INIT()
12 call MPI_COMM_RANK(MPI_COMM_WORLD, rank)
13 call MPI_FILE_OPEN(MPI_COMM_WORLD, "data.dat", MPI_MODE_RDONLY, MPI_INFO_NULL, &
14 fh)
15
16 call MPI_FILE_READ(fh, values, 3, MPI_INTEGER, iostatus)
17 call MPI_TYPE_SIZE(MPI_INTEGER, bytes_in_integer)
18 offset=8*bytes_in_integer
19 call MPI_FILE_SEEK(fh, offset, MPI_SEEK_CUR)
20 call MPI_FILE_READ(fh, values(4), 3, MPI_INTEGER, iostatus)
21 offset=4*bytes_in_integer
22 call MPI_FILE_SEEK(fh, offset, MPI_SEEK_SET)
23 call MPI_FILE_READ(fh, values(7), 4, MPI_INTEGER, iostatus)
24
25 print *, "process", rank, ":", values(:)
26
27 call MPI_FILE_CLOSE(fh)
28 call MPI_FINALIZE()
29 end program seek
```

# MPI-IO

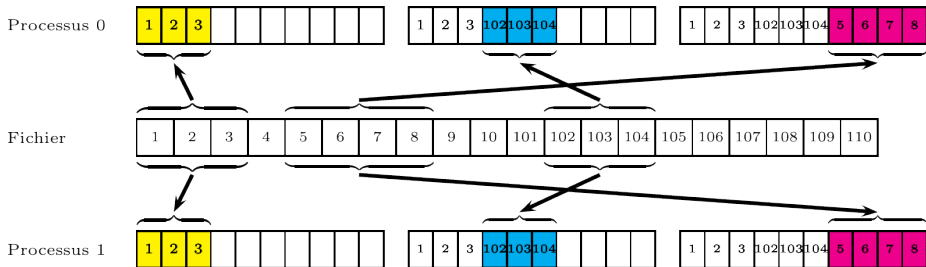


Figure 56 – Example of MPI\_File\_seek ()

```
> mpiexec -n 2 seek  
process 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8  
process 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
```

## Access to end of file

- Writing to the end of the file increases the file size.
- Reading at the end of the file does not retrieve any data. When reading, using `MPI_Get_count()` allows you to know the number of elements actually read.

## Nonblocking Data Access

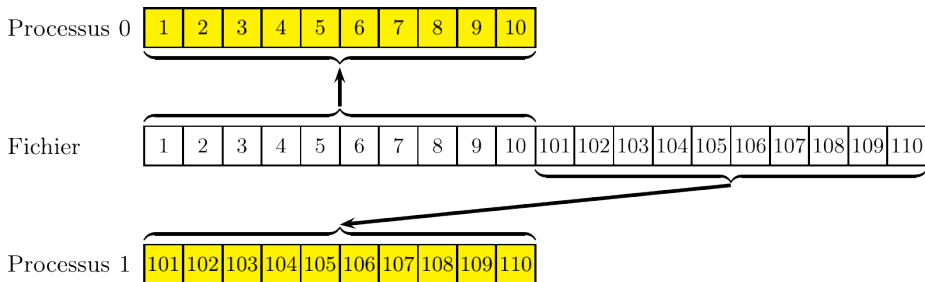
- Nonblocking operations enable overlapping of I/O operations and computations.
- The semantic of nonblocking I/O calls is similar to the semantic of nonblocking communications between processes.
- A first nonblocking I/O call initiates the I/O operation and a separate request call is needed to complete the I/O requests (`MPI_Test()`, `MPI_Wait()`, etc.).

```
1 program iread_at
2   use mpi_f08
3   implicit none
4
5   integer, parameter           :: nb_values=10
6   integer                     :: i,nb_iterations=0,rank,bytes_in_integer,code
7   TYPE(MPI_Request)           :: request
8   TYPE(MPI_File)              :: fh
9   integer(kind=MPI_OFFSET_KIND) :: offset
10  integer, dimension(nb_values) :: values
11  TYPE(MPI_Status)             :: iostatus
12  logical                      :: finished
13
14  call MPI_INIT()
15  call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
```



```
16  call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
17                      fh)
18
19  call MPI_TYPE_SIZE(MPI_INTEGER,bytes_in_integer)
20
21  offset=rank*nb_values*bytes_in_integer
22  call MPI_FILE_IREAD_AT(fh,offset,values,nb_values, &
23                        MPI_INTEGER,request)
24
25  do while (nb_iterations < 5000)
26      nb_iterations=nb_iterations+1
27      ! Overlapping the I/O operation with computations
28      ...
29      call MPI_TEST(request,finished,iostatus)
30      if (finished) exit
31  end do
32  if (.not. finished) call MPI_WAIT(request,iostatus)
33  print *, "After",nb_iterations,"iterations, process",rank,":",values
34
35  call MPI_FILE_CLOSE(fh)
36  call MPI_FINALIZE()
37
38  end program  iread_at
```

## MPI-IO



**Figure 57** – Example of `MPI_File_iread_at()`

```
> mpiexec -n 2 iread_at
```

```
After 1 iterations, process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
After 1 iterations, process 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

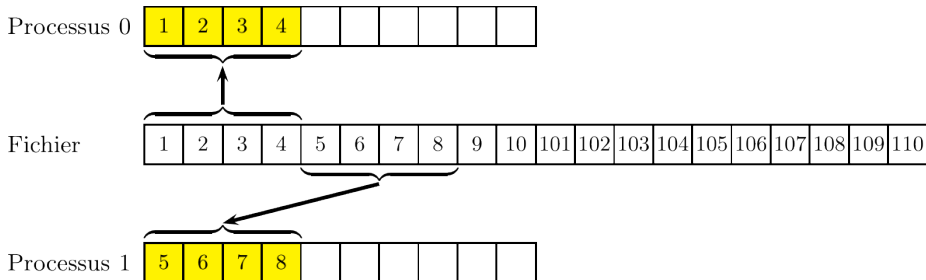
```
1 program iwrite
2   use mpi_f08
3   implicit none
4
5   integer, parameter           :: nb_values=10
6   TYPE(MPI_File)              :: fh
7   TYPE(MPI_Request)           :: request
8   integer                     :: code, nb_it=0
9   integer(kind=MPI_OFFSET_KIND) :: offset
10  integer, dimension(nb_values) :: values, temps
11  logical                      :: finished
12
13  call MPI_INIT()
14  !...
15  call MPI_FILE_OPEN(MPI_COMM_WORLD, "data.dat", MPI_MODE_WRONLY+MPI_MODE_CREATE, &
16                    MPI_INFO_NULL, fh)
17
18  temp = values
19  call MPI_FILE_SEEK(fh, offset, MPI_SEEK_SET)
20  call MPI_FILE_IWRITE(fh, temp, nb_values, MPI_INTEGER, request)
21  do while (nb_it < 5000)
22    nb_it = nb_it+1
23    ...
24    call MPI_TEST(request, finished, MPI_STATUS_IGNORE)
25    if (finished) then
26      temp = values
27      call MPI_FILE_SEEK(fh, offset, MPI_SEEK_SET)
28      call MPI_FILE_IWRITE(fh, temp, nb_values, MPI_INTEGER, request)
29    end if
30  end do
31  call MPI_WAIT(request, MPI_STATUS_IGNORE)
32  call MPI_FILE_CLOSE(fh)
33  call MPI_FINALIZE()
34 end program iwrite
```

## Nonblocking and collective data access routines

- It is possible to perform operations that are both **collective** and **nonblocking**.
- Only one collective nonblocking operation can be in progress on any MPI process at a time.
- Between the two parts of the collective nonblocking operation, individual operations on the file are allowed, but the memory region involved in the collective operation cannot be modified.

```
1 program iread_all
2
3 use mpi_f08
4 implicit none
5
6 integer :: rank,code
7 TYPE(MPI_File) :: fh
8 integer, parameter :: nb_values=10
9 integer, dimension(nb_values) :: values
10 TYPE(MPI_Status) :: iostatus
11 TYPE(MPI_Request) :: req
12
13 call MPI_INIT()
14 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
15
16 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
17 fh)
18
19 call MPI_FILE_IREAD_ALL(fh,values,4,MPI_INTEGER,req)
20 print *, "Process : ",rank
21 call MPI_WAIT(req,iostatus)
22
23 print *, "process",rank,":",values(1:4)
24
25 call MPI_FILE_CLOSE(fh)
26 call MPI_FINALIZE()
27
28 end program iread_all
```

# MPI-IO



**Figure 58** – Example of `MPI_File_iread_all()`

```
> mpiexec -n 2 iread_all
Process      : 0
process 0   : 1, 2, 3, 4
Process      : 1
process 1   : 5, 6, 7, 8
```

## MPI Hands-On – Exercise 7 : Read an MPI-IO file

- We have a binary file `data.dat` with 484 integer values.
- With 4 processes, it consists of reading the 121 first values on process 0, the 121 next on the process 1, and so on.
- We will use 4 different methods :
  - Read via explicit offsets, in individual mode
  - Read via shared file pointers, in collective mode
  - Read via individual file pointers, in individual mode
  - Read via shared file pointers, in individual mode
- To compile use `make`, to execute use `make exe`, and to verify the results use `make verification` which build figure file corresponding to the four cases.

## MPI 4.x



## Interopérabilité

For example, for `MPI_RECV()` the interface with the `mpi` module is :

```
<type> buf(*)  
INTEGER :: count, datatype, source, tag, comm, ierror  
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: statut
```

With the `mpi_f08` module :

```
TYPE(*), DIMENSION(..) :: buf  
INTEGER :: count, source, tag  
TYPE(MPI_DATATYPE) :: datatype  
TYPE(MPI_COMM) :: comm  
TYPE(MPI_STATUS) :: statut  
INTEGER, optional :: ierror
```

These new types are in fact INTEGER

```
TYPE, BIND(C) :: MPI_COMM  
  INTEGER :: MPI_VAL  
END TYPE MPI_COMM
```

With the `mpi_f08` module, expression `comm%mpi_val` is equivalent to argument `comm` in `mpi` module.

## Adding

- Large count
- Partitioned communication
- MPI Session
- Others

# Large count

- Count parameters were in `integer` or `int`.
- MPI 4.0 add new functions with `MPI_Count` instead.
- In *C* these new functions have `_c` at the end.

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

- In *Fortran* count in `integer` can be changed in `integer(kind=MPI_COUNT_KIND)`
- Only available with the `mpi_f08` module
- No change in the name of function with polymorphism

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: dest, tag
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

## Partitioned communication

- Multiple contribution to a communication.
- Usefull in hybrid.
- Init with `MPI_Psend_init()` or `MPI_Precv_init()` by providing the count by partition and the number of partition.
- `MPI_Start()` to start the communication.
- `MPI_Pready()` to indicate that a partition is ready.
- Could not mix `MPI_Recv()` and `MPI_Psend_init()`.
- `MPI_Wait()` to wait for the end of communication.
- `MPI_Parrived()` to know if a partition has been received.

# Session

- A way to do multiple `MPI_Init()`/`MPI_Finalize()`.
- `MPI_Session_init()` to start a session.
- `MPI_Session_finalize()` to end a session.
- No more `MPI_COMM_WORLD`.
- *Process Sets* : `mpi://WORLD` and `mpi://SELF`.
- `MPI_Group_from_session_pset()` to make a group from a *pset*.
- `MPI_Comm_create_from_group()` to make a communicator from a group.
- `MPI_Session_get_num_psets()` to known the number of *pset* available.
- `MPI_Session_get_nth_pset()` to get the name of a *pset*.

## Others

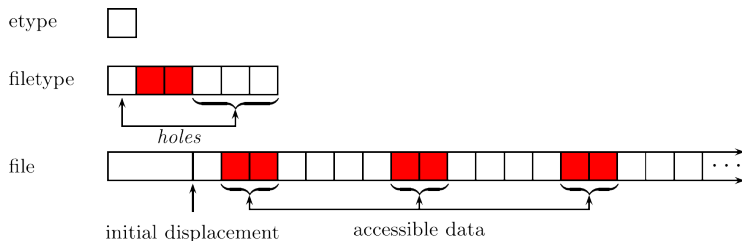
- Add of `MPI_Isendrecv` and `MPI_Isendrecv_replace`.
- Add persistent collective communication.
- Add option `mpi_initial_errhandler` for *mpiexec* to specify the default errhandler.

## MPI-IO Views

# MPI-IO Views

## The View Mechanism

- **File Views** is a mechanism which accesses data in a high-level way. A **view** describes a template for accessing a file.
- The view that a given process has of an open file is defined by three components : the **elementary data type**, **file type** and an initial **displacement**.
- The view is determined by the repetition of the filetype pattern, beginning at the displacement.



**Figure 59** – Tiling a file with a filetype



# MPI-IO Views

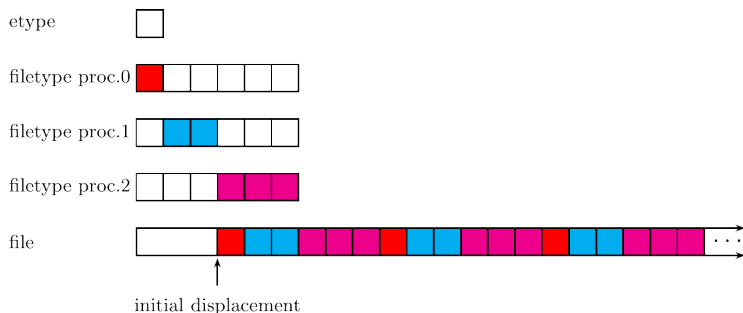
## The View Mechanism

- **File Views** are defined using **MPI datatypes**.
- **Derived datatypes** can be used to structure accesses to the file. For example, elements can be skipped during data access.
- The default view is a linear byte stream (displacement is zero, etype and filetype equal to `MPI_BYTE`).

## Multiple Views

- Each process can successively use **several views** on the same file.
- Each process can define its own view of the file and access complementary parts of it.

## MPI-IO Views



**Figure 60** – Separate views, each using a different filetype, can be used to access the file

### Limitations :

- Shared file pointer routines are not useable except when all the processes have the same file view.
- If the file is opened for writing, the different views may not overlap, even partially.

## Changing the process's view of the data in the file : `MPI_File_set_view()`

```
MPI_FILE_SET_VIEW(fh, displacement, etype, filetype,  
                 mode, info, code)  
  
TYPE(MPI_File), intent(in)           :: fh  
integer(kind=MPI_OFFSET_KIND), intent(in) :: displacement  
TYPE(MPI_Datatype), intent(in)       :: etype, filetype  
character(len=*), intent(in)         :: mode  
TYPE(MPI_Info), intent(in)           :: info  
integer, optional, intent(out)       :: code
```

- This operation is **collective** throughout the file handle. The values for the **initial displacement** and the **filetype** may vary between the processes in the group. The extents of **elementary types** must be identical.
- In addition, the individual file pointers and the shared file pointer are **reset to zero**, taking the initial displacement into account.

### Notes :

- The datatypes passed in must have been committed using the `MPI_Type_commit()` subroutine.
- MPI defines three data representations (**mode**) : "native", "internal" or "external32".

### Subarray datatype constructor

A derived data type useful to create a filetype is the “subarray” type, that we introduce here. This type allows creating a subarray from an array and can be defined with the `MPI_Type_create_subarray()` subroutine.

The `shape` of an array is a vector for which each element equals the number of array elements in each dimension. For example, the array `T(10, 0:5, -10:10)` (or `T[10][6][21]`), its shape is the `(10,6,21)` vector.

# MPI-IO Views / Derived Datatypes

```
MPI_TYPE_CREATE_SUBARRAY(nb_dims, shape_array, shape_sub_array, coord_start,  
                          order, old_type, new_type, code)  
  
integer, intent(in)           :: nb_dims  
integer, dimension(nb_dims), intent(in) :: shape_array, shape_sub_array, coord_start  
integer, intent(in)           :: order  
TYPE(MPI_Datatype), intent(in) :: old_type  
TYPE(MPI_Datatype), intent(out) :: new_type  
integer, optional, intent(out) :: code
```

## Explanation of the arguments

- `nb_dims` : number of dimension 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 the start coordinates of the subarray to be `array(2, 3)`, we must have `coord_start(:)=(/ 1, 2 /)`
- `order` : storage order of elements
  - `MPI_ORDER_FORTRAN` for the ordering used by Fortran arrays (column-major order)
  - `MPI_ORDER_C` for the ordering used by C arrays (row-major order)

# MPI-IO Views / Derived Datatypes

## Exchanges between 2 process with subarray

BEFORE

1	5	9
2	6	10
3	7	11
4	8	12

Processus 0

-1	-5	-9
-2	-6	-10
-3	-7	-11
-4	-8	-12

Processus 1

AFTER

1	5	9
-7	-11	10
-8	-12	11
4	8	12

Processus 0

-1	-5	-9
-2	-6	-10
-3	2	6
-4	3	7

Processus 1

## Exchanges between the two processes (Part 1/2)

```
1 program subarray
2
3 use mpi_f08
4 implicit none
5
6 integer,parameter :: nb_lines=4,nb_columns=3,&
7                       tag=1000,nb_dims=2
8 integer           :: code,rank,i
9 TYPE(MPI_Datatype) :: type_subarray
10 integer,dimension(nb_lines,nb_columns) :: tab
11 integer,dimension(nb_dims)           :: shape_array,shape_subarray,coord_start
12 TYPE(MPI_Status)   :: msgstatus
13
14 call MPI_INIT()
15 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
16
17 !Initialization of the tab array on each process
18 tab(:,:) = reshape( (/ (sign(i,-rank),i=1,nb_lines*nb_columns) /) , &
19                   (/ nb_lines,nb_columns /) )
```

# MPI-IO Views / Derived Datatypes

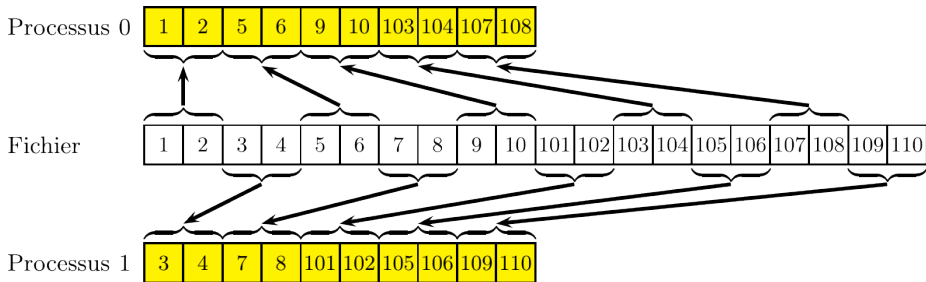
## Exchanges between the two processes (Part 2/2)

```
20  !Shape of the tab array from which a subarray will be extracted
21  shape_tab(:) = shape(tab)
22  !The F95 shape function gives the shape of the array put in argument.
23  !ATTENTION, if the concerned array was not allocated on all the processes,
24  !it is necessary to explicitly put the shape of the array in order for it
25  !to be known on all the processes, shape_array(:) = (/ nb_lines,nb_columns /)
26
27  !Shape of the subarray
28  shape_subarray(:) = (/ 2,2 /)
29
30  !Start coordinates of the subarray
31  !For the process 0 we start from the tab(2,1) element
32  !For the process 1 we start from the tab(3,2) element
33  coord_start(:) = (/ rank+1,rank /)
34
35  !Creation of the type_subarray derived datatype
36  call MPI_TYPE_CREATE_SUBARRAY(nb_dims,shape_array,shape_subarray,coord_start,&
37                               MPI_ORDER_FORTRAN,MPI_INTEGER,type_subarray)
38  call MPI_TYPE_COMMIT(type_subarray)
39
40  !Exchange of the subarrays
41  call MPI_SENDRECV_REPLACE(tab,1,type_subarray,mod(rank+1,2),tag,&
42                           mod(rank+1,2),tag,MPI_COMM_WORLD,msgstatus)
43  call MPI_TYPE_FREE(type_subarray)
44  call MPI_FINALIZE()
45  end program subarray
```



# MPI-IO Views

## Example 1 : Reading non-overlapping sequences of data segments in parallel

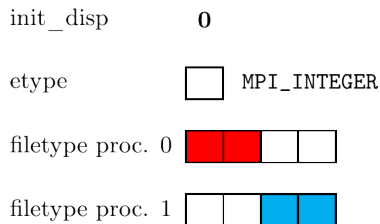


**Figure 61** – Example 1 : Reading non-overlapping sequences of data segments in parallel

```
> mpiexec -n 2 read_view01
process 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
process 0 : 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
```

# MPI-IO Views

## Example 1



**Figure 62** – Example 1 (continued)

```
1  if (rank == 0) coord=1
2  if (rank == 1) coord=3
3
4  call MPI_TYPE_CREATE_SUBARRAY(1, (/4/), (/2/), (/coord - 1/), &
5                               MPI_ORDER_FORTRAN, MPI_INTEGER, filetype)
6  call MPI_TYPE_COMMIT(filetype)
7
8  init_displacement=0
9
10 call MPI_FILE_SET_VIEW(handle, init_displacement, MPI_INTEGER, filetype, &
11                        "native", MPI_INFO_NULL)
```

# MPI-IO Views

## Example 1 : code

```
1 program read_view01
2 use mpi_f08
3 implicit none
4 integer, parameter :: nb_values=10
5 integer :: rank, coord, code
6 TYPE(MPI_Datatype) :: filetype
7 TYPE(MPI_File) :: handle
8 integer(kind=MPI_OFFSET_KIND) :: init_displacement
9 integer, dimension(nb_values) :: values
10 TYPE(MPI_Status) :: iostatus
11 call MPI_INIT(code)
12 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
13
14 if (rank == 0) coord=1
15 if (rank == 1) coord=3
16
17 call MPI_TYPE_CREATE_SUBARRAY(1, (/4/), (/2/), (/coord - 1/), &
18 MPI_ORDER_FORTRAN, MPI_INTEGER, filetype)
19 call MPI_TYPE_COMMIT(filetype)
20
21 call MPI_FILE_OPEN(MPI_COMM_WORLD, "data.dat", MPI_MODE_RDONLY, MPI_INFO_NULL, &
22 handle)
23
24 init_displacement=0
25 call MPI_FILE_SET_VIEW(handle, init_displacement, MPI_INTEGER, filetype, &
26 "native", MPI_INFO_NULL)
27 call MPI_FILE_READ(handle, values, nb_values, MPI_INTEGER, iostatus)
28
29 print *, "process", rank, ":", values(:)
30
31 call MPI_FILE_CLOSE(handle)
32 call MPI_FINALIZE()
33 end program read_view01
```

# MPI-IO Views

## Example 2 : Reading data using successive views (Part 1/2)



Figure 63 – Example 2 : Reading data using successive views

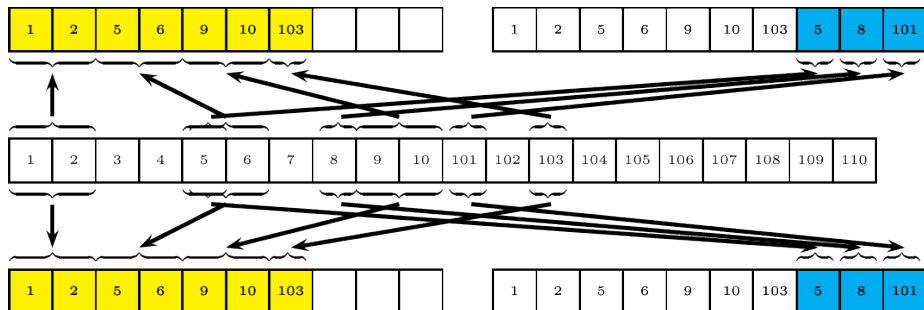
```
1 program read_view02
2
3 use mpi
4 implicit none
5
6 integer, parameter :: nb_values=10
7 integer :: rank,code,bytes_in_integer
8 TYPE(MPI_File) :: handle
9 TYPE(MPI_Datatype) :: filetype_1,filetype_2
10 integer(kind=MPI_OFFSET_KIND) :: init_displacement
11 integer, dimension(nb_values) :: values
12 TYPE(MPI_Status) :: iostatus
13
14 call MPI_INIT()
15 call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
```

## Example 2 (Part 2/2)

```
16 call MPI_TYPE_CREATE_SUBARRAY(1, (/4/), (/2/), (/0/), &
17                               MPI_ORDER_FORTRAN, MPI_INTEGER, filetype_1)
18 call MPI_TYPE_COMMIT(filetype_1)
19
20 call MPI_TYPE_CREATE_SUBARRAY(1, (/3/), (/1/), (/2/), &
21                               MPI_ORDER_FORTRAN, MPI_INTEGER, filetype_2)
22 call MPI_TYPE_COMMIT(filetype_2)
23
24 call MPI_FILE_OPEN(MPI_COMM_WORLD, "data.dat", MPI_MODE_RDONLY, MPI_INFO_NULL, &
25                   handle)
26
27 ! Read using the first view
28 init_displacement=0
29 call MPI_FILE_SET_VIEW(handle, init_displacement, MPI_INTEGER, filetype_1, &
30                       "native", MPI_INFO_NULL)
31 call MPI_FILE_READ(handle, values, 4, MPI_INTEGER, iostatus)
32 call MPI_FILE_READ(handle, values(5), 3, MPI_INTEGER, iostatus)
33
34 ! Read using the second view
35 call MPI_TYPE_SIZE(MPI_INTEGER, nb_octets_entier)
36 init_displacement=2*nb_octets_entier
37 call MPI_FILE_SET_VIEW(handle, init_displacement, MPI_INTEGER, filetype_2, &
38                       "native", MPI_INFO_NULL)
39 call MPI_FILE_READ(handle, values(8), 3, MPI_INTEGER, iostatus)
40
41 print *, "process", rank, ":", values(:)
42
43 call MPI_FILE_CLOSE(handle)
44 call MPI_FINALIZE()
45 end program read_view02
```

# MPI-IO Views

## Example 2 : Illustration



```
> mpiexec -n 2 read_view02
```

```
process 1 : 1, 2, 5, 6, 9, 10, 103, 5, 8, 101  
process 0 : 1, 2, 5, 6, 9, 10, 103, 5, 8, 101
```

# MPI-IO Views

## Example 3 : Dealing with holes in datatypes (Part 1/2)

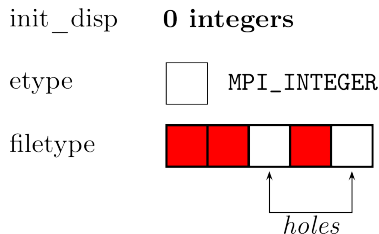


Figure 64 – Example 3 : Dealing with holes in datatypes

```
1 program read_view03_indexed
2 use mpi
3 implicit none
4 integer, parameter :: nb_values=9
5 integer :: rank, bytes_in_integer, code
6 TYPE(MPI_File) :: handle
7 TYPE(MPI_Datatype) :: filetype_tmp, filetype
8 integer(kind=MPI_OFFSET_KIND) :: init_displacement
9 integer(kind=MPI_ADDRESS_KIND) :: lb, extent
10 integer, dimension(2) :: blocklengths, indices
11 integer, dimension(nb_values) :: values
12 TYPE(MPI_Status) :: iostatus
13 call MPI_INIT()
14 call MPI_COMM_RANK(MPI_COMM_WORLD, rank)
```

# MPI-IO Views

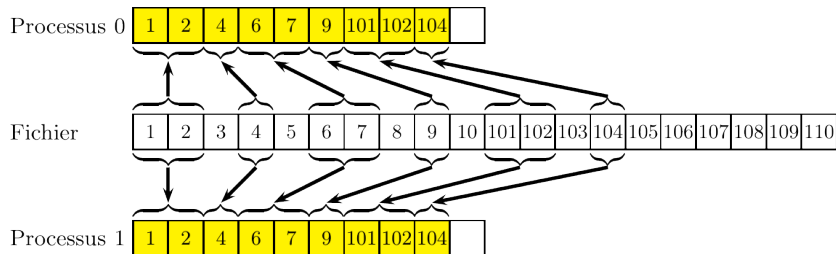
## Example 3 (Part 2/2)

```
15  ! filetype_tmp: MPI type with an extent of 4*MPI_INTEGER
16  indices(1)=0
17  blocklengths(1)=2
18  indices(2)=3
19  blocklengths(2)=1
20  call MPI_TYPE_INDEXED(2,blocklengths,indices,MPI_INTEGER,filetype_tmp)
21
22  ! filetype: MPI type with an extent of 5*MPI_INTEGER
23  call MPI_TYPE_SIZE(MPI_INTEGER,bytes_in_integer)
24  call MPI_TYPE_GET_EXTENT(filetype_tmp,lb,extent)
25  extent = extent + bytes_in_integer
26  call MPI_TYPE_CREATE_RESIZED(filetype_tmp,lb,lb+extent,filetype)
27  call MPI_TYPE_COMMIT(filetype)
28
29  call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
30  handle)
31
32  init_displacement=0
33  call MPI_FILE_SET_VIEW(handle,init_displacement,MPI_INTEGER,filetype, &
34  "native",MPI_INFO_NULL)
35
36  call MPI_FILE_READ(handle,values,9,MPI_INTEGER,iostatus)
37
38  print *, "process",rank,":",values(:)
39
40  call MPI_FILE_CLOSE(handle)
41  call MPI_FINALIZE()
42
43  end program read_view03_indexed
```



# MPI-IO Views

## Example 3 : Illustration



```
> mpiexec -n 2 read_view03
```

```
process 0 : 1, 2, 4, 6, 7, 9, 101, 102, 104
```

```
process 1 : 1, 2, 4, 6, 7, 9, 101, 102, 104
```

## Example 3 : Alternative implementation using a structure datatype

```
1 program read_view03_struct
2   [...]
3   integer(kind=MPI_ADDRESS_KIND), dimension(2) :: displacements
4   [...]
5
6   call MPI_TYPE_CREATE_SUBARRAY(1, (/3/), (/2/), (/0/), MPI_ORDER_FORTRAN, &
7     MPI_INTEGER, tmp_filetype1)
8
9   call MPI_TYPE_CREATE_SUBARRAY(1, (/2/), (/1/), (/0/), MPI_ORDER_FORTRAN, &
10    MPI_INTEGER, tmp_filetype2)
11
12   call MPI_TYPE_SIZE(MPI_INTEGER, bytes_in_integer)
13
14   displacements(1) = 0
15   displacements(2) = 3*bytes_in_integer
16
17   call MPI_TYPE_CREATE_STRUCT(2, (/1,1/), displacements, &
18     (/tmp_filetype1, tmp_filetype2/), filetype)
19   call MPI_TYPE_COMMIT(filetype)
20
21   [...]
22
23 end program read_view03_struct
```

## Conclusion

MPI-IO offers a high-level interface and a very large set of functionalities. It is possible to carry out complex operations and take advantage of optimizations implemented in the library. MPI-IO also offers good portability

## Advice

- The use of explicitly positioned subroutines in files should be reserved for special cases since the implicit use of individual pointers with views provides a higher level interface.
- When the operations involve all the processes (or a subset identifiable by an MPI sub-communicator), it is generally necessary to favor the **collective** form of the operations.
- Exactly as for the processing of messages when these represent an important part of the application, **nonblocking** is a privileged way of optimization to be implemented by programmers, but this should only be implemented **after** ensuring the correctness of behavior of the application in blocking mode.

## Conclusion

## Conclusion

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

## MPI Hands-On – Exercise 8 : Poisson's equation

Resolution of the following Poisson equation :

$$\begin{cases} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) & \text{in } [0, 1] \times [0, 1] \\ u(x, y) = 0 & \text{on the boundaries} \\ f(x, y) = 2 \cdot (x^2 - x + y^2 - y) \end{cases}$$

We will solve this equation with a domain decomposition method :

- The equation is discretized on the domain with a finite difference method.
- The obtained system is resolved with a Jacobi solver.
- The global domain is split into sub-domains.

The exact solution is known and is  $u_{exact}(x, y) = xy(x - 1)(y - 1)$

## MPI Hands-On – Exercise 8 : Poisson's equation

To discretize the equation, we define a grid with a set of points  $(x_i, y_j)$

$$x_i = i h_x \quad \text{for } i = 0, \dots, ntx + 1$$

$$y_j = j h_y \quad \text{for } j = 0, \dots, nty + 1$$

$$h_x = \frac{1}{(ntx + 1)}$$

$$h_y = \frac{1}{(nty + 1)}$$

$h_x$  :  $x$ -wise step

$h_y$  :  $y$ -wise step

$ntx$  : number of  $x$ -wise interior points

$nty$  : number of  $y$ -wise interior points

In total, there are  $ntx+2$  points in the  $x$  direction  
and  $nty+2$  points in the  $y$  direction.

## MPI Hands-On – Exercise 8 : Poisson's equation

- Let  $u_{ij}$  be the estimated solution at position  $x_i = ih_x$  and  $y_j = jh_y$ .
- The Jacobi solver consist of computing :

$$u_{ij}^{n+1} = c_0(c_1(u_{i+1j}^n + u_{i-1j}^n) + c_2(u_{ij+1}^n + u_{ij-1}^n) - f_{ij})$$

$$\text{with : } c_0 = \frac{1}{2} \frac{h_x^2 h_y^2}{h_x^2 + h_y^2}$$

$$c_1 = \frac{1}{h_x^2}$$

$$c_2 = \frac{1}{h_y^2}$$



## MPI Hands-On – Exercise 8 : Poisson's equation

- In parallel, the interface values of subdomains must be exchanged between the neighbours.
- We use ghost cells as receive buffers.

## MPI Hands-On – Exercise 8 : Poisson's equation

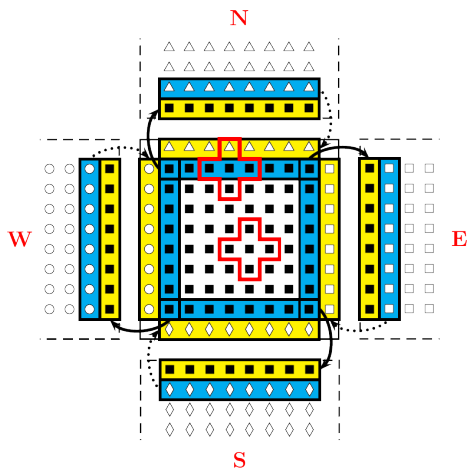


Figure 65 – Exchange points on the interfaces

## MPI Hands-On – Exercise 8 : Poisson's equation

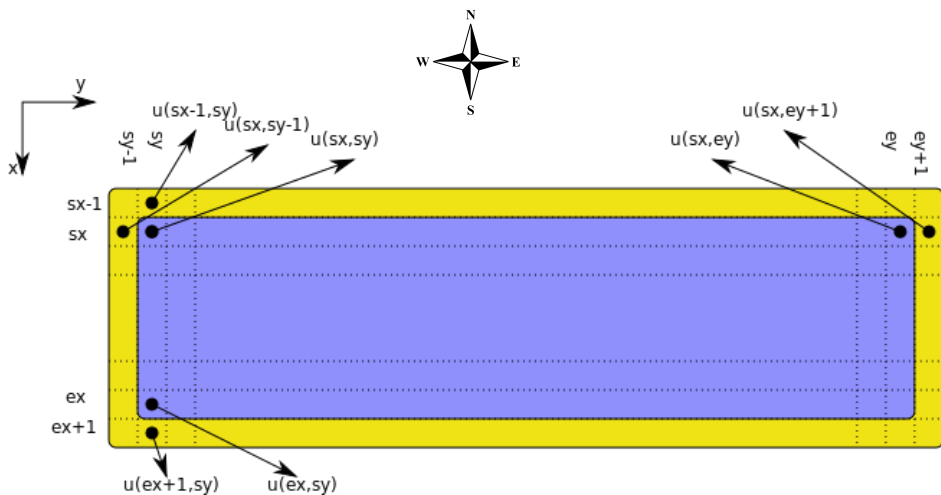
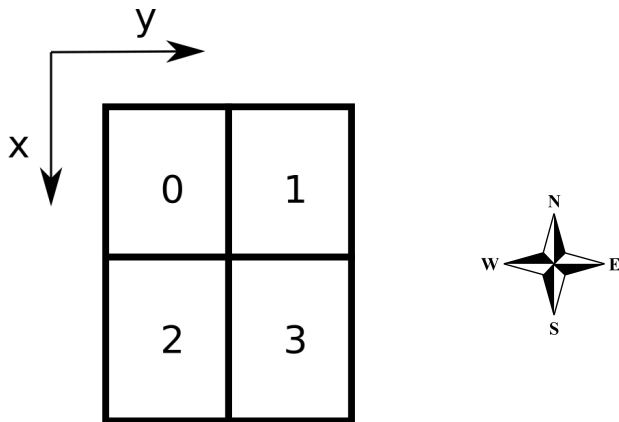


Figure 66 – Numeration of points in different sub-domains

## MPI Hands-On – Exercise 8 : Poisson's equation



**Figure 67** – Process rank numbering in the sub-domains

## MPI Hands-On – Exercise 8 : Poisson's equation

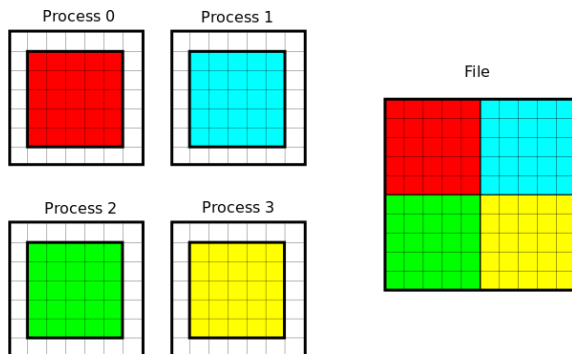


Figure 68 – Writing the global matrix  $u$  in a file

You need to :

- Define a view, to see only the owned part of the global matrix  $u$  ;
- Define a type, in order to write the local part of matrix  $u$  (without ghost points) ;
- Apply the view to the file ;
- Write using only one call.

## MPI Hands-On – Exercise 8 : Poisson's equation

- Initialisation of the MPI environment.
- Creation of the 2D Cartesian topology
- Determination of the array indexes for each sub-domain.
- Determination of the 4 neighbour processes for each sub-domain.
- Creation of two derived datatypes, *type\_line* and *type\_column*.
- Exchange the values on the interfaces with the other sub-domains.
- Computation of the global error. When the global error is lower than a specified value (machine precision for example), we consider that we have reached the exact solution.
- Collecting of the global matrix  $u$  (the same one as we obtained in the sequential) in an MPI-IO file `data.dat`.

## MPI Hands-On – Exercise 8 : Poisson's equation

- A skeleton of the parallel version is proposed : It consists of a main program (`poisson.f90`) and several subroutines. All the modifications have to be done in the `parallel.f90` file.
- To compile use `make`, to execute use `make exe`. To verify the results, use `make verification` which runs a reading program of the `data.dat` file and compares it with the sequential version.