Introduction

The goal of parallel programming is to :

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

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

Introduction: Sequential programming model

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

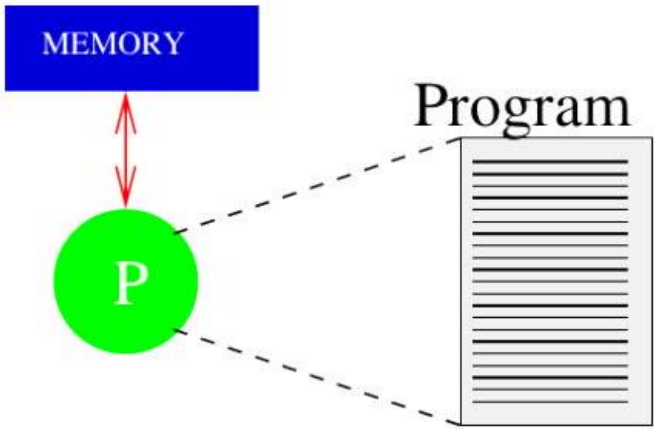

Figure: Sequential programming model

Message passing programming model

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

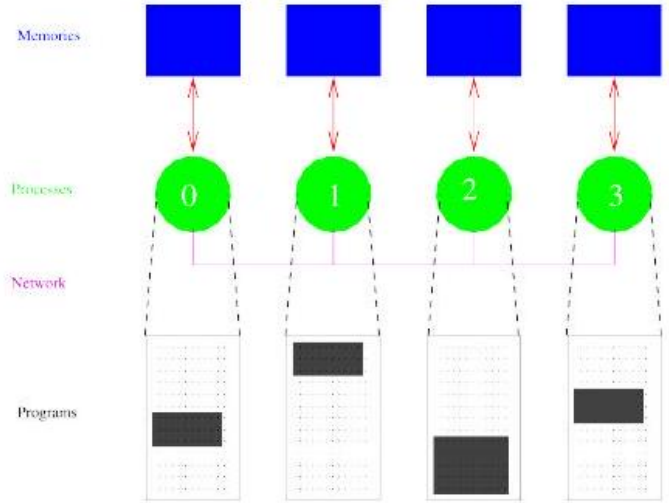

Figure 2 - Message Passing Programming Model

Message content

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

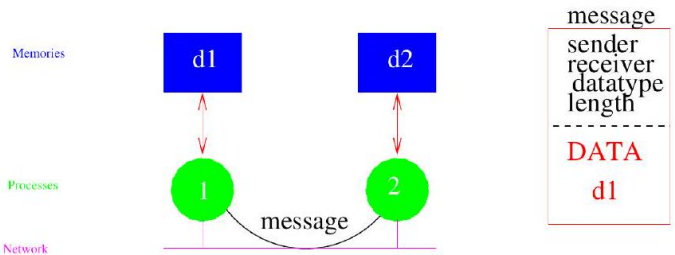

Figure 4 - Message Construction

Environment

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

Supercomputer architecture

Most supercomputers are distributed-memory computers. They are made up of many nodes and memory is shared within each node.
Noeud 0

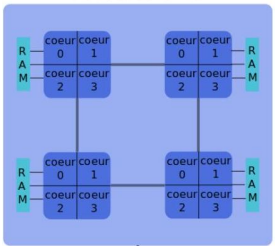

Figure 5 - Supercomputor architecture

Jean Zay

- 1116 nodes
- 2 Intel Cascade Lake processor (20 cores), 2,5 Ghz by node
- 4 GPU Nvidia V100 by node (on 391 nodes)
- 44640 cores
- 214 TB (192 GB by node)

MPI vs OpenMP

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

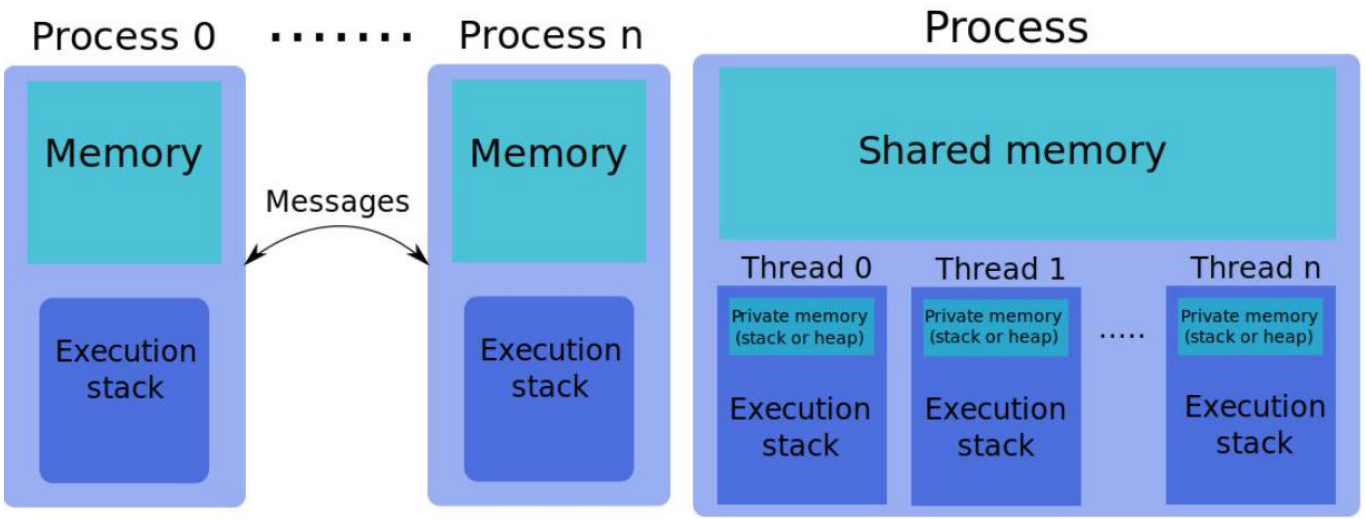

Figure: MPI scheme vs OpenMP scheme

Domain decomposition

A scheme 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.

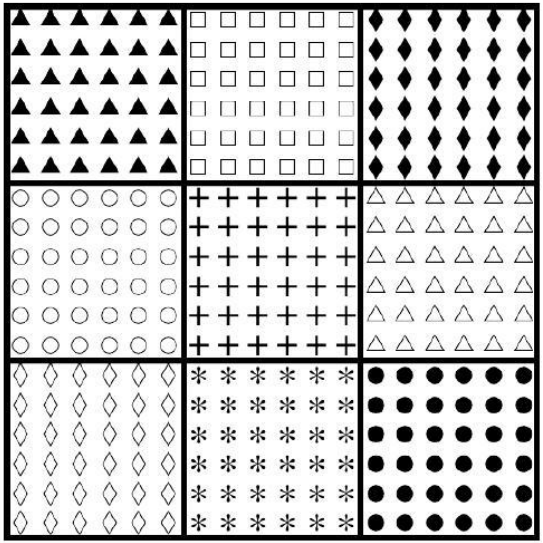

Figure: Decomposition in subdomains

Description

- Every program unit calling MPI subroutines has to include a header file. In Fortran, we use the `mpi\_f08` module introduced in MPI-3. Before in `MPI-2`, we used the `module mpi`, and in `MPI-1`, it was the `mpif.h` file).
- The `MPI\_Init()` subroutine initializes the MPI environment
- The `MPI\_Finalize()` subroutine disables the MPI environment.

```
1    program hello
2        use mpi_f08
3        implicit none
4        integer, optional, intent(out) :: ierr
5        call MPI_INIT(ierr)
6        print *, 'Bonjour, monde!'
7        call MPI\_Finalize(ierr)
8    end program hello
```

Differences between C/C++ and Fortran

In a C/C++ program :

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

```
1  int MPI_Init(int *argc, char ***argv);
```

Communicators

• All the MPI operations occur in a defined set of processes, called communicator. The default communicator is `MP I\_COMM\_WORLD`, which includes all the active processes.

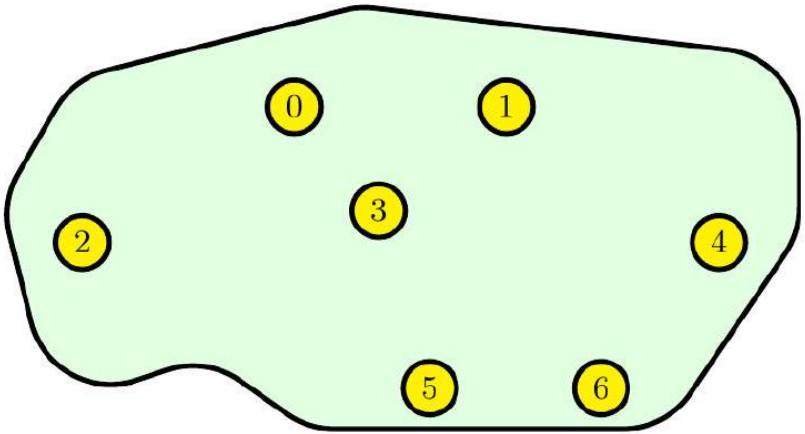

Figure: MPI_COMM_WORLD Communicator

Termination of a program

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

```
1    MPI_ABORT(comm, erreur, code)
2    TYPE (MPI_Comm), intent(in) :: comm
3    integer, intent(in) :: error
4    integer, optional, intent(out) :: code
```

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

Rank and size

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

```
1    MPI_COMM_SIZE (comm, nb_procs,code)
2    TYPE (MP I_Comm) :: comm
3    integer, intent(out) :: nb_procs
4    integer, optional, intent(out) :: code
```

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

```
1    MPI_COMM_RANK (comm,rank,code)
2    TYPE (MPI_Comm), intent(in) :: comm
3    integer, intent(out) :: rank
4    integer, optional, intent(out) :: code
```

Example

```
program who_am_I
    use
    implicit none
    integer :: nb_procs,rank
    call MPI_INIT()
    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
    call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
    print *,'I am the process ',rank,' among ',nb_procs
    call MPI_FINALIZE()
end program who_am_I
```

compile and run will give you:

```
$>$ mpiexec -n 7 who_am_I\\
I am process 3 among 7
I am process 0 among 7
I am process 4 among 7
I am process 1 among 7
I am process 5 among 7
I am process 2 among 7
I am process 6 among 7
```

Compilation and execution of an MPI code

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

```
1 mpif90 <options> -c source.f90\\
2 $>$ mpif90 source.o -o my\_executable\_file
```

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

Exercise 1 : MPI Environment

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

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

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

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

(use % symbol in C : a%b)

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

General Concepts

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

Point-to-point Communications

Blocking Send `MPI_Send`

```
1     MPI_SEND (buf, count, datatype, dest, tag, comm, code)
2     TYPE(*), dimension(..), intent(in) :: buf
3     integer, intent(in) :: count, dest, tag
4     TYPE (MPI_Datatype), intent(in) :: datatype
5     TYPE (MPI_Comm), intent(in) :: comm
6     integer, optional, intent(out) :: code
```

Sending, from the address `buf`, a message of count elements of type `datatype`, tagged `tag`, to the process of rank `dest` in the communicator `comm`.
Remark:
This call is blocking : the execution remains blocked until the message can be re-written without risk of overwriting the value to be sent. In other words, the execution is blocked as long as the message has not been received.

Blocking Receive MPI_Recv

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

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

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

Point-to-point Communications

Example (see Fig. 10)

```
program point_to_point
    use mpi_f08
    implicit none
    TYPE (MPI_Status) :: status_msg
    integer, parameter :: tag=100
    integer :: rank,value
    call MPI_INIT()
    call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
    if (rank == 2) then
        value=1000
        call MPI_SEND(value,1,MPI_INTEGER,5,tag,MPI_COMM_WORLD)
    elseif (rank == 5) then
        call MPI_RECV (value,1,MPI_INTEGER,2,tag,MPI_COMM_WORLD,status_msg)
        print *,'I, process 5, I received ',value,' from the process 2.'
    end if
    call MPI_FINALIZE()
end program point_to_point
```

```
> mpiexec -n 7 point_to_point
I, process 5, I received }1000\mathrm{ from the process 2
```

Point-to-point Communications

Fortran MPI Datatype

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

```
MPI_TYPE_CREATE_F90_INTEGER(),MPI_TYPE_CREATE_F90_REAL() and MPI\_TYPE\_CREATE\_F90\_COMPLEX()
```

```
MPI_TYPE_CREATE_F90_INTEGER(r, newtype, code)
INTEGER, INTENT(IN) :: r
TYPE (MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: code
MPI_TYPE_CREATE_F90_REAL(p, r, newtype, code)
INTEGER, INTENT(IN) :: p, r
TYPE (MPI_Datatype), INTENT (OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: code
```

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

Point-to-point Communications

Other possibilities

- When receiving a message, the rank of the sender process and the tag can be replaced by *jokers* : `MPI\_ANY\_SOURCE` and `MPI\_ANY\_TAG`, respectively.
- A communication involving the dummy process of rank `MPI\_PROC\_NULL` has no effect.
- `MPI\_STATUS\_IGNORE` is a predefined constant, which can be used instead of the status variable.
- It is possible to send more complex data structures by creating derived datatypes.
- There are other operations, which carry out both send and receive operations simultaneously : `MPI\_Sendrecv()` and `MPI\_Sendrecv\_replace()`.

Point-to-point Communications

Simultaneous send and receive MPI_Sendrecv

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

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

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

Remark :

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

Simultaneous send and receive MPI_Sendrecv

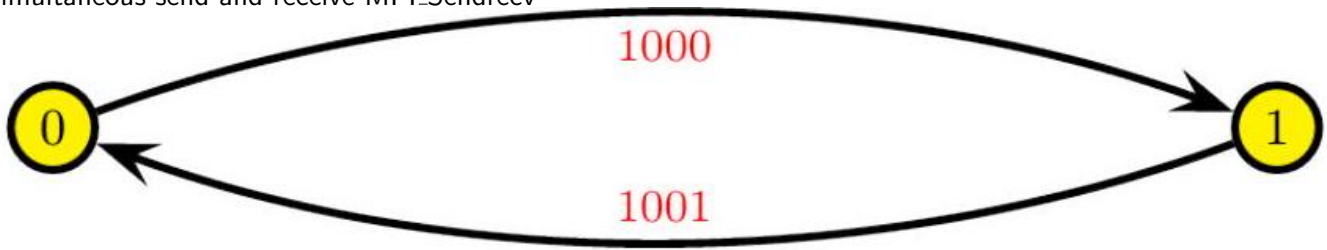

Figure 11 - sendrecv Communication between the Processes 0 and 1

Point-to-point Communications

Example (see Fig. 11)

```
program sendrecv
    use mpi_f08
    implicit none
    integer :: rank,value,num_proc
    integer,parameter :: tag=110
    call MPI_INIT()
    call MPI_COMM_RANK(MPI_COMM_WORLD,rank)
    ! We define the process we'll communicate with (we suppose that we have exactly 2 processes)
    num_proc=mod (rank+1,2)
    call MPI_SENDRECV (rank+1000,1,MPI_INTEGER,num_proc,tag,value,1,MPI_INTEGER, &
            num_proc,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE)
    print *,'I, process ',rank,', I received',value,'from process ', num_proc
    call MPI_FINALIZE()
end program sendrecv
```

```
>mpiexec -n 2 sendrecv
I, process 1, I received 1000 from process 0
I, process 0, I received 1001 from process 1
```

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 MP I_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.

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

Point-to-point Communications

Simultaneous send and receive MPI_Sendrecv_replace

```
1  MPI_SENDRECV_REPLACE (buf,count, datatype,
2      dest, sendtag,
3      source, recvtag, comm, status_msg, code)
4
5      TYPE (*),dimension(..) :: buf
6      integer, intent(in) :: count, source, dest, sendtag, recvtag
7      TYPE(MPI_Datatype), intent(in) :: datatype
8      TYPE (MPI_Comm), intent(in) :: comm
9      TYPE (MPI_Status) :: status_msg
10     integer, optional, intent(out) :: code
```

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

Remark:

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

Point-to-point Communications

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

Point-to-point Communications

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

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. $90 in such a way that the process 0 sends a message containing 1000 random reals to process 1.
2. Ping-Pong : complete the script ping_pong_2.f90 in such a way that the process 1 sends back the message to the process 0 , and measure the communication duration with the MPI_Wt ime () function.
3. Ping-Pong match : complete the script ping_pong_3.f 90 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 compil the first step : make ping_pong_1
- To execute the first step : make exe1
- To compil the second step : make ping_pong_2
- To execute the second step : make exe2
- To compil the last step : make ping_pong_3
- To execute the last step : make exe3
- The generation of random numbers uniformly distributed in the range [0,1[is made by calling the Fortran random_number subroutine :

```
1    call random_number(variable)
```

variable can be a scalar or an array

- The time duration measurements can be done like this :

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