

Message Passing Interface (MPI)

Isabelle DUPAYS

Marie FLÉ

Jérémie GAIDAMOUR

Dimitri LECAS

MPI – Plan I

1	Introduction	5
1.1	Availability and updating	6
1.2	Introduction	7
1.3	Concept of message passing	8
1.4	Distributed memory	13
1.5	History	18
1.6	Library	20
2	Environment	24
3	Point-to-point Communications	31
3.1	General Concepts	32
3.2	Blocking send and receive	34
3.3	Predefined MPI Datatypes	37
3.4	Other Possibilities	39
4	Collective communications	47
4.1	General concepts	48
4.2	Global synchronization: MPI_BARRIER()	50
4.3	Global distribution : MPI_BCAST()	51
4.4	Selective distribution: MPI_SCATTER()	54
4.5	Collection : MPI_GATHER()	57
4.6	Gather-to-all : MPI_ALLGATHER()	60
4.7	Extended gather : MPI_GATHERV()	63
4.8	Collection and distribution: MPI_ALLTOALL()	67

MPI – Plan II

4.9 Global reduction	71
4.10 Additions	80
5 Communication Modes	81
5.1 Point-to-Point Send Modes	82
5.2 Blocking call	83
5.3 Nonblocking communication	96
6 Derived datatypes	106
6.1 Introduction	107
6.2 Contiguous datatypes	109
6.3 Constant stride	110
6.4 Commit derived datatypes	112
6.5 Examples	113
6.6 Homogenous datatypes of variable strides	119
6.7 Subarray datatype constructor	125
6.8 Heterogenous datatypes	130
6.9 Size of MPI datatype	134
6.10 Conclusion	139
7 Communicators	140
7.1 Introduction	141
7.2 Example	142
7.3 Default communicator	143
7.4 Groups and communicators	144

MPI – Plan III

7.5 Partitioning of a communicator	145
7.6 Communicator built from a group	149
7.7 Topologies	150
8 MPI-IO	172
8.1 Introduction	173
8.2 File Manipulation	178
8.3 Data access: Concepts	181
8.4 Noncollective data access	185
8.5 Collective data access	198
8.6 Positioning the file pointers	209
8.7 File Views	212
8.8 Nonblocking Data Access	226
8.9 Advice	235
8.10 Definitions	236
9 Conclusion	238
10 Index	240
10.1 Constants MPI index	241
10.2 Subroutines MPI index	243

1	Introduction	
1.1	Availability and updating	6
1.2	Introduction	7
1.3	Concept of message passing	8
1.4	Distributed memory	13
1.5	History	18
1.6	Library	20
2	Environment	
3	Point-to-point Communications	
4	Collective communications	
5	Communication Modes	
6	Derived datatypes	
7	Communicators	
8	MPI-IO	
9	Conclusion	
10	Index	

1 – Introduction

1.1 – Availability and updating

This document is likely to be updated regularly. The most recent version is available on the Web server of IDRIS, section "Training Course Materials":

<https://cours.idris.fr>

- IDRIS

Institut for Development and Resources in Intensive Scientific Computing

Rue John Von Neumann

Bâtiment 506

BP 167

91403 ORSAY CEDEX

France

<http://www.idris.fr>

- Translated with the help of Cynthia TAUPIN.

1 – Introduction

1.2 – Introduction

Parallélism

The goal of parallel programming is to :

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

For group work, coordination is required. **MPI** is a library which allows process coordination by using a message-passing paradigm.

1 – Introduction

1.3 – Concept of message passing

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.

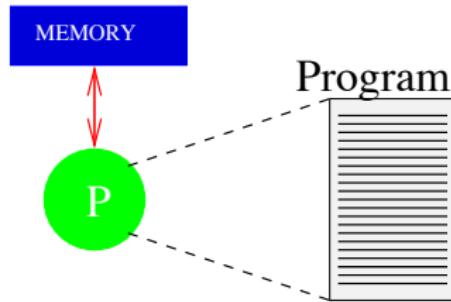


Figure 1: Sequential programming model

Message passing programming model

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

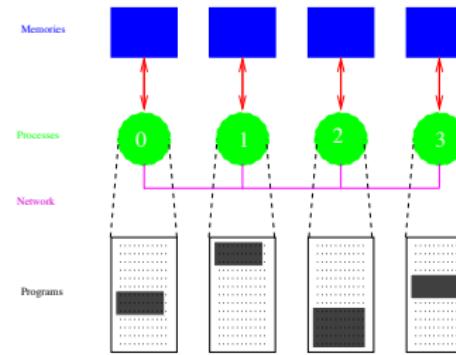


Figure 2: Message Passing Programming Model

Message Passing concepts

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

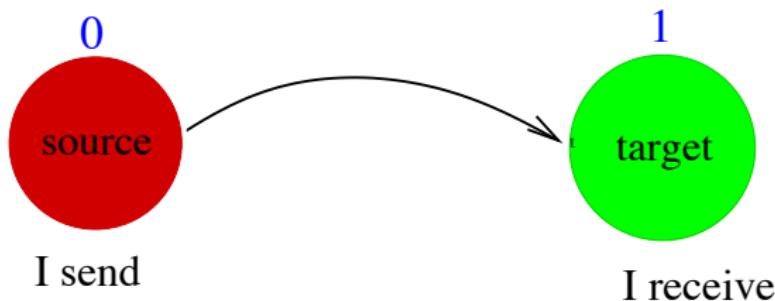


Figure 3: Message Passing

Message content

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

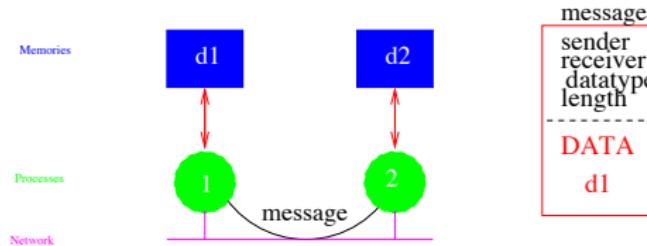


Figure 4: Message Construction

Environment

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

1 – Introduction

1.4 – Distributed memory

Supercomputer architecture

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

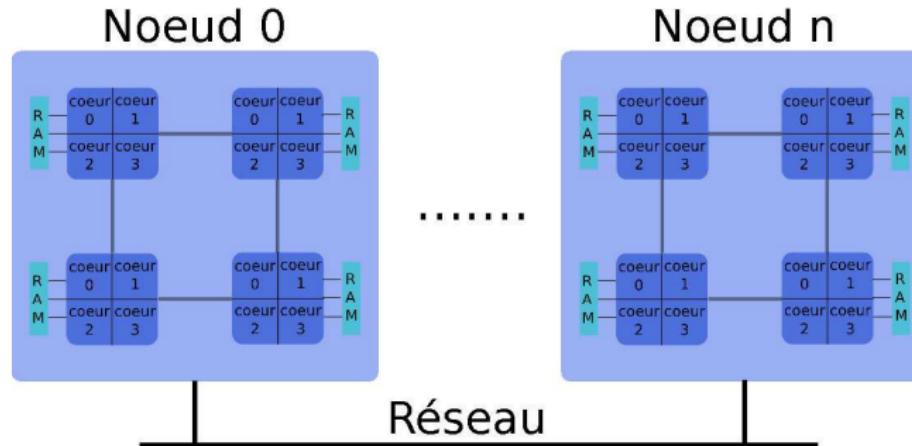
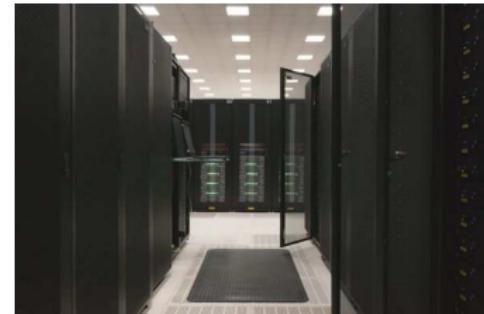


Figure 5: Supercomputer architecture

Ada

- 332 nodes
- 4 eight-core Intel SandyBridge processors, 2,7 GHz by node
- 10 624 cores
- 46 TB (304 nodes with 128 GB and 28 nodes with 256 GB)
- 230 Tflop/s peak
- 192 Tflop/s (linpack)
- 244 kWatt
- 786 MFLOPS/watt



Turing

- 6 144 nodes
- 16 POWER A2 processor 1,6 GHz by node
- 98 304 cores
- 393 216 logical cores
- 96 TiB (16 GB per node)
- 1 258 Tflop/s peak
- 1 073 Tflop/s (linpack)
- 493 kWatt
- 2 176 MFLOPS/watt



MPI vs OpenMP

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

Processus 0 Processus n

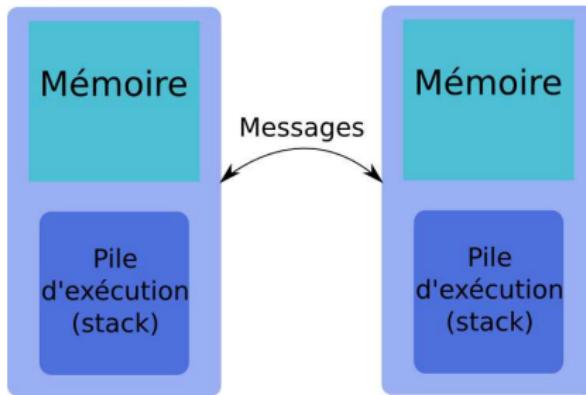


Figure 6: MPI scheme

Processus

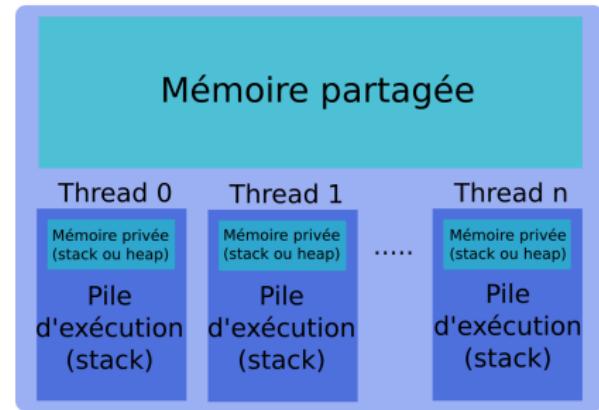


Figure 7: OpenMP scheme

Domain decomposition

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

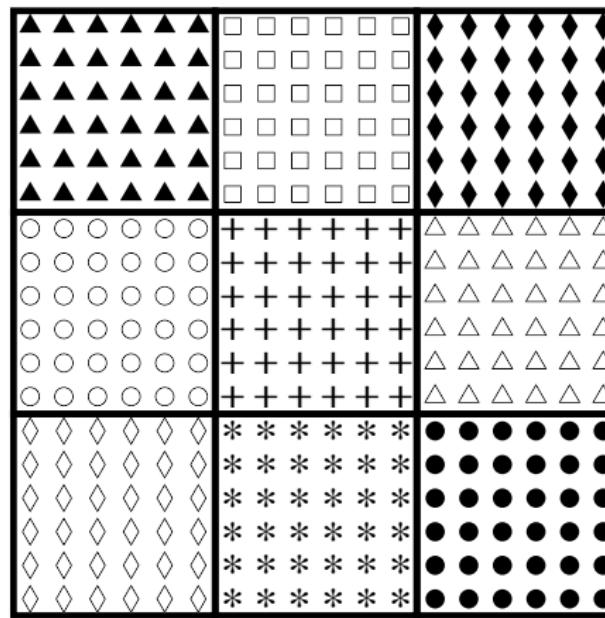


Figure 8: Decomposition in subdomains

1 – Introduction

1.5 – History

History

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

MPI 3.0

- Version 3.0: September 2012
 - Changes and important additions compared to version 2.2 ;
 - Principal changes :
 - Nonblocking collective communications
 - Revised implementation of one-sided communications
 - Fortran (2003-2008) bindings
 - C++ bindings removed
 - Interfacing of external tools (for debugging and performance measurements)
 - etc.
- see http://meetings mpi-forum.org/MPI_3.0_main_page.php
<https://svn mpi-forum.org/trac/mpi-forum-web/wiki>

1 – Introduction

1.6 – Library

Library

- Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*. High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart, 2012. <https://fs.hlrs.de/projects/par/mpi/mpi30/>
- William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd edition. MIT Press, 1999.
- William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.
- Peter S. Pacheco. *Parallel Programming with MPI*, Morgan Kaufman Publishers, 1997.
- Additional references :
<http://www.mpi-forum.org/docs/>
<http://www.mcs.anl.gov/research/projects/mpi/learning.html>

Open source MPI implementations

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

- MPICH2 : <http://www.mpich.org/>
- Open MPI : <http://www.open-mpi.org/>

Tools

- Debuggers
 - [Totalview](#)
<http://www.roguewave.com/products/totalview.aspx>
 - [DDT](#)
<http://www.allinea.com/products/ddt/>
- Performance measurement
 - [MPE : MPI Parallel Environment](#)
<http://www.mcs.anl.gov/research/projects/perfvis/download/index.htm>
 - [Scalasca : Scalable Performance Analysis of Large-Scale Applications](#)
<http://www.scalasca.org/>
 - [Vampir](#)
<http://www.vampir.eu/>

Open source parallel scientific libraries

- **ScaLAPACK** : Linear algebra problem solvers using direct methods. Sources can be downloaded from the website <http://www.netlib.org/scalapack/>.
- **PETSc** : Linear and non-linear algebra problem solvers using iterative methods. Sources can be downloaded from the website <http://www.mcs.anl.gov/petsc/>.
- **MUMPS** : MULTifrontal Massively Parallel sparse direct Solvers. Sources can be downloaded from the website <http://graal.ens-lyon.fr/MUMPS/>.
- **FFTW** : Fast Fourier Transform. Sources can be downloaded from the website <http://www.fftw.org>.

1 Introduction

2 Environment

3 Point-to-point Communications

4 Collective communications

5 Communication Modes

6 Derived datatypes

7 Communicators

8 MPI-IO

9 Conclusion

10 Index

Description

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

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

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

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

Difference between c and Fortran

For the c/c++ program :

- You need to include the `mpi.h` file.
- The code argument is the return value.
- Only the MPI prefix and the first following letter are in upper-case letters.
- Except for `MPI_INIT()`, the function arguments are identical to Fortran.

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

Communicators

- All the MPI operations require the **communicators** to be carried out. The default communicator is `MPI_COMM_WORLD` which includes all the active processes.

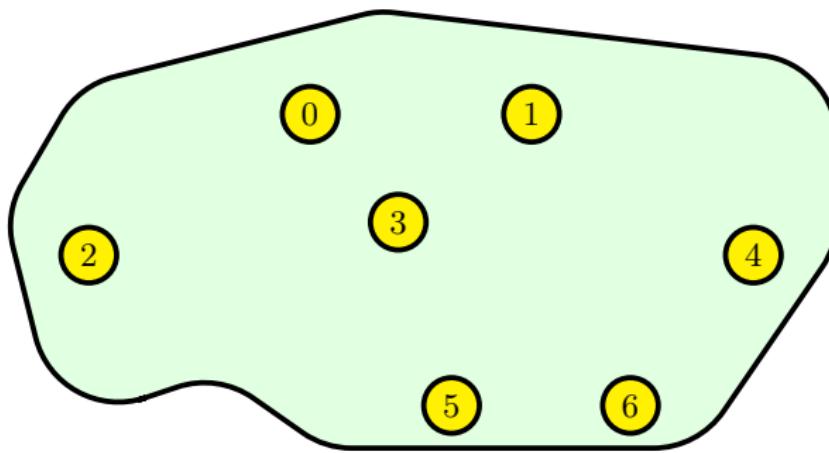


Figure 9: MPI_COMM_WORLD Communicator

Termination of a program

A program can sometimes be in a situation which requires it to stop execution before the normal ending. This is the case when one of the processes cannot allocate the memory needed for its calculation. In this case, the **MPI_ABORT()** subroutine must be used and not the Fortran instruction *stop*.

```
integer, intent(in) :: comm, error
integer, intent(out) :: code
call MPI_ABORT(comm, error, code)
```

- **comm** : All the processus belonging to this communicator will be stopped; it is advised to use **MPI_COMM_WORLD**;
- **error** : Error number is return to the UNIX environment.

Code

It is not necessary to check the code value after calls to MPI routines. By default, when MPI encounters a problem, the program is automatically stopped as in an implicit call to **MPI_ABORT()** subroutine.

Rank and size

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

```
integer, intent(out) :: nb_procs,code  
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
```

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

```
integer, intent(out) :: rank,code  
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
```

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

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

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

1	Introduction	
2	Environment	
3	Point-to-point Communications	
3.1	General Concepts	32
3.2	Blocking send and receive	34
3.3	Predefined MPI Datatypes	37
3.4	Other Possibilities	39
4	Collective communications	
5	Communication Modes	
6	Derived datatypes	
7	Communicators	
8	MPI-IO	
9	Conclusion	
10	Index	

3 – Point-to-point Communications

3.1 – General Concepts

General Concepts

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

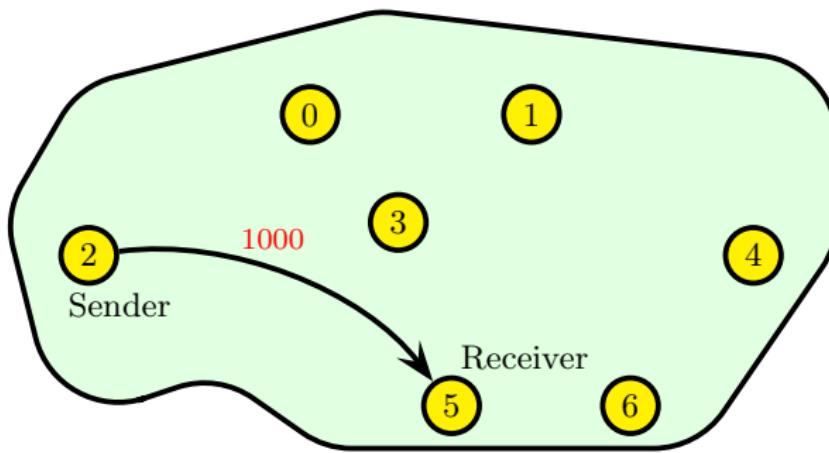


Figure 10: Point-to-point communication

General Concepts

- The sender and the receiver are identified by their **rank** in the communicator.
- The so-called **message envelope** is composed of:
 - The rank of the sender process
 - The rank of the receiver process
 - The message tag
 - The communicator which defines the process group and context of the communication
- The exchanged data are predefined (integer, real, etc.) or individual derived datatypes.
- In each case, there are several transfer **modes** which make calls to different protocols.

3 – Point-to-point Communications

3.2 – Blocking send and receive

Blocking Send **MPI_SEND**

```
<type>:: buf
integer :: count, datatype
integer :: dest, tag, comm, code
call MPI_SEND(buf,count,datatype,dest,tag,comm,code)
```

Send, starting at position **buf**, a message of **count** element, type **datatype**, tagged with **tag**, to the process of rank **dest** in the communicator **comm**.

Remark:

This call is blocking: The execution remains blocked until the message can be re-written without risk of overwriting the value to be sent.

Blocking Receive `MPI_RECV`

```
<type>:: buf
integer :: count, datatype
integer :: source, tag, comm, code
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_RECV(buf,count,datatype,source,tag,comm,status,code)
```

Receive, starting at the position `buf`, a message of `count` element, type `datatype` tagged `tag`, from the processus `source` and store it starting at position `buf`.

Remarks:

- `status` receives information about the communication : source, tag, code,
- The `MPI_RECV` will not work with a `MPI_SEND` unless these two calls have the same envelope (`source, dest, tag, comm`).
- This call is blocking: The execution remains blocked until the message content corresponds to the received message.

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

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

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

3 – Point-to-point Communications

3.3 – Predefined MPI Datatypes

Fortran MPI Datatypes

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

Table 1: Predefined MPI Datatypes (Fortran)

C MPI Datatypes

MPI Type	C Type
MPI_CHAR	<code>signed char</code>
MPI_SHORT	<code>signed short</code>
MPI_INT	<code>signed int</code>
MPI_LONG	<code>signed long int</code>
MPI_UNSIGNED_CHAR	<code>unsigned char</code>
MPI_UNSIGNED_SHORT	<code>unsigned short</code>
MPI_UNSIGNED	<code>unsigned int</code>
MPI_UNSIGNED_LONG	<code>unsigned long int</code>
MPI_FLOAT	<code>float</code>
MPI_DOUBLE	<code>double</code>
MPI_LONG_DOUBLE	<code>long double</code>

Table 2: Predefined MPI Datatypes (C)

3 – Point-to-point Communications

3.4 – Other Possibilities

Other possibilities

- 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 which can be used instead of the status variable.
- There are syntactic variants, `MPI_SENDRECV()` and `MPI_SENDRECV_REPLACE()` which carry out both send and receive operations (in the first case, the reception zone must be different than the send zone).
- It is possible to create more complex data structures by using derived datatypes.

Simultaneous send and receive **MPI_SENDRECV**

```
<type>:: sendbuf, recvbuf
integer :: sendcount, recvcount
integer :: sendtype, recvtype
integer :: source, dest, sendtag, recvtag, comm, code
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_SENDRECV(sendbuf,sendcount,sendtype,
                  dest,sendtag,
                  recvbuf,recvcount,recvtype,
                  source,recvtag, comm, status, code)
```

- Send, starting at position **sendbuf**, a message of **sendcount** element, type **sendtype**, labeled **sendtag**, to the process **dest** in the communicator **comm**.
- Receive, a message of **recvcount** element, type **recvtype**, tagged **recvtag**, from the process **source** in the communicator **comm** and store it starting at position **recvbuf**.

Simultaneous send and receive **MPI_SENDRECV**

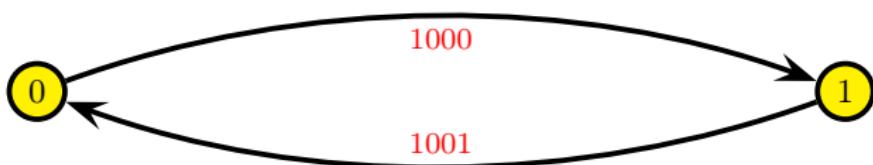


Figure 11: **sendrecv** Communication between the Processes 0 and 1

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

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

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

Attention!

It should be noted that in the case of a **synchronous** implementation of the **MPI_SEND()** subroutine, the code in the preceding example will deadlock if, instead of using the **MPI_RECV()** subroutine, we use a **MPI_SEND()** subroutine followed by a **MPI_RECV()** one. In fact, each of the two processes will wait for a receive command which will never come because the two sends would stay suspended.

```
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)
```

Simultaneous send and receive **MPI_SENDRECV_REPLACE**

```
<type>  :: buf
integer :: count
integer :: datatype
integer :: source, dest, sendtag, recvtag, comm, code
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_SENDRECV_REPLACE(buf,count,datatype,
                           dest,sendtag,
                           source, recvtag, comm, status, code)
```

- Send starting at position **buf** a message of **count** element of type **datatype**, labeled **sendtag**, to the processus **dest** in the communicator **comm** ;
- Receive a message of **count** element of type **datatype** tagged **recvtag** from the processus **source** in the communicator **comm** and store it at the same position **buf**.

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

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

1	Introduction
2	Environment
3	Point-to-point Communications
4	Collective communications
4.1	General concepts
4.2	Global synchronization: MPI_BARRIER()
4.3	Global distribution : MPI_BCAST()
4.4	Selective distribution: MPI_SCATTER()
4.5	Collection : MPI_GATHER()
4.6	Gather-to-all : MPI_ALLGATHER()
4.7	Extended gather : MPI_GATHERV()
4.8	Collection and distribution: MPI_ALLTOALL()
4.9	Global reduction
4.10	Additions
5	Communication Modes
6	Derived datatypes
7	Communicators
8	MPI-IO
9	Conclusion
10	Index

4 – Collective communications

4.1 – General concepts

General concepts

- **Collective** communications allow making a series of point-to-point communications in one single call.
- A collective communication always concerns all the processes of the indicated **communicator**.
- For each process, the call ends when its participation in the collective call is completed, in the sense of point-to-point communications (therefore, when the concerned memory area can be changed).
- It is unnecessary 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 calls to subroutines. An advantage of this is that collective communications never interfere with point-to-point communications.

Types of collective communications

There are three types of subroutines :

- ① One which ensures global synchronizations : `MPI_BARRIER()`
- ② Ones which only transfer data :
 - Global distribution of data : `MPI_BCAST()`
 - Selective distribution of data : `MPI_SCATTER()`
 - Collection of distributed data : `MPI_GATHER()`
 - Collection of distributed data by all the processes: `MPI_ALLGATHER()`
 - Collection and selective distribution by all the processes of distributed data:
`MPI_ALLTOALL()`
- ③ Ones which, in addition to the communications management, carry out operations on the transferred data :
 - Reduction operations (sum, product, maximum, minimum, etc.), whether of a predefined or personal type : `MPI_REDUCE()`
 - Reduction operations with distributing of the result (this is in fact equivalent to an `MPI_REDUCE()` followed by an `MPI_BCAST()`) : `MPI_ALLREDUCE()`

4 – Collective communications

4.2 – Global synchronization: MPI_BARRIER()

Global synchronization : MPI_BARRIER()

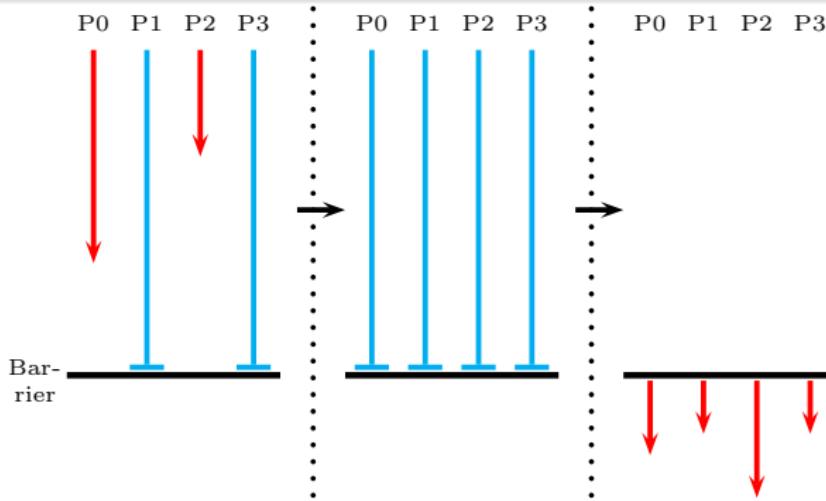


Figure 12: Global Synchronization : MPI_BARRIER()

```
integer, intent(out) :: code  
call MPI_BARRIER(MPI_COMM_WORLD,code)
```

4 – Collective communications

4.3 – Global distribution : MPI_BCAST()

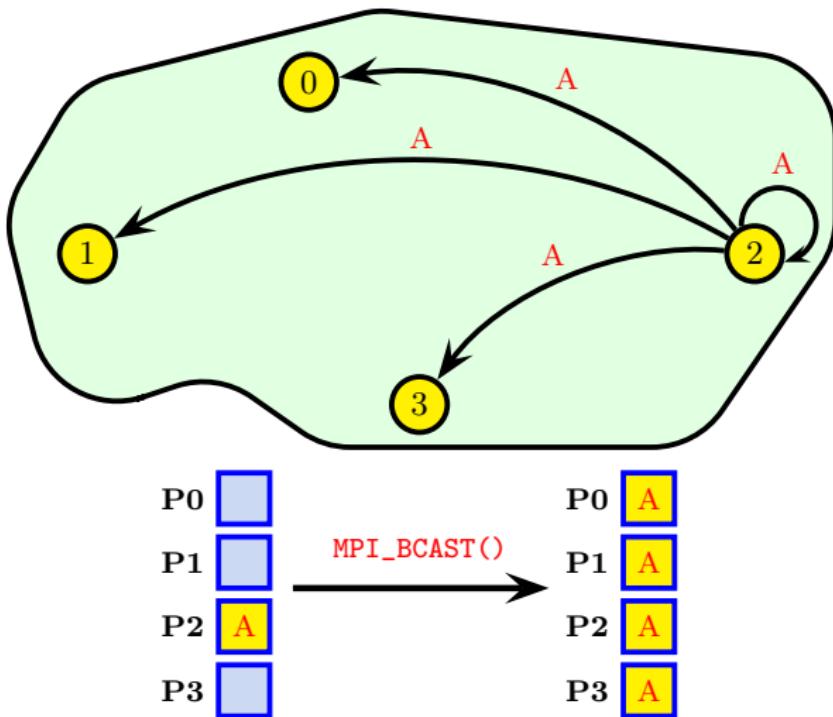


Figure 13: Global distribution : MPI_BCAST()

Global distribution : MPI_BCAST()

```
<type> :: buffer
integer :: count, datatype, root, comm, code
call MPI_BCAST(buffer, count, datatype, root, comm, code)
```

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

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

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

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

4 – Collective communications

4.4 – Selective distribution: MPI_SCATTER()

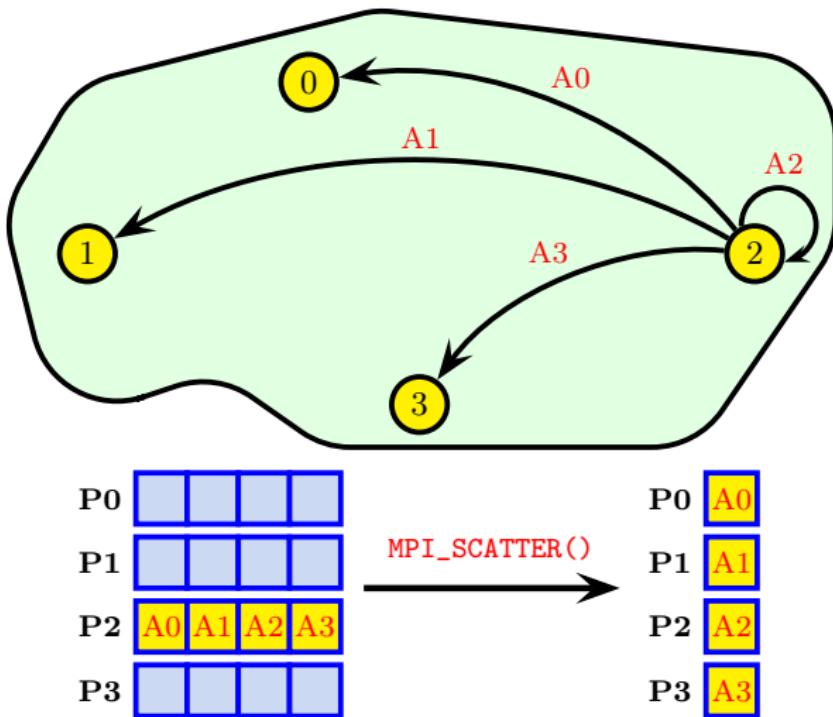


Figure 14: Selected distribution : MPI_SCATTER()

Selective distribution : MPI_SCATTER()

```
<type> :: sendbuf, recvbuf
integer :: sendcount, recvcount
integer :: sendtype, recvtype
integer :: root, comm, code

call MPI_SCATTER(sendbuf,sendcount,sendtype,
                 recvbuf,recvcount,recvtype,root,comm,code)
```

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

Remarks:

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

```

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

```

```

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

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

```

4 – Collective communications

4.5 – Collection : MPI_GATHER()

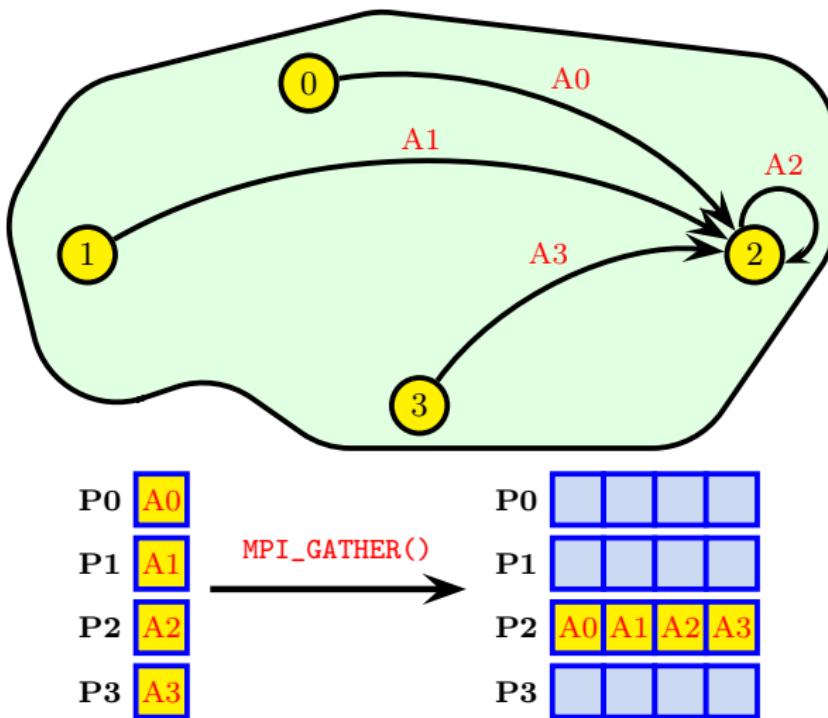


Figure 15: Collection: MPI_GATHER()

Collection: MPI_GATHER()

```
<type> :: sendbuf, recvbuf  
integer :: sendcount, recvcount  
integer :: sendtype, recvtype  
integer :: root, comm, code  
  
call MPI_GATHER(sendbuf,sendcount,sendtype,  
                 recvbuf,recvcount,recvtype,root,comm,code)
```

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

Remarks:

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

```

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

```

> mpiexec -n 4 gather

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

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

4 – Collective communications

4.6 – Gather-to-all : MPI_ALLGATHER()

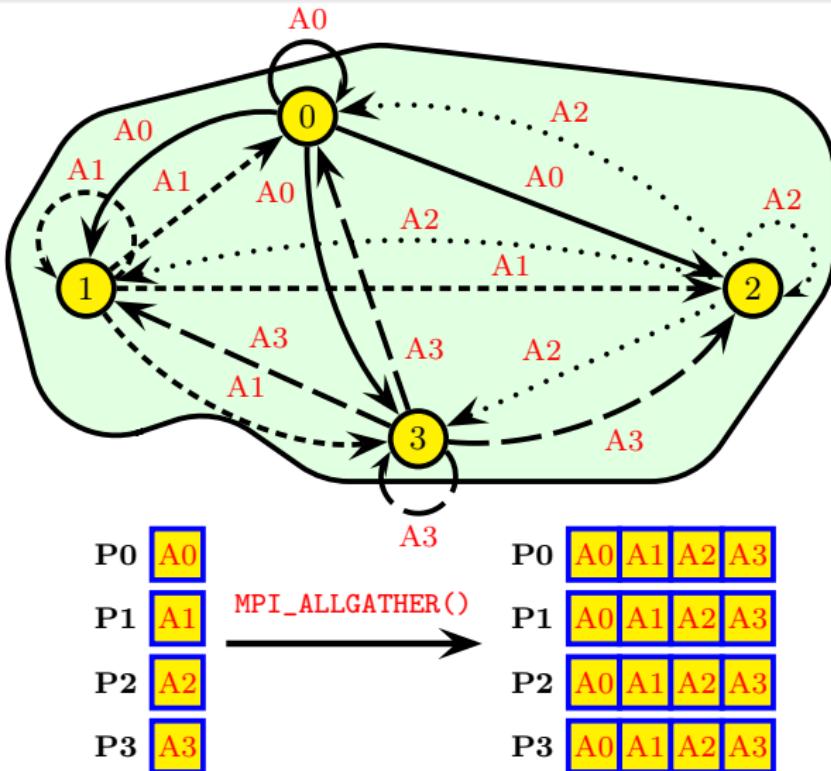


Figure 16: Gather-to-all: MPI_ALLGATHER()

Gather-to-all : MPI_ALLGATHER()

Corresponds to an MPI_GATHER() followed by an MPI_BCAST():

```
<type> :: sendbuf, recvbuf
integer :: sendcount, recvcount
integer :: sendtype, recvtype
integer :: comm, code

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

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

Remarks:

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

```

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

```

> mpiexec -n 4 allgather

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

4 – Collective communications

4.7 – Extended gather : MPI_GATHERV()

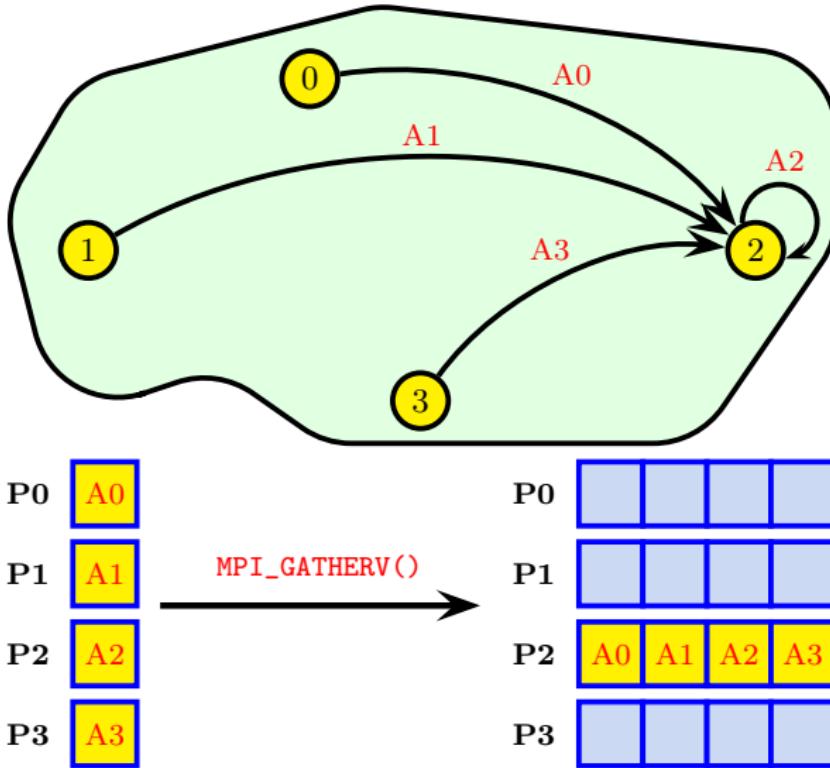


Figure 17: Extended gather: MPI_GATHERV()

Extended Gather : MPI_GATHERV()

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

```
<type > :: sendbuf, recvbuf
integer :: sendcount
integer :: sendtype, recvtype
integer, dimension(:) :: recvcounts, displs
integer :: root, comm, code

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

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

Remarks:

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

```
1 program gatherv
2 use mpi
3 implicit none
4 INTEGER, PARAMETER :: nb_values=10
5 INTEGER :: nb_procs, rank, block_length, i, code
6 REAL, DIMENSION(nb_values) :: data,remainder
7 REAL, ALLOCATABLE, DIMENSION(:) :: values
8 INTEGER, ALLOCATABLE, DIMENSION(:) :: nb_elements_received,displacement
9
10 CALL MPI_INIT(code)
11 CALL MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
12 CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
13
14 block_length=nb_values/nb_procs
15 remainder=mod(nb_values,nb_procs)
16 if(remainder > rank) block_length = block_length + 1
17 ALLOCATE(values(block_length))
18 values(:) = (/ (1000.+(rank*(nb_values/nb_procs))+min(rank,remainder)+i, &
19 i=1,block_length) /)
20
21 PRINT *, 'I, process ', rank,'sent my values array : ',&
22 values(1:block_length)
23
24 IF (rank == 2) THEN
25     ALLOCATE(nb_elements_received(nb_procs),displacement(nb_procs))
26     nb_elements_received(1) = nb_values/nb_procs
27     if (remainder > 0) nb_elements_received(1)=nb_elements_received(1)+1
28     displacement(1) = 0
29     DO i=2,nb_procs
30         displacement(i) = displacement(i-1)+nb_elements_received(i-1)
31         nb_elements_received(i) = nb_values/nb_procs
32         if (remainder > i-1) nb_elements_received(i)=nb_elements_received(i)+1
33     END DO
34 END IF
35
```

```
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, process 2, received ', data (1:nb_values)
CALL MPI_FINALIZE(code)
end program gatherv
```

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

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

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

4 – Collective communications

4.8 – Collection and distribution: MPI_ALLTOALL()

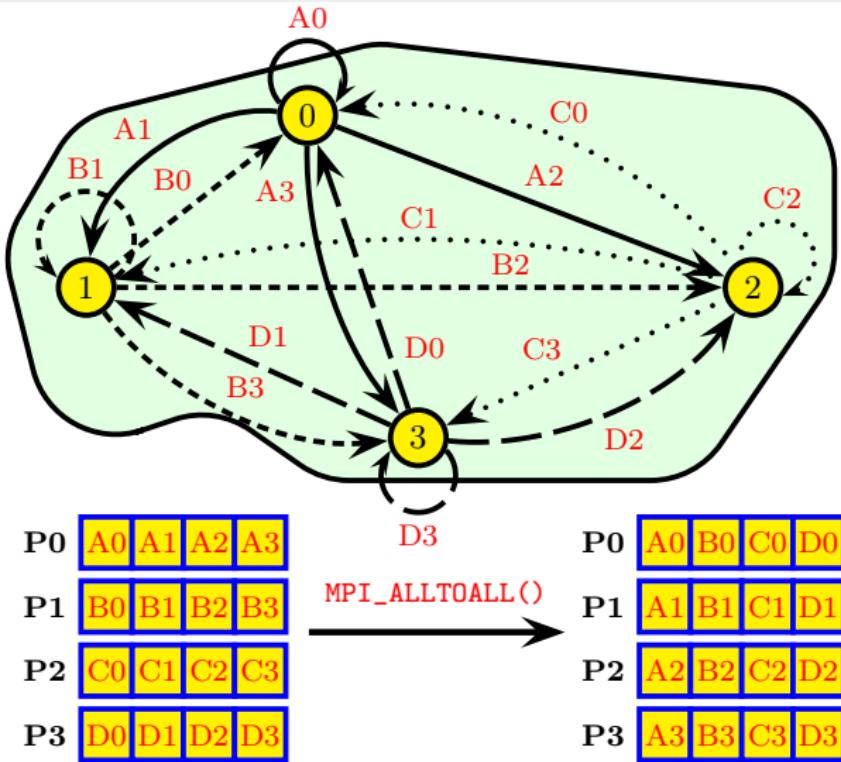


Figure 18: Collection and distribution: MPI_ALLTOALL()

Collection and distribution: MPI_ALLTOALL()

```
<type>  :: sendbuf, recvbuf
integer :: sendcount, recvcount
integer :: sendtype, recvtype
integer :: comm, code

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

Like **MPI_ALLGATHER()** except that each process sends distinct data: The i-th process sends its j-th chunk to the j-th process which places it in its i-th chunk.

Remark:

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

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

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

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

4 – Collective communications

4.9 – Global reduction

Global reduction

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

Operations

Table 3: Main Predefined Reduction Operations (there are also other logical operations)

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

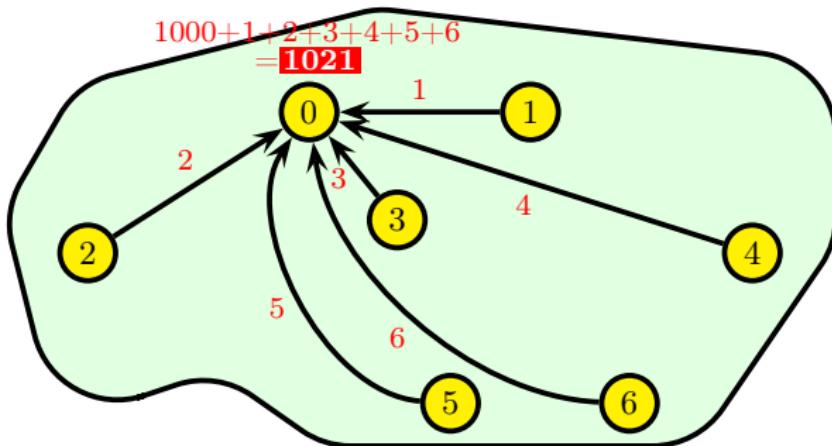


Figure 19: Distributed reduction (sum)

Global reduction : MPI_REDUCE()

```
<type> :: sendbuf, recvbuf
integer :: count, datatype, root
integer :: op, comm, code

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

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

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

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

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

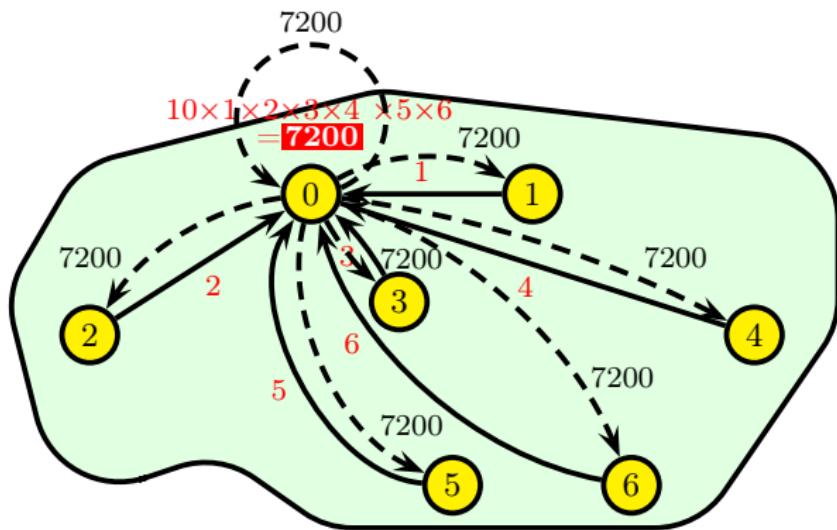


Figure 20: Distributed reduction (product) with distribution of the result

Global all-reduction : **MPI_ALLREDUCE()**

```
<type> :: sendbuf, recvbuf
integer :: count, datatype
integer :: op, comm, code

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

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

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

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

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

4 – Collective communications

4.10 – Additions

Additions

- The `MPI_SCATTERV()`, `MPI_GATHERV()`, `MPI_ALLGATHERV()` and `MPI_ALLTOALLV()` subroutines extend `MPI_SCATTER()`, `MPI_GATHER()`, `MPI_ALLGATHER()` and `MPI_ALLTOALL()` in case the processes have different numbers of elements to transmit or gather.
- For each reduction operation, the keyword `MPI_IN_PLACE` can be used in order to keep the result in the same place as the sending buffer: call
`MPI_REDUCE(MPI_IN_PLACE, recvbuf, ...)`
- 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()`

1	Introduction	
2	Environment	
3	Point-to-point Communications	
4	Collective communications	
5	Communication Modes	
5.1	Point-to-Point Send Modes	82
5.2	Blocking call.....	83
5.3	Nonblocking communication	96
6	Derived datatypes	
7	Communicators	
8	MPI-IO	
9	Conclusion	
10	Index	

5 – Communication Modes

5.1 – Point-to-Point Send Modes

Point-to-Point Send Modes

<i>Mode</i>	Blocking	Non-blocking
Standard send	<code>MPI_SEND()</code>	<code>MPI_ISEND()</code>
Synchronous send	<code>MPI_SSEND()</code>	<code>MPI_ISSEND()</code>
Buffered send	<code>MPI_BSEND()</code>	<code>MPI_IBSEND()</code>
Ready send	<code>MPI_RSEND()</code>	<code>MPI_IRSEND()</code>
Receive	<code>MPI_RECV()</code>	<code>MPI_IRecv()</code>

5 – Communication Modes

5.2 – Blocking call

Definition

- A call is blocking if the memory space used for the communication can be reused immediately after the exit of the call.
- The data sent can be modified after the call.
- The data received can be read after the call.

5 – Communication Modes

5.2 – Blocking call

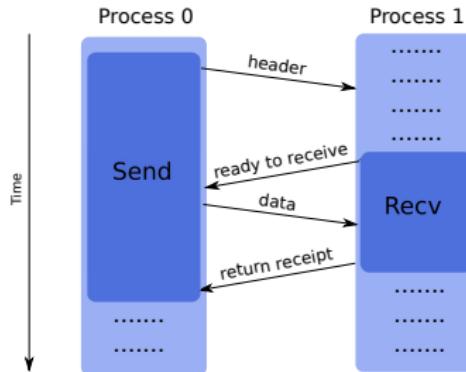
5.2.1 – Synchronous Sends

Definition

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

Rendezvous Protocol

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



Interfaces

```
TYPE(*), intent(in) :: values
integer, intent(in) :: count, type, dest, tag, comm
integer, intent(out) :: code

call MPI_SSEND(values, count, type, dest, tag, comm, code)
```

Advantages

- Low resource consumption (no buffer)
- Rapid if the receiver is ready (no copying in a buffer)
- Reception guaranteed through synchronization

Disadvantages

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

Deadlock example

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

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

5 – Communication Modes

5.2 – Blocking call

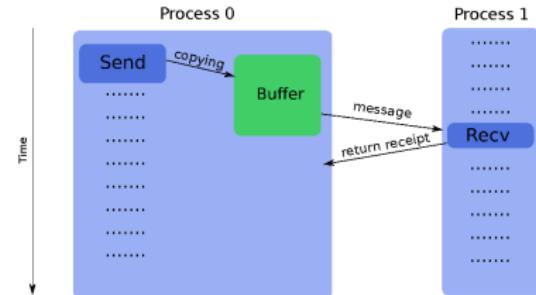
5.2.2 – Buffered sends

Definition

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

Protocol with user buffer on the sender side

In this approach, the buffer is on the sender side and is managed explicitly by the application. A buffer managed by MPI can exist on the receiver side. Many variants are possible. The return receipt is optional.



Buffered sends

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

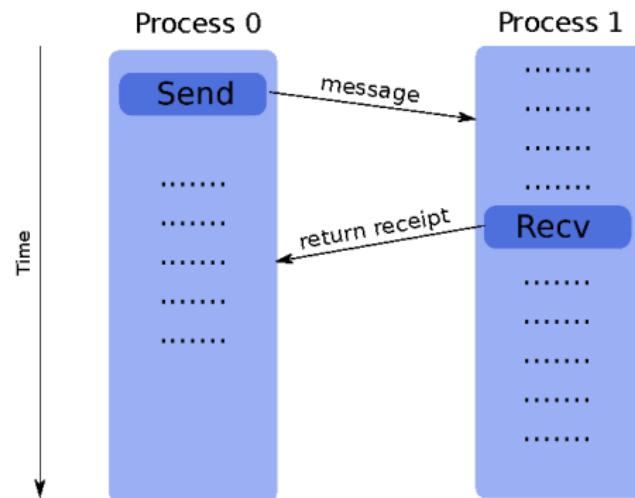
Interfaces

```
TYPE(*), intent(in)  :: values
integer, intent(in)  :: count, type, dest, tag, comm
integer, intent(out) :: code
TYPE(*) :: buf
integer :: size

call MPI_BSEND (values, count, type, dest, tag, comm, code)
call MPI_BUFFER_ATTACH (buf, size, code)
call MPI_BUFFER_DETACH (buf, size, code)
```

The eager protocol

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



Advantages

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

Disadvantages

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

No deadlocks

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

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

5 – Communication Modes

5.2 – Blocking call

5.2.3 – Standard sends

Standard sends

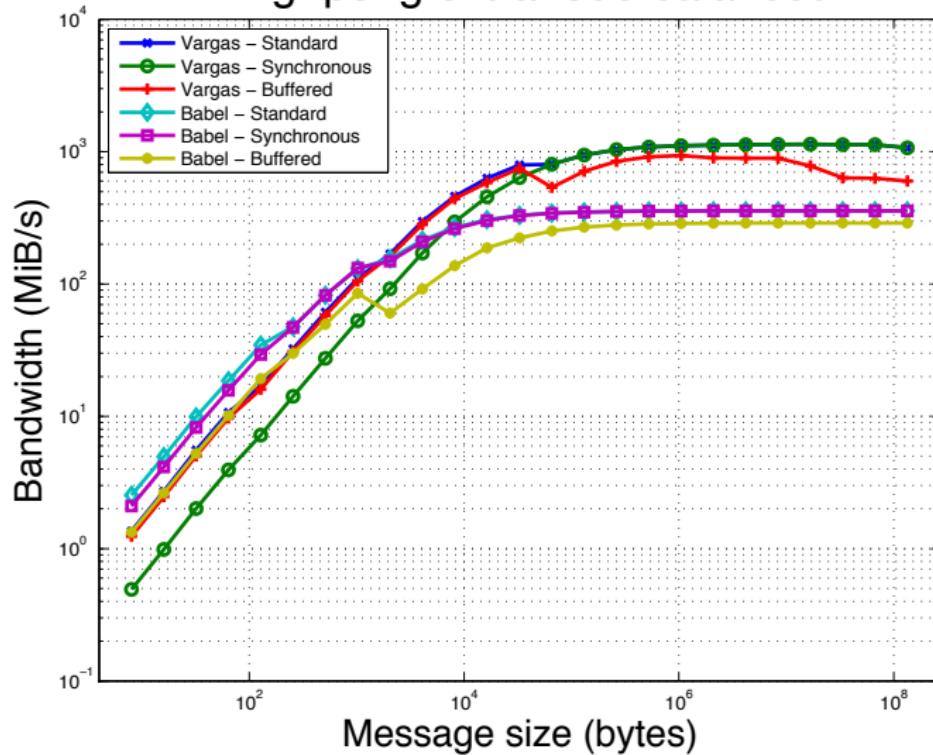
A standard send is made by calling the `MPI_SEND()` subroutine. In most implementations, the buffered mode switches to synchronous mode when the message size is large.

Interfaces

```
TYPE(*), intent(in) :: values
integer, intent(in) :: count, type, dest, tag, comm
integer, intent(out) :: code

call MPI_SEND(values, count, type, dest, tag, comm, code)
```

Ping–pong extranode balanced



Advantages

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

Disadvantages

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

5 – Communication Modes

5.2 – Blocking call

5.2.5 – Ready sends

Ready sends

A ready send is made by calling the `MPI_RSEND()` subroutine. Attention: it is obligatory to make these calls only when the receive is already posted. **Their use is strongly advised against.**

Advantages

- Slightly more efficient than the synchronous send because the ready send synchronization protocol can be simplified

Disadvantages

- Errors if the receiver is not ready during the send

5 – Communication Modes

5.3 – Nonblocking communication

Presentation

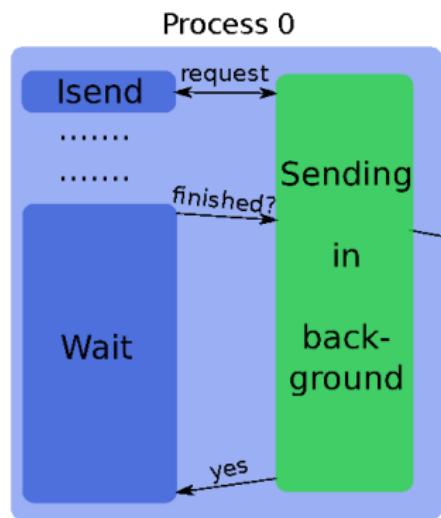
The overlap of communications by computations is a method which allows executing communications operations in the background while the program continues to operate. On Ada, the latency of a communication internode is $1.5 \mu\text{s}$, or 4000 processor cycles.

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

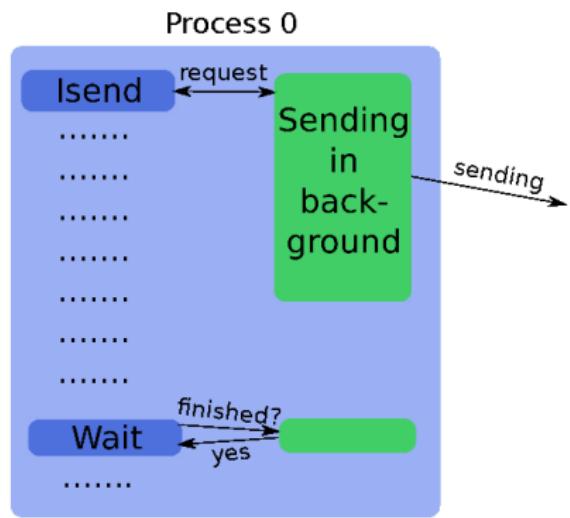
Definition

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

Partial overlap



Full overlap



Advantages

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

Disadvantages

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

Usage

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 for the end of the local contribution by making a call to `MPI_WAIT()` (or one of its variants).

Communications overlap with all the operations that occur between these two steps. Access to data being received is not permitted before the end of the `MPI_WAIT()`. (Access to data being sent is also not permitted for MPI implementations previous to the 2.2 version.)

Interfaces

MPI_ISEND() **MPI_ISSEND()** and **MPI_IBSEND()** for nonblocking send

```
TYPE(*), intent(in) :: values
integer, intent(in) :: count, datatype, dest, tag, comm
integer, intent(out) :: req, code

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

MPI_IRecv() for nonblocking receive.

```
TYPE(*), intent(in) :: values
integer, intent(in) :: count, type, source, tag, comm
integer, intent(out) :: req, code

call MPI_IRecv(values, count, type, source, tag, comm, req, code)
```

Interfaces

MPI_WAIT() wait for the end of a communication, **MPI_TEST()** is the nonblocking version.

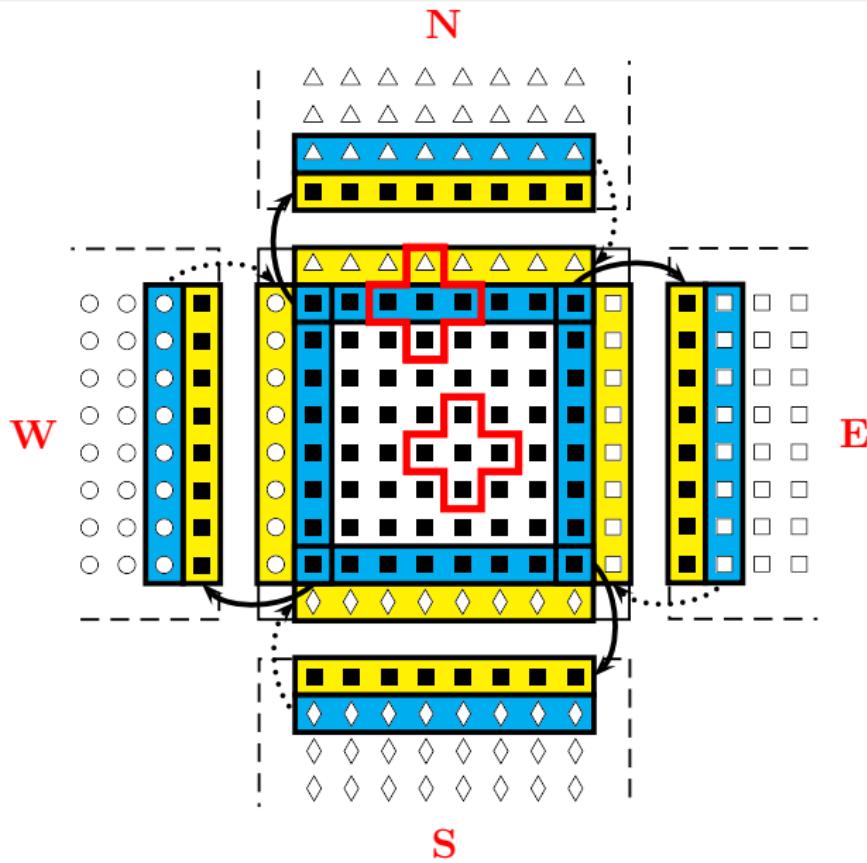
```
integer, intent(inout) :: req
integer, dimension(MPI_STATUS_SIZE), intent(out) :: statut
integer, intent(out) :: code
logical, intent(out) :: flag

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

MPI_WAITALL() (**MPI_TESTALL()**) await the end of all communications.

```
integer, intent(in) :: count
integer, dimension(count) :: reqs
integer, dimension(MPI_STATUS_SIZE,count), intent(out) :: statuts
integer, intent(out) :: code
logical, intent(out) :: flag

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



```
1 SUBROUTINE start_communication(u)
2   ! Send to the North and receive from the South
3   CALL MPI_IRecv( u(,), 1, rowtype, neighbor(S), &
4     tag, comm2d, request(1), code)
5   CALL MPI_Isend( u(,), 1, rowtype, neighbor(N), &
6     tag, comm2d, request(2), code)
7
8   ! Send to the South and receive from the North
9   CALL MPI_IRecv( u(,), 1, rowtype, neighbor(N), &
10    tag, comm2d, request(3), code)
11  CALL MPI_Isend( u(,), 1, rowtype, neighbor(S), &
12    tag, comm2d, request(4), code)
13
14  ! Send to the West and receive from the East
15  CALL MPI_IRecv( u(,), 1, columntype, neighbor(E), &
16    tag, comm2d, request(5), code)
17  CALL MPI_Isend( u(,), 1, columntype, neighbor(W), &
18    tag, comm2d, request(6), code)
19
20  ! Send to the East and receive from the West
21  CALL MPI_IRecv( u(,), 1, columntype, neighbor(W), &
22    tag, comm2d, request(7), code)
23  CALL MPI_Isend( u(,), 1, columntype, neighbor(E), &
24    tag, comm2d, request(8), code)
25 END SUBROUTINE start_communication
26 SUBROUTINE end_communication(u)
27   CALL MPI_Waitall(2*NB_NEIGHBORS,request,tab_status,code)
28 END SUBROUTINE end_communication
```

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

Overlap levels on different machines

<i>Machine</i>	<i>Level</i>
Blue Gene/Q, PAMID_THREAD_MULTIPLE=0	32%
Blue Gene/Q, PAMID_THREAD_MULTIPLE=1	100%
Ada, default	0%
Ada MP_USE_BULK_XFER=yes	100%

Measurements taken by overlapping a computational kernel with a communication kernel which have the same execution time and using different communication methods (intra/extr-nodes, by pairs, random processes, ...). The results can be totally different depending on the communication scenario used.

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

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

1	Introduction	
2	Environment	
3	Point-to-point Communications	
4	Collective communications	
5	Communication Modes	
6	Derived datatypes	
6.1	Introduction	107
6.2	Contiguous datatypes	109
6.3	Constant stride	110
6.4	Commit derived datatypes	112
6.5	Examples	113
6.6	Homogenous datatypes of variable strides	119
6.7	Subarray datatype constructor	125
6.8	Heterogenous datatypes	130
6.9	Size of MPI datatype	134
6.10	Conclusion	139
7	Communicators	
8	MPI-IO	
9	Conclusion	
10	Index	

6 – Derived datatypes

6.1 – Introduction

Introduction

- In communications, exchanged data have different datatypes: `MPI_INTEGER`, `MPI_REAL`, `MPI_COMPLEX`, etc.
- We can create more complex data structures by using subroutines such as `MPI_TYPE_CONTIGUOUS()`, `MPI_TYPE_VECTOR()`, `MPI_TYPE_INDEXED()` or `MPI_TYPE_CREATE_STRUCT()`
- Derived datatypes allow exchanging non-contiguous or non-homogenous data in the memory and limiting the number of calls to communications subroutines.

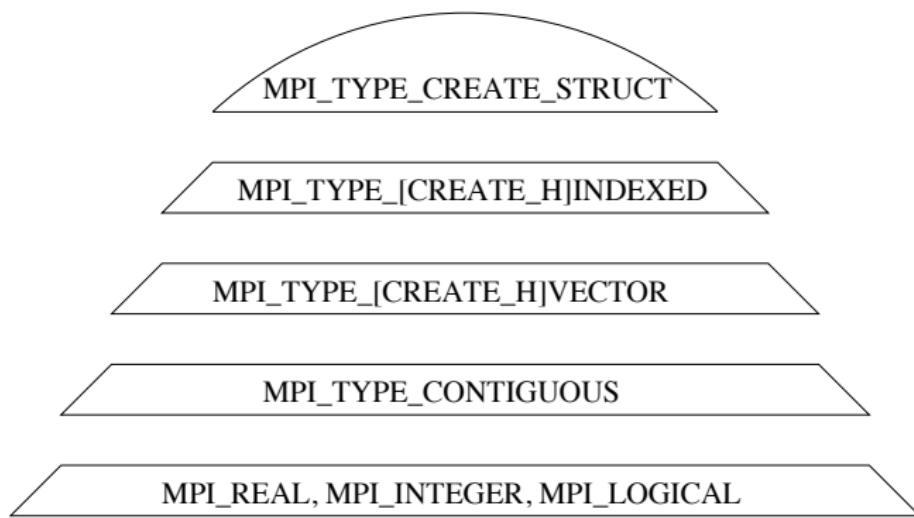


Figure 21: Hierarchy of the MPI constructors

6 – Derived datatypes

6.2 – Contiguous datatypes

Contiguous datatypes

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

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

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

Figure 22: MPI_TYPE_CONTIGUOUS subroutine

```
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

Constant stride

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

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

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

Figure 23: MPI_TYPE_VECTOR subroutine

```
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)
```

Constant stride

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

```
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)
```

6 – Derived datatypes

6.4 – Commit derived datatypes

Commit derived datatypes

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

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

call MPI_TYPE_COMMIT(new_type,code)
```

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

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

call MPI_TYPE_FREE(new_type,code)
```

6 – Derived datatypes

6.5 – Examples

6.5.1 – The datatype "matrix row"

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

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

6 – Derived datatypes

6.5 – Examples

6.5.2 – The datatype "matrix line"

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

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

6 – Derived datatypes

6.5 – Examples

6.5.3 – The datatype "matrix block"

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

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

6 – Derived datatypes

6.6 – Homogenous datatypes of variable strides

Homogenous datatypes of variable strides

- `MPI_TYPE_INDEXED()` allows creating a data structure composed of a sequence of blocks containing a variable number of elements separated by a variable stride in memory. The stride is given in number of **elements**.
- `MPI_TYPE_CREATE_HINDEXED()` has the same functionality as `MPI_TYPE_INDEXED()` except that the strides separating two data blocks are given in **bytes**.

This subroutine is useful when the old datatype is not an MPI base datatype(`MPI_INTEGER`, `MPI_REAL`, ...). We cannot therefore give the stride in number of elements of the old datatype.

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

`nb=3, blocks_lengths=(2,1,3), displacements=(0,3,7)`

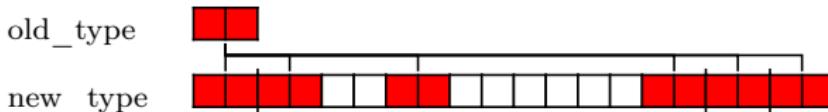


Figure 24: The `MPI_TYPE_INDEXED` constructor

```

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=4, blocks_lengths=(2,1,2,1), displacements=(2,10,14,24)`

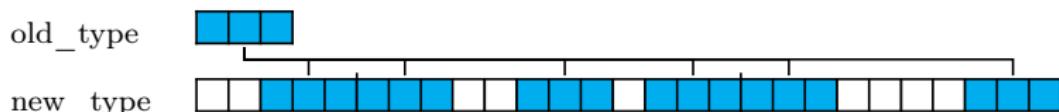


Figure 25: The `MPI_TYPE_CREATE_HINDEXED` constructor

```

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)

```

Example : triangular matrix

In the following example, each of the two processes :

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

										Before								After																																																																																																																																																																																																																																																																																							
Process 0										<table border="1"> <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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>								1	9	17	25	33	41	49	57										2	10	18	26	34	42	50	58										3	11	19	27	35	43	51	59										4	12	20	28	36	44	52	60										5	13	21	29	37	45	53	61										6	14	22	30	38	46	54	62										7	15	23	31	39	47	55	63										8	16	24	32	40	48	56	64										<table border="1"> <tr><td>1</td><td>-2</td><td>-3</td><td>-5</td><td>-8</td><td>-14</td><td>-22</td><td>-32</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>2</td><td>10</td><td>-4</td><td>-6</td><td>-11</td><td>-15</td><td>-23</td><td>-38</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>3</td><td>11</td><td>19</td><td>-7</td><td>-12</td><td>-16</td><td>-24</td><td>-39</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>4</td><td>12</td><td>20</td><td>28</td><td>-13</td><td>-20</td><td>-29</td><td>-40</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>5</td><td>13</td><td>21</td><td>29</td><td>37</td><td>-21</td><td>-30</td><td>-47</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>6</td><td>14</td><td>22</td><td>30</td><td>38</td><td>46</td><td>-31</td><td>-48</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></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>-56</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>								1	-2	-3	-5	-8	-14	-22	-32										2	10	-4	-6	-11	-15	-23	-38										3	11	19	-7	-12	-16	-24	-39										4	12	20	28	-13	-20	-29	-40										5	13	21	29	37	-21	-30	-47										6	14	22	30	38	46	-31	-48										7	15	23	31	39	47	55	-56										8	16	24	32	40	48	56	64									
1	9	17	25	33	41	49	57																																																																																																																																																																																																																																																																																																		
2	10	18	26	34	42	50	58																																																																																																																																																																																																																																																																																																		
3	11	19	27	35	43	51	59																																																																																																																																																																																																																																																																																																		
4	12	20	28	36	44	52	60																																																																																																																																																																																																																																																																																																		
5	13	21	29	37	45	53	61																																																																																																																																																																																																																																																																																																		
6	14	22	30	38	46	54	62																																																																																																																																																																																																																																																																																																		
7	15	23	31	39	47	55	63																																																																																																																																																																																																																																																																																																		
8	16	24	32	40	48	56	64																																																																																																																																																																																																																																																																																																		
1	-2	-3	-5	-8	-14	-22	-32																																																																																																																																																																																																																																																																																																		
2	10	-4	-6	-11	-15	-23	-38																																																																																																																																																																																																																																																																																																		
3	11	19	-7	-12	-16	-24	-39																																																																																																																																																																																																																																																																																																		
4	12	20	28	-13	-20	-29	-40																																																																																																																																																																																																																																																																																																		
5	13	21	29	37	-21	-30	-47																																																																																																																																																																																																																																																																																																		
6	14	22	30	38	46	-31	-48																																																																																																																																																																																																																																																																																																		
7	15	23	31	39	47	55	-56																																																																																																																																																																																																																																																																																																		
8	16	24	32	40	48	56	64																																																																																																																																																																																																																																																																																																		
Process 1										<table border="1"> <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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></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><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>								-1	-9	-17	-25	-33	-41	-49	-57										-2	-10	-18	-26	-34	-42	-50	-58										-3	-11	-19	-27	-35	-43	-51	-59										-4	-12	-20	-28	-36	-44	-52	-60										-5	-13	-21	-29	-37	-45	-53	-61										-6	-14	-22	-30	-38	-46	-54	-62										-7	-15	-23	-31	-39	-47	-55	-63										-8	-16	-24	-32	-40	-48	-56	-64										<table border="1"> <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><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>9</td><td>-10</td><td>-18</td><td>-26</td><td>-34</td><td>-42</td><td>-50</td><td>-58</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>17</td><td>34</td><td>-19</td><td>-27</td><td>-35</td><td>-43</td><td>-51</td><td>-59</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>18</td><td>35</td><td>44</td><td>-28</td><td>-36</td><td>-44</td><td>-52</td><td>-60</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>25</td><td>36</td><td>45</td><td>52</td><td>-37</td><td>-45</td><td>-53</td><td>-61</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>26</td><td>41</td><td>49</td><td>53</td><td>58</td><td>-46</td><td>-54</td><td>-62</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>27</td><td>42</td><td>50</td><td>54</td><td>59</td><td>61</td><td>-55</td><td>-63</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>33</td><td>43</td><td>51</td><td>57</td><td>60</td><td>62</td><td>63</td><td>-64</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>								-1	-9	-17	-25	-33	-41	-49	-57										9	-10	-18	-26	-34	-42	-50	-58										17	34	-19	-27	-35	-43	-51	-59										18	35	44	-28	-36	-44	-52	-60										25	36	45	52	-37	-45	-53	-61										26	41	49	53	58	-46	-54	-62										27	42	50	54	59	61	-55	-63										33	43	51	57	60	62	63	-64									
-1	-9	-17	-25	-33	-41	-49	-57																																																																																																																																																																																																																																																																																																		
-2	-10	-18	-26	-34	-42	-50	-58																																																																																																																																																																																																																																																																																																		
-3	-11	-19	-27	-35	-43	-51	-59																																																																																																																																																																																																																																																																																																		
-4	-12	-20	-28	-36	-44	-52	-60																																																																																																																																																																																																																																																																																																		
-5	-13	-21	-29	-37	-45	-53	-61																																																																																																																																																																																																																																																																																																		
-6	-14	-22	-30	-38	-46	-54	-62																																																																																																																																																																																																																																																																																																		
-7	-15	-23	-31	-39	-47	-55	-63																																																																																																																																																																																																																																																																																																		
-8	-16	-24	-32	-40	-48	-56	-64																																																																																																																																																																																																																																																																																																		
-1	-9	-17	-25	-33	-41	-49	-57																																																																																																																																																																																																																																																																																																		
9	-10	-18	-26	-34	-42	-50	-58																																																																																																																																																																																																																																																																																																		
17	34	-19	-27	-35	-43	-51	-59																																																																																																																																																																																																																																																																																																		
18	35	44	-28	-36	-44	-52	-60																																																																																																																																																																																																																																																																																																		
25	36	45	52	-37	-45	-53	-61																																																																																																																																																																																																																																																																																																		
26	41	49	53	58	-46	-54	-62																																																																																																																																																																																																																																																																																																		
27	42	50	54	59	61	-55	-63																																																																																																																																																																																																																																																																																																		
33	43	51	57	60	62	63	-64																																																																																																																																																																																																																																																																																																		

Figure 26: Exchange between the two processes

```

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

```

6 – Derived datatypes

6.7 – Subarray datatype constructor

Subarray datatype constructor

The `MPI_TYPE_CREATE_SUBARRAY()` subroutine allows creating a subarray from an array.

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 the number of elements in one dimension.
- The **shape** of an array is a vector for which each dimension equals the **extent**.

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**; so its shape is the **(10,6,21)** vector.

```
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)
```

Explanation of the arguments

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

BEFORE			AFTER		
1	5	9	1	5	9
2	6	10	-7	-11	10
3	7	11	-8	-12	11
4	8	12	4	8	12
Process 0			Process 0		
-1	-5	-9	-1	-5	-9
-2	-6	-10	-2	-6	-10
-3	-7	-11	-3	2	6
-4	-8	-12	-4	3	7
Process 1			Process 1		

Figure 27: Exchanges between the two processes

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

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

6 – Derived datatypes

6.8 – Heterogenous datatypes

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

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

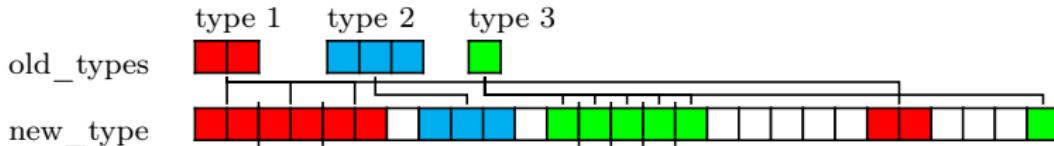


Figure 28: The `MPI_TYPE_CREATE_STRUCT` constructor

```
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)
```

Compute the displacement between two values

- **MPI_TYPE_CREATE_STRUCT()** is useful for creating MPI datatypes corresponding to Fortran derived datatypes or to C structures.
- The memory alignment of heterogeneous data structures is different for each architecture and each compiler.
- The displacement between two components of a Fortan derived datatype (or of a C structure) can be obtained by calculating the difference between their memory addresses.
- **MPI_GET_ADDRESS()** provides the address of a variable. It's equivalent of & operator in C.
- Warning, even in C, it is better to use this subroutine for portability reasons.

```
<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)
```

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

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

6 – Derived datatypes

6.9 – Size of MPI datatype

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

```
integer, intent(in) :: type_data
integer, intent(out) :: size, code

call MPI_TYPE_SIZE(datatype,size,code)
```

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

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

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

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

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

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

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

MPI datatype :



Two successive elements : `1. 2. 3. 4. 5. 6. 7. 8. 9. 10.`

`size = 12` (3 integers); `lower bound = 4` (1 integer); `extent = 16` (4 integers)

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

2D View :

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

1D View :



`size = 12` (3 integers); `lower bound = 0`; `extent = 44` (11 integers)

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

```

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

```

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

A matrix on the process 1

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

6 – Derived datatypes

6.10 – Conclusion

Conclusion

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

1	Introduction	
2	Environment	
3	Point-to-point Communications	
4	Collective communications	
5	Communication Modes	
6	Derived datatypes	
7	Communicators	
7.1	Introduction	141
7.2	Example	142
7.3	Default communicator	143
7.4	Groups and communicators	144
7.5	Partitioning of a communicator	145
7.6	Communicator built from a group	149
7.7	Topologies	150
8	MPI-IO	
9	Conclusion	
10	Index	

7 – Communicators

7.1 – Introduction

Introduction

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

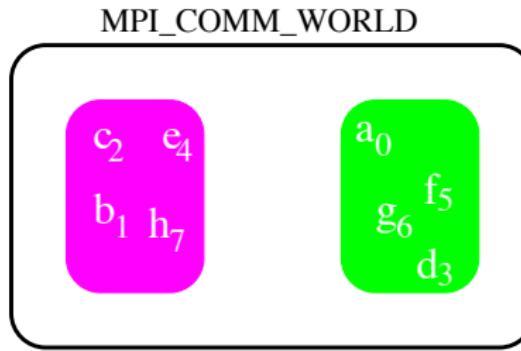


Figure 29: Communicator partitioning

7 – Communicators

7.2 – Example

Example

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

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

7 – Communicators

7.3 – Default communicator

Default communicator

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

7 – Communicators

7.4 – Groups and communicators

Groups and communicators

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

Partitioning of a communicator

In order to solve the problem example:

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

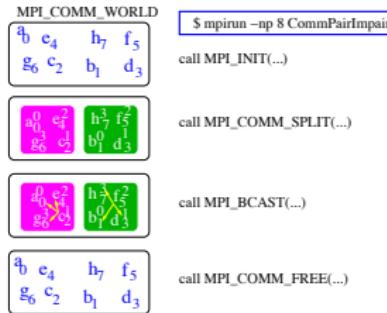


Figure 30: Communicator creation/destruction

Partitioning of a communicator with `MPI_COMM_SPLIT()`

The `MPI_COMM_SPLIT()` subroutine allows:

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

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

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

Example

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

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

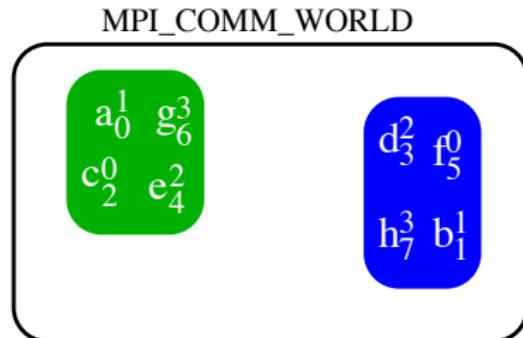


Figure 31: Construction of the `CommEvenOdd` communicator with `MPI_COMM_SPLIT()`

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

7 – Communicators

7.6 – Communicator built from a group

Communicator built from a group

- We can also build a communicator by defining a group of processes:
Call to `MPI_COMM_GROUP()`, `MPI_GROUP_INCL()`, `MPI_COMM_CREATE()`,
`MPI_GROUP_FREE()`
- For the example case, this method presents many disadvantages because it requires:
 - Naming the two communicators differently (for example `comm_even` and `comm_odd`).
 - Going into the groups to build these two communicators.
 - Leaving MPI to determine the process rank order.
 - Testing the validity of the communicator.

7 – Communicators

7.7 – Topologies

Topologies

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

7 – Communicators

7.7 – Topologies

7.7.1 – Cartesian topologies

Cartesian topologies

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

```
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,reorder,comm_new,code)
```

Example

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

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

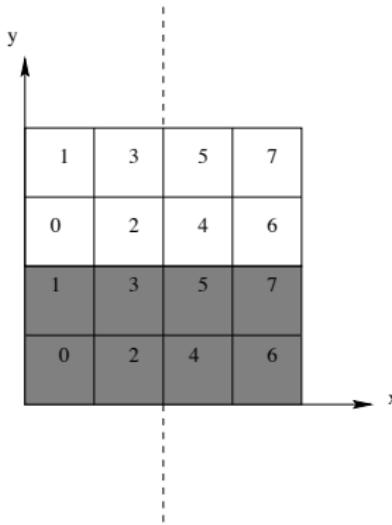


Figure 32: A 2D periodic Cartesian topology in y

3D Example

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

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

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

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

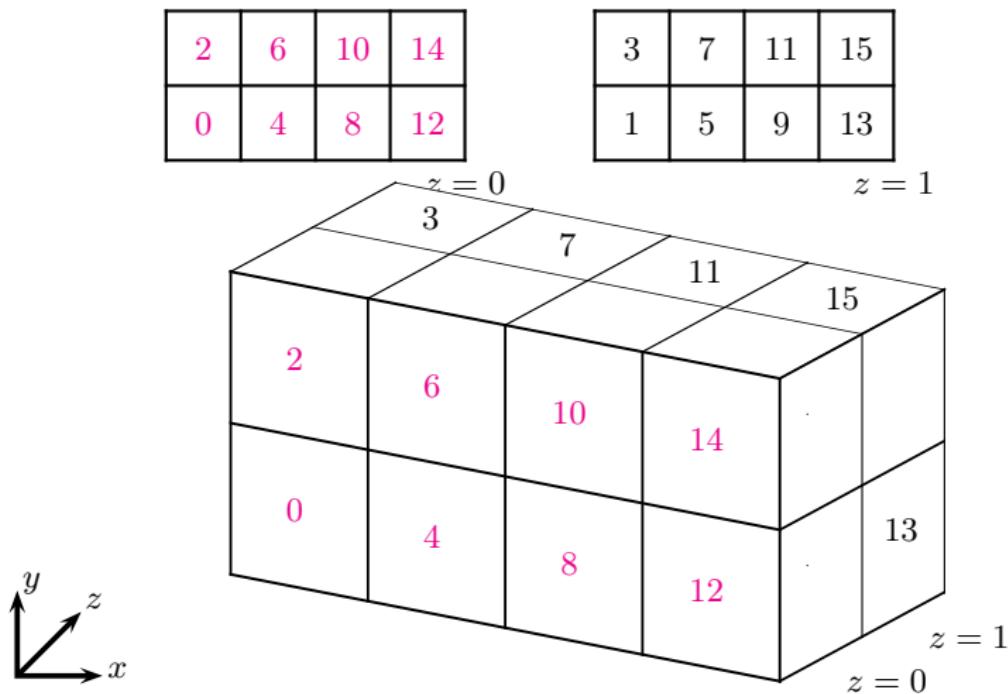


Figure 33: A 3D non-periodic Cartesian topology

Process distribution

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

```
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, then we leave to MPI the choice of the number of processes in each direction according to the total number of processes.

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

Rank of a process

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

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

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

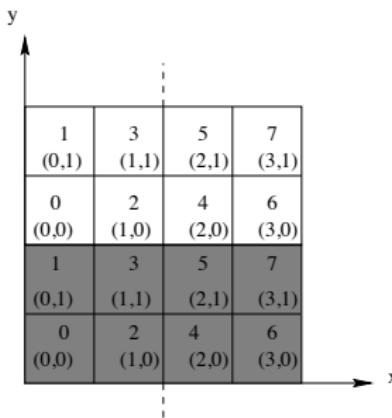


Figure 34: A 2D periodic Cartesian topology in y

```

coords(1)=dims(1)-1
do i=0,dims(2)-1
  coords(2) = i
  call MPI_CART_RANK(comm_2D,coords,rank(i),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.

```

Coordinates of a process

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

```
integer, intent(in)          :: comm, rank, ndims  
integer, dimension(ndims),intent(out) :: coords  
integer, intent(out)           :: code  
  
call MPI_CART_COORDS(comm, rank, ndims, coords, code)
```

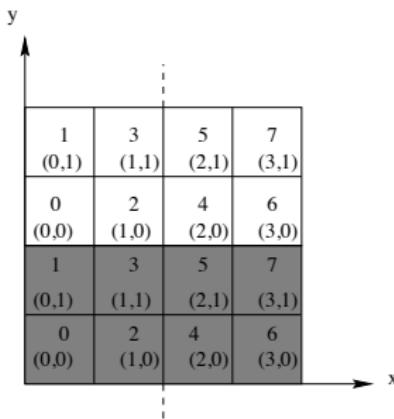


Figure 35: A 2D periodic Cartesian topology in y

```

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)

```

Rank of neighbours

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

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

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

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

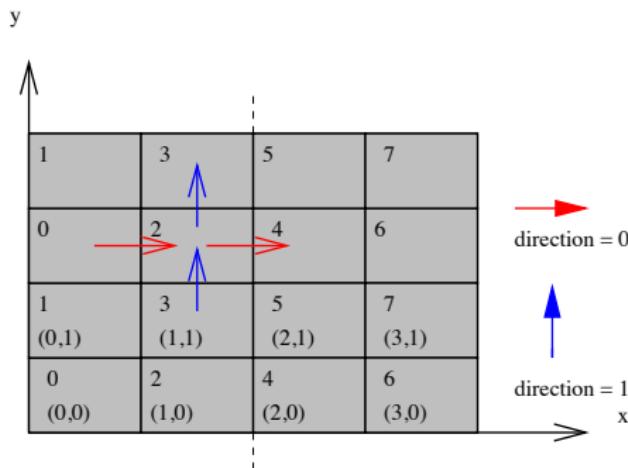


Figure 36: Call of the MPI_CART_SHIFT() subroutine

```
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
```

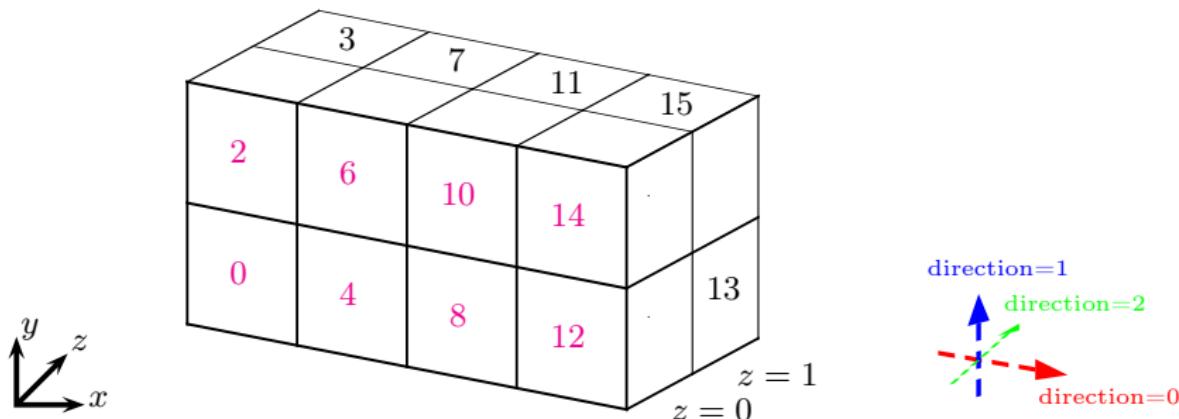


Figure 37: Call of the MPI_CART_SHIFT() subroutine

```
call MPI_CART_SHIFT(comm_3D,0,1,rank_left,rank_right,code)
```

.....
For the process 0, rank_left=-1, rank_right=4

```
call MPI_CART_SHIFT(comm_3D,1,1,rank_low,rank_high,code)
```

.....
For the process 0, rank_low=-1, rank_high=2

```
call MPI_CART_SHIFT(comm_3D,2,1,rank_ahead,rank_before,code)
```

.....
For the process 0, rank_ahead=-1, rank_before=1

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

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

7 – Communicators

7.7 – Topologies

7.7.2 – Subdividing a Cartesian topology

Subdividing a Cartesian topology

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

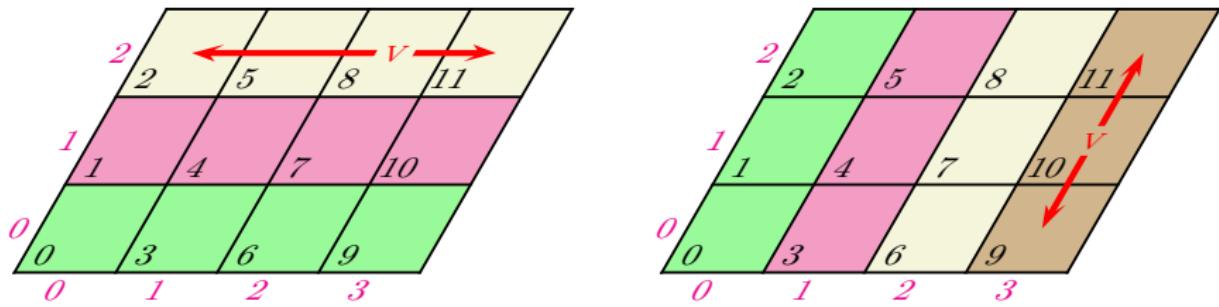


Figure 38: Two examples of data distribution in a degenerated 2D topology

Subdividing a Cartesian topology

There are two ways to degenerate a topology :

- By using the **MPI_COMM_SPLIT()** general subroutine
- By using the **MPI_CART_SUB()** subroutine designed for this purpose

```
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)
```

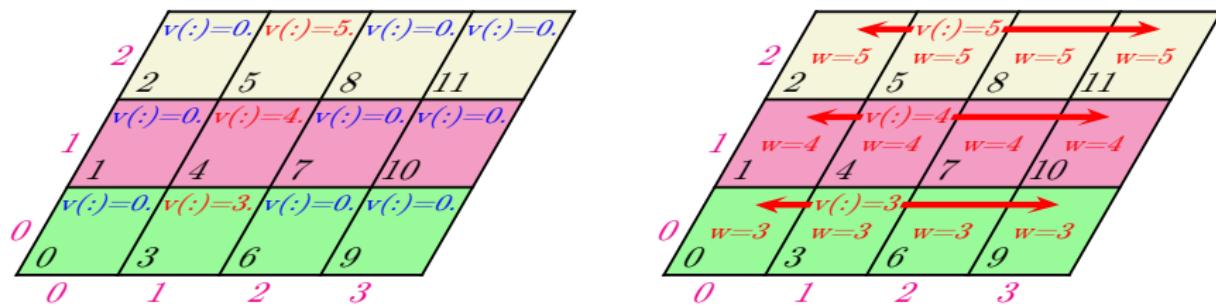


Figure 39: Broadcast of a V array in the degenerated 2D grid.

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

```
13 call MPI_INIT(code)
14
15 ! Creation of the initial 2D grid
16 Dim2D(1) = 4
17 Dim2D(2) = 3
18 Period(:) = .false.
19 ReOrder = .false.
20 call MPI_CART_CREATE(MPI_COMM_WORLD,NDim2D,Dim2D,Period,ReOrder,Comm2D,code)
21 call MPI_COMM_RANK(Comm2D,rank,code)
22 call MPI_CART_COORDS(Comm2D,rank,NDim2D,Coord2D,code)
23
24 ! Initialization of the V vector
25 if (Coord2D(1) == 1) V(:)=real(rank)
26
27 ! Every row of the grid must be a 1D cartesian topology
28 remain_dims(1) = .true.
29 remain_dims(2) = .false.
30 ! Subdivision of the 2D cartesian grid
31 call MPI_CART_SUB(Comm2D,remain_dims,Comm1D,code)
32
33 ! The processes of column 2 distribute the V vector to the processes of their row
34 call MPI_SCATTER(V,1,MPI_REAL,W,1,MPI_REAL,1,Comm1D,code)
35
36 print '("Rank : ",I2," ; Coordinates : (",I1,",",I1,") ; W = ",F2.0)', &
37 rank,Coord2D(1),Coord2D(2),W
38
39 call MPI_FINALIZE(code)
40 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	Communication Modes	
6	Derived datatypes	
7	Communicators	
8	MPI-IO	
8.1	Introduction	173
8.2	File Manipulation	178
8.3	Data access: Concepts	181
8.4	Noncollective data access	185
8.5	Collective data access	198
8.6	Positioning the file pointers	209
8.7	File Views	212
8.8	Nonblocking Data Access	226
8.9	Advice	235
8.10	Definitions	236
9	Conclusion	
10	Index	

8 – MPI-IO

8.1 – Introduction

Input/Output Optimisation

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

The MPI-IO interface

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

Challenges

- The goal of **MPI-IO**, via the high-level interface that it proposes, is to provide **simplicity**, **expressiveness** and **flexibility** to the user, while authorising **optimised** implementations which take into account the software and hardware specificities of I/O devices on the target machines.
- It is the user's responsibility to leverage some of the optimisations enabled by the library (ex: **collective** or **non-blocking** I/O).
- But many other essential optimisations implemented **in the libraries** improve performances automatically.

Example of a sequential optimisation implemented by I/O libraries

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

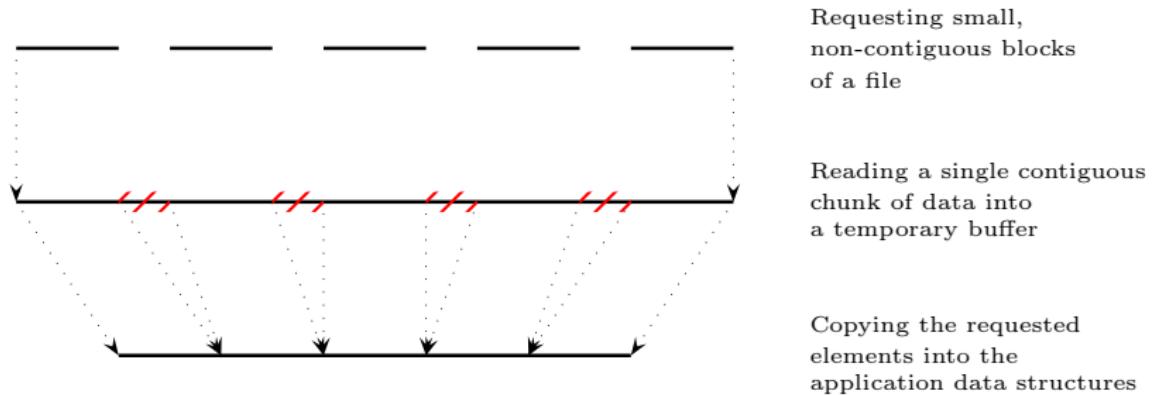


Figure 40: Data sieving mechanism improving I/O access on small, non-contiguous data set.

Example of a parallel optimisation

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

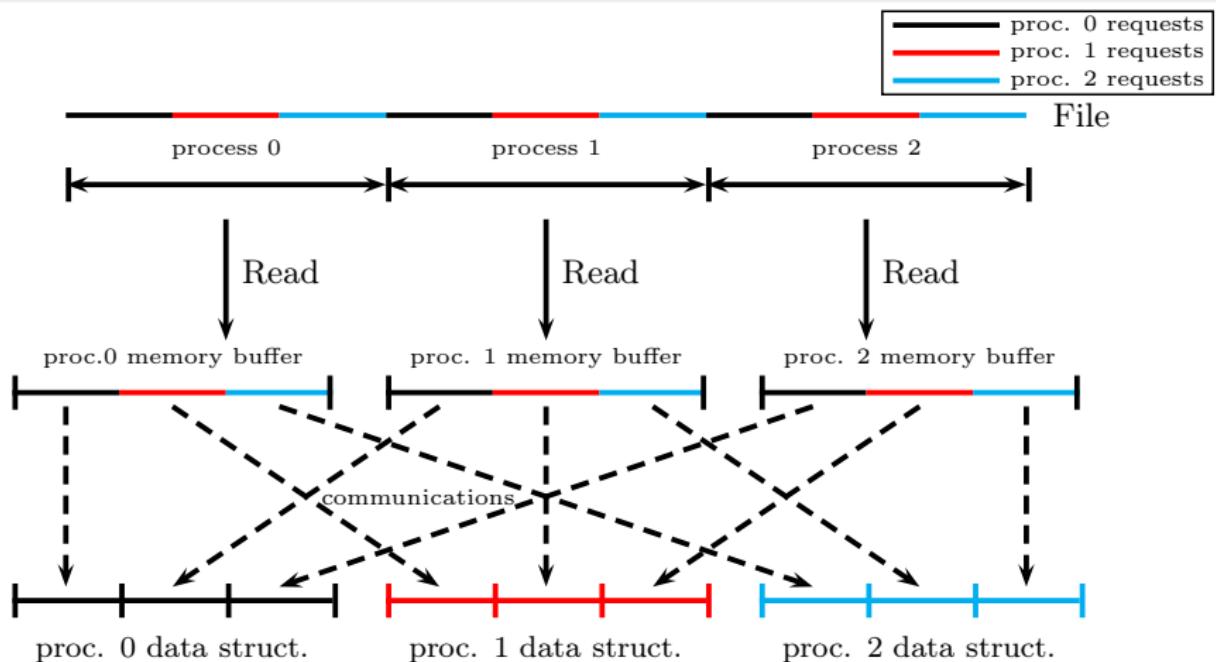


Figure 41: Read operation performed in two steps by a group of processes

8 – MPI-IO

8.2 – File Manipulation

Working with files

- Opening and closing files are **collective operations** within the scope of a communicator.
- Opening a file generates a **file handle**, an opaque representation of the opened file. File handles can be subsequently used to access files in MPI I/O subroutines.
- Access modes describe the opening mode, access rights, etc. Modes are specified at the opening of a file, using predefined MPI constants that can be combined together.
- All the processes of the communicator participate in subsequent collective operations.
- We are only describing here the open/close subroutines but others file management operations are available (preallocation, deletion, etc.). For instance, **MPI_FILE_GET_INFO()** returns details on a file handle (information varies with implementations).

```
1 program open01
2   use mpi
3   implicit none
4
5   integer :: fh,code
6
7   call MPI_INIT(code)
8
9   call MPI_FILE_OPEN(MPI_COMM_WORLD,"file.data", &
10                 MPI_MODE_RDWR + MPI_MODE_CREATE,MPI_INFO_NULL,fh,code)
11
12  call MPI_FILE_CLOSE(fh,code)
13  call MPI_FINALIZE(code)
14
15 end program open01
```

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

Table 4: Access modes which can be defined at the opening of files

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

8 – MPI-IO

8.3 – Data access: Concepts

Data access routines

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

Table 5: Summary of the data access subroutines

Position-ing	Synchro-nism	Coordination	
		<i>noncollective</i>	<i>collective</i>
explicit offsets	blocking	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	nonblocking	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END

see next page

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

File Views

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

8 – MPI-IO

8.4 – Noncollective data access

8.4.1 – Data access with explicit offsets

Explicit Offsets

- Explicit offset operations perform data access directly at the file **position**, given as an argument.
- The offset is expressed as a multiple of the **elementary data type** of the current view (therefore, the default offset unit is bytes).
- The datatype and the number of elements in the memory buffer are specified as arguments (ex: **MPI_INTEGER**)

```
1 program write_at
2 use mpi
3 implicit none
4
5 integer, parameter :: nb_values=10
6 integer :: i,rank,fh,code,bytes_in_integer
7 integer(kind=MPI_OFFSET_KIND) :: offset
8 integer, dimension(nb_values) :: values
9 integer, dimension(MPI_STATUS_SIZE) :: status
10
11 call MPI_INIT(code)
12 call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
13
14 values(:)= (/i+rank*100,i=1,nb_values/)
15 print *, "Write process",rank, ":",values(:)
16
17 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_WRONLY + MPI_MODE_CREATE, &
18 MPI_INFO_NULL,fh,code)
19
20 call MPI_TYPE_SIZE(MPI_INTEGER,bytes_in_integer,code)
21
22 offset=rank*nb_values*bytes_in_integer
23 call MPI_FILE_WRITE_AT(fh,offset,values,nb_values,MPI_INTEGER, &
24 status,code)
25
26 call MPI_FILE_CLOSE(fh,code)
27 call MPI_FINALIZE(code)
28 end program write_at
```

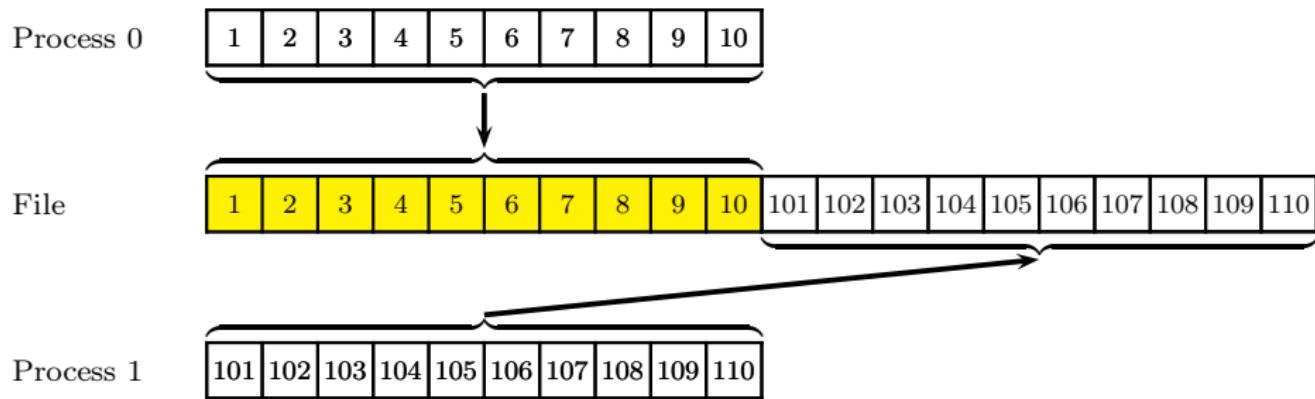


Figure 42: MPI_FILE_WRITE_AT()

```
> 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
```

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

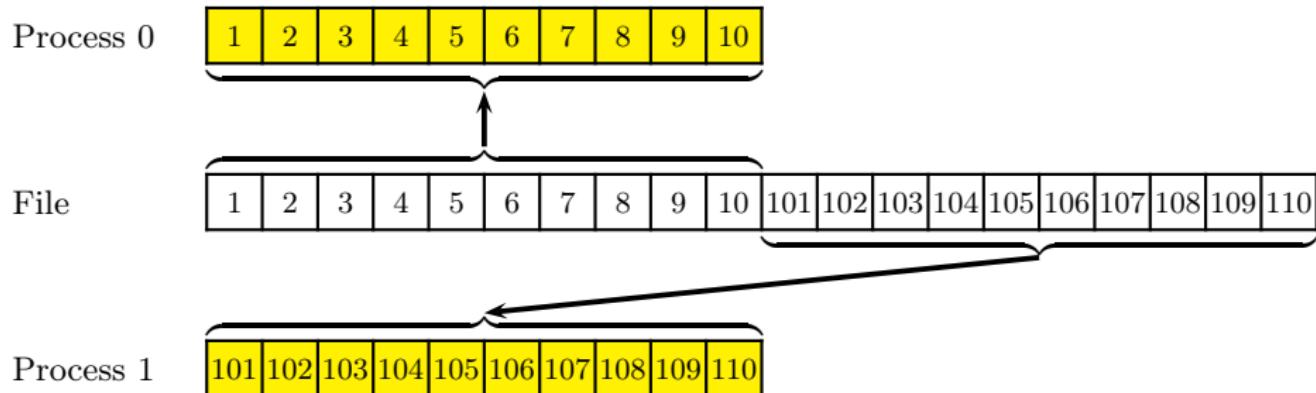


Figure 43: MPI_FILE_READ_AT()

```
> 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
```

8 – MPI-IO

8.4 – Noncollective data access

8.4.2 – Data access with individual file pointers

Individual file pointers

- MPI maintains **one** individual file pointer per **process** per **file handle**.
- The current value of this pointer implicitly specifies the offset in the data access routines.
- After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next data item.
- The shared file pointer is neither used nor updated.

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

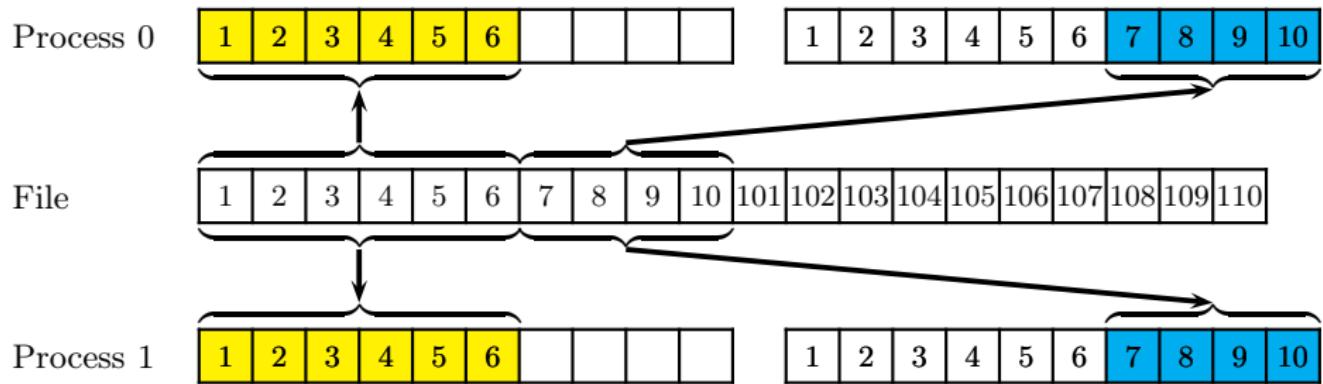


Figure 44: Example 1 of MPI_FILE_READ()

```
> 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
```

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

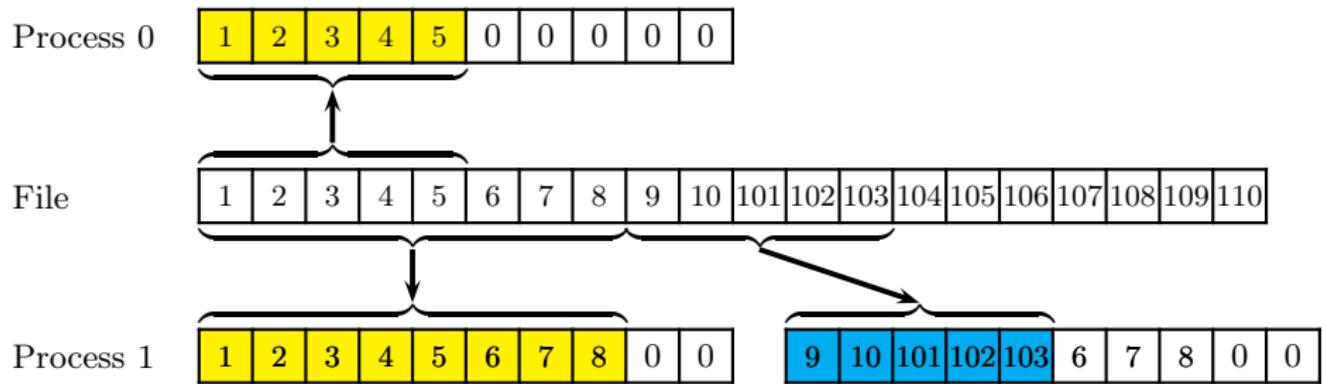


Figure 45: Example 2 of MPI_FILE_READ()

```
> 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
```

8 – MPI-IO

8.4 – Noncollective data access

8.4.3 – Data access with shared file pointers

Shared file pointer

- MPI maintains only one shared file pointer per collective `MPI_FILE_OPEN` (shared among processes in the communicator group).
- All processes must use the **same file view**.
- For the noncollective shared file pointer routines, the serialisation ordering **is not deterministic**. To enforce a specific order, the user needs to use other synchronisation means or use collective variants.
- After a shared file pointer operation, the shared file pointer is updated to point to the next data item, that is, just after the last one accessed by the operation.
- The individual file pointers are neither used nor updated.

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

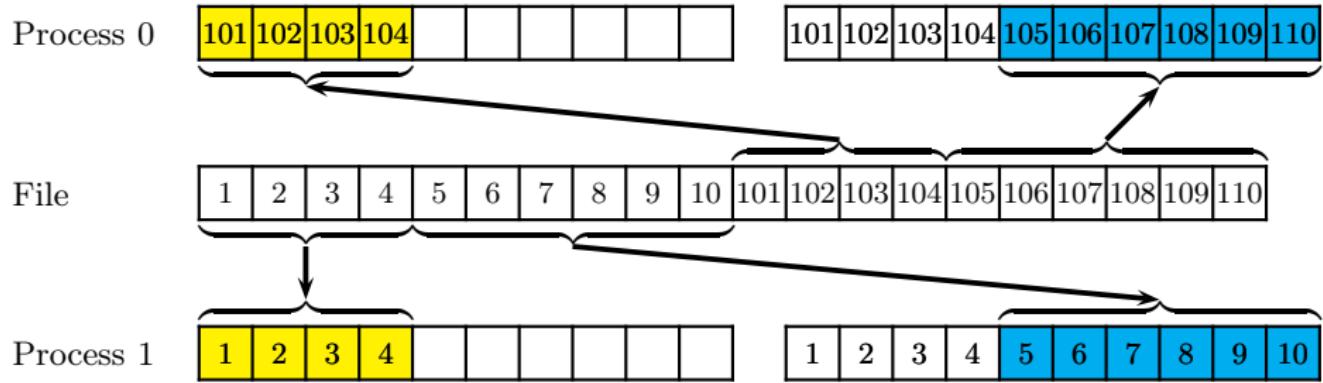


Figure 46: Example of MPI_FILE_READ_SHARED()

```
> 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
```

8 – MPI-IO

8.5 – Collective data access

Collective data access

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

8 – MPI-IO

8.5 – Collective data access

8.5.1 – Data access with explicit offsets

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

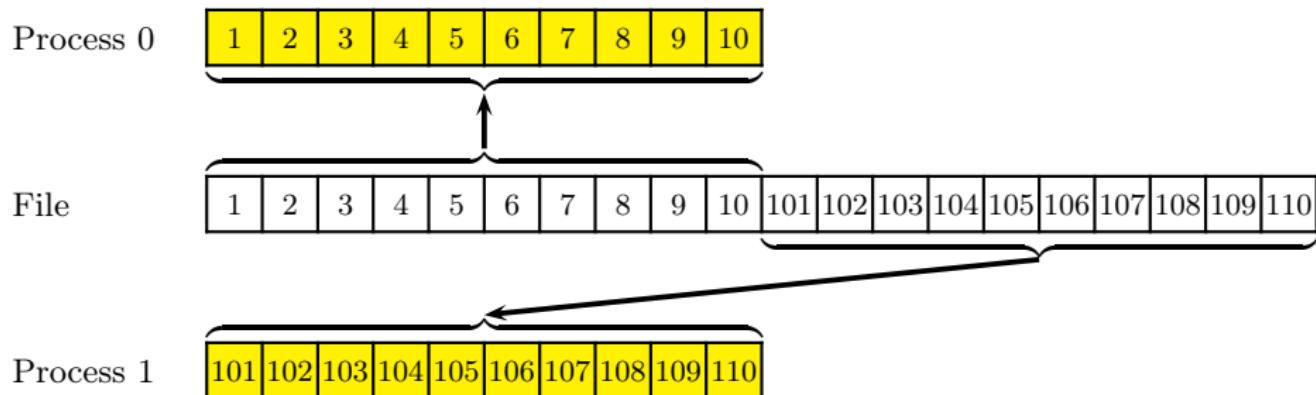


Figure 47: Example of MPI_FILE_READ_AT_ALL()

```
> 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
```

8 – MPI-IO

8.5 – Collective data access

8.5.2 – Data access with individual file pointers

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

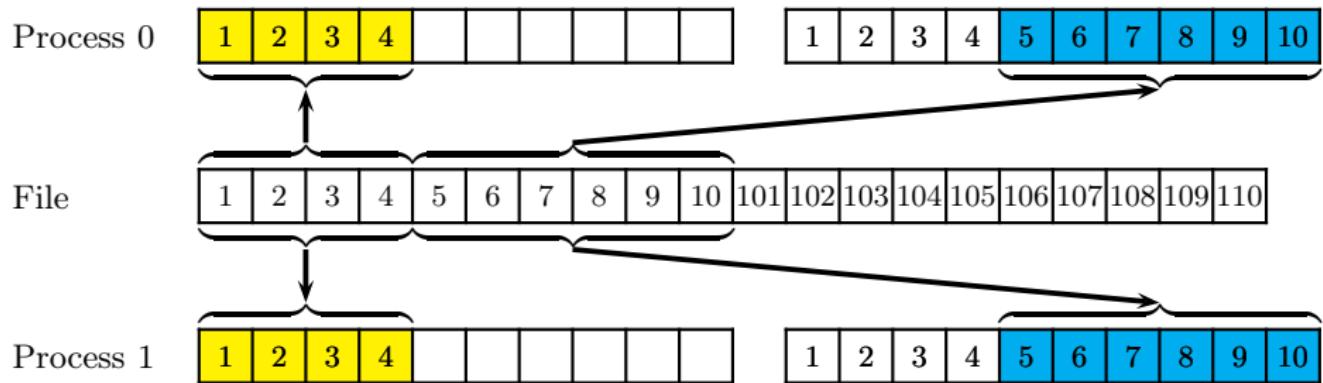


Figure 48: Example 1 of MPI_FILE_READ_ALL()

```
> 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
```

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

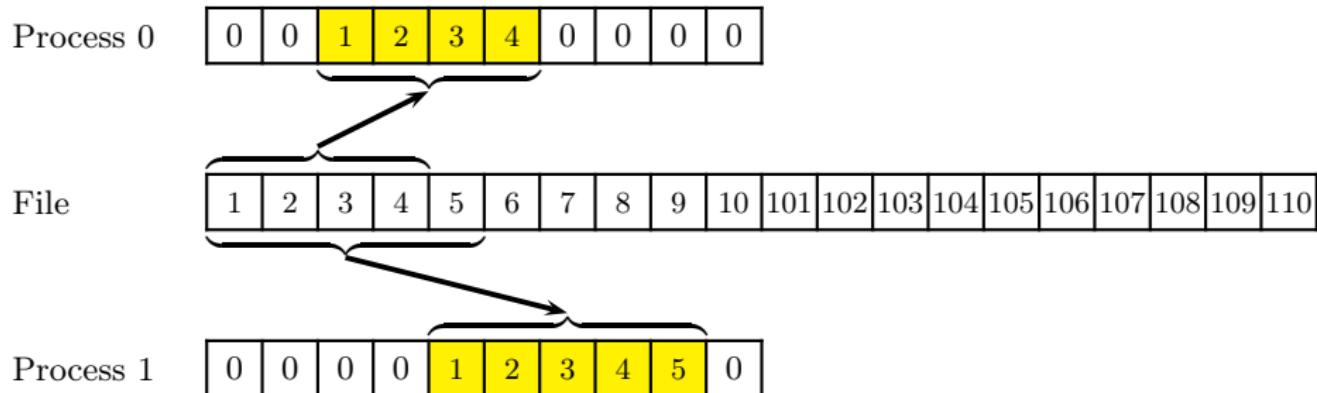


Figure 49: Example 2 of MPI_FILE_READ_ALL()

```
> 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
```

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

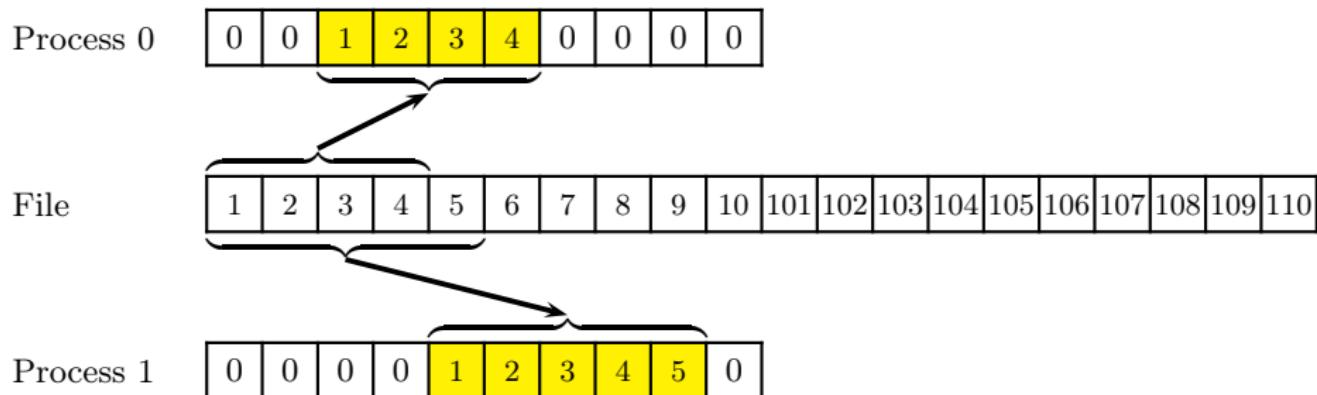


Figure 50: Example 3 of MPI_FILE_READ_ALL()

```
> 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
```

8 – MPI-IO

8.5 – Collective data access

8.5.3 – Data access with shared file pointers

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

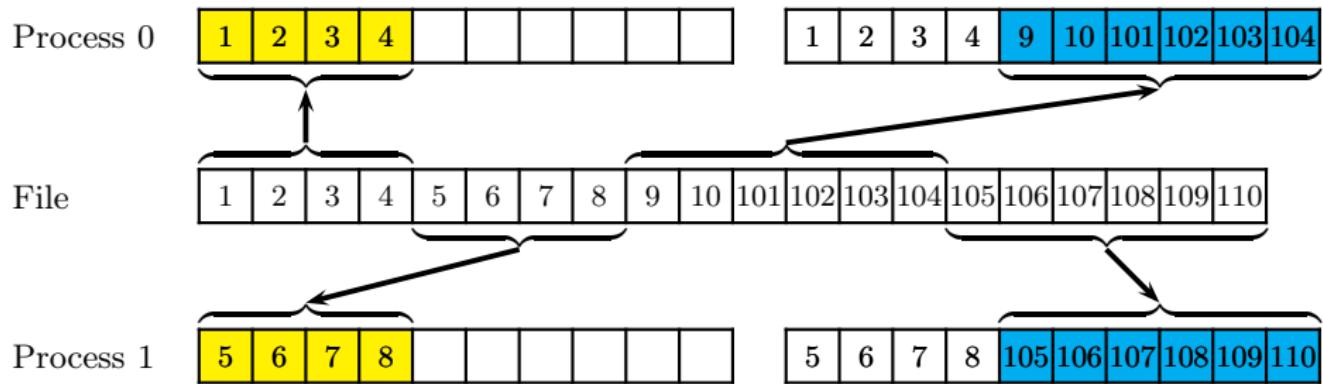


Figure 51: Example of MPI_FILE_ORDERED()

```
> 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
```

8 – MPI-IO

8.6 – Positioning the file pointers

Positioning the file pointers

- `MPI_FILE_GET_POSITION()` and `MPI_FILE_GET_POSITION_SHARED()` returns the current position of the individual pointers and the shared file pointer (respectively).
- `MPI_FILE_SEEK()` and `MPI_FILE_SEEK_SHARED()` updates the file pointer values by using the following possible modes:
 - `MPI_SEEK_SET`: The pointer is set to offset.
 - `MPI_SEEK_CUR`: The pointer is set to the current pointer position plus offset.
 - `MPI_SEEK_END`: The pointer is set to the end of file plus offset.
- The `offset` can be negative, which allows seeking backwards.

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

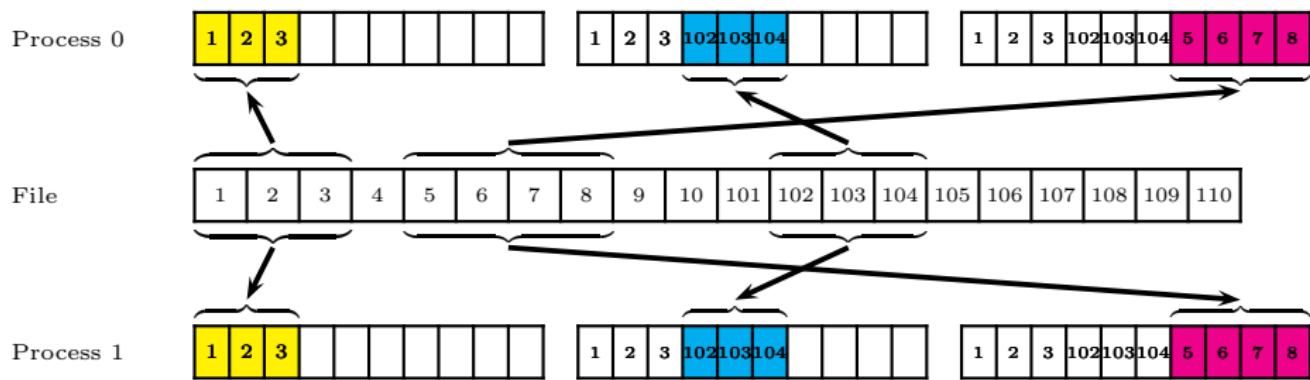


Figure 52: Example of MPI_FILE_SEEK()

```
> 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
```

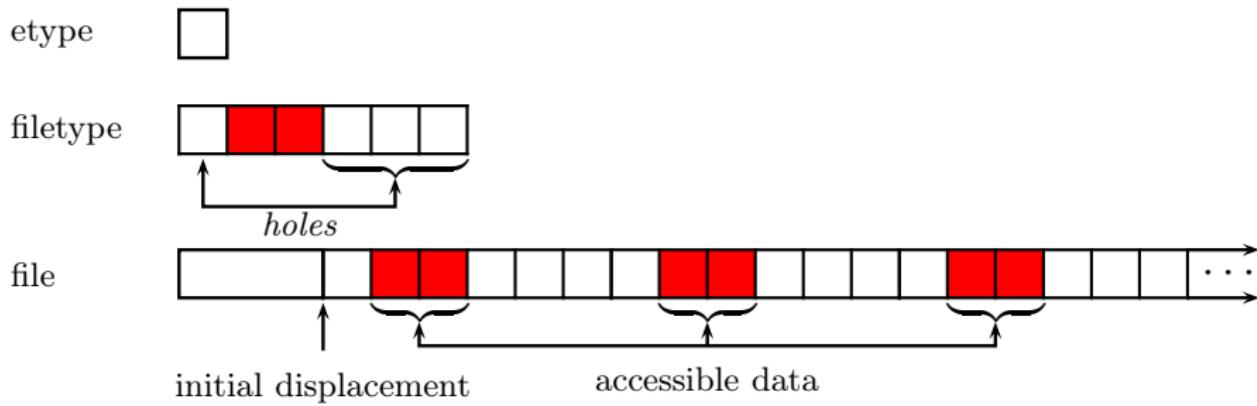
8 – MPI-IO

8.7 – File Views

8.7.1 – Definition

The View Mechanism

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



The View Mechanism

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

Multiple Views

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

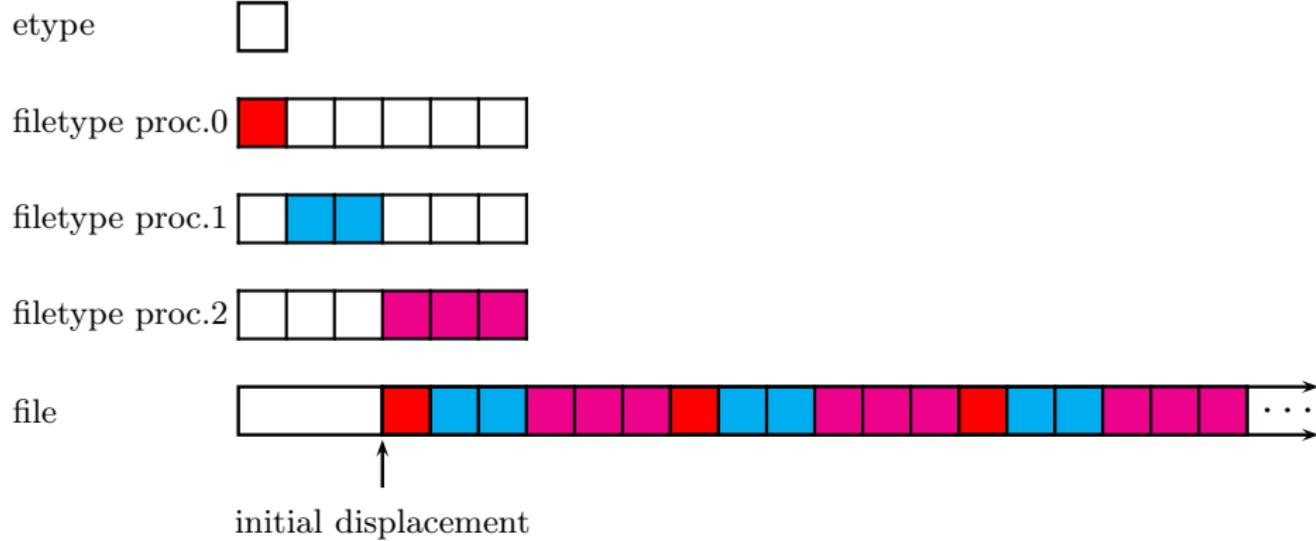


Figure 54: Separate views, each using a different filetype, can be used to access the file

Limitations:

- Shared file pointer routines are not useable except when all the processes have the same file view.
- If the file is opened for writing, neither the **etype** nor the **filetype** is permitted to contain overlapping regions.

Changing the process's view of the data in the file: **MPI_FILE_SET_VIEW()**

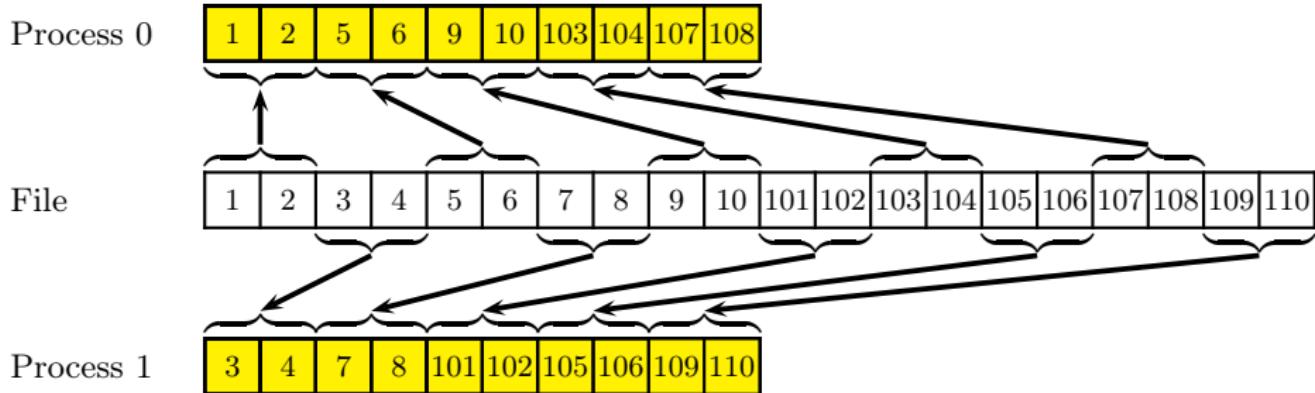
```
1 integer :: fh
2 integer(kind=MPI_OFFSET_KIND) :: displacement
3 integer :: etype
4 integer :: filetype
5 character(len=*) :: mode
6 integer :: info
7 integer :: code
8
9 call MPI_FILE_SET_VIEW(fh,
10                      displacement,etype,filetype,
11                      mode,info,code)
```

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

Notes :

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

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



```
> mpiexec -n 2 read_view01
```

```
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
```

Example 1 (continued)

init_disp

0

etype



filetype proc. 0



filetype proc. 1



```
1 if (rank == 0) coord=1
2 if (rank == 1) coord=3
3
4 call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/coord - 1/), &
5                               MPI_ORDER_FORTRAN,MPI_INTEGER,filetype,code)
6 call MPI_TYPE_COMMIT(filetype,code)
7
8 ! Using an intermediate variable for portability reasons
9 init_displacement=0
10
11 call MPI_FILE_SET_VIEW(handle,init_displacement,MPI_INTEGER,filetype, &
12                         "native",MPI_INFO_NULL,code)
```

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

Example 2: Reading data using successive views

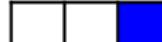
init_disp 0

etype  MPI_INTEGER

filetype_1 

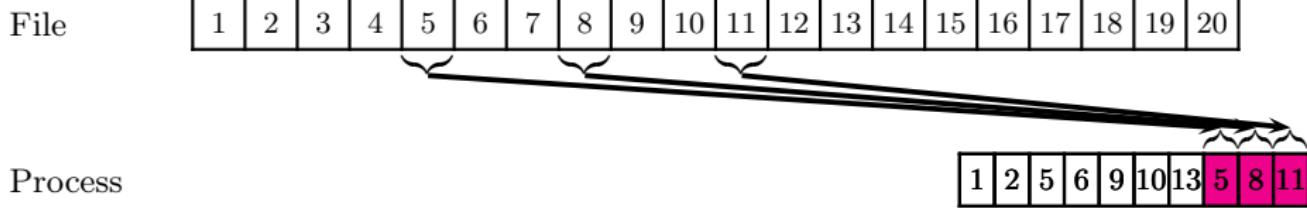
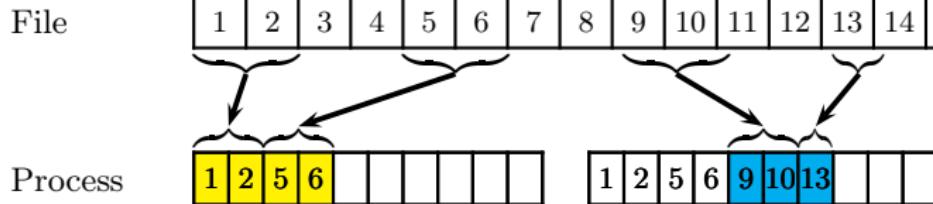
init_disp 2 integers

etype  MPI_INTEGER

filetype_2 

```
1 program read_view02
2
3 use mpi
4 implicit none
5
6 integer, parameter :: nb_values=10
7 integer :: rank,handle,code, &
8           filetype_1,filetype_2,nb_octets_entier
9 integer(kind=MPI_OFFSET_KIND) :: init_displacement
10 integer, dimension(nb_values) :: values
11 integer, dimension(MPI_STATUS_SIZE) :: status
12
13 call MPI_INIT(code)
14 call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
```

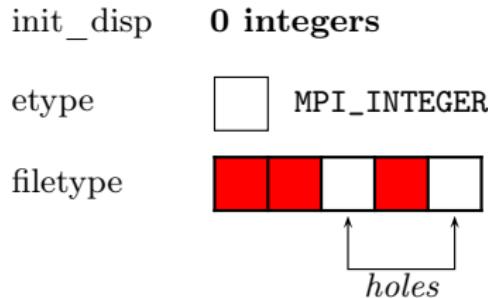
```
15 call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/0/), &
16                                MPI_ORDER_FORTRAN,MPI_INTEGER,filetype_1,code)
17 call MPI_TYPE_COMMIT(filetype_1,code)
18
19 call MPI_TYPE_CREATE_SUBARRAY(1,(/3/),(/1/),(/2/), &
20                                MPI_ORDER_FORTRAN,MPI_INTEGER,filetype_2,code)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



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

```
Read process 1 : 1, 2, 5, 6, 9, 10, 13, 5, 8, 11  
Read process 0 : 1, 2, 5, 6, 9, 10, 13, 5, 8, 11
```

Example 3: Dealing with holes in datatypes



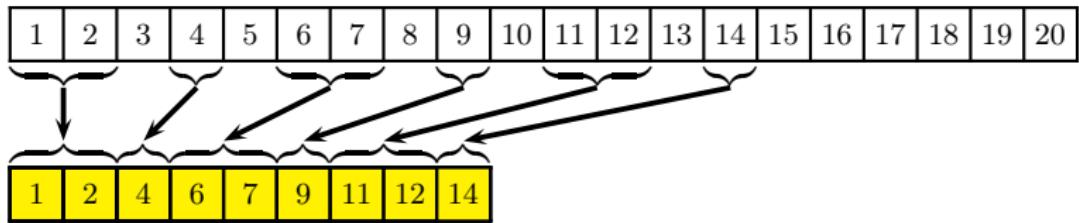
```

1 program read_view_03_indexed
2
3 use mpi
4 implicit none
5
6 integer, parameter :: nb_values=9
7 integer :: rank,handle,bytes_in_integer,code
8 integer :: filetype_tmp,filetype
9 integer(kind=MPI_OFFSET_KIND) :: init_displacement
10 integer(kind=MPI_ADDRESS_KIND) :: lb,extent
11 integer, dimension(2) :: blocklens,indices
12 integer, dimension(nb_values) :: values
13 integer, dimension(MPI_STATUS_SIZE) :: status
14
15 call MPI_INIT(code)
16 call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
17

```

```
18 ! filetype_tmp: MPI type with an extent of 4*MPI_INTEGER
19 indices(1)=0
20 blocklens(1)=2
21 indices(2)=3
22 blocklens(2)=1
23 call MPI_TYPE_INDEXED(2,blocklens,indices,MPI_INTEGER,filetype_tmp,code)
24
25 ! filetype: MPI type with an extent of 5*MPI_INTEGER
26 call MPI_TYPE_SIZE(MPI_INTEGER,bytes_in_integer,code)
27 call MPI_TYPE_GET_EXTENT(filetype_tmp,lb,extent,code)
28 extent = extent + bytes_in_integer
29 call MPI_TYPE_CREATE_RESIZED(filetype_tmp,lb,lb+extent,filetype,code)
30 call MPI_TYPE_COMMIT(filetype,code)
31
32 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
33                  handle,code)
34
35 init_displacement=0
36 call MPI_FILE_SET_VIEW(handle,init_displacement,MPI_INTEGER,filetype, &
37                        "native",MPI_INFO_NULL,code)
38
39 call MPI_FILE_READ(handle,values,9,MPI_INTEGER,status,code)
40
41 print *, "Read process",rank,":",values(:)
42
43 call MPI_FILE_CLOSE(handle,code)
44 call MPI_FINALIZE(code)
45
46 end program read_view03_indexed
```

File



Process

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

```
Lecture, processus 0 : 1, 2, 4, 6, 7, 9, 11, 12, 14
Lecture, processus 1 : 1, 2, 4, 6, 7, 9, 11, 12, 14
```

Example 3 (cont.): Alternative implementation using a structure type

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

8 – MPI-IO

8.8 – Nonblocking Data Access

Nonblocking Data Access

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

8 – MPI-IO

8.8 – Nonblocking Data Access

8.8.1 – Data Access with Explicit Offsets

```
1 program iread_at
2   use mpi
3   implicit none
4
5   integer, parameter :: nb_values=10
6   integer :: i,nb_iterations=0,rank,bytes_in_integer, &
7             fh,request,code
8   integer(kind=MPI_OFFSET_KIND) :: offset
9   integer, dimension(nb_values) :: values
10  integer, dimension(MPI_STATUS_SIZE) :: status
11  logical :: finish
12
13  call MPI_INIT(code)
14  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
```

```
1 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
2           fh,code)
3
4 call MPI_TYPE_SIZE(MPI_INTEGER,bytes_in_integer,code)
5
6 offset=rank*nb_values*bytes_in_integer
7 call MPI_FILE_IREAD_AT(fh,offset,values,nb_values, &
8           MPI_INTEGER,requests,code)
9
10 do while (nb_iterations < 5000)
11   nb_iterations=nb_iterations+1
12   ! Overlapping the I/O operation with computations
13   ...
14   call MPI_TEST(request,finish,status,code)
15   if (finish) exit
16 end do
17 print *, "After",nb_iterations,"iterations, read process",rank,":",values
18
19 call MPI_FILE_CLOSE(fh,code)
20 call MPI_FINALIZE(code)
21
22 end program iread_at
```

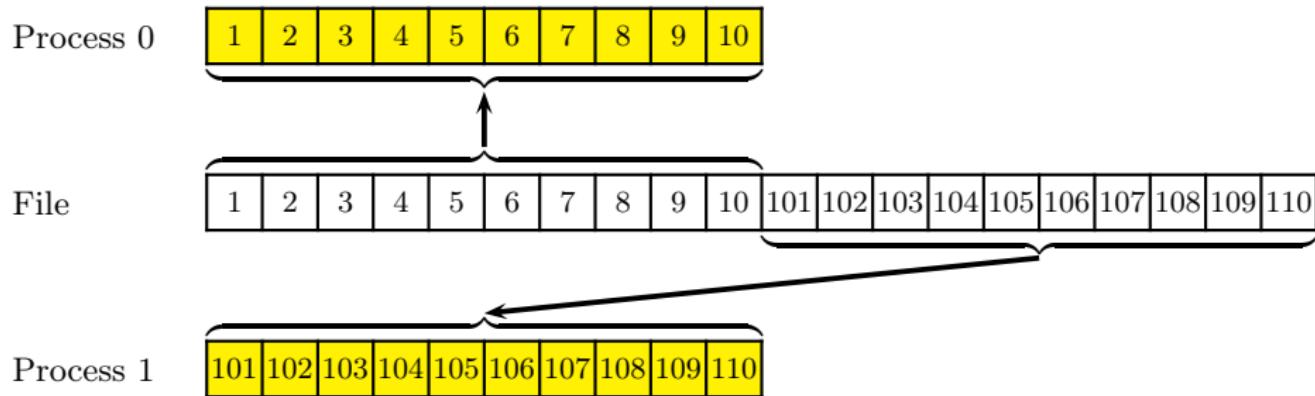


Figure 55: Example of MPI_FILE_IREAD_AT()

```
> 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
```

8 – MPI-IO

8.8 – Nonblocking Data Access

8.8.2 – Data access with individual file pointers

```
1 program iread
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_values=10
6   integer                      :: rank,fh,request,code
7   integer, dimension(nb_values) :: values
8   integer, dimension(MPI_STATUS_SIZE) :: status
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
12
13  call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14      fh,code)
15
16  call MPI_FILE_IREAD(fh,values,nb_values,MPI_INTEGER,request,code)
17 ! Overlapping the I/O operation with computations
18 ...
19  call MPI_WAIT(request,status,code)
20  print *, "Read process",rank,":",values(:)
21
22  call MPI_FILE_CLOSE(fh,code)
23  call MPI_FINALIZE(code)
24 end program iread
```

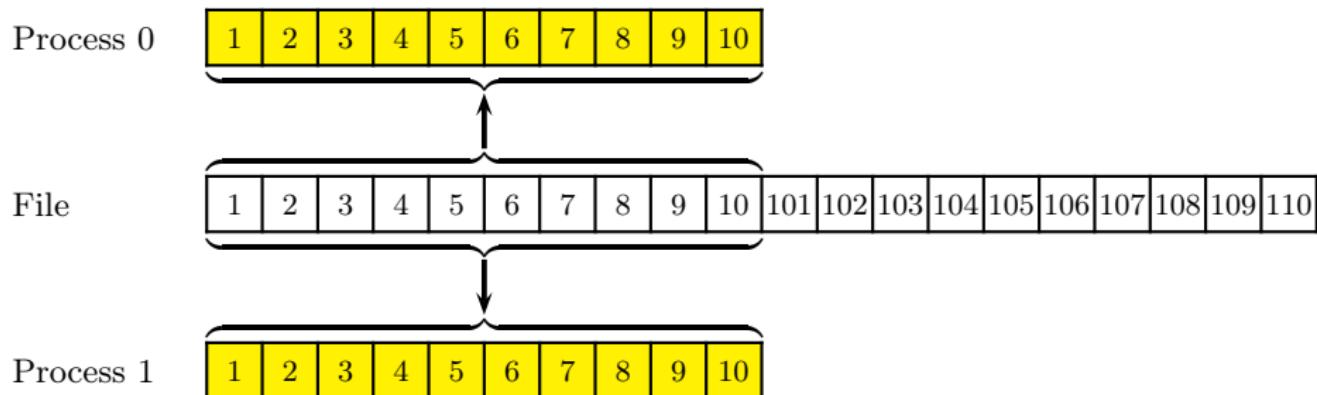


Figure 56: Example of MPI_FILE_IREAD()

```
> 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
```

Split collective data access routines

- The split collective routines support a restricted form of **nonblocking** operations for **collective** data access.
- A single collective operation is split into two parts: a begin routine and an end routine.
- On any MPI process, each file handle can only have one active split collective operation at any time.
- Collective I/O operations are **not** permitted concurrently with a split collective access on the same file handle (but non-collective I/O are allowed). The buffer passed to a begin routine must not be used while the routine is outstanding.

```
1 program read_ordered_begin_end
2
3 use mpi
4 implicit none
5
6 integer :: rank,fh,code
7 integer, parameter :: nb_values=10
8 integer, dimension(nb_values) :: values
9 integer, dimension(MPI_STATUS_SIZE) :: status
10
11 call MPI_INIT(code)
12 call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
13
14 call MPI_FILE_OPEN(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15 fh,code)
16
17 call MPI_FILE_READ_ORDERED_BEGIN(fh,values,4,MPI_INTEGER,code)
18 print *, "Process number      :",rank
19 call MPI_FILE_READ_ORDERED_END(fh,values,status,code)
20
21 print *, "Read process",rank,: ,values(1:4)
22
23 call MPI_FILE_CLOSE(fh,code)
24 call MPI_FINALIZE(code)
25
26 end program read_ordered_begin_end
```

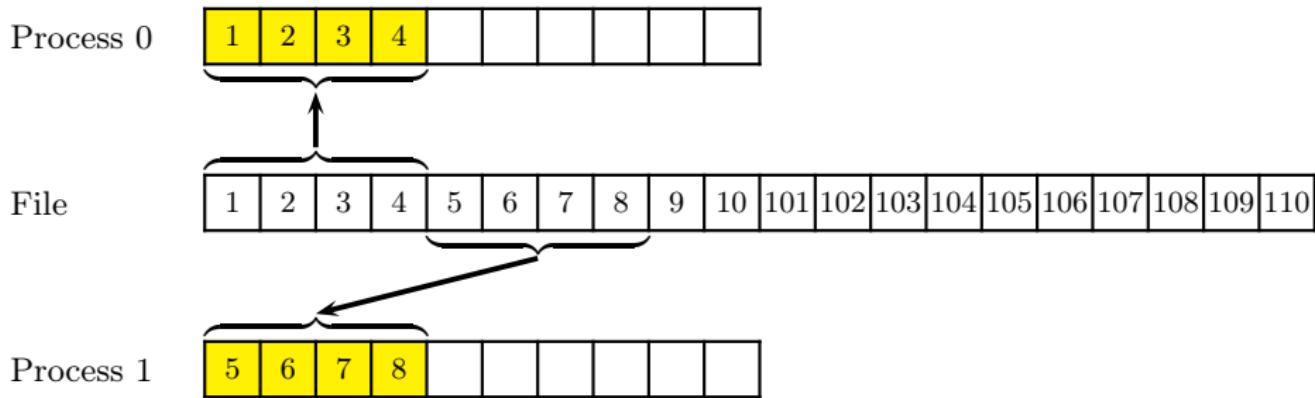


Figure 57: Example of MPI_FILE_READ_ORDERED_BEGIN()

```
> 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
```

8 – MPI-IO

8.9 – Advice

Conclusion

MPI-IO provides a high-level I/O interface and a rich set of functionalities. Complex operations can be performed easily using an MPI-like interface and MPI libraries provide suitable optimisations. MPI-IO also achieves portability.

Advice

- Avoid subroutines with explicit positioning and prefer the use of shared or individual **pointers** as they provide a higher-level interface.
- Take advantage of **collective I/O** operations as they are generally more efficient.
- Use **asynchronous I/O** only after getting correct behaviour on a blocking version.

8 – MPI-IO

8.10 – Definitions

Definitions (files)

file : An MPI file is an ordered collection of typed data items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.

file handle : A file handle is an opaque object created by `MPI_FILE_OPEN()` and freed by `MPI_FILE_CLOSE()`. All operations on an open file reference the file through the file handle.

file pointer : A file pointer is an implicit offset maintained by MPI.

offset : An offset is a position in the file relative to the current view, expressed as a count of etypes. Holes in the view's filetype are skipped when calculating this position.

Definitions (views)

- displacement** : A file displacement is an absolute byte position relative to the beginning of a file. The displacement defines the location where a view begins.
- etype** : An etype (elementary datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes.
- filetype** : A filetype is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent.
- view** : A view defines the current set of data visible and accessible from an open file as an ordered set of etypes.

- 1 Introduction
- 2 Environment
- 3 Point-to-point Communications
- 4 Collective communications
- 5 Communication Modes
- 6 Derived datatypes
- 7 Communicators
- 8 MPI-IO
- 9 Conclusion
- 10 Index

9 – Conclusion

Conclusion

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

1	Introduction	
2	Environment	
3	Point-to-point Communications	
4	Collective communications	
5	Communication Modes	
6	Derived datatypes	
7	Communicators	
8	MPI-IO	
9	Conclusion	
10	Index	
10.1	Constants MPI index	241
10.2	Subroutines MPI index	243

mpi	25
mpi.h	26
MPI_ADDRESS_KIND	111, 121, 130–132, 134, 135, 137, 222
MPI_ANY_SOURCE	39, 45
MPI_ANY_TAG	39, 45
MPI_BSEND_OVERHEAD	91
MPI_BSEND_OVERHEAD()	88
MPI_BYTE	184, 213
MPI_CHARACTER	132
MPI_COMM_NULL	146
MPI_COMM_WORLD	27–30, 36, 42, 43, 45, 50, 53, 56, 59, 62, 65, 66, 69, 75, 78, 86, 91, 113–118, 124, 128, 129, 132, 133, 137, 138, 143, 148, 152, 154, 164, 165, 170, 179, 186, 188, 191, 193, 196, 199, 201, 203, 205, 207, 210, 218–220, 222, 223, 227, 228, 230, 233
MPI_COMPLEX	107
MPI_IN_PLACE	80
MPI_INFO_NULL	179, 186, 188, 191, 193, 196, 199, 201, 203, 205, 207, 210, 217, 218, 220, 223, 228, 230, 233
MPI_INTEGER	36, 42, 43, 45, 53, 75, 78, 86, 91, 107, 111, 119, 129, 132, 136–138, 185, 186, 188, 191, 193, 196, 199, 201, 203, 205, 207, 210, 217, 218, 220, 223, 225, 228, 230, 233
MPI_LOGICAL	132
MPI_MODE_CREATE	179, 186
MPI_MODE_RDONLY	188, 191, 193, 196, 199, 201, 203, 205, 207, 210, 218, 220, 223, 228, 230, 233
MPI_MODE_RDWR	179

MPI_MODE_WRONLY	186
MPI_OFFSET_KIND	186, 188, 199, 210, 215, 218, 219, 222, 227
MPI_ORDER_C	126
MPI_ORDER_FORTRAN	126, 129, 217, 218, 220, 225
MPI_PROC_NULL	39, 161
MPI_PROD	78
MPI_REAL	56, 59, 62, 66, 69, 107, 109–111, 113–117, 119, 124, 132, 148, 170
MPI_SEEK_CUR	209, 210
MPI_SEEK_END	209
MPI_SEEK_SET	209, 210
MPI_SOURCE	45
MPI_STATUS_IGNORE	39, 42, 43, 86, 91
MPI_STATUS_SIZE	35, 36, 40, 44, 45, 101, 113, 115, 117, 124, 128, 132, 137, 186, 188, 191, 193, 196, 199, 201, 203, 205, 207, 210, 218, 219, 222, 227, 230, 233
MPI_SUM	75
MPI_TAG	45
MPI_UNDEFINED	146
mpif.h	25

MPI_ABORT	28, 28
MPI_ALLGATHER	49, <u>61</u> , 61, 62, 68, 80
MPI_ALLGATHERV	80
MPI_ALLREDUCE	49, 71, <u>77</u> , 77, 78
MPI_ALLTOALL	49, <u>68</u> , 68, 69, 80
MPI_ALLTOALLV	80
MPI_ALLTOALLW	80
MPI_BARRIER	49, <u>50</u> , 50
MPI_BCAST	49, <u>52</u> , 52, 53, 61, 71, 148
MPI_BSEND	82, 88–91
MPI_BUFFER_ATTACH	88, 91
MPI_BUFFER_DETACH	88, 91
MPI_CART_COOARDS	<u>159</u> , 159, 160, 165, 170
MPI_CART_CREATE	151, 152, 154, 165, 170
MPI_CART_RANK	<u>157</u> , 157, 158
MPI_CART_SHIFT	<u>161</u> , 161–163, 165
MPI_CART_SUB	168, 170
MPI_COMM_CREATE	144, 149
MPI_COMM_DUP	144
MPI_COMM_FREE	144, 148
MPI_COMM_GROUP	149
MPI_COMM_RANK	29, 30, 36, 42, 45, 53, 56, 59, 62, 65, 69, 75, 78, 86, 91, 113, 115, 117, 124, 128, 132, 137, 148, 165, 170, 186, 188, 191, 193, 196, 199, 201, 203, 205, 207, 210, 218, 219, 222, 227, 230, 233
MPI_COMM_SIZE	29, 30, 45, 56, 59, 62, 65, 69, 75, 78, 164
MPI_COMM_SPLIT	144, <u>146</u> , 146–148, 168

MPI_DIMS_CREATE	156, 156, 164
MPI_EXSCAN	80
MPI_FILE_CLOSE 179, 186, 188, 191, 193, 196, 199, 201, 203, 205, 207, 210, 218, 220, 223, 228, 230, 233, 236	
MPI_FILE_GET_INFO	178
MPI_FILE_GET_POSITION	209
MPI_FILE_GET_POSITION_SHARED	209
MPI_FILE_IREAD	183, 230
MPI_FILE_IREAD_AT	182, 228
MPI_FILE_IREAD_SHARED	183
MPI_FILE_IWRITE	183
MPI_FILE_IWRITE_AT	182
MPI_FILE_IWRITE_SHARED	183
MPI_FILE_OPEN . 179, 186, 188, 191, 193, 195, 196, 199, 201, 203, 205, 207, 210, 218, 220, 223, 228, 230, 233, 236	
MPI_FILE_READ	183, 191, 193, 210, 218, 220, 223
MPI_FILE_READ_ALL	183, 201, 203, 205
MPI_FILE_READ_ALL_BEGIN	183
MPI_FILE_READ_ALL_END	183
MPI_FILE_READ_AT	182, 188
MPI_FILE_READ_AT_ALL	182, 199
MPI_FILE_READ_AT_ALL_BEGIN	182
MPI_FILE_READ_AT_ALL_END	182
MPI_FILE_READ_ORDERED	183, 207
MPI_FILE_READ_ORDERED_BEGIN	183, 233

MPI_FILE_READ_ORDERED_END	183, 233
MPI_FILE_READ_SHARED	183, 196
MPI_FILE_SEEK	209, 210
MPI_FILE_SEEK_SHARED	209
MPI_FILE_SET_VIEW	<u>215</u> , 215, 217, 218, 220, 223
MPI_FILE_WRITE	183
MPI_FILE_WRITE_ALL	183
MPI_FILE_WRITE_ALL_BEGIN	183
MPI_FILE_WRITE_ALL_END	183
MPI_FILE_WRITE_AT	182, 186
MPI_FILE_WRITE_AT_ALL	182
MPI_FILE_WRITE_AT_ALL_BEGIN	182
MPI_FILE_WRITE_AT_ALL_END	182
MPI_FILE_WRITE_ORDERED	183
MPI_FILE_WRITE_ORDERED_BEGIN	183
MPI_FILE_WRITE_ORDERED_END	183
MPI_FILE_WRITE_SHARED	183
MPI_FINALIZE ..	25, 30, 36, 42, 45, 53, 56, 59, 62, 66, 69, 75, 78, 86, 91, 114, 116, 118, 124,
	129, 133, 138, 143, 148, 165, 170, 179, 186, 188, 191, 193, 196, 199, 201, 203, 205,
	207, 210, 218, 220, 223, 228, 230, 233
MPI_GATHER	49, <u>58</u> , 58, 59, 61, 64, 80
MPI_GATHERV	<u>64</u> , 64, 66, 80
MPI_GET_ADDRESS	131, 133
MPI_GROUP_FREE	149
MPI_GROUP_INCL	149

MPI_IBSEND	82, 90, <u>100</u> , 100
MPI_INIT 25, 26, 30, 36, 42, 45, 53, 56, 59, 62, 65, 69, 75, 78, 86, 91, 113, 115, 117, 124, 128, 132, 137, 143, 148, 164, 170, 179, 186, 188, 191, 193, 196, 199, 201, 203, 205, 207, 210, 218, 219, 222, 227, 230, 233	
MPI_IRecv	82, 96, 99, <u>100</u> , 100, 103
MPI_ISEND	82
MPI_ISEND	82, 96, 99, <u>100</u> , 100, 103
MPI_ISSEND	82, <u>100</u> , 100
MPI_OP_CREATE	71
MPI_OP_FREE	71
MPI_RECV	<u>35</u> , 35, 36, 43, 45, 82, 86, 91, 114, 116, 118, 133, 138
MPI_REDUCE	49, 71, <u>74</u> , 74, 75, 80
MPI_RSEND	82, 95
MPI_SCAN	71, 80
MPI_SCATTER	49, <u>55</u> , 55, 56, 80, 170
MPI_SCATTERV	80
MPI_SEND	<u>34</u> , 34–36, 43, 45, 82, 92, 114, 116, 118, 133, 138
MPI_SENDRECV	39, <u>40</u> , 40, <u>41</u> , 42, 43, 139
MPI_SENDRECV_REPLACE	39, <u>44</u> , 44, 122, 124, 129
MPI_SSEND	82, 85, 86
MPI_TEST	101, 226, 228
MPI_TESTALL	<u>101</u> , 101
MPI_TYPE_COMMIT . <u>112</u> , 112, 113, 115, 117, 124, 129, 133, 138, 215, 217, 218, 220, 223, 225	
MPI_TYPE_CONTIGUOUS	107, <u>109</u> , 109, 113

- MPI_TYPE_CREATE_HINDEXED 119, 121, 121
MPI_TYPE_CREATE_HVECTOR 111, 111, 119
MPI_TYPE_CREATE_RESIZED 135, 135, 137, 223
MPI_TYPE_CREATE_STRUCT 107, 130, 131, 133, 225
MPI_TYPE_CREATE_STRUCT constructor 130
MPI_TYPE_CREATE_SUBARRAY 125, 126, 129, 217, 218, 220, 225
MPI_TYPE_FREE 112, 114, 116, 118, 124, 129, 133
MPI_TYPE_GET_EXTENT 119, 134, 137, 138, 223
MPI_TYPE_INDEXED 107, 119, 120, 120, 124, 130, 136, 223
MPI_TYPE_SIZE 91, 119, 134, 134, 137, 186, 188, 199, 210, 220, 223, 225, 228
MPI_TYPE_VECTOR 107, 110, 110, 115, 117, 136, 137
MPI_WAIT 96, 98, 99, 101, 101, 226, 230
MPI_WAITALL 101, 101, 103