MPI

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

CNRS — IDRIS

v5.5.1 February 6th 2026

Plan I

Plan II

Plan III

Plan IV

Introduction

Introduction

Availability and updating

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

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

Introduction

Parallelism

The goal of parallel programming is to :

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

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

Introduction

Sequential progamming model

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

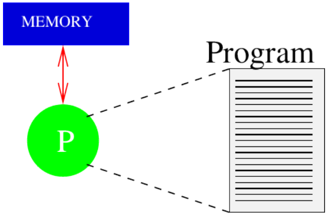


Figure 1 – Sequential programming model

Introduction

Message passing programming model

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

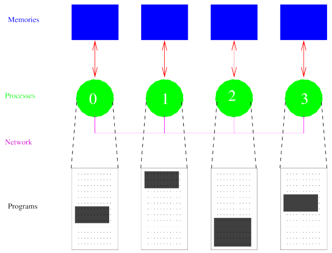


Figure 2 – Message Passing Programming Model

Introduction

Message Passing concepts

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

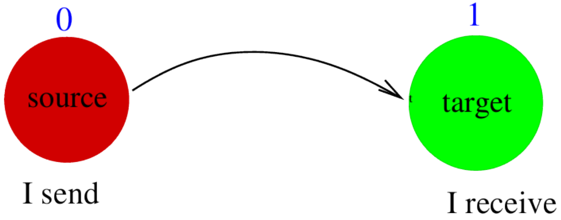


Figure 3 – Message Passing

Introduction

Message content

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

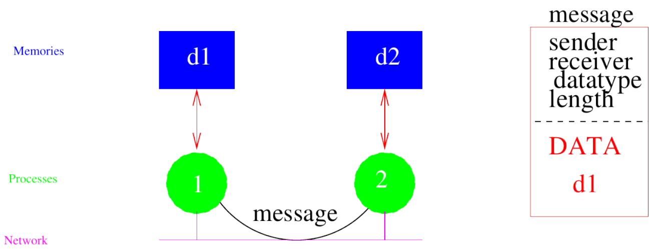


Figure 4 – Message Construction

Introduction

Environment

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

Introduction

Supercomputer architecture

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

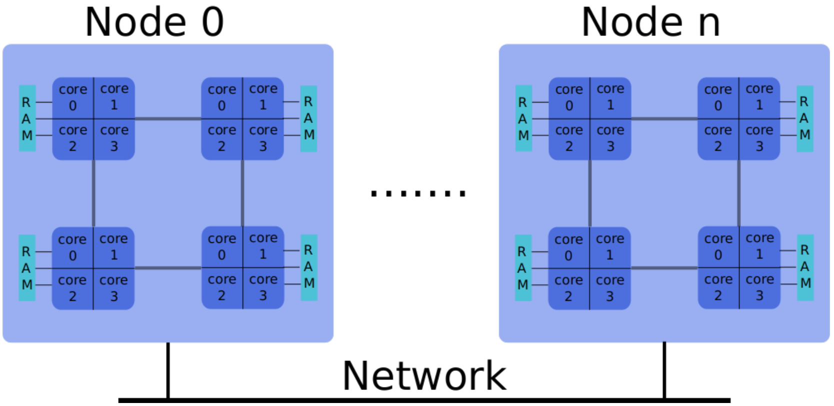


Figure 5 – Supercomputer architecture

Introduction

Jean Zay

720 nodes

- 2 Intel Cascade Lake processor (20 cores) at 2.5 Ghz by node
- 192 GB by node

396 nodes

- 2 Intel Cascade Lake processor (20 cores) at 2.5 Ghz by node
- 192 GB by node
- 126 nodes with 4 GPU Nvidia Tesla V100 SXM2 16 GB
- 270 nodes with 4 GPU Nvidia Tesla V100 SXM2 32 GB

31 nodes

- 2 Intel Cascade Lake 6226 processor (12 cores at 2.7 GHz), 24 cores by node
- 20 nodes with 384 GB
- 11 nodes with 768 GB
- 8 GPU Nvidia Tesla V100 SXM2 32GB

Jean Zay

52 nodes

- 2 AMD Milan EPYC 7543 processor (32 cores at 2.80 GHz), 64 cores by node
- 512 GB of memory by node
- 8 GPU Nvidia A100 SXM4 80 GB

364 nodes

- 2 Intel Xeon Platinum 8468 processor (48 cores at 2.10 GHz), 96 cores by node
- 512 GB of memory by node
- 4 GPU Nvidia H100 SXM5 80 GB

Introduction

MPI vs OpenMP

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

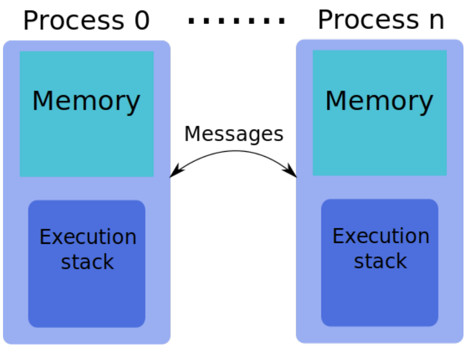


Figure 6 – MPI scheme

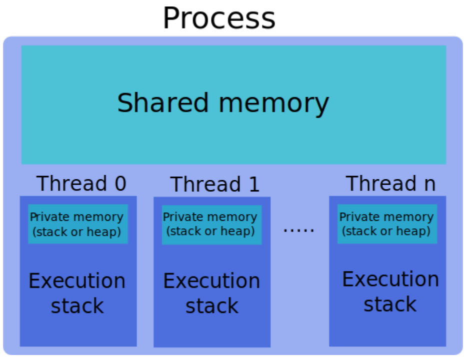


Figure 7 – OpenMP scheme

Introduction

Domain decomposition

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

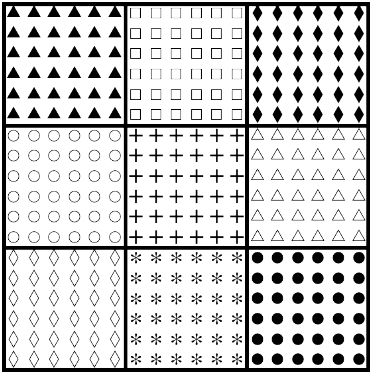


Figure 8 – Decomposition in subdomains

Introduction

History

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

Introduction

MPI 4.0

Version 4.0 : June 2021

- Large count
- Partitioned communication
- MPI Session

Version 4.1 : November 2023

MPI 5.0

Version 5.0 : June 2025
Definition of an ABI (Application Binary Interface)

Introduction

Library

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

Introduction

Open source MPI implementations

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

- MPICH : `http://www.mpich.org`
- Open MPI : `http://www.open-mpi.org`

Introduction

Tools

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

Introduction

Open source parallel scientific libraries

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

Environment

Environment

Description

- Any program unit calling MPI functions must include the `mpi4py` module.
- The `MPI_Init()` subroutine initializes the MPI environment :

```
mpi4py.MPI.Init()
```

  By default, `mpi4py` automatically initializes the MPI environment when you import the module.
- The `MPI_Finalize()` subroutine disables this environment :

```
mpi4py.MPI.Finalize()
```

  By default, `mpi4py` deactivates the MPI environment at the end of the program's execution.

Environment

Communicators

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

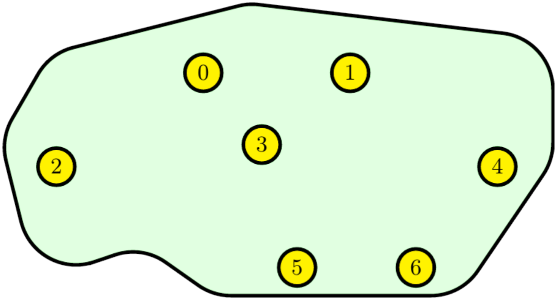


Figure 9 – MPI_COMM_WORLD Communicator

Environment

Termination of a program

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

```
mpi4py.MPI.Comm.Abort(errorcode=0)
```

- `comm` (instance de la classe `mpi4py.MPI.Comm`) : the communicator of which all the processes will be stopped ; it is advised to use MPI.`COMM_WORLD` ;
- `errorcode` : the error number returned to the UNIX environment.

Errors management in `mpi4py` is done via exceptions. Errors returned from MPI calls within Python code will raise an instance of the exception class MPI.Exception, which is a subclass of the standard Python exception `RuntimeError`. If the exception is not handled, only the process is stop unless the option `-m mpi4py` is used with the Python interpreter.

Environment

Rank and size

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

```
# Return the number of process
mpi4py.MPI.Comm.Get_size()
```

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

```
# Return the rank
mpi4py.MPI.Comm.Get_rank()
```

Environment

Example

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_procs = comm.Get_size()

print(f"I am process {rank} of {nb_procs}")
```

```
> mpiexec -n 7 python -m mpi4py who_am_I.py

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
```

Environnement

Compilation and execution of an MPI code

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

```
> mpiexec -n <number of processes> python -m mpi4py my_script_file.py
```

MPI Hands-On – Exercise 1 : MPI Environment

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

```
> mpiexec -n 4 python -m mpi4py ./even_odd
I am process 0, my rank is even
I am process 2, my rank is even
I am process 3, my rank is odd
I am process 1, my rank is odd
```

- To test whether the rank is odd or even, the operator *modulo* is `%` : `a%b`
- To execute your program, use the command `make exe`
- For the program to be recognized by the Makefile, it must be named `even_odd.py`

Point-to-point Communications

Point-to-point Communications

General Concepts

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

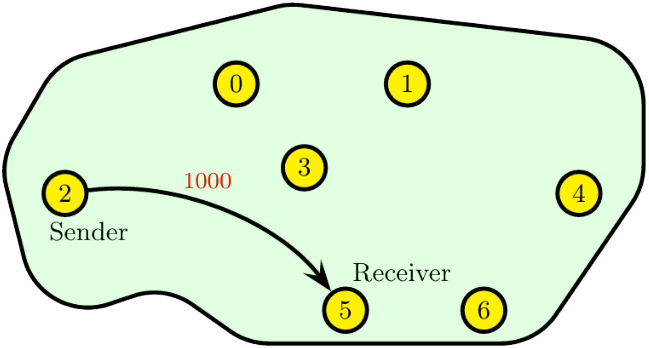


Figure 10 – Point-to-point communication

Point-to-point Communications

General Concepts

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

Point-to-point Communications

MPI4PY Two kind of message

- There are two types of messages with mpi4py ;
- One type is for Python objects, with a communication function name in lowercase ;
- This type of message uses serialization and is less efficient
- Only one Python object can be communicated with this type of message ;
- The received object is the return value of the function receiving the message.
- The other type is for contiguous arrays, such as with NumPy, with a communication function name with the first letter in uppercase ;
- For this type of message, a triple (buffer, length, type) must be provided for communication ;
- Length and type are optional ;
- The received object is in the function arguments ;
- This type of message is more efficient.

Point-to-point Communications

Blocking Send `MPI_Send`

```
mpi4py.MPI.Comm.Send([buf, count, datatype], dest, tag=0)
mpi4py.MPI.Comm.send(obj, dest, tag=0)
```

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

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

Point-to-point Communications

Blocking Receive MPI_Recv

```
mpi4py.MPI.Comm.Recv([buff, count, datatype],
                     source=ANY_SOURCE, tag=ANY_TAG, status=None)
# Return the object received
mpi4py.MPI.Comm.recv(source=ANY_SOURCE, tag=ANY_TAG, status=None)
```

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

Remarks :

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

Point-to-point Communications

Example (see Fig. 10)

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
tag = 100

if rank == 2:
    value = 1000
    comm.send(valeur, dest=5, tag=tag)
elif rank == 5:
    value = comm.recv(source=2, tag=tag)
    print(f"I, process 5, I received {value} from the process 2.")
```

```
> mpiexec -n 7 python -m mpi4py point_to_point.py

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

Exemple (voir Fig. 10)

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
tag = 100

value = np.zeros(1, dtype=np.int32)

if rank == 2:
    value[0] = 1000
    comm.Send(value, dest=5, tag=tag)
elif rank == 5:
    comm.Recv(value, source=2, tag=tag)
    print(f"I, process 5, I received {value[0]} from the process 2.")
```

Point-to-point Communications

MPI4PY base datatypes

MPI4PY Type	Numpy Type
`mpi4py.MPI.INT`	`numpy.int32`
`mpi4py.MPI.LONG`	`numpy.int64`
`mpi4py.MPI.FLOAT`	`numpy.float32`
`mpi4py.MPI.DOUBLE`	`numpy.float64`
`mpi4py.MPI.BYTE`	`numpy.byte`

There are function to convert between the numpy datatypes and the mpi4py datatypes

```
# Return a mpi4py type
mpi4py.util.dtlib.from_numpy_dtype(dtype)
# Return a dtype
mpi4py.util.dtlib.to_numpy_dtype(datatype)
```

Point-to-point Communications

Other possibilities

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

Point-to-point Communications

Simultaneous send and receive MPI_Sendrecv

```
mpi4py.MPI.Comm.Sendrecv([sendbuf, sendcount, sendtype],
                         dest, sendtag=0,
                         recvbuf=[recvbuf, recvcount, recvtype],
                         source=ANY_SOURCE, recvtag=ANY_TAG, status=None)
# Return the object received
mpi4py.MPI.Comm.sendrecv(sendobj, dest, sendtag=0,
                         recvbuf=None, source=ANY_SOURCE, recvtag=ANY_TAG,
                         status=None)
```

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

Remark :

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

Point-to-point Communications

Simultaneous send and receive `MPI_Sendrecv`

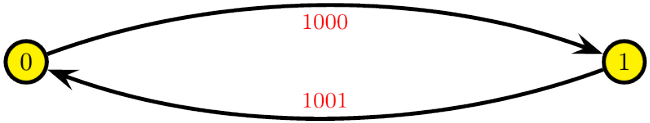


Figure 11 – `sendrecv` Communication between the Processes 0 and 1

Point-to-point Communications

Example (see Fig. 11)

```python
from mpi4py import MPI

tag = 110

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

num_proc = (rank + 1) % 2
message = np.zeros(1, dtype=np.int32)
value = np.zeros(1, dtype=np.int32)
message[0] = rank + 1000

comm.Sendrecv(message, dest=num_proc, sendtag=tag,
              recvbuf=value, source=num_proc, recvtag=tag)

print(f"I, process {rank}, I received {value[0]} from process {num_proc}.")
```

```
> mpiexec -n 2 python -m mpi4py sendrecv.py

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

Point-to-point Communications

Be careful !

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

```
val[0] = rank + 1000
comm.Send(val, dest=num_proc, tag=tag)
comm.Recv(val, source=num_proc, tag=tag)
```

Point-to-point Communications

Simultaneous send and receive MPI_Sendrecv_replace

```
mpi4py.MPI.Comm.Sendrecv_replace([buf,count,datatype],
                                 dest, sendtag=0,
                                 source=ANY_SOURCE, recvtag=ANY_TAG,
                                 status=None)
```

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

Remark :

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

Point-to-point Communications

Example

```python
from mpi4py import MPI
import numpy as np

m = 4
tag = 11

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
intnp = np.int32
A = np.zeros((m, m), dtype=intnp)

if rank == 0:
    A = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12],
                  [13, 14, 15, 16]], dtype=intnp)
    comm.Send([A, 3], dest=1, tag=tag)
else:
    statut = MPI.Status()
    comm.Recv([A[0, 1:], 3], status=statut)
    print(f"I process {rank}, I received 3 elements from the process "
          f"{statut.source} with tag "
          f"{statut.tag} the elements are {A[0, 1]} {A[0, 2]} {A[0, 3]}.")
```

Point-to-point Communications

```
> mpiexec -n 2 python -m mpi4py wildcard.py
I, process 1, I received 3 elements from the process 0
 with tag 11 the elements are 1 2 3.
```

MPI Hands-On – Exercise 2 : Ping-pong

- Point to point communications : *Ping-Pong* between two processes

- This exercice is composed of 3 steps :

  1. *Ping* : complete the script `ping_pong_1.py` in such a way that the process 0 sends a message containing 1000 random reals to process 1.

  2. *Ping-Pong* : complete the script `ping_pong_2.py` in such a way that the process 1 sends back the message to the process 0, and measure the communication duration with the `MPI_Wtime()` function.

  3. *Ping-Pong match* : complete the script `ping_pong_3.py` in such a way that processes 0 and 1 perform 9 *Ping-Pong*, while varying the message size, and measure the communication duration each time. The corresponding bandwidths will be printed.

MPI Hands-On – Exercise 2 : Ping-pong

Remarks :

- To execute the first step : `make exe1`
- To execute the second step : `make exe2`
- To execute the last step : `make exe3`

- The generation of random numbers uniformly distributed in the range [0,1[is made in Python by calling the `random.rand` function of numpy.

- The time duration measurements can be done like this :

```
time_begin=MPI.Wtime();
..................................................
time_end=MPI.Wtime();
print(f"... in {time_end-time_begin} seconds.\n")
```

Collective communications

Collective communications

General concepts

- Collective communications allow making a series of point-to-point communications in one single call.
- A collective communication always concerns all the processes of the indicated communicator.
- For each process, the call ends when its participation in the collective call is completed, in the sense of point-to-point communications (therefore, when the concerned memory area can be changed).
- The management of tags in these communications is transparent and system-dependent. Therefore, they are never explicitly defined during calls to subroutines. An advantage of this is that collective communications never interfere with point-to-point communications.

Collective communications

Types of collective communications

There are three types of subroutines :

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

Collective communications

Global synchronization : `MPI_Barrier()`

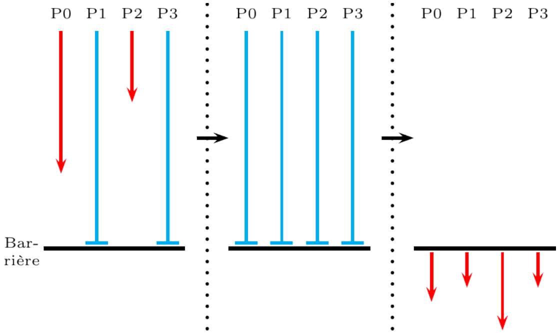


Figure 12 – Global Synchronization : `MPI_Barrier()`

```
mpi4py.MPI.Comm.Barrier()
mpi4py.MPI.Comm.barrier()
```

Collective communications

Global distribution : MPI_Bcast()

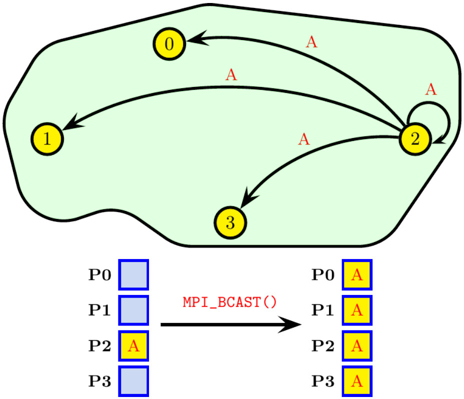


Figure 13 – Global distribution : `MPI_Bcast()`

Collective communications

Global distribution : MPI_Bcast()

```
mpi4py.MPI.Comm.Bcast([buffer, count, datatype], root=0)
# Return the distributed object
mpi4py.MPI.Comm.bcast(obj, root=0)
```

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

Collective communications

Example of MPI_Bcast()

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

value = np.zeros(1, dtype=np.int32)

if rank == 2:
    value[0] = 1002

comm.Bcast(value, root=2)

print(f"I, process {rank}, received {value[0]} of process 2")
```

```
> mpiexec -n 4 python -m mpi4py bcast.py

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

Collective communications

Selective distribution : `MPI_Scatter()`

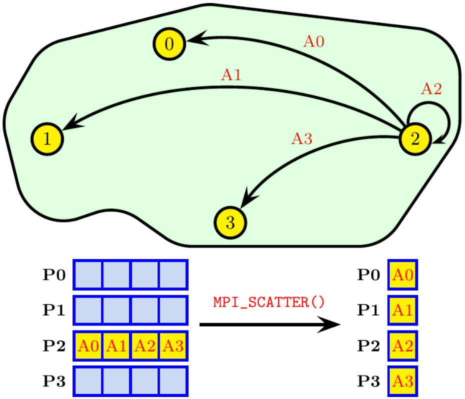


Figure 14 – Selective distribution : `MPI_Scatter()`

Collective communications

Selective distribution : MPI_Scatter()

```
mpi4py.MPI.Comm.Scatter([sendbuf, sendcount, sendtype],
                        [recvbuf, recvcount, recvtype], root=0)
# Return the object
mpi4py.MPI.Comm.scatter(sendobj, root=0)
```

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

Remarks :

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

Collective communications

Example of MPI_Scatter()

```python
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

nb_values = 8
block_length = nb_values // size

if rank == 2:
    values = np.arange(1001, 1001 + nb_values, dtype=np.float32)
    print(f"I, process {rank}, send my values array :{values}")
else:
    values = None

recvdata = np.empty(block_length, dtype=np.float32)
comm.Scatter(values, recvdata, root=2)
print(f"I, process {rank}, received {recvdata} from process 2")
```

```
> mpiexec -n 4 python -m mpi4py scatter.py
I, process 2 send my values array :
[1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.]

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

Collective communications

Collection : MPI_Gather()

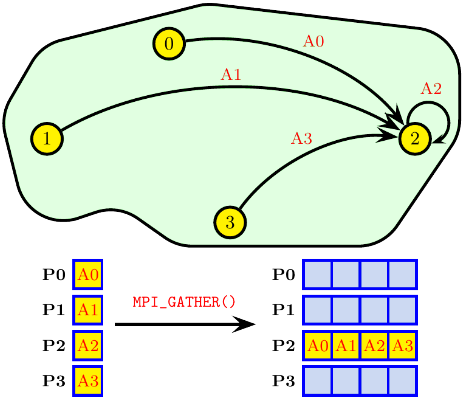


Figure 15 – Collection : `MPI_Gather()`

Collective communications

Collection : MPI_Gather()

```
mpi4py.MPI.Comm.Gather([sendbuf, sendcount, sendtype],
                       [recvbuf, recvcount, recvtype],root=0)
# Return for root an object list
mpi4py.MPI.Comm.gather(sendobj, root=0)
```

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

Remarks :

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

Collective communications

Collection : MPI_Gather()

```
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_procs = comm.Get_size()

nb_values = 8
block_length = nb_values // nb_procs

values = np.arange(1001+rank*block_length,
                   1001+(rang+1)*block_length,
                   dtype=np.float32)
print(f"I, process {rank}, sent my values array : {values}")

if rank == 2:
    recvdata = np.empty(nb_values, dtype=np.float32)
else:
    recvdata = None
comm.Gather(values, recvdata, root=2)
if rank == 2:
    print(f"I, process {rank}, received {recvdata}")
```

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

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

Collective communications

Gather-to-all : `MPI_Allgather()`

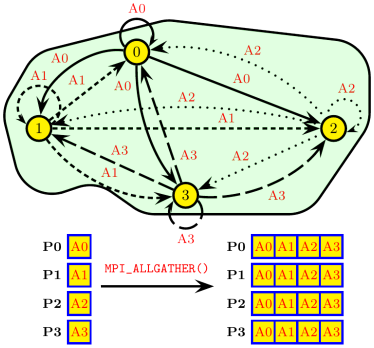


Figure 16 – Gather-to-all : `MPI_Allgather()`

Collective communications

Gather-to-all : `MPI_Allgather()`

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

```
mpi4py.MPI.Comm.Allgather([sendbuf, sendcount, sendtype],
                          [recvbuf, recvcount, recvtype])
# Return an object list
mpi4py.MPI.Comm.allgather(sendobj)
```

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

Remarks :

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

Collective communications

Example of MPI_Allgather()

```python
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_procs = comm.Get_size()

nb_values = 8
block_length = nb_values // nb_procs

values = np.arange(1001+rank*block_length,
                   1001+(rank+1)*block_length,
                   dtype=np.float32)
recvdata = np.empty(nb_values, dtype=np.float32)

comm.Allgather(values, recvdata)

print(f"I, process {rank}, received {recvdata}")
```

```
> mpiexec -n 4 python -m mpi4py allgather.py

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

Collective communications

Extended gather : `MPI_Gatherv()`

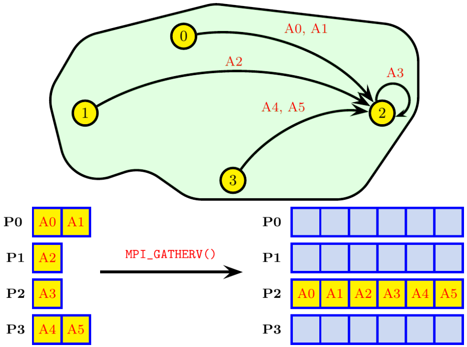


Figure 17 – Extended gather : `MPI_Gatherv()`

Collective communications

Extended Gather : MPI_Gatherv()

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

```
mpi4py.MPI.Comm.Gatherv([sendbuf, sendcount, sendtype],
                        [recvbuf, recvcount, displs, recvtype], root=0)
```

The i-th process of the communicator comm sends to process root, a message starting at position sendbuf, of sendcount element of type sendtype, and root receives at position recvbuf, of 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 process root must be such that the data size sent and received is the same.

Collective communications

Example of `MPI_Gatherv()`

```python
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_procs = comm.Get_size()

nb_values = 10
block_length = nb_values // nb_procs
remainder = nb_values % nb_procs
if rank < remainder:
    block_length += 1

values = np.empty(block_length, dtype=np.float32)
begin = 1001+rank*(nb_values // nb_procs)+(rank if rank < remainder else remainder)
values = np.arange(begin, begin+block_length, dtype=np.float32)
print(f"I, process {rank} send my values array : {values}")

if rank == 2:
    nb_elements_received = np.empty(nb_procs, dtype=np.int32)
    displacement = np.empty(nb_procs, dtype=np.int32)
    nb_elements_received[0] = nb_values // nb_procs
    if reste > 0:
        nb_elements_received[0] += 1
    displacement[0] = 0
    for i in range(1, nb_procs):
      displacement[i] = displacement[i-1] + nb_elements_received[i-1]
      nb_elements_received[i] = nb_values // nb_procs
      if i < remainder:
          nb_elements_received[i] += 1
    recvdata = np.empty(nb_values, dtype=np.float32)
else :
    nb_elements_received = None
    displacement = None
    recvdata = None
```

Collective communications

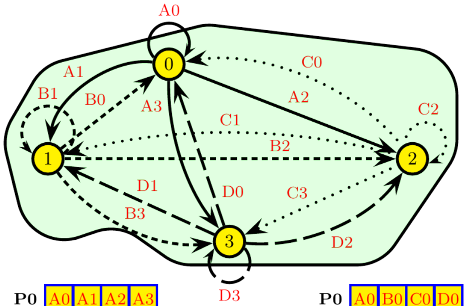
Example of MPI_Gatherv()

```
comm.Gatherv(values, [recvdata, nb_elements_received, displacement,MPI.FLOAT],
             root=2)
if rank == 2:
    print(f"I, process {rank}, received {recvdata}")
```

```
> mpiexec -n 4 python -m mpi4py gatherv.py

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

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

Collective communications

Collection and distribution : `MPI_Alltoall()`



P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

$\xrightarrow{\texttt{MPI\_ALLTOALL()}}$

P0	A0	B0	C0	D0
P1	A1	B1	C1	D1
P2	A2	B2	C2	D2
P3	A3	B3	C3	D3

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

Collective communications

Collection and distribution : `MPI_Alltoall()`

```
mpi4py.MPI.Comm.Alltoall([sendbuf, sendcount, sendtype],
                         [recvbuf, recvcount, recvtype])
# Return a list of objects
mpi4py.MPI.Comm.alltoall(sendobj)
```

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

Remark :

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

Collective communications

Example of MPI_Alltoall()

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_procs = comm.Get_size()

nb_values = 8
begin = 1001+rank*nb_values
values = np.arange(begin, begin+nb_values, dtype=np.float32)
print(f"I, process {rank} sent my values array : {values}")

block_length = nb_values // nb_procs
recvdata = np.empty(nb_values, dtype=np.float32)
comm.Alltoall(values, recvdata)

print(f"I, process {rank} received : {recvdata}")
```

Collective communications

Example of MPI_Alltoall()

```
> mpiexec -n 4 python -m mpi4py alltoall.py
I, process 1 sent my values array :
[1009. 1010. 1011. 1012. 1013. 1014. 1015. 1016.]
I, processus 0 sent my values array :
[1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.]
I, processus 2 sent my values array :
[1017. 1018. 1019. 1020. 1021. 1022. 1023. 1024.]
I, processus 3 sent my values array :
[1025. 1026. 1027. 1028. 1029. 1030. 1031. 1032.]

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

Collective communications

Global reduction

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

Collective communications

Distributed reduction : MPI_Reduce

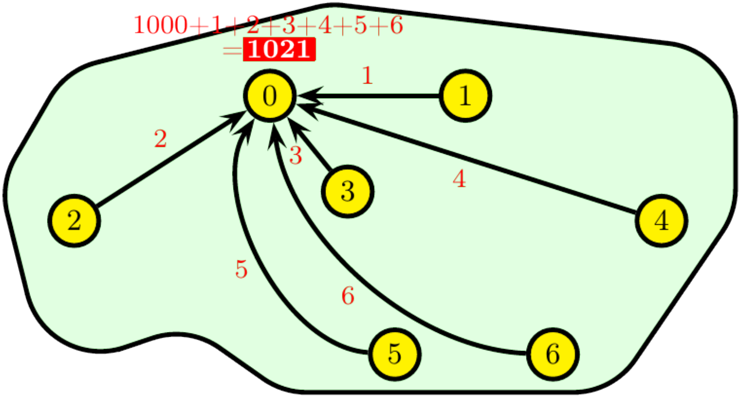


Figure 19 – Distributed reduction (sum)

Collective communications

Operations

Name	Operation
`mpi4py.MPI.SUM`	Sum of elements
`mpi4py.MPI.PROD`	Product of elements
`mpi4py.MPI.MAX`	Maximum of elements
`mpi4py.MPI.MIN`	Minimum of elements
`mpi4py.MPI.MAXLOC`	Maximum of elements and location
`mpi4py.MPI.MINLOC`	Minimum of elements and location
`mpi4py.MPI.LAND`	Logical AND
`mpi4py.MPI.LOR`	Logical OR
`mpi4py.MPI.LXOR`	Logical exclusive OR

Collective communications

Global reduction : `MPI_Reduce()`

```
mpi4py.MPI.Reduce([sendbuf,count,datatype], recvbuf, op=SUM, root=0)
# Return the result of reduction only for root
mpi4py.MPI.reduce(sendobj, op=SUM, root=0)
```

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

Collective communications

Example of MPI_Reduce()

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_procs = comm.Get_size()

if rank == 0:
    value = np.array([1000], np.int32)
else:
    value = np.array([rank], np.int32)
total_sum = np.zeros(1, np.int32)

comm.Reduce(value, total_sum, op=MPI.SUM, root=0)

if rank == 0:
    print(f"I, process 0, have the global total_sum value {total_sum[0]}")
```

```
> mpiexec -n 7 python -m mpi4py reduce.py

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

Collective communications

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

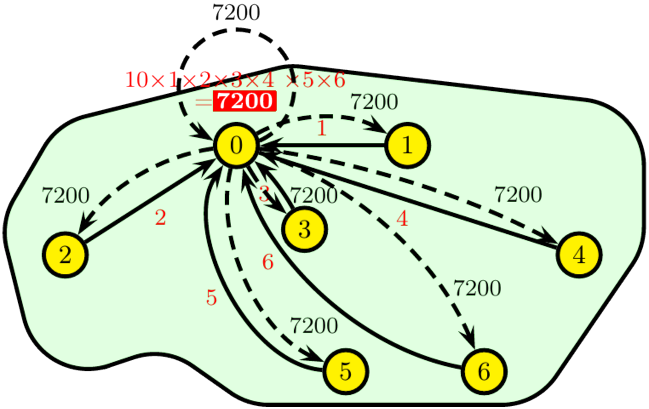


Figure 20 – Distributed reduction (product) with distribution of the result

Collective communications

Global all-reduction : MPI_Allreduce()

```
mpi4py.MPI.Comm.Allreduce([sendbuf, count, datatype], recvbuf, op=SUM)
# Return the result of reduction
mpi4py.MPI.Comm.allreduce(sendobj, op=SUM)
```

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

Collective communications

Example of MPI_Allreduce()

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_procs = comm.Get_size()

if rank == 0:
    value = np.array([10], np.int32)
else:
    value = np.array([rank], np.int32)
product = np.zeros(1, np.int32)

comm.Allreduce(value, product, op=MPI.PROD)

print(f"I, process {rank}, received the value of the global product"
      f" {produit[0]}")
```

Collective communications

Example of MPI_Allreduce()

```
> mpiexec -n 7 python -m mpi4py allreduce.py

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

Collective communications

Additions

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

  ```
  Allreduce(MPI.IN_PLACE,[sendrecvbuf,...);
  ```

Collective communications

Additions

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

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

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

Communication Modes

Communication Modes

Point-to-Point Send Modes

Mode	Blocking	Non-blocking
Standard send	`MPI_Send()`	`MPI_Isend()`
Synchronous send	`MPI_Ssend()`	`MPI_Issend()`
Buffered send	`MPI_Bsend()`	`MPI_Ibsend()`
Receive	`MPI_Recv()`	`MPI_Irecv()`

Communication Modes

Blocking call

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

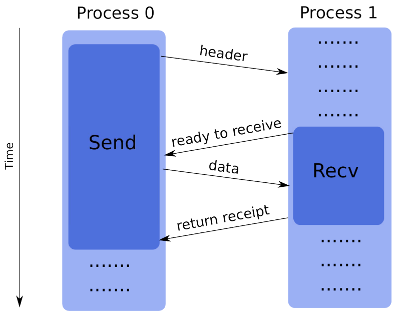
Communication Modes

Synchronous sends

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

Rendezvous Protocol

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



Communication Modes

Interface of `MPI_Ssend()`

```
mpi4py.MPI.Comm.Ssend([values, count, msgtype], dest, tag=0)
mpi4py.MPI.Comm.ssend(obj, dest, tag=0)
```

Advantages of synchronous mode

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

Disadvantages of synchronous mode

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

Communication Modes

Deadlock example

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

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

tag = 110

num_proc = (rank + 1) % 2

tmp = np.array([rank + 1000], dtype=np.int32)
comm.Ssend(tmp, dest=num_proc, tag=tag)
value = np.zeros(1, dtype=np.int32)
comm.Recv(value, source=num_proc, tag=tag)

print(f"I, process {rank} received {value} from process {num_proc}")
```

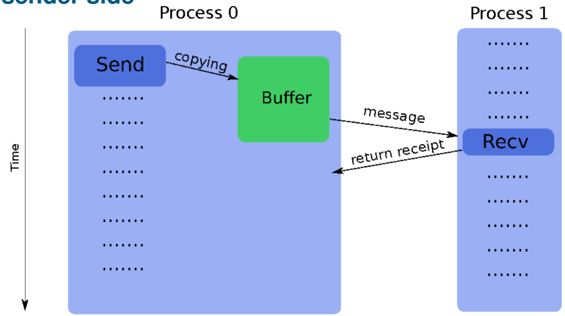
Communication Modes

Buffered sends

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

Protocol with user buffer on the sender side

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



Communication Modes

Buffered sends

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

Interfaces

```
mpi4py.MPI.Attach_buffer(buf)
mpi4py.MPI.Detach_buffer()
mpi4py.MPI.Comm.Bsend([values, count, msgtype], dest, tag=0)
mpi4py.MPI.Comm.bsend(obj, dest, tag=0)
```

Communication Modes

Advantages of buffered mode

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

Disadvantages of buffered mode

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

Communication Modes

No deadlocks

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

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

tag = 110
nb_elt = 1
nb_msg = 1

typesize = MPI.INT.Get_size()
# Convert BSEND_OVERHEAD to integer number
overhead = int(1 + (MPI.BSEND_OVERHEAD / typesize))
bufsize = nb_msg * (nb_elt + overhead)
buffer = np.empty(bufsize, dtype=np.int32)
MPI.Attach_buffer(buffer)
# We assume to have exactly 2 processes
num_proc = (rank + 1) % 2
tmp = np.array([rank + 1000], dtype=np.int32)
comm.Bsend(tmp, dest=num_proc, tag=tag)
value = np.zeros(1, dtype=np.int32)
comm.Recv(value, source=num_proc, tag=tag)

print(f"I, process {rank} received {value} from process {num_proc}")
MPI.Detach_buffer()
```

Communication Modes

Standard sends

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

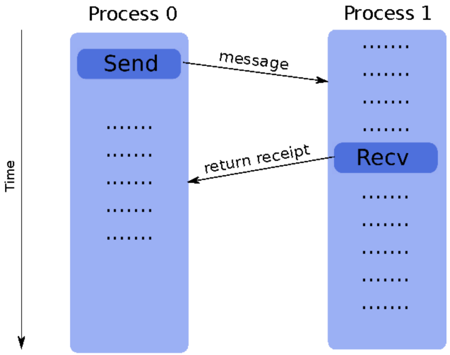
Interfaces

```
mpi4py.MPI.Comm.Send([values, count, msgtype], dest, tag=0)
mpi4py.MPI.Comm.send(obj, dest, tag=0)
```

Communication Modes

The eager protocol

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



Communication Modes

Advantages of standard mode

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

Disadvantages of standard mode

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

Communication Modes

Number of received elements

```
mpi4py.MPI.Comm.Recv([buff, count, datatype],
                     source=ANY_SOURCE, tag=ANY_TAG, status=None)
# Return the object received
mpi4py.MPI.Comm.recv(source=ANY_SOURCE, tag=ANY_TAG, status=None)
```

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

```
# Return the number of element
mpi4py.MPI.Status.Get_count(datatype=BYTE)
```

Communication Modes

Number of received elements

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

```
mpi4py.MPI.Comm.Probe(source=ANY_SOURCE, tag=ANY_TAG, status=None)
```

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

```
comm.Probe(status=status)
msgsize = status.Get_count(MPI_INT)
buf = np.empty(msgsize, dtype=int32)
comm.Recv(buf, source=status.source, tag=status.tag)
```

Communication Modes

Presentation

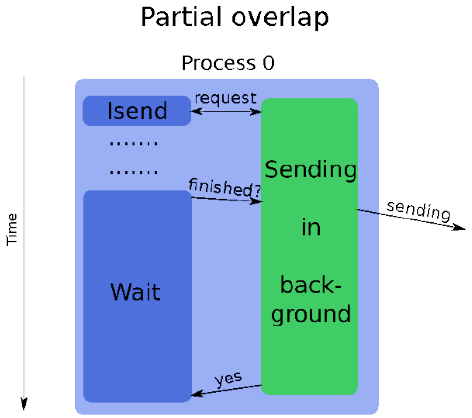
The overlap of communications by computations is a method which allows executing communications operations in the background while the program continues to operate. On Jean Zay, the latency of a communication internode is 1.5 μs, or 2500 processor cycles.

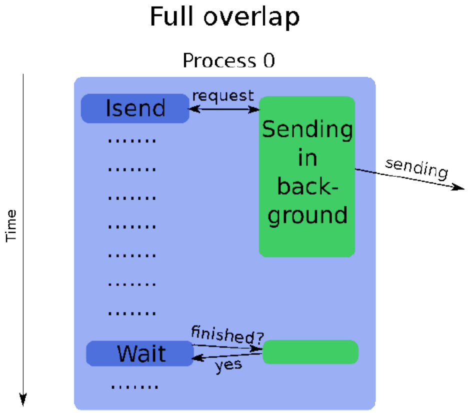
- It is thus possible, if the hardware and software architecture allows it, to hide all or part of communications costs.
- The computation-communication overlap can be seen as an additional level of parallelism.
- This approach is used in MPI by using nonblocking subroutines (i.e. `MPI_Isend()`, `MPI_Irecv()`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

Non blocking communication

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

Communication Modes





Communication Modes

Advantages of non blocking call

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

Disadvantages of non blocking call

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

Communication Modes

Interfaces

MPI_Isend() MPI_Issend() and MPI_Ibsend() for nonblocking send

```
# These functions return an instance of the request class
mpi4py.MPI.Comm.Isend([values, count, datatype], dest=dest, tag=tag)
mpi4py.MPI.Comm.Issend([values, count, datatype], dest=dest, tag=tag)
mpi4py.MPI.Comm.Ibsend([values, count, datatype], dest=dest, tag=tag)
mpi4py.MPI.Comm.isend(obj, dest, tag=0)
mpi4py.MPI.Comm.issend(obj, dest, tag=0)
mpi4py.MPI.Comm.ibsend(obj, dest, tag=0)
```

MPI_Irecv() for nonblocking receive.

```
These functions return an instance of the request class
mpi4py.MPI.Comm.Irecv([values, count, msgtype], source=ANY_SOURCE, tag=ANY_TAG)
mpi4py.MPI.Comm.irecv(buf=None, source=ANY_SOURCE, tag=ANY_TAG)
```

Communication Modes

Interfaces

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

```
mpi4py.MPI.Request.Wait(status=None)
# Return the object received (only for request made with irecv)
mpi4py.MPI.Request.wait(status=None)
# Return the bool
mpi4py.MPI.Request.Test(status=None)
# Return a tuple with a bool and the object received
mpi4py.MPI.Request.test(status=None)
```

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

```
mpi4py.MPI.Request.Waitall(requests, statuses=None)
# Return a list of received objects
mpi4py.MPI.Request.waitall(requests, statuses=None)
# Return a bool showing if all communications are finished
mpi4py.MPI.Request.Testall(requests, statuses=None)
# Return a tuple with a bool and a list of received objects
mpi4py.MPI.Request.testall(requests, statuses=None)
```

Communication Modes

Interfaces

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

```
# Return the index of the finished request
mpi4py.MPI.Request.Waitany(requests, status=None)
# Return a tuple containing the index and the received object if any
mpi4py.MPI.Request.waitany(requests, status=None)
# Returns a tuple containing a boolean indicating whether communication occurred
# and the index of that communication
mpi4py.MPI.Request.Testany(requests, status=None)
# Returns a tuple containing a boolean, the index, and the optionally received
# object
mpi4py.MPI.Request.testany(requests, status=None)
```

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

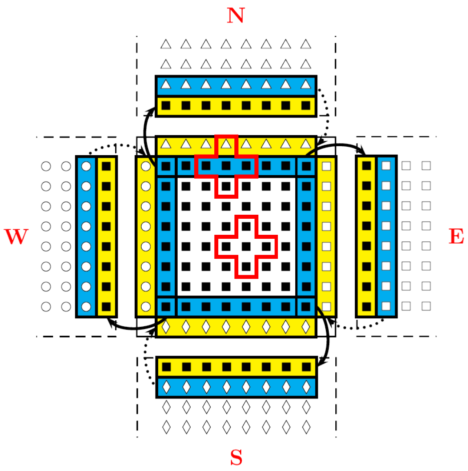
```
# Return the indexes of the finished requests
mpi4py.MPI.Request.Waitsome(requests, statuses=None)
# Return a tuple containing a list of index and a list of received object if any
mpi4py.MPI.Request.waitsome(requests, statuseses=None)
# Return a liste of index or None
mpi4py.MPI.Request.Testsome(requests, statuses=None)
# Returns a tuple containing a list of index, and a list of received object if any
mpi4py.MPI.Request.testsome(requests, statuses=None)
```

Communication Modes

Request management

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

Communication Modes



Communication Modes

```python
def start_communication():
  # Send to the North and receive form the South
  req[0] = comm2d.Isend(sendbuf=[u[], 1, type_line], dest=voisin[N])
  req[1] = comm2d.Irecv(recvbuf=[u[], 1, type_line], source=voisin[S])
  # Send to the North and receive form the South
  req[2] = comm2d.Isend(sendbuf=[u[], 1, type_line], dest=voisin[S])
  req[3] = comm2d.Irecv(recvbuf=[u[], 1, type_line], source=voisin[N])
  # Send to the West and receive from the East
  req[4] = comm2d.Isend(sendbuf=[u[], 1, type_column], dest=voisin[W])
  req[5] = comm2d.Irecv(recvbuf=[u[], 1, type_column], source=voisin[E])
  # Send to the East and receive from the West
  req[6] = comm2d.Isend(sendbuf=[u[], 1, type_column], dest=voisin[E])
  req[7] = comm2d.Irecv(recvbuf=[u[], 1, type_column], source=voisin[W])

def end_communication():
  MPI.Request.Waitall(req)
```

Communication Modes

```python
while not convergence and it< d.it_max :
  it = it+1
  # Swap u and u_new
  u, u_new = u_new, u
  start_communication()
  computation()
  end_communication()
  border_comptutation()
  diffnorm = global_error()
  convergence = diffnorm < 2e-16
```

Communication Modes

Overlap levels on different machines

Machine	*Level*
Jean Zay (Intel MPI)	43%
Jean Zay (Intel MPI) I_MPI_ASYNC_PROGRESS=yes	95%

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

An overlap of 0% means that the total execution time is twice the time of a compute (or a communication) kernel.
An overlap of 100% means that the total execution time is the same as the time of a compute (or a communication) kernel.

MPI 3.x

Nonblocking collectives

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

```
# Return a request
Ibarrier()
```

Communication Modes

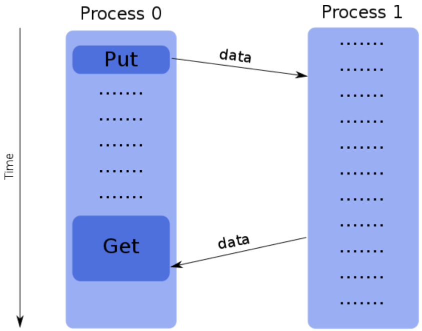
Usage example of `MPI_Ibarrier`
How to manage communications when we don't kown at each iteration if our neighbors will send a message.

```
isAllFinish = False
isMySendFinish = False
reqs = []

# Synchronous send of all messages
for i in range(m):
  reqs.append(comm.Issend(sbuf[i], dest=dst[i], tag))

while not isAllFinish:
  # Check if a message is ready to be received
  if (comm.Iprobe(source=MPI.ANY_SOURCE, tag=tag, status=astat)):
    # Receive the message
    comm.Recv(rbuf, source=astat.source, tag=tag)

  if not isMySendFinish:
    # Check if all  Isend are finished
    if MPI.Request.Testall(reqs):
      # In this case we start the ibarrier
      reqb = comm.Ibarrier(comm)
      isMySendFinish = True
  else:
    # Test if everybody has done the ibarrier
    isAllFinish = reqb.Test()
```

Communication Modes

One-Sided Communications

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



Communication Modes

General approach

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

Synchronization methods

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

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

Communication Modes

```python
from mpi4py import MPI
import numpy as np

# Initialisation
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Global parameters
n = 4
m = 4

# Size of real in bytes
size_real = MPI.DOUBLE.Get_size()

if rank == 0:
    tab = np.zeros(m)

win_local = np.zeros(n)
# Creation of the shared memory window
dim_win = size_real * n
win = MPI.Win.Create(win_local, disp_unit=size_real, comm=comm)
```

Communication Modes

```python
if rank == 0:
    for i in range(m):
        tab[i] = i + 1
else:
    for i in range(n):
        win_local[i] = 0.0

win.Fence()

# Operation on the shared window
if rank == 0:
    target = 1
    nb_elements = 2
    displacement = 1
    win.Put([tab, nb_elements, MPI.DOUBLE], target,
            [displacement, nb_elements, MPI.DOUBLE])

win.Fence()

asum = 0.0
if rank == 0:
    for i in range(m - 1):
        asum += tab[i]
    tab[m - 1] = asum
else:
    for i in range(n - 1):
        asum += win_local[i]
    win_local[n - 1] = asum

win.Fence()

if rank == 0:
    nb_elements = 1
    displacement = m-1
    win.Get([tab, nb_elements, MPI.DOUBLE], target,
            [displacement, nb_elements, MPI.DOUBLE])
```

Communication Modes

Advantages of One-Sided Communications

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

Disadvantages of One-Sided Communications

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

Derived datatypes

Derived datatypes

Introduction

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

Derived datatypes

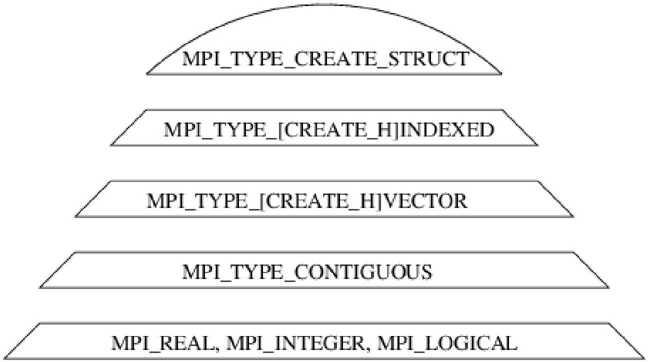
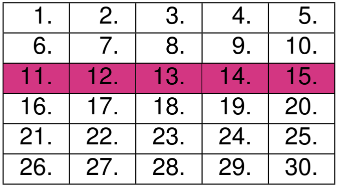

MPI_TYPE_CREATE_STRUCT
MPI_TYPE_[CREATE_H]INDEXED
MPI_TYPE_[CREATE_H]VECTOR
MPI_TYPE_CONTIGUOUS
MPI_REAL, MPI_INTEGER, MPI_LOGICAL

Figure 21 – Hierarchy of the MPI constructors

Derived datatypes

Contiguous datatypes

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



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

```
new_type = MPI.FLOAT.Create_contiguous(5)
```

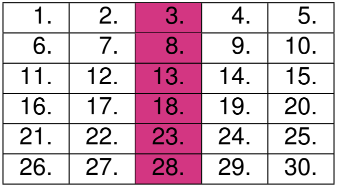
Figure 22 – MPI_Type_contiguous subroutine

```
# Return a type
mpi4py.MPI.Datatype.Create_contiguous(count)
```

Derived datatypes

Constant stride

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



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

```
  new_type = MPI.FLOAT.Create_vector(6, 1, 5)
```

Figure 23 – MPI_Type_vector subroutine

```
# Return a type
mpi4py.MPI.Datatype.Create_vector(count, blocklength, stride)
```

Derived datatypes

Constant stride

- `MPI_Type_create_hvector()` creates a data structure from a homogenous set of existing datatype separated by a constant stride in memory.
  The stride is given in bytes.
- This call is useful when the old type is no longer a base datatype (`MPI_INT`, `MPI_FLOAT`,...) 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.

```
# Return a type
mpi4py.MPI.Datatype.Create_hvector(count, blocklength, stride)
```

Derived datatypes

Commit derived datatypes

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

```
mpi4py.MPI.Datatype.Commit()
```

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

```
mpi4py.MPI.Datatype.Free()
```

Derived datatypes

```python
from mpi4py import MPI
import numpy as np

# Init environment MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Definitions of parameters
nb_lines = 6
nb_columns = 5
tag = 100

# Initialization of the matrix on every process
a = np.full((nb_lines, nb_columns), rank,dtype=np.float32)

# Definition of datatype type_line
type_line = MPI.FLOAT.Create_contiguous(nb_columns)
type_line.Commit()
```

Derived datatypes

```python
if rank == 0:
    # Send the first line
    comm.Send([a, 1, type_line], 1, tag)
else:
    # Receive in last line
    comm.Recv([a[nb_lines-1, 0:], nb_columns, MPI.FLOAT], 0, tag)

# Free type_line
type_line.Free()
```

Derived datatypes

```python
from mpi4py import MPI
import numpy as np

# Initialisation environment MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Definitions of parameters
nb_lines = 6
nb_columns = 5
tag = 100

# Initialisation of the matrix on every prrocess
a = np.full((nb_lines, nb_columns), rank, dtype=np.float32)

# Definition of datatype type_column
type_column = MPI.FLOAT.Create_vector(nb_lines, 1, nb_columns)
type_column.Commit()
```

Derived datatypes

```python
if rank == 0:
    # Send
    comm.Send([a[0, 0:], nb_lines, MPI.FLOAT], 1, tag)
else:
    # Receive in second-to-last column
    comm.Recv([a[0, nb_columns-2:], 1, type_column], 0, tag)

# Free the datatype
type_column.Free()
```

Derived datatypes

```python
from mpi4py import MPI
import numpy as np

# Initi environnment MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Definitions of parameters
nb_lines = 6
nb_columns = 5
tag = 100
nb_lines_bloc = 3
nb_columns_bloc = 2

# Init of matrix on every process
a = np.full((nb_lines, nb_columnns), rank, dtype=np.float32)

# Definition of datatype type_bloc
type_bloc = MPI.FLOAT.Create_vector(nb_lines_bloc, nb_columns_bloc,
                                    nb_columns)
type_bloc.Commit()
```

Derived datatypes

```
if rank == 0:
    # Send a block
    comm.Send([a, 1, type_bloc], 1, tag)
else:
    # Receive a  block
    comm.Recv([a[nb_lines-3, nb_colums-2:], 1, type_bloc], 0, tag)

# Free the datatype
type_bloc.Free()
```

Derived datatypes

Homogenous datatypes of variable strides

- `MPI_Type_indexed()` allows creating a data structure composed of a sequence of blocks containing a variable number of elements separated by a variable stride in memory. The stride is given in number of elements.
- `MPI_Type_create_hindexed()` has the same functionality as `MPI_Type_indexed()` except that the strides separating two data blocks are given in bytes. This subroutine is useful when the old datatype is not an MPI base datatype(`MPI_INT`, `MPI_FLOAT`, ...). We cannot therefore give the stride in number of elements of the old datatype.
- For `MPI_Type_create_hindexed()`, as for `MPI_Type_create_hvector()`, use `MPI_Type_size()` or `MPI_Type_get_extent()` in order to obtain in a portable way the size of the stride in bytes.

Derived datatypes

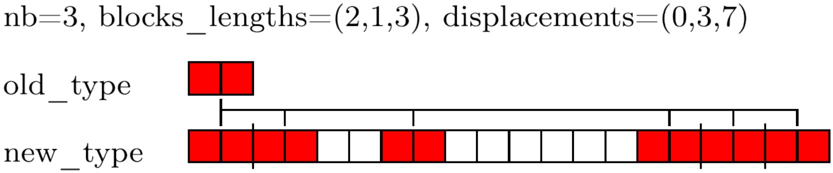


Figure 24 – The `MPI_Type_indexed` constructor

```
# Return a datatype
mpi4py.MPI.Create_indexed(blocklengths, displacements)
```

Derived datatypes

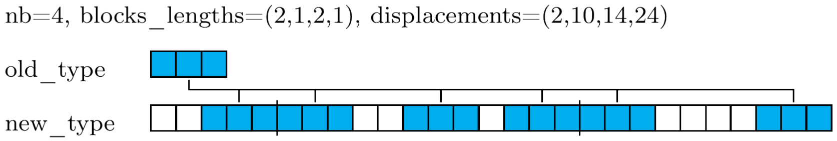


Figure 25 – The `MPI_Type_create_hindexed` constructor

```
# Return a datatype
mpi4py.MPI.Datatype.Create_hindexed(blocklengths, displacements)
```

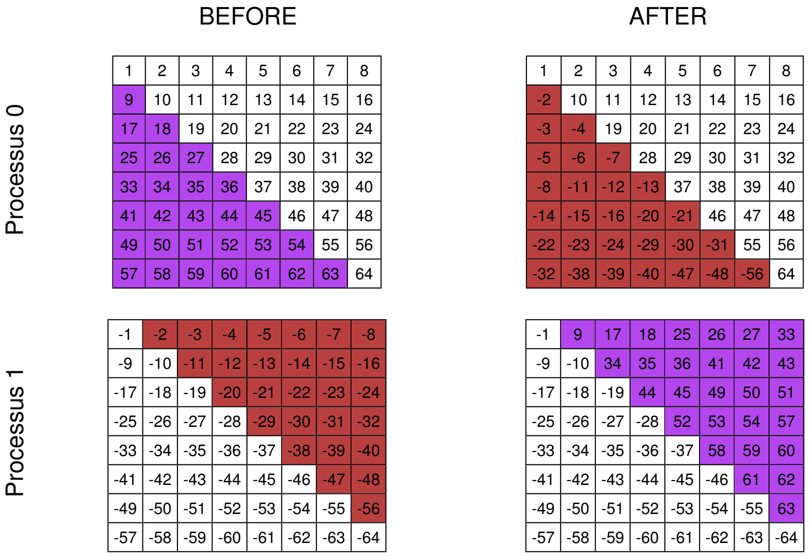
Derived datatypes

Example : triangular matrix

In the following example, each of the two processes :

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

Derived datatypes



Processus 0 — BEFORE

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

Processus 0 — AFTER

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

Processus 1 — BEFORE

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

Processus 1 — AFTER

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

Derived datatypes

```python
from mpi4py import MPI
import numpy as np

# Parameters
n = 8
tag = 100

# Creation of the matrix a
a = np.zeros((n, n), dtype=np.float32)

# Environnment MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

sign = 1
if rank == 1:
    sign = -1

# Initialisation of the matrix in every process
for i in range(n):
    for j in range(n):
        a[i, j] = sign * (1 + i * n + j)

# Creation of datatype type_triangle
block_lengths = [i if rank == 0 else n - i - 1 for i in range(n)]
displacements = [n * i if rank == 0 else (n + 1) * i + 1 for i in range(n)]

type_triangle = MPI.FLOAT.Create_indexed(block_lengths, displacements)
type_triangle.Commit()

# Swap of matrix
comm.Sendrecv_replace([a, 1, type_triangle],
                      source=(rank + 1) % 2, sendtag=tag,
                      dest=(rank + 1) % 2, recvtag=tag)

# Free datatype triangle
type_triangle.Free()
```

Derived datatypes

Size of datatype

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

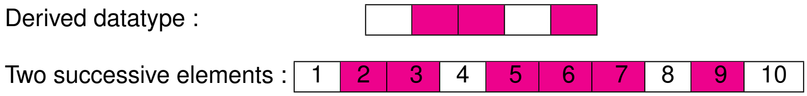
```
# Return the size in bytes of a datatype
mpi4py.MPI.Datatype.Get_size()
```

- 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).

```
# Return a tuple with the lower bound and the extent
mpi4py.MPI.Datatype.Get_extent()
```

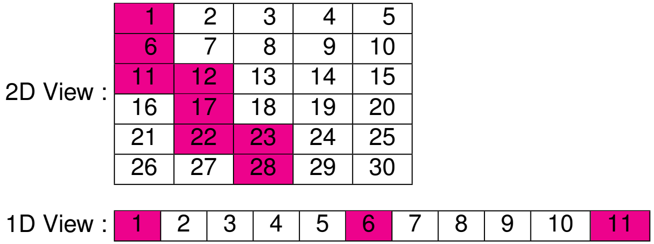
Derived datatypes

Example 1 : `MPI_Type_indexed(2,{2,1},{1,4},MPI_INT,&type)`



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

Example 2 : `MPI_Type_vector(3,1,nb_columns,MPI_INT,&type_half_column)`



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

Derived datatypes

Modify the extent

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

```
# Return a datatype
mpi4py.MPI.Datatype.Create_resized(lb, extent)
```

Derived datatypes

```python
from mpi4py import MPI
import numpy as np

# Parameters
nb_lines = 6
nb_columns = 5
tag = 100

# Get rank of process
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

sign = 1
if rank == 1:
    sign = -1

# Init matrix on every process
a = np.zeros((nb_lines, nb_columns), dtype=np.int32)
for i in range(nb_lines):
    for j in range(nb_columns):
        a[i, j] = sign * (1 + nb_columns * i + j)

# Build the datatype type_half_column1
size_half_column = nb_lines // 2
type_half_column1 = MPI.INT.Create_vector(size_half_column, 1, nb_columns)
type_half_column1.Commit()

# Size of MPI.INT
size_integer = MPI.INT.Get_size()

# Information about the datatype type_half_column1
lower_bound1, extend1 = type_half_column1.Get_extent()
if rank == 0:
    print(f"type_half_column1: lower_bound={lower_bound1}, extend={extend1}")
```

Derived datatype

```python
# Build datatype type_half_column2
lower_bound2 = 0
extend2 = size_integer
type_half_column2 = type_half_column1.Create_resized(lower_bound2, extend2)
type_half_column2.Commit()

# Information about the datatype type_half_column2
lower_bound2, extend2 = type_half_column2.Get_extent()
if rank == 0:
    print(f"type_half_column2: lower_bound={lower_bound2}, extend={extend2}")

# Send matrix to process 1 with datatype demi_colonne2
if rank == 0:
    comm.Send([a, 2, type_half_column2], dest=1, tag=tag)
else:
    # Receive from process 0
    comm.Recv([a[nb_lines-2, 0:], 6, MPI.INT], source=0, tag=tag)
```

```
> mpiexec -n 2 python -m mpi4py half_column.py
 type_half_column1: lower_bound=0, extend=44
 type_half_column2: lower_bound=0, extend=4
```

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

Derived datatypes

Conclusion

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

Derived datatypes

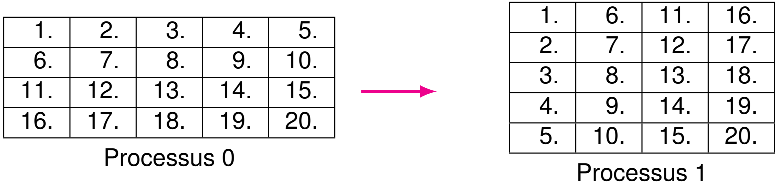
Memento

Subroutines	`blocks_lengths`	`strides`	`old_types`
`MPI_Type_Contiguous()`	constant*	constant*	constant
`MPI_Type_[Create_H]Vector()`	constant	constant	constant
`MPI_Type_[Create_H]Indexed()`	*variable*	*variable*	constant
`MPI_Type_Create_Struct()`	*variable*	*variable*	*variable*

(*) hidden parameter, equal to 1

MPI Hands-On – Exercise 4 : Matrix transpose

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



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

Processus 0

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

Processus 1

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

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

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

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

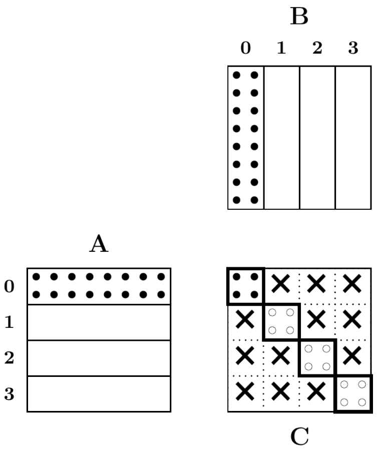


Figure 26 – Distributed matrix product

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

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

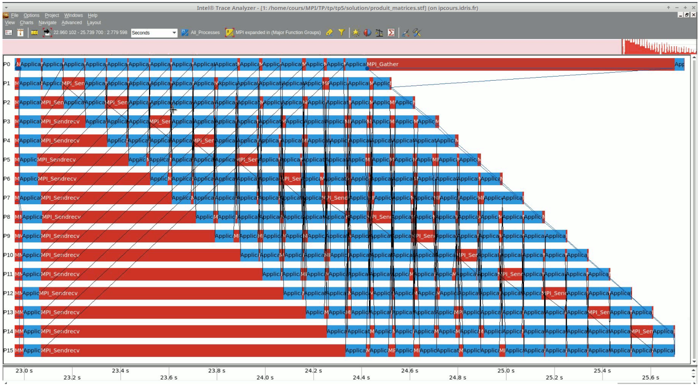

Intel® Trace Analyzer - [1: /home/cours/MPI/TP/tp/tp5/solution/produit_matrices.stf] (on ipcours.idris.fr)
File Options Project Windows Help
View Charts Navigate Advanced Layout
22.960 102 - 25.739 700 : 2.779 598
Seconds
All_Processes
MPI expanded in (Major Function Groups)
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
MPI_Gather
MPI_Sendrecv
Application
23.0 s
23.2 s
23.4 s
23.6 s
23.8 s
24.0 s
24.2 s
24.4 s
24.6 s
24.8 s
25.0 s
25.2 s
25.4 s
25.6 s

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

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

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

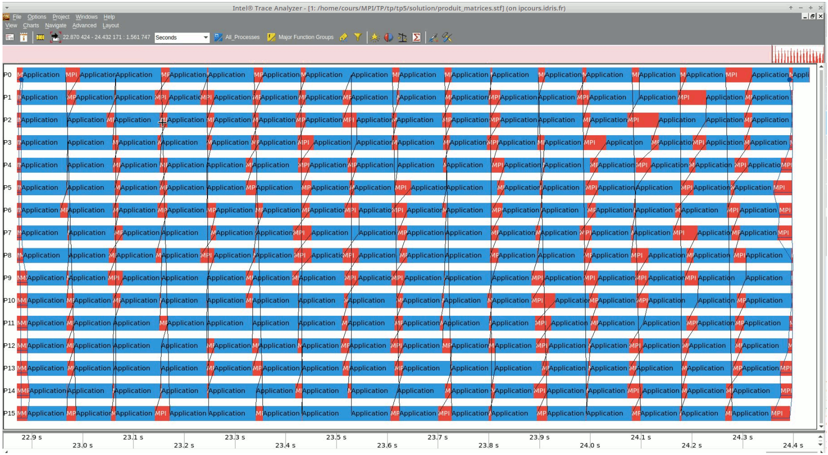


Figure 28 – Parallel matrix product on 16 processes, for a matrix size of 1024 (second algorithm)

Communicators

Communicators

Introduction

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

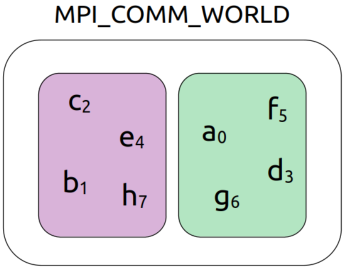


Figure 29 – Communicator partitioning

Communicators

Example

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

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

Communicators

Default communicator

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

Communicators

Groups and communicators

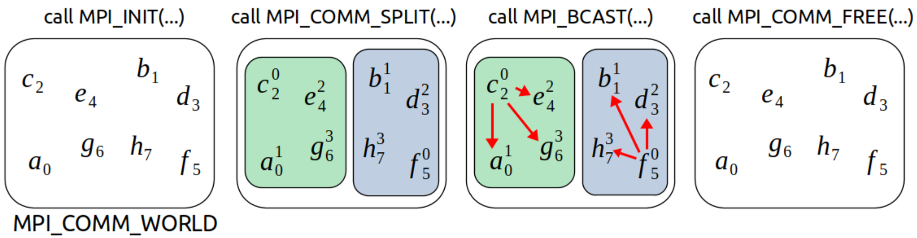
- A communicator consists of :
  - A group, which is an ordered group of processes.
  - A communication context put in place by calling one of the communicator construction subroutines, which allows determination of the communication space.
- The communication contexts are managed by MPI (the programmer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

Communicators

Partitioning of a communicator

In order to solve the problem example :

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



call MPI_INIT(...)
call MPI_COMM_SPLIT(...)
call MPI_BCAST(...)
call MPI_COMM_FREE(...)
c_2 e_4 b_1 d_3 a_0 g_6 h_7 f_5
MPI_COMM_WORLD
c_2^0 e_4^2 a_0^1 g_6^3 b_1^1 d_3^2 h_7^3 f_5^0

Figure 30 – Communicator creation/destruction

Communicators

Partitioning of a communicator with `MPI_Comm_split()`

The `MPI_Comm_split()` subroutine allows :

- Partitioning a given communicator into as many communicators as we want.
- Giving the same name to all these communicators : it will have the value of the communicator containing the current process.
- 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

```
# Return a communicator
mpi4py.MPI.Comm.Split(color=0, key=0)
```

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

Communicators

Example

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

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

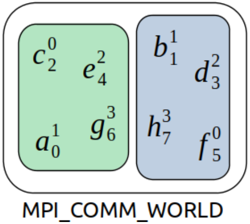
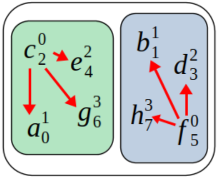


Figure 31 – Construction of the `CommEvenOdd` communicator with `MPI_Comm_split()`

Communicators

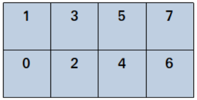
```python
from mpi4py import MPI
import numpy as np

# Initialisation of matrix and parameters
m = 16
a = np.zeros(m, dtype=np.float32)

# Get the rank
comm = MPI.COMM_WORLD
rank_in_world = comm.Get_rank()

# Initialisation of matrix a
for i in range(m):
    a[i] = 0.
if rank_in_world == 2:
    for i in range(m):
        a[i] = 2.
if rank_in_world == 5:
    for i in range(m):
        a[i] = 5.

# Define key for splitting into even and odd
key = rank_in_world
if (rank_in_world == 2) or (rank_in_world == 5):
    key = -1

# Creation of evenodd communicator
commEvenOdd = comm.Split(rank_in_world % 2, key)

# Broadcast a message by the process 0 of every communicator
# to other process in the group
commEvenOdd.Bcast([a, m, MPI.FLOAT], root=0)

# Free communicator
commEvenOdd.Free()
```



Communicators

Topologies

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



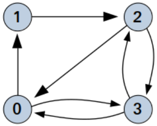


Figure 32 – A 2D Cartesian topology (left) and a Graph topology (right)

Communicators

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.

```
mpi4py.MPI.Intracomm.Create_cart(dims, periods=None, reorder=False)
```

Communicators

Example

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

```
dims = [4,2]
periods = [False, True]
comm_2D = MPI.COMM_WORLD.Create_cart(dims, periods)
```

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.

Communicators

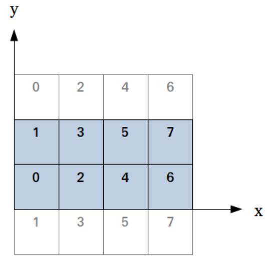


Figure 33 – A 2D periodic Cartesian topology in y

Communicators

3D Example

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

```
dims = [4,2,2]
comm_3D = MPI.COMM_WORLD.Create_cart(dims)
```

Communicators

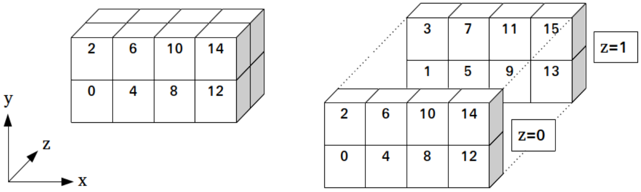


Figure 34 – A 3D non-periodic Cartesian topology

Communicators

Process distribution

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

```
# Return a balanced distribution of processes per coordinate direction
mpi4py.MPI.Compute_dims(nnodes,dims)
```

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

`dims` in entry	`Compute_dims`	`dims` in exit
(0,0)	(8,dims)	(4,2)
(0,0,0)	(16,dims)	(4,2,2)
(0,4,0)	(16,dims)	(2,4,2)
(0,3,0)	(16,dims)	`error`

Communicateurs

Rank and coordinates of a process

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

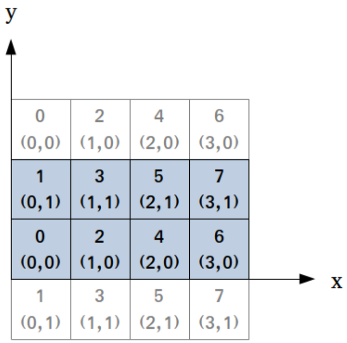


Figure 35 – A 2D periodic Cartesian topology in y

Communicators

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.

```
# Retourne le rang
mpi4py.MPI.Cartcomm.Get_cart_rank(coords)
```

Communicators

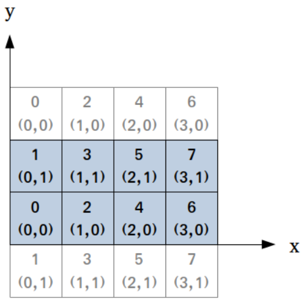


Figure 36 – A 2D periodic Cartesian topology in y

```
coords[0] = dims[0]-1
for i in range(dims[1]):
  coords[1]=i
  rang[i] = comm_2D.Get_cart_rank(coords)
............................................
i=0,in entry coords=[3,0],in exit rang[0]=6.
i=1,in entry coords=[3,1],in exit rang[1]=7.
```

Communicators

Coordinates of a process

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

```
# Return coordinates
mpi4py.MPI.Cartcomm.Get_coords(rank)
```

Communicators

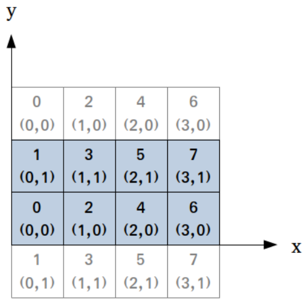


Figure 37 – A 2D periodic Cartesian topology in y

```
if (rank%2 == 0):
  coords = comm_2D.Get_coords(rank)
............................................
In entry, the rank values are : 0,2,4,6.
In exit, the coords values are :
(0,0),(1,0),(2,0),(3,0).
```

Communicators

Rank of neighbours

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

```
# Retourne un tuple (rang_precedent, rang_suivant)
mpi4py.MPI.Cartcomm.Shift(direction, disp)
```

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

Communicators

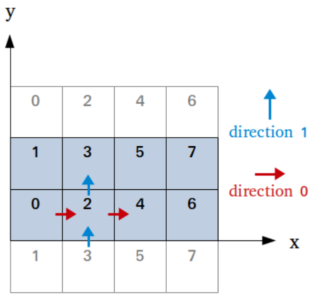


Figure 38 – Call of the MPI_Cart_shift() subroutine

```
rank_left, rank_right = comm_2D.Shift(0,1)
............................................
For the process 2, rank_left=0, rank_right=4
```

```
rank_low, rank_high = comm_2D.Shift(1,1)
............................................
For the process 2, rank_low=3, rank_high=3
```

Communicators

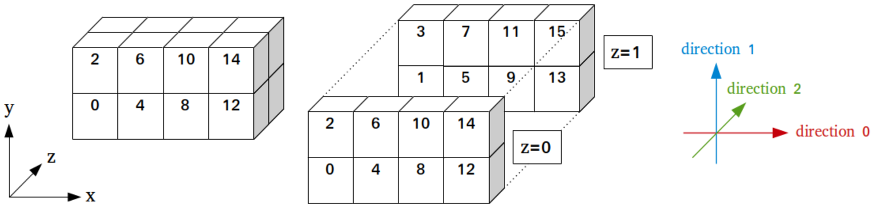


Figure 39 – Call of the MPI_Cart_shift() subroutine

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

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

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

Communicators

Example

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

```python
from mpi4py import MPI

# Parameters
ndims = 2
N = 1
E = 2
S = 3
W = 4

comm = MPI.COMM_WORLD
nb_procs = comm.Get_size()

# Know the number of processes along x and y
dims = [0, 0]
dims = MPI.Compute_dims(nb_procs, dims)
```

Communicators

```python
# 2D y-periodic grid creation
periods = [False, True]
comm_2D = comm.Create_cart(dims, periods)

# Know my coordinates in the topology
rang_ds_topo = comm_2D.Get_rank()
coords = comm_2D.Get_coords(rang_ds_topo)

# Search of my West and East neighbors
voisin_W, voisin_E = comm_2D.Shift(0, 1)

# Search of my South and North neighbors
voisin_S, voisin_N = comm_2D.Shift(1, 1)
```

Subdividing a Cartesian topology

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

Communicators

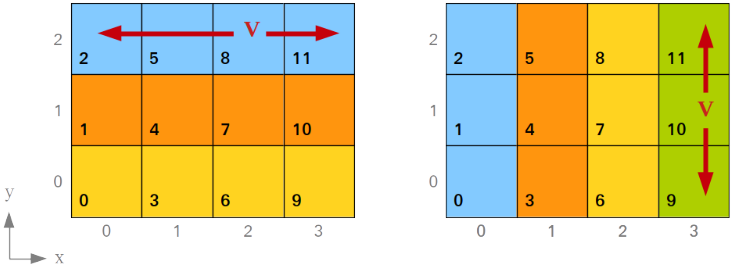


Figure 40 – Two examples of data distribution in a degenerated 2D topology

Communicators

Subdividing a Cartesian topology

There are two ways to degenerate a topology :

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

```
# Return a topology
mpi4py.MPI.Cartcomm.Sub(remain_dims)
```

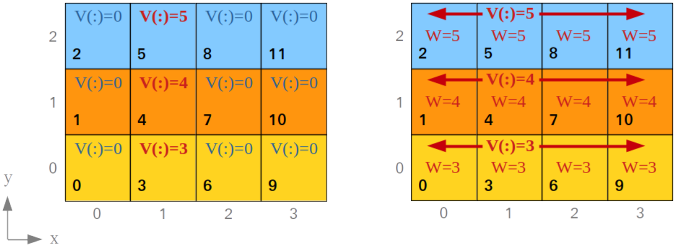


Figure 41 – Scatter of the *V* array in the degenerated 2D grid.

Communicators

```python
from mpi4py import MPI
import numpy as np

# Parameters
NDim2D = 2
m = 4
Dim2D = [4, 3]

# Creation of the initial 2D grid
comm = MPI.COMM_WORLD
comm_2D = comm.Create_cart(Dim2D)

# Compute rank and coordinates of the process
rank = comm_2D.Get_rank()
Coord2D = comm_2D.Get_coords(rang)
```

Communicators

```python
# Initialisation of V vector
if Coord2D[0] == 1:
    V = np.full(m, rank,dtype=np.float32)
else:
    V = np.zeros(m,dtype=np.float32)

# Every line of grid must be in 1D cartesian topology
remain_dims = [1, 0]

# Subdivision of 2D cartesian grid
comm_1D = comm_2D.Sub(remain_dims)

# Processes of the column 2 distribute the vector V
# to the processes of their line
W = np.zeros(1, dtype=np.float32)
comm_1D.Scatter([V, 1, MPI.FLOAT], [W, 1, MPI.FLOAT], root=1)

# Print results
print(f"Rank : {rank} ; Coordinates : ({Coord2D[0]}, {Coord2D[1]});"
      f" W = {W[0]}")
```

Communicators

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

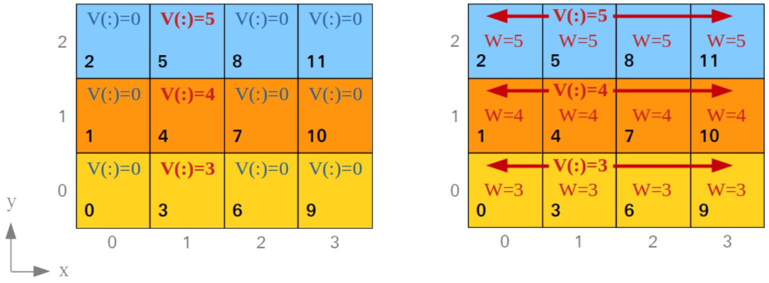


Figure 42 – Scatter of the *V* array in the degenerated 2D grid.

MPI Hands-On – Exercise 6 : Communicators

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

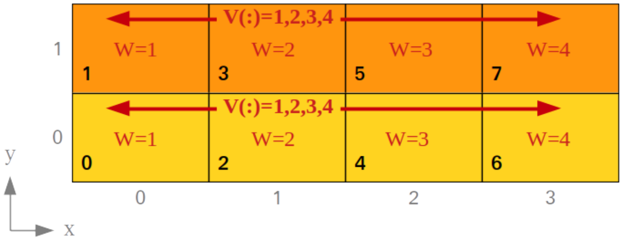


Figure 43 – Subdivision of a 2D topology and communication using the obtained 1D topology

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

MPI-IO

Input/Output Optimisation

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

MPI-IO

The MPI-IO interface

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

MPI-IO

Example of a sequential optimisation implemented by I/O libraries

- I/O performance suffers considerably when making many small I/O requests (latencies).
- 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.

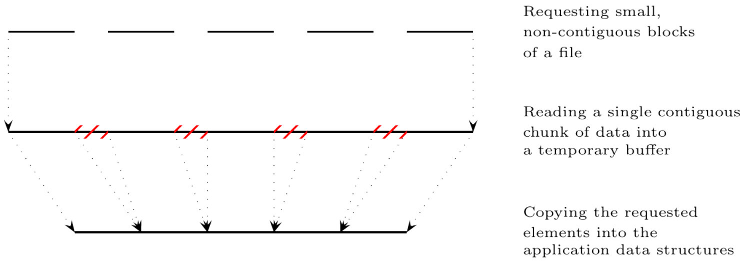

Requesting small, non-contiguous blocks of a file
Reading a single contiguous chunk of data into a temporary buffer
Copying the requested elements into the application data structures

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

MPI-IO

Example of a parallel optimisation

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

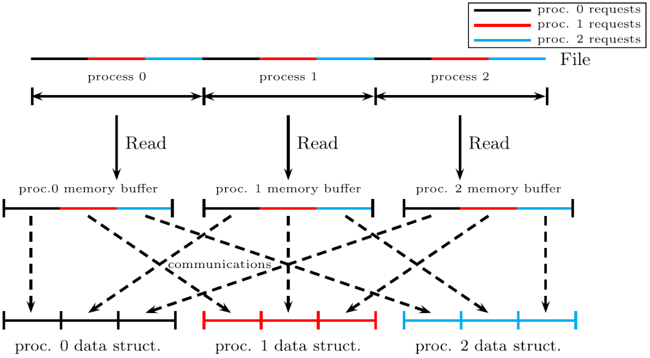


Figure 45 – Read operation performed in two steps by a group of processes

MPI-IO

Open a file

```
# Return a file descriptor
mpi4py.MPI.File.Open(comm, filename, amode=MODE_RDONLY, info=INFO_NULL)
```

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

MPI-IO

Mode

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

Modes can be combined with operator \|.

MPI-IO

Closing a file

```
mpi4py.MPI.File.Close()
```

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

MPI-IO

```python
from mpi4py import MPI

# Open the file
fh = MPI.File.Open(MPI.COMM_WORLD, "fichier.txt",
                   MPI.MODE_RDWR | MPI.MODE_CREATE)

# In python no need to check

# Close the file
fh.Close()
```

```
> ls -l file.data

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

MPI-IO

Data access routines

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

MPI-IO

Positioning	Synchronism	noncollective	collective
explicit offsets	blocking	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	nonblocking	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_iread_at_all MPI_File_iwrite_at_all
individual file pointers	blocking	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	nonblocking	MPI_File_iread MPI_File_iwrite	MPI_File_iread_all MPI_File_iwrite_all
shared file pointer	blocking	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	nonblocking	MPI_File_iread_shared MPI_File_iwrite_shared	MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end

MPI-IO

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`.

MPI-IO

Explicit Offsets

```
mpi4py.MPI.File.Read_at(offset, [buf, count, datatype], status=None)
mpi4py.MPI.File.Write_at(offset, [buf, count, datatype], status=None)
```

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

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

for i in range(nb_values):
    values[i] = i + 1 + rank * 100
print(f"process {rank} : {values}")

# Open the file
fh = MPI.File.Open(comm, "donnees.dat",
                   MPI.MODE_WRONLY | MPI.MODE_CREATE)

# Compute the file position
bytes_in_integer = MPI.INT.Get_size()
offset = rank * nb_values * bytes_in_integer

# Write values
fh.Write_at(offset, values)

# Close the file
fh.Close()
```

MPI-IO

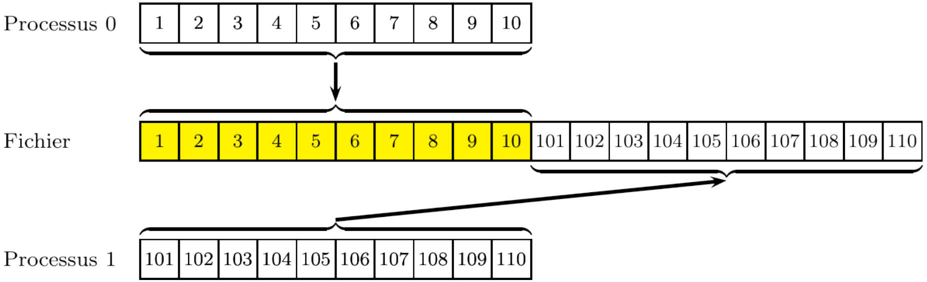


Figure 46 – `MPI_File_write_at()`

```
> mpiexec -n 2 python -m mpi4py write_at.py

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

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

# Open the file
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Compute the offset
bytes_in_int = MPI.INT.Get_size()
offset = rank * nb_values * bytes_in_int

# Read the data
fh.Read_at(offset, values)
print(f"process {rank} : {values}")

# Close the file
fh.Close()
```

MPI-IO

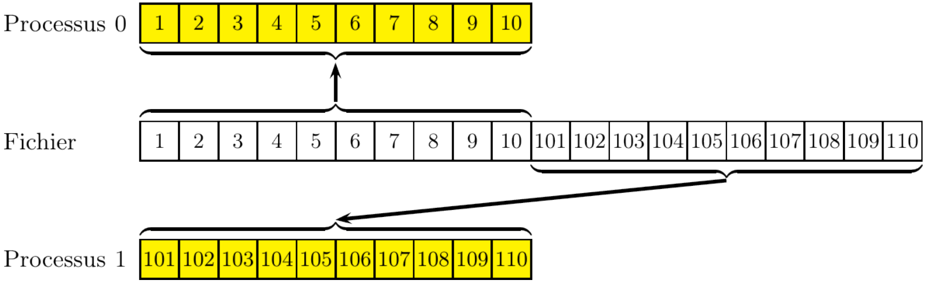


Figure 47 – `MPI_File_read_at()`

```
> mpiexec -n 2 python -m mpi4py read_at.py

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

MPI-IO

Individual file pointers

```
mpi4py.MPI.File.Read([buf, count, datatype], status=None)
mpi4py.MPI.File.Write([buf, count, datatype], status=None)
```

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

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

# Open the file
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Read
fh.Read([values, 6])
fh.Read([values[6:], 4])
print(f"Lecture processus {rank} : {values}")

# Close the file
fh.Close()
```

MPI-IO

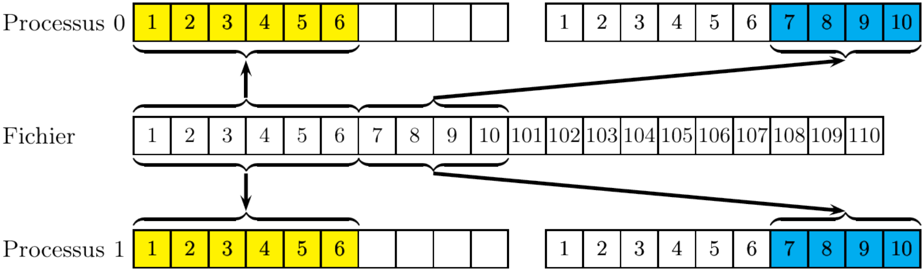


Figure 48 – Example 1 of `MPI_File_read()`

```
> mpiexec -n 2 python -m mpi4py read01.py

processus 1 :  [1  2  3  4  5  6  7  8  9  10]
processus 0 :  [1  2  3  4  5  6  7  8  9  10]
```

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

# Open the file
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Read data
if rank == 0:
    fh.Read([values, 5])
else:
    fh.Read([values, 8])
    fh.Read([values, 5])
print(f"process {rank} : {values[:8]}")

# Close the file
fh.Close()
```

MPI-IO

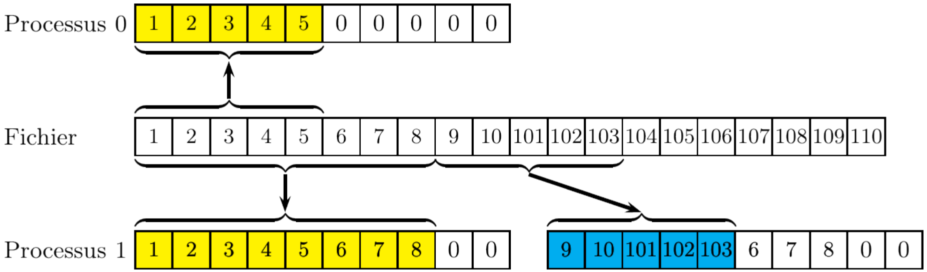


Figure 49 – Example 2 of `MPI_File_read()`

```
> mpiexec -n 2 python -m mpi4py read02.py

processus 0 : [1 2 3 4 5 0 0 0]
processus 1 : [   9  10 101 102 103    6    7    8]
```

MPI-IO

Shared file pointers

```
mpi4py.MPI.File.Read_shared([buf, count, datatype], status=None)
mpi4py.MPI.File.Write_shared([buf, count, datatype], status=None)
```

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

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

# Open the file
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Read data with shared pointer
fh.Read_shared([values, 4, MPI.INT])
fh.Read_shared([values[4:], 6, MPI.INT])

print(f"process {rank} : {values}")

# Close the file
fh.Close()
```

MPI-IO

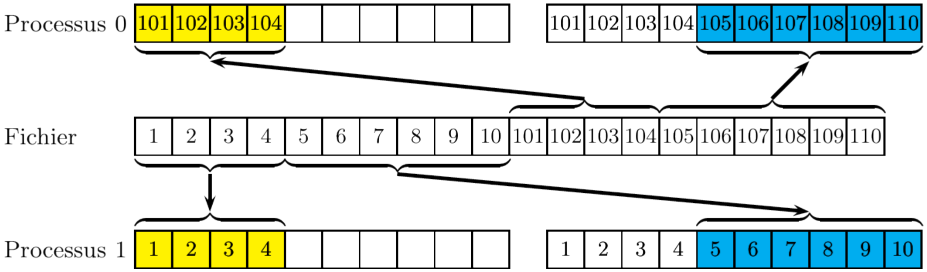


Figure 50 – Example of `MPI_File_read_shared()`

```
> mpiexec -n 2 read_shared01

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

MPI-IO

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.

MPI-IO

Interfaces

```
mpi4py.MPI.File.Read_at_all(offset, [buf, count, datatype], status=None)
mpi4py.MPI.File.Write_at_all(offset, [buf, count, datatype], status=None)
mpi4py.MPI.File.Read_all([buf, count, datatype], status=None)
mpi4py.MPI.File.Write_all([buf, count, datatype], status=None)
mpi4py.MPI.File.Read_ordered([buf, count, datatype], status=None)
mpi4py.MPI.File.Write_ordered([buf, count, datatype], status=None)
```

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

# Open the file
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Compute offset
nb_bytes_int = MPI.INT.Get_size()
offset = rank * nb_values * nb_bytes_int

# Read data
fh.Read_at_all(offset, values)
print(f"process {rank} : {values}")

# Close the file
fh.Close()
```

MPI-IO

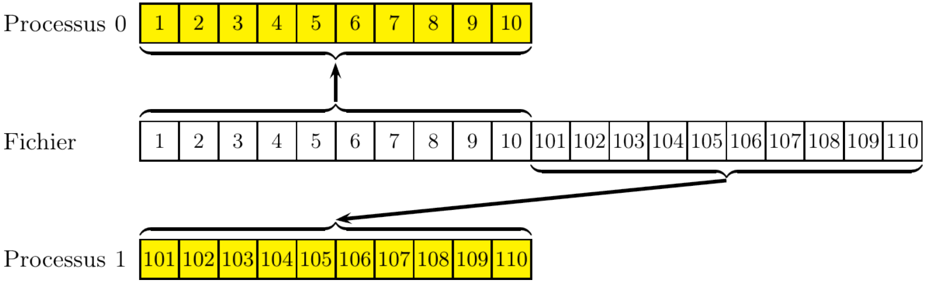


Figure 51 – Example of `MPI_File_read_at_all()`

```
> mpiexec -n 2 python -m mpi4py read_at_all.py

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

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

# Open the file
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Read data
fh.Read_all([values, 6])
fh.Read_all([values[6:], 4])
print(f"process {rank} : {values}")

# Close the file
fh.Close()
```

MPI-IO

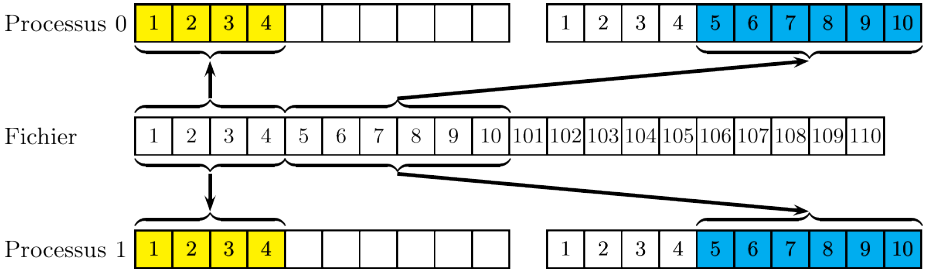


Figure 52 – Example 1 of `MPI_File_read_all()`

```
> mpiexec -n 2 python -m mpi4py readall01.py

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

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

# Open the file
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Read the data
if rank == 0:
    fh.Read_all([values[2:], 4])
else:
    fh.Read_all([values[4:], 5])
print(f"process {rank} : {values}")

# Close the file
fh.Close()
```

MPI-IO

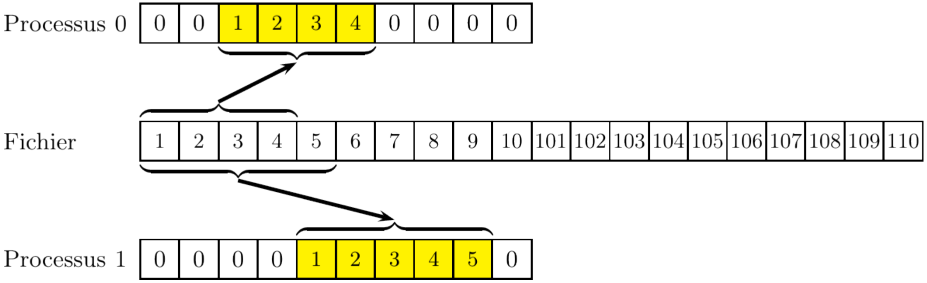


Figure 53 – Example 2 of `MPI_File_read_all()`

```
> mpiexec -n 2 python -m mpi4py ./read_all02.py

process 0 : [0 0 1 2 3 4 0 0 0 0]
process 1 : [0 0 0 0 1 2 3 4 5 0]
```

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

# Open the file
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Read data
if rank == 0:
    fh.Read_all([values[2:], 4])
else:
    fh.Read_all([values[4:], 5])
print(f"process {rank} : {values}")

# Close the file
fh.Close()
```

MPI-IO

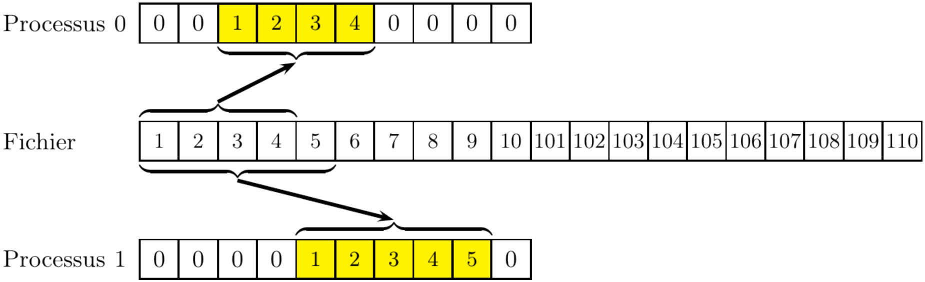


Figure 54 – Example 3 of `MPI_File_read_all()`

```
> mpiexec -n 2 python -m mpi4py ./read_all02.py

process 0 : [0 0 1 2 3 4 0 0 0 0]
process 1 : [0 0 0 0 1 2 3 4 5 0]
```

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

# Open the file
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Read data with shared pointer
fh.Read_ordered([values, 4, MPI.INT])
fh.Read_ordered([values[4:], 6, MPI.INT])

print(f"process {rank} : {values}")

# Close the file
fh.Close()
```

MPI-IO

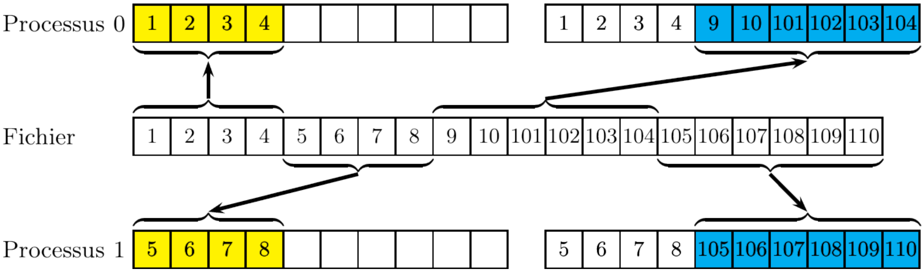


Figure 55 – Example of `MPI_File_ordered()`

```
> mpiexec -n 2 python -m mpi4py ./readOrdered.py

Lecture processus 0 : [   1    2    3    4    9   10 101 102 103 104]
Lecture processus 1 : [   5    6    7    8 105 106 107 108 109 110]
```

MPI-IO

Positioning the file pointers

```
mpi4py.MPI.File.Seek(offset, whence=SEEK_SET)
mpi4py.MPI.File.Seek_shared(offset, whence=SEEK_SET)
```

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

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)

fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
fh.Read([values, 3, MPI.INT])
nb_bytes_int = MPI.INT.Get_size()
offset = 8 * nb_bytes_int
fh.Seek(offset, MPI.SEEK_CUR)
fh.Read([values[3:], 3, MPI.INT])
offset = 4 * nb_bytes_int
fh.Seek(offset, MPI.SEEK_SET)
fh.Read([values[6:], 4, MPI.INT])
print(f"Lecture processus {rank} : {values}")
fh.Close()
```

MPI-IO

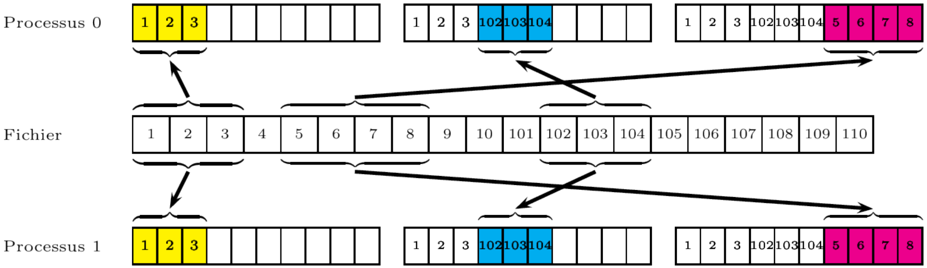


Figure 56 – Example of `MPI_File_seek()`

```
> mpiexec -n 2 python -m mpi4py ./seek.py

process 1 : [   1    2    3 102 103 104    5    6    7    8]
process 0 : [   1    2    3 102 103 104    5    6    7    8]
```

MPI-IO

Access to end of file

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

MPI-IO

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.).

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)
```

MPI-IO

```python
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
nb_bytes_int = MPI.INT.Get_size()
offset = rank * nb_values * nb_bytes_int
request = fh.Iread_at(offset, values)

nb_iterations = 0
finish = False
while nb_iterations < 5000 and not finish:
    nb_iterations += 1
    # Computation to overlap I/O operation
    # ...
    finish = request.Test()
if not finish:
    request.Wait()
print(f"After {nb_iterations} iterations, read process "
      f"{rank} : {values}")
fh.Close()
```

MPI-IO

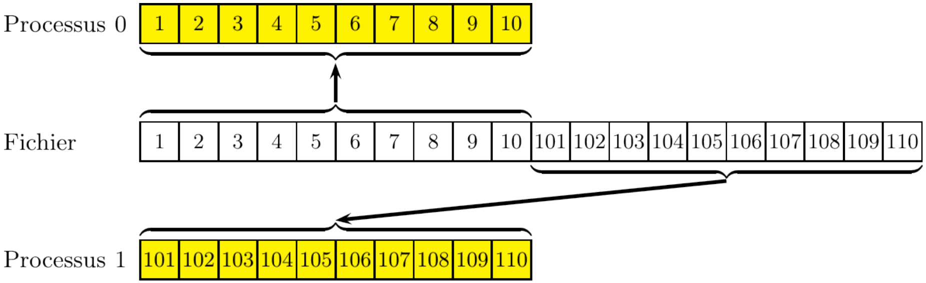


Figure 57 – Example of `MPI_File_iread_at()`

```
> mpiexec -n 2 python -m mpi4py iread_at.py

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

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)
temp = np.zeros(nb_values, dtype=np.int32)

fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_WRONLY | MPI.MODE_CREATE)
temp = values
nb_iterations = 0
request = fh.Iwrite(temp)
while nb_iterations < 5000:
    nb_iterations += 1
    finished = request.Test()
    if finished:
        temp = values
        fh.Seek(offset, MPI.SEEK_SET)
        request = fh.Iwrite(temp)
request.Wait()
fh.Close()
```

MPI-IO

Nonblocking and collective data access routines

- It is possible to perform operations that are both collective and nonblocking.
- The I/O operation is initiated with `MPI_File_iread(_at)_all` or `MPI_File_iwrite(_at)_all` and ended with a separate request call, but for shared file pointers we have split collective nonblocking operations `MPI_File_read_ordered_begin` / `MPI_File_read_ordered_end` or `MPI_File_write_ordered_begin` / `MPI_File_write_ordered_end`. Only one such split collective nonblocking operation can be in progress on any file handle at a time.
- Between the two parts of the collective nonblocking operation, operations on the file are allowed, but the memory region involved in the collective operation cannot be modified.

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_values = 10
values = np.zeros(nb_values, dtype=np.int32)
fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
fh.Read_ordered_begin([values,4,MPI.INT])
print(f"Process {rank}")
fh.Read_ordered_end([values,4,MPI.INT])
print(f"Process {rank} : {values[0]} {values[1]} {values[2]} {values[3]}")
fh.Close()
```

MPI-IO

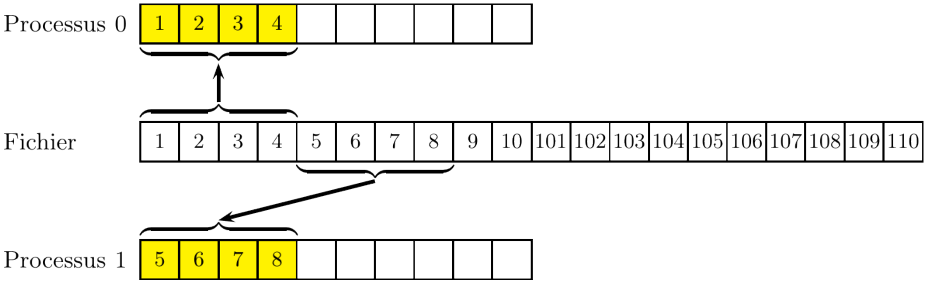


Figure 58 – Example of `MPI_File_read_ordered_begin/end()`

```
> mpiexec -n 2 python -m mpi4py ./readOrderedBeginEnd.py

Process 0
Process 1
process 1 : 1 2 3 4
process 0 : 5 6 7 8
```

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

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

MPI Version

MPI Version

MPI Version

It is possible to know the version of the MPI library used. The version is represented by two integers `MPI.VERSION` and `MPI.SUBVERSION`.

```python
from mpi4py import MPI

print(f"MPI Version : {MPI.VERSION}.{MPI.SUBVERSION}")
```

```
> mpiexec -n 1 python -m mpi4py version.py

MPI Version : 3.1
```

MPI 4.0

Adding

- Large count
- Partitioned communication
- MPI Session
- Others

Large count

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

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

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

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

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

Partitioned communication

- Multiple contributions to a communication.
- Useful in hybrid.
- Init with `MPI_Psend_init()` or `MPI_Precv_init()` by providing the number of partitions and the number of elements by partition.
- `MPI_Start()` to start the communication.
- `MPI_Pready()` to indicate that a partition is ready to be sent.
- `MPI_Parrived()` to know if a partition has been received.
- `MPI_Wait()` to wait for the end of the communication.
- Cannot mix `MPI_Recv()` and `MPI_Psend_init()`.

Session

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

Others

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

MPI 4.1

Ajout

- File mpif.h is deprecated

MPI 5.0

Ajout

- Definition of an ABI. The advantage of an ABI is to be able to run a program using a different MPI library than the one used during compilation.

MPI-IO Views

MPI-IO Views

The View Mechanism

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

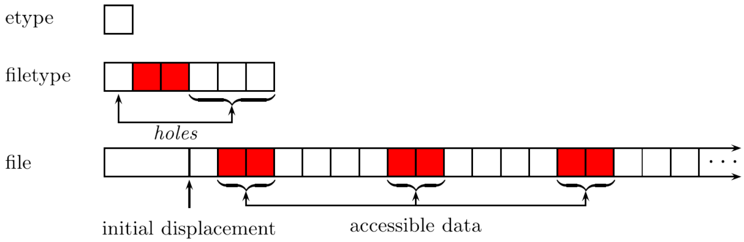


Figure 59 – Tiling a file with a filetype

MPI-IO Views

The View Mechanism

- File Views are defined using MPI derived datatypes.
- You can define holes (gaps) in a view so that certain parts of the data are ignored.
- The default view is a linear byte stream (displacement is zero, etype and filetype equal to `MPI_BYTE`).

Multiple Views

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

MPI-IO Views

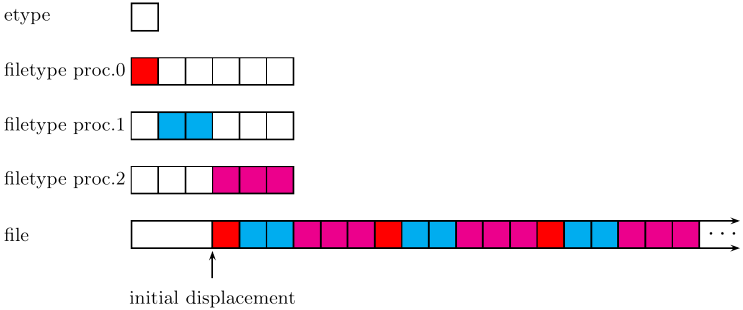


Figure 60 – Separate views, each using a different filetype, can be used to access the file

Limitations :

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

MPI-IO Views

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

```
mpi4py.MPI.File.Set_view(disp=0, etype=BYTE, filetype=None, datarep='native',
                         info=INFO_NULL)
```

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

Notes :

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

MPI-IO Views / Derived Datatypes

Subarray datatype constructor

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

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

MPI-IO Views / Derived Datatypes

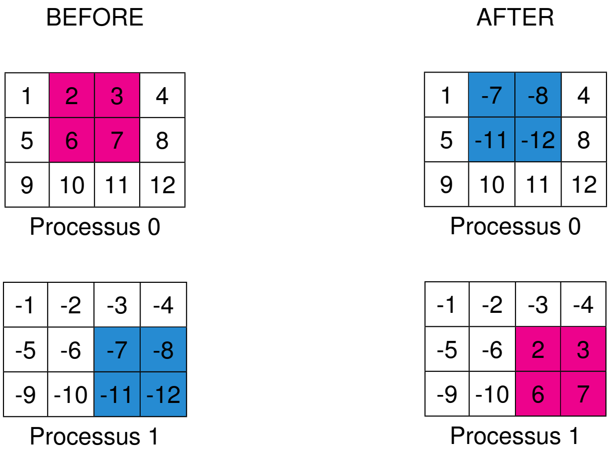
```
# Return a type
mpi4py.MPI.Datatype.Create_subarray(sizes, subsizes, starts, order=ORDER_C)
```

Explanation of the arguments

- sizes : shape of the array from which a subarray will be extracted
- subsizes : shape of the subarray
- starts : 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 `starts(:)=(/ 1,2 /)`
- order : storage order of elements
  - `MPI_ORDER_FORTRAN` for the ordering used by Fortran arrays (column-major order)
  - `MPI_ORDER_C` for the ordering used by C arrays (row-major order)

MPI-IO Views / Derived Datatypes

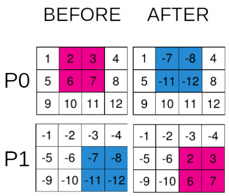
Exchanges between 2 process with subarray



MPI-IO Views / Derived Datatypes

Exchanges between the two processes (Part 1/2)

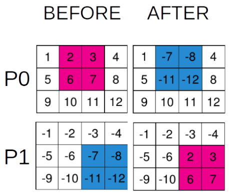
```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_lines = 3
nb_columns = 4
sign = 1
tag = 1000

# Initialisation of array tab on every processes
if rank == 1:
    sign = -1
tab = np.zeros((nb_lines, nb_columns), dtype=np.int32)
for i in range(nb_lines):
    for j in range(nb_columns):
        tab[i, j] = sign * (1 + i * nb_columns + j)
```



MPI-IO Views / Derived Datatypes

Exchanges between the two processes (Part 2/2)

```
# Shape of big array
profil_tab = [nb_lines, nb_columns]
# Shape of little array
profil_sub_tab = [2, 2]
# Coordinate of little array start
coord_start = [rank, rank + 1]
# Creation of type_sub_tab
type_sub_tab = MPI.INT.Create_subarray(profil_tab, profil_sub_tab,
                                       coord_start)
type_sub_tab.Commit()
# Permutation of sub array
comm.Sendrecv_replace([tab, 1, type_sub_tab], (rank + 1) % 2, tag,
                      (rank + 1) % 2, tag)
type_sub_tab.Free()
```



MPI-IO Views

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

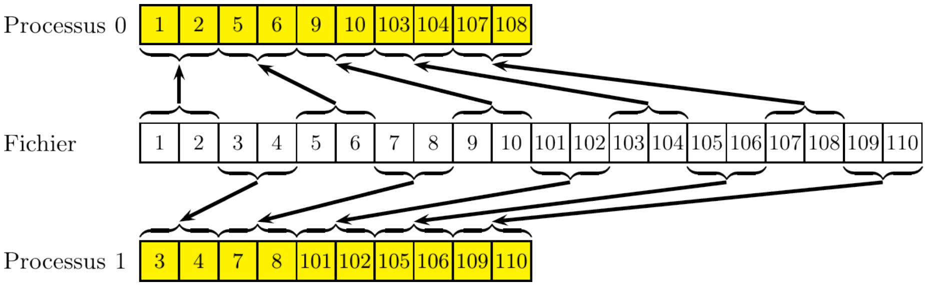


Figure 61 – Example 1 : Reading non-overlapping sequences of data segments in parallel

```
> mpiexec -n 2 python -m mpi4py read_view01.py

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

MPI-IO Views

Example 1

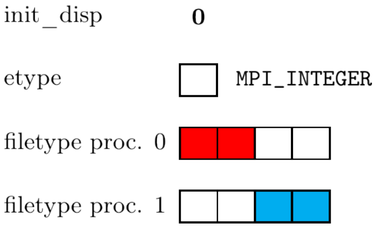
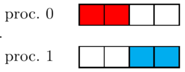


Figure 62 – Example 1 (continued)

```
if rank == 0:
  coord = 0
elif rank == 1:
  coord = 2
motif = MPI.INT.Create_subarray([4],[2],[coord])
motif.Commit()
fh.Set_view(0,MPI.INT,motif)
```

MPI-IO Views

Example 1 : code



```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
  coord = 0
elif rank == 1:
  coord = 2
motif = MPI.INT.Create_subarray([4], [2], [coord])
motif.Commit()

fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
fh.Set_view(0, MPI.INT, motif)
values = np.empty(10, dtype=np.int32)
fh.Read(values)
print(f"process {rank} : {values}")
fh.Close()
```

MPI-IO Views

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

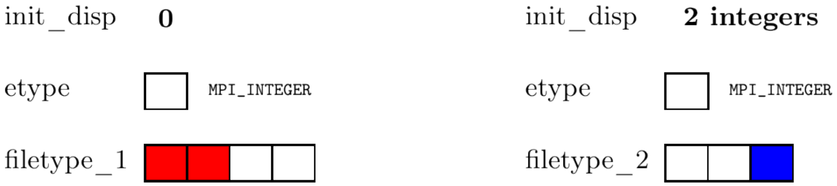
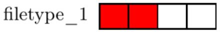
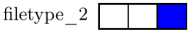


Figure 63 – Example 2 : Reading data using successive views

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
values = np.empty(10, dtype=np.int32)
```

MPI-IO Views



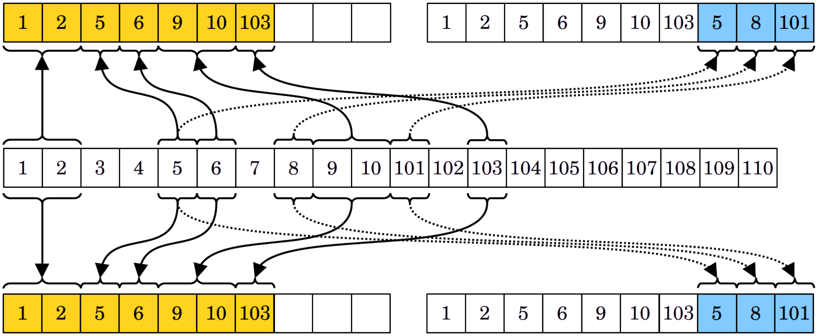


Example 2 (Part 2/2)

```
motif_1 = MPI.INT.Create_subarray([4], [2], [0])
motif_1.Commit()
motif_2 = MPI.INT.Create_subarray([3], [1], [2])
motif_2.Commit()

fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
fh.Set_view(0, MPI.INT, motif_1)
fh.Read([values, 3, MPI.INT])
fh.Read([values[3:], 4, MPI.INT])
nb_bytes_integer = MPI.INT.Get_size()
fh.Set_view(2*nb_bytes_integer, MPI.INT, motif_2)
fh.Read([values[7:], 3, MPI.INT])
print(f"Lecture processus {rank} : {values}")
fh.Close()
```

MPI-IO Views

Example 2 : Illustration



```
> mpiexec -n 2 python -m mpi4py read_view02.py

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

MPI-IO Views

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

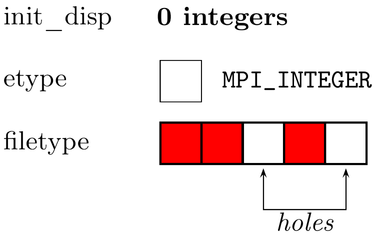
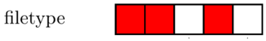


Figure 64 – Example 3 : Dealing with holes in datatypes

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

values = np.empty(9, dtype=np.int32)
```

MPI-IO Views

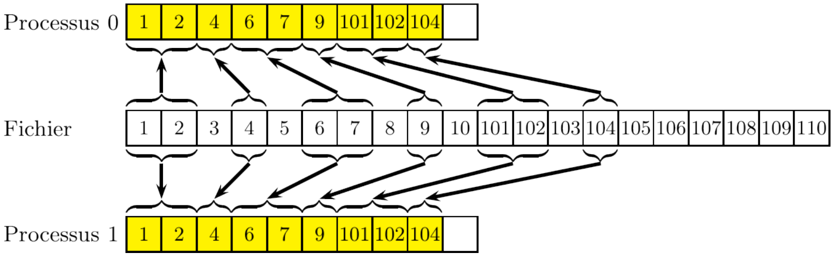


Example 3 (Part 2/2)

```
motif = MPI.INT.Create_indexed([2, 1], [0, 3])
nb_bytes_integer = MPI.INT.Get_size()
_, extent = motif.Get_extent()
motif = motif.Create_resized(0, extent + nb_bytes_integer)
motif.Commit()

fh = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
fh.Set_view(0, MPI.INT, motif)
fh.Read(values)
print(f"process {rank} : {values}")
fh.Close()
```

MPI-IO Views

Example 3 : Illustration



```
> mpiexec -n 2 python -m mpi4py read_view03.py

Lecture processus 0 : [  1   2   4   6   7   9 101 102 104]
Lecture processus 1 : [  1   2   4   6   7   9 101 102 104]
```

MPI-IO Views

Conclusion about MPI-IO and views

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

Advice

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

Conclusion

Conclusion

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

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

Resolution of the following Poisson equation :

$$\begin{cases} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} & = f(x,y) \quad \text{in } [0,1]\text{x}[0,1] \\ u(x,y) & = 0. \quad \text{on the boundaries} \\ f(x,y) & = 2.\left(x^2 - x + y^2 - y\right) \end{cases}$$

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

We will solve this equation with a domain decomposition method :

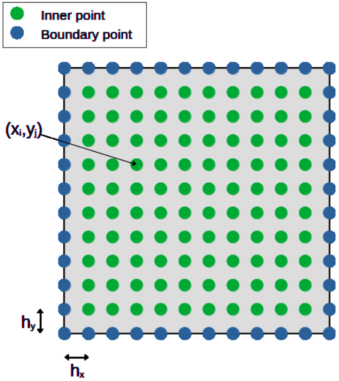
- The equation is discretized on the domain with a finite difference method.
- The obtained system is resolved with a Jacobi solver.
- The global domain is split into sub-domains. Each process solves the equation on one of the sub-domains and stores the result in a file.

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

The study domain is discretized according to a regular grid consisting of a set of points of coordinates (x_i, y_j) on which the solution will be approximated. We define :

- ntx the number of inner points following x
- nty the number of inner points following y

Note : In total there are $(ntx + 2) \times (nty + 2)$ points with the boundary points.



- $h_x = \frac{1}{ntx+1}$ the x-wise step
- $h_y = \frac{1}{nty+1}$ the y-wise step
- $x_i = ih_x$ for $i \in \{0, ..., ntx + 1\}$ the points coordinates following x
- $y_j = jh_y$ for $j \in \{0, ..., nty + 1\}$ the points coordinates following y
- $u_{i,j} = u(x_i, y_j)$ and $f_{i,j} = f(x_i, y_j)$ for $(i, j) \in \{0, ..., ntx + 1\} \times \{0, ..., nty + 1\}$

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

Using the finite difference method, the partial derivates $\frac{\partial^2 u}{\partial x^2}$ et $\frac{\partial^2 u}{\partial y^2}$ at a point (x_i, y_j) can be approximated as a function of the values of u in a close neighborhood (i.e., for small h_x and h_y) :

$$\underbrace{\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}}_{f_{i,j}} \simeq \frac{\frac{u_{i+1,j} - u_{i,j}}{h_x} - \frac{u_{i,j} - u_{i-1,j}}{h_x}}{h_x} + \frac{\frac{u_{i,j+1} - u_{i,j}}{h_y} - \frac{u_{i,j} - u_{i,j-1}}{h_y}}{h_y}.$$

Some algebra yields

$$u_{i,j} \simeq \frac{h_x^2 h_y^2}{2(h_x^2 + h_y^2)} \left[\frac{1}{h_x^2} \left(u_{i+1,j} + u_{i-1,j} \right) + \frac{1}{h_y^2} \left(u_{i,j+1} + u_{i,j-1} \right) - f_{i,j} \right].$$

The Jacobi method is an iterative method which amounts to perform, at each iteration,

$$u_{i,j}^{n+1} = \frac{h_x^2 h_y^2}{2(h_x^2 + h_y^2)} \left[\frac{1}{h_x^2} \left(u_{i+1,j}^n + u_{i-1,j}^n \right) + \frac{1}{h_y^2} \left(u_{i,j+1}^n + u_{i,j-1}^n \right) - f_{i,j} \right].$$

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

- In parallel, each process solves the equation on one of the sub-domains.
- The interface values of subdomains must be exchanged between the neighbouring processes.
- We use ghost cells ; these cells serve as a reception buffer for exchanges between neighbors.

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

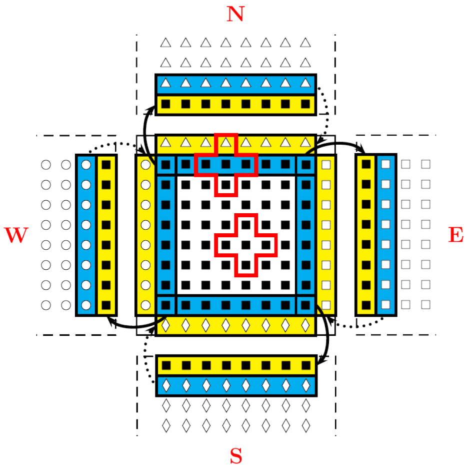


Figure 65 – Exchange points on the interfaces

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

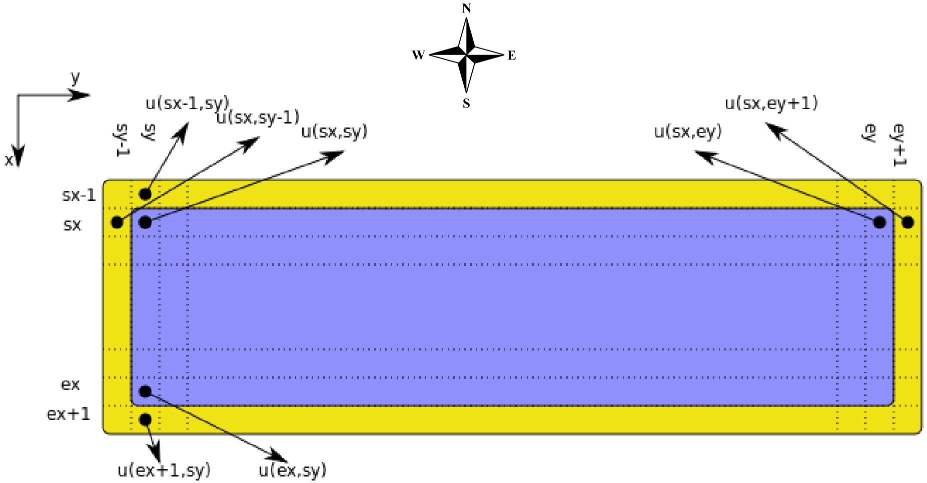


Figure 66 – Numeration of points in different sub-domains

MPI Hands-On – Exercise 8 : Poisson’s equation

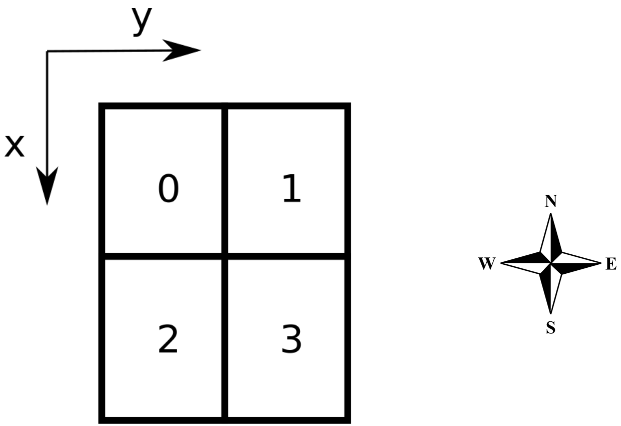


Figure 67 – Process rank numbering in the sub-domains

MPI Hands-On – Exercise 8 : Poisson’s equation

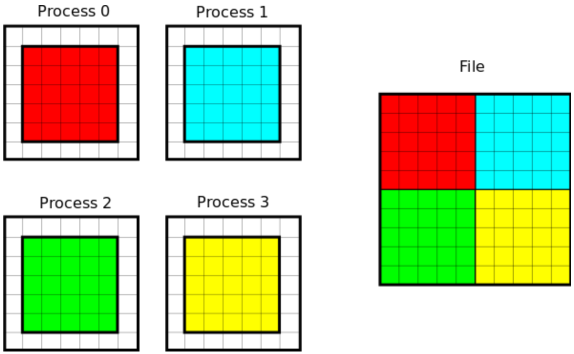


Figure 68 – Writing the global matrix u in a file

The processes write the global solution in a file. You need to :

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

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

- A skeleton of the parallel version is proposed : It consists of a main program (`poisson.py`) and several subroutines. The following steps are to be implemented **in the `parallel.py` file**.
  - Initialisation of the MPI environment.
  - Creation of the 2D Cartesian topology
  - Determination of the array indexes for each sub-domain.
  - Determination of the 4 neighbour processes for each sub-domain.
  - Creation of two derived datatypes, *type_line* and *type_column*.
  - Exchange the values on the interfaces with the other sub-domains.
  - Computation of the global error. When the global error is lower than a specified value (machine precision for example), we consider that we have reached the exact solution.
  - Collecting of the global matrix u (the same one as we obtained in the sequential) in an MPI-IO file `data.dat`.

- To compile use make, to execute use make exe. To verify the results, use make verification which runs a reading program of the `data.dat` file and compares it with the sequential version.