Democritos/ICTP course in “Tools for computational physics

MPI tutorial

Stefano Cozzini cozzini@democritos.it
Democritos/INFM + SISSA

Models for parallel computing

- Shared memory (load, store, lock, unlock)
- Message Passing (send, receive, broadcast, ...)
- Transparent (compiler works magic)
- Directive-based (compiler needs help)
- Others (BSP, OpenMP, ...)

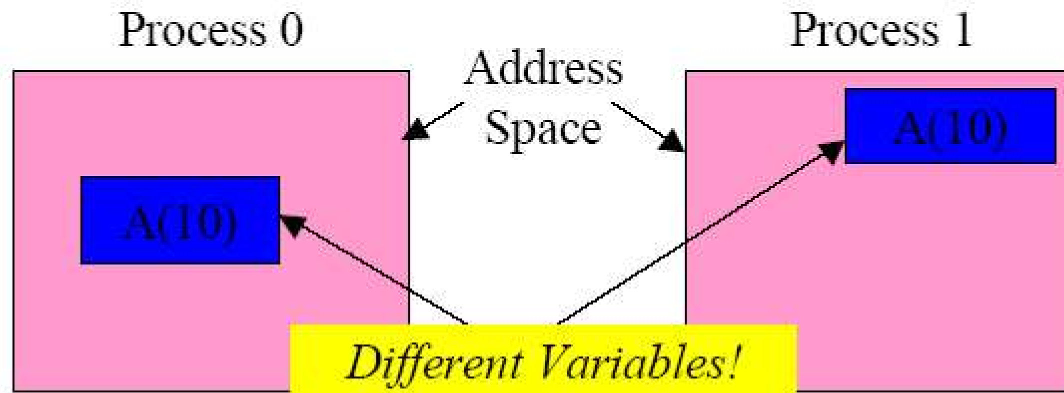
Message passing paradigm

- Parallel programs consist of separate processes, each with its own address space
  - Programmer manages memory by placing data in a particular process
- Data sent explicitly between processes
  - Programmer manages memory motion
- Collective operations
  - On arbitrary set of processes
- Data distribution
  - Also managed by programmer

Types of parallel programming

- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
- Task Parallel - different instructions on different data (MIMD)
- SPMD (single program, multiple data) not synchronized at individual operation level
- SPMD is equivalent to MIMD since each MIMD program can be made SPMD (similarly for SIMD, but not in practical sense.)
- Message passing is for MIMD/SPMD parallelism. HPF is an example of an SIMD

Distributed memory (shared nothing approach)



What is MPI?

- A message-passing library specification
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for end users, library writers, and tool developers

What is MPI?

A STANDARD...

The actual implementation of the standard is demanded to the software developers of the different systems

In all systems MPI has been implemented as a library of subroutines over the network drivers and primitives

many different implementations

- LAM/MPI (today's TOY) www.lam-mpi.org
- MPICH

Goals of the MPI standard

MPI's prime goals are:

- To provide source-code portability
- To allow efficient implementations

MPI also offers:

- A great deal of functionality
- Support for heterogeneous parallel architectures

MPI references

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including talks and tutorials, a FAQ, other MPI pages

How to program with MPI

- MPI is a library
  - All operations are performed with routine calls
  - Basic definitions are in
    - mpi.h for C
    - mpif.h for Fortran 77 and 90
    - MPI module for Fortran 90 (optional)

Basic Features of MPI Programs

Calls may be roughly divided into four classes:

- Calls used to initialize, manage, and terminate communications
- Calls used to communicate between pairs of processors. (Pair communication)
- Calls used to communicate among groups of processors. (Collective communication)
- Calls to create data types.

MPI basic functions (subroutines)

MPI_INIT: initialize MPI
MPI_COMM_SIZE: how many PE ?
MPI_COMM_RANK: identify the PE
MPI_SEND :
MPI_RECV:
MPI_FINALIZE: close MPI

- All you need is to know this 6 calls

A 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>
void 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

Compiling MPI Programs

NO STANDARD: left to the implementations:

Generally:

- You should specify the appropriate include directory (i.e. -I/mpidir/include)
- You should specify the mpi library (i.e. -L/mpidir/lib -lmpi)
- Usually MPI compiler wrappers do this job for you. (i.e. Mpif77)
  - ✓ Check on your machine...

Running MPI programs

- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
- Many implementations provided mpirun –np 4 a.out to run an MPI program
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- **mpiexec <args>** is part of MPI-2, as arecommendation, but not requirement, for implementors.
- Many parallel systems use a *batch* environment to share resources among users
- The specific commands to run a program on a parallel system are defined by the environment installed on the parallel computer

Basic Structures of MPI Programs

- Header files
- MPI Communicator
- MPI Function format
- Communicator Size and Process Rank
- Initializing and Exiting MPI

Header files

All Subprogram that contains calls to MPI subroutine must include the MPI header file

C:

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

Fortran:

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

The header file contains definitions of MPI constants, MPI types and functions

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

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**

C and Fortran: a note

- C and Fortran bindings correspond closely
- In C:
  - mpi.h must be #included
  - MPI functions return error codes or
  - MPI_SUCCESS
- In Fortran:
  - mpif.h must be included, or use MPI module
  - All MPI calls are to subroutines, with a place for the return error code in the last argument.

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:

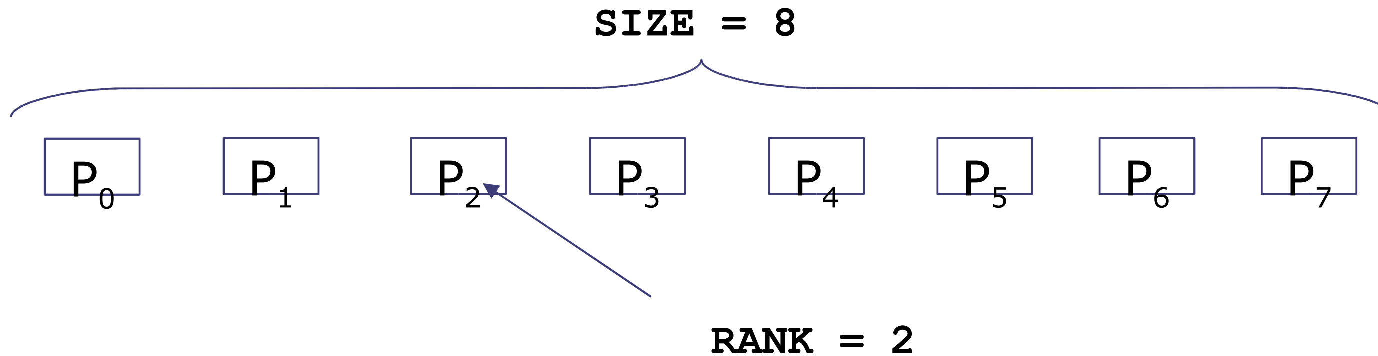
```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran:

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

OUTPUT: RANK

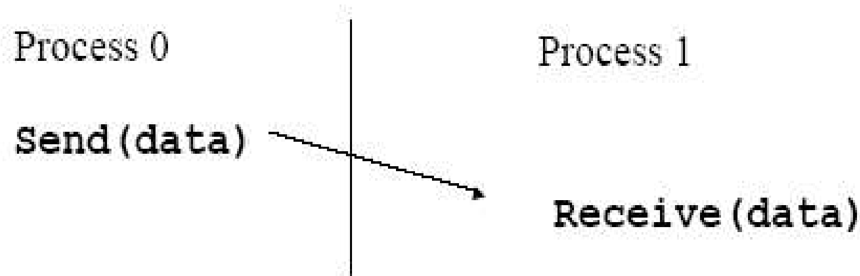
Communicator Size and Process Rank, cont.



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

MPI basic send/receive



- questions:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize messages?
  - What will it mean for these operations to complete?

Basic concepts

- Processes can be collected into groups
- Each message is sent in a context, and
- must be received in the same context
- A group and context together form a
- communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called **MPI_COMM_WORLD**

MPI datatypes

- 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 thelanguage (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
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_PACKED	
MPI_BYTE	

C - MPI Basic Datatypes

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

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
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes

MPI : the call

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

MPI: again on send

- MPI_send is blocking
  - When the control is returned it is safe to change data in BUFFER !!
- The user does not know if MPI implementation:
  - copies BUFFER in an internal buffer, start communication, and returns control before all the data are transferred. (BUFFERING)
  - create links between processors, send data and return control when all the data are sent (but NOT received)
  - uses a combination of the above methods

MPI: receiving message

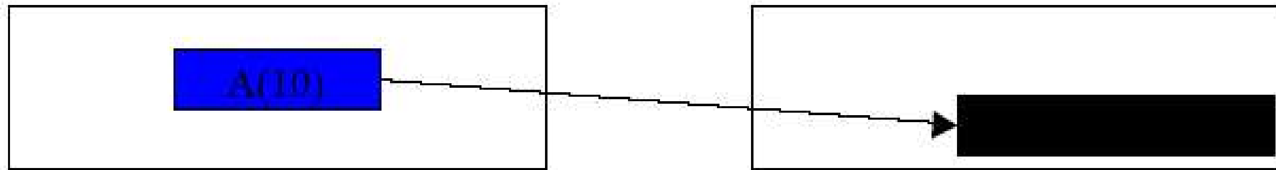
- The simplest call :
  - Call MPI_recv(buffer, count, data_type, source, tag, communicator, status, error)
- Similar to send with the following differences:
  - SOURCE is the sender ; can be set as MPI_any_source (receive a message from any processor within the communicator)
  - TAG the label of message: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

MPI: a fortran example..

```fortran
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
```


33

Summary: MPI send/receive



MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

- Datatype Basic for heterogeneity
  - Derived for non-contiguous
- Contexts
  - Message safety for libraries
- Buffering
  - Robustness and correctness

Tag and context

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tagsused by other libraries.
  - this can be defeated by use of "wild card" tags.
- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when al ibrary sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing

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)
```

Definitions (Blocking and non-Blocking)

- "***Completion***" of the communication means that memory locations used in the message transfer can be safely accessed
  - Send: variable sent can be reused after completion
  - Receive: variable received can now be used

- MPI communication modes differ in what conditions are needed for completion

- Communication modes can be ***blocking*** or ***non-blocking***
  - ***Blocking***: return from routine implies completion
  - ***Non-blocking***: routine returns immediately, user must test for completion

Communication Modes and MPI Subroutines

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

MPI: different ways to communicate

- MPI different "sender mode" :
  - MPI_SSEND: synchronous way: return the control when all the message is received
  - MPI_ISEND: non blocking: start the communication and return control
  - MPI_BSEND: buffered send: creates a buffer,copies the data and returns control
- In the same way different MPI receiving:
  - MPI _IRECV etc…

39

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.

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 occours).

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 occours).

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
2. 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 An 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 !

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 beconstructed “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

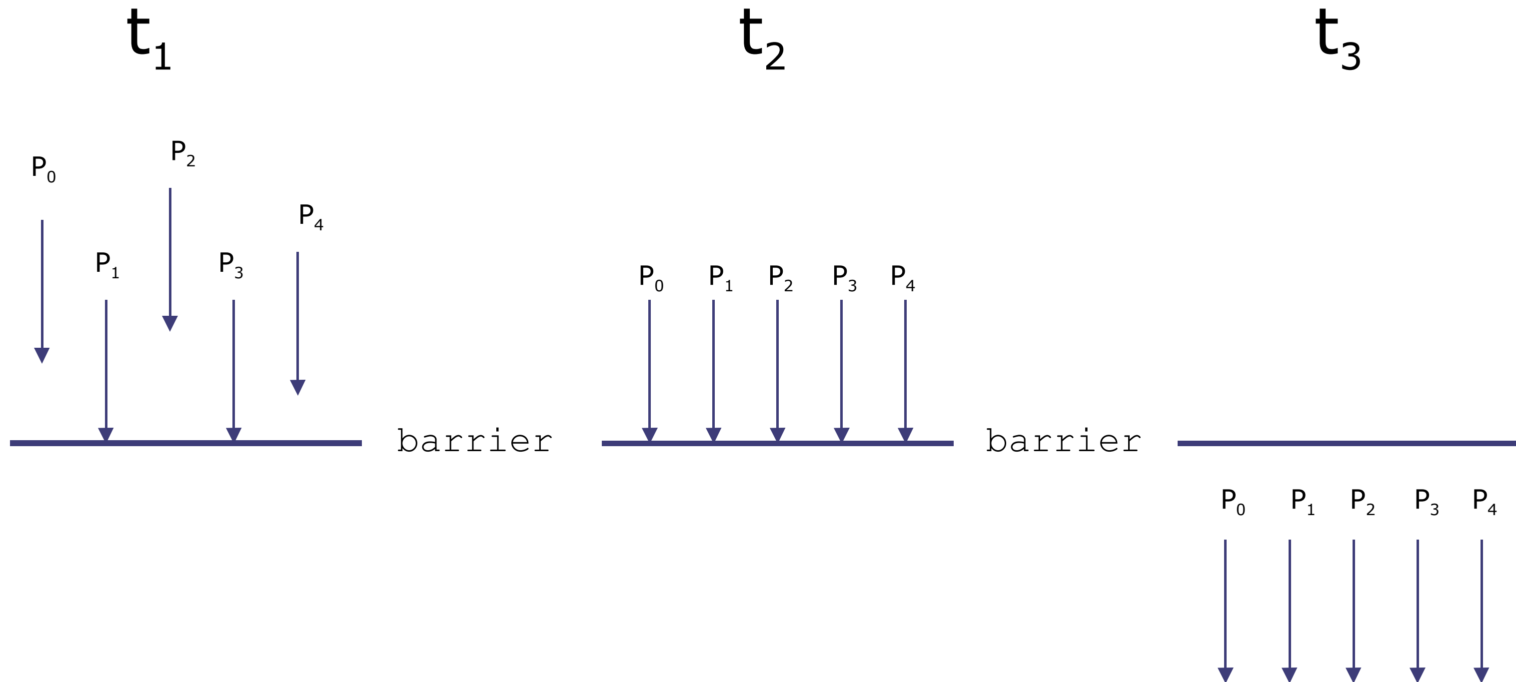
Fortran:

```
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**

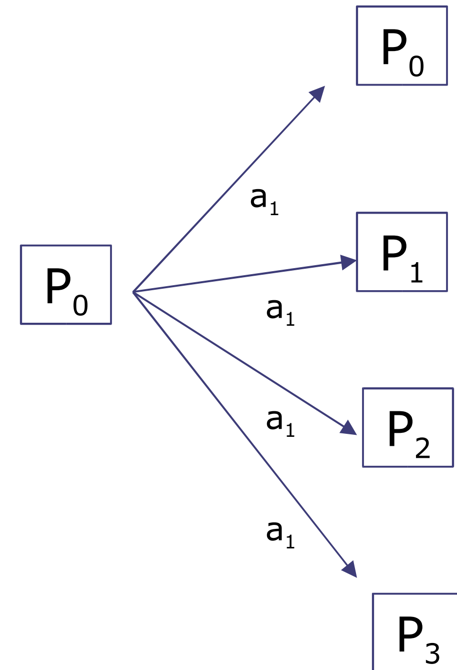
C:

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

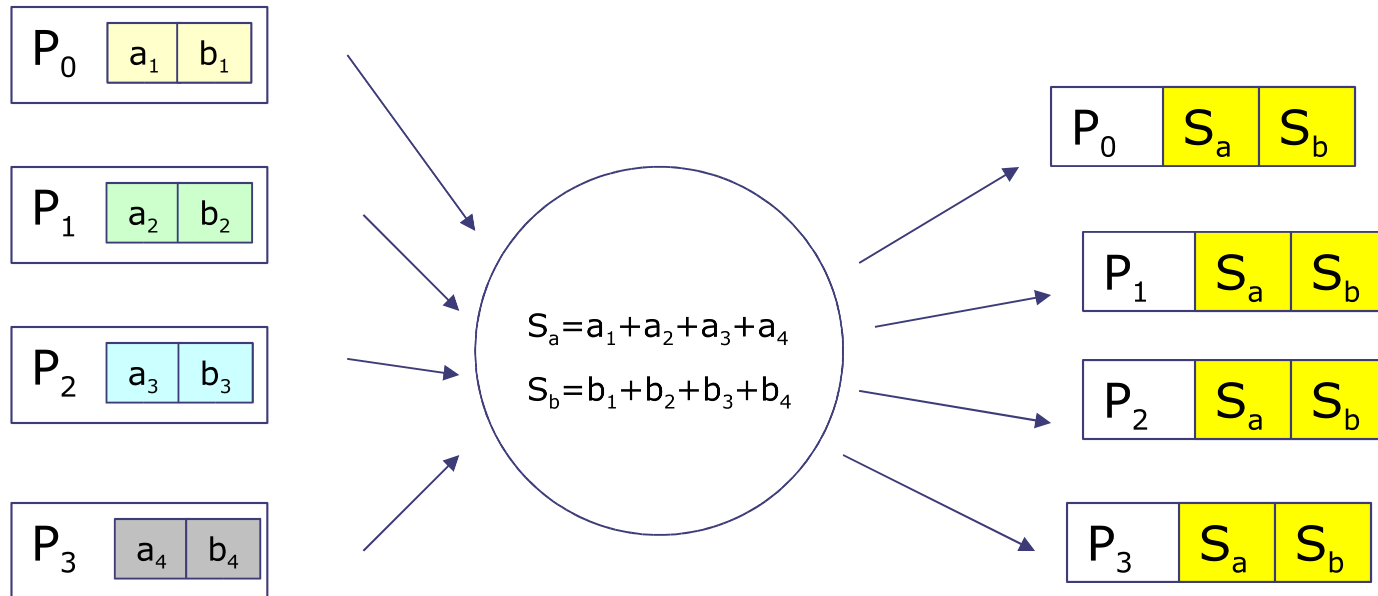


Reduction

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 occours)

```
MPI_ALLREDUCE( snd_buf, rcv_buf, count, type, op, comm, ierr)
```

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

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

MPI_Alltoall

Fortran:

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

(sender: sndbuf, sndcount, sndtype; receiver: rcvbuf, rcvcount, rcvtype)

sndbuf:

P_0	a_1	a_2	a_3	a_4
P_0	b_1	b_2	b_3	b_4
P_0	c_1	c_2	c_3	c_4
P_0	d_1	d_2	d_3	d_4

rcvbuf:

P_0	a_1	b_1	c_1	d_1
P_0	a_2	b_2	c_2	d_2
P_0	a_3	b_3	c_3	d_3
P_0	a_4	b_4	c_4	d_4

Very useful to implement data transposition

Reduce, cont.

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)
```

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: sndbuf, sndcount, sndtype — receiver: rcvbuf, rcvcount, rcvtype

```
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

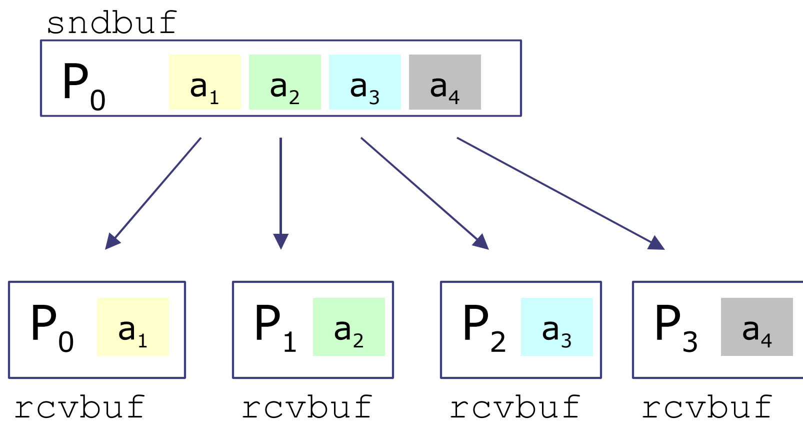
Fortran:

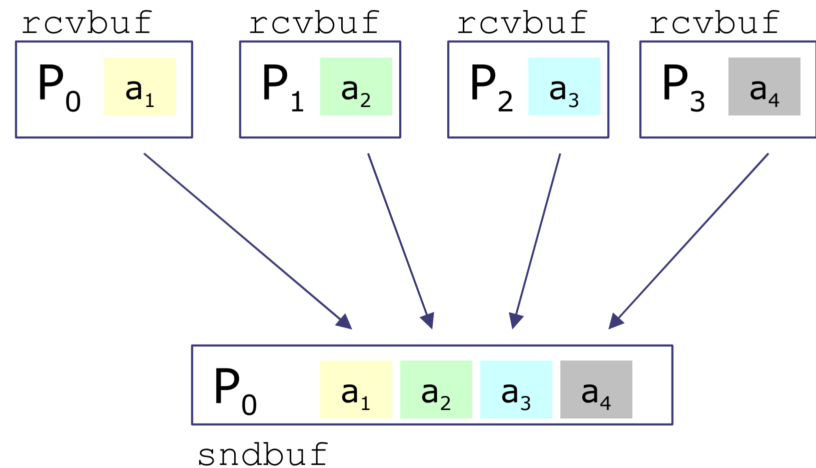
sender: sndbuf, sndcount, sndtype — receiver: rcvbuf, rcvcount, rcvtype

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

- 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 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

A very useful site...

- http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/main.htm
  - The examples from Using MPI, 2nd Edition are available here, along with Makefiles and autoconf-style configure scripts.