MPI parallel programming paradigm

strongly inspired from IDRIS course with python implementation point of view

HPC 4GMM 2021/2022

F. Couderc[1] and L. Giraud[2]

[1]CNRS, Institut de Mathématiques de Toulouse.
[2]INRIA Bordeaux Sud-Ouest.

Outline

1- Introduction

1.1- What is MPI ?

1.2- Programming Model

2- Environnement

2.1- MPI Initialization and Finalization

2.2- Communicators

2.3- Rank and number of processes

2.4- Exercice

Outline

3- Point-to-point communications

- 3.1- General concepts
- 3.2- Some data types
- 3.3- Example from mpi4py documentation
- 3.4- Exercice
- 3.5- Other possibilities
- 3.6- Communication ring
- 3.7- Exercice

Outline

4- Collective communications

- 4.1- General concepts
- 4.2- Global synchronization
- 4.3- Global broadcast
- 4.4- Exercice
- 4.5- Selective broadcasts
- 4.6- Exercice
- 4.7- Exercice
- 4.8- Global reductions
- 4.9- Exercice
- 4.10- Exercice

Outline

5- Optimizations

- 5.1- Introduction
- 5.2- Point-to-point send/recv modes

6- Process topologies

- 6.1- Introduction
- 6.2- Cartesian topology
- 6.3- Processes graph

1- Introduction

1.1- What is MPI ?

Main Idea

A parallel paradigm is primarily driven by the memory view: with MPI, all the **processes** have a disjoint memory space and share information through explicit message exchange (data is moved from the address space of one process to that of another process through cooperative operations on each process.).

MPI (abbrevation for Message Passing Interface) is:

- a specification for the developers and users of **message passing libraries** (OpenMPI, MPICH, Intel MPI, etc ...) and has resulted from the efforts of numerous individuals and groups that began in 1992 (The MPI-4.0 standard is under development http://www.mpi-forum.org/).
- Simply stated, the goal is to provide a widely used standard for writing message passing programs. The interface attempts to be:
  - Practical (relatively easy to use),
  - Portable (using the same program from a machine to another),
  - Efficient (message exchanges between processes optimizated to be efficient at execution),
  - Flexible (support different data types in messages for example).

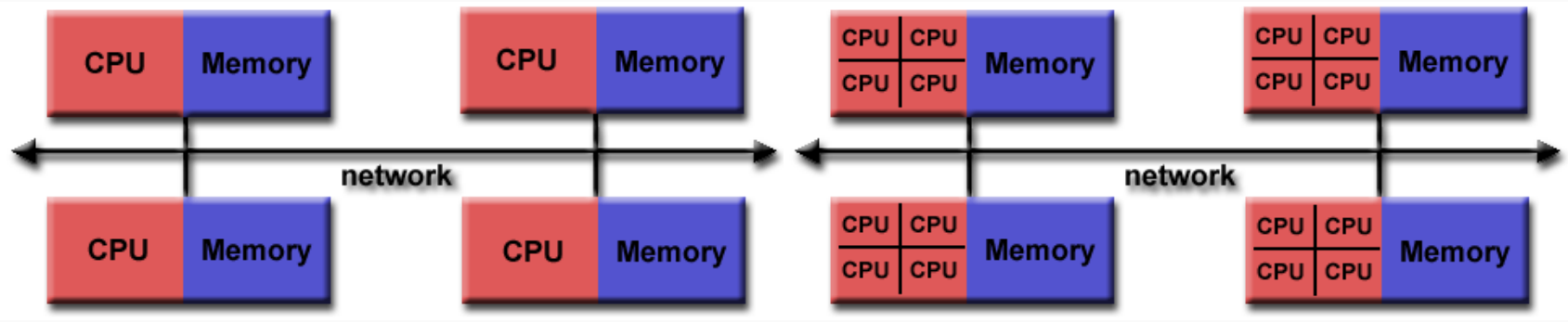


Figure 1: (left) distributed model; (right) hybrid shared/distributed model.

- Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).
- As architecture trends changed, shared memory were combined over networks. MPI implementors adapted their libraries to handle both types. Today, MPI runs on virtually any hardware platform:
    - distributed Memory,
    - shared Memory,
    - hybrid.
- The programming model **clearly remains a distributed memory model** however, regardless of the underlying physical architecture of the machine.
- All parallelism is explicit: **the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs**.

2- Environnement

2.1- MPI Initialization and Finalization

1. Any C/C++ or Fortran program unit calling MPI routines must include a header file (**#include <mpi.h>** in C and **use mpi** in Fortran).
2. The MPI routine MPI_Init allows to initialize the necessary environment.
3. Conversely, the MPI routine MPI_Finalize deactivates this environment.

C

```c
#include <mpi.h>
...
int MPI_Init(int *argc, char ***argv);
...
int MPI_Finalize(void);
```

The python **mpi4py** package is an object-oriented wrapping to the MPI standard and makes initialization automatic at import operation.

Python

```python
from mpi4py import MPI
```

2.2- Communicators

- MPI uses objects called **communicators** to define which collection of processes may communicate with each other.
- Most MPI routines require you to specify a communicator as an argument.
- The default communicator is `MPI_Comm_World` which includes all active processes (invoked from the execution command line **`mpirun -np x ./...`**).

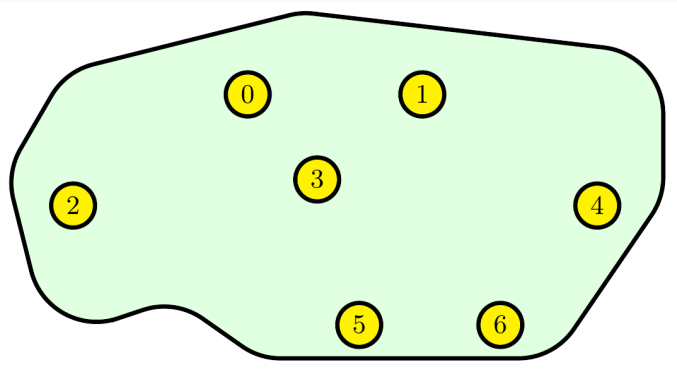


Figure 2: Communicator `MPI_Comm_World` (with 7 processes invoked at execution command line).

Remark

It is possible for the programmer to define as many process subgroups as desired, especially for running several models at once with regular rendezvous points, but this feature will not be covered in this course.

2.3- Rank and number of processes

- At any time, the **number of processes** managed by a communicator can be known by the MPI routine `MPI_Comm_Size`.
- In the same way, the MPI routine `MPI_Comm_Rank` aims to obtain the **rank of a process**, a number between 0 and the value returned by `MPI_Comm_Size` -1.

C

```c
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Python

```python
comm = MPI.COMM_WORLD
size = comm.size
rank = comm.rank
```

2.4- Exercice

Call the Anaconda environment necessary to import the mpi4py Python package:

bash

```
source activate HPC
```

Watch, run and play with the **hello-world.py** program by typing the command:

Python

```
mpirun -np 4 --oversubscribe python hello-world.py
```

3- Point-to-point communications

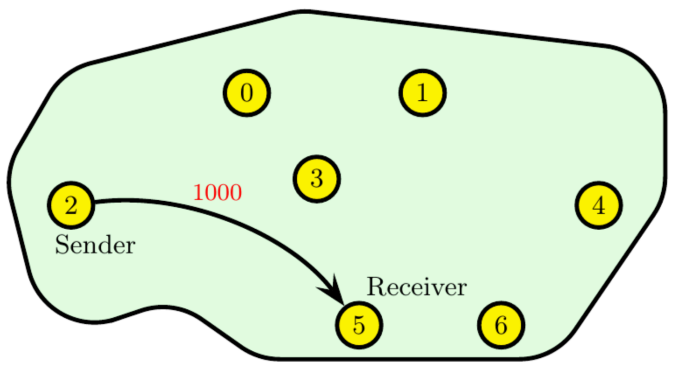


Figure 3: Point-to-point communication within the communicator `MPI_Comm_World`.

basic idea

A so-called point-to-point communication takes place between two processes, one called the sending process and the other the receiving process.

3.1- General concepts

- The sender and receiver are identified by their **rank** in the communicator.
- The object communicated from one process to another is called **message**.
- The so-called **envelope** of a message is composed of:
  - the communicator in which the message transfer occurs.
  - the rank of the sending process in the communicator,
  - the rank of the receiving process in the communicator,
  - the tag of the message, an identification number in the communicator used.
- The exchanged **data are typed** (integers, reals, etc or personal derived types).
- In each case, there are several **transfer modes**, using different protocols (see chapter 5).

Extract from the OpenMpi v4.1 manual: https://www.open-mpi.org/doc/v4.1/

C

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

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
     MPI_Comm comm, MPI_Status *status)
```

- **buf**: initial address of send buffer (choice).
- **count**: number of elements send (nonnegative integer).
- **datatype**: datatype of each send buffer element (handle).
- **dest**: rank of destination (integer).
- **tag**: message tag (integer).
- **comm**: Communicator (handle).

3.1- Python routines

The **mpi4py** package documentation can be found here: https://mpi4py.readthedocs.io.

Python

```
       comm.send(buf, dest  =0, tag=0, status=None)
data = comm.recv(     source=0, tag=0, status=None)
```

- **buf**: data of any type to be emitted (thanks to the object-oriented wrapping).
- **data**: variable receiving data of the same type as buf.
- **dest**: rank of the recipient process.
- **source**: rank of the emitting process.
- **tag**: message label.
- **status**: status of the request at the output of the subroutine call.
- **comm**: communicator (such as `MPI_COMM_WORLD`).

Which contains also an optimized version for **numpy** package arrays,

Python (numpy)

```
comm.Send(sendbuf, dest  =0, tag=0, status=None)
comm.Recv(recvbuf, source=0, tag=0, status=None)
```

- **sendbuf**: numpy array.
- **recvbuf**: numpy array.

3.2- Some data types

MPI Type	C Type	Numpy Python Type (`dtype` argument)
MPI_Int	`int` (32 bits)	`np.int32` (`MPI.INT32`)
MPI_Long	`long int` (64 bits)	`np.int64` (`MPI.INT64`)
MPI_Real4	`float` (32 bits)	`np.float32` (`MPI.REAL4`)
MPI_Real8	`double` (64 bits)	`np.float64` (`MPI.REAL8`)
MPI_Real16	`long double` (128 bits)	`np.float128` (`MPI.REAL16`)
etc ...	etc ...	etc ...

Comments

- The other possibilities of datatype can be easily found in courses (IDRIS) or technical documents (MPI 3.1 manual).
- The above basic datatypes can be assembled as desired in derived datatype, with the use of MPI routines to construct them.
- While the definition of datatype to be passed in the MPI messages is extremely clear in C/C++/Fortran, porting the equivalent to the **`mpi4py`** package is not at all obvious and relatively poorly documented.

3.3- Example from mpi4py documentation

Python

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

comm = MPI.COMM_WORLD
rank = comm.rank

# passing MPI datatypes explicitly
if rank == 0:
  data = np.arange(1000, dtype=np.int32)
  comm.Send([data, MPI.INT], dest=1, tag=1)
elif rank == 1:
  data = np.empty(1000, dtype=np.int32)
  comm.Recv([data, MPI.INT], source=0, tag=1)

# automatic MPI datatype discovery
if rank == 0:
  data = np.arange(100, dtype=np.float64)
  comm.Send(data, dest=1, tag=2)
elif rank == 1:
  data = np.empty(100, dtype=np.float64)
  comm.Recv(data, source=0, tag=2)
```

Python program implementing a point to point communication where the rank 0 sends to rank 1 numpy arrays, with passing the MPI datatype or not.

3.4- Exercice

Watch, run and play with the program **`ping_pong.py`** by typing the command:

Python

```
mpirun -np 2 python ping_pong.py
```

3.5- Other possibilities

- When a message is received, the process rank and the tag can be wildcards (joker) respectively by `MPI_Any_Source` and `MPI_Any_Tag`.
- A communication with the "dummy" process of rank `MPI_Proc_Null` has no effect. (`MPI.PROC_NULL` for **mpi4py** if imported as MPI).
- `MPI_Graph_Create` is a predefined constant, which can be used instead of the status variable.
- There are variants, `MPI_SendRecv` and `MPI_SendRecv_Replace`, which sequence a send and a receive simultaneously.
- More complex data structures can be created using subroutines such as `MPI_Type_Contiguous`, `MPI_Type_Vector`, `MPI_Type_Indexed` or `MPI_Type_Struct`.

3.5- Send/Recv in the same time

Simultaneous send and receive operation MPI_SendRecv :

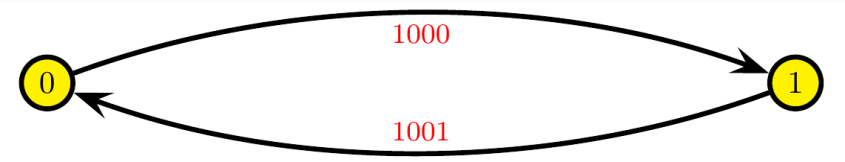


Figure 4: **sendrecv** communication between processes 0 and 1.

Python

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD

size = comm.size
rank = comm.rank

num_proc = (rank+1)%2

valeur = comm.sendrecv(rank+1000,dest=num_proc)
```

3.5- Send/Recv in the same time: deadlock situation

Python

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD

size = comm.size
rank = comm.rank

         comm.send(rank+1000,dest  =(rank+1)%size)
valeur = comm.recv(rank+1000,source=(rank-1)%size)
```

Warning !

If the subroutine `MPI_Send` is implemented synchronously (see section 5.2) in the version of the MPI library implemented, the above code would be in a deadlock situation if instead of `MPI_SendRecv` a `MPI_Send` followed by a `MPI_Recv` was used.

Indeed, each of the two processes would never come, since the two sendings would remain in suspense. It is therefore absolutely necessary to avoid these cases.

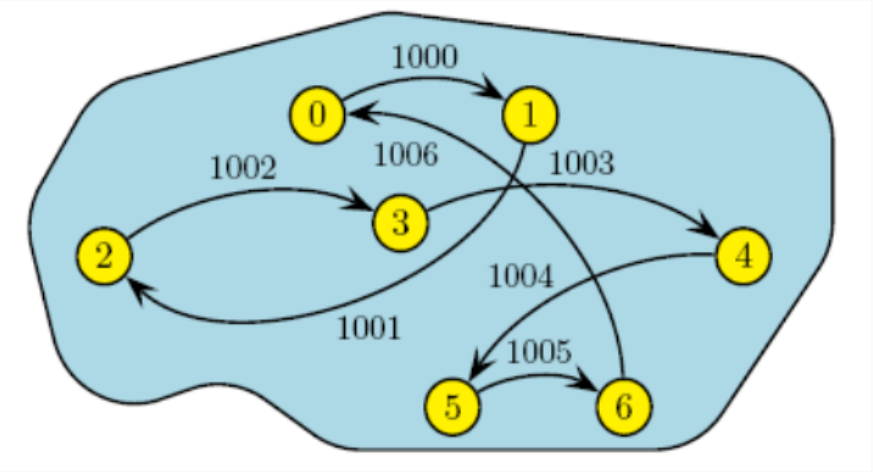
3.5- Exercice

Watch, run and play with the program **`ping_pong_deadlock.py`** by typing the command:

Python

```
mpirun -np 2 python ping_pong_deadlock.py
```

3.6- Communication ring



Python

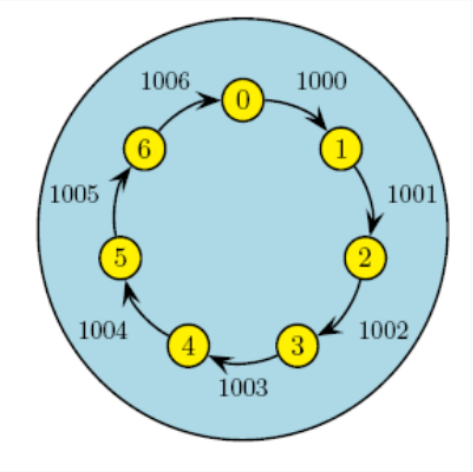
```
from mpi4py import MPI

comm = MPI.COMM_WORL
rank = comm.rank

          comm.send(1000+rank, dest   = (rank+1)%size )
valeur = comm.recv(            source = (rank-1)%size )
```

- If all the processes send and receive, all the communications can potentially start simultaneously and will therefore not take place in a ring, i.e. starting from process of rank 0 sending to process of rank 1,
- in addition to the above mentioned problem of portability, in case the implementation of the MPI_Send is done synchronously in the version of the MPI library implemented.

3.6- Communication ring



In order for communications to be truly ring-based, like token passing between processes, it is necessary to proceed differently and have one process initiate the chain:

Python

```
...
if rank == 0:
   comm.send(1000+rank, dest=1)
   valeur = comm.recv(source=size-1)
else:
   valeur = comm.recv(source=rank-1)
   comm.send(1000+rank, dest=(rank+1)%size)
```

3.7- Exercice

Watch, run and play with the **`ring.py`** program by typing the command:

Python

```
mpirun --oversubscribe -np 7 python ring.py
```

4- Collective communications

4.1- General concepts

- **Collective** communications allow **a series of point-to-point communications** to be made in a single operation (data broadcasting from one/all/all processes to all/one/all processes with eventualy some reduction operations like sum, product, min, max, etc ...).
- A collective communication always involves **all the processes of the specified communicator**.
- For each of the processes, the **call ends when its participation in the collective operation is completed** (not necessary to add a global synchronisation (barrier) after a collective operation).
- The management of **tags** in these communications is transparent and at the charge of the system, and so not provided at call.

There are three types of routines:

- the one that provides global synchronizations: `MPI_Barrier`,
- those which only diffuse data:
  - global data broadcast: `MPI_Bcast`;
  - selective data broadcast: `MPI_Scatter`;
  - distributed data collection: `MPI_Gather`;
  - distributed data collection with global broadcast: `MPI_AllGather`;
  - collection and selective distribution by all the processes of distributed data: `MPI_AllToAll`.
- those who, in addition to transfer data, perform operations on:
  - reduction operations, whether of a predefined type (sum, product, maximum, minimum, etc.) or of a personal type: `MPI_Reduce`;
  - reduction operations with broadcast of the result : `MPI_AllReduce` (this is in fact a `MPI_Reduce` followed by a `MPI_Bcast`).

4.2- Global synchronization: MPI_Barrier

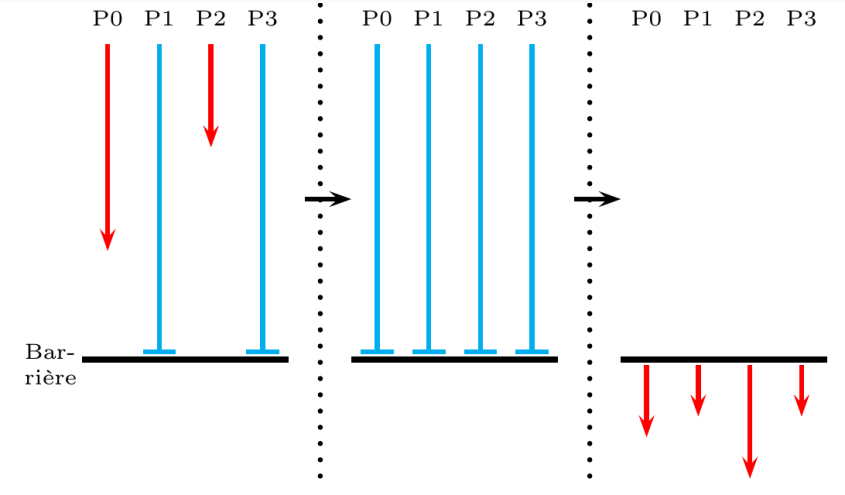


Figure 5: When the command MPI_Barrier is called, all processes wait for all others before continuing to run the program.

Python

```
comm = MPI.COMM_WORLD
rank = comm.rank

for k in range(size):
   comm.Barrier()
   if rank == k:
      ...
```

4.3- Global broadcast: MPI_Bcast

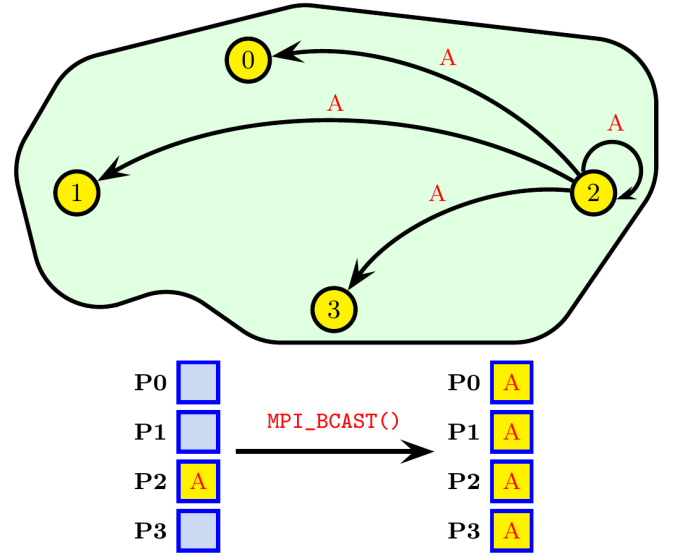


Figure 6: Communication of a process-specific data structure(s) to all other processes in a communicator.

Python

```
comm = MPI.COMM_WORLD
rank = comm.rank

if rank == 2:
   valeur = 1000+rank
   comm.bcast(data, root=2)
else:
   valeur = comm.bcast(None, root=2)
```

4.4- Exercice

Watch, run and play with the program **`bcast.py`** by typing the command:

Python

```
mpirun -np 4 --oversubscribe python bcast.py
```

4.5- Selective broadcast: MPI_Scatter

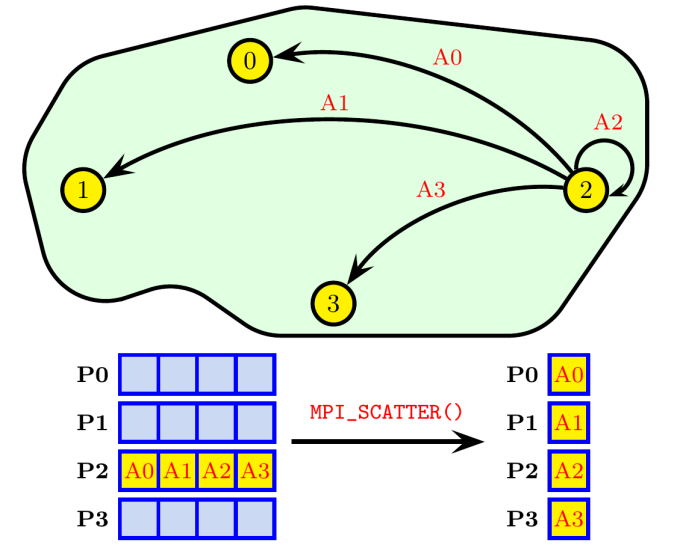


Figure 7: Communication of the components of a vector of data (structures) of the size of the number of processes and carried by a single process to all other processes in the communicator.

Python

```
comm = MPI.COMM_WORLD
size = comm.size
rank = comm.rank
data = None

if rank == 2:
   data = [(x+1)**x for x in range(size)]

data = comm.scatter(data, root=2)
```

4.6- Exercice

Watch, run and play with the **scatter.py** program by typing the command:

Python

```
mpirun -np 4 --oversubscribe python scatter.py
```

4.6- Selective broadcast: MPI_Gather

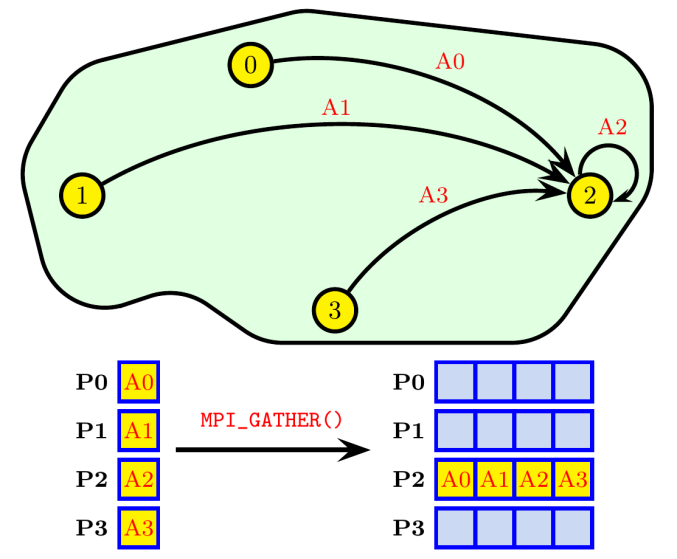


Figure 8: Communication of all (data structures) specific to each process of a communicator to a single process of the communicator in the form of a vector.

Python

```
comm = MPI.COMM_WORL
rank = comm.rank

data = (rank+1)**rank
data = comm.gather(data, root=2)
```

4.7- Exercice

Watch, run and play with the **gather.py** program by typing the command:

Python

```
mpirun -np 4 --oversubscribe python gather.py
```

- A reduction is an operation applied to a set of elements to obtain a single value. Typical examples are summing the elements of a vector `SUM(V(:))` or finding the maximum value element in a vector `MAX(V(:))`.
- MPI provides high-level routines for performing reductions on data spread over a set of processes, with recovery of the result on a single process (`MPI_Reduce`) or on all of them (`MPI_AllReduce` , which is actually just a `MPI_Reduce` followed by a `MPI_Bcast`).
- If several elements are concerned by process, the reduction function is applied to each of them.

4.8- Operators

Nom	Opération	Python operator (if imported as `MPI`)
MPI_Sum	Sum of the elements	`MPI.SUM`
MPI_Prod	Product of the elements	`MPI.PROD`
MPI_Max	Maximum of the elements	`MPI.MAX`
MPI_Min	Minimum of the elements	`MPI.MIN`
MPI_MaxLoc	Index for the maximum element	`MPI.MAXLOC`
MPI_MinLoc	Index for the minimum element	`MPI.MINLOC`
etc ...	etc ...	etc ...

4.8- Selective reduction: MPI_Reduce

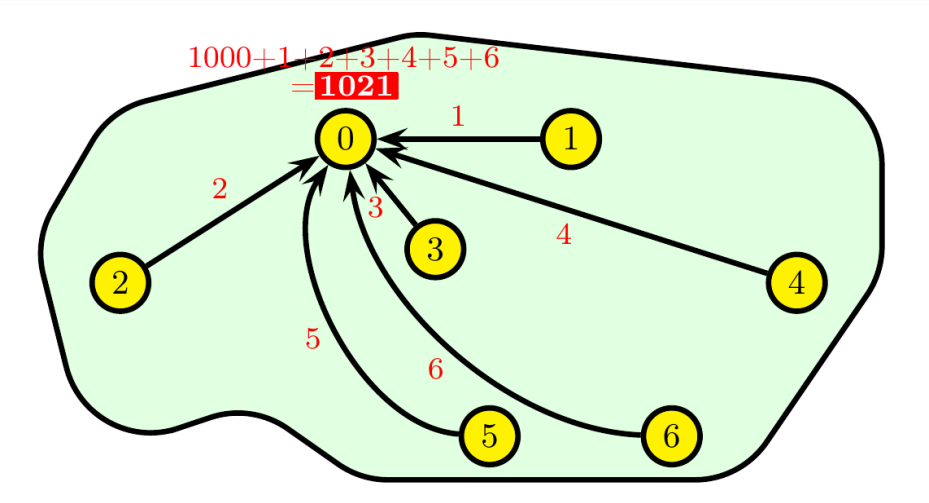


Figure 9: Process 0 receives the sum of the values carried by the other processes (equal to the rank) of the communicator and adds it to its own value 1000.

Python

```
comm = MPI.COMM_WORLD

rank = comm.rank

if rank == 0:
   valeur = 1000
else:
   valeur = rank

valeur = comm.reduce(sendobj=valeur, op=MPI.SUM, root=0)
```

4.9- Exercice

Watch, run and play with the **`reduce.py`** program by typing the command:

Python

```
mpirun -np 7 --oversubscribe python reduce.py
```

4.9- Distributed reduction: MPI_AllReduce

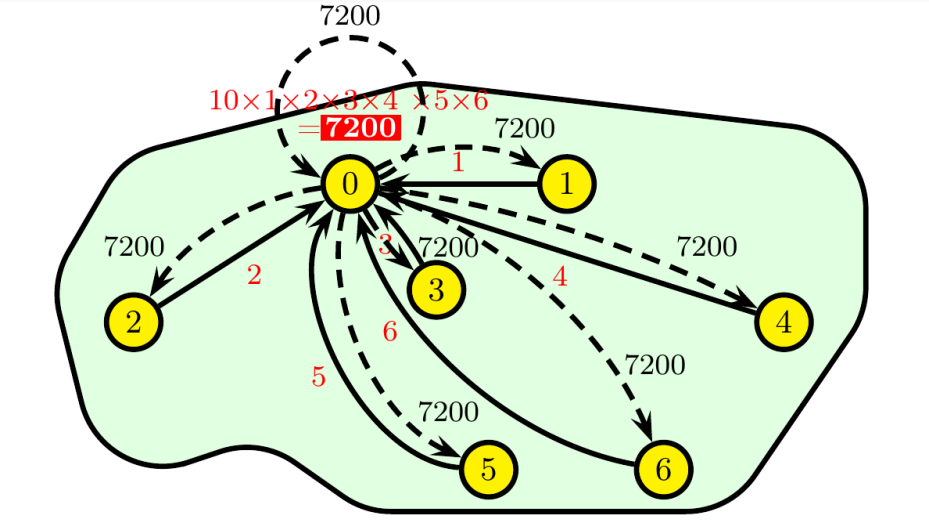


Figure 10: Process 0 receives the product of the values carried by the other processes (equal to the rank) of a communicator, multiplies it to its own value 10 and returns the result to the set.

Python

```
comm = MPI.COMM_WORLD

rank = comm.rank

if rank == 0:
   valeur = 10
else:
   valeur = rank

valeur = comm.allreduce(sendobj=valeur, op=MPI.PROD)
```

Watch, run and play with the program **`allreduce.py`** by typing the command:

Python

```
mpirun -np 7 --oversubscribe python allreduce.py
```

5- Optimizations

5.1- Introduction

Goal

Optimisation must be a key concern when the proportion of communications to calculations becomes quite large (and so calculations themself very often optimized ...).

Solutions

Communication optimizations can be achieved at different levels, the main ones being from the most important to the less one:

- rethinking the algoritms to be cheap regarding communications frequencies and message length.
- recover communications by computations,
- avoid, if possible, copying the message to a temporary memory space (buffering),
- minimize the extra costs incurred by repetitive calls to communication routines.

With MPI the sending of a message can be done in different modes:

- **standard**: is actually a "non-mode" that lets the MPI implementation choose which communication mode is preferable. This might be heavily dependent on the implementation.
- **synchroneous**: synchronises the sending and receiving processes. The sending of the message is completed if the reception is posted and the reading is completed. Generally avoids recopying the message.
- **buffered**: 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.
- **ready**: the sending of the message can only start if the reception has been posted before (this mode is interesting for client-server applications).

As an indication, here are the different cases envisaged by the MPI standard, knowing that the implementations can be different:

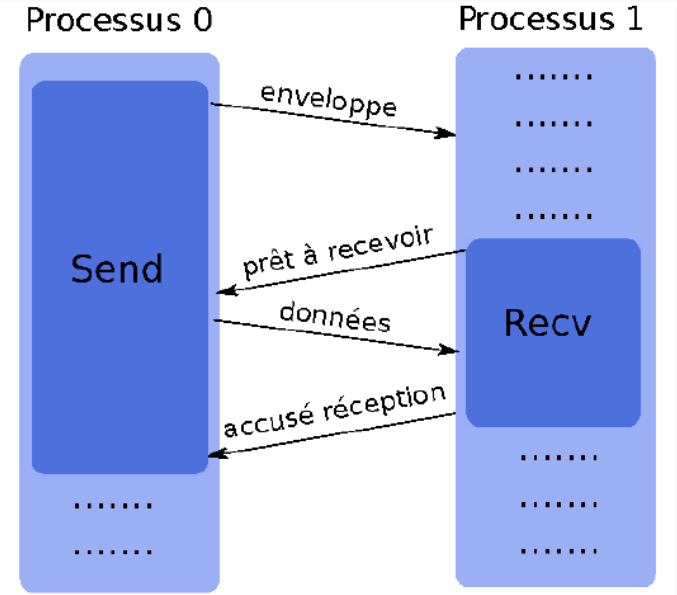
modes	blocking	non-blocking
standard sending	MPI_Send (*)	MPI_ISend
synchroneous sending	MPI_SSend	MPI_ISSend
buffered sending	MPI_BSend	MPI_IBSend
reception	MPI_Recv	MPI_IRecv

(*) Depending on the MPI implementation, a standard send can be blocking with temporary copy or synchronous with the receive, depending on the size of the message to be sent.

5.2- Synchroneous blocking point-to-point communications.

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.

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



- Avantages: low resource consumption (no buffering), fast if the receiver is ready (no copying into a buffer), knowledge of reception thanks to synchronisation.
- Drawbacks: waiting time if the receiver is not ready, risk of deadlock situation.

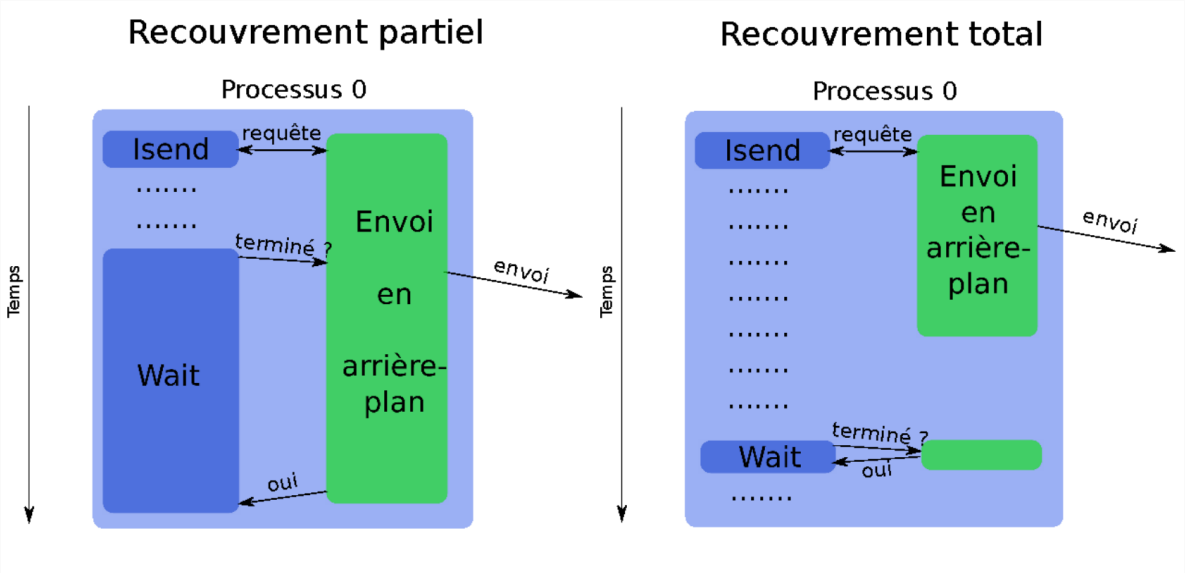


Figure 11: on the left the request MPI_Wait is called while the sending MPI_ISend is not completely finished contrary to the case on the right where the sending has been completely covered by other instructions in the program.

Definition

A non-blocking call returns the hand very quickly, but does not allow immediate reuse of the memory space used in the communication. It is necessary to make sure that the communication is finished (with MPI_Wait for example) before using it again.

Advantages

- Possibility to hide all or part of the communication costs.
- No chance of deadlock.
- Keeping the hand on synchronization to reuse the memory space communicated after breaking points.

Drawbacks

- Little more complicated to implement and maintain ... that's all ! must be preferred ...

6- Process topologies

6.1- Introduction

- In most applications, especially in domain partitioning methods where the computational domain is matched to the process grid, it is interesting to be able to arrange the processes in a regular topology.
- MPI allows the definition of virtual topologies of the Cartesian or more generally graph type:
    - Cartesian topologies:
        - each process is defined in a process grid,
        - the grid can be periodic or not,
        - processes are identified by their coordinates in the grid.
    - Graph topologies:
        - generalization to more complex topologies.

- A Cartesian topology is defined when a set of processes belonging to a given communicator call the MPI routine MPI_Cart_Create .
- We define:
  - an integer ndims (2 below) representing the number of grid dimensions,
  - an array of integer values **dims** of dimension ndims indicating the number of processes in each dimension,
  - an array of logical values **periods** of dimension ndims indicating the periodicity in each dimension,
  - a reorder logical option indicating whether the process numbering can be changed by MPI.

Python

```
comm = MPI.COMM_WORLD
dims = [nx, ny]
periods = [False] * len(dims)
topo = comm.Create_cart(dims, periods=periods, reorder=False)
coords = topo.Get_coords(topo.rank)
left, right = topo.Shift(0,1)
down, up    = topo.Shift(1,1)
```

6.3- Processes graph

- In certain applications (unstructured meshes for example), the domain decomposition is no longer a regular grid but a graph in which a sub-domain can have any one or more neighbours. The MPI routine MPI_Graph_Create then makes it possible to define a graph type topology by indicating the neighbours of each sub-domain.
- However, such a graph can perfectly be constructed by hand ...