Message Passing Interface (MPI)
4.8 All-to-all: MPI_ALLTOALL() ................................................................. 91
4.9 Global reduction ............................................................................. 98
4.10 Additions ......................................................................................... 120

5 One-sided Communication ................................................................. 121
5.1 Introduction ....................................................................................... 122
  5.1.1 Reminder: The Concept of Message-Passing .............................. 123
  5.1.2 The Concept of One-sided Communication ................................. 124
  5.1.3 RMA Approach of MPI ............................................................. 125
5.2 Memory Window ............................................................................... 126
5.3 Data Transfer .................................................................................... 130
5.4 Completion of the Transfer: the synchronization ......................... 134
  5.4.1 Active Target Synchronization .................................................. 135
  5.4.2 Passive Target Synchronization ............................................... 144
5.5 Conclusions ...................................................................................... 148

6 Derived datatypes ............................................................................... 149
6.1 Introduction ....................................................................................... 150
6.2 Contiguous datatypes ...................................................................... 152
6.3 Constant stride ................................................................................ 153
6.4 Other subroutines ............................................................................ 155
6.5 Examples .......................................................................................... 156
  6.5.1 The datatype "matrix row" ......................................................... 156
  6.5.2 The datatype "matrix line" ........................................................ 158
MPI – Plan IV

8.6 Topologies ................................................................. 224
  8.6.1 Cartesian topologies ............................................... 225
  8.6.2 Subdividing a cartesian topology .............................. 240

9 MPI-IO ........................................................................... 246
  9.1 Introduction ................................................................ 247
    9.1.1 Presentation .......................................................... 247
    9.1.2 Issues .................................................................. 249
    9.1.3 Definitions ........................................................... 252
  9.2 File management .......................................................... 254
  9.3 Reads/Writes : general concepts .................................... 257
  9.4 Individual Reads/Writes ............................................... 261
    9.4.1 Via explicit offsets .................................................. 261
    9.4.2 Via Individual File Pointers ...................................... 270
    9.4.3 Shared File Pointers ............................................... 282
  9.5 Collective Reads/Writes ................................................ 289
    9.5.1 Via explicit offsets .................................................. 290
    9.5.2 Via individual file pointers ...................................... 294
    9.5.3 Via shared file pointers .......................................... 308
  9.6 Explicit positioning of pointers in a file ......................... 314
  9.7 Definition of views ......................................................... 323
  9.8 NonBlocking Reads/Writes .......................................... 377
    9.8.1 Via explicit displacements ...................................... 378
9.8.2 Via individual file pointers ......................................................... 383
9.8.3 NonBlocking and collective Reads/Writes ........................................ 387
9.9 Advices .................................................................................. 392

10 Conclusion ............................................................................. 393
1 Introduction
1.1 Availability and updating .............................................. 8
1.2 Definitions ............................................................... 9
1.3 Message Passing concepts ............................................. 12
1.4 History ......................................................................... 15
1.5 Library .......................................................................... 17

2 Environment

3 Point-to-point Communications

4 Collective communications

5 One-sided Communication

6 Derived datatypes

7 Optimizations

8 Communicators

9 MPI-IO

10 Conclusion
This document is likely to be updated regularly. The most recent version is available on the Web server of IDRIS, section Support de cours: https://cours.idris.fr

IDRIS
Institut du développement et des ressources en informatique scientifique
Rue John Von Neumann
Bâtiment 506
BP 167
91403 ORSAY CEDEX
France
http://www.idris.fr

Translated with the help of Yazeed Zabrie.
1 – Introduction

1.2 – Definitions

The sequential programming model:

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

Figure 1: Sequential programming model
In the message passing programming model:

- the program is written in a classic language (Fortran, C, C++, etc.)
- all the variables of the program are private and reside in the local memory of each process
- each process may execute different parts of a program
- a variable is exchanged between two or more processes via a call to subroutines.
The **SPMD** execution model:
- **Single Program, Multiple Data**;
- the same program is executed by all the processes;
- all machines support this programming model and some support only this one;
- it is a particular case of the general model of **MPMD** (*Multiple Program, Multiple Data*), which it can also emulate.

![Diagram of Single Program, Multiple Data](image)

**Figure 3: Single Program, Multiple Data**
If a message is sent to a process, the latter must receive it.

![Diagram of message passing](image)

Figure 4: Message Passing
A message consists of data chunks passing from the sending process to the receiving process.

In addition to the data (scalar variables, arrays, etc.) to be sent, a message has to contain the following information:

- the identifier of the sending process;
- the datatype;
- the length;
- the identifier of the receiving process.

Figure 5: Message Construction
The exchanged messages are interpreted and managed by an environment comparable to telephony, fax, postal mail, e-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 independent processes executing, each one, their own code and communicating via calls to MPI library subroutines.
Version 1.0: in June 1994, the MPI forum (Message Passing Interface Forum), with the participation of about forty organizations, came to the definition of a set of subroutines concerning the MPI message passing library.

Version 1.1: June 1995, with only minor changes.

Version 1.2: in 1997, with minor changes for a better consistency of the naming of some subroutines.

Version 1.3: September 2008, with clarifications in MPI 1.2, according to clarifications themselves made by MPI-2.1

Version 2.0: released in July 1997, this version brought deliberately non-integrated, essential additions in MPI 1.0 (process dynamic management, one-sided communications, parallel I/O, etc.).

Version 2.1: June 2008, with only clarifications in MPI 2.0 but without any changes.

Version 2.2: September 2009, with only "small" additions
Version 3.0: September 2012

➤ changes and important additions compared to version 2.2;
➤ main changes:
   ➤ nonblocking collective communications;
   ➤ revision of the implementation of one-sided communications;
   ➤ Fortran (2003-2008) bindings;
   ➤ C++ bindings removed;
   ➤ interfacing of external tools (for debugging and performance measures);
   ➤ etc.

➤ see http://meetings.mpi-forum.org/MPI_3.0_main_page.php
https://svn.mpi-forum.org/trac/mpi-forum-web/wiki
https://fs.hlrs.de/projects/par/mpi/mpi30/

https://fs.hlrs.de/projects/par/mpi/mpi22/


Peter S. Pacheco: *Parallel Programming with MPI*, Morgan Kaufman Ed., 1997

Additional references:
http://www.mpi-forum.org/docs/
http://www.mpi-forum.org/mpi2_1/index.htm
MPI open source implementations: they can be installed on a large number of architectures but their performances are generally below the ones of the constructors’ implementations.

1. **MPICH2**: http://www.mpich.org/
2. **Open MPI**: http://www.open-mpi.org/
The debuggers

2. DDT http://www.allinea.com/products/ddt/

Tools for performance measurements

1. MPE: MPI Parallel Environment
2. Scalasca: Scalable Performance Analysis of Large-Scale Applications
   http://www.scalasca.org/
3. Vampir
   http://www.vampir.eu/
Some open source parallel scientific libraries:

1. **ScalAPACK**: linear algebraic problem solvers using direct methods. The sources can be downloaded from the website http://www.netlib.org/scalapack/

2. **PETSc**: linear and non-linear algebraic problem solvers using iterative methods. The sources can be downloaded from the website http://www.mcs.anl.gov/petsc/

3. **MUMPS**: MUltifrontal Massively Parallel sparse direct Solvers. The sources can be downloaded from the website http://graal.ens-lyon.fr/MUMPS/

4. **FFTW**: fast Fourier transforms. The sources can be downloaded from the website http://www.fftw.org
1 Introduction

2 Environment
   2.1 Description
   2.2 Exemple

3 Point-to-point Communications

4 Collective communications

5 One-sided Communication

6 Derived datatypes

7 Optimizations

8 Communicators

9 MPI-IO

10 Conclusion
Every program unit calling MPI subroutines has to include a header file. In Fortran, we must use the `mpi` module introduced in MPI-2 (in MPI-1, it was the `mpif.h` file), and in C/C++ the `mpi.h` file.

The `MPI_INIT()` subroutine initializes the necessary environment:

```fortran
integer, intent(out) :: code
call MPI_INIT(code)
```

The `MPI_FINALIZE()` subroutine disables this environment:

```fortran
integer, intent(out) :: code
call MPI_FINALIZE(code)
```
All the operations made by MPI are related to **communicators**. The default communicator is **MPI_COMM_WORLD** which includes all the active processes.

![MPI_COMM_WORLD Communicator](image)

**Figure 6: MPI_COMM_WORLD Communicator**
At any moment, we can know the number of processes managed by a given communicator by the `MPI_COMM_SIZE()` subroutine:

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

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

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

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

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, code)
```
program who_am_I
  use mpi
  implicit none
  integer :: nb_procs, rank, code

  call MPI_INIT(code)

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

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

  call MPI_FINALIZE(code)
end program who_am_I

> mpiexec -n 7 who_am_I
I am the process 3 among 7
I am the process 0 among 7
I am the process 4 among 7
I am the process 1 among 7
I am the process 5 among 7
I am the process 2 among 7
I am the process 6 among 7
1 Introduction

2 Environment

3 Point-to-point Communications
   3.1 General Concepts ................................................................. 27
   3.2 Predefined MPI Datatypes ......................................................... 33
   3.3 Other Possibilities ................................................................. 35
   3.4 Example: communication ring ................................................. 41

4 Collective communications

5 One-sided Communication

6 Derived datatypes

7 Optimizations

8 Communicators

9 MPI-IO

10 Conclusion
A point-to-point communication occurs between two processes, one is called the sender process and the other one is called the receiver process.
A point-to-point communication occurs between two processes, one is called the sender process and the other one is called the receiver process.

Figure 7: Point-to-point communication
A point-to-point communication occurs between two processes, one is called the **sender** process and the other one is called the **receiver** process.
A **point-to-point** communication occurs between two processes, one is called the **sender** process and the other one is called the **receiver** process.

![Diagram of point-to-point communication](image-url)
The sender and the receiver are identified by their rank in the communicator.

The so-called message envelope is composed of:

1. the rank of the sender process;
2. the rank of the receiver process;
3. the tag of the message;
4. the communicator which define the process group and communication context.

The exchanged data are predefined (integer, real, etc) or personal derived datatypes.

There are in each case many communication modes, calling different protocols which will be discussed in Chapter 7.
program point_to_point
  use mpi
  implicit none

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

  call MPI_INIT(code)

  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, code)

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

  call MPI_FINALIZE(code)

end program point_to_point

> mpiexec -n 7 point_to_point

I, process 5, I have received 1000 from the process 2
### 3 – Point-to-point Communications

#### 3.2 – Predefined MPI Datatypes

<table>
<thead>
<tr>
<th>MPI Type</th>
<th>Fortran Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER</td>
</tr>
</tbody>
</table>

Table 1: Predefined MPI Datatypes (Fortran)
<table>
<thead>
<tr>
<th>MPI Type</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
</tbody>
</table>

Table 2: Predefined MPI Datatypes (C)
On the reception of a message, the rank of the sender and the tag can be replaced respectively by the `MPI_ANY_SOURCE` and `MPI_ANY_TAG` wildcards.

A communication with the dummy process of rank `MPI_PROC_NULL` has no effect.

`MPI_STATUS_IGNORE` is a predefined constant that can be used instead of the status variable.

`MPI_SUCCESS` is a predefined constant which allows testing the return code of a MPI subroutine.

There are syntactic variants, `MPI_SENDRECV()` and `MPI_SENDRECV_REPLACE()`, which combine a send and a receive operations (for the former, the receive memory zone must be different from the send one).

It is possible to create more complex data structures using derived datatypes (see the Chapter 6)
program sendrecv
  use mpi
  implicit none
  integer :: rank, value, num_proc, code
  integer, parameter :: tag=110

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

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

  call MPI_SENDrecv(rank+1000,1, MPI_INTEGER, num_proc, tag, value, 1, MPI_INTEGER, &
                    num_proc, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE, code)

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

  call MPI_FINALIZE(code)
end program sendrecv
Warning! It must be noticed that if the **MPI_SEND()** subroutine is implemented in a **synchronous** way (see the Chapter 7) in the used MPI implementation, the previous code will deadlock if, rather than to use the **MPI_SENDRECV()** subroutine, we used a **MPI_SEND()** subroutine followed by a **MPI_RECV()** one. In this case, each of the two processes would wait a receive command which will never occur, because the two sends would stay suspended. Therefore, for correctness and portability reasons, it is absolutely necessary to avoid such situations.

```fortran
    call MPI_SEND(rank+1000,1,MPI_INTEGER,num_proc,tag,MPI_COMM_WORLD,code)
    call MPI_RECV(value,1,MPI_INTEGER,num_proc,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE,code)
```
program joker
  use mpi
  implicit none
  integer, parameter :: m=4, tag1=11, tag2=22
  integer, dimension(m,m) :: A
  integer, dimension(m,m) :: nb_procs, rank, code
  integer, dimension(MPI_STATUS_SIZE) :: status
  integer :: nb_elements, i
  integer, dimension(:), allocatable :: C
  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nb_procs, code)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, code)
  A(:, :) = 0
if (rank == 0) then
    ! Initialisation of the matrix A on the process 0
    A(:, :) = reshape((/ (i, i=1, m*m) /), (/ m, m /))
    ! Sending of 2 elements of the matrix A to the process 1
    call MPI_SEND(A(1, 1), 2, MPI_INTEGER, 1, tag1, MPI_COMM_WORLD, code)
    ! Sending of 3 elements of the matrix A to the process 2
    call MPI_SEND(A(1, 2), 3, MPI_INTEGER, 2, tag2, MPI_COMM_WORLD, code)
else
    ! We check before the receive if a message has arrived and from which process
    call MPI_PROBE(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status, code)
    ! We check how many elements must be received
    call MPI_GET_COUNT(status, MPI_INTEGER, nb_elements, code)
    ! We allocate the reception array C on each process
    if (nb_elements /= 0 ) allocate (C(1:nb_elements))
    ! We receive the message
    call MPI_RECV(C, nb_elements, MPI_INTEGER, status(MPI_SOURCE), status(MPI_TAG), &
                              MPI_COMM_WORLD, status, code)
    print *, 'I, process ', rank, ', I have received ', nb_elements, &
    'elements from the process ', &
    status(MPI_SOURCE), 'My array C is ', C(:)
end if

call MPI_FINALIZE(code)
end program joker
mpiexec -n 3 sendrecv1

I, process 1, I have received 2 elements from the process 0. My array C is 1 2.

I, process 2, I have received 3 elements from the process 0. My array C is 5 6 7.
3 – Point-to-point Communications
3.4 – Example: communication ring

Figure 9: Communication Ring
3 – Point-to-point Communications
3.4 – Example: communication ring

Figure 9: Communication Ring
If all the processes execute a send followed by a receive, all the communications could potentially start simultaneously and then will not occur in a ring way (outside the already mentioned portability problem if the implementation of `MPI_SEND()` is made in a synchronous way in the MPI library):

```fortran
...  
value=rank+1000  
call MPI_SEND(value,1,MPI_INTEGER,num_proc_next,tag,MPI_COMM_WORLD,code)  
call MPI_RECV(value,1,MPI_INTEGER,num_proc_previous,tag,MPI_COMM_WORLD,&  
   status,code)  
...  
```
To ensure that the communications will really happen in a ring way, like for the exchange of a token between processes, it is necessary to use a different method and to have one process which starts the chain:

Figure 10: Communication Ring
To ensure that the communications will really happen in a ring way, like for the exchange of a token between processes, it is necessary to use a different method and to have one process which starts the chain:

Figure 10: Communication Ring
To ensure that the communications will really happen in a ring way, like for the exchange of a token between processes, it is necessary to use a different method and to have one process which starts the chain:

![Communication Ring Diagram](image)

Figure 10: Communication Ring
To ensure that the communications will really happen in a ring way, like for the exchange of a token between processes, it is necessary to use a different method and to have one process which starts the chain:

Figure 10: Communication Ring
To ensure that the communications will really happen in a ring way, like for the exchange of a token between processes, it is necessary to use a different method and to have one process which starts the chain:

![Communication Ring Diagram]

Figure 10: Communication Ring
To ensure that the communications will really happen in a ring way, like for the exchange of a token between processes, it is necessary to use a different method and to have one process which starts the chain:

![Communication Ring](image)

**Figure 10: Communication Ring**
To ensure that the communications will really happen in a ring way, like for the exchange of a token between processes, it is necessary to use a different method and to have one process which starts the chain:

Figure 10: Communication Ring
To ensure that the communications will really happen in a ring way, like for the exchange of a token between processes, it is necessary to use a different method and to have one process which starts the chain:

Figure 10: Communication Ring
program ring
  use mpi
  implicit none
  integer, dimension(MPI_STATUS_SIZE) :: status
  integer, parameter :: tag=100
  integer :: nb_procs,rank,value, &
              num_proc_previous,num_proc_next,code
  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nb_procs, code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)

  num_proc_next=mod(rank+1,nb_procs)
  num_proc_previous=mod(nb_procs+rank-1,nb_procs)

  if (rank == 0) then
    call MPI_SEND(rank+1000,1,MPI_INTEGER,num_proc_next,tag, &
                  MPI_COMM_WORLD,code)
    call MPI_RECV(value,1,MPI_INTEGER,num_proc_previous,tag, &
                   MPI_COMM_WORLD,status,code)
  else
    call MPI_RECV(value,1,MPI_INTEGER,num_proc_previous,tag, &
                   MPI_COMM_WORLD,status,code)
    call MPI_SEND(rank+1000,1,MPI_INTEGER,num_proc_next,tag, &
                  MPI_COMM_WORLD,code)
  end if

  print *,'I, process ',rank,', I have received ',value,' from process ', &
          num_proc_previous
  call MPI_FINALIZE(code)
end program ring
mpiexec -n 7 ring

I, process 1, I have received 1000 from process 0
I, process 2, I have received 1001 from process 1
I, process 3, I have received 1002 from process 2
I, process 4, I have received 1003 from process 3
I, process 5, I have received 1004 from process 4
I, process 6, I have received 1005 from process 5
I, process 0, I have received 1006 from process 6
4 Collective communications

4.1 General concepts ................................................................. 55
4.2 Global synchronization: MPI_BARRIER() .................................. 57
4.3 Broadcast: MPI_BCAST() ....................................................... 60
4.4 Scatter: MPI_SCATTER() ....................................................... 66
4.5 Gather: MPI_GATHER() .......................................................... 72
4.6 Gather-to-all: MPI_ALLGATHER() ............................................. 78
4.7 Extended gather: MPI_GATHERV() .......................................... 84
4.8 All-to-all: MPI_ALLTOALL() ............................................... 91
4.9 Global reduction .................................................................. 98
4.10 Additions ........................................................................... 120

5 One-sided Communication
6 Derived datatypes
7 Optimizations
8 Communicators
9 MPI-IO
10 Conclusion
The collective communications allow to make a series of point-to-point communications in one single call.

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

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

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

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

1. the one which ensures the global synchronizations: `MPI_BARRIER()`.
2. the ones which only transfer data:
   - global distribution of data: `MPI_BCAST()`;
   - selective distribution of data: `MPI_SCATTER()`;
   - collection of distributed data: `MPI_GATHER()`;
   - collection by all the processes of distributed data: `MPI_ALLGATHER()`;
   - selective distribution, by all the processes, of distributed data: `MPI_ALLTOALL()`.
3. the ones which, in addition to the communications management, carry out operations on the transferred data:
   - reduction operations (sum, product, maximum, minimum, etc.) whether they are of a predefined or personal type: `MPI_REDUCE()`;
   - reduction operations with broadcasting of the result (it is in fact equivalent to an `MPI_REDUCE()` followed by an `MPI_BCAST()`): `MPI_ALLREDUCE()`.
4 – Collective communications

4.2 – Global synchronization: MPI_BARRIER()

Figure 11: Global Synchronization: MPI_BARRIER()

integer, intent(out) :: code

call MPI_BARRIER(MPI_COMM_WORLD, code)
4 – Collective communications
4.2 – Global synchronization: MPI_BARRIER()

integer, intent(out) :: code

call MPI_BARRIER(MPI_COMM_WORLD, code)

Figure 11: Global Synchronization : MPI_BARRIER()
4 – Collective communications

4.2 – Global synchronization: MPI_BARRIER()

**Figure 11:** Global Synchronization: MPI_BARRIER()

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

call MPI_BARRIER(MPI_COMM_WORLD,code)
```
4 – Collective communications
4.3 – Broadcast : MPI_BCAST()
4 – Collective communications
4.3 – Broadcast: MPI_BCAST()
4 – Collective communications
4.3 – Broadcast: MPI_BCAST()

Figure 12: Broadcast: MPI_BCAST()
4 – Collective communications

4.3 – Broadcast : MPI_BCAST()

Figure 12: Broadcast : MPI_BCAST()
Figure 12: Broadcast: MPI_BCAST()
program bcast
  use mpi
  implicit none

  integer :: rank, value, code

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

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

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

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

  call MPI_FINALIZE(code)
end program bcast

> mpiexec -n 4 bcast
I, process 2 I have received 1002 of the process 2
I, process 0 I have received 1002 of the process 2
I, process 1 I have received 1002 of the process 2
I, process 3 I have received 1002 of the process 2
4 – Collective communications

4.4 – Scatter: MPI_SCATTER()
4 – Collective communications
4.4 – Scatter: MPI_SCATTER()
4 – Collective communications
4.4 – Scatter : MPI_SCATTER()
4 – Collective communications

4.4 – Scatter: MPI_SCATTER()
Figure 13: Scatter: MPI_SCATTER()
program scatter
  use mpi
  implicit none

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

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

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

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

> mpiexec -n 4 scatter
I, process 2 send my values array :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
I, process 0, I have received 1001. 1002. of the process 2
I, process 1, I have received 1003. 1004. of the process 2
I, process 3, I have received 1007. 1008. of the process 2
I, process 2, I have received 1005. 1006. of the process 2
4 – Collective communications
4.5 – Gather: MPI Gather

Figure 14: Gather: MPI Gather

P0
A0
P1
A1
P2
A2
P3
A3

MPI_GATHER()
4 – Collective communications

4.5 – Gather: MPI_GATHER()

Figure 14: Gather: MPI_GATHER()
4 – Collective communications
4.5 – Gather: MPI_GATHER()
4 – Collective communications

4.5 – Gather : **MPI_GATHER()**

**Figure 14**: Gather : **MPI_GATHER()**
4 – Collective communications
4.5 – Gather: MPI_GATHER()
program gather
  use mpi
  implicit none
  integer, parameter :: nb_values=8
  integer :: nb_procs, rank, block_length, i, code
  real, dimension(nb_values) :: data
  real, allocatable, dimension(:) :: values

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

  block_length=nb_values/nb_procs
  allocate(values(block_length))

  values(:)=(/(1000.+rank*block_length+i,i=1,block_length)/)

  print *, 'I, process ', rank, ', send my values array : ', &
  values(1:block_length)

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

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

  call MPI_FINALIZE(code)
end program gather

> mpiexec -n 4 gather
I, process 1 send my values array : 1003. 1004.
I, process 0 send my values array : 1001. 1002.
I, process 2 send my values array : 1005. 1006.
I, process 3 send my values array : 1007. 1008.
I, process 2 have received 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
4 – Collective communications

4.6 – Gather-to-all: MPI_ALLGATHER()
4 – Collective communications

4.6 – Gather-to-all: MPI_ALLGATHER()

Figure 15: Allgather: MPI_ALLGATHER()
4 – Collective communications
4.6 – Gather-to-all: MPI_ALLGATHER()
4 – Collective communications
4.6 – Gather-to-all: MPI_ALLGATHER()
4 – Collective communications

4.6 – Gather-to-all: MPI_ALLGATHER()
program allgather
  use mpi
  implicit none

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

  call MPI_INIT(code)

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

  block_length=nb_values/nb_procs
  allocate(values(block_length))

  values(:)=/(1000.+rank*block_length+i, i=1, block_length)/

  call MPI_ALLGATHER(values, block_length, MPI_REAL, data, block_length, &
                     MPI_REAL, MPI_COMM_WORLD, code)

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

  call MPI_FINALIZE(code)
end program allgather

> mpiexec -n 4 allgather

I, process 1, I have received 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
I, process 3, I have received 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
I, process 2, I have received 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
I, process 0, I have received 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
4 – Collective communications
4.7 – Extended gather: MPI_GATHERV()

Figure 16: GatherV: MPI_GATHERV()
4 – Collective communications

4.7 – Extended gather: MPI_GATHERV()

Figure 16: Gatherv: MPI_GATHERV()}
4 – Collective communications

4.7 – Extended gather: MPI_GATHERV()
4 – Collective communications
4.7 – Extended gather: MPI_GATHERV()
Figure 16: Gatherv : MPI_GATHERV()
program gatherv
  use mpi
  implicit none

  INTEGER, PARAMETER :: nb_values=10
  INTEGER :: nb_procs, rank, block_length, i, code
  REAL, DIMENSION(nb_values) :: data, remainder
  REAL, ALLOCATABLE, DIMENSION(:) :: values
  INTEGER, ALLOCATABLE, DIMENSION(:) :: nb_elements_received, displacement

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

  block_length=nb_values/nb_procs
  remainder=mod(nb_values,nb_procs)
  if(remainder > rank) block_length = block_length + 1
  ALLOCATE(values(block_length))
  values(:) = (/((1000.+(rank*(nb_values/nb_procs))+min(rank,remainder)+i, &
                 i=1,block_length)/)

  PRINT *, 'I, process ', rank,'send my values array : ',
        values(1:block_length)

  IF (rank == 2) THEN
    ALLOCATE(nb_elements_received(nb_procs),displacement(nb_procs))
    nb_elements_received(1) = nb_values/nb_procs
    if (remainder > 0) nb_elements_received(1)=nb_elements_received(1)+1
    displacement(1) = 0
    DO i=2,nb_procs
      displacement(i) = displacement(i-1)+nb_elements_received(i-1)
      nb_elements_received(i) = nb_values/nb_procs
      if (remainder > i-1) nb_elements_received(i)=nb_elements_received(i)+1
    END DO
  END IF
END
CALL MPI_GATHERV(values, block_length, MPI_REAL, data, nb_elements_received, &displacement, MPI_REAL, 2, MPI_COMM_WORLD, code)

IF (rank == 2) PRINT *, 'I, processus 2 receives ', data(1:nb_values)
CALL MPI_FINALIZE(code)

end program gatherv

> mpiexec -n 4 gatherv

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

4 – Collective communications
4.8 – All-to-all : MPI_ALLTOALL()
4 – Collective communications

4.8 – All-to-all: MPI_ALLTOALL()

Figure 17: Alltoall: MPI_ALLTOALL()
4 – Collective communications
4.8 – All-to-all: MPI_ALLTOALL()

Figure 17: Alltoall: MPI_ALLTOALL()
4 – Collective communications

4.8 – All-to-all: MPI_ALLTOALL()
4 – Collective communications
4.8 – All-to-all: MPI_ALLTOALL()
program alltoall
  use mpi
  implicit none

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

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

  values(:)=(/(1000.+rank*nb_values+i, i=1,nb_values)/)
  block_length=nb_values/nb_procs

  print *,’I, process ’, rank,’send my values array : ’,&
          values(1:nb_values)

  call MPI_ALLTOALL(values, block_length, MPI_REAL, data, block_length, &
                    MPI_REAL, MPI_COMM_WORLD, code)

  print *,’I, process ’, rank,’, I have received ’, data(1:nb_values)

  call MPI_FINALIZE(code)
end program alltoall
mpiexec -n 4 alltoall

I, process 1 send my values array:
1009. 1010. 1011. 1012. 1013. 1014. 1015. 1016.

I, process 0 send my values array:
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.

I, process 2 send my values array:
1017. 1018. 1019. 1020. 1021. 1022. 1023. 1024.

I, process 3 send my values array:
1025. 1026. 1027. 1028. 1029. 1030. 1031. 1032.

I, process 0, I have received 1001. 1002. 1009. 1010. 1017. 1018. 1025. 1026.

I, process 2, I have received 1005. 1006. 1013. 1014. 1021. 1022. 1029. 1030.

I, process 1, I have received 1003. 1004. 1011. 1012. 1019. 1020. 1027. 1028.

I, process 3, I have received 1007. 1008. 1015. 1016. 1023. 1024. 1031. 1032.
A reduction is an operation applied to a set of elements in order to obtain one single value. Classical examples are the sum of the elements of a vector (\texttt{SUM(A(:))}) or the search of the maximum value element in a vector (\texttt{MAX(V(:))}).

MPI proposes high-level subroutines in order to operate reductions on distributed data on a group of processes. The result is obtained on one process (\texttt{MPI_REDUCE()}) or on all (\texttt{MPI_ALLREDUCE()}), which is in fact equivalent to an \texttt{MPI_REDUCE()} followed by an \texttt{MPI_BCAST()}. If several elements are implied by process, the reduction function is applied to each one of them.

The \texttt{MPI_SCAN()} subroutine allows also to make partial reductions by considering, for each process, the previous processes of the group and itself.

The \texttt{MPI_OP_CREATE()} and \texttt{MPI_OP_FREE()} subroutines allow personal reduction operations.
Table 3: Main Predefined Reduction Operations (there are also other logical operations)

<table>
<thead>
<tr>
<th>Name</th>
<th>Opération</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUM</td>
<td>Sum of elements</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product of elements</td>
</tr>
<tr>
<td>MPI_MAX</td>
<td>Maximum of elements</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum of elements</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum of elements and location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum of elements and location</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
</tbody>
</table>
Figure 18: Distributed reduction (sum)
Figure 18: Distributed reduction (sum)
Figure 18: Distributed reduction (sum)
Figure 18: Distributed reduction (sum)
Figure 18: Distributed reduction (sum)
Figure 18: Distributed reduction (sum)
Figure 18: Distributed reduction (sum)

1000 + 1 + 2 + 3 + 4 + 5
Figure 18: Distributed reduction (sum)
program reduce
  use mpi
  implicit none
  integer :: nb_procs,rank,value,sum,code

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

  if (rank == 0) then
    value=1000
  else
    value=rank
  endif

  call MPI_REDUCE(value,sum,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,code)

  if (rank == 0) then
    print *,’I, process 0, I have the global sum value ’,sum
  end if

  call MPI_FINALIZE(code)
end program reduce

> mpiexec -n 7 reduce
I, process 0, I have the global sum value 1021
Figure 19: Distributed reduction (product) with broadcast of the result
Figure 19: Distributed reduction (product) with broadcast of the result
Figure 19: Distributed reduction (product) with broadcast of the result
Figure 19: Distributed reduction (product) with broadcast of the result
Figure 19: Distributed reduction (product) with broadcast of the result
Figure 19: Distributed reduction (product) with broadcast of the result
Figure 19: Distributed reduction (product) with broadcast of the result
Figure 19: Distributed reduction (product) with broadcast of the result

\[10 \times 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 7200\]
Figure 19: Distributed reduction (product) with broadcast of the result
program allreduce

    use mpi
    implicit none

    integer :: nb_procs,rank,value,product,code

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

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

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

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

    call MPI_FINALIZE(code)

end program allreduce
mpiexec -n 7 allreduce

I, process 6, I have received the value of the global product 7200
I, process 2, I have received the value of the global product 7200
I, process 0, I have received the value of the global product 7200
I, process 4, I have received the value of the global product 7200
I, process 5, I have received the value of the global product 7200
I, process 3, I have received the value of the global product 7200
I, process 1, I have received the value of the global product 7200
The MPI_SCATTERV(), MPI_GATHERV(), MPI_ALLGATHERV() and MPI_ALLTOALLV() subroutines extend MPI_SCATTER(), MPI_GATHER(), MPI_ALLGATHER() and MPI_ALLTOALL() in the case where the number of elements to transmit or gather is different following the processes.

Two new subroutines have been added to extend the possibilities of collective subroutines in some particular cases:

- MPI_ALLTOALLW() : MPI_ALLTOALLV() version where the displacements are expressed in bytes and not in elements,
- MPI_EXSCAN() : exclusive version of MPI_SCAN().
5 One-sided Communication
5.1 Introduction
5.1.1 Reminder: The Concept of Message-Passing
5.1.2 The Concept of One-sided Communication
5.1.3 RMA Approach of MPI
5.2 Memory Window
5.3 Data Transfer
5.4 Completion of the Transfer: the synchronization
5.4.1 Active Target Synchronization
5.4.2 Passive Target Synchronization
5.5 Conclusions
There are various approaches to transfer data between two different processes. Among the most commonly used are:

1. Point-to-point communications by message-passing (MPI, etc.)
2. One-sided communications (direct access to the memory of a distant process). Also called RMA for Remote Memory Access, it is one of the major contributions of MPI.
In message-passing, a sender (origin) sends a message to a destination process (target) which will make all what is necessary to receive this message. This requires that the sender as well as the receiver be involved in the communication. This can be restrictive and difficult to implement in some algorithms (for example when it is necessary to manage a global counter).
The concept of one-sided communication is not new, MPI having simply unified the already existing constructors’ solutions (such as shmem (CRAY), lapi (IBM), ...) by offering its own RMA primitives. Through these subroutines, a process has a direct access (in read, write or update) to the memory of another remote process. In this approach, the remote process does not have to participate in the data-transfer process.

The principle advantages are the following:

- enhanced performances when the hardware allows it,
- a simpler programming for some algorithms.
The use of MPI RMA is done in three steps:

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

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

3. completion of current transfers by a step of synchronization, the data are then available.
All the processes participating in an one-sided communication have to specify which part of their memory will be available to the other processes; it is the notion of memory window.

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

MPI Window Object win1

0

First local window

Second local window

1

MPI Window Object win2

Second local window

First local window

Figure 21: Creation of two MPI window objects, win1 and win2
Once the transfers are finished, the window has to be freed with the
\texttt{MPI\_WIN\_FREE()} subroutine.

\texttt{MPI\_WIN\_GET\_ATTR()} allows to know the characteristics of a local memory
window by using the keywords \texttt{MPI\_WIN\_BASE}, \texttt{MPI\_WIN\_SIZE} or
\texttt{MPI\_WIN\_DISP\_UNIT}.

Remark :

The choice of the displacement unit associated with the local memory window is
important (essential in a heterogenous environment and facilitating the encoding
in all the cases). The size of a MPI datatype is obtained by calling the
\texttt{MPI\_TYPE\_SIZE()} subroutine.
program window

    use mpi   
    implicit none                      

    integer :: code, rank, size_real, win, n=4
    integer (kind=MPI_ADDRESS_KIND) :: dim_win, size, base, unit
    real(kind=kind(1.d0)), dimension(:,), allocatable :: win_local
    logical :: flag                      

    call MPI_INIT(code)                   
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, code) 
    call MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION, size_real, code) 

    if (rank==0) n=0
    allocate(win_local(n))               
    dim_win = size_real*n                

    call MPI_WIN_CREATE(win_local, dim_win, size_real, MPI_INFO_NULL, & 
                        MPI_COMM_WORLD, win, code) 
    call MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, code) 
    call MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, code) 
    call MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, unit, flag, code) 
    call MPI_WIN_FREE(win,code)          
    print *,"process", rank,"size, base, unit = " & 
         ,size, base, unit                
    call MPI_FINALIZE(code)              
end program window

> mpiexec -n 3 window
process 1 size, base, unit = 32 17248330400 8
process 0 size, base, unit = 0 2 8
process 2 size, base, unit = 32 17248330400 8
MPI allows a process to read (\texttt{MPI\_GET()})
, to write (\texttt{MPI\_PUT()}) and to update
(\texttt{MPI\_ACCUMULATE()}) data located in the local memory window of a remote process.

We call the process which makes the calling of the initialization subroutine of the
transfer \texttt{origin} and we call the process which has the local memory window which will
be used in the transfer procedure \texttt{target}.

During the initialization of the transfer, the target process does not call any MPI
subroutine. All the necessary informations are specified under the form of parameters
during the calling of the MPI subroutine by the origin.
In particular, we find:

☞ parameters related to the origin:
  - the datatype of elements;
  - their number;
  - the memory address of the first element.

☞ parameters related to the target:
  - the rank of the target as well as the MPI window object, which uniquely determines a local memory window;
  - a displacement in this local window;
  - the number of elements to be transferred and their datatype.
![Example of a MPI_PUT](image)

**Displacement of 40 units in the local window**

**First Local Window**

- B(1:10)

**First Local Window**
Remarks:

☞ The MPI_GET syntax is identical to the MPI_PUT syntax, only the direction of the data transfer is reversed.

☞ The RMA transfer data subroutines are nonblocking primitives (deliberate choice of MPI).

☞ On the target process, the only accessible data are the ones contained in the local memory window.

☞ MPI_ACCUMULATE() admits among its parameters an operation which should be either of the MPI_REPLACE type, or one of the predefined reduction operations: MPI_SUM, MPI_PROD, MPI_MAX, etc. This cannot be in any case an operation defined by the user.
The data transfer starts after the call to one of the nonblocking subroutines (MPI_PUT(), ...). But when is the transfer completed and the data really available?

After a synchronization by the programmer.

These synchronizations can be classified into two types:

1. **Active target** synchronization (collective operation, all the processes associated with the window involved in the synchronization);

2. **Passive target** synchronization (only the origin process calls the synchronization subroutine).
5 – One-sided Communication

5.4 – Completion of the Transfer: the synchronization

5.4.1 – Active Target Synchronization

☞ It is made by using the MPI subroutine **MPI_WIN_FENCE()**.

☞ **MPI_WIN_FENCE()** is a collective operation on all the processes associated with the MPI window object.

☞ **MPI_WIN_FENCE()** acts as a synchronization barrier. It waits for the end of all the data transfers (RMA or not) using the local memory window and initiated since the last call to **MPI_WIN_FENCE()**.

☞ This primitive will help separate the calculation parts of the code (where data of the local memory window are used via *load* or *store*) of RMA data transfer parts.

☞ An *assert* argument of the **MPI_WIN_FENCE()** primitive, of integer type, allows to refine its behavior for better performances. Different values are predefined **MPI_MODE_NOSTORE**, **MPI_MODE_NOPUT**, **MPI_MODE_NOPRECEDE**, **MPI_MODE_NOSUCCEED**. A zero value for this argument is always valid.
Remarks:

☞ The fact of having chosen nonblocking RMA subroutines for initialisation of the transfers and having a synchronization for the completion of actual transfers authorizes the implementation to gather, during the execution, different transfers towards the same target in one single transfer. The latencies effects are therefore reduced and the performances are improved.

☞ The collective character of the synchronization results in that we have nothing to do with One Sided Communication... In fact all the processes of the communicator will have to participate in the synchronization, which makes it of less interest!
The Proper Use of **MPI_WIN_FENCE()**

It is important to make sure that between two successive calls to **MPI_WIN_FENCE()** either there is only local operations (load/store) on variables that are in the local memory window of the process, or there is **MPI_PUT()** RMA operations or **MPI_ACCUMULATE()**, but never both at once!

```
0  MPI_WIN_FENCE()  1  MPI_WIN_FENCE()
     |                        MPI_WIN_FENCE()
0  MPI_WIN_FENCE()  win_loc(:) = win_loc(:) + 1.0
     MRI_PUT()
     MPI_WIN_FENCE()
     MPI_WIN_FENCE()
```

Is the previous program compliant with the proper use of **MPI_WIN_FENCE()**?

All depends on the code portion represented by ①. If this latter does not produce a load/store on the local window (assignment or use of a variable stored in the local window), then it is correct; otherwise, the result is undefined.
Example

Figure 23: Example corresponding to the ex_fence program
program ex_fence
    use mpi
    implicit none

    integer, parameter :: assert=0
    integer :: code, rank, size_real, win, i, nb_elements, target, m=4, n=4
    integer (kind=MPI_ADDRESS_KIND) :: displacement, dim_win
    real(kind=kind(1.d0)), dimension(:), allocatable :: win_local, tab

    call MPI_INIT(code)
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, code)
    call MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION, size_real, code)

    if (rank==0) then
        n=0
        allocate(tab(m))
    endif

    allocate(win_local(n))
    dim_win = size_real*n

    call MPI_WIN_CREATE(win_local, dim_win, size_real, &
                        MPI_INFO_NULL, &
                        MPI_COMM_WORLD, win, code)
if (rank==0) then
    tab(:) = (/ (i, i=1,m) /)
else
    win_local(:) = 0.0
end if

call MPI_WIN_FENCE(assert,win,code)
if (rank==0) then
    target = 1; nb_elements = 2; displacement = 1
    call MPI_PUT(tab, nb_elements, MPI_DOUBLE_PRECISION, target, displacement, &
                 nb_elements, MPI_DOUBLE_PRECISION, win, code)
end if

call MPI_WIN_FENCE(assert,win,code)
if (rank==0) then
    tab(m) = sum(tab(1:m-1))
else
    win_local(n) = sum(win_local(1:n-1))
endif

call MPI_WIN_FENCE(assert,win,code)
if (rank==0) then
    nb_elements = 1; displacement = m-1
    call MPI_GET(tab, nb_elements, MPI_DOUBLE_PRECISION, target, displacement, &
                 nb_elements, MPI_DOUBLE_PRECISION, win, code)
end if
call MPI_WIN_FENCE(assert, win, code)
if (rank==0) then
    tab(m) = sum(tab(1:m-1))
else
    win_local(:) = win_local(:) + 1
endif

call MPI_WIN_FENCE(assert, win, code)
if (rank==0) then
    nb_elements = m-1; displacement = 1
    call MPI_ACCUMULATE(tab(2), nb_elements, MPI_DOUBLE_PRECISION, target, &
                        displacement, nb_elements, MPI_DOUBLE_PRECISION, &
                        MPI_SUM, win, code)
end if

call MPI_WIN_FENCE(assert, win, code)
call MPI_WIN_FREE(win, code)
if (rank==0) then
    print *,"process", rank, "tab=",tab(:)
else
    print *,"process", rank, "win_local=",win_local(:)
endif

call MPI_FINALIZE(code)
end program ex_fence
Some Details and Restrictions...

It is possible to work on different local memory windows which overlap, even if this is not recommended (such a use implies so many restrictions). We will presume next never being in this case.

It is always important to separate by a call to \texttt{MPI\_WIN\_FENCE()} a store and by a call to the \texttt{MPI\_PUT()} or \texttt{MPI\_ACCUMULATE()} subroutine that accesses the same local memory window even at different locations that do not overlap.

Between two successive calls to the \texttt{MPI\_WIN\_FENCE()} subroutine, we have the following constraints:

- the \texttt{MPI\_PUT()} subroutines do not allow the overlapping inside the same local memory window. In other terms, the memory areas involved during calls to many \texttt{MPI\_PUT()} subroutines that modify the same local memory window, must not overlap;
• the **MPI_ACCUMULATE()** subroutines allow the overlapping inside the same local memory window, on the condition that the datatypes and the used reduction operations be identical during all of these calls;
• using consecutively the **MPI_PUT()** and **MPI_ACCUMULATE()** subroutines do not allow the overlapping inside the same local memory window;
• a *load* and a call to the **MPI_GET()** subroutine can access concurrently whatever part of the local window, provided that it was not updated previously either by a *store*, or when calling the **MPI_PUT()** or **MPI_ACCUMULATE()** subroutine.
It is made via calls to the MPI `MPI_WIN_LOCK()` and `MPI_WIN_UNLOCK()` subroutines.

Unlike the synchronization by `MPI_WIN_FENCE()` (which is a collective operation of barrier type), here only the origin process will participate in the synchronization. Consequently, all the necessary calls to the data transfer (initialization of the transfer, synchronization) are only made by the origin process; this is true One Sided Communication.

The lock and unlock operations apply only to a given local memory window (i.e. identified by a target process number and an MPI window object). The period which starts at lock and ends at unlock is called an access period to the local memory window. It is only during this period that the origin process will have access to the local memory window of the target process.
In order to use it, it only requires the origin process to surround the call to the RMA initialization primitives of data transfer by `MPI_WIN_LOCK()` and `MPI_WIN_UNLOCK()`. For the target process, no MPI subroutine call is necessary.

When `MPI_WIN_UNLOCK()` returns control, all the data transfers initiated after the `MPI_WIN_LOCK()` are completed.

The first `MPI_WIN_LOCK()` argument allows to specify if the fact of making many simultaneous accesses via RMA operations on the same local memory window is authorized (`MPI_LOCK_SHARED`) or not (`MPI_LOCK_EXCLUSIVE`).

A basic use of passive target synchronizations consists of creating RMA blocking versions (`put, get, accumulate`) without the target having to call MPI subroutines.
subroutine get_blocking(orig_addr, orig_count, orig_datatype, target_rank, &
   target_disp, target_count, target_datatype, win, code)
   integer, intent(in) :: orig_count, orig_datatype, target_rank, target_count, &
   target_datatype, win
   integer, intent(out) :: code
   integer(kind=MPI_ADDRESS_KIND), intent(in) :: target_disp
   real(kind=kind(1.d0)), dimension(:) :: orig_addr
   call MPI_WIN_LOCK(MPI_LOCK_SHARED, target_rank, 0, win, code)
   call MPI_GET(orig_addr, orig_count, orig_datatype, target_rank, target_disp, &
   target_count, target_datatype, win, code)
   call MPI_WIN_UNLOCK(target_rank, win, code)
end subroutine get_blocking
Remark Concerning the Fortran Codes

For portability, during the use of passive target synchronizations (\texttt{MPI\_WIN\_LOCK()}, \texttt{MPI\_WIN\_UNLOCK()}) , it is necessary to allocate the memory window with \texttt{MPI\_ALLOC\_MEM()}. This function accepts as argument pointers of C type (i.e. Fortran CRAY pointers, which do not belong to the Fortran95 standard). In the case where these latter ones are not available, a C program must be used to do the allocation of the memory window...
The RMA concepts of MPI are difficult to implement on non-trivial applications. An in-depth knowledge of the standard is necessary in order not to fall in many traps.

The performances can widely vary from one implementation to another.

The advantage of the RMA concept of MPI resides essentially in the passive target approach. It is only in this case that the use of RMA subroutines is really essential (application requiring that a process access data that belong to a remote process without interruption of the latter...
# 6 - Derived datatypes

## 6.1 Introduction

## 6.2 Contiguous datatypes

## 6.3 Constant stride

## 6.4 Other subroutines

## 6.5 Examples

### 6.5.1 The datatype "matrix row"

### 6.5.2 The datatype "matrix line"

### 6.5.3 The datatype "matrix block"

## 6.6 Homogenous datatypes of variable strides

## 6.7 Subarray Datatype Constructor

## 6.8 Heterogenous datatypes

## 6.9 Subroutines annexes

## 6.10 Conclusion
In the communications, the exchanged data have datatypes: `MPI_INTEGER`, `MPI_REAL`, `MPI_COMPLEX`, etc.

We can create more complex data structures by using subroutines such as `MPI_TYPE_CONTIGUOUS()`, `MPI_TYPE_VECTOR()`, `MPI_TYPE_CREATE_HVECTOR()`.

Each time that we use a datatype, it is mandatory to validate it by using the `MPI_TYPE_COMMIT()` subroutine.

If we wish to reuse the same name to define another derived datatype, we have to free it first with the `MPI_TYPE_FREE()` subroutine.
Figure 24: Hierarchy of the MPI constructors
6 – Derived datatypes
6.2 – Contiguous datatypes

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

```
1. 6. 11. 16. 21. 26.
2. 7. 12. 17. 22. 27.
3. 8. 13. 18. 23. 28.
5. 10. 15. 20. 25. 30.
```

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

Figure 25: MPI_TYPE_CONTIGUOUS subroutine

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

call MPI_TYPE_CONTIGUOUS(count, old_type, new_type, code)
```
6 – Derived datatypes
6.3 – Constant stride

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

![Figure 26: MPI_TYPE_VECTOR subroutine](image)

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

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

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

call MPI_TYPE_CREATE_HVECTOR(count, block_length, stride, old_type, new_type, code)
```
Before using a new derived datatype, it is necessary to validate it with the `MPI_TYPE_COMMIT()` subroutine.

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

call MPI_TYPE_COMMIT(new_type,code)
```

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

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

call MPI_TYPE_FREE(new_type,code)
```
6 – Derived datatypes
6.5 – Examples
6.5.1 – The datatype "matrix row"

```fortran
program column
  use mpi
  implicit none

  integer, parameter :: nb_lines=5,nb_columns=6
  integer, parameter :: tag=100
  real, dimension(nb_lines,nb_columns) :: a
  integer, dimension(MPI_STATUS_SIZE) :: status
  integer :: rank,code,
  type_column

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

  ! Initialization of the matrix on each process
  a(:,:,)=real(rank)

  ! Definition of the type_column datatype
  call MPI_TYPE_CONTIGUOUS(nb_lines,MPI_REAL,type_column,code)

  ! Validation of the type_column datatype
  call MPI_TYPE_COMMIT(type_column,code)
```

! Sending of the first column
if ( rank == 0 ) then
    call MPI_SEND(a(1,1),1,type_column,1,tag,MPI_COMM_WORLD,code)

! Reception in the last column
elseif ( rank == 1 ) then
    call MPI_RECV(a(1,nb_columns),nb_lines,MPI_REAL,0,tag,&
                  MPI_COMM_WORLD,status,code)
end if

! Free the datatype
call MPI_TYPE_FREE(type_column,code)

call MPI_FINALIZE(code)
end program column
program line
   use mpi
   implicit none

   integer, parameter :: nb_lines=5,nb_columns=6
   integer, parameter :: tag=100
   real, dimension(nb_lines,nb_columns) :: a
   integer, dimension(MPI_STATUS_SIZE) :: status
   integer :: rank,code,type_line

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

! Initialization of the matrix on each process
   a(:,:,)=real(rank)

! Definition of the datatype type_line
   call MPI_TYPE_VECTOR(nb_columns,1,nb_lines,MPI_REAL,type_line,code)

! Validation of the datatype type_ligne
   call MPI_TYPE_COMMIT(type_line,code)
! Sending of the second line
if ( rank == 0 ) then
  call MPI_SEND(a(2,1),nb_columns,MPI_REAL,1,tag,MPI_COMM_WORLD,code)

! Reception in the next to last line
elseif ( rank == 1 ) then
  call MPI_RECV(a(nb_lines-1,1),1,type_line,0,tag,&
  MPI_COMM_WORLD,status,code)
end if

! Free the datatype type_ligne
call MPI_TYPE_FREE(type_line,code)

call MPI_FINALIZE(code)

end program line
program block
  use mpi
  implicit none

  integer, parameter :: nb_lines=5,nb_columns=6
  integer, parameter :: tag=100
  integer, parameter :: nb_lines_block=2,nb_columns_block=3
  real, dimension(nb_lines,nb_columns) :: a
  integer, dimension(MPI_STATUS_SIZE) :: status
  integer :: rank,code,type_block

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

  ! Initialization of the matrix on each process
  a(:, :) = real(rank)

  ! Creation of the datatype type_bloc
  call MPI_TYPE_VECTOR(nb_columns_block,nb_lines_block,nb_lines,&
                       MPI_REAL,type_block,code)

  ! Validation of the datatype type_block
  call MPI_TYPE_COMMIT(type_block,code)
! Sending of a block
if ( rank == 0 ) then
  call MPI_SEND(a(1,1),1,type_block,1,tag,MPI_COMM_WORLD,code)

! Reception of the block
elseif ( rank == 1 ) then
  call MPI_RECV(a(nb_lines-1,nb_columns-2),1,type_block,0,tag,&
                 MPI_COMM_WORLD,status,code)
end if

! Freeing of the datatype type_block
call MPI_TYPE_FREE(type_block,code)
call MPI_FINALIZE(code)
end program block
MPI_TYPE_INDEXED() allows to create a data structure composed of a sequence of blocks containing a variable number of elements separated by a variable stride in memory. The latter is given in number of elements.

MPI_TYPE_CREATE_HINDEXED() has the same functionality as MPI_TYPE_INDEXED() except that the strides that separates two data blocks are given in bytes. This subroutine is useful when the 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.
Figure 27: The MPI_TYPE_INDEXED constructor

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

call MPI_TYPE_INDEXED(nb,block_lengths,displacements,old_type,new_type,code)
```
nb=3, blocks_lengths=(2, ), displacements=(0, )

old_type

new_type

Figure 27: The MPI_TYPE_INDEXED constructor

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

integer, intent(out) :: new_type, code

call MPI_TYPE_INDEXED(nb, block_lengths, displacements, old_type, new_type, code)
**Figure 27: The MPI_TYPE_INDEXED constructor**

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

integer, intent(out) :: new_type, code

call MPI_TYPE_INDEXED(nb, block_lengths, displacements, old_type, new_type, code)
```
nb=3, blocks_lengths=(2,1,3), displacements=(0,3,7)

![Diagram of mpi_type_indexed constructor]

**Figure 27: The MPI_TYPE_INDEXED constructor**

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

call MPI_TYPE_INDEXED(nb, block_lengths, displacements, old_type, new_type, code)
```
**Figure 28: The MPI_TYPE_CREATE_HINDEXED constructor**

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

call MPI_TYPE_CREATE_HINDEXED(nb, block_lengths, displacements, old_type, new_type, code)
```
nb=4, blocks_lengths=(2, ), displacements=(2, )

old_type

new_type

Figure 28: The MPI_TYPE_CREATE_HINDEXED constructor

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

integer,intent(out) :: new_type,code

call MPI_TYPE_CREATE_HINDEXED(nb, block_lengths, displacements, old_type, new_type, code)
```
nb=4, blocks_lengths=(2,1, ), displacements=(2,10, )

old_type

new_type

Figure 28: The MPI_TYPE_CREATE_HINDEXED constructor

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

integer,intent(out) :: new_type,code

call MPI_TYPE_CREATE_HINDEXED(nb, block_lengths,displacements,old_type,new_type,code)
```
nb=4, blocks_lengths=(2,1,2, ), displacements=(2,10,14, )

old_type

new_type

Figure 28: The MPI_TYPE_CREATE_HINDEXED constructor

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

integer,intent(out) :: new_type,code

call MPI_TYPE_CREATE_HINDEXED(nb, block_lengths,displacements,
old_type,new_type,code)
```
nb=4, blocks_lengths=(2,1,2,1), displacements=(2,10,14,24)

old_type

new_type

Figure 28: The MPI_TYPE_CREATE_HINDEXED constructor

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

call MPI_TYPE_CREATE_HINDEXED(nb, block_lengths, displacements, old_type, new_type, code)
```
In the following example, each one of the two processes:

1. initializes its matrix (positive growing numbers on the process 0 and negative decreasing numbers on the 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 and receives a triangular matrix which it stores in the place of the matrix which it has sent via the **MPI_SENDRECV_REPLACE()** subroutine;
4. frees its resources and exits MPI.
Table: Homogenous datatypes of variable strides

**Before**

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

**Process 0**

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>-2</th>
<th>-3</th>
<th>-4</th>
<th>-5</th>
<th>-6</th>
<th>-7</th>
<th>-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-9</td>
<td>-17</td>
<td>-25</td>
<td>-33</td>
<td>-41</td>
<td>-49</td>
<td>-57</td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>-10</td>
<td>-18</td>
<td>-26</td>
<td>-34</td>
<td>-42</td>
<td>-50</td>
<td>-58</td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>-11</td>
<td>-19</td>
<td>-27</td>
<td>-35</td>
<td>-43</td>
<td>-51</td>
<td>-59</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>-12</td>
<td>-20</td>
<td>-28</td>
<td>-36</td>
<td>-44</td>
<td>-52</td>
<td>-60</td>
<td></td>
</tr>
<tr>
<td>-5</td>
<td>-13</td>
<td>-21</td>
<td>-29</td>
<td>-37</td>
<td>-45</td>
<td>-53</td>
<td>-61</td>
<td></td>
</tr>
<tr>
<td>-6</td>
<td>-14</td>
<td>-22</td>
<td>-30</td>
<td>-38</td>
<td>-46</td>
<td>-54</td>
<td>-62</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td>-16</td>
<td>-24</td>
<td>-32</td>
<td>-40</td>
<td>-48</td>
<td>-56</td>
<td>-64</td>
<td></td>
</tr>
</tbody>
</table>

**Process 1**

**Figure 29: Exchange between the two processes**
### Before

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>17</td>
<td>25</td>
<td>33</td>
<td>41</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>18</td>
<td>26</td>
<td>34</td>
<td>42</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>19</td>
<td>27</td>
<td>35</td>
<td>43</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>20</td>
<td>28</td>
<td>36</td>
<td>44</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>21</td>
<td>29</td>
<td>37</td>
<td>45</td>
<td>53</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>22</td>
<td>30</td>
<td>38</td>
<td>46</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>23</td>
<td>31</td>
<td>39</td>
<td>47</td>
<td>55</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
</tr>
</tbody>
</table>

### After

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-2</td>
<td>-3</td>
<td>-5</td>
<td>-8</td>
<td>-14</td>
<td>-22</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>-4</td>
<td>-6</td>
<td>-11</td>
<td>-15</td>
<td>-23</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>19</td>
<td>-7</td>
<td>-12</td>
<td>-16</td>
<td>-24</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>20</td>
<td>28</td>
<td>-13</td>
<td>-20</td>
<td>-29</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>21</td>
<td>29</td>
<td>37</td>
<td>-21</td>
<td>-30</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>22</td>
<td>30</td>
<td>38</td>
<td>46</td>
<td>-31</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>23</td>
<td>31</td>
<td>39</td>
<td>47</td>
<td>55</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
</tr>
</tbody>
</table>

### Process 0

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-9</td>
<td>-17</td>
<td>-25</td>
<td>-33</td>
<td>-41</td>
<td>-49</td>
</tr>
<tr>
<td>-2</td>
<td>-10</td>
<td>-18</td>
<td>-26</td>
<td>-34</td>
<td>-42</td>
<td>-50</td>
</tr>
<tr>
<td>-3</td>
<td>-11</td>
<td>-19</td>
<td>-27</td>
<td>-35</td>
<td>-43</td>
<td>-51</td>
</tr>
<tr>
<td>-4</td>
<td>-12</td>
<td>-20</td>
<td>-28</td>
<td>-36</td>
<td>-44</td>
<td>-52</td>
</tr>
<tr>
<td>-5</td>
<td>-13</td>
<td>-21</td>
<td>-29</td>
<td>-37</td>
<td>-45</td>
<td>-53</td>
</tr>
<tr>
<td>-6</td>
<td>-14</td>
<td>-22</td>
<td>-30</td>
<td>-38</td>
<td>-46</td>
<td>-54</td>
</tr>
<tr>
<td>-8</td>
<td>-16</td>
<td>-24</td>
<td>-32</td>
<td>-40</td>
<td>-48</td>
<td>-56</td>
</tr>
</tbody>
</table>

### Process 1

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-9</td>
<td>-17</td>
<td>-25</td>
<td>-33</td>
<td>-41</td>
<td>-49</td>
</tr>
<tr>
<td>-2</td>
<td>-10</td>
<td>-18</td>
<td>-26</td>
<td>-34</td>
<td>-42</td>
<td>-50</td>
</tr>
<tr>
<td>-3</td>
<td>-11</td>
<td>-19</td>
<td>-27</td>
<td>-35</td>
<td>-43</td>
<td>-51</td>
</tr>
<tr>
<td>-4</td>
<td>-12</td>
<td>-20</td>
<td>-28</td>
<td>-36</td>
<td>-44</td>
<td>-52</td>
</tr>
<tr>
<td>-5</td>
<td>-13</td>
<td>-21</td>
<td>-29</td>
<td>-37</td>
<td>-45</td>
<td>-53</td>
</tr>
<tr>
<td>-6</td>
<td>-14</td>
<td>-22</td>
<td>-30</td>
<td>-38</td>
<td>-46</td>
<td>-54</td>
</tr>
<tr>
<td>-8</td>
<td>-16</td>
<td>-24</td>
<td>-32</td>
<td>-40</td>
<td>-48</td>
<td>-56</td>
</tr>
</tbody>
</table>

**Figure 29:** Exchange between the two processes
program triangle
  use mpi
  implicit none
  integer, parameter :: n=8, tag=100
  real, dimension(n,n) :: a
  integer, dimension(MPI_STATUS_SIZE) :: status
  integer :: i, code
  integer :: rank, type_triangle
  integer, dimension(n) :: block_lengths, displacements

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

  ! Initialization of the matrix on each process
  a(:, :) = reshape((/ (sign(i, -rank), i=1, n*n) /), (/n, n/))

  ! Creation of the triangular matrix datatype sup for the process 0
  ! and of the inferior triangular matrix datatype for the process 1
  if (rank == 0) then
    block_lengths(:) = (/ (i-1, i=1, n) /)
    displacements(:) = (/ (n*(i-1), i=1, n) /)
  else
    block_lengths(:) = (/ (n-i, i=1, n) /)
    displacements(:) = (/ (n*(i-1)+i, i=1, n) /)
  endif

  call MPI_TYPE_INDEXED(n, block_lengths, displacements, MPI_REAL, type_triangle, code)
call MPI_TYPE_COMMIT(type_triangle, code)

  ! Permutation of the inferior and superior triangular matrices
  call MPI_SENDRECV_REPLACE(a, 1, type_triangle, mod(rank+1, 2), tag, mod(rank+1, 2), &
                             tag, MPI_COMM_WORLD, status, code)

  ! Freeing of the triangle datatype
  call MPI_TYPE_FREE(type_triangle, code)
call MPI_FINALIZE(code)
end program triangle
The **MPI_TYPE_CREATE_SUBARRAY()** subroutine allows to create a subarray from an array.

```plaintext
integer,intent(in) :: nb_dims
integer,dimension(ndims),intent(in) :: shape_array,shape_sub_array,coord_start
integer,intent(in) :: order,old_type
integer,intent(out) :: new_type,code
call MPI_TYPE_CREATE_SUBARRAY(nb_dims,shape_array,shape_sub_array,coord_start,
order,old_type,new_type,code)
```
Reminder of the vocabulary relative to the arrays in Fortran 95

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

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

☞ nb_dims : rank of the array
☞ shape_array : shape of the array from which a subarray will be extracted
☞ shape_sub_array : shape of the subarray
☞ coord_start : start coordinates if the indices of the array start at 0. For example, if we want that the start coordinates of the subarray be array(2,3), we must have coord_start(:)=(/ 1,2 /)
☞ order : storage order of elements
  1. MPI_ORDER_FORTRAN for the ordering used by Fortran arrays (column-major order)
  2. MPI_ORDER_C for the ordering used by C arrays (row-major order)
### Figure 30: Exchanges between the two processes

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process 0</strong></td>
<td><strong>Process 0</strong></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td><strong>Process 1</strong></td>
<td><strong>Process 1</strong></td>
</tr>
<tr>
<td>-1</td>
<td>-5</td>
</tr>
<tr>
<td>-2</td>
<td>-6</td>
</tr>
<tr>
<td>-3</td>
<td>-7</td>
</tr>
<tr>
<td>-4</td>
<td>-8</td>
</tr>
<tr>
<td>-7</td>
<td>-11</td>
</tr>
<tr>
<td>-8</td>
<td>-12</td>
</tr>
<tr>
<td>-1</td>
<td>-5</td>
</tr>
<tr>
<td>-2</td>
<td>-6</td>
</tr>
<tr>
<td>-3</td>
<td>2</td>
</tr>
<tr>
<td>-4</td>
<td>3</td>
</tr>
</tbody>
</table>
program subarray
  use mpi
  implicit none

  integer,parameter :: nb_lines=4,nb_columns=3,&
                      tag=1000,nb_dims=2
  integer :: code,rank,type_subarray,i
  integer,dimension(nb_lines,nb_columns) :: tab
  integer,dimension(nb_dims) :: shape_array,shape_subarray,coord_start
  integer,dimension(MPI_STATUS_SIZE) :: status

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

! Initialization of the tab array on each process
  tab(:,:) = reshape( (/ (sign(i,-rank),i=1,nb_lines*nb_columns) /) , &
                     (/ nb_lines,nb_columns /) )
!Shape of the tab array from which a subarray will be extracted
shape_tab(:) = shape(tab)
!The F95 shape function gives the profile of array put in argument.
!ATTENTION, if the concerned array has not been allocated on all the processes, 
!it will become necessary to explicitly put the shape of the array in order to 
!become known on all the processes, shape_array(:) = (/ nb_lines,nb_columns) /

!Shape of the subarray
shape_subarray(:) = (/ 2,2 /)

!Start coordinates of the subarray
!For the process 0 we start from the tab(2,1) element
!For the process 1 we start from the tab(3,2) element
coord_start(:) = (/ rank+1,rank /)

!Creation of the type_subarray derived datatype
call MPI_TYPE_CREATE_SUBARRAY(nb_dims,shape_array,shape_subarray,coord_start,&
   MPI_ORDER_FORTRAN,MPI_INTEGER,type_subarray,code)
call MPI_TYPE_COMMIT(type_subarray,code)

!Exchange of the subarrays
call MPI_SENDRECV_REPLACE(tab,1,type_subarray,mod(rank+1,2),tag,&
   mod(rank+1,2),tag,MPI_COMM_WORLD,status,code)
call MPI_TYPE_FREE(type_subarray,code)
call MPI_FINALIZE(code)
end program subarray
The MPI_TYPE_CREATE_STRUCT() subroutine is the constructor of more general datatypes.

It has the same functionalities as MPI_TYPE_INDEXED() but it allows in addition the replication of different datatype blocks.

The MPI_TYPE_CREATE_STRUCT() parameters are the same as the ones of MPI_TYPE_INDEXED() with in addition:

- the old_types field is now a vector of MPI datatypes;
- taking into account the heterogeneity of data and their alignment in memory, the calculation of the displacement between two elements is in the difference of their addresses;
- MPI, via MPI_GET_ADDRESS(), provides a portable subroutine which allows to return the address of a variable.
integer, intent(in) :: nb
integer, intent(in), dimension(nb) :: blocks_lengths
integer(kind=MPI_ADDRESS_KIND), intent(in), dimension(nb) :: displacements
integer, intent(in), dimension(nb) :: old_types

integer, intent(out) :: new_type, code

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

不同类型

old_types

<table>
<thead>
<tr>
<th>type 1</th>
<th>type 2</th>
<th>type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

new_type

Figure 31: The MPI_TYPE_CREATE_STRUCT constructor

integer, intent(in) :: variable
integer(kind=MPI_ADDRESS_KIND), intent(out) :: address_variable
integer, intent(out) :: code

call MPI_GET_ADDRESS(variable, address_variable, code)
\[ \text{nb}=5, \text{blocks\_lengths}=(3,\quad\quad), \text{displacements}=(0,\quad\quad), \]
\[ \text{old\_types}=(\text{type}1,\quad\quad) \]

\begin{tabular}{ccc}
\text{type 1} & \text{type 2} & \text{type 3} \\
\end{tabular}

\begin{tabular}{c}
\text{old\_types} \\
\end{tabular}

\begin{tabular}{c}
\text{new\_type} \\
\end{tabular}

\[ \text{Figure 31: The MPI\_TYPE\_CREATE\_STRUCT constructor} \]

```
integer,intent(in) :: nb
integer,intent(in),dimension(nb) :: blocks_lengths
integer(kind=MPI\_ADDRESS\_KIND),intent(in),dimension(nb) :: displacements
integer,intent(in),dimension(nb) :: old_types

integer, intent(out) :: new_type,code

\text{call MPI\_TYPE\_CREATE\_STRUCT(nb,blocks\_lengths,displacements,}
\text{old\_types,new\_type,code)}
```

```
<type>,intent(in) :: variable
integer(kind=MPI\_ADDRESS\_KIND),intent(out) :: address\_variable
integer,intent(out) :: code

\text{call MPI\_GET\_ADDRESS(variable,address\_variable,code)}
```
nb=5, blocks_lengths=(3,1, ), displacements=(0,7, ),
old_types=(type1,type2, )

old_types

type 1  type 2  type 3

new_type

Figure 31: The MPI_TYPE_CREATE_STRUCT constructor

integer,intent(in) :: nb
integer,intent(in),dimension(nb) :: blocks_lengths
integer(kind=MPI_ADDRESS_KIND),intent(in),dimension(nb) :: displacements
integer,intent(in),dimension(nb) :: old_types
integer, intent(out) :: new_type,code
call MPI_TYPE_CREATE_STRUCT(nb,blocks_lengths,displacements,
old_types,new_type,code)

<type>,intent(in) :: variable
integer(kind=MPI_ADDRESS_KIND),intent(out) :: address_variable
integer,intent(out) :: code
call MPI_GET_ADDRESS(variable,address_variable,code)
nb=5, blocks_lengths=(3,1,5), displacements=(0,7,11),
old_types=(type1,type2,type3,)

<table>
<thead>
<tr>
<th>old_types</th>
<th>new_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>type 1</td>
<td>type 2</td>
</tr>
</tbody>
</table>

Figure 31: The MPI_TYPE_CREATE_STRUCT constructor

```fortran
integer,intent(in) :: nb
integer,intent(in),dimension(nb) :: blocks_lengths
integer(kind=MPI_ADDRESS_KIND),intent(in),dimension(nb) :: displacements
integer,intent(in),dimension(nb) :: old_types

integer, intent(out) :: new_type,code

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

```fortran
<type>,intent(in) :: variable
integer(kind=MPI_ADDRESS_KIND),intent(out) :: address_variable
integer,intent(out) :: code

call MPI_GET_ADDRESS(variable,address_variable,code)
```
\begin{align*}
nb &= 5, \quad \text{blocks\_lengths} = (3, 1, 5, 1, ), \\
&\quad \text{displacements} = (0, 7, 11, 21, ), \\
&\quad \text{old\_types} = (\text{type1, type2, type3, type1, })
\end{align*}

\begin{table}[h]
\begin{tabular}{ccc}
  type 1 & type 2 & type 3 \\
  \includegraphics[width=0.2\textwidth]{old_types} & \includegraphics[width=0.2\textwidth]{new_type} & \includegraphics[width=0.2\textwidth]{new_type}
\end{tabular}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure}
\caption{The MPI\_TYPE\_CREATE\_STRUCT constructor}
\end{figure}

\begin{verbatim}
integer,intent(in) :: nb
integer,intent(in),dimension(nb) :: blocks_lengths
integer(kind=MPI\_ADDRESS\_KIND),intent(in),dimension(nb) :: displacements
integer,intent(in),dimension(nb) :: old_types

integer, intent(out) :: new_type,code

call MPI\_TYPE\_CREATE\_STRUCT(nb,blocks_lengths,displacements,
old_types,new_type,code)
\end{verbatim}

\begin{verbatim}
<type>,intent(in) :: variable
integer(kind=MPI\_ADDRESS\_KIND),intent(out) :: address_variable
integer,intent(out) :: code

call MPI\_GET\_ADDRESS(variable,address_variable,code)
\end{verbatim}
nb=5, blocks_lengths=(3,1,5,1,1), displacements=(0,7,11,21,26),
old_types=(type1,type2,type3,type1,type3)

<table>
<thead>
<tr>
<th>type 1</th>
<th>type 2</th>
<th>type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>old_types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>new_type</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 31: The MPI_TYPE_CREATE_STRUCT constructor

integer, intent(in) :: nb
integer, intent(in), dimension(nb) :: blocks_lengths
integer(kind=MPI_ADDRESS_KIND), intent(in), dimension(nb) :: displacements
integer, intent(in), dimension(nb) :: old_types

integer, intent(out) :: new_type, code

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

<type>, intent(in) :: variable
integer(kind=MPI_ADDRESS_KIND), intent(out) :: address_variable
integer, intent(out) :: code

call MPI_GET_ADDRESS(variable, address_variable, code)
program Interaction_Particles
  use mpi
  implicit none

  integer, parameter :: n=1000,tag=100
  integer, dimension(MPI_STATUS_SIZE) :: status
  integer :: rank,code,type_particle,i
  integer, dimension(4) :: types,blocks_lengths
  integer(kind=MPI_ADDRESS_KIND), dimension(4) :: displacements,addresses

  type Particule
    character(len=5) :: category
    integer :: mass
    real, dimension(3) :: coords
    logical :: class
  end type Particule

  type(Particule), dimension(n) :: p,temp_p

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

  ! Construction of the datatype
  types = (/MPI_CHARACTER,MPI_INTEGER,MPI_REAL,MPI_LOGICAL/) 
  blocks_lengths= (/5,1,3,1/)
call MPI_GET_ADDRESS(p(1)%category,addresses(1),code)
call MPI_GET_ADDRESS(p(1)%mass,addresses(2),code)
call MPI_GET_ADDRESS(p(1)%coords,addresses(3),code)
call MPI_GET_ADDRESS(p(1)%class,addresses(4),code)

! Calculation of displacements relative to the start address
do i=1,4
    displacements(i)=addresses(i) - addresses(1)
end do
call MPI_TYPE_CREATE_STRUCT(4,blocks_lengths,displacements,types,type_particle, &
    code)

! Validation of the structured datatype
call MPI_TYPE_COMMIT(type_particle,code)

! Initialization of particles for each process
....

! Sending of particles from 0 towards 1
if (rank == 0) then
call MPI_SEND(p(1)%category,n,type_particle,1,tag,MPI_COMM_WORLD,code)
else
call MPI_RECV(temp_p(1)%category,n,type_particle,0,tag,MPI_COMM_WORLD, &
    status,code)
endif

! Freeing of the datatype
call MPI_TYPE_FREE(type_particle,code)
call MPI_FINALIZE(code)
end program Interaction_Particles
The total size of a datatype: \texttt{MPI\_TYPE\_SIZE()}

\begin{verbatim}
integer, intent(in) :: type_data
integer, intent(out) :: size, code

call MPI\_TYPE\_SIZE(type_data, size, code)
\end{verbatim}

The extent as well as the inferior limit of a derived datatype, by taking into account eventual memory alignments: \texttt{MPI\_TYPE\_GET\_EXTENT()}

\begin{verbatim}
integer, intent(in) :: type_derived
integer(kind=\texttt{MPI\_ADDRESS\_KIND}), intent(out):: bound_inf_align, size_align
integer, intent(out) :: code

call MPI\_TYPE\_GET\_EXTENT(type_derived, bound_inf_align, size_align, code)
\end{verbatim}

We can modify the inferior bound of a derived datatype and its extent in order to create a new adapted datatype of the precedent

\begin{verbatim}
integer, intent(in) :: old_type
integer(kind=\texttt{MPI\_ADDRESS\_KIND}), intent(in) :: new_bound_inf, new_size
integer, intent(out) :: new_type, code

call MPI\_TYPE\_CREATE\_RESIZED(old_type, new_bound_inf, new_size, new_type, code)
\end{verbatim}
PROGRAM half_line
USE mpi
IMPLICIT NONE
INTEGER,PARAMETER :: nb_lines=5,nb_columns=6,&
   half_line=nb_columns/2,tag=1000
INTEGER,DIMENSION(nb_lines,nb_columns) :: A
INTEGER :: typeHalfLine,typeHalfLine2
INTEGER :: code,size_integer,rank,i
INTEGER(kind=MPI_ADDRESS_KIND) :: boundInf=0, sizeDisplacement
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

CALL MPI_INIT(code)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
!Initialization of the A matrix on each process
A(:,:,)=RESHAPE( (/ (SIGN(i,-rank),i=1,nb_lines*nb_columns) /), &
   (/ nb_lines,nb_columns /))

!Construction of the derived datatype typeHalfLine
CALL MPI_TYPE_VECTOR(half_line,1,nb_lines,MPI_INTEGER,typeHalfLine,code)
!Know the size of the datatype MPI_INTEGER
CALL MPI_TYPE_SIZE(MPI_INTEGER, size_integer, code)
sizeDisplacement = size_integer
CALL MPI_TYPE_CREATE_RESIZED(typeHalfLine,boundInf,sizeDisplacement,&
   typeHalfLine2,code)
!Validation of the datatype typeHalfLine2
CALL MPI_TYPE_COMMIT(typeHalfLine2,code)

IF (rank == 0) THEN
  !Sending of the A matrix to the process 1 with the derived datatype typeHalfLine2
  CALL MPI_SEND(A(1,1), 2, typeHalfLine2, 1, tag, &
                MPI_COMM_WORLD, code)
ELSE
  !Reception for the process 1 in the A matrix
  CALL MPI_RECV(A(1,nb_columns-1), 6, MPI_INTEGER, 0, tag,&
                 MPI_COMM_WORLD,status, code)

  PRINT *,’A matrix on the process 1’
  DO i=1,nb_lines
      PRINT *,A(i,:)
  END DO
END IF

CALL MPI_FINALIZE(code)
END PROGRAM half_line

> mpiexec -n 4 half_ligne
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
The MPI derived datatypes are powerful portable mechanisms to describe data. They allow, when they are combined with subroutines like `MPI_SENDRECV()`, to simplify 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 the problems of domain decomposition with regular or irregular meshes.
7 Optimizations

7.1 Introduction ................................................................. 195
7.2 Point-to-Point Send Modes ............................................. 196
  7.2.1 Key Terms ............................................................... 197
  7.2.2 Synchronous Sends .................................................. 198
  7.2.3 Buffered Sends ....................................................... 200
  7.2.4 Standard Sends ....................................................... 204
  7.2.5 Ready Sends ............................................................ 205
  7.2.6 Performances of Send Modes .................................... 206
7.3 Computation-Communication Overlap .............................. 207

8 Communicators

9 MPI-IO

10 Conclusion
Optimization of MPI communications must be a primary concern when their part becomes quite important compared to the calculation one.

The optimization of communications, beyond the choice of an algorithm as efficient as possible, can be accomplished at many levels including, for example:

- by selecting the most suitable communication mode;
- by overlapping communications with computations.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Blocking</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard send</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>Synchronous send</td>
<td>MPI_Ssend</td>
<td>MPI_Issend</td>
</tr>
<tr>
<td>Buffered send</td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td>Ready send</td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
</tr>
<tr>
<td>Receive</td>
<td>MPI_Recv</td>
<td>MPI_Irecv</td>
</tr>
</tbody>
</table>
It is important to understand well the definition of some MPI terms.

- **Blocking call**: a call is blocking if the memory space used for the communication can be reused immediately after the exit of the call. The data that have been or will be sent are the data that were in this space at the moment of the call. If it is a receive, the data must have already been received in this space (if the return code is `MPI_SUCCESS`).

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

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

- **Buffered send**: a buffered send implies the copying of data in an intermediate memory space. There is then no coupling between the two processes of communication. So the return of this type of send does not mean that the receive occurred.
A synchronous send is made by calling the `MPI_Ssend` or `MPI_Issend` subroutine.

**Rendezvous Protocol**

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

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

Disadvantages

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

A buffered send is made by calling the `MPI_Bsend` or `MPI_Ibsend` subroutine. The buffers have to be managed manually (with calls to `MPI_Attach` and `MPI_Detach`). They have to be allocated by taking into account the header size of messages (by adding the constant `MPI_BSEND_OVERHEAD` for each message instance).
This approach is the one generally used for the `MPI_Bsend` or `MPI_Ibsend`. In this approach, the buffer is on the sender side and is managed explicitly by the application. A buffer managed by MPI can exist on the receiver side. Many variants are possible. The return receipt is optional.
The eager protocol is often used for standard sends of small-size messages. It can also be used for sends with `MPI_Bsend` with small messages (implementation-dependent) and by bypassing the user buffer on the sender side. In this approach, the buffer is on the receiver side. The return receipt is optional.
### Advantages
- No need to wait for the receiver (copying in a buffer)
- No risks of deadlocks

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

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

Advantages

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

Disadvantages

- Little control over the really used mode (often accessible via environment variables)
- Risk of deadlock according to the actual mode
- Behavior that can vary according to the architecture and the problem size
7 – Optimizations
7.2 – Point-to-Point Send Modes
7.2.5 – Ready Sends

Ready Sends

A ready send is made by calling the `MPI_Rsend` or `MPI_Irsend` subroutine.

Warning: it is obligatory to make these calls only when the receive is already posted. Their use is not recommended.

Advantages

- Slightly more efficient than the synchronous mode because the protocol of synchronization can be simplified

Disadvantages

- Errors if the receiver is not ready during the send
Ping-pong extranode balanced

![Graph showing bandwidth (MiB/s) vs. message size (bytes) for different send modes of Vargas and Babel.]
The overlap of communications by computations is a method which allows to execute communications operations in background while the program continues to operate.

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

**Partial overlap**

- Process 0
  - Isend
  - Sending
  - Wait
  - Sending in background
  - finished?
  - yes

**Full overlap**

- Process 0
  - Isend
  - Sending in background
  - Sending
  - Wait
  - finished?
  - yes

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

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

The message send is made in two steps:

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

The communications overlap with all the operations that occur between these two steps. The access to data being in receive is not permitted before the end of the MPI_Wait (the access to data being in send is also not permitted for the MPI implementations previous to the 2.2 version).
integer, dimension(2) :: req

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

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

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

  ! Compute the exterior domain
  call compute_domaine_exterieur(data_int,data_bound,data_ext)
end do
Overlap Level on Different Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue Gene/P DCMF_INTERRUPT=0</td>
<td>34%</td>
</tr>
<tr>
<td>Blue Gene/P DCMF_INTERRUPT=1</td>
<td>100%</td>
</tr>
<tr>
<td>Power6 InfiniBand</td>
<td>38%</td>
</tr>
<tr>
<td>NEC SX-8</td>
<td>10%</td>
</tr>
<tr>
<td>CURIE</td>
<td>0%</td>
</tr>
</tbody>
</table>

Measures done by overlapping a computational kernel with a communication kernel which have the same execution time and by using different schemes of communications (intra/extra-nodes, by pairs, random processes ...). According to the scheme of communication, the results can be totally different.

An overlap of 0% means that the total time of execution is 2x the time of a computational (or communication) kernel.
An overlap of 100% means that the total time of execution is 1x the time of a computational (or communication) kernel.
1 Introduction
2 Environment
3 Point-to-point Communications
4 Collective communications
5 One-sided Communication
6 Derived datatypes
7 Optimizations
8 Communicators
  8.1 Introduction ................................................................. 214
  8.2 Default Communicator ..................................................... 215
  8.3 Example ........................................................................ 216
  8.4 Groups and Communicators ............................................. 218
  8.5 Communicator Resulting from Another .............................. 220
  8.6 Topologies ................................................................. 224
    8.6.1 Cartesian topologies ................................................. 225
    8.6.2 Subdividing a cartesian topology ................................. 240
9 MPI-IO
10 Conclusion
Communicators usage consists of partitioning a group of processes in order to create subgroups on which we can carry out operations such as collective or point-to-point communications. Each created subgroup will have its own communication space.

Figure 32: Communicator partitioning
It is the story of the chicken and the egg...

☞ We can only create a communicator from another communicator.

☞ Luckily, this has been resolved by assuming that the chicken existed first. In fact, a communicator is provided by default, whose the `MPI_COMM_WORLD` identifier is an integer value defined in the header files.

☞ This `MPI_COMM_WORLD` initial communicator is created for all the execution period of the program from a call to the `MPI_INIT()` subroutine.

☞ This communicator can only be destroyed via the calling of `MPI_FINALIZE()`.

☞ By default, it sets therefore the range of collective and point-to-point communications to all the processes of the application.
In the following example, we will:
☞ put together on one hand the even-ranked processes and on the other hand the odd-ranked processes;
☞ broadcast a collective message only to even-ranked processes and another only to odd-ranked processes.

```
$ mpirun -np 8 CommPairImpair
```

call MPI_INIT(...)

call MPI_COMM_SPLIT(...)

call MPI_BCAST(...)

call MPI_COMM_FREE(...)

Figure 33: Communicator creation/destruction
What to do in order for the rank 2 process to broadcast this message to the even-ranked process subgroup, for example?

Looping on `send/recv` can be very detrimental especially if the number of processes is high. Also, a test would be compulsory in the loop in order to know if the rank of the process to which the rank 2 process must send the message is even or odd.

The solution is to create a communicator containing these processes in a way that the rank 2 process broadcasts the message only to them.

![Diagram](MPI_COMM_WORLD)

Figure 34: A new communicator
A communicator consists:

1. of a group, which is an ordered group of processes;
2. of a communication context made at the calling of the communicator construction subroutine, which allows to define the communication space.

The communication contexts are managed by MPI (the programmer has no action on them: it is an opaque attribute).

In practice, in order to build a communicator, there are two ways to do this:

1. through a group of processes;
2. directly from another communicator.
In the MPI library, many subroutines exist in order to build communicators: `MPI_CART_CREATE()`, `MPI_CART_SUB()`, `MPI_COMM_CREATE()`, `MPI_COMM_DUP()`, `MPI_COMM_SPLIT()`.

The communicators’ constructors are collective calls.

The communicators that the programmer creates can be freed by using the `MPI_COMM_FREE()` subroutine.
The direct use of groups presents in this case many disadvantages, because it requires to:

☞ name differently the two communicators (for example `comm_even` and `comm_odd`);
☞ go through the groups to build these two communicators;
☞ leave the ordering of the process rank to MPI in these two communicators;
☞ make conditional tests when calling the `MPI_BCAST()` subroutine:

```fortran
if (comm_even /= MPI_COMM_NULL) then
    ! Broadcast of the message only to even-ranked processes
    call MPI_BCAST(a,m,MPI_REAL,rank_in_even,comm_even,code)
elseif (comm_odd /= MPI_COMM_NULL) then
    ! Broadcast of the message only to odd-ranked processes
    call MPI_BCAST(a,m,MPI_REAL,rank_in_odd,comm_odd,code)
end if
```
The **MPI_COMM_SPLIT()** subroutine allows to partition a given communicator in as many communicators as we want...

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

<table>
<thead>
<tr>
<th>process</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank_world</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>color</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>key</td>
<td>2</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>rank_new_com</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 35: Construction of communicators with **MPI_COMM_SPLIT()**

A process that is assigned a color equal to the **MPI_UNDEFINED** value will belong only to its initial communicator.
Let’s see how to proceed in order to build the communicator which will subdivide the communication space between odd-ranked and even-ranked processes via the `MPI_COMM_SPLIT()` constructor.

<table>
<thead>
<tr>
<th>process</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank_world</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>color</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>key</td>
<td>0</td>
<td>1</td>
<td>-1</td>
<td>3</td>
<td>4</td>
<td>-1</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>rank_even_odd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

In pratice, this is set up very simply...
program EvenOdd
  use mpi
  implicit none

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

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

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

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

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

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

  ! Destruction of the communicators
  call MPI_COMM_FREE(CommEvenOdd,code)
call MPI_FINALIZE(code)
end program EvenOdd
In most applications, especially in domain decomposition methods where we match the calculation domain to the grid of processes, it is interesting to be able to arrange the processes according to a regular topology.

MPI allows to define cartesian or graph virtual topologies.

- **Cartesian topologies:**
  - each process is defined in a grid;
  - the grid can be periodic or not;
  - the processes are identified by their coordinates in the grid.

- **Graph Topologies:**
  - generalization to more complex topologies.
A cartesian topology is defined when a group of processes belonging to a given communicator `comm_old` calls the `MPI_CART_CREATE()` subroutine.

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

integer, intent(out) :: comm_new, code

call MPI_CART_CREATE(comm_old, ndims,dims,periods,reorganization,comm_new,code)
```
Example on a grid having 4 domains along x and 2 along y, periodic in y.

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

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

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

If `reorganization = .false.` then the rank of the processes in the new communicator (comm_2D) is the same as in the old communicator (MPI_COMM_WORLD). If `reorganization = .true.`, the MPI implementation chooses the order of the processes.
Figure 37: 2D periodic cartesian topology in y
Example on a 3D grid having 4 domains along x, 2 along y and 2 along z, non-periodic.

```fortran
use mpi
integer :: comm_3D, code
integer, parameter :: ndims = 3
integer, dimension(ndims) :: dims
logical, dimension(ndims) :: periods
logical :: reorganization

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

call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, dims, periods, reorganization, comm_3D, code)
```
Figure 38: None periodic cartesian topology 3D
The **MPI_DIMS_CREATE()** subroutine returns the number of processes in each dimension of the grid according to the total number of processes.

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

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

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

<table>
<thead>
<tr>
<th>dims in entry</th>
<th>call MPI_DIMS_CREATE</th>
<th>dims en exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(8,2,dims,code)</td>
<td>(4,2)</td>
</tr>
<tr>
<td>(0,0,0)</td>
<td>(16,3,dims,code)</td>
<td>(4,2,2)</td>
</tr>
<tr>
<td>(0,4,0)</td>
<td>(16,3,dims,code)</td>
<td>(2,4,2)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>(16,3,dims,code)</td>
<td>error</td>
</tr>
</tbody>
</table>
In a cartesian topology, the `MPI_CART_RANK()` subroutine returns the rank of the associated process to the coordinates in the grid.

```fortran
integer, intent(in) :: comm_new
integer, dimension(ndims),intent(in) :: coords
integer, intent(out) :: rank, code

call MPI_CART_RANK(comm_new,coords,rank,code)
```
Figure 39: Periodic cartesian topology 2D in y

```fortran
coords(1)=dims(1)-1
do i=0,dims(2)-1
    coords(2) = i
    call MPI_CART_RANK(comm_2D,coords,rank(i),code)
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.
In a cartesian topology, the `MPI_CART_COORDS()` subroutine returns the coordinates of a process of a given rank in the grid.

```fortran
integer, intent(in) :: comm_new, rank, ndims
integer, dimension(ndims),intent(out) :: coords
integer, intent(out) :: code

call MPI_CART_COORDS(comm_new, rank, ndims, coords, code)
```
if (mod(rank,2) == 0) then
    call MPI_CART_COORDS(comm_2D,rank,2,coords,code)
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)

Figure 40: Periodic cartesian ropology 2D in y
In a cartesian topology, a process that calls the `MPI_CART_SHIFT()` subroutine can get the rank of its neighboring processes in a given direction.

```fortran
integer, intent(in) :: comm_new, direction, step
integer, intent(out) :: rank_previous, rank_next
integer, intent(out) :: code

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

- The `direction` parameter corresponds to the displacement axis (xyz).
- The `step` parameter corresponds to the displacement step.
Figure 41: Call of the MPI_CART_SHIFT() subroutine

Call MPI_CART_SHIFT(comm_2D, 0, 1, rank_left, rank_right, code)

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

Call MPI_CART_SHIFT(comm_2D, 1, 1, rank_low, rank_high, code)

For the process 2, rank_low=3, rank_high=3
Figure 42: Call of the MPI_CART_SHIFT() subroutine

```fortran
call MPI_CART_SHIFT(comm_3D, 0, 1, rank_left, rank_right, code)
```
For the process 0, \( rank_{left} = -1 \), \( rank_{right} = 4 \)

```fortran
call MPI_CART_SHIFT(comm_3D, 1, 1, rank_low, rank_high, code)
```
For the process 0, \( rank_{low} = -1 \), \( rank_{high} = 2 \)

```fortran
call MPI_CART_SHIFT(comm_3D, 2, 1, rank_ahead, rank_before, code)
```
For the process 0, \( rank_{ahead} = -1 \), \( rank_{before} = 1 \)
Program Example:

```fortran
program decomposition
  use mpi
  implicit none

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

  call MPI_INIT(code)

  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)

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

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

INSTITUT DU DÉVELOPPEMENT ET DES RÉSSOURCES EN INFORMATIQUE SCIENTIFIQUE

MPI – Version 4.3 – May 2013

! 2D y-periodic grid creation
periods(1) = .false.
periods(2) = .true.
reorganization = .false.

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

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

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

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

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

call MPI_FINALIZE(code)
end program decomposition
The goal is, by example, to degenerate a 2D or 3D cartesian topology in, respectively, a 1D or 2D cartesian topology.

For MPI, degenerating a 2D (or 3D) cartesian topology creates as many communicators as there are rows or columns (resp. planes) in the initial cartesian grid.

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.
Figure 43: Two examples of data distribution in a degenerated 2D 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.

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

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

![Diagram](image)

**Figure 44:** Initial representation of a V array in the 2D grid and final representation after the distribution of that one on the degenerated 2D grid
program CommCartSub
  use mpi
  implicit none

  integer :: Comm2D,Comm1D,rank,code
  integer,parameter :: NDim2D=2
  integer,dimension(NDim2D) :: Dim2D,Coord2D
  logical,dimension(NDim2D) :: Period,remain_dims
  logical :: Reorder
  integer,parameter :: m=4
  real, dimension(m) :: V=0.
  real :: W=0.
call MPI_INIT(code)

! Creation of the initial 2D grid
Dim2D(1) = 4
Dim2D(2) = 3
Period(:) = .false.
ReOrder = .false.
call MPI_CART_CREATE(MPI_COMM_WORLD,NDim2D,Dim2D,Period,ReOrder,Comm2D,code)
call MPI_COMM_RANK(Comm2D,rank,code)
call MPI_CART_COORDS(Comm2D,rank,NDim2D,Coord2D,code)

! Initialization of the V vector
if (Coord2D(1) == 1) V(:)=real(rank)

! Every row of the grid must be a 1D cartesian topology
remain_dims(1) = .true.
remain_dims(2) = .false.
! Subdivision of the 2D cartesian grid
call MPI_CART_SUB(Comm2D,remain_dims,Comm1D,code)

! The processes of the column 2 distribute the V vector to the processes of their row
call MPI_SCATTER(V,1,MPI_REAL,W,1,MPI_REAL,1,Comm1D,code)

print '("Rank : ",I2," ; Coordinates : (",I1,"","I1,")) ; W = ",F2.0")', &
    rank,Coord2D(1),Coord2D(2),W

call MPI_FINALIZE(code)
end program CommCartSub
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) ; \( W = 4. \)
1 Introduction
2 Environment
3 Point-to-point Communications
4 Collective communications
5 One-sided Communication
6 Derived datatypes
7 Optimizations
8 Communicators
9 MPI-IO

9.1 Introduction ................................................................. 247
   9.1.1 Presentation .......................................................... 247
   9.1.2 Issues ................................................................. 249
   9.1.3 Definitions ............................................................ 252

9.2 File management ............................................................ 254

9.3 Reads/Writes: general concepts ........................................ 257

9.4 Individual Reads/Writes .................................................. 261
   9.4.1 Via explicit offsets ................................................. 261
   9.4.2 Via Individual File Pointers ....................................... 270
   9.4.3 Shared File Pointers ............................................... 282

9.5 Collective Reads/Writes .................................................. 289
   9.5.1 Via explicit offsets ................................................. 290
   9.5.2 Via individual file pointers ....................................... 294
   9.5.3 Via shared file pointers ......................................... 308

9.6 Explicit positioning of pointers in a file ............................ 314

9.7 Definition of views ........................................................ 323

9.8 NonBlocking Reads/Writes .............................................. 377
Very logically, the applications that make large calculations also handle large amounts of data, and generate therefore a significant number of I/O.

Thus, their effective treatment sometimes affects very strongly the global performances of applications.
The I/O optimization of parallel codes is made by the combination:

- of their parallelization, in order to avoid creating a bottleneck due to their serialization;
- of explicitly implemented techniques at the level of programming (nonblocking reads / writes);
- of specific operations supported by the operating system (grouping of requests, buffer management of I/O, etc.).

The goals of MPI-IO, via the high-level interface that it proposes, are to provide simplicity, expressivity and flexibility, while authorizing performing implementations that take into account the software and hardware specificities of I/O devices of the target machines.

MPI-IO provides an interface modeled on the one used for message passing. The definition of data accessed according to the processes is made by the use of (basic or derived) datatypes. As for the notions of nonblocking and collective operations, they are managed similarly to what MPI proposes for the messages.

MPI-IO authorises both sequential and random accesses.
It is a high-level interface, where, as we said, some optimization techniques are accessible to the users, but where many basic optimizations can be transparently implemented. Two important examples are:

- The case of multiple accesses, by one single process, to small discontiguous blocks: this can be treated by a data sieving mechanism, after grouping of a set of requests, reading of a big contiguous block of the disk towards a buffer memory area, then allocation to the user memory areas of the adequate subgroups of data;
Requests on small non-contiguous blocks of a file

Reading of a big contiguous block and transfer in a buffer memory area

Memory copies of the required elements in the variables of the program

Figure 45: Data sieving mechanism in the case of multiple accesses, by one single process, to small discontiguous blocks
the case of access, by a group of processes, to discontiguous blocks (case of distributed arrays, for example) : this can be treated by collective I/O by dividing the operations into two phases.

Figure 46: Read in two phases, by a group of processes
**file**: an MPI file is an ordered group of typed data. A file is collectively opened by all the processes of a communicator. All the later collective I/O operations are made in this context.

**displacement**: it is an absolute address, in bytes, in relation to the beginning of the file and from which a view starts.

**etype**: it is the data unit used to calculate the position and to access data. This can be any MPI datatype, predefined or created as a derived datatype.

**filetype**: it is a mask that constitutes the basis of the partitioning of a file between processes (if the file is seen as a paving in a dimension, the filetype is the *elementary tile* which serves the paving). It is either an etype, or an MPI derived datatype constructed as a repetition of occurrences of such an etype (the *holes* — non-accessed parts of the file — must also be a multiple of the used etype).
view: it defines the group of visible (and therefore accessible) data of a file, once this one is opened. It is an ordered group of elementary datatypes.

offset: it is the position in the file, expressed by the number of etype, relatively to the current view (the holes defined in the view are not taken into account to calculate the offsets).

file handle: it is an opaque object created at the opening and destroyed at the closing of a file. All the operations on a file are made by specifying its file handle as a reference.

file pointer: they are automatically updated by MPI and determine offsets inside the file. There are two types: the individual file pointers which are specific to each process that have opened the file; the shared file pointers which are common to all the processes that have opened the file.

file size: the MPI file size is measured in bytes.
The file management tasks are collective operations made by all the processes of the indicated communicator.

We are only describing here the principal subroutines (opening, closing) but others are available (deletion, etc.).

The attributes (describing the access rights, the opening mode, the possible destruction at the closing, etc.) must be precised by sum on predefined constants.

All the processes of the communicator inside of which a file is open will participate in the later collective operations of data access.

The opening of a file returns a file handle, which will be later used in all the operations relative to this file.

The available information via the `MPI_FILE_SET_INFO()` subroutine varies from one implementation to another.
Table 4: Attributes that can be positioned during the opening of files

<table>
<thead>
<tr>
<th>Attribut</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MODE_RDONLY</td>
<td>read only</td>
</tr>
<tr>
<td>MPI_MODE_RDWR</td>
<td>reading and writing</td>
</tr>
<tr>
<td>MPI_MODE_WRONLY</td>
<td>write only</td>
</tr>
<tr>
<td>MPI_MODE_CREATE</td>
<td>create the file if it does not exist</td>
</tr>
<tr>
<td>MPI_MODE_EXCL</td>
<td>error if the file exists</td>
</tr>
<tr>
<td>MPI_MODE_UNIQUE_OPEN</td>
<td>error if the file is already open by another application</td>
</tr>
<tr>
<td>MPI_MODESEQUENTIAL</td>
<td>sequential access</td>
</tr>
<tr>
<td>MPI_MODE_APPEND</td>
<td>pointers at the end of file (add mode)</td>
</tr>
<tr>
<td>MPI_MODE_DELETE_ON_CLOSE</td>
<td>delete after the closing</td>
</tr>
</tbody>
</table>
program open01

use mpi
implicit none

integer :: fh,code

call MPI_INIT(code)

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

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

end program open01

> ls -l file.data
-rw------- 1 name grp 0 Feb 08 12:13 file.data
The data transfers between files and memory areas of processes are made via explicit calls to read and write subroutines.

We distinguish three aspects to file access:

- **positioning**, which can be explicit (by specifying for example the desired number of bytes from the beginning of the file) or implicit, via pointers managed by the system (these pointers can be of two types: either individual to each process, or shared by all the processes);
- **synchronism**, the accesses can be blocking or nonblocking;
- **coordination**, the accesses can be collective (that is to say made by all the processes of the communicator inside of which the file is opened) or specific only to one or many processes.

There are many available variants: we will describe some of them.
Table 5: Summary of possible access types

<table>
<thead>
<tr>
<th>Positioning</th>
<th>Syncronism</th>
<th>Coordination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>individual</td>
<td>collective</td>
</tr>
<tr>
<td>explicit</td>
<td>blocking</td>
<td>MPI_FILE_READ_AT</td>
</tr>
<tr>
<td>offsets</td>
<td></td>
<td>MPI_FILE_WRITE_AT AT</td>
</tr>
<tr>
<td></td>
<td>nonblocking</td>
<td>MPI_FILE_IREAD_AT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_FILE_IWRITE_AT AT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_FILE_READ_AT_ALL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_FILE_WRITE_AT_ALL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_FILE_IREAD_AT_ALL_BEGIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_FILE_IWRITE_AT_ALL_BEGIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_FILE_READ_AT_ALL_END</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_FILE_WRITE_AT_ALL_END</td>
</tr>
</tbody>
</table>

*see next page*
<table>
<thead>
<tr>
<th>Positioning</th>
<th>Synchronization</th>
<th>Coordination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>individual</td>
</tr>
<tr>
<td>individual</td>
<td>blocking</td>
<td>MPI_FILE_READ</td>
</tr>
<tr>
<td>file</td>
<td>nonblocking</td>
<td>(MPI_FILE_WRITE</td>
</tr>
<tr>
<td>pointers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_FILE_IREAD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(MPI_FILE_IWRITE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shared</td>
<td>blocking</td>
<td>MPI_FILE_READ_SHARED</td>
</tr>
<tr>
<td>file</td>
<td>nonblocking</td>
<td>(MPI_FILE_WRITE_SHARED</td>
</tr>
<tr>
<td>pointers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_FILE_IREAD_SHARED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(MPI_FILE_IWRITE_SHARED</td>
</tr>
</tbody>
</table>
It is possible to mix the access types performed at the same file inside an application.

The accessed memory areas are described by three quantities:

- the initial address of the concerned area;
- the number of elements;
- the datatype, which must match a sequence of contiguous copies of the etype of the current "view".
The offset is expressed in the number of occurrences of a datatype, which must be a multiple of the etype of the current "view".

The file must not have been opened with the MPI_MODE_SEQUENTIAL attribute.
program write_at
   use mpi
   implicit none

   integer, parameter :: nb_values=10
   integer :: i,rank,fh,code,nb_bytes_integer
   integer(kind=MPI_OFFSET_KIND) :: offset
   integer, dimension(nb_values) :: values
   integer, dimension(MPI_STATUS_SIZE) :: status

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

   values(:)= /(i+rank*100,i=1,nb_values)/
   print *, "Write process",rank, ":",values(:)

   call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_WRONLY + MPI_MODE_CREATE, &
                     MPI_INFO_NULL,fh,code)

   call MPI_TYPE_SIZE(MPI_INTEGER,nb_bytes_integer,code)

   offset=rank*nb_values*nb_bytes_integer
   call MPI_FILE_WRITE_AT(fh,offset,values,nb_values,MPI_INTEGER, &
                          status,code)

   call MPI_FILE_CLOSE(fh,code)
   call MPI_FINALIZE(code)
end program write_at
Process 0

1 2 3 4 5 6 7 8 9 10

File

Process 1

101 102 103 104 105 106 107 108 109 110

Figure 47: MPI_FILE_WRITE_AT()
Figure 47: MPI_FILE_WRITE_AT()
Process 0

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

File

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td>108</td>
<td>109</td>
<td>110</td>
</tr>
</tbody>
</table>

Process 1

| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |

Figure 47: MPI_FILE_WRITE_AT()

```bash
> mpiexec -n 2 write_at

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

use mpi
implicit none

integer, parameter :: nb_values=10
integer :: rank,fh,code,nb_bytes_integer
integer(kind=MPI_OFFSET_KIND) :: offset
integer, dimension(nb_values) :: values
integer, dimension(MPI_STATUS_SIZE) :: status

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

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

call MPI_TYPE_SIZE(MPI_INTEGER,nb_bytes_integer,code)

offset=rank*nb_values*nb_bytes_integer
call MPI_FILE_READ_AT(fh,offset,values,nb_values,MPI_INTEGER, &
status,code)
print *, "Read process",rank,"::",values(:)

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

end program read_at
Process 0

File

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

Process 1

Figure 48: MPI_FILE_READ_AT()

> mpiexec -n 2 read_at

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

File

Process 1

Figure 48: MPI_FILE_READ_AT()

`mpiexec -n 2 read_at`

Read process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
Figure 48: MPI_FILE_READ_AT()

```
> mpiexec -n 2 read_at

Read process 0: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1: 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```
In these cases, an individual pointer is managed by the system, per file and per process.

For a given process, two successive accesses to the same file allows therefore the automatic access to consecutive elements of it.

In all these subroutines, the shared pointers are never accessed or modified.

After each access, the pointer is positioned on the next elementary datatype.

The file must not have been opened with the `MPI_MODE_SEQUENTIAL` attribute.
program read01

use mpi
implicit none

integer, parameter :: nb_values=10
integer :: rank,fh,code
integer, dimension(nb_values) :: values
integer, dimension(MPI_STATUS_SIZE) :: status

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

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

call MPI_FILE_READ(fh,values,6,MPI_INTEGER,status,code)
call MPI_FILE_READ(fh,values(7),4,MPI_INTEGER,status,code)

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

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

end program read01
Process 0

File

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |

Process 1

Figure 49: Example 1 of MPI_FILE_READ()

```bash
$ mpiexec -n 2 read01
Read process 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 0: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```
Figure 49: Example 1 of MPI_FILE_READ()

```
$ mpiexec -n 2 read01
Read process 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 0: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```
Figure 49: Example 1 of `MPI_FILE_READ()`

```
> mpiexec -n 2 read01
Read process 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 0: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```
Figure 49: Example 1 of MPI_FILE_READ()

mpiexec -n 2 read01

Read process 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 0: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Figure 49: Example 1 of MPI_FILE_READ()

```
> mpiexec -n 2 read01
Read process 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 0: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```
program read02
  use mpi
  implicit none

  integer, parameter :: nb_values=10
  integer :: rank,fh,code
  integer, dimension(nb_values) :: values=0
  integer, dimension(MPI_STATUS_SIZE) :: status

  call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY(MPI_INFO_NULL), &
  fh,code)
  
  if (rank == 0) then
    call MPI_FILE_READ(fh,values,5,MPI_INTEGER,status,code)
  else
    call MPI_FILE_READ(fh,values,8,MPI_INTEGER,status,code)
call MPI_FILE_READ(fh,values,5,MPI_INTEGER,status,code)
  end if
  print *, "Read process",rank,": ",values(1:8)
call MPI_FILE_CLOSE(fh,code)
call MPI_FINALIZE(code)
end program read02
Process 0

<table>
<thead>
<tr>
<th>File</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
<th>106</th>
<th>107</th>
<th>108</th>
<th>109</th>
<th>110</th>
</tr>
</thead>
</table>

Process 1

Figure 50: Example 2 of MPI\_FILE\_READ()

```
> mpiexec -n 2 read02

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

File

Process 1

Figure 50: Example 2 of MPI_FILE_READ()

> mpiexec -n 2 read02

Read process 0: 1, 2, 3, 4, 5, 0, 0, 0
Read process 1: 9, 10, 101, 102, 103, 6, 7, 8
Figure 50: Example 2 of MPI_FILE_READ()

```
> mpiexec -n 2 read02

Read process 0: 1, 2, 3, 4, 5, 0, 0, 0
Read process 1: 9, 10, 101, 102, 103, 6, 7, 8
```
Figure 50: Example 2 of MPI_FILE_READ()

```
> mpiexec -n 2 read02
Read process 0 : 1, 2, 3, 4, 5, 0, 0, 0
Read process 1 : 9, 10, 101, 102, 103, 6, 7, 8
```
There is one and only shared pointer per file, common to all the processes of the communicator in which the file is opened.

All the processes which make an I/O operation that uses the shared pointer must employ the same view of the file in order to do this.

If we use noncollective variants of the subroutines, the read order is not deterministic. If the treatment must be deterministic, we have to use the collective variants.

After each access, the pointer is positioned on the next etype.

In all of these subroutines, the individual pointers are never accessed or modified.
program read_shared01

use mpi
implicit none

integer :: rank, fh, code
integer, parameter :: nb_values = 10
integer, dimension(nb_values) :: values
integer, dimension(MPI_STATUS_SIZE) :: status

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

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

call MPI_FILE_READ_SHARED(fh, values, 4, MPI_INTEGER, status, code)
call MPI_FILE_READ_SHARED(fh, values(5), 6, MPI_INTEGER, status, code)

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

call MPI_FILE_CLOSE(fh, code)
call MPI_FINALIZE(code)
end program read_shared01
Process 0

Figure 51: Example of MPI_FILE_READ_SHARED()

> mpiexec -n 2 read_shared01

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

File

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

Process 1

1 2 3 4

Figure 51: Example of MPI_FILE_READ_SHARED()

```bash
> mpiexec -n 2 read_shared01
Read process 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 0: 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```
Process 0

File

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

Process 1

1 2 3 4

Figure 51: Example of MPI_FILE_READ_SHARED()
Figure 51: Example of MPI_FILE_READ_SHARED()

> mpiexec -n 2 read_shared01

Read process 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 0: 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
Figure 51: Example of MPI_FILE_READ_SHARED()

```bash
> mpiexec -n 2 read_shared01

Read process 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 0: 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```
All the processes of the communicator inside of which a file is open participate in the collective operations of data access.

The collective operations perform generally better than the individual operations, because they authorize more automatically implemented optimization techniques (like the accesses in two phases — see the paragraph 2).

In the collective operations, the accesses are executed in the order of the process ranks. The treatment is therefore in this case deterministic.
program read_at_all
    use mpi
    implicit none

    integer, parameter :: nb_values=10
    integer :: rank,fh,code,nb_bytes_integer
    integer(kind=MPI_OFFSET_KIND) :: offset_file
    integer, dimension(nb_values) :: values
    integer, dimension(MPI_STATUS_SIZE) :: status

    call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
    fh,code)
call MPI_TYPE_SIZE(MPI_INTEGER,nb_bytes_integer,code)
    offset_file=rank*nb_values*nb_bytes_integer
call MPI_FILE_READ_AT_ALL(fh,offset_file,values,nb_values, &
    MPI_INTEGER,status,code)
    print *, "Read process",rank,"::",values(:)
call MPI_FILE_CLOSE(fh,code)
call MPI_FINALIZE(code)
end program read_at_all
Process 0

File

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |

Process 1

Figure 52: Example of MPI_FILE_READ_AT_ALL()

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

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

File

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

Process 1

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

Figure 52: Example of MPI_FILE_READ_AT_ALL()
Figure 52: Example of MPI\_FILE\_READ\_AT\_ALL()

```
mpiexec -n 2 read_at_all
Read process 0: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1: 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```
program read_all01
    use mpi
    implicit none

    integer :: rank, fh, code
    integer, parameter :: nb_values=10
    integer, dimension(nb_values) :: values
    integer, dimension(MPI_STATUS_SIZE) :: status

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

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

    call MPI_FILE_READ_ALL(fh, values, 4, MPI_INTEGER, status, code)
    call MPI_FILE_READ_ALL(fh, values(5), 6, MPI_INTEGER, status, code)

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

    call MPI_FILE_CLOSE(fh, code)
    call MPI_FINALIZE(code)
end program read_all01
Process 0

File

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |

Process 1

Figure 53: Example 1 of MPI_FILE_READ_ALL()

```bash
> mpiexec -n 2 read_all01
Read process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```
Process 0

File

Process 1

Figure 53: Example 1 of MPI_FILE_READ_ALL()

> mpiexec -n 2 read_all01

Read process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
> mpiexec -n 2 read_all01

Read process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Figure 53: Example 1 of MPI_FILE_READ_ALL()

> mpiexec -n 2 read_all01

Read process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Figure 53: Example 1 of MPI_FILE_READ_ALL()

```bash
> mpiexec -n 2 read_all01
Read process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```
program read_all02
  use mpi
  implicit none

  integer, parameter :: nb_values=10
  integer :: rank,fh,index1,index2,code
  integer, dimension(nb_values) :: values=0
  integer, dimension(MPI_STATUS_SIZE) :: status

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

  if (rank == 0) then
    index1=3
    index2=6
  else
    index1=5
    index2=9
  end if

  call MPI_FILE_READ_ALL(fh,values(index1),index2-index1+1, &
  MPI_INTEGER,status,code)
  print *, "Read process",rank,"::",values(:)

  call MPI_FILE_CLOSE(fh,code)
  call MPI_FINALIZE(code)
end program read_all02
Process 0

File

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |

Process 1

Figure 54: Example 2 of MPI_FILE_READ_ALL()

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

```
0 0 1 2 3 4 0 0 0 0
```

File

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

Process 1

```
0 0 0 0 0 0 0 0 0 0
```

Figure 54: Example 2 of MPI_FILE_READ_ALL()

```
> mpiexec -n 2 read_all02
Read process 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
Read process 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```
Figure 54: Example 2 of MPI_FILE_READ_ALL()

$>$ mpiexec -n 2 read_all02

Read process 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
Read process 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
program read_all03
  use mpi
  implicit none

  integer, parameter :: nb_values=10
  integer :: rank, fh, code
  integer, dimension(nb_values) :: values = 0
  integer, dimension(MPI_STATUS_SIZE) :: status

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

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

  if (rank == 0) then
    call MPI_FILE_READ_ALL(fh, values(3), 4, MPI_INTEGER, status, code)
  else
    call MPI_FILE_READ_ALL(fh, values(5), 5, MPI_INTEGER, status, code)
  end if

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

  call MPI_FILE_CLOSE(fh, code)
  call MPI_FINALIZE(code)
end program read_all03
Process 0

File

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

Process 1

Figure 55: Example 3 of MPI_FILE_READ_ALL()

> mpiexec -n 2 read_all03
Read process 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
Read process 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
Figure 55: Example 3 of MPI_FILE_READ_ALL()

Process 0

Process 1

File

Read process 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
Read process 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0

> mpiexec -n 2 read_all03
Figure 55: Example 3 of MPI_FILE_READ_ALL()

> mpiexec -n 2 read_all03

Read process 1: 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
Read process 0: 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
program read_ordered
  use mpi
  implicit none

  integer :: rank, fh, code
  integer, parameter :: nb_values = 10
  integer, dimension(nb_values) :: values

  integer, dimension(MPI_STATUS_SIZE) :: status

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

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

  call MPI_FILE_READ_ORDERED(fh, values, 4, MPI_INTEGER, status, code)
  call MPI_FILE_READ_ORDERED(fh, values(5), 6, MPI_INTEGER, status, code)

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

  call MPI_FILE_CLOSE(fh, code)
  call MPI_FINALIZE(code)
end program read_ordered
Process 0

File

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

Process 1

Figure 56: Example of `MPI_FILE_ORDERED()`

```bash
> mpiexec -n 2 read_ordered

Read process 1 : 5, 6, 7, 8, 105, 106, 107, 108, 109, 110
Read process 0 : 1, 2, 3, 4, 9, 10, 101, 102, 103, 104
```
Figure 56: Example of MPI_FILE_ORDERED()

```
> mpiexec -n 2 read_ordered

Read process 1 : 5, 6, 7, 8, 105, 106, 107, 108, 109, 110
Read process 0 : 1, 2, 3, 4, 9, 10, 101, 102, 103, 104
```
Figure 56: Example of MPI_FILE_ORDERED()

> mpiexec -n 2 read_ordered

Read process 1: 5, 6, 7, 8, 105, 106, 107, 108, 109, 110
Read process 0: 1, 2, 3, 4, 9, 10, 101, 102, 103, 104
9.5 – Collective Reads/Writes

Process 0

File

Process 1

Figure 56: Example of MPI_FILE_ORDERED()

> mpiexec -n 2 read_ordered

Read process 1: 5, 6, 7, 8, 105, 106, 107, 108, 109, 110
Read process 0: 1, 2, 3, 4, 9, 10, 101, 102, 103, 104
Figure 56: Example of MPI\_FILE\_ORDERED()
The `MPI_FILE_GET_POSITION()` and `MPI_FILE_GET_POSITION_SHARED()` subroutines allow to know respectively the current value of individual pointers and the value of the shared pointer.

It is possible to explicitly position the individual pointers by the help of the `MPI_FILE_SEEK()` subroutine, and also the shared pointer with the `MPI_FILE_SEEK_SHARED()` subroutine.

There are three possible modes to set the offset of a pointer:

- `MPI_SEEK_SET` sets an absolute offset;
- `MPI_SEEK_CUR` sets a relative offset;
- `MPI_SEEK_END` positions the pointer at the end of the file, to which a possible displacement is added.

With `MPI_SEEK_CUR`, we can specify a negative offset, which allows to return back in the file.
program seek
    use mpi
    implicit none
    integer, parameter :: nb_values=10
    integer :: rank,fh,nb_bytes_integer,code
    integer(kind=MPI_OFFSET_KIND) :: offset
    integer, dimension(nb_values) :: values
    integer, dimension(MPI_STATUS_SIZE) :: status
    call MPI_INIT(code)
    call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
    call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
                     fh,code)
    call MPI_FILE_READ(fh,values,3,MPI_INTEGER,status,code)
    call MPI_TYPE_SIZE(MPI_INTEGER,nb_bytes_integer,code)
    offset=8*nb_bytes_integer
    call MPI_FILE_SEEK(fh,offset,MPI.Seek_Cur,code)
    call MPI_FILE_READ(fh,values(4),3,MPI_INTEGER,status,code)
    offset=4*nb_bytes_integer
    call MPI_FILE_SEEK(fh,offset,MPI.Seek_Set,code)
    call MPI_FILE_READ(fh,values(7),4,MPI_INTEGER,status,code)
    print *, "Read process",rank,"::",values(:)
    call MPI_FILE_CLOSE(fh,code)
    call MPI_FINALIZE(code)
end program seek
Process 0

File

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

Process 1

Figure 57: Example of MPI_FILE_SEEK()

```bash
mpiexec -n 2 seek
Read process 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Read process 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
```
Figure 57: Example of MPI_FILE_SEEK()

```
mpiexec -n 2 seek
Read process 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Read process 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
```
Figure 57: Example of MPI_FILE_SEEK()

```bash
> mpiexec -n 2 seek
Read process 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Read process 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
```
Figure 57: Example of MPI_FILE_SEEK()

> mpiexec -n 2 seek

Read process 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Read process 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Figure 57: Example of MPI_FILE_SEEK()

```
$ mpiexec -n 2 seek

Read process 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Read process 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
```
Figure 57: Example of MPI_FILE_SEEK()

> mpiexec -n 2 seek

Read process 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Read process 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Figure 57: Example of MPI_FILE_SEEK()

```
> mpiexec -n 2 seek

Read process 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Read process 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
```
The views are a flexible and powerful mechanism for describing the accessed areas in the files.

The views are constructed by the help of MPI derived datatypes.

Each process has its own view (or its own views) of a file, defined by three variables: a displacement, an etype and a filetype. A view is defined as a repetition of the filetype, once the initial positioning is made.

It is possible to define holes in a view, by not taking into account some data parts.

Different processes can perfectly have different views of the file, in order to access complementary parts of it.

A given process can define and use many different views of the same file.

A shared pointer may be used with a view only if all the processes have the same view.
If the file is open for writing, the described areas by the etypes and the filetypes cannot overlap, even partially.

The default view consists of a simple sequence of bytes (zero initial displacement, etype and filetype equal to \texttt{MPI\_BYTE}).

Figure 58: etype and filetype
If the file is open for writing, the described areas by the etypes and the filetypes cannot overlap, even partially.

The default view consists of a simple sequence of bytes (zero initial displacement, etype and filetype equal to `MPI_BYTE`).

![Diagram of etype and filetype](image_url)

Figure 58: etype and filetype
If the file is open for writing, the described areas by the etypes and the filetypes cannot overlap, even partially.

The default view consists of a simple sequence of bytes (zero initial displacement, etype and filetype equal to `MPI_BYTE`).

![Diagram of etype and filetype](image)

Figure 58: etype and filetype
If the file is open for writing, the described areas by the etypes and the filetypes cannot overlap, even partially.

The default view consists of a simple sequence of bytes (zero initial displacement, etype and filetype equal to \texttt{MPI\_BYTE}).

![Diagram of etype and filetype]

Figure 58: etype and filetype
If the file is open for writing, the described areas by the etypes and the filetypes cannot overlap, even partially.

The default view consists of a simple sequence of bytes (zero initial displacement, etype and filetype equal to `MPI_BYTE`).

![Diagram](image)

Figure 58: etype and filetype
If the file is open for writing, the described areas by the etypes and the filetypes cannot overlap, even partially.

The default view consists of a simple sequence of bytes (zero initial displacement, etype and filetype equal to `MPI_BYTE`).

![Diagram of etype and filetype](image)

**Figure 58**: etype and filetype
<table>
<thead>
<tr>
<th>etype</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>filetype proc.0</td>
<td></td>
</tr>
<tr>
<td>filetype proc.1</td>
<td></td>
</tr>
<tr>
<td>filetype proc.2</td>
<td></td>
</tr>
<tr>
<td>file</td>
<td></td>
</tr>
</tbody>
</table>

Figure 59: Example of definition of different filetypes according to the processes
Figure 59: Example of definition of different filetypes according to the processes
Figure 59: Example of definition of different filetypes according to the processes
etype

filetype proc.0

filetype proc.1

filetype proc.2

file

initial displacement

Figure 59: Example of definition of different filetypes according to the processes
Figure 59: Example of definition of different filetypes according to the processes
Figure 59: Example of definition of different filetypes according to the processes
program read_view01

use mpi
implicit none

integer, parameter :: nb_values=10
integer :: rank,fh,filetype_temp1,filetype_temp2
integer :: filetype,code,size_integer
integer(kind=MPI_OFFSET_KIND) :: initial_displacement
integer(kind=MPI_ADDRESS_KIND), dimension(2) :: displacements
integer, dimension(nb_values) :: values
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
fh,code)
call MPI_TYPE_CREATE_SUBARRAY(1,(/3/),(/2/),(/1/),MPI_ORDER_FORTRAN, &
MPI_INTEGER, filetype_temp1, code)
call MPI_TYPE_CREATE_SUBARRAY(1,(/2/),(/1/),(/1/),MPI_ORDER_FORTRAN, &
MPI_INTEGER, filetype_temp2, code)
call MPI_TYPE_SIZE(MPI_INTEGER,size_integer,code)

displacements(1) = 0
displacements(2) = 3*size_integer
call MPI_TYPE_CREATE_STRUCT(2,(/1,1/),displacements,&
(/filetype_temp1,filetype_temp2/),filetype, code)
call MPI_TYPE_COMMIT(filetype,code)

! Do not omit to go through an intermediate variable of kind 
! MPI_OFFSET_KIND, for portability reasons
initial_displacement=0
call MPI_FILE_SET_VIEW(fh,initial_displacement,MPI_INTEGER,filetype, &
"native",MPI_INFO_NULL,code)
call MPI_FILE_READ(fh,values,7,MPI_INTEGER,status,code)
call MPI_FILE_READ(fh,values(8),3,MPI_INTEGER,status,code)

print *,"Read process",rank,":",values(:)
call MPI_FILE_CLOSE(fh,code)
call MPI_FINALIZE(code)
end program read_view01
Process 0

File

Process 1

Figure 61: Example 1 of MPI_FILE_SET_VIEW()

`> mpiexec -n 2 read_view01`

Read process 1: 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
Read process 0: 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
Figure 61: Example 1 of MPI_FILE_SET_VIEW()

> mpiexec -n 2 read_view01

Read process 1: 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
Read process 0: 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
Figure 61: Example 1 of MPI_FILE_SET_VIEW()

```
> mpiexec -n 2 read_view01

Read process 1 : 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
Read process 0 : 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
```
Figure 61: Example 1 of MPI_FILE_SET_VIEW()
Figure 61: Example 1 of MPI_FILE_SET_VIEW()
Figure 61: Example 1 of MPI_FILE_SET_VIEW()

```bash
> mpiexec -n 2 read_view01
Read process 1 : 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
Read process 0 : 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
```
Figure 61: Example 1 of MPI_FILE_SET_VIEW()
Figure 61: Example 1 of MPI_FILE_SET_VIEW()

```
> mpiexec -n 2 read_view01

Read process 1 : 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
Read process 0 : 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
```
Figure 61: Example 1 of MPI_FILE_SET_VIEW()
initial Disp 0

etype

filetype proc. 0

filetype proc. 1

Figure 62: Filetype used in example 2 of MPI_FILE_SET_VIEW()
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, code)
call MPI_FILE_OPEN(MPI_COMM_WORLD, "data.dat", MPI_MODE_RDONLY, MPI_INFO_NULL, & fh, code)

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

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

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

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

print *, "Read process", rank, ",: ", values(:)
call MPI_FILE_CLOSE(fh, code)
call MPI_FINALIZE(code)
end program read_view02
Process 0

File

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
<th>106</th>
<th>107</th>
<th>108</th>
<th>109</th>
<th>110</th>
</tr>
</thead>
</table>

Process 1

Figure 63: Example 2 of MPI_FILE_SET_VIEW()

```bash
> mpiexec -n 2 read_view02
Read process 1: 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0: 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
```
Figure 63: Example 2 of MPI_FILE_SET_VIEW()

```
$ mpiexec -n 2 read_view02
Read process 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0 : 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
```
Figure 63: Example 2 of MPI_FILE_SET_VIEW()

> mpiexec -n 2 read_view02

Read process 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0 : 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
> mpiexec -n 2 read_view02

Read process 1: 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0: 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
Figure 63: Example 2 of `MPI_FILE_SET_VIEW()`

```
> mpiexec -n 2 read_view02
Read process 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0 : 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
```
Figure 63: Example 2 of MPI_FILE_SET_VIEW()

```
> mpiexec -n 2 read_view02
Read process 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0 : 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
```
Figure 63: Example 2 of MPI_FILE_SET_VIEW()

Process 0
[1 2 5 6 9 10]

File
[1 2 3 4 5 6 7 8 9 10 101 102 103 104 105 106 107 108 109 110]

Process 1
[3 4 7 8 101 102]

mpiexec -n 2 read_view02

Read process 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0 : 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
Figure 63: Example 2 of MPI_FILE_SET_VIEW()

```bash
> mpiexec -n 2 read_view02

Read process 1: 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0: 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
```
Figure 63: Example 2 of MPI_FILE_SET_VIEW()

> mpiexec -n 2 read_view02

Read process 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0 : 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
Figure 63: Example 2 of MPI_FILE_SET_VIEW()

> mpiexec -n 2 read_view02

Read process 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0 : 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
Figure 63: Example 2 of MPI_FILE_SET_VIEW(

```bash
> mpiexec -n 2 read_view02
Read process 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Read process 0 : 1, 2, 5, 6, 9, 10, 103, 104, 107, 108
```
program read_view03

  use mpi
  implicit none

  integer, parameter :: nb_values=10
  integer :: rank,fh,code, &
             filetype_1,filetype_2,filetype_3,nb_bytes_integer
  integer(kind=MPI_OFFSET_KIND) :: initial_displacement
  integer, dimension(nb_values) :: values
  integer, dimension(MPI_STATUS_SIZE) :: status

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
  call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
                     fh,code)
call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/0/), &
    MPI_ORDER_FORTRAN, MPI_INTEGER, filetype_1, code)
call MPI_TYPE_COMMIT(filetype_1, code)
call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/2/), &
    MPI_ORDER_FORTRAN, MPI_INTEGER, filetype_2, code)
call MPI_TYPE_COMMIT(filetype_2, code)
call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/1/),(/3/), &
    MPI_ORDER_FORTRAN, MPI_INTEGER, filetype_3, code)
call MPI_TYPE_COMMIT(filetype_3, code)

initial_displacement=0
call MPI_FILE_SET_VIEW(fh, initial_displacement, MPI_INTEGER, filetype_1, &
    "native", MPI_INFO_NULL, code)
call MPI_FILE_READ(fh, values, 4, MPI_INTEGER, status, code)
call MPI_FILE_SET_VIEW(fh, initial_displacement, MPI_INTEGER, filetype_2, &
    "native", MPI_INFO_NULL, code)
call MPI_FILE_READ(fh, values(5), 3, MPI_INTEGER, status, code)
call MPI_TYPE_SIZE(MPI_INTEGER, nb_bytes_integer, code)
initial_displacement=2*nb_bytes_integer
call MPI_FILE_SET_VIEW(fh, initial_displacement, MPI_INTEGER, filetype_3, &
    "native", MPI_INFO_NULL, code)
call MPI_FILE_READ(fh, values(8), 3, MPI_INTEGER, status, code)

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

call MPI_FILE_CLOSE(fh, code)
call MPI_FINALIZE(code)
end program read_view03
Process 0

File

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |

Process 1

Figure 65: Example 3 of MPI_FILE_SET_VIEW()

```bash
$ mpiexec -n 2 read_view03
Read process 1: 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0: 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
Figure 65: Example 3 of MPI_FILE_SET_VIEW()

```
> mpiexec -n 2 read_view03
Read process 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
**Figure 65: Example 3 of MPI_FILE_SET_VIEW()**

- **Process 0**:
  - Read process 1: 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
  - Read process 0: 1, 2, 5, 6, 3, 4, 7, 6, 10, 104

```bash
$ mpiexec -n 2 read_view03
```

```plaintext
Read process 1: 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0: 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
Figure 65: Example 3 of MPI_FILE_SET_VIEW()
Figure 65: Example 3 of `MPI_FILE_SET_VIEW()`

```bash
> mpiexec -n 2 read_view03

Read process 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
Figure 65: Example 3 of MPI_FILE_SET_VIEW()

```
> mpiexec -n 2 read_view03

Read process 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
Figure 65: Example 3 of MPI_FILE_SET_VIEW()

```
> mpiexec -n 2 read_view03
Read process 1: 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0: 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
Figure 65: Example 3 of MPI_FILE_SET_VIEW()

```
> mpiexec -n 2 read_view03
Read process 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
Figure 65: Example 3 of MPI_FILE_SET_VIEW()

```
> mpiexec -n 2 read_view03
Read process 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
Figure 65: Example 3 of MPI_FILE_SET_VIEW()

> mpiexec -n 2 read_view03

Read process 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Figure 65: Example 3 of MPI_FILE_SET_VIEW()
Figure 65: Example 3 of MPI_FILE_SET_VIEW()

```
> mpiexec -n 2 read_view03

Read process 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
Figure 65: Example 3 of MPI_FILE_SET_VIEW()

```
> mpiexec -n 2 read_view03

Read process 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
Figure 65: Example 3 of MPI_FILE_SET_VIEW()

```bash
> mpiexec -n 2 read_view03

Read process 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
Figure 65: Example 3 of `MPI_FILE_SET_VIEW()`

```
> mpiexec -n 2 read_view03
Read process 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
Read process 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```
The nonblocking I/O are implemented according to the model used for the nonblocking communications.

A nonblocking access must later lead to an explicit test of completeness or to a standby (via `MPI_TEST()`, `MPI_WAIT()`, etc.), in a way similar to the management of nonblocking messages.

The advantage is to make an overlap between the computations and the I/O.
program iread_at

use mpi
implicit none

integer, parameter :: nb_values=10
integer :: i,nb_iterations=0,rank,nb_bytes_integer, &
            fh, request, code
integer(kind=MPI_OFFSET_KIND) :: offset
integer, dimension(nb_values) :: values
integer, dimension(MPI_STATUS_SIZE) :: status
logical :: finish

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

call MPI_TYPE_SIZE(MPI_INTEGER,nb_bytes_integer,code)

offset=rank*nb_values*nb_bytes_integer
call MPI_FILE_IREAD_AT(fh,offset,values,nb_values,& MPI_INTEGER requests,code)

do while (nb_iterations < 5000)
    nb_iterations=nb_iterations+1
    ! Computations overlapping the time consumed by the read operation ...
    call MPI_TEST(request,finish,status,code)
    if (finish) exit
end do
print *,"After",nb_iterations,"iterations, read process",rank,"":",values

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

end program iread_at
Process 0

File

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |

Process 1

Figure 66: Example of MPI_FILE_IREAD_AT()

> mpiexec -n 2 iread_at

After 1 iterations, read process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
After 1 iterations, read process 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
Figure 66: Example of MPI_FILE_IREAD_AT()

```bash
$ mpiexec -n 2 iread_at
After 1 iterations, read process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
After 1 iterations, read process 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```
After 1 iterations, read process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
After 1 iterations, read process 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
program iread
  use mpi
  implicit none

  integer, parameter :: nb_values=10
  integer :: rank,fh,request,code
  integer, dimension(nb_values) :: values
  integer, dimension(MPI_STATUS_SIZE) :: status

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

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

  call MPI_FILE_IREAD(fh,values,nb_values,MPI_INTEGER, request,code)
  ! Computation overlapping the time consumed by the read operation
  ...
  call MPI_WAIT(request,status,code)
  print *, "Read process",rank,"":"",values(:)

  call MPI_FILE_CLOSE(fh,code)
  call MPI_FINALIZE(code)
end program iread
Figure 67: Example of MPI_FILE_IREAD()

mpiexec -n 2 iread

Read process 0: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Figure 67: Example of MPI_FILE_IREAD()

```bash
> mpiexec -n 2 iread
Read process 0: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```
Figure 67: Example of MPI_FILE_IREAD()

```
> mpiexec -n 2 iread

Read process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Read process 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```
It is possible to carry out operations which are both collective and nonblocking, via a particular form of nonblocking collective operation.

This latter requires a call to two distinct subroutines, one for starting the operation and the other to end it.

We cannot modify the concerned memory area between the two phases of the operation.

Nevertheless, it is possible during this time to make noncollective operations on the file.

There can be only one such operation happening at the same time per process.
program read_ordered_begin_end

use mpi
implicit none

integer :: rank, fh, code
integer, parameter :: nb_values=10
integer, dimension(nb_values) :: values
integer, dimension(MPI_STATUS_SIZE) :: status

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

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

call MPI_FILE_READ_ORDERED_BEGIN(fh, values, 4, MPI_INTEGER, code)
print *, "Process number ":", rank

call MPI_FILE_READ_ORDERED_END(fh, values, status, code)

print *, "Read process", rank, ":", values(1:4)

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

end program read_ordered_begin_end
Process 0

File

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |

Process 1

Figure 68: Example of MPI_FILE_READ_ORDERED_BEGIN()

> mpiexec -n 2 read_ordered_begin_end

Process number : 0
Read process 0 : 1, 2, 3, 4
Process number : 1
Read process 1 : 5, 6, 7, 8
Figure 68: Example of MPI_FILE_READ_ORDERED_BEGIN()

```
> mpiexec -n 2 read_ordered_begin_end
Process number : 0
Read process 0 : 1, 2, 3, 4
Process number : 1
Read process 1 : 5, 6, 7, 8
```
Figure 68: Example of `MPI_FILE_READ_ORDERED_BEGIN()`

```bash
> mpiexec -n 2 read_ordered_begin_end

Process number : 0
Read process 0 : 1, 2, 3, 4
Process number : 1
Read process 1 : 5, 6, 7, 8
```
As we saw, MPI-IO provides a very rich group of functionalities, as well as a high-level interface. This latter, while remaining portable, allows at the same time to hide on users complex operations and to implement transparently optimizations specific to the target machines.

Some choices are clearly advisable:

- when the operations imply all the processes, or a group of these processes that can be defined in a particular communicator, the collective form of operations must be generally preferred;
- the use of subroutines of explicit positioning in the files are to be employed only in particular cases, the implicit use of shared or individual pointers provides a high-level interface;
- exactly as for the treatment of messages when those ones represent an important part of the application, the nonblocking is a preferred way of optimization to be implemented by the programers, but this must be implemented only after the verification of the correct behavior of the application in blocking mode.
1 Introduction

2 Environment

3 Point-to-point Communications

4 Collective communications

5 One-sided Communication

6 Derived datatypes

7 Optimizations

8 Communicators

9 MPI-IO

10 Conclusion
Use blocking point-to-point communications, this before going to nonblocking communications. It will be necessary then to try to make computations/communications overlap.

Use the blocking I/O functions, this before going to nonblocking I/O. Similarly, it will be necessary then to make I/O-computations overlap.

Write the communications as if the sendings were synchronous (`MPI_SSEND()`).

Avoid the synchronization barriers (`MPI_BARRIER()`), especially on the blocking collective functions.

The MPI/OpenMP hybrid programming can bring gains of scalability, in order for this approach to function well, it is obviously necessary to have good OpenMP performances inside each MPI process. A course is given at IDRIS (https://cours.idris.fr/).