

Introduction to MPI

Anne Cadiou ¹ Fabrice Roy ² (translation)

¹Laboratoire de Mécanique des Fluides et d'Acoustique

²Laboratoire Univers et Théories

Ecole Thématique PhyNuBE : Première rencontre de Physique Nucléaire
de Basse Energie 2021
(december 9th 2021)

Message passing programming model

Definition

- 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.

This slide comes from the IDRIS MPI course

Why use this programming model ?

Modern supercomputers

- Distributed memory computers composed of several nodes
- One or several processors in each node
- ⇒ One or several processes of a distributed application can run on each node
- The nodes are connected with a high performance network
- ⇒ Messages can be exchanged through this network to enable distribution of the workload, synchronization, etc.

Introduction to MPI

Definition

- **M**essage **P**assing **I**nterface
- Library and standard for communications between computing nodes

Usable on computers with shared or distributed memory

Fast and portable

MPI

- Manages message passing between processes (data transfer, synchronization, global operations)
- Based on SPMD principle (Single Program Multiple Data)
- Each process has its own data
- Communications between processes are done in a communicator
- Each process is identified by its rank within the communicator

History

- Concept of standard discussed in 1991
- First standard presented at Supercomputing 93'
- MPI-1 release, 1994
 - library of functions usable with C, C++, Fortran
- MPI-2, 1998
 - MPI I/O,
 - dynamic processes management,
 - one-sided communications,
 - C++ interface becomes obsolete (external "C")
- MPI-3, 2012
 - collective non-blocking operations,
 - explicit functions for shared memory architectures,
 - modern Fortran interface (2003, 2008)
 - C++ support is dropped
 - interfacing with external tools (debugging, profiling)
- MPI-4 ?
 - accelerators support ?
 - fault tolerance ?
 - Discussion at EuroMPI, sept. 2019
 - Usage survey (feb. 2019), <https://bosilca.github.io/MPIsurvey/>

Classic implementations for C, C++, Fortran. There are also implementations in Python, Julia, Java, OCaml, Perl, etc.

Alternatives ?

Other kind of parallelism

- OpenMP, TBB, Pthreads (shared memory)
- OpenACC, CUDA, OpenCL (accelerators)
- StarPU (task parallelism)
- kokkos, Alpaka (cross-platforms)

General structure of a MPI program

```
// beginnng of the program  
  
// include of the library  
  
// initialization of the MPI environment  
  
// calls to the library to exchange messages  
  
// closing of the MPI environment  
  
// end of the program
```


Include MPI

Fortran

```
include 'mpif.h'
```

```
use mpi
```

```
f95 prog.f -lmpi  
mpif90 prog.f
```

```
mpirun -np 4 ./a.out
```

Fortran 2008 (MPI-3)

```
use mpi_f08
```

C

```
#include <mpi.h>
```

```
gcc prog.c -lmpi  
mpicc prog.c
```

```
mpirun -np 4 ./a.out
```

C++

```
#include <mpi.h>
```

```
extern "C" {  
    #include <mpi.h>  
}
```

```
gcc prog.cpp -lmpi  
mpicxx prog.cpp
```

```
mpirun -np 4 ./a.out
```

Python

```
import mpi4py.MPI as MPI
```

```
mpirun -np 4 python \  
script.py
```

MPI Environment

Fortran Initialization

```
CALL MPI_INIT(IERR)
```

Closing

```
CALL MPI_FINALIZE(IERR)
```

C Initialization

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

Closing

```
int MPI_Finalize(void)
```

Python Automatic initialization

```
print(MPI.Is_initialized())

mpi4py.rc.initialize = False
MPI.Init()
```

Closing

```
print(MPI.Is_finalized())

mpi4py.rc.finalize = False
MPI.Finalize()
```

MPI creates a communicator that contains all processes.
Its default name is MPI_COMM_WORLD (MPI.COMM_WORLD for Python)

Communicator(s)

Type MPI : MPI_Comm

example :

Fortran 2008

```
use mpi_f08
Type(MPI_Comm) :: comm =
    MPI_COMM_WORLD
```

C

```
#include <mpi.h>
MPI_Comm comm =
    MPI_COMM_WORLD;
```

Python

```
comm = MPI.COMM_WORLD
```

Older Fortran

```
use mpi
Integer :: comm =
    MPI_COMM_WORLD
```

- Defines a group of active processes
 - Can be created or destroyed during the execution (\geq MPI-2)
- A process has one or several identifiers (communicator, rank)
 - A process can belong to several communicators.
 - A process can have a different rank in each communicator.
- MPI communications must specify the communicator in which they take place.

A communicator can be decomposed, associated with a particular topology, etc.

Syntax of calls to MPI variables and functions

Use the prefix MPI_

mpi4py with Python is based on the obsolete C++ syntax. Class MPI.

Fortran

```
CALL MPI_XXX(parameter,...,
             ierr)
call mpi_xxx(parameter,...,
             ierr)

CALL MPI_BSEND(buf,count,
               type,dest,tag,comm,
               ierr)
```

ierr returns the error code.

MPI_SUCCESS if ok.

C

```
rc = MPI_Xxx(parameter,...)

rc = MPI_Bsend(&buf,count,
               type,dest,tag,comm)
```

rc returns the error code.
MPI_SUCCESS if ok.

Warning : in C, you must respect the case. MPI is always capital, first letter of the second word in capital.

Python

based on C++ interface from MPI-2

```
comm = MPI.COMM_WORLD

# communication of
# Python objects (
# SLOW)
comm.xxx(data,parameters
,...)
comm.bsend(buffer,dest=0,
           tag=1)

# communication of buffer-
# like objects
comm.Xxx([data,count,MPI.
         type],parameters
,...)
comm.Bsend([buffer,
           bufsize,MPI.INT],
           dest=0,tag=1)
```

Example of process identification

Initialize the MPI environment and returns the process rank for each active process in the default communicator.

Execution :

```
mpirun -np 4 ./a.out
```

```
mpirun -np 4 python script.py
```

Output :

```
Hello! I am process 0 of 4 on plume.  
Hello! I am process 1 of 4 on plume.  
Hello! I am process 2 of 4 on plume.  
Hello! I am process 3 of 4 on plume.
```

Example of process identification in Fortran

```
PROGRAM hello

  USE mpi
  IMPLICIT NONE

  INTEGER :: numtasks, rank, reslen, ierr = 0
  CHARACTER(MPI_MAX_PROCESSOR_NAME) :: hostname

  CALL MPI_INIT(ierr)

  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

  CALL MPI_GET_PROCESSOR_NAME(hostname, reslen, ierr)
  WRITE(*, '(2A,I2,A,I2,3A)') &
    'Hello! ', &
    'I am process ', rank, &
    ' of ', numtasks, &
    ' on ', hostname(1:reslen), '.'

  CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

Example of process identification in C

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    int numtasks, rank, reslen, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Get_processor_name(hostname, &reslen);
    printf("Hello! I am process %d of %d on %s.\n",rank,numtasks,hostname);

    MPI_Finalize();
}
```

Example of process identification in Python

```
#!/usr/bin/env python
"""
for python3
"""

import mpi4py.MPI as MPI

rank = MPI.COMM_WORLD.Get_rank()
numtasks = MPI.COMM_WORLD.Get_size()
hostname = MPI.Get_processor_name()

mess = "Hello! I am process %d of %d on %s."
print(mess % (rank, numtasks, hostname))
```


Communications

Communications exchange messages between at least 2 processes

- one sends
- the other receives

Different kinds of communications

- Point to point communications
- Global (or collective) communications
- One-sided communications, where data movement is decoupled from process synchronization (\geq MPI-2)

Communications can be blocking or non-blocking
The messages exchanged are typed data.

Main data types in Fortran

MPI data type	Fortran data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

Main data types in C

MPI data type	C data type
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Main data types in Python

MPI data type	NumPy data type
<code>MPI.INTEGER</code>	<code>np.intc</code>
<code>MPI.LONG</code>	<code>np.int</code>
<code>MPI.FLOAT</code>	<code>np.float32</code>
<code>MPI.DOUBLE</code>	<code>np.float64</code>

Point to point communications

Communication between 2 processes : the sender and the receiver
It includes 2 stages : sending and receiving

A message consists of its header, which contains

- the communicator
- the rank of the sending process
- the rank of the receiving process
- a (tag) which allows the program to distinguish between different messages

and its content, which contains

- the exchanged data
- its type
- its size

Sending/receiving a message

Fortran

```
CALL MPI_SEND(message,size,type,destination,tag,MPI_COMM_WORLD,ierr)
```

```
CALL MPI_RECV(message,size,type,source,tag,MPI_COMM_WORLD,status,ierr)
```

C

```
MPI_Send(void* data, int count, datatype, int dest, int tag, communicator)
```

```
MPI_Recv(void* data, int count, datatype, int source, int tag, communicator, status)
```

Python

```
MPI.COMM_WORLD.send(data, dest=#, tag=#)
```

```
MPI.COMM_WORLD.Send(data, dest=#, tag=#)
```

```
data = MPI.COMM_WORLD.recv(source=#, tag=#)
```

```
MPI.COMM_WORLD.Recv(data, source=#, tag=#)
```

(send/recv : generic Python objects, Send/Recv : NumPy arrays, faster)

Example : One process reads data

Process 0 (always exists) reads an integer and sends it to the other processes, which send back this integer multiplied by their rank in the communicator. (master/slave setup)

```
$ mpirun -np 4 ./a.out

Enter integer number:
12
Slave 1 has received n=12 from 0
Slave 2 has received n=12 from 0
Slave 3 has received n=12 from 0
Master 0 received from slave 1: 12
Master 0 received from slave 2: 24
Master 0 received from slave 3: 36
```

Example in Fortran

```

1 PROGRAM point_a_point
2
3     USE mpi
4     IMPLICIT NONE
5
6     INTEGER, DIMENSION(MPI_STATUS_SIZE) :: statut
7     INTEGER, PARAMETER :: tagm=101, tagr=201, master=0
8     INTEGER :: rang,nprocs,i,n,ierr
9
10    CALL MPI_INIT(ierr)
11
12    CALL MPI_COMM_RANK(MPI_COMM_WORLD,rang,ierr)
13    CALL MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
14
15    IF (rang == master) THEN
16        WRITE(*,*) 'Enter integer number:'
17        READ(*,*) n
18        DO i=1,nprocs-1
19            CALL MPI_SEND(n,1,MPI_INTEGER,i,tagm,MPI_COMM_WORLD,ierr)
20        END DO
21        DO i=1,nprocs-1
22            CALL MPI_RECV(n,1,MPI_INTEGER,i,tagr,MPI_COMM_WORLD,statut,ierr)
23            WRITE(*,'(A,I2,A,I2,A,I3)') "Master",rang," received from slave ",i,": n=",n
24            WRITE(*,*) n,statut(MPI_SOURCE),statut(MPI_TAG),statut(MPI_ERROR)
25        END DO
26    ELSE

```



```
28  CALL MPI_RECV(n,1,MPI_INTEGER,master,tagm,MPI_COMM_WORLD,statut,ierr)
    WRITE(*,'(A,I2,A,I3,A,I2)') "Slave ",rang," has received n=",n," from ",master
    WRITE(*,*) n,statut(MPI_SOURCE),statut(MPI_TAG),statut(MPI_ERROR)
30  n = n*rang
    CALL MPI_SEND(n,1,MPI_INTEGER,master,tagr,MPI_COMM_WORLD,ierr)
32  END IF

34  CALL MPI_FINALIZE(ierr)

36  END PROGRAM point_a_point
```

Example in C

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]) {
5
6     int rang,nprocs;
7     int master,tagm,tagr;
8     int n,i;
9     MPI_Status status;
10
11     MPI_Init(&argc,&argv);
12
13     master = 0;
14     tagm = 101;
15     tagr = 201;
16
17     MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
18     MPI_Comm_rank(MPI_COMM_WORLD,&rang);
19
20     if (rang == master) {
```

```
22 printf("Enter integer number:\n");
scanf("%d",&n);
24 for (i=1; i<nprocs; i++) {
    MPI_Send(&n,1,MPI_INT,i,tagm,MPI_COMM_WORLD);
}
26 for (i=1; i<nprocs; i++) {
    MPI_Recv(&n,1,MPI_INT,i,tagr,MPI_COMM_WORLD,&status);
28    printf("Master %d received from slave %d: %d\n",rang, i, n);
}
30 } else {
    MPI_Recv(&n,1,MPI_INT,master,tagm,MPI_COMM_WORLD,&status);
32    printf("Slave %d has received n=%d from %d\n",rang,n,master);
    n = n*rang;
34    MPI_Send(&n,1,MPI_INT,master,tagr,MPI_COMM_WORLD);
}
36
38 MPI_Finalize();
}
```

Example in Python

```
#!/usr/bin/env python
2  """ for python3 """
import mpi4py.MPI as MPI
4
rang = MPI.COMM_WORLD.Get_rank()
6 nprocs = MPI.COMM_WORLD.Get_size()

8 master = 0
tagm = 101
10 tagr = 201

12 if rang == master:
    print('Enter integer number:')
14    n = int(input())
    for i in range(1,nprocs):
16        MPI.COMM_WORLD.send(n,dest=i,tag=tagm)
    for i in range(1,nprocs):
18        n = MPI.COMM_WORLD.recv(source=i,tag=tagr)
        print("Master ",rang," received from slave ",i,": n=",n)
20 else:
    n = MPI.COMM_WORLD.recv(source=0,tag=tagm)
22    print("Slave ",rang," has received n=",n," from ",master)
    n = n*rang
24    MPI.COMM_WORLD.send(n,dest=0,tag=tagr)
```

Example in Python with NumPy structures

```
#!/usr/bin/env python
2 """
for python3
4 """
6 import mpi4py.MPI as MPI
import numpy as np
8
rang = MPI.COMM_WORLD.Get_rank()
10 nprocs = MPI.COMM_WORLD.Get_size()
12 master = 0
tagm = 101
14 tagr = 201
16 if rang == master:
    print('Enter integer number:')
    n = input()
    data = np.array([n],dtype='i')
    20 for i in range(1,nprocs):
        MPI.COMM_WORLD.Send([data,MPI.INT],dest=i,tag=tagm)
    22 for i in range(1,nprocs):
        MPI.COMM_WORLD.Recv([data,MPI.INT],source=i,tag=tagr)
        print("Master ",rang," received from slave ",i,": n=",data[0])
24 else:
```

```
26 data = np.empty(1, dtype='i')
    MPI.COMM_WORLD.Recv([data, MPI.INT], source=0, tag=tagm)
28 print("Slave ", rang, " has received n=", data[0], " from ", master)
    data[0] = data[0]*rang
30 MPI.COMM_WORLD.Send([data, MPI.INT], dest=0, tag=tagr)
```

```
(mpi4py3) $ mpirun -np 4 python point_a_point_numpy.py
```

```
Enter integer number:
```

```
12
```

```
Slave 1 has received n= 12 from 0
```

```
Slave 3 has received n= 12 from 0
```

```
Master 0 received from slave 1 : n= 12
```

```
Slave 2 has received n= 12 from 0
```

```
Master 0 received from slave 2 : n= 24
```

```
Master 0 received from slave 3 : n= 36
```

Blocking communications

`MPI_SEND` and `MPI_RECV` are blocking communications

- Execution stops until sending and receiving are completed
- Warning : headers and datas in send and receive functions must match (source, tag, etc.)

Advantage

- Data is completely sent or received before it can be accessed or modified by the program

Disadvantage

- Computations stop until the end of the communication

Optimize point to point communications

Communications are blocking because of

- data are copied in temporary memory space (buffer)
- synchronization (the application waits for a matching receive begins before it continues the execution after a send)

Optimize consists in **minimizing the time spent doing something other than computations** (overhead)

Options :

- Overlap communications with computations
- Avoid using buffers
- Minimize overheads related to multiple calls to communications functions

Communications can be standard, synchronous, buffered ou persistent.

Asynchronous communications

The execution continues before the communication has completed : it is possible to compute during the communication.

Fortran

```
CALL MPI_ISEND(message,size,type,destination,tag,comm,request,ierr)
```

```
CALL MPI_IRecv(message,size,type,source,tag,comm,request,ierr)
```

C

```
MPI_Isend(void* data, int count, datatype, int dest, int tag, comm, request)
```

```
MPI_Irecv(void* data, int count, datatype, int source, int tag, comm, status)
```

Python

```
MPI.COMM_WORLD.isend(data, dest=#, tag=#)
```

```
MPI.COMM_WORLD.Isend(data, dest=#, tag=#)
```

```
data = MPI.COMM_WORLD.irecv(source=#, tag=#)
```

```
MPI.COMM_WORLD.Irecv(data, source=#, tag=#)
```

Related check functions

Fortran

Wait until a request has completed

```
CALL MPI_WAIT(request,status,ierr)
```

Check whether a request has completed

```
CALL MPI_TEST(request,flag,status,ierr)
```

Check if a message has arrived

```
CALL MPI_PROBE(source,tag,status,comm,ierr)
```

There are asynchronous versions of these functions.

C

Wait until a request has completed

```
MPI_Wait(request,status)
```

Check whether a request has completed

```
MPI_Test(request,flag,status)
```

Check if a message has arrived

```
MPI_Probe(source,tag,comm,flag,status)
```

Python

Wait until a request has completed

```
req.wait()
```

```
MPI.Request.Wait(req)
```

Check whether a request has completed

```
MPI_Test(request,flag,status)
```

Check if a message has arrived

```
MPI_Probe(source,tag,comm,flag,status)
```

Main point to point communication functions

Fortran	C	Python (Numpy)	type
Send	Send	Send	Send
• MPI_SEND	• MPI_Send	• MPI.Send	blocking, standard
• MPI_ISEND	• MPI_Isend	• MPI.Isend	non blocking, standard
• MPI_SSEND	• MPI_Ssend	• MPI.Ssend	blocking, synchronous
• MPI_ISSEND	• MPI_Issend	• MPI.Issend	non blocking, synchronous
• MPI_BSEND	• MPI_Bsend	• MPI.Bsend	blocking, buffered
• MPI_IBSEND	• MPI_Ibsend	• MPI.Ibsend	non blocking, buffered
Receive	Receive	Receive	Receive
• MPI_RECV	• MPI_Recv	• MPI.Recv	blocking, standard
• MPI_IRECV	• MPI_Irecv	• MPI.Irecv	non blocking, standard
Check	Check	Check	Check
• MPI_WAIT	• MPI.Wait	• MPI.Wait	wait until the communication has completed

Some simple rules

- Initialize receptions before sends
- ⇒ Write calls to `MPI_Irecv` before `MPI_Send`
- Avoid use of buffers
- ⇒ Use synchronous functions `MPI_Ssend`
- Overlap communications with computations
- ⇒ Use non blocking communications `MPI_Isend` and `MPI_Irecv`

Collective communications

- Communication that involves every processes in the communicator (point-to-point communications sequence).
- Processes call the same function with corresponding arguments.
- There is no tag.
- Some collective communication have only one sending process or one receiving process, usually called `root` process.

Main functions

Send and receive

- Synchronization
- Broadcast of a data from the root process to every other processes in the communicator
- Scatter of data from the root process on each process
- Gather of data on the root process

Operation on the data

- Reduction (max, min, sum, product, etc.)

Synchronization

Fortran

```
CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
```

C

```
MPI_Barrier(MPI_COMM_WORLD)
```

Python

```
comm = MPI.COMM_WORLD
```

```
comm.barrier()
```

```
comm.Barrier()
```

Broadcast

Send data from one process to all others (broadcast).

Fortran

```
CALL MPI_BCAST(message,size,type,source,MPI_COMM_WORLD,ierr)
```

C

```
MPI_Bcast(message,size,type,source,MPI_COMM_WORLD)
```

Python

```
comm = MPI.COMM_WORLD
```

```
data = comm.bcast(data,root=source)
```

```
comm.Bcast(data,root=source)
```

Example in Fortran

```
PROGRAM bcast
2  USE mpi
   IMPLICIT NONE
4  INTEGER :: n,nprocs,rang
   INTEGER :: ierr = 0
6  INTEGER, PARAMETER :: idat=91

8  CALL MPI_INIT(ierr)
   CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
10  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rang, ierr)

12  IF (rang.eq.0) THEN
      OPEN(unit=idat,file='data.txt')
14      READ(idat,*) n
      CLOSE(idat)
16  END IF

18  CALL MPI_BCAST(n,1,MPI_INT,0,MPI_COMM_WORLD,ierr)

20  print *, 'Process',rang, ' has received ',n

22  CALL MPI_FINALIZE(ierr)

24  END PROGRAM bcast
```

File :

```
cat data.txt
```

```
29
```

Execution :

```
mpirun -np 4 ./a.out
```

Output :

```
Process 0 has received 29
Process 1 has received 29
Process 2 has received 29
Process 3 has received 29
```

Example in C

```
1 #include "mpi.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char *argv[]) {
6
7     int nprocs,rang;
8     int n;
9
10    MPI_Init(&argc,&argv);
11    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
12    MPI_Comm_rank(MPI_COMM_WORLD, &rang);
13
14    if (rang==0) {
15        FILE *fp = fopen("data.txt", "r");
16        fscanf(fp, "%d", &n);
17        fclose(fp);
18    }
19
20    MPI_Bcast(&n,1,MPI_INTEGER,0,MPI_COMM_WORLD);
21
22    printf("Process %d has received %d \n",rang,n);
23
24    MPI_Finalize();
25    return 0;
26 }
```

Example in Python

```
#!/usr/bin/env python
2 """
  for python3
4 """

6 import mpi4py.MPI as MPI

8 comm = MPI.COMM_WORLD
  rang = comm.Get_rank()
10 nprocs = comm.Get_size()

12 if rang == 0:
    f = open('data.txt','r')
14     n = int(f.read())
  else:
16     n = None

18 n = comm.bcast(n,root=0)

20 print("Process ",rang,"has received ",n)
```

Example in Python with NumPy

```
#!/usr/bin/env python
2 """
for python3
4 """

6 import mpi4py.MPI as MPI
import numpy as np

8 comm = MPI.COMM_WORLD
10 rang = comm.Get_rank()
nprocs = comm.Get_size()

12 if rang == 0:
14     f = open('data.txt','r')
    n = np.array([int(f.read())], dtype=int)
16 else:
    n = np.zeros(1, dtype=int)

18 comm.Bcast(n,root=0)

20 print("Process ",rang,"has received ",n[0])
```

Selective distribution of data

Distribute data from one process to all processes.

Fortran

```
CALL MPI_SCATTER(sendbuf, sendcount, sendtype, &  
    recvbuf, recvcount, recvttype, &  
    root, comm, ierr)
```

C

```
MPI_Scatter(sendbuf, sendcount, sendtype,  
    recvbuf, recvcount, recvttype,  
    root, comm)
```

Python

```
comm = MPI.COMM_WORLD
```

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

```
comm.Scatter([senddata, data_size, data_type], [recvdata, data_size, data_type], root)
```


Example : distribution of a 2D array on 4 processes

Example inspired by <https://computing.llnl.gov/tutorials/mpi>.

Goal : distribute 4×5 array on 4 processes.

$$\begin{pmatrix} 1.0 & 2.0 & 3.0 & 4.0 & 4.5 \\ 5.0 & 6.0 & 7.0 & 8.0 & 8.5 \\ 9.0 & 10.0 & 11.0 & 12.0 & 12.5 \\ 13.0 & 14.0 & 15.0 & 16.0 & 16.5 \end{pmatrix}$$

Execution :

```
mpirun -np 4 ./a.out
```

```
mpirun -np 4 python script.py
```

Output :

```
rank= 0 Results: 1.00000000 2.00000000 3.00000000 4.00000000 4.50000000
rank= 1 Results: 5.00000000 6.00000000 7.00000000 8.00000000 8.50000000
rank= 2 Results: 9.00000000 10.00000000 11.00000000 12.00000000 12.50000000
rank= 3 Results: 13.00000000 14.00000000 15.00000000 16.00000000 16.50000000
```

Example in Fortran

```

1  program scatter
2      include 'mpif.h'
3
4      integer SIZE_X,SIZE_Y
5      parameter(SIZE_X=5,SIZE_Y=4)
6      integer numtasks, rank, sendcount, recvcount, source, ierr
7
8      real*4 sendbuf(SIZE_X,SIZE_Y), recvbuf(SIZE_X)
9      data sendbuf /1.0, 2.0, 3.0, 4.0, 4.5, &
10         5.0, 6.0, 7.0, 8.0, 8.5, &
11         9.0, 10.0, 11.0, 12.0, 12.5, &
12         13.0, 14.0, 15.0, 16.0, 16.5/
13
14      call MPI_INIT(ierr)
15      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
16      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
17
18      if (numtasks .eq. SIZE_Y) then
19
20         source = 1
21         sendcount = SIZE_X
22         recvcount = SIZE_X

```

```
24      call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, &
25                        recvbuf, recvcount, MPI_REAL, &
26                        source, MPI_COMM_WORLD, ierr)
27
28      print *, 'rank= ',rank,' Results: ',recvbuf
29
30  else
31      print *, 'Must specify',SIZE_Y,' processors. Terminating.'
32  endif
33
34  call MPI_FINALIZE(ierr)
35
36  end
```

Example in C

```
1 #include "mpi.h"
2 #include <stdio.h>
3 #define SIZE_X 5
4 #define SIZE_Y 4
5
6 int main(int argc, char *argv[]) {
7
8     int numtasks, rank, sendcount, recvcnt, source;
9     float sendbuf[SIZE_Y][SIZE_X] = {
10         {1.0, 2.0, 3.0, 4.0, 4.5},
11         {5.0, 6.0, 7.0, 8.0, 8.5},
12         {9.0, 10.0, 11.0, 12.0, 12.5},
13         {13.0, 14.0, 15.0, 16.0, 16.5} };
14     float recvbuf[SIZE_X];
15
16     MPI_Init(&argc,&argv);
17     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
19
20     if (numtasks == SIZE_Y) {
21         // define source task and elements to send/receive, then perform collective
22         scatter
23         source = 1;
24         sendcount = SIZE_X;
25         recvcnt = SIZE_X;
```

```
26     MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,  
                MPI_FLOAT,source,MPI_COMM_WORLD);  
  
28     printf("rank= %d Results: %f %f %f %f %f\n",rank,recvbuf[0],  
            recvbuf[1],recvbuf[2],recvbuf[3],recvbuf[4]);  
30 }  
   else  
32     printf("Must specify %d processors. Terminating.\n",SIZE_Y);  
  
34     MPI_Finalize();  
   return 0;  
36 }
```

Example in Python

```
#!/usr/bin/env python
2  """
   for python3
4  """

6  import mpi4py.MPI as MPI

8  comm = MPI.COMM_WORLD
   rang = comm.Get_rank()
10  nprocs = comm.Get_size()

12  if rang == 0:
       data = [[1.0, 2.0, 3.0, 4.0, 4.5], \
14             [5.0, 6.0, 7.0, 8.0, 9.5], \
               [9.0, 10.0, 11.0, 12.0, 12.5], \
16             [13.0, 14.0, 15.0, 16.0, 16.5]]
   else:
18       data = None

20  data = comm.scatter(data, root=0)
   print('rank=',rang,'Results:',data)
```

Example in Python with NumPy

```
#!/usr/bin/env python
2  """
   for python3
4  """

6  import mpi4py.MPI as MPI
   import numpy as np

8

10 comm = MPI.COMM_WORLD
   rang = comm.Get_rank()
   nprocs = comm.Get_size()

12 my_N = 5
14 N = my_N * nprocs

16 if rang == 0:
    data = np.array([[1.0, 2.0, 3.0, 4.0, 4.5], \
18                    [5.0, 6.0, 7.0, 8.0, 9.5], \
                    [9.0, 10.0, 11.0, 12.0, 12.5], \
20                    [13.0, 14.0, 15.0, 16.0, 16.5]], dtype=np.float64)
   else:
22     data = np.empty(N, dtype=np.float64)

24 recv_data = np.empty(my_N, dtype=np.float64)
   comm.Scatter([data,my_N,MPI.DOUBLE], [recv_data,my_N,MPI.DOUBLE], root=0)
26 print('rank=',rang,'Results:',recv_data)
```

Aggregation of the data

Aggregate data from all the processes to one process (root), possibly with broadcast of the result

Fortran

```
CALL MPI_GATHER(sendbuf, sendcount, sendtype, &  
                recvbuf, recvcount, recvtype, &  
                root, comm, ierr)
```

C

```
MPI_Gather(sendbuf, sendcount, sendtype,  
           recvbuf, recvcount, recvtype,  
           root, comm)
```

Python

```
comm = MPI.COMM_WORLD
```

```
data = comm.gather(sendbuff, root)
```

```
comm.Gather([senddata, data_size, data_type], [recvdata, data_size, data_type], root)
```


Reduction

Some operations are carried out on the transferred data, possibly with broadcast of the result.

Fortran

```
CALL MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, ierr)
CALL MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, ierr)
```

C

```
MPI_Reduce(sendbuf,recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root,
           MPI_Comm comm)
MPI_Reduce(sendbuf,recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Python

```
comm = MPI.COMM_WORLD
```

```
comm.reduce(sendobj=None, recvobj=None, op=MPI.SUM, root=0)
comm.allreduce(sendobj=None, recvobj=None, op=MPI.SUM)
```

```
comm.Reduce(sendbuf, recvbuf, op=MPI.SUM, root=0)
comm.Allreduce(sendbuf, recvbuf, op=MPI.SUM)
```

Main reduction operators

<code>MPI.MIN</code>	min
<code>MPI.MAX</code>	max
<code>MPI.SUM</code>	sum
<code>MPI.PROD</code>	product
<code>MPI.MAXLOC</code>	index of the max value
<code>MPI.MINLOC</code>	index of the min value

(Python syntax)

Example of the use of the reduction

Sum the elements of an array of size $N = 1001$. The array is distributed in p chunks, p is the number of processes used.

Execution for $p = 4$:

```
mpirun -np 4 ./a.out
```

```
mpirun -np 4 python script.py
```

Output :

```
[ 0 ] part : 251.000000
[ 2 ] part : 250.000000
[ 3 ] part : 250.000000
[ 1 ] part : 250.000000
Sum : 1001.000000
```

Example in Fortran

```
PROGRAM main

  USE mpi
  IMPLICIT NONE

  INTEGER, parameter :: N=1001
  INTEGER :: ierr, i, rank, nprocs
  INTEGER :: nstart, nstop, npart, ncount, nrem
  DOUBLE PRECISION, allocatable :: vec(:)
  DOUBLE PRECISION :: local_sum, total_sum

  CALL MPI_INIT(ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  CALL MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)

  ncount = N/nprocs
  nrem = MOD(N,nprocs)
  IF (rank < nrem) THEN
    nstart = rank * (ncount + 1)
    nstop = nstart + ncount
  ELSE
    nstart = rank * ncount + nrem
    nstop = nstart + (ncount - 1)
  END IF
  npart = nstop-nstart+1

  allocate(vec(npart))
```

```
DO i=1,npart
    vec(i) = 1.0D0
END DO
local_sum = 0.0D0
DO i=1,npart
    local_sum = local_sum + vec(i)
END DO
WRITE(*,'(A,I3,A,F15.8)') "[" ,rank,"] part : ",local_sum
CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)

CALL MPI_REDUCE(local_sum, total_sum, 1, MPI_DOUBLE, MPI_SUM, 0 , MPI_COMM_WORLD
    , ierr)

IF (rank.eq.0) THEN
    WRITE(*,*) "Sum : ", total_sum
END IF

deallocate(vec)

CALL MPI_FINALIZE(ierr)

END PROGRAM
```

Example in C

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    int i, N;
    int nprocs, rank;

    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    double *vec;
    double local_sum,sum;

    N=1001;

    int count = N/nprocs;
    int remainder = N%nprocs;
    int start, stop;
```

```
if (rank < remainder) {
    start = rank * (count + 1);
    stop = start + count;
} else {
    start = rank * count + remainder;
    stop = start + (count - 1);
}

int npart = stop-start+1;
vec = malloc(sizeof(double)*npart);

for (i=0; i<npart; i++) {
    vec[i] = 1.0;
}
local_sum = 0.0;
for (i=0; i<npart; i++) {
    local_sum += vec[i];
}
printf("[ %d ] part : %g\n",rank,local_sum);
MPI_Barrier(MPI_COMM_WORLD);

MPI_Reduce(&local_sum,&sum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

if (rank == 0) {
    printf("Sum : %g\n",sum);
}
MPI_Finalize();
return 0;
}
```

Example in Python

```
#!/usr/bin/env python
""" for python3 """

import mpi4py.MPI as MPI
import numpy as np
import part as part

comm = MPI.COMM_WORLD
nprocs = comm.size
rank = comm.rank

# Define the size of the problem
N = 1001
start,end = part.partition(rank,nprocs,N)
vec = np.ones((end-start+1),dtype=np.float64)

# Calculate the local sum of local vectors
local_sum = vec.sum()
print("[ %d ] part : %f"%(rank, local_sum))
comm.barrier()

# Get the global sum
global_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)
if rank == 0:
    print("Sum : %f"%(global_sum))
```


Example in Python with NumPy

```
#!/usr/bin/env python
""" for python3 """

import mpi4py.MPI as MPI
import numpy as np
import part as part

comm = MPI.COMM_WORLD
nprocs = comm.size
rank = comm.rank

# Define the size of the problem
N = 1001
start,end = part.partition(rank,nprocs,N)
vec = np.ones((end-start+1),dtype=np.float64)

# Calculate the local sum of local vectors
local_sum = vec.sum()
print("[ %d ] part : %f"%(rank, local_sum))
comm.Barrier()

# Get the global sum
global_sum = np.zeros(1, dtype='float64')
comm.Reduce(local_sum, global_sum, op=MPI.SUM, root=0)
if rank == 0:
    print("Sum : %f"%(global_sum[0]))
```

Other features

The MPI library allows other kinds of features, for instance :

- Definition and use of derived data types
- Creation of communicators
- Use of topologies for communicators
- Parallel I/O

References

- Parallélisme sur machines à mémoire distribuée, Bastien Di Pierro, LyonCalcul, 2016 (in French)
- Calcul parallèle avec MPI, Guy Moebs, Univ. Nantes, 2010 (in French)
- Formations MPI, IDRIS (in French and English)
- <https://computing.llnl.gov/tutorials/mpi>
- <https://www.mpi-forum.org/docs>
- OpenMPI documentation
- mpi4py documentation