

# IPM School of Physics Workshop on High Performance Computing/HPC08 16-21 February 2008

## MPI tutorial



Luca Heltai <luca.heltai@sissa.it>

Stefano Cozzini <cozzini@democritos.it>

Democritos/INFM + SISSA



# When and When NOT?

- Use MPI when you need:
  - parallel code that is portable across platforms
  - higher performance, e.g. when small-scale "Loop-level" parallelism does not provide enough speedup
  
- Do not use MPI when:
  - "Loop level" parallelism is enough (e.g. Using OpenMP)
  - You can use a pre-existing library of parallel routines

Questions 1: General



# Types of MPI Routines

- The MPI standard includes routines for the following operations:
  - Point-to-point communication
  - Collective communications
  - (Process groups)
  - (Process topologies)
  - (Environment management and inquiry)

# MPI basic functions (subroutines)

**MPI\_INIT: initialize MPI**

**MPI\_COMM\_SIZE: how many Processors?**

**MPI\_COMM\_RANK: identify the Processor**

**MPI\_SEND : send data**

**MPI\_RECV: receive data**

**MPI\_FINALIZE: close MPI**

- (Almost) All you need is to know this 6 calls

# Your First Program: Hello World!

## Fortran

```
PROGRAM hello
  INCLUDE 'mpif.h'
  INTEGER err
  CALL MPI_INIT(err)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
  print *, 'I am ', rank, ' of ', size
  CALL MPI_FINALIZE(err)
END
```

## C

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char * argv[])
{
  int rank, size;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD,&rank );
  MPI_Comm_size( MPI_COMM_WORLD,&size );
  printf( "I am %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

# Notes on “Hello”

- All MPI programs begin with MPI\_Init and end with MPI\_Finalize
- MPI\_COMM\_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI “job”
- Each statement executes **independently** in each process
  - **including the printf/print statements**
- I/O not part of MPI-1
  - print and write to standard output or error not part of either MPI-1 or MPI-2
  - output order is undefined (may be interleaved by character, line, or blocks of characters),
    - A consequence of the requirement that non-MPI statements execute independently



# Initializing and Exiting MPI

Initializing the MPI environment

C: `int MPI_Init(int *argc, char ***argv);`

Fortran:

```
INTEGER IERR  
CALL MPI_INIT(IERR)
```

Finalizing MPI environment

C: `int MPI_Finalize();`

Fortran:

```
INTEGER IERR  
CALL MPI_FINALIZE(IERR)
```

This two subprograms should be called by all processes, and no other MPI calls are allowed before `mpi_init` and after `mpi_finalize`

# MPI Communicator

The Communicator is a variable identifying a group of processes that are allowed to communicate with each other.

There is a default communicator (automatically defined):

## **MPI\_COMM\_WORLD**

identify the group of all processes.

- All MPI communication subroutines have a communicator argument.
- The Programmer could define many communicator at the same time



# Communicator Size and Process Rank

How many processors are associated with a communicator?

C:

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

Fortran:

```
INTEGER COMM, SIZE, IERR
```

```
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
```

**OUTPUT: SIZE**

What is the ID of a processor in a group?

C:

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

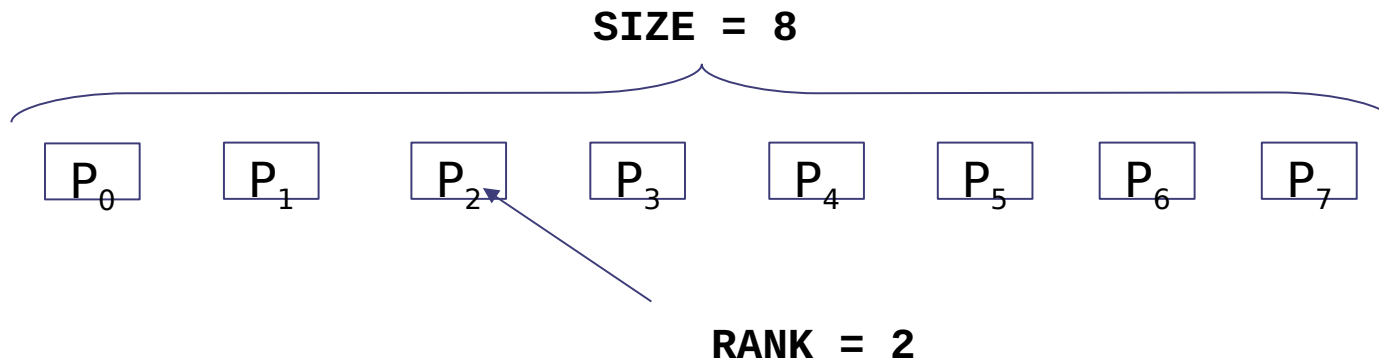
Fortran:

```
INTEGER COMM, RANK, IERR
```

```
CALL MPI_COMM_RANK(COMM, RANK, IERR)
```

**OUTPUT: RANK**

# Communicator Size and Process Rank, cont.



**Size** is the number of processors associated to the communicator

**rank** is the index of the process within a group associated to a communicator (**rank** = 0,1,...,N-1). The rank is used to identify the source and destination process in a communication

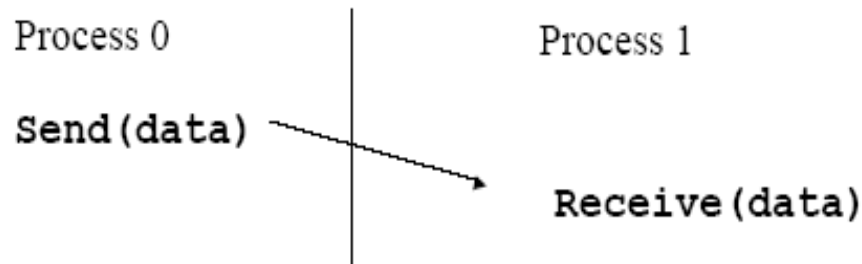
# Communication Ingredients

To send a message via mail we typically have:

- An envelope (with possibly some hints on the content itself... i.e., advertisement, bills, greetings....)
- A message
- A destination address
- A sender address

For MPI it is exactly the same thing...

# MPI basic send/receive



- questions:
  - How will “data” be described? **datatypes**
  - How will processes be identified? **rank/comm**
  - How will the receiver recognize messages? **tag**
  - What will it mean for these operations to complete? **blocking/non-blocking**

# Describing Data

- The data in a message to send or receive is described by a triple (address, count, datatype), where
  - An **MPI datatype** is recursively defined as:
    - predefined, corresponding to a data type from the language (e.g., MPI\_INT, MPI\_DOUBLE)
    - a contiguous array of MPI datatypes
    - a strided block of datatypes
    - an indexed array of blocks of datatypes
    - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays



# Fortran - MPI Basic Datatypes

MPI Data type	Fortran Data type
<b>MPI_INTEGER</b>	<b>INTEGER</b>
<b>MPI_REAL</b>	<b>REAL</b>
<b>MPI_DOUBLE_PRECISION</b>	<b>DOUBLE PRECISION</b>
<b>MPI_COMPLEX</b>	<b>COMPLEX</b>
<b>MPI_DOUBLE_COMPLEX</b>	<b>DOUBLE COMPLEX</b>
<b>MPI_LOGICAL</b>	<b>LOGICAL</b>
<b>MPI_CHARACTER</b>	<b>CHARACTER(1)</b>
<b>MPI_PACKED</b>	
<b>MPI_BYTE</b>	



# C - MPI Basic Datatypes

MPI Data type	C Data type
<b>MPI_CHAR</b>	<b>signed char</b>
<b>MPI_SHORT</b>	<b>signed short int</b>
<b>MPI_INT</b>	<b>signed int</b>
<b>MPI_LONG</b>	<b>Signed long int</b>
<b>MPI_UNSIGNED_CHAR</b>	<b>unsigned char</b>
<b>MPI_UNSIGNED_SHORT</b>	<b>unsigned short int</b>
<b>MPI_UNSIGNED</b>	<b>unsigned int</b>
<b>MPI_UNSIGNED_LONG</b>	<b>unsigned long int</b>
<b>MPI_FLOAT</b>	<b>float</b>
<b>MPI_DOUBLE</b>	<b>double</b>
<b>MPI_LONG_DOUBLE</b>	<b>long double</b>
<b>MPI_BYTE</b>	
<b>MPI_PACKED</b>	



# Data Tag

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying `MPI_ANY_TAG` as the tag in a receive



# Our First Sent Message....

## The simplest call:

`MPI_send( buffer, count, data_type, destination,tag, communicator)`

where:

**BUFFER:** data to send

**COUNT:** number of elements in buffer .

**DATA\_TYPE :** which kind of data types in buffer ?

**DESTINATION** the receiver

**TAG:** the label of the message

**COMMUNICATOR** set of processors involved

# ...and our First Received message.

- The simplest call :
  - Call `MPI_recv( buffer, count, data_type, source, tag, communicator, status, error )`
- Similar to send with the following differences:
  - **SOURCE** is the sender ; can be set as `MPI_any_source` ( receive a message from any processor within the communicator )
  - **TAG** the label of message: can be set as `MPI_any_tag`: receive any kind of message
  - **STATUS** integer array with information on message in case of error

# The status array

- **Status** is a data structure allocated in the user's program.

- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```



# MPI: a FORTRAN example..

```

Program MPI
  Implicit None
  !
  Include 'mpif.h'
  !
  Integer                :: rank
  Integer                :: buffer
  Integer, Dimension( 1:MPI_status_size ) :: status
  Integer                :: error
  !
  Call MPI_init( error )
  Call MPI_comm_rank( MPI_comm_world, rank, error )
  !
  If( rank == 0 ) Then
    buffer = 33
    Call MPI_send( buffer, 1, MPI_integer, 1, 10, &
                  MPI_comm_world, error )
  End If
  !
  If( rank == 1 ) Then
    Call MPI_recv( buffer, 1, MPI_integer, 0, 10, &
                  MPI_comm_world, status, error )
    Print*, 'Rank ', rank, ' buffer=', buffer
    If( buffer /= 33 ) Print*, 'fail'
  End If
  Call MPI_finalize( error )
End Program MPI

```



# Communication Modes

## 4 Send types:

- **Standard:** let MPI decide the best strategy...
- **Synchronous:** it is complete when the receiver acknowledged the reception of the message
- **Buffered:** it is complete when the data has been copied to a local buffer
- **Ready:** requires a receiver to be already waiting for the message

**ONLY ONE Receive type!**

# Blocking and Non-Blocking

Q: When is a SEND instruction complete?

A: When it is safe to change the data that we sent.

Q: When is a RECEIVE instruction complete?

A: When it is safe to access the data we received.

With both communications (send and receive) we have two choices:

- Start a communication and wait for it to complete:  
**BLOCKING** approach
- Start a communication and return control to the main program:  
**NON-BLOCKING** approach

The Non-Blocking approach **REQUIRES** us to check for completion before we can **modify/access** the **sent/received** data!!!

# Pros and Cons of Non-Blocking Send and Receive

Non-Blocking communications allows the separation between the initiation of the communication and the completion.

Advantages: between the initiation and completion the program could do other useful computation (latency hiding).

Disadvantages: the programmer has to insert code to check for completion.

Questions 2: MPI Basic



# Communication Modes and MPI Subroutines

Mode	Completion Condition	Blocking subroutine	Non-blocking subroutine
Standard send	Message sent (receive state unknown)	<b>MPI_SEND</b>	<b>MPI_ISEND</b>
receive	Completes when a message has arrived	<b>MPI_RECV</b>	<b>MPI_IRECV</b>
Synchronous send	Only completes when the receive has completed	<b>MPI_SSEND</b>	<b>MPI_ISSEND</b>
Buffered send	Always completes, irrespective of receiver	<b>MPI_BSEND</b>	<b>MPI_IBSEND</b>
Ready send	Always completes, irrespective of whether the receive has completed	<b>MPI_RSEND</b>	<b>MPI_IRSEND</b>





# Non-Blocking Send and Receive

Fortran:

**MPI\_ISEND**(buf, count, type, dest, tag, comm, req, ierr)

**MPI\_Irecv**(buf, count, type, dest, tag, comm, req, ierr)

**buf** array of type **type** see table.

**count** (INTEGER) number of element of **buf** to be sent

**type** (INTEGER) MPI type of **buf**

**dest** (INTEGER) rank of the destination process

**tag** (INTEGER) number identifying the message

**comm** (INTEGER) communicator of the sender and receiver

**req** (INTEGER) output, identifier of the communications handle

**ierr** (INTEGER) output, error code (if **ierr=0** no error occurs)



# Non-Blocking Send and Receive

C:

```

int MPI_Isend(void *buf, int count,
    MPI_Datatype type, int dest, int tag,
    MPI_Comm comm, MPI_Request *req);
    
```

```

int MPI_Irecv (void *buf, int count,
    MPI_Datatype type, int dest, int tag,
    MPI_Comm comm, MPI_Request *req);
    
```

# Waiting and Testing for Completion

Fortran:

```
MPI_WAIT(req, status, ierr)
```

A call to this subroutine cause the code to wait until the communication pointed by req is complete.

**req** (INTEGER) input/output, identifier associated to a communications event (initiated by **MPI\_ISEND** or **MPI\_IRECV**).

**Status** (INTEGER) array of size **MPI\_STATUS\_SIZE**, if **req** was associated to a call to **MPI\_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

**ierr** (INTEGER) output, error code (if **ierr=0** no error occurs).

C:

```
int MPI_Wait(MPI_Request *req, MPI_Status *status);
```

# Waiting and Testing for Completion

Fortran:

```
MPI_TEST(req, flag, status, ierr)
```

A call to this subroutine sets **flag** to **.true.** if the communication pointed by **req** is complete, sets **flag** to **.false.** otherwise.

**req** (INTEGER) input/output, identifier associated to a communications event (initiated by **MPI\_ISEND** or **MPI\_IRECV**).

**Flag** (LOGICAL) output, **.true.** if communication **req** has completed **.false.** otherwise

**Status** (INTEGER) array of size **MPI\_STATUS\_SIZE**, if **req** was associated to a call to **MPI\_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

**ierr** (INTEGER) output, error code (if **ierr=0** no error occurs).

C:

```
int MPI_Wait(MPI_Request *req, int *flag, MPI_Status *status);
```

# MPI: a case study

Problem: exchanging data between two processes

```

If( rank == 0 ) Then
  Call MPI_send( buffer1, 1, MPI_integer, 1, 10, &
                MPI_comm_world, error )
  Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                MPI_comm_world, status, error )
Else If( rank == 1 ) Then
  Call MPI_send( buffer2, 1, MPI_integer, 0, 20, &
                MPI_comm_world, error )
  Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                MPI_comm_world, status, error )
End If
  
```

**DEADLOCK**

# Solution A

USE BUFFERED SEND: **bsend**  
 send and go back so the deadlock is avoided

```

If( rank == 0 ) Then
  Call MPI_Bsend( buffer1, 1, MPI_integer, 1, 10, &
                 MPI_comm_world, error )
  Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                MPI_comm_world, status, error )
Else If( rank == 1 ) Then
  Call MPI_Bsend( buffer2, 1, MPI_integer, 0, 20, &
                 MPI_comm_world, error )
  Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                MPI_comm_world, status, error )
End If
  
```

## NOTES:

1. Requires a copy therefore is not efficient for large data set memory problems

# Solution B

Use non blocking SEND : **isend**  
 send go back but now is not safe to change the buffer

```

If( rank == 0 ) Then
  Call MPI_Isend( buffer1, 1, MPI_integer, 1, 10, &
                 MPI_comm_world, REQUEST, error )
  Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                MPI_comm_world, status, error )
Else If( rank == 1 ) Then
  Call MPI_Isend( buffer2, 1, MPI_integer, 0, 20, &
                 MPI_comm_world, REQUEST, error )
  Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                MPI_comm_world, status, error )
End If
Call MPI_wait( REQUEST, status ) ! Wait until send is complete
  
```

## NOTES:

- 1 A **handle** is introduced to test the status of message.
2. More efficient of the previous solutions



# Solution C

Exchange send/recv order on one processor

```

If( rank == 0 ) Then
    Call MPI_send( buffer1, 1, MPI_integer, 1, 10, &
                  MPI_comm_world, error )
    Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                  MPI_comm_world, status, error )
Else If( rank == 1 ) Then
    Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                  MPI_comm_world, status, error )
    Call MPI_send( buffer2, 1, MPI_integer, 0, 20, &
                  MPI_comm_world, error )
End If
    
```

**NOTES:**

efficient and suggested !

Questions 3: ISEND-IRECEIVE

32





# Collective operation (1)

- *Collective* routines provide a higher-level way to organize a parallel program
- Each process executes the same communication operations
- MPI provides a rich set of collective operations...

# Collective Communications (2)

- Communications involving group of processes in a communicator.
- Groups and communicators can be constructed “by hand” or using topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations.
- Three classes of operations: synchronization, data movement, collective computation.

# MPI\_Barrier

Stop processes until all processes within a communicator reach the barrier

Almost never required in a parallel program

Occasionally useful in measuring performance and load balancing

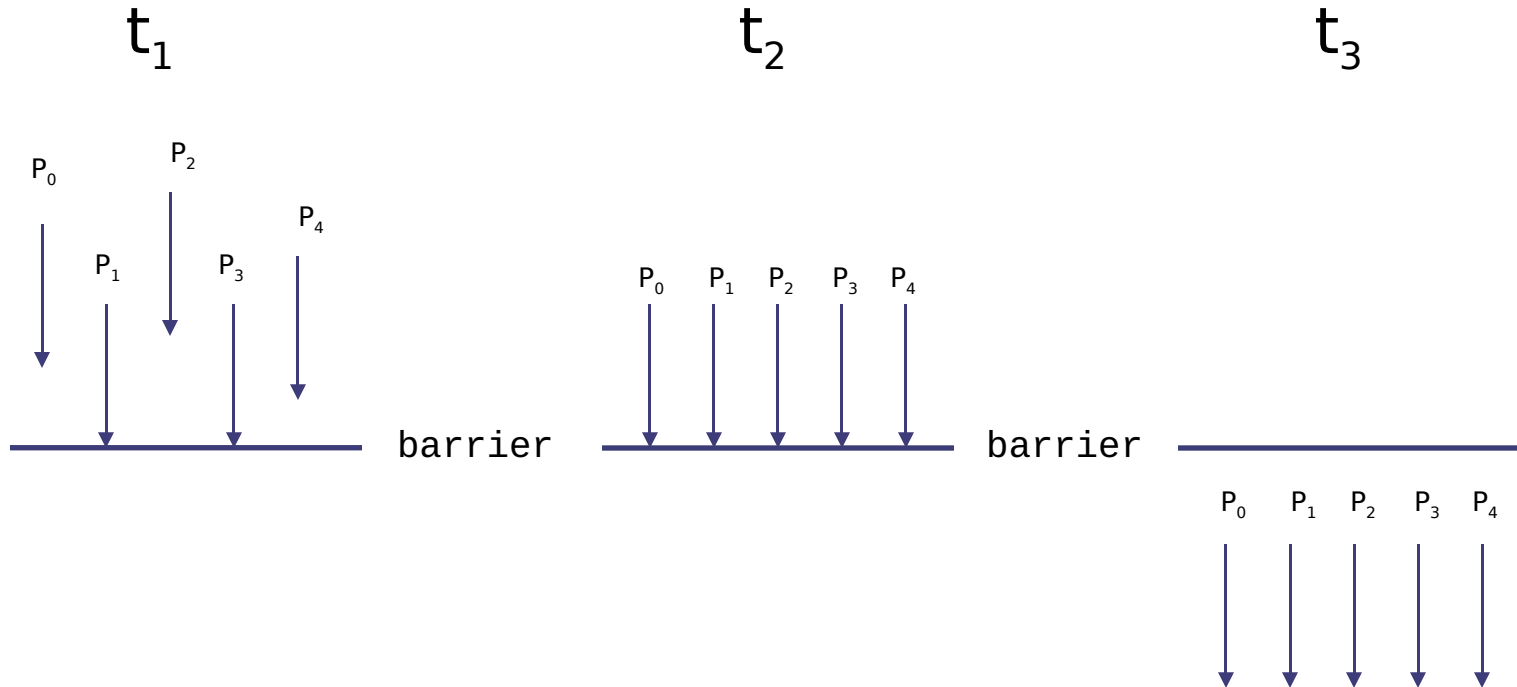
Fortran:

```
CALL MPI_BARRIER( comm, ierr)
```

C:

```
int MPI_Barrier(MPI_Comm comm)
```

# Barrier



# Broadcast (MPI\_BCAST)

One-to-all communication: same data sent from root process to all others in the communicator

Fortran:

```
INTEGER count, type, root, comm, ierr
```

```
CALL MPI_BCAST(buf, count, type, root, comm, ierr)
```

Buf array of type type

C:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype  

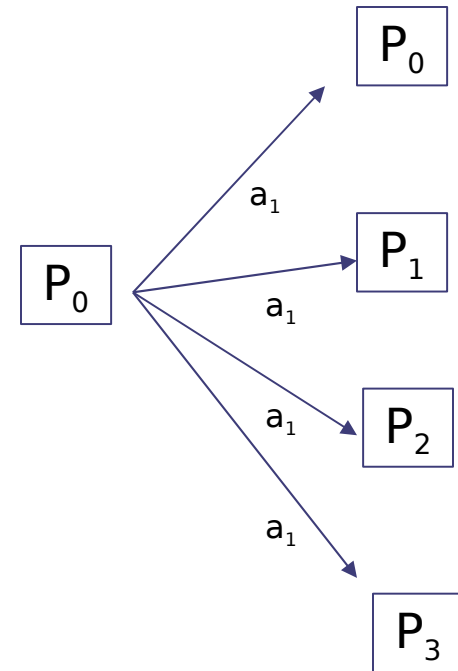
  datatype, int root, MPI_Comm comm)
```

All processes must specify same root, rank and comm

# Broadcast

```

PROGRAM broad_cast
  INCLUDE 'mpif.h'
  INTEGER ierr, myid, nproc, root
  INTEGER status(MPI_STATUS_SIZE)
  REAL A(2)
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
  root = 0
  IF( myid .EQ. 0 ) THEN
    a(1) = 2.0
    a(2) = 4.0
  END IF
  CALL MPI_BCAST(a, 2, MPI_REAL, 0, MPI_COMM_WORLD, ierr)
  WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
  CALL MPI_FINALIZE(ierr)
END
  
```

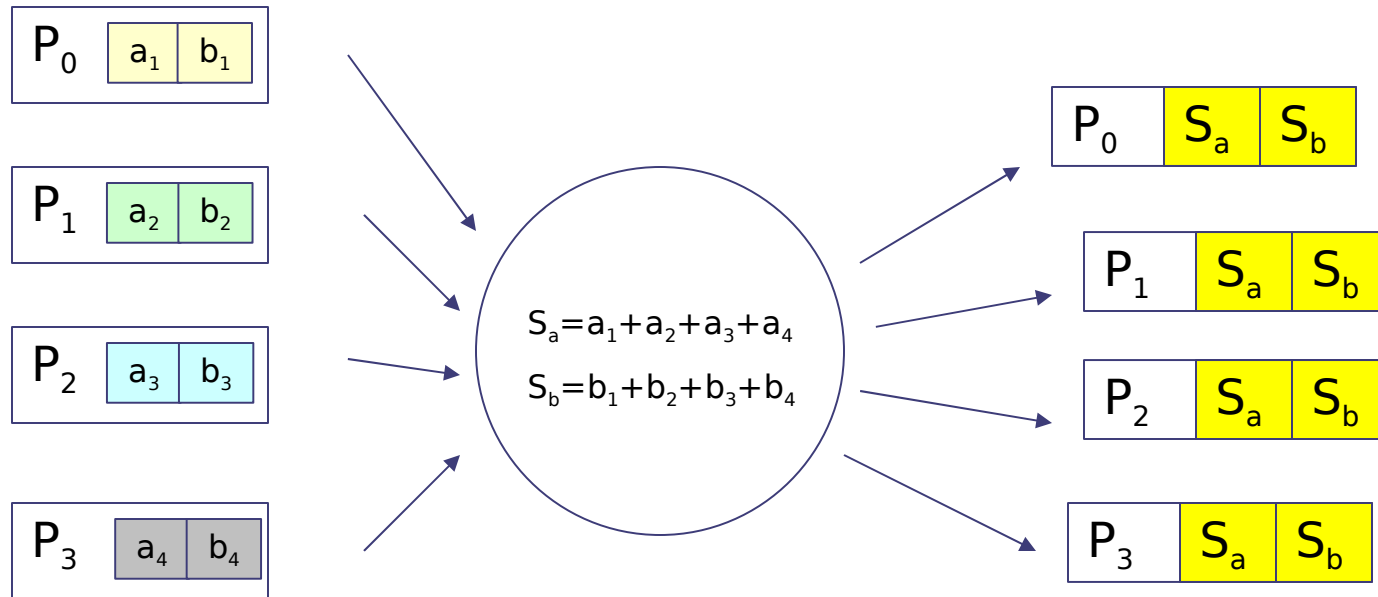


# Reduction

The reduction operation allow to:

- Collect data from each process
- Reduce the data to a single value
- Store the result on the root processes
- Store the result on all processes

# Reduce, Parallel Sum



Reduction function works with arrays  
 other operation: product, min, max, and, ....  
 Internally is usually implemented with a  
 binary tree



# MPI\_REDUCE and MPI\_ALLREDUCE

Fortran:

**MPI\_REDUCE( snd\_buf, rcv\_buf, count, type, op, root, comm, ierr )**

snd\_buf      input array of type type containing local values.  
 rcv\_buf      output array of type type containing global results  
 count        ( INTEGER ) number of element of snd\_buf and rcv\_buf  
 type ( INTEGER ) MPI type of snd\_buf and rcv\_buf  
 op            ( INTEGER ) parallel operation to be performed  
 root ( INTEGER ) MPI id of the process storing the result  
 comm ( INTEGER ) communicator of processes involved in the operation  
 ierr ( INTEGER ) output, error code (if ierr=0 no error occurs)

**MPI\_ALLREDUCE( snd\_buf, rcv\_buf, count, type, op, comm, ierr )**

The argument root is missing, the result is stored to all processes.



# MPI\_Reduce and MPI\_Allreduce

C:

```
int MPI_Reduce(void * snd_buf, void * rcv_buf, int count,
              MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)
```

```
int MPI_Allreduce(void * snd_buf, void * rcv_buf, int count,
                 MPI_Datatype type, MPI_Op op, MPI_Comm comm)
```

# Predefined Reduction Operations

MPI op	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

# Reduce, example

```

PROGRAM reduce
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), res(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
a(1) = 2.0
a(2) = 4.0
CALL MPI_REDUCE(a, res, 2, MPI_REAL, MPI_SUM, root,
& MPI_COMM_WORLD, ierr)
IF( myid .EQ. 0 ) THEN
  WRITE(6,*) myid, ': res(1)=', res(1), 'res(2)=', res(2)
END IF
CALL MPI_FINALIZE(ierr)
END
  
```

# MPI\_Scatter

One-to-all communication: different data sent from root process to all others in the communicator

Fortran:

sender
receiver

```
CALL MPI_SCATTER(sndbuf, sndcount, sndtype, rcvbuf, rcvcount,
                rcvtype, root, comm, ierr)
```

- Arguments definition are like other MPI subroutine
- **sndcount** is the number of elements sent to each process, not the size of **sndbuf**, that should be **sndcount** times the number of process in the communicator
- The sender arguments are significant only at root

# MPI\_Gather

One-to-all communication: different data collected by the root process, from all others processes in the communicator. Is the opposite of Scatter

Fortran:

```
CALL MPI_GATHER(sndbuf, sndcount, sndtype, rcvbuf, rcvcount,
               rcvtype, root, comm, ierr)
```

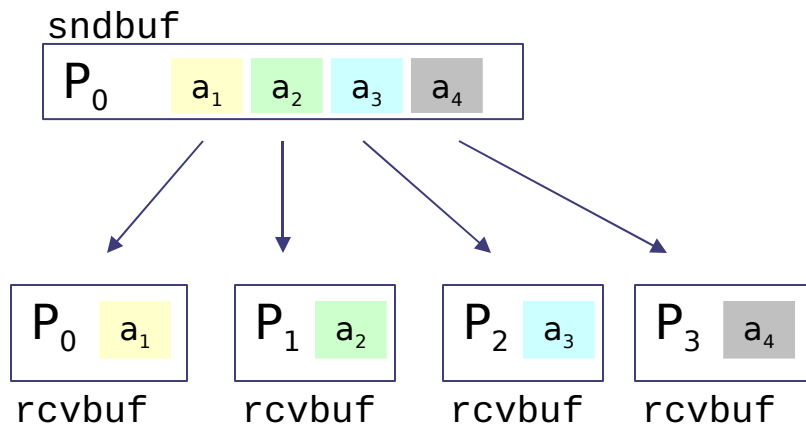
sender
receiver

- Arguments definition are like other MPI subroutine
- **rcvcount** is the number of elements collected from each process, not the size of **rcvbuf**, that should be **rcvcount** times the number of process in the communicator
- The receiver arguments are significant only at root

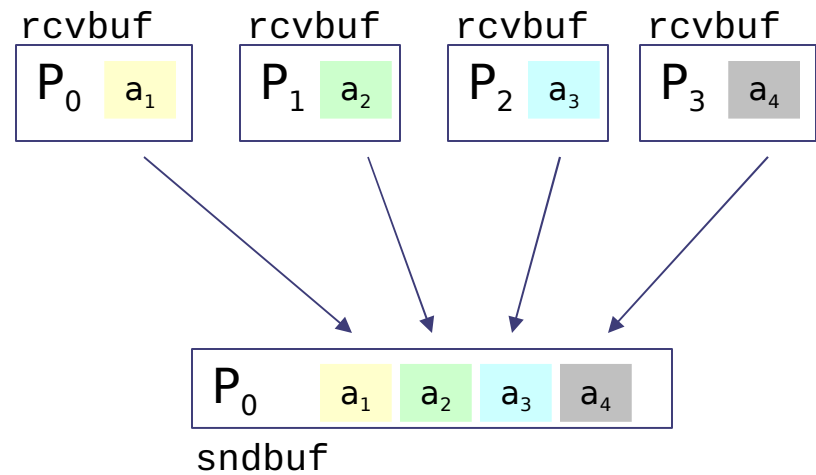


# Scatter/Gather

## Scatter



## Gather



# Scatter/Gather examples

## scatter

```

PROGRAM scatter
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, nsnd, I, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(16), B(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
IF( myid .eq. root ) THEN
  DO i = 1, 16
    a(i) = REAL(i)
  END DO
END IF
nsnd = 2
CALL MPI_SCATTER(a, nsnd, MPI_REAL, b, nsnd,
& MPI_REAL, root, MPI_COMM_WORLD, ierr)
WRITE(6,*) myid, ': b(1)=', b(1), 'b(2)=', b(2)
CALL MPI_FINALIZE(ierr)
END
  
```

## gather

```

PROGRAM gather
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, nsnd, I, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(16), B(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
b(1) = REAL( myid )
b(2) = REAL( myid )
nsnd = 2
CALL MPI_GATHER(b, nsnd, MPI_REAL, a, nsnd,
& MPI_REAL, root, MPI_COMM_WORLD, ierr)
IF( myid .eq. root ) THEN
  DO i = 1, (nsnd*nproc)
    WRITE(6,*) myid, ': a(i)=', a(i)
  END DO
END IF
CALL MPI_FINALIZE(ierr)
END
  
```



# Which MPI routines ?

- For simple applications, these are common:
  - Point-to-point communication
    - MPI\_Irecv, MPI\_Isend, MPI\_Wait, MPI\_Send, MPI\_Recv
  - Startup
    - MPI\_Init, MPI\_Finalize
  - Information on the processes
    - MPI\_Comm\_rank, MPI\_Comm\_size, MPI\_Get\_processor\_name
  - Collective communication
    - MPI\_Allreduce, MPI\_Bcast, MPI\_Allgather

Questions 4: collective



# Useful sites...

- <http://webct.ncsa.uiuc.edu:8900/public/MPI/>
  - Online MPI lecture and tutorial at NCSA.
- <http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/main.htm>
  - Examples from the Using MPI book
- <http://www.lam-mpi.org/tutorials/>
  - A collection of links to more MPI tutorials