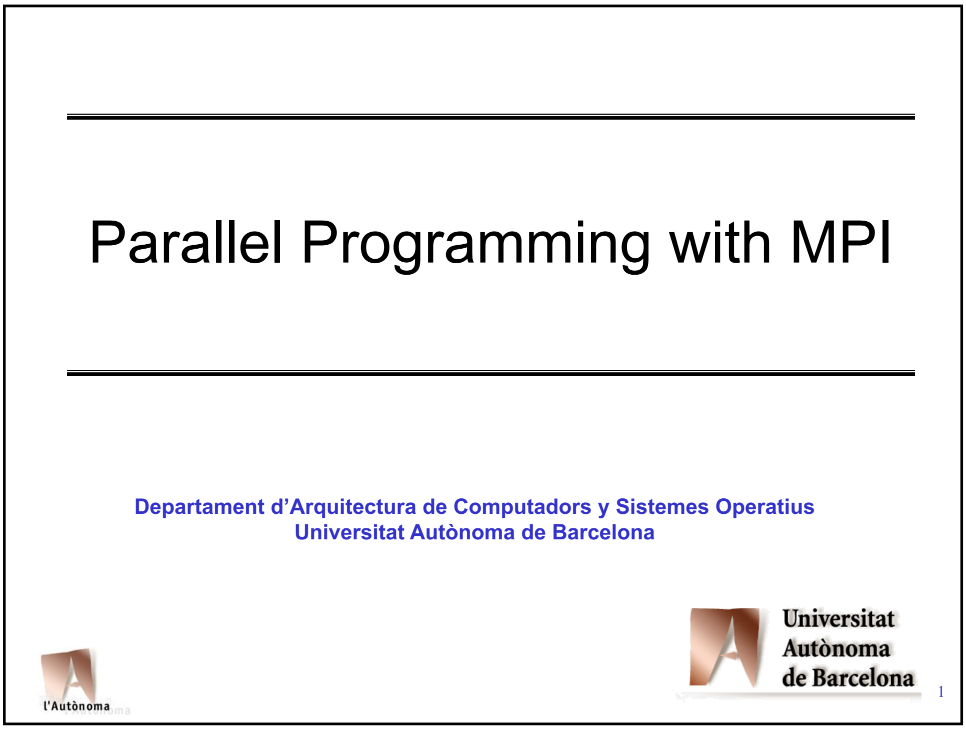


Parallel Programming with MPI

Departament d'Arquitectura de Computadors y Sistemes Operatius
Universitat Autònoma de Barcelona

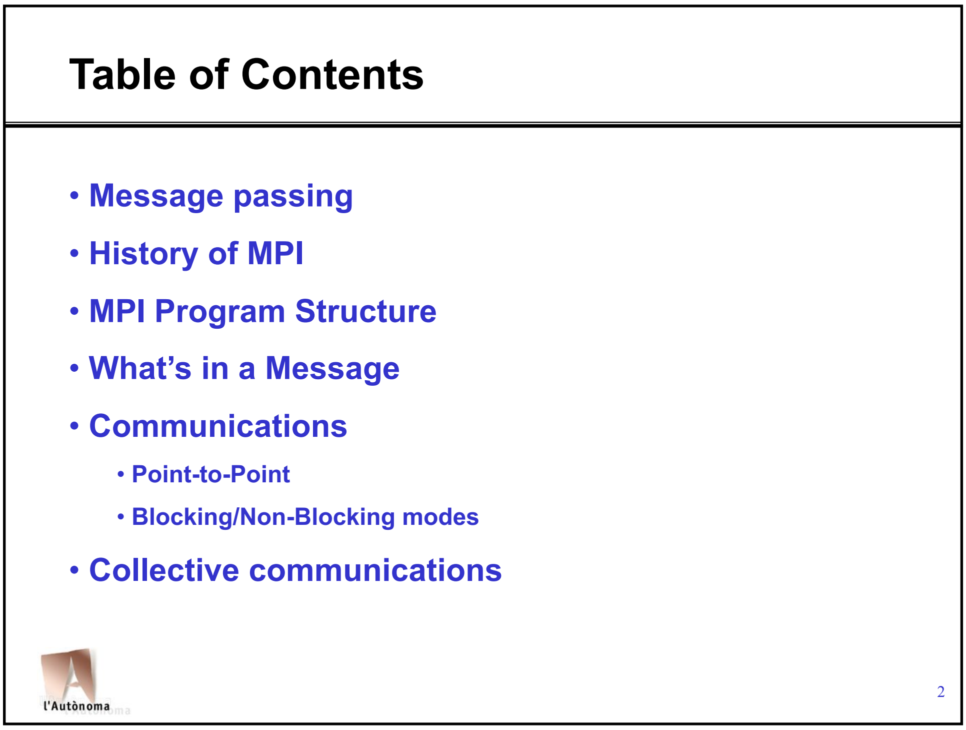


Table of Contents

- Message passing
- History of MPI
- MPI Program Structure
- What's in a Message
- Communications
  - Point-to-Point
  - Blocking/Non-Blocking modes
- Collective communications

Message passing

Among the approaches to writing parallel programs:

- use of a directives-based data-parallel language
  - A serial code is made parallel by adding directives (appear as comments in the serial code) that tell the compiler how to distribute data across the processors.
  - The details of how data distribution, computation, and communications are to be done are left to the compiler.
  - shared memory – all processes use the same memory
  - High Performance Fortran (HPF) or OpenMP

- explicit message passing via library calls from standard programming languages
  - it is left up to the programmer to explicitly divide data and work across the processors as well as manage the communications among them.

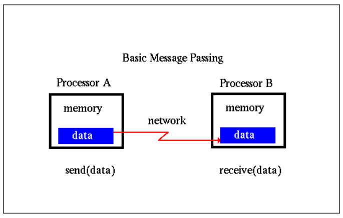
Message Passing: Model

- A process is (traditionally): a program counter and address space
  - may have multiple threads (program counters and stacks) sharing a single address space.

•A distributed memory parallel computation consists of a number of processes, each working on some local data.

- Each process has purely local variables, and there is no mechanism for any process to directly access the memory of another.

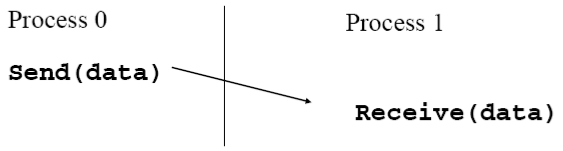
- Sharing of data between processes takes place by message passing, that is, by explicitly sending and receiving data between processes.



Message Passing: Model

- **Message passing model is for communication among processes, which have separate address spaces.**
  - What is the message? DATA
  - Allows for passing data between processes in a distributed memory environment

- **Interprocess communication consists of:**
  - Synchronization
  - Movement of data from one process's address space to another's.
  - Any change in the receiving process's memory is made with the receiver's explicit participation



Message Passing: Parallel Programming Issues

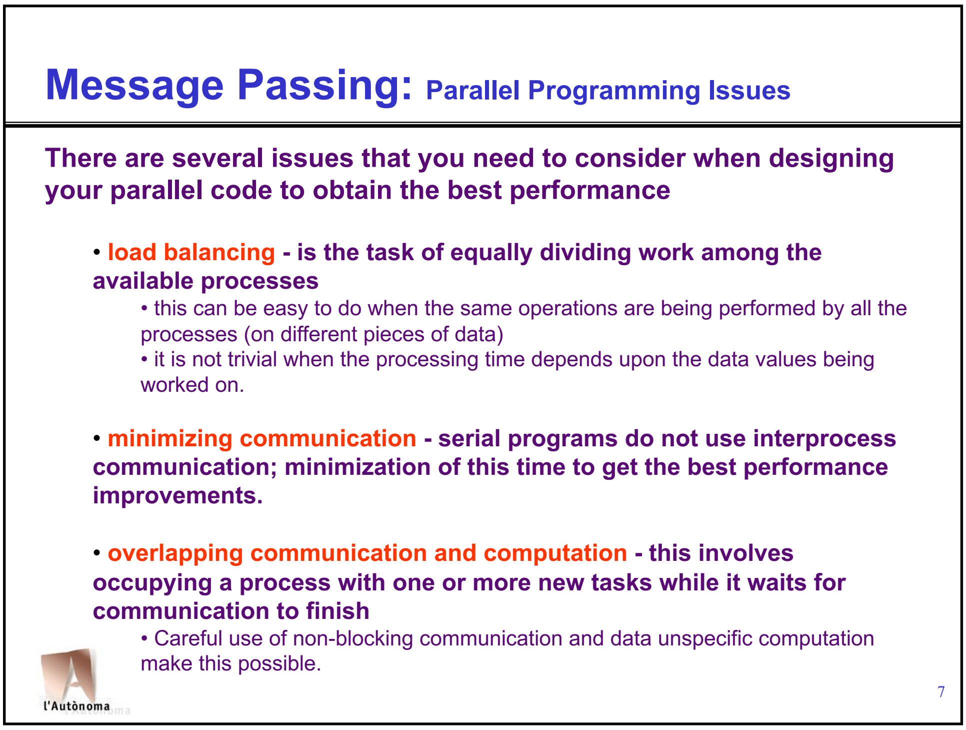
The main goal of writing a parallel program is to get better performance over the serial version.

Execution time is a major concern in parallel programming

1. **Computation time - time spent performing computations on the data**
   - Ideally, N processors working on a problem finish the job in 1/Nth the time of the serial job. This would be the case if all the processors' time was spent in computation.

2. **Idle time - time a process spends waiting for data from other processors**
   - During this time, the processors do no useful work.

3. **Communication time - time it takes to send and receive messages**
   - **The cost of communication can be measured in terms of:**
     - Latency is the time it takes to set up the envelope for communication
     - Bandwidth is the actual speed of transmission, or bits per unit time.



Message Passing: Parallel Programming Issues

There are several issues that you need to consider when designing your parallel code to obtain the best performance

- **load balancing - is the task of equally dividing work among the available processes**
  - this can be easy to do when the same operations are being performed by all the processes (on different pieces of data)
  - it is not trivial when the processing time depends upon the data values being worked on.

- **minimizing communication - serial programs do not use interprocess communication; minimization of this time to get the best performance improvements.**

- **overlapping communication and computation - this involves occupying a process with one or more new tasks while it waits for communication to finish**
  - Careful use of non-blocking communication and data unspecific computation make this possible.

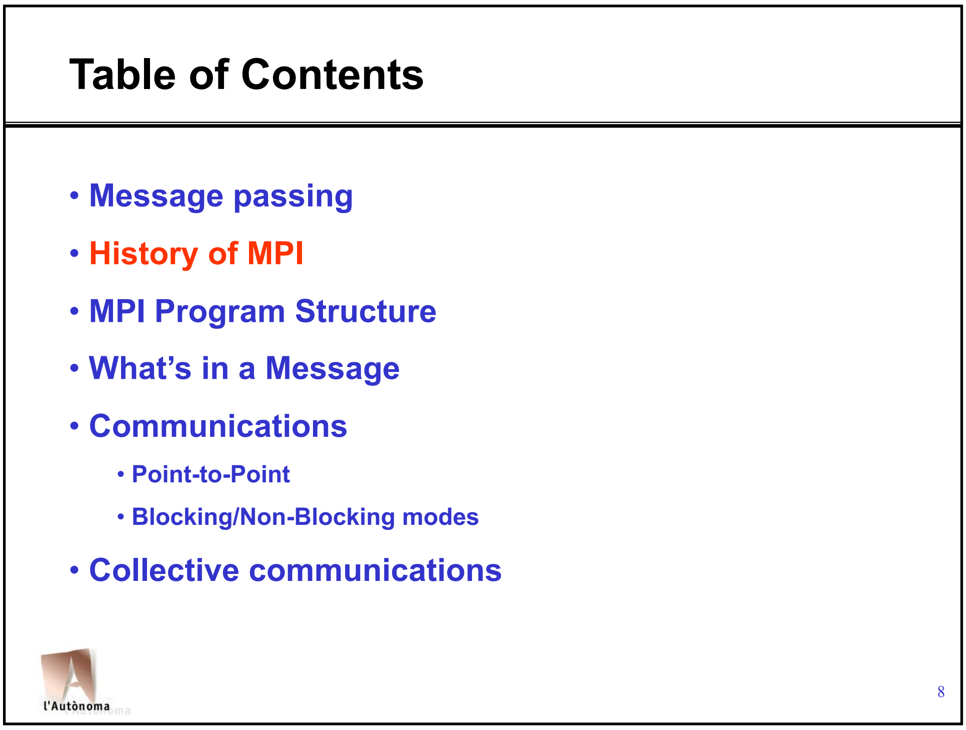


Table of Contents

- **Message passing**
- **History of MPI**
- **MPI Program Structure**
- **What’s in a Message**
- **Communications**
  - **Point-to-Point**
  - **Blocking/Non-Blocking modes**
- **Collective communications**

What is MPI?

- Sixty people from forty different organizations led by the MPI Forum

- MPI - "Message Passing Interface“

- MPI is intended as a standard implementation of the message passing model of parallel computing
  - specifies the names, calling sequences, and results of subroutines and functions to be called from Fortran 77 and C, respectively.
  - all implementations of MPI must conform to these rules to ensure portability.

- By itself, it is NOT a library - but rather the specification of what such a library should be

- Implementation - a library of functions (in C) or subroutines (in Fortran) that you insert into source code to perform data communication between processes

l'Autònoma

What is MPI?

- MPI’s prime goals are:
  - provide source-code portability and run on any platform that supports the MPI standard
  - allow for efficient implementation

•MPI-1 Standard (1994-2008)

•MPI-2 Standard (1998-2009)

- It provides for additional features not present in MPI-1, including tools for parallel I/O, C++ and Fortran 90 bindings, shared memory management and dynamic process management.

•MPI-3 Standard (2012-)

- Includes nonblocking collectives, new one-sided communications operations, and Fortran 2008 bindings

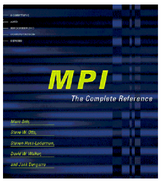
•MPI-4 Standard (2016-)

•Standards documents

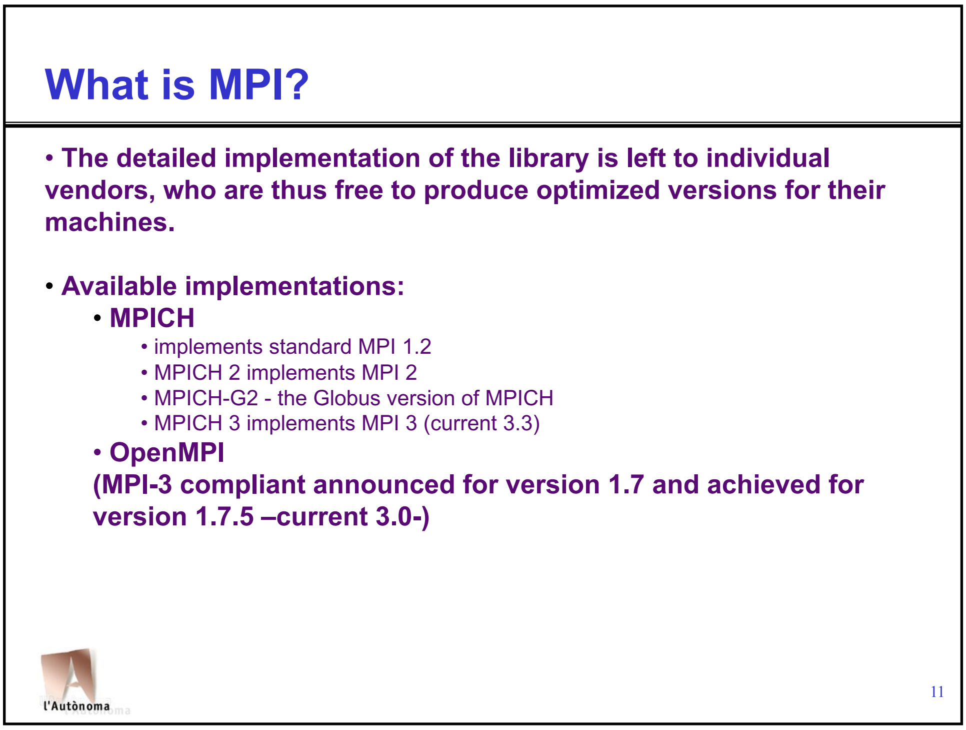
– http://www.mpi-forum.org/docs/docs.html (postscript versions)

– http://www.mcs.anl.gov/research/projects/mpi/

•Successor to PVM – Parallel Virtual Machine



l'Autònoma



What is MPI?

- The detailed implementation of the library is left to individual vendors, who are thus free to produce optimized versions for their machines.

- Available implementations:
  - MPICH
    - implements standard MPI 1.2
    - MPICH 2 implements MPI 2
    - MPICH-G2 - the Globus version of MPICH
    - MPICH 3 implements MPI 3 (current 3.3)
  - OpenMPI
    (MPI-3 compliant announced for version 1.7 and achieved for version 1.7.5 –current 3.0-)

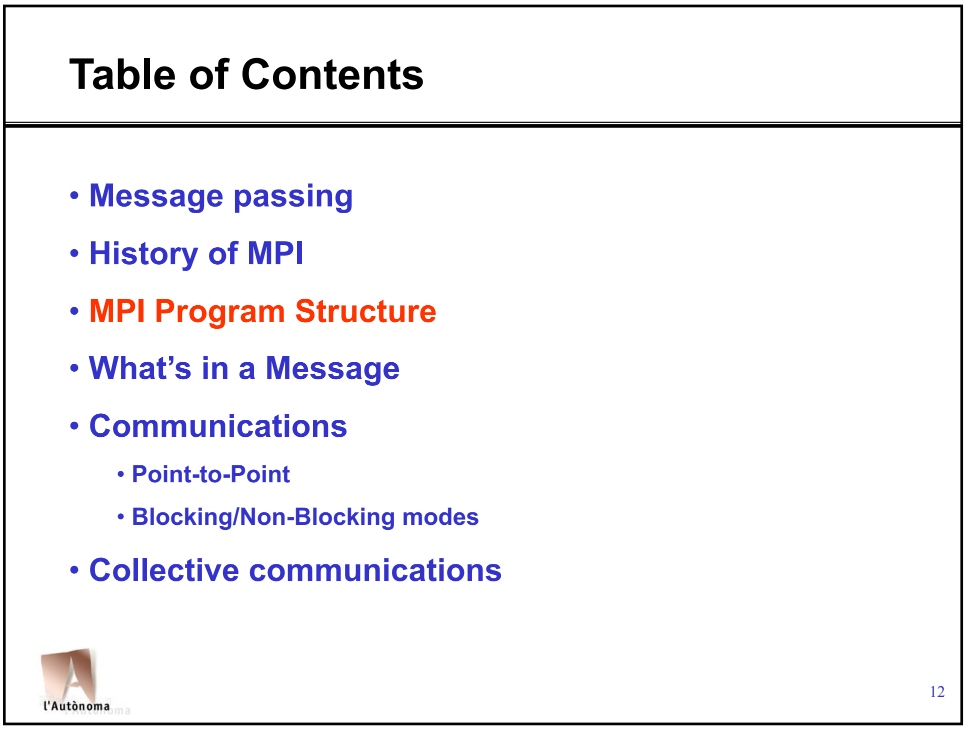


Table of Contents

- Message passing
- History of MPI
- MPI Program Structure
- What’s in a Message
- Communications
  - Point-to-Point
  - Blocking/Non-Blocking modes
- Collective communications

Compiling and running MPI programs

- The MPI standard does not specify how MPI programs are to be started. Implementations vary from machine to machine.
- When compiling an MPI program, it may be necessary to link against the MPI library

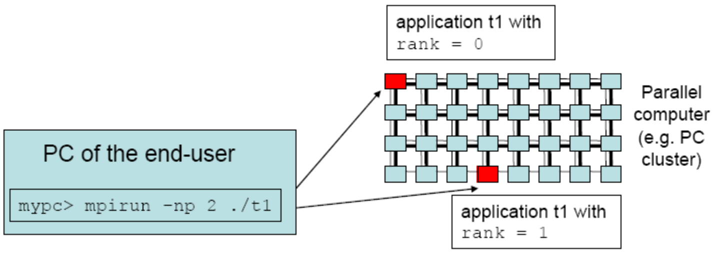
```
-lmpi
$ mpicc -o t1 t1.c -lmpi
```

- To run an MPI code, you commonly use a "wrapper" called mpirun, mpiexe or mpprun
- To run the executable `t1` on two processors:

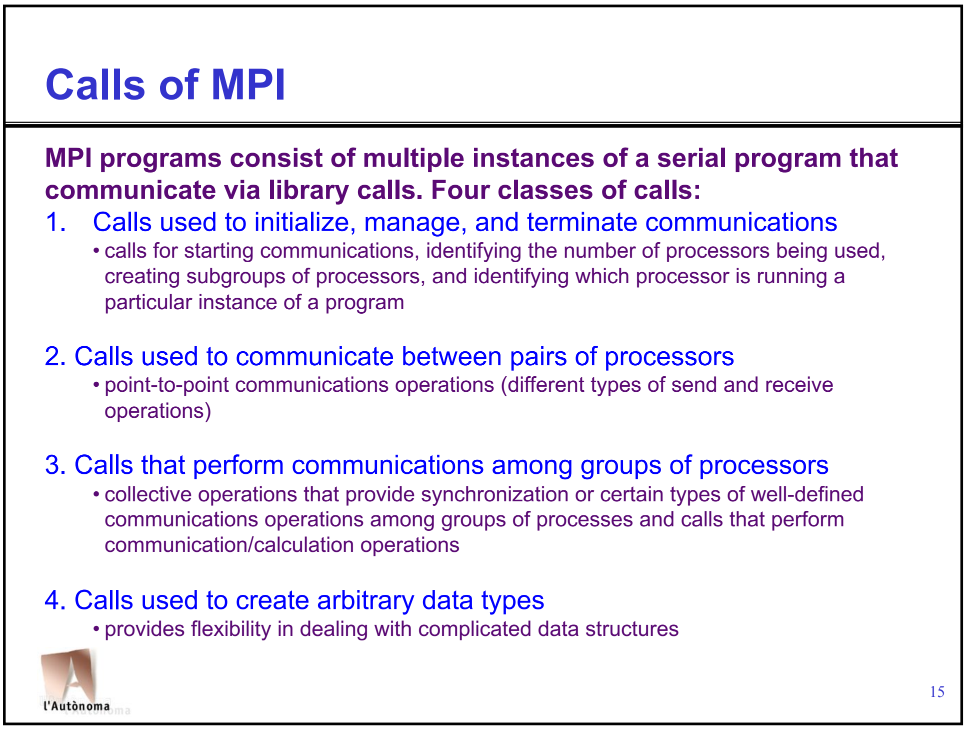
```
$ mpirun -np 2 t1
```

Compiling and running MPI programs

- `mpirun` starts the application `t1` two times
  - as specified with the –np
- on two currently available processors
- telling to one process that its rank is 0 and the other that its rank is 1



```
mypc> mpirun -np 2 ./t1
```



Calls of MPI

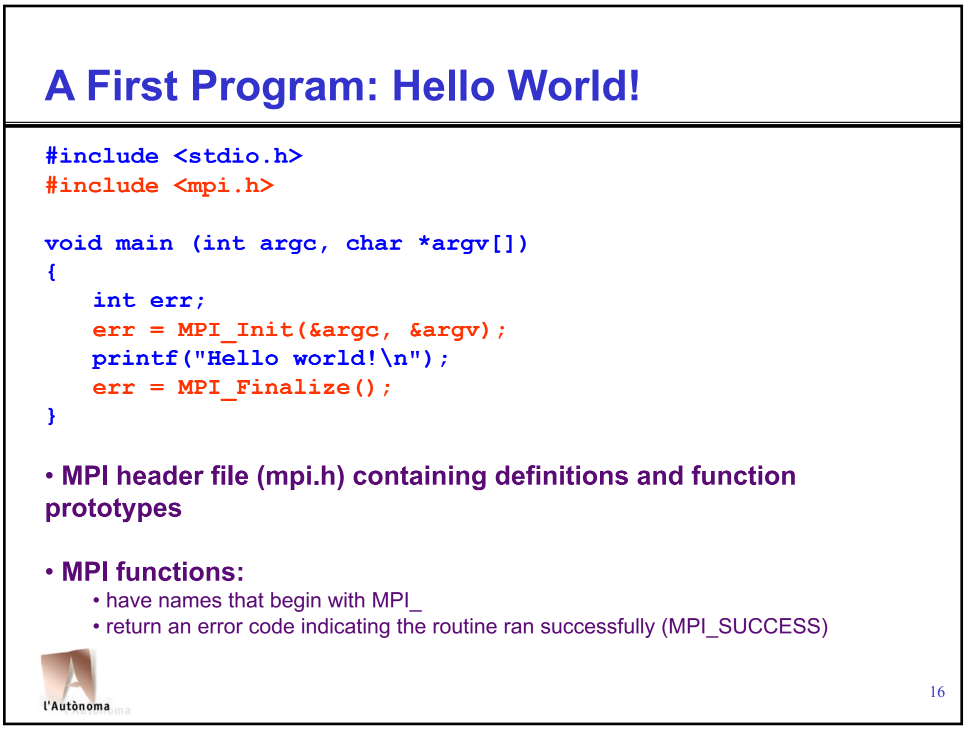
MPI programs consist of multiple instances of a serial program that communicate via library calls. Four classes of calls:

1. Calls used to initialize, manage, and terminate communications
   - calls for starting communications, identifying the number of processors being used, creating subgroups of processors, and identifying which processor is running a particular instance of a program

2. Calls used to communicate between pairs of processors
   - point-to-point communications operations (different types of send and receive operations)

3. Calls that perform communications among groups of processors
   - collective operations that provide synchronization or certain types of well-defined communications operations among groups of processes and calls that perform communication/calculation operations

4. Calls used to create arbitrary data types
   - provides flexibility in dealing with complicated data structures



A First Program: Hello World!

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

void main (int argc, char *argv[])
{
   int err;
   err = MPI_Init(&argc, &argv);
   printf("Hello world!\n");
   err = MPI_Finalize();
}
```

- **MPI header file (mpi.h) containing definitions and function prototypes**

- **MPI functions:**
  - have names that begin with MPI_
  - return an error code indicating the routine ran successfully (MPI_SUCCESS)

A First Program: Hello World!

Each processor executes a copy of the entire process

The output of the program executed with 4 processors is:

```
Hello world!
Hello world!
Hello world!
Hello world!
```

However, different processors can be made to do different things using program branches, e.g.

```
if (I am processor 1)
...do something...
if (I am processor 2)
...do something else...
```

Better Hello World

```
#include <mpi.h>
#include <stdio.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( “Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Better Hello World

- **Running this code on four processors will produce a result like:**

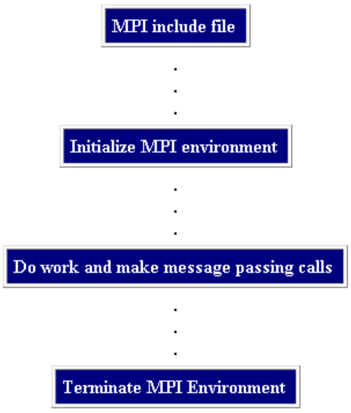
```
Hello world from process 2 of 4
Hello world from process 1 of 4
Hello world from process 3 of 4
Hello world from process 0 of 4
```

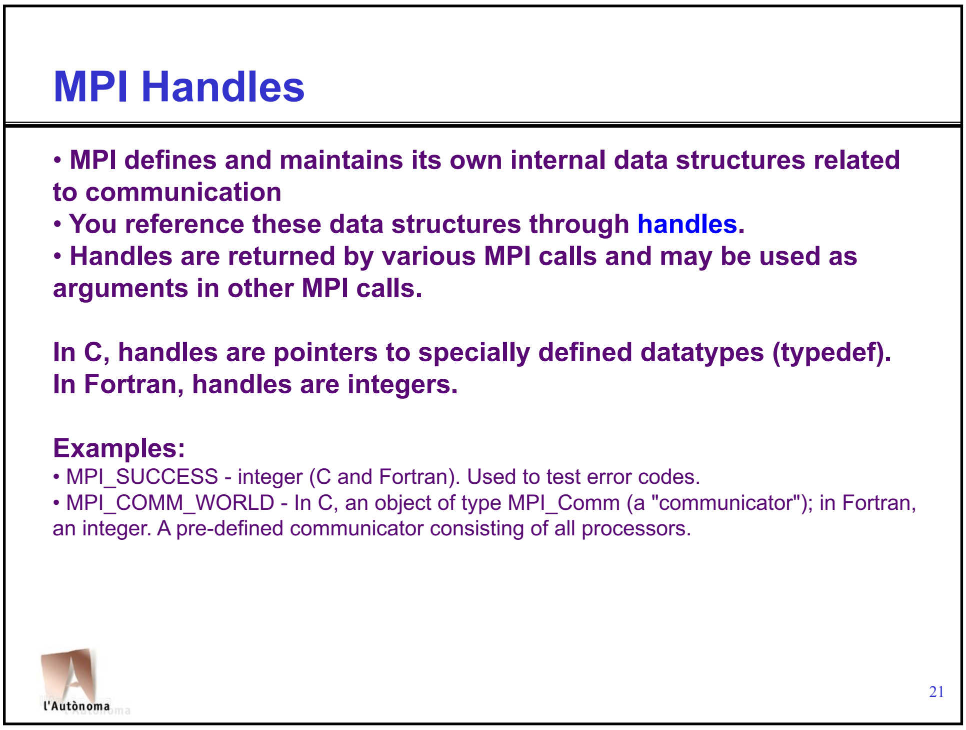
- **Each processor executes the same code, including probing for its rank and size and printing the string.**

- **The order of the printed lines is essentially random!**
  - There is no synchronization of operations on different processors.
  - Each time the code is run, the order of the output lines may change.

MPI Program Structure

- **Handles**
- **MPI Communicator**
- **Header files**
- **MPI function format**
- **Initializing MPI**
- **Communicator Size**
- **Process Rank**
- **Exiting MPI**


MPI include file
Initialize MPI environment
Do work and make message passing calls
Terminate MPI Environment



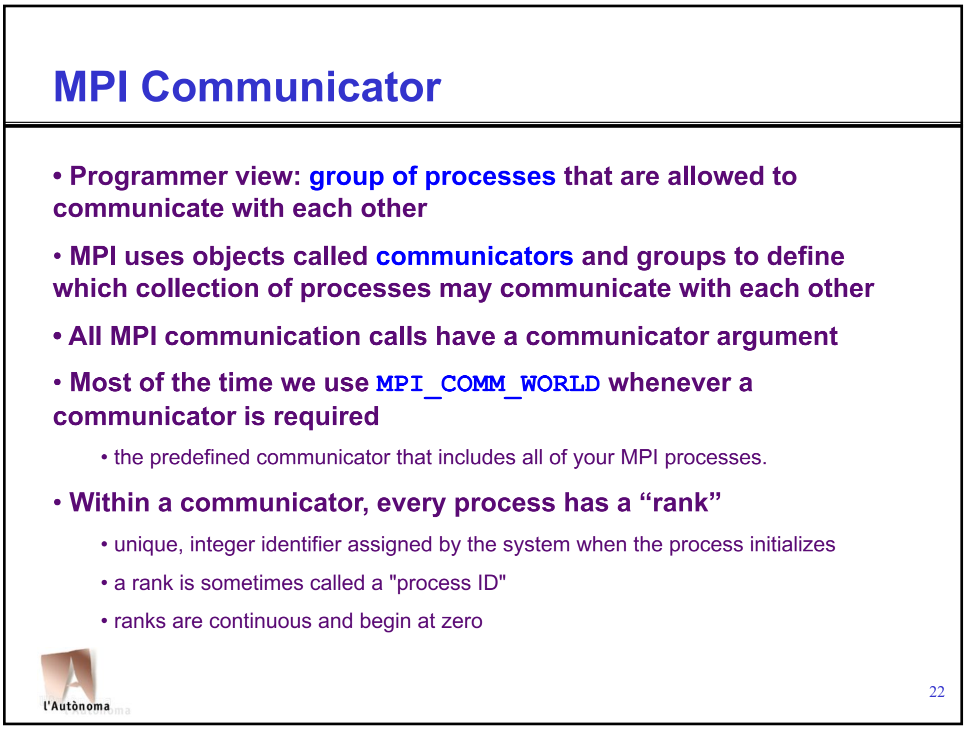
MPI Handles

- **MPI defines and maintains its own internal data structures related to communication**
- **You reference these data structures through handles.**
- **Handles are returned by various MPI calls and may be used as arguments in other MPI calls.**

In C, handles are pointers to specially defined datatypes (typedef). In Fortran, handles are integers.

Examples:

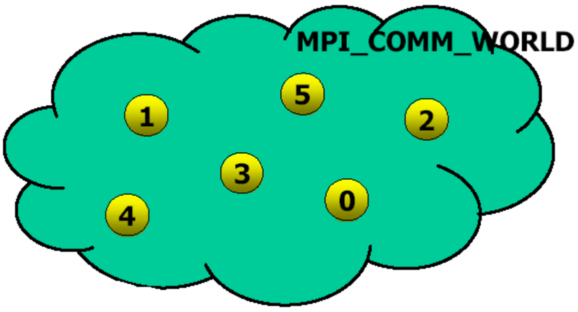
- MPI_SUCCESS - integer (C and Fortran). Used to test error codes.
- MPI_COMM_WORLD - In C, an object of type MPI_Comm (a "communicator"); in Fortran, an integer. A pre-defined communicator consisting of all processors.



MPI Communicator

- **Programmer view: group of processes that are allowed to communicate with each other**
- **MPI uses objects called communicators and groups to define which collection of processes may communicate with each other**
- **All MPI communication calls have a communicator argument**
- **Most of the time we use `MPI_COMM_WORLD` whenever a communicator is required**
  - the predefined communicator that includes all of your MPI processes.
- **Within a communicator, every process has a “rank”**
  - unique, integer identifier assigned by the system when the process initializes
  - a rank is sometimes called a "process ID"
  - ranks are continuous and begin at zero

MPI Communicator



Header Files

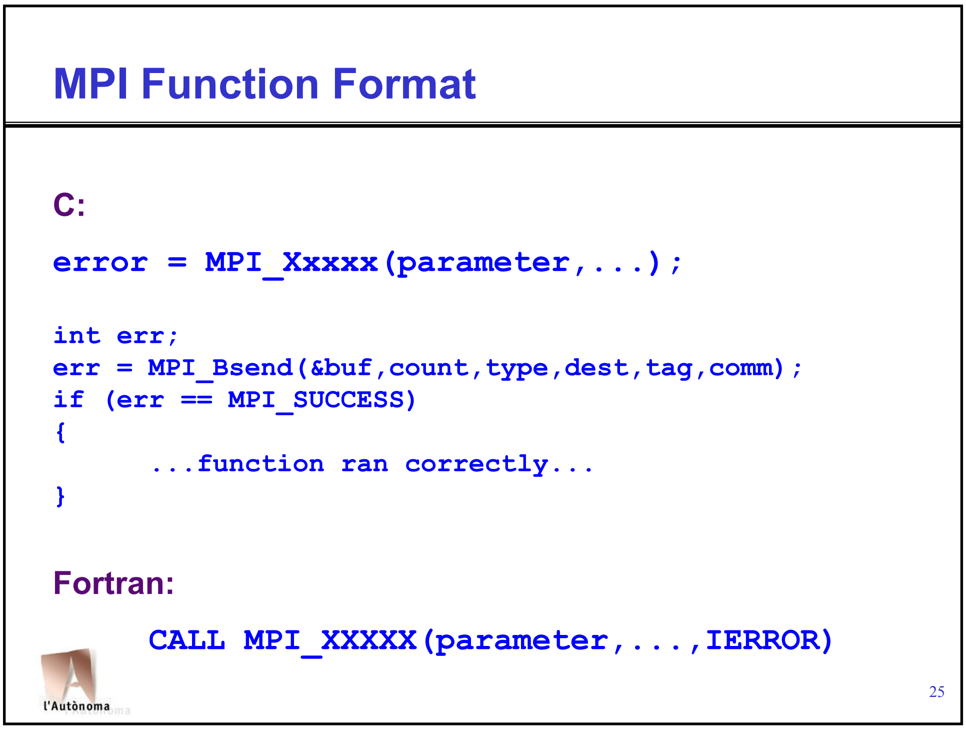
MPI constants and handles are defined here

C:

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

Fortran:

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



MPI Function Format

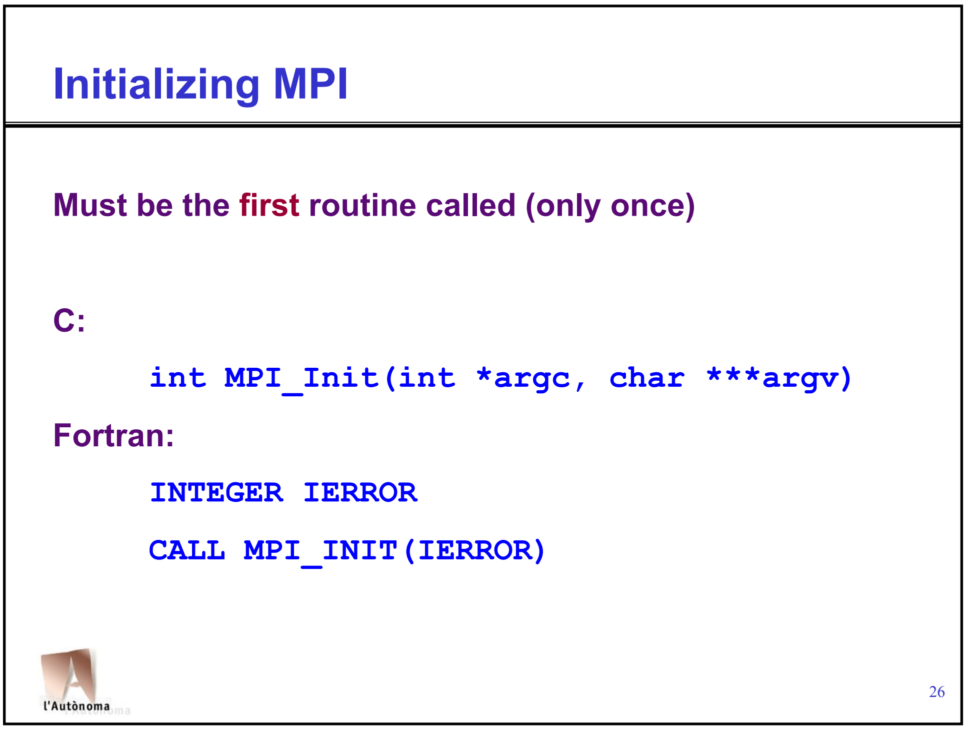
C:

```
error = MPI_Xxxxx(parameter,...);

int err;
err = MPI_Bsend(&buf,count,type,dest,tag,comm);
if (err == MPI_SUCCESS)
{
      ...function ran correctly...
}
```

Fortran:

```
      CALL MPI_XXXXX(parameter,...,IERROR)
```



Initializing MPI

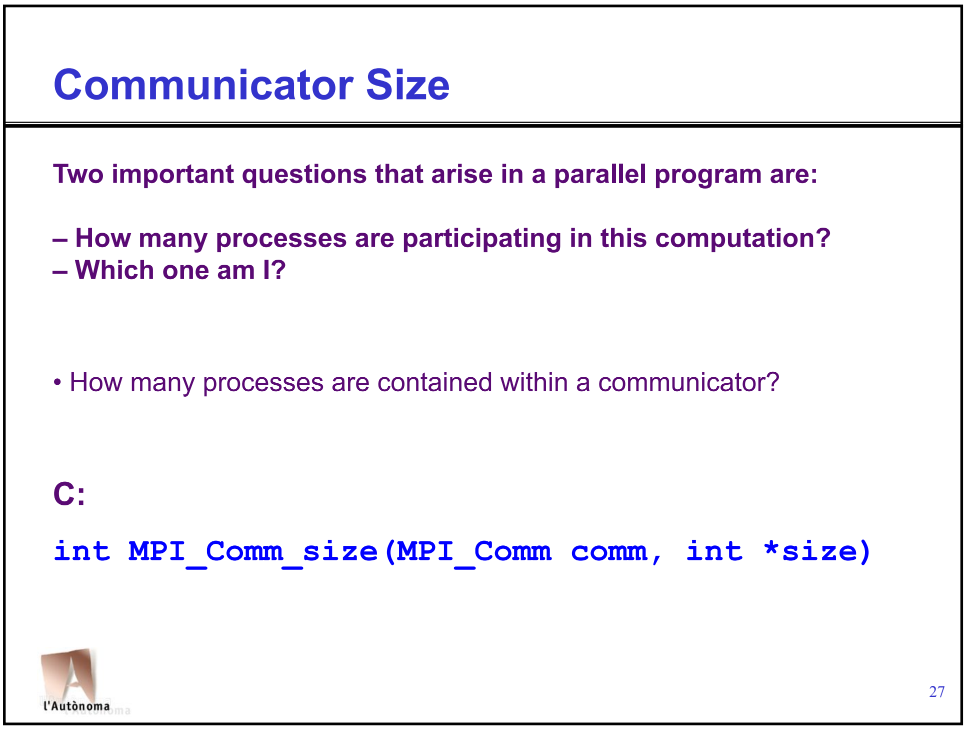
Must be the first routine called (only once)

C:

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

Fortran:

```
      INTEGER IERROR
      CALL MPI_INIT(IERROR)
```



Communicator Size

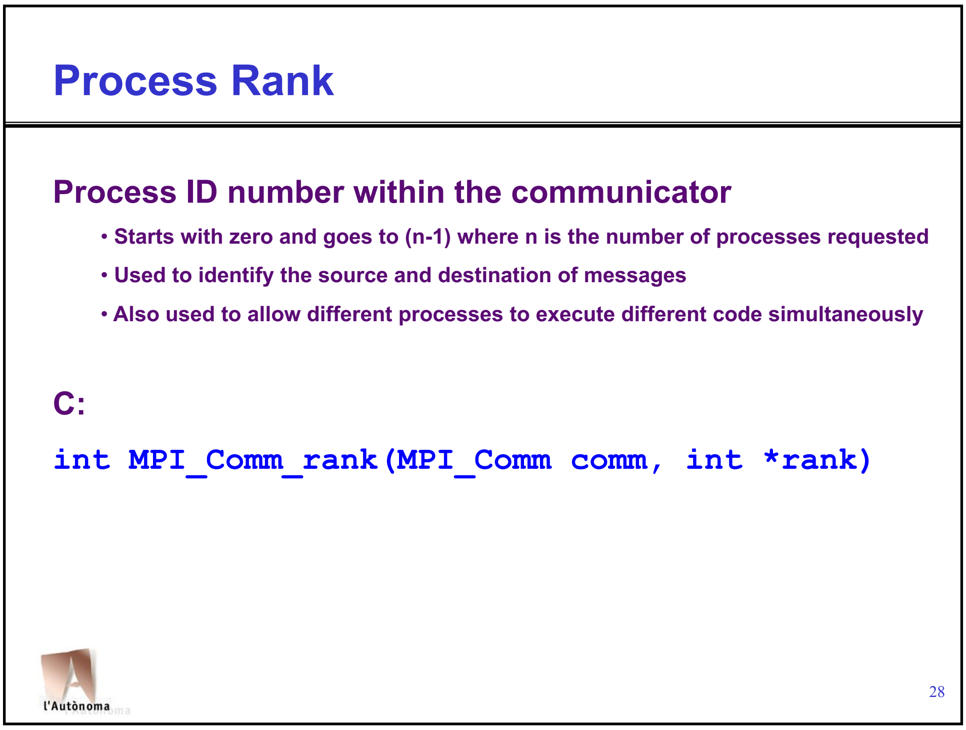
Two important questions that arise in a parallel program are:

– How many processes are participating in this computation?
– Which one am I?

- How many processes are contained within a communicator?

C:

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



Process Rank

Process ID number within the communicator

- **Starts with zero and goes to (n-1) where n is the number of processes requested**
- **Used to identify the source and destination of messages**
- **Also used to allow different processes to execute different code simultaneously**

C:

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

Exiting MPI

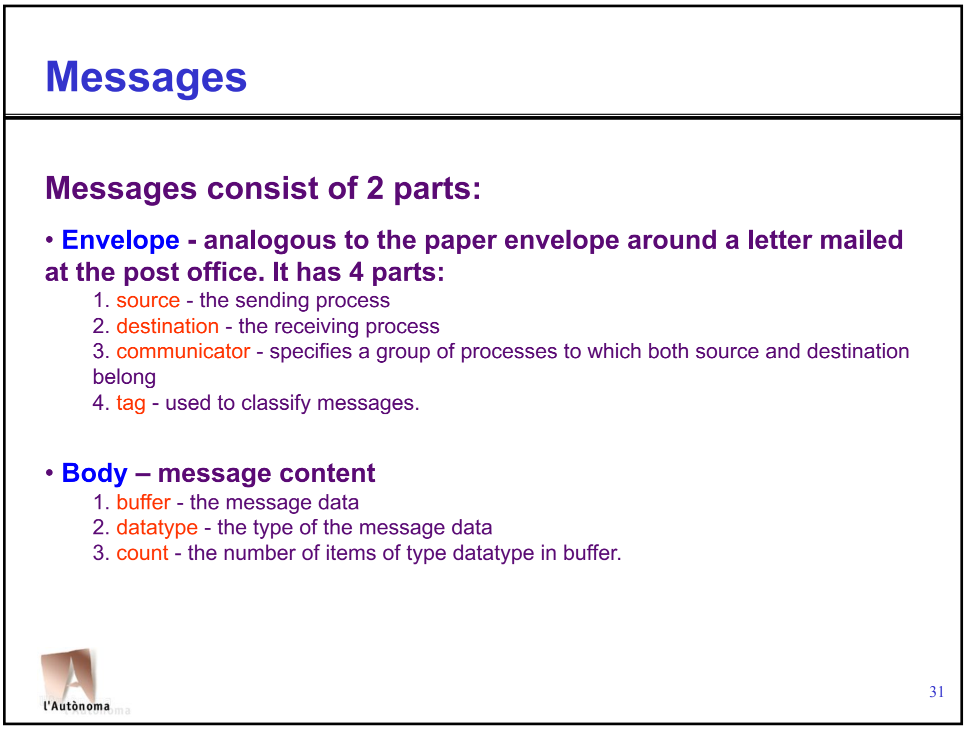
Must be called as last function involving communication by "all" processes

C:

```
int MPI_Finalize()
```

Table of Contents

- **Message passing**
- **History of MPI**
- **MPI Program Structure**
- **What's in a Message**
- **Communications**
  - **Point-to-Point**
  - **Blocking/Non-Blocking modes**
- **Collective communications**

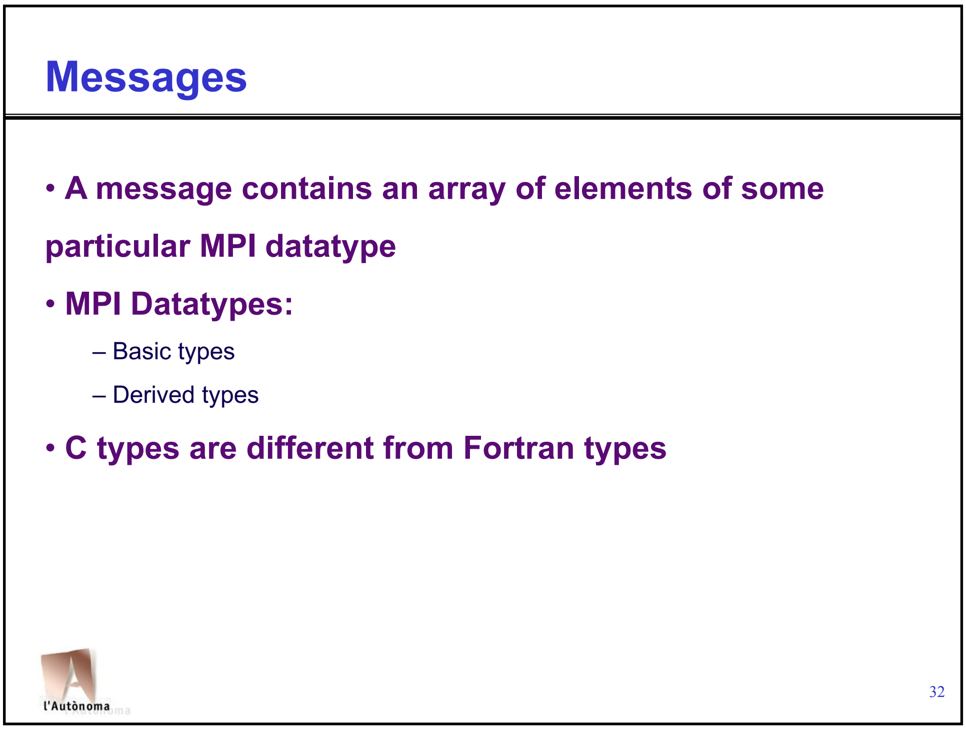


Messages

Messages consist of 2 parts:

- **Envelope - analogous to the paper envelope around a letter mailed at the post office. It has 4 parts:**
  1. source - the sending process
  2. destination - the receiving process
  3. communicator - specifies a group of processes to which both source and destination belong
  4. tag - used to classify messages.

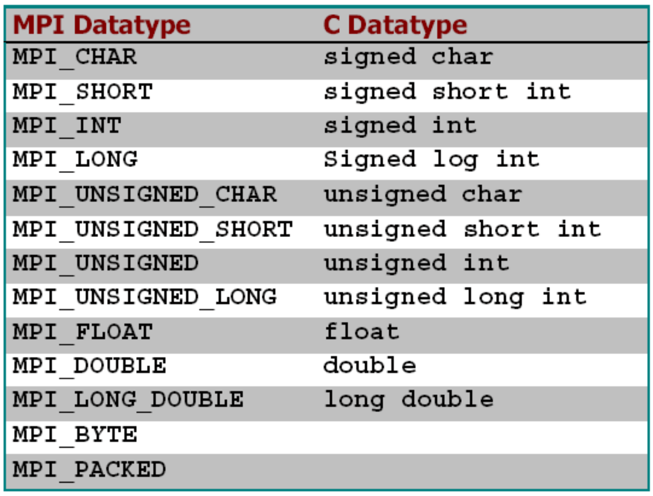
- **Body – message content**
  1. buffer - the message data
  2. datatype - the type of the message data
  3. count - the number of items of type datatype in buffer.



Messages

- **A message contains an array of elements of some particular MPI datatype**
- **MPI Datatypes:**
  - – Basic types
  - – Derived types
- **C types are different from Fortran types**

MPI Basic Datatypes - C



MPI Datatype	C Datatype
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	

Rules and Rationale

- Programmer declares variables to have “normal” C/Fortran type, but uses matching MPI datatypes as arguments in MPI routines

- General rule: MPI datatype specified in a *receive* must match the MPI datatype specified in the *send*

Example

```
char mess[MAXSIZE]; /* message (other types possible) */
…
if (myRank != 0)
{/* all processes send to root        */
    /* create message */
    sprintf(message, "Hello from %d", myRank);
    dest  = 0;                   /* destination is root  */
    count = strlen(mess) + 1;  /* include '\0' in message */

    MPI_Send(mess, count, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
else
{/* root (0) process receives and prints messages */
    /* from each processor in rank order          */
    for(source = 1; source < numProc; source++)
    {
      MPI_Recv(mess,MAXSIZE,MPI_CHAR,source,tag,MPICOMM_WORLD, &stat);
      printf("%s\n", mess);
    }
}
```

l'Autònoma

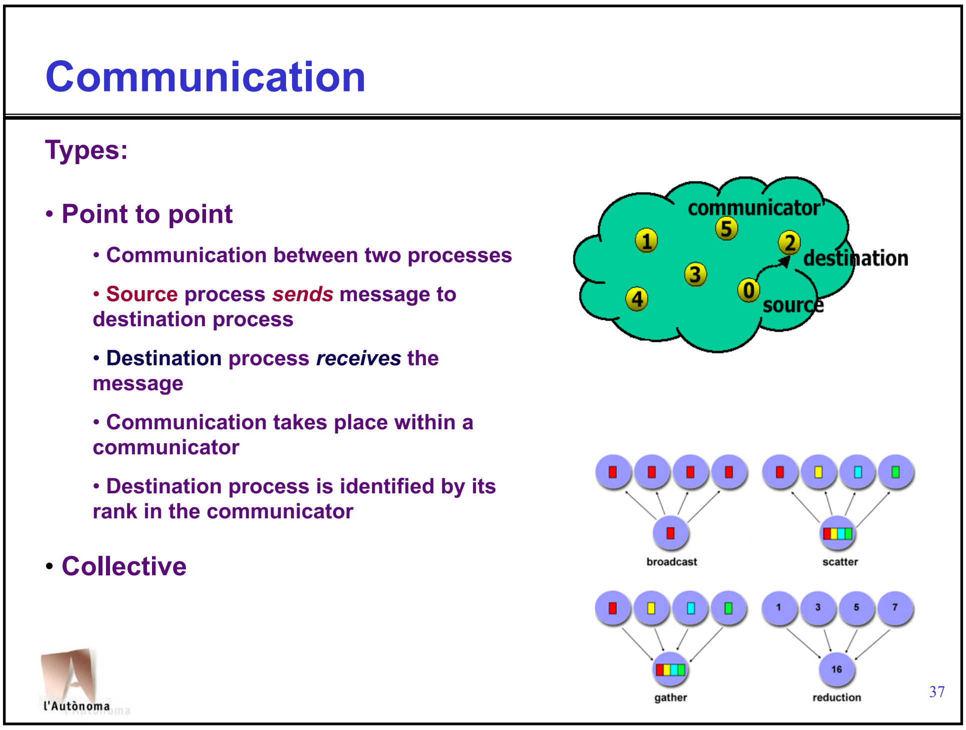
35

Table of Contents

- Message passing
- History of MPI
- MPI Program Structure
- What's in a Message
- Communications
  - Point-to-Point
  - Blocking/Non-Blocking modes
- Collective communications

l'Autònoma

36

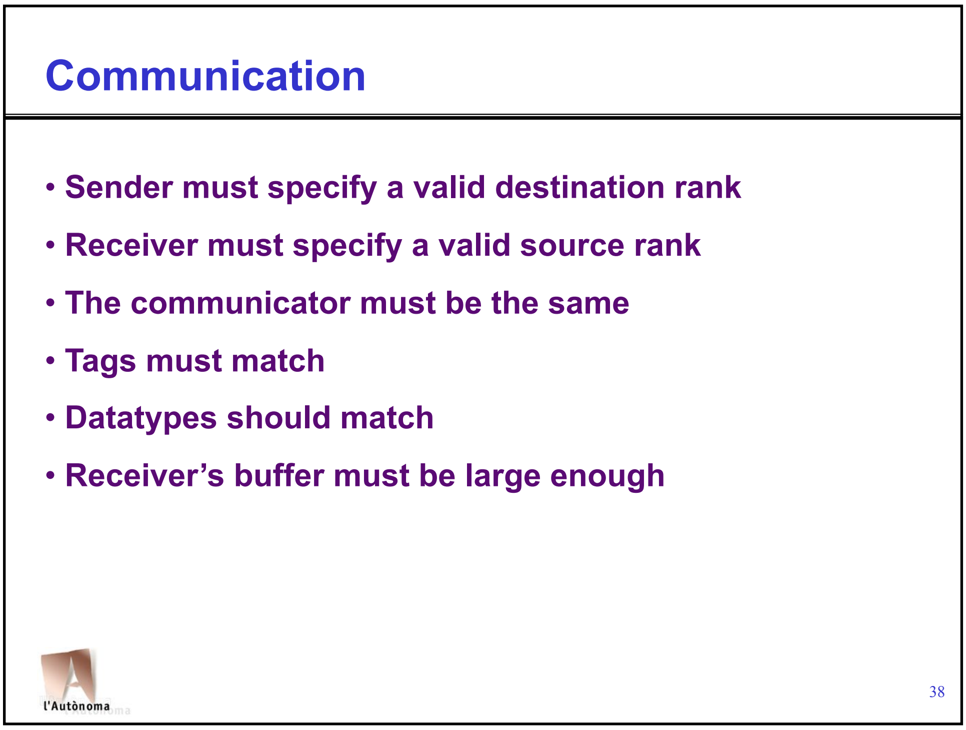


Communication

Types:

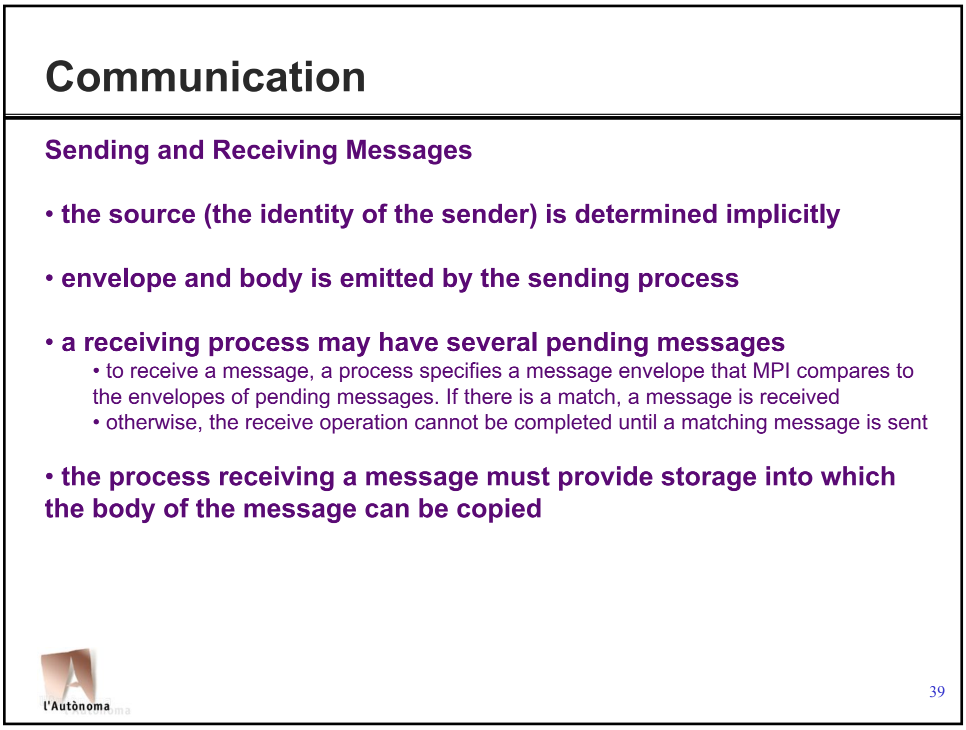
- Point to point
  - Communication between two processes
  - Source process *sends* message to destination process
  - Destination process *receives* the message
  - Communication takes place within a communicator
  - Destination process is identified by its rank in the communicator
- Collective

communicator
1
5
2
destination
3
0
4
source
broadcast
scatter
1
3
5
7
gather
16
reduction



Communication

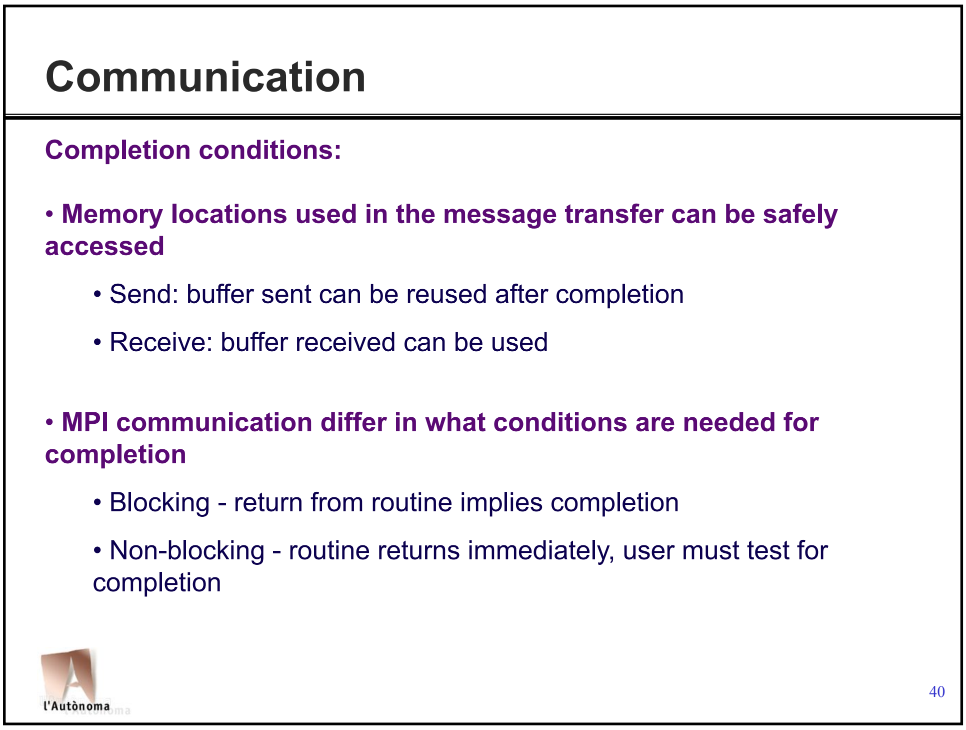
- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Datatypes should match
- Receiver's buffer must be large enough



Communication

Sending and Receiving Messages

- **the source (the identity of the sender) is determined implicitly**
- **envelope and body is emitted by the sending process**
- **a receiving process may have several pending messages**
  - to receive a message, a process specifies a message envelope that MPI compares to the envelopes of pending messages. If there is a match, a message is received
  - otherwise, the receive operation cannot be completed until a matching message is sent
- **the process receiving a message must provide storage into which the body of the message can be copied**



Communication

Completion conditions:

- **Memory locations used in the message transfer can be safely accessed**
  - Send: buffer sent can be reused after completion
  - Receive: buffer received can be used
- **MPI communication differ in what conditions are needed for completion**
  - Blocking - return from routine implies completion
  - Non-blocking - routine returns immediately, user must test for completion

Sending a Message

Blocking send:

C:

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

`buf` starting *address* of the data to be sent

`count` number of elements to be sent

`datatype` MPI datatype of each element

`dest` rank of destination process

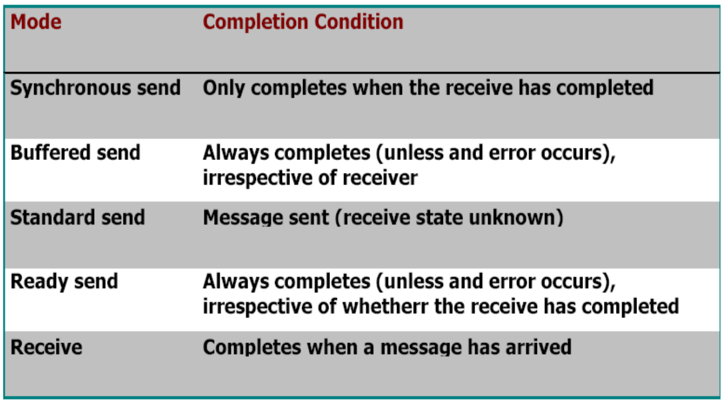
`tag` message marker (set by user)

`comm` MPI communicator of processors involved

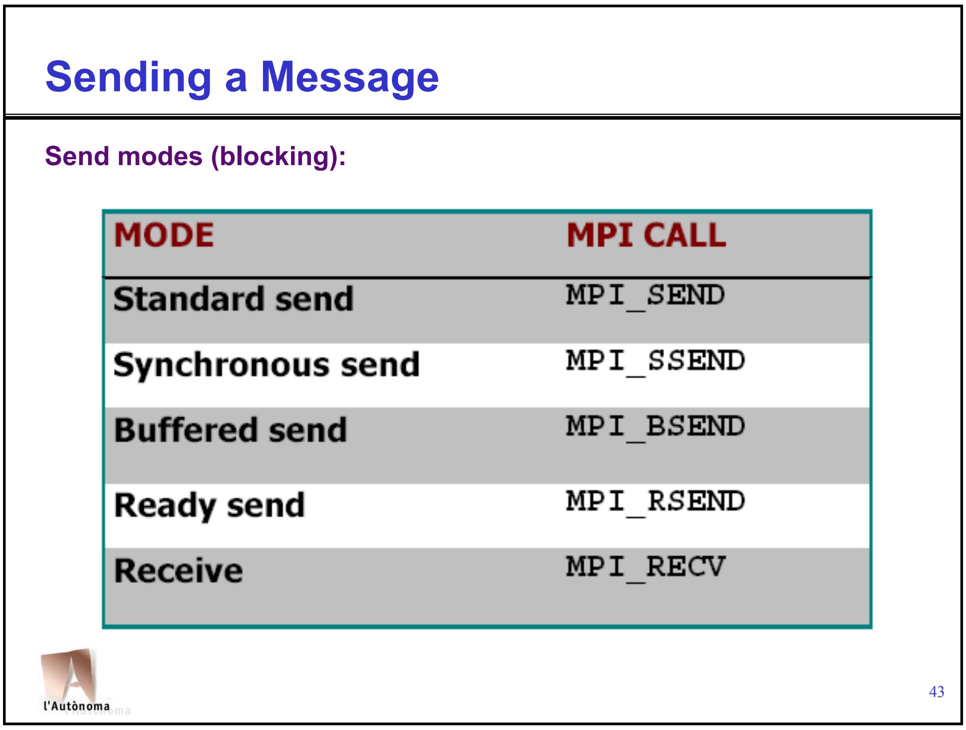
```
MPI_Send(data,500,MPI_FLOAT,6,33,MPI_COMM_WORLD)
```

Sending a Message

Send modes (blocking):



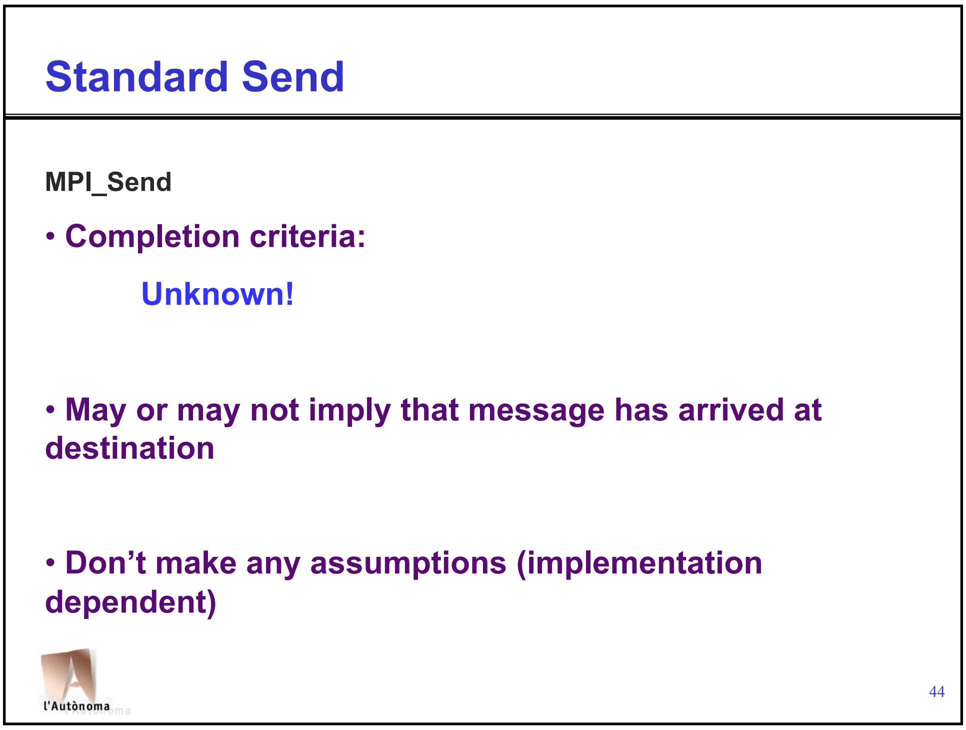
Mode	Completion Condition
Synchronous send	Only completes when the receive has completed
Buffered send	Always completes (unless and error occurs), irrespective of receiver
Standard send	Message sent (receive state unknown)
Ready send	Always completes (unless and error occurs), irrespective of whetherr the receive has completed
Receive	Completes when a message has arrived



Sending a Message

Send modes (blocking):

MODE	MPI CALL
Standard send	MPI_SEND
Synchronous send	MPI_SSEND
Buffered send	MPI_BSEND
Ready send	MPI_RSEND
Receive	MPI_RECV



Standard Send

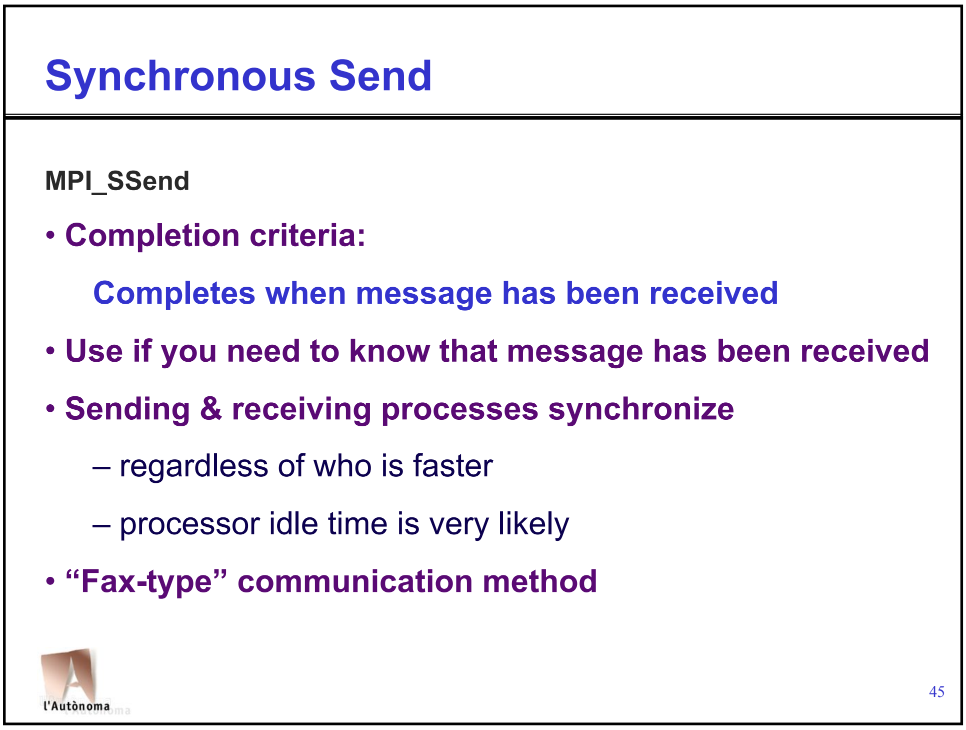
MPI_Send

- Completion criteria:

  Unknown!

- May or may not imply that message has arrived at destination

- Don’t make any assumptions (implementation dependent)



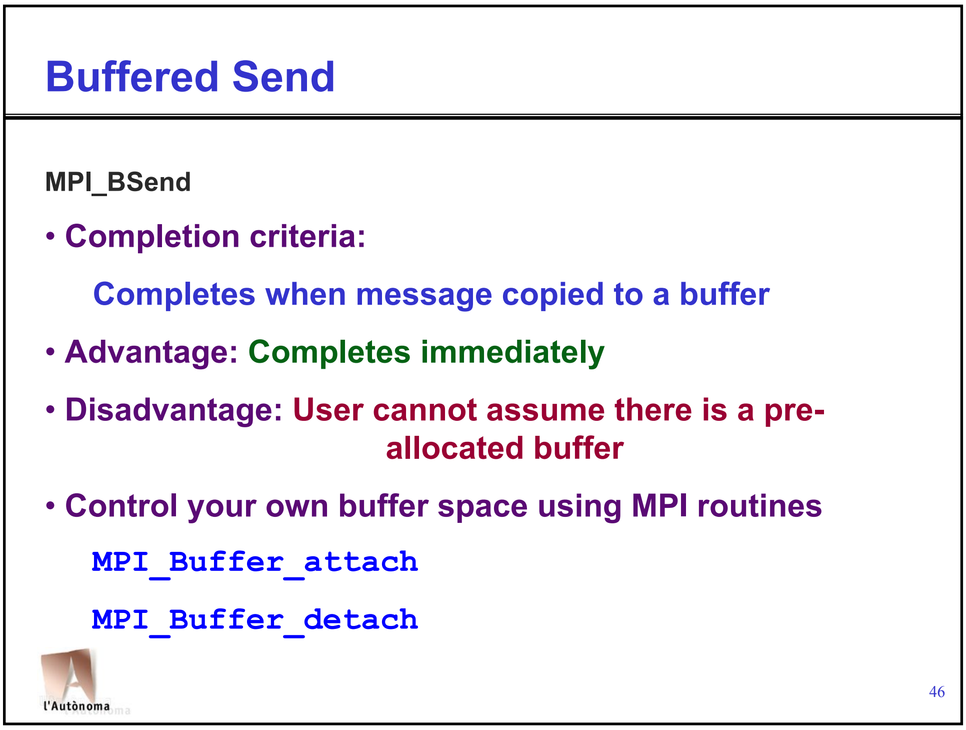
Synchronous Send

MPI_SSend

- **Completion criteria:**

  Completes when message has been received

- **Use if you need to know that message has been received**
- **Sending & receiving processes synchronize**
  - regardless of who is faster
  - processor idle time is very likely
- **“Fax-type” communication method**



Buffered Send

MPI_BSend

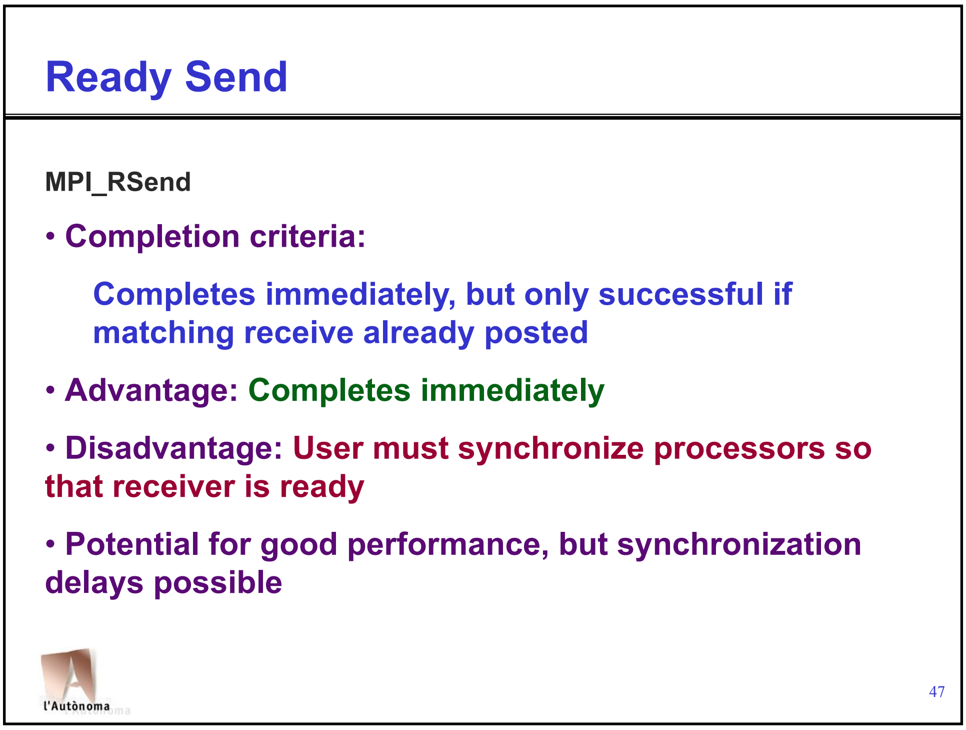
- **Completion criteria:**

  Completes when message copied to a buffer

- **Advantage: Completes immediately**
- **Disadvantage: User cannot assume there is a pre-allocated buffer**
- **Control your own buffer space using MPI routines**

  `MPI_Buffer_attach`

  `MPI_Buffer_detach`



Ready Send

MPI_RSend

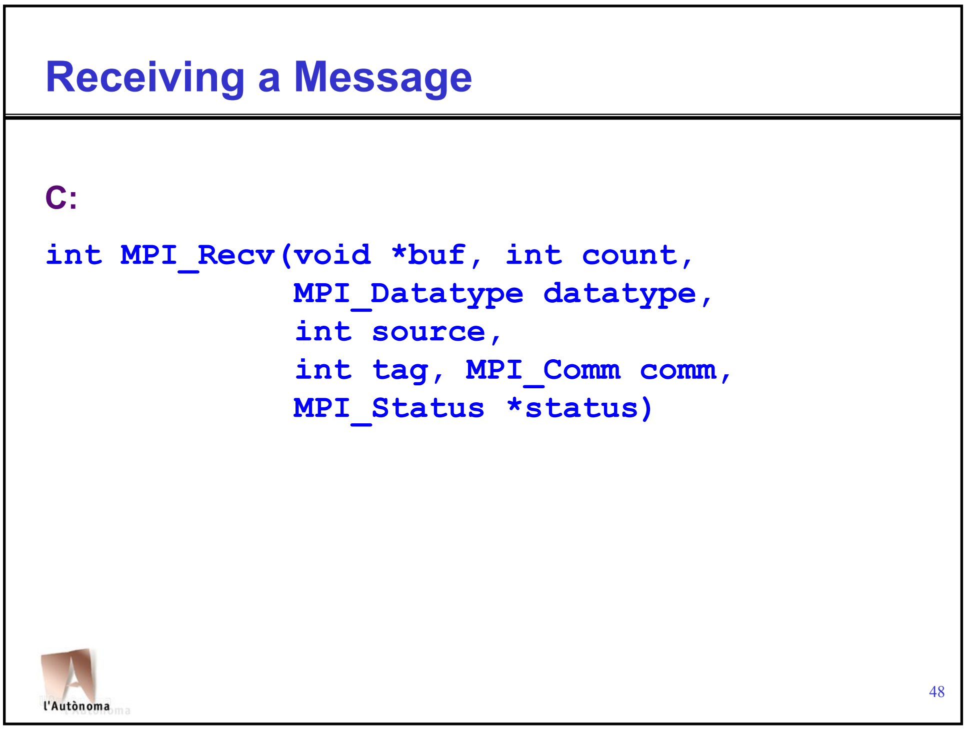
- Completion criteria:

  Completes immediately, but only successful if matching receive already posted

- Advantage: Completes immediately

- Disadvantage: User must synchronize processors so that receiver is ready

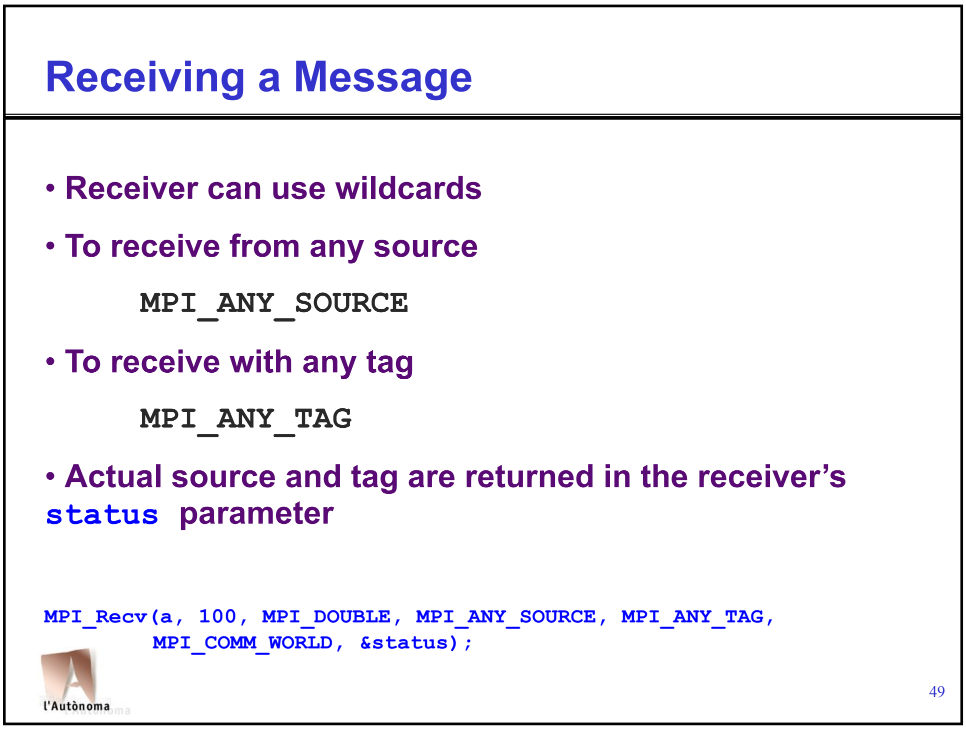
- Potential for good performance, but synchronization delays possible



Receiving a Message

C:

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



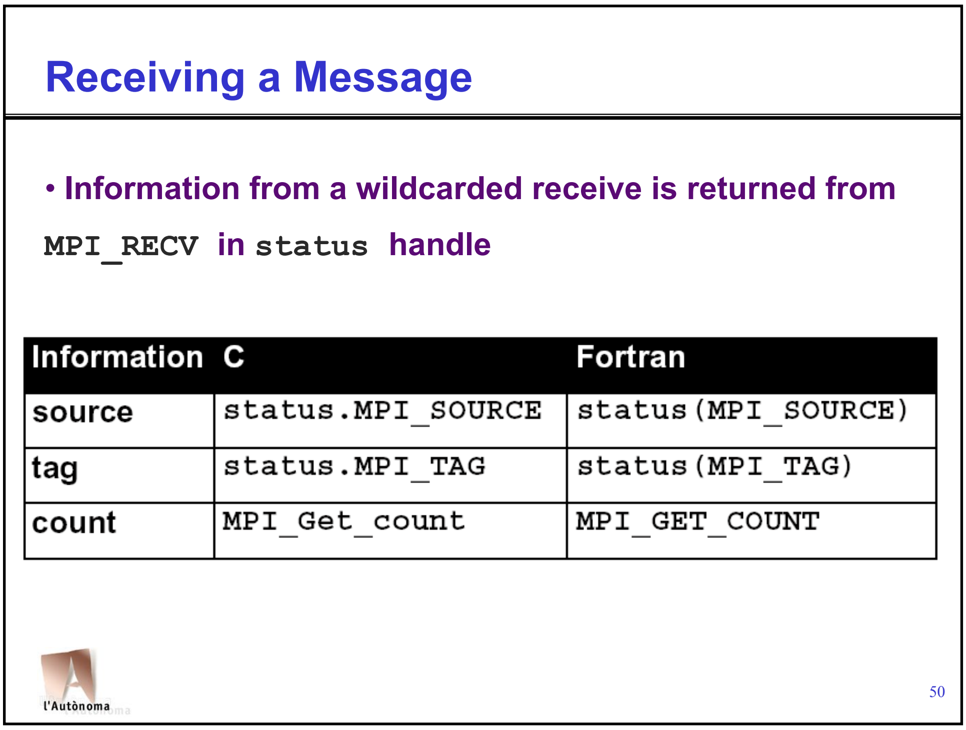
Receiving a Message

- Receiver can use wildcards
- To receive from any source

  `MPI_ANY_SOURCE`
- To receive with any tag

  `MPI_ANY_TAG`
- Actual source and tag are returned in the receiver’s `status` parameter

```
MPI_Recv(a, 100, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status);
```



Receiving a Message

- Information from a wildcarded receive is returned from `MPI_RECV` in `status` handle

Information	C	Fortran
source	`status.MPI_SOURCE`	`status(MPI_SOURCE)`
tag	`status.MPI_TAG`	`status(MPI_TAG)`
count	`MPI_Get_count`	`MPI_GET_COUNT`

Receiving a Message

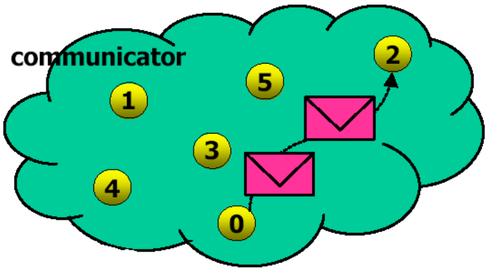
Received Message Count

- Message received may not fill receive buffer
- `count` is number of elements actually received

C:

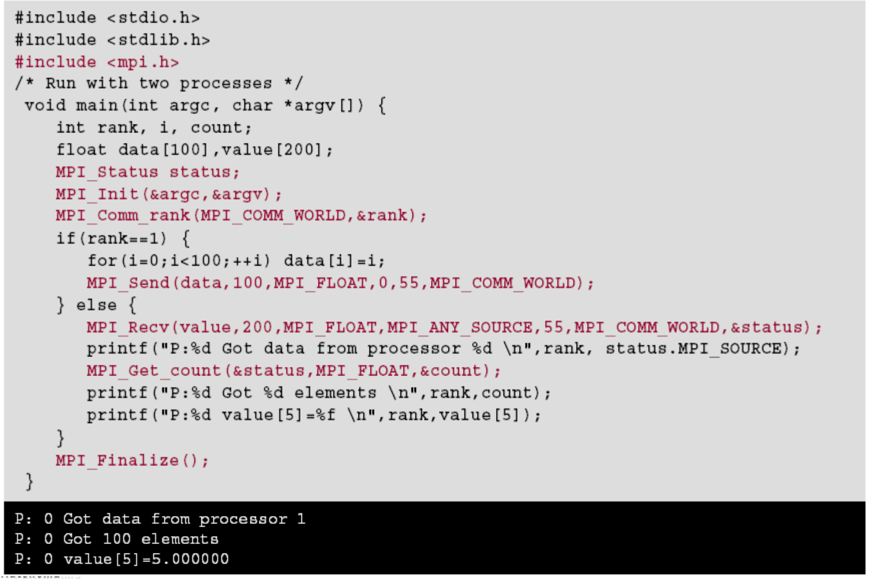
```
   int MPI_Get_count (MPI_Status *status,
                     MPI_Datatype datatype,
                     int *count)
```

Receiving a Message



- Messages do not overtake each other
- Example: Process 0 sends two messages
  Process 2 posts two receives that match either message
  Order preserved

Sample Program #1 - C

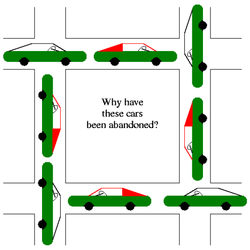


```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/* Run with two processes */
 void main(int argc, char *argv[]) {
    int rank, i, count;
    float data[100],value[200];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==1) {
       for(i=0;i<100;++i) data[i]=i;
       MPI_Send(data,100,MPI_FLOAT,0,55,MPI_COMM_WORLD);
    } else {
       MPI_Recv(value,200,MPI_FLOAT,MPI_ANY_SOURCE,55,MPI_COMM_WORLD,&status);
       printf("P:%d Got data from processor %d \n",rank, status.MPI_SOURCE);
       MPI_Get_count(&status,MPI_FLOAT,&count);
       printf("P:%d Got %d elements \n",rank,count);
       printf("P:%d value[5]=%f \n",rank,value[5]);
    }
    MPI_Finalize();
 }
```

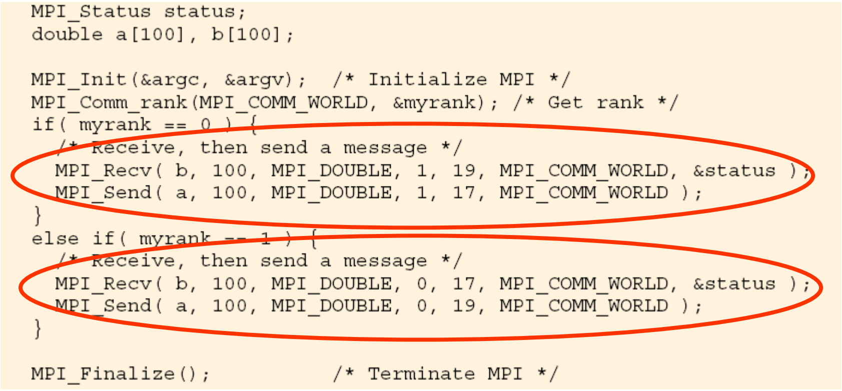
```
P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=5.000000
```

Deadlocks

- Deadlock occurs when 2 (or more) processes are blocked and each is waiting for the other to make progress.
- Neither process makes progress because each depends on the other to make progress first.
- Avoiding deadlock requires careful organization of the communication in a program.



Deadlocks



```
MPI_Status status;
double a[100], b[100];

MPI_Init(&argc, &argv);  /* Initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
if( myrank == 0 ) {
  /* Receive, then send a message */
  MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
  MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
}
else if( myrank == 1 ) {
  /* Receive, then send a message */
  MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
  MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
}

MPI_Finalize();          /* Terminate MPI */
```

Timers

- Time is measured in seconds
- Action execution time is measured by consulting the timer before and after

C:

```
double MPI_Wtime(void);
```

```
double t1, t2, ttotal;
t1 = MPI_Wtime(void);
... do action
t2 = MPI_Wtime(void);
ttotal = t2-t1;
```

Table of Contents

- Message passing
- History of MPI
- MPI Program Structure
- What's in a Message
- Communications
  - Point-to-Point
  - Blocking/Non-Blocking modes
- Collective communications

Non-Blocking Communications

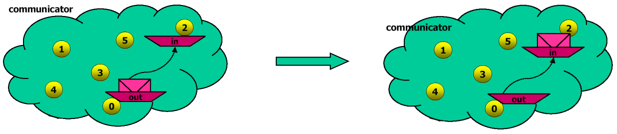
Separate communication into three phases:

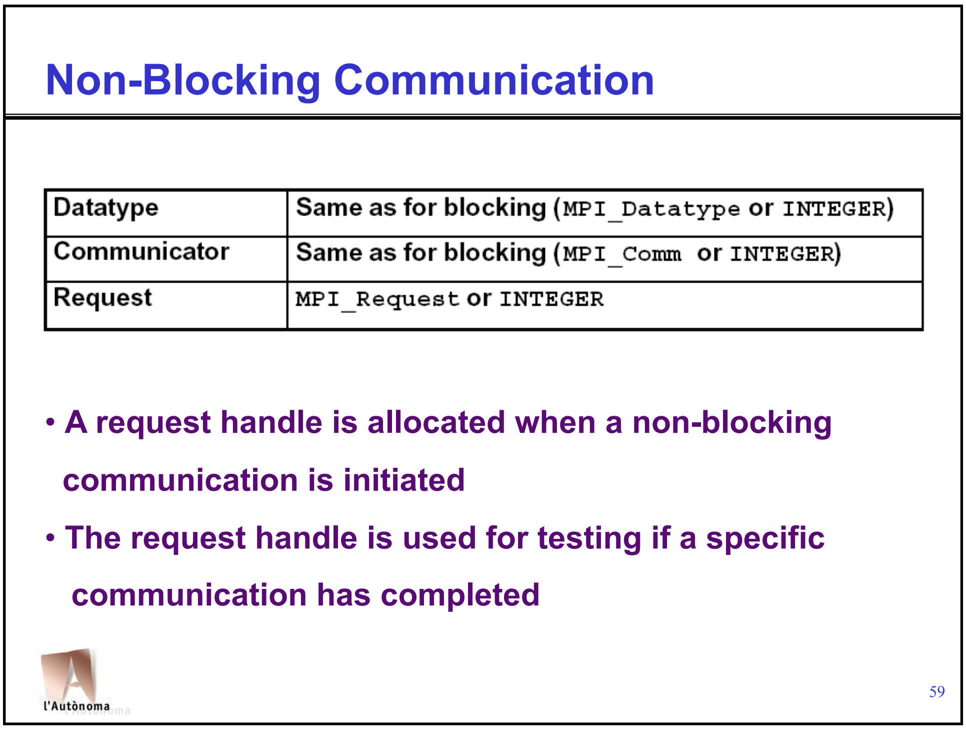
1. Initiate non-blocking communication ("post" a send or receive)

2. Do some other work not involving the data in transfer

- Overlap calculation and communication
- Latency hiding

3. Wait for non-blocking communication to complete

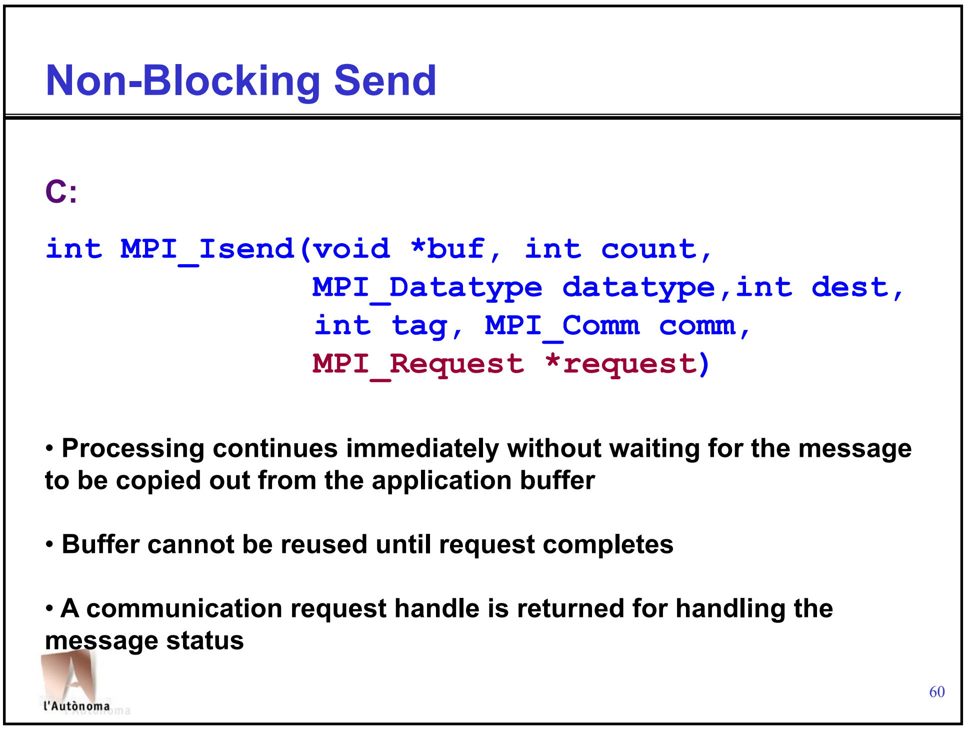




Non-Blocking Communication

Datatype	Same as for blocking (`MPI_Datatype` or `INTEGER`)
Communicator	Same as for blocking (`MPI_Comm` or `INTEGER`)
Request	`MPI_Request` or `INTEGER`

- A request handle is allocated when a non-blocking communication is initiated
- The request handle is used for testing if a specific communication has completed



Non-Blocking Send

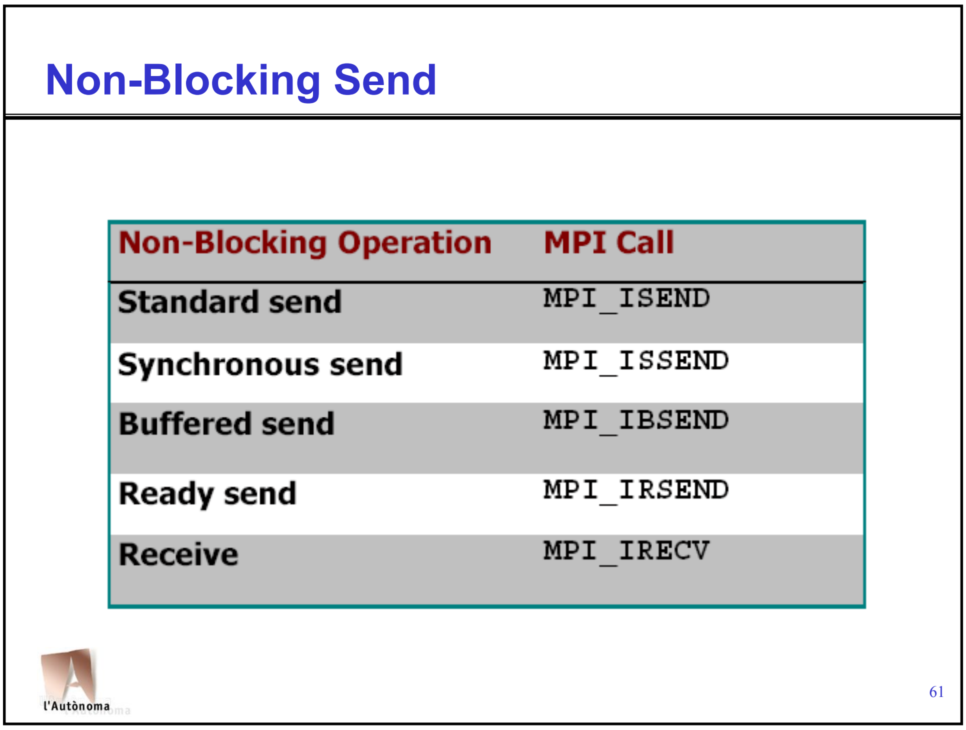
C:

```
int MPI_Isend(void *buf, int count,
              MPI_Datatype datatype,int dest,
              int tag, MPI_Comm comm,
              MPI_Request *request)
```

- Processing continues immediately without waiting for the message to be copied out from the application buffer

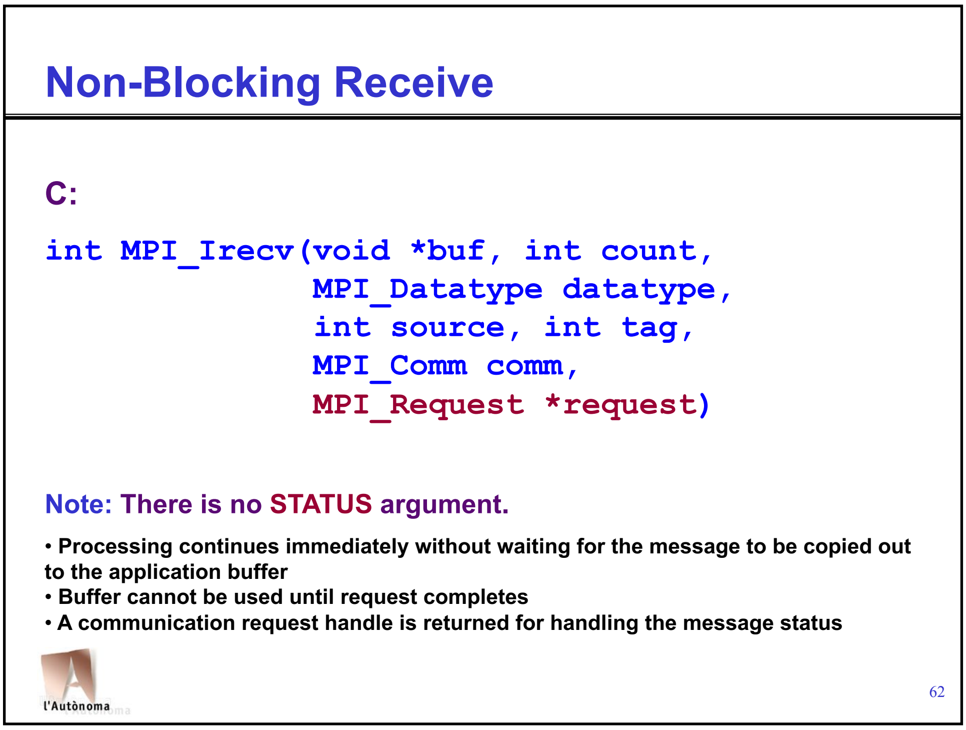
- Buffer cannot be reused until request completes

- A communication request handle is returned for handling the message status



Non-Blocking Send

Non-Blocking Operation	MPI Call
Standard send	`MPI_ISEND`
Synchronous send	`MPI_ISSEND`
Buffered send	`MPI_IBSEND`
Ready send	`MPI_IRSEND`
Receive	`MPI_IRECV`



Non-Blocking Receive

C:

```
int MPI_Irecv(void *buf, int count,
              MPI_Datatype datatype,
              int source, int tag,
              MPI_Comm comm,
              MPI_Request *request)
```

Note: There is no STATUS argument.

- Processing continues immediately without waiting for the message to be copied out to the application buffer
- Buffer cannot be used until request completes
- A communication request handle is returned for handling the message status

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking

- A blocking send can be used with a non-blocking receive, and vice-versa

- Blocking and non-blocking sends can use any mode -- synchronous, buffered, standard, or ready

Completion Tests

- Posted sends and receives must be completed.
- The completion status can be checked by calling one of completion routines.
- Waiting vs. Testing

wait	routine does not return until completion finished (blocking)
test	routine returns a TRUE or FALSE value depending on whether or not the communication has completed (non-blocking)

Completion Tests

C:

```
int MPI_Wait(MPI_Request *request,
            MPI_Status *status)


int MPI_Test(MPI_Request *request,
             int *flag,
             MPI_Status *status)
```

Note: Here is where `STATUS` appears.

Completion Tests

- Test or wait for completion of one (and only one) message

  `MPI_Waitany`

  `MPI_Testany`

- Test or wait for completion of all messages

  `MPI_Waitall`

  `MPI_Testall`

- Test or wait for completion of as many messages as possible

  `MPI_Waitsome`

  `MPI_Testsome`

Comparisons & General Use

Blocking:

call MPI_RECV (x,N,MPI_Datatype,…,status,…)

Non-Blocking:

call MPI_IRECV (x,N,MPI_Datatype,…,request,…)

… do work that does not involve array x

call MPI_WAIT (request,status)

… do work that does involve array x

Non-Blocking:

call MPI_IRECV (x,N,MPI_Datatype,…,request,…)

call MPI_TEST (request,flag,status,…)

do while (flag .eq. FALSE)

… work that does not involve the array x …

call MPI_TEST (request,flag,status,…)

end do

do work

Non-blocking - Sample Program #2

```
/* deadlock avoided */
#include
#include

void main (int argc, char **argv) {

int myrank;
MPI_Request request;
MPI_Status status;
double a[100], b[100];

MPI_Init(&argc, &argv);  /* Initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
if( myrank == 0 ) {
  /* Post a receive, send a message, then wait */
  MPI_Irecv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &request );
  MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  MPI_Wait( &request, &status );
}
else if( myrank == 1 ) {
  /* Post a receive, send a message, then wait */
  MPI_Irecv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &request );
  MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
  MPI_Wait( &request, &status );
}

MPI_Finalize();          /* Terminate MPI */

}
```

Comparisons & General Use

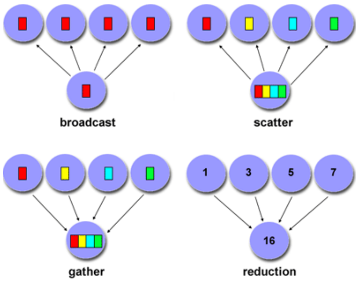
- use of non-blocking routines makes it much easier to write deadlock-free code
- on systems where latencies are large, posting receives early is often an effective, simple strategy for masking communication overhead.
- using non-blocking send and receive routines may increase code complexity, which can make code harder to debug and harder to maintain

Table of Contents

- Message passing
- History of MPI
- MPI Program Structure
- What’s in a Message
- Communications
  - Point-to-Point
  - Blocking/Non-Blocking modes
- Collective communications

Collective Communication

- Communications involving a group of processes
- Called by *all* processes in a communicator
- Examples:
  - Barrier synchronization
  - Broadcast, scatter, gather (Data Distribution)
  - Global sum, global maximum, etc. (Collective Operations)



Collective Communication

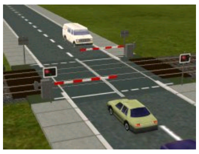
- Collective communication will not interfere with point-to-point communication and vice-versa
- All processes must call the collective routine
- Synchronization not guaranteed (except for barrier)
- Blocking and non-blocking collective communication available (from MPI 3.0)
- No tags
- Receive buffers must be exactly the right size

Barrier Synchronization

- Red light for each processor: turns green when all processors have arrived
- Slower than hardware barriers (example: Cray T3E)

C:

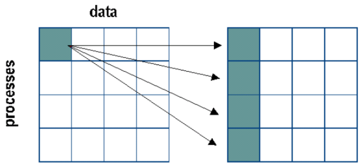
```
int MPI_Barrier (MPI_Comm comm)
```



Broadcast

- One-to-all communication: same data sent from root process to all the others in the communicator
- All processes must specify same root rank and communicator
- C:

```
int MPI_Bcast (void *buffer,
               int count,
               MPI_Datatype datatype,
                    int root,
               MPI_Comm comm)
```



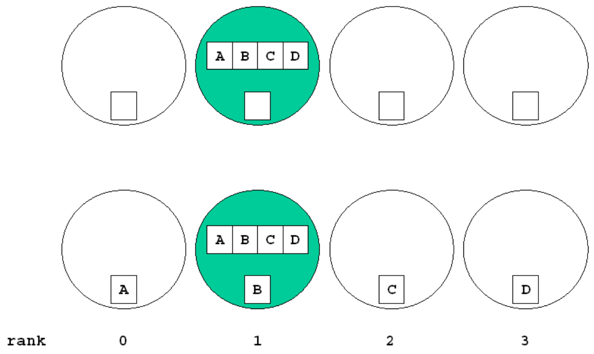
Broadcast - Sample Program #4

```
#include<mpi.h>
void main (int argc, char *argv[]) {
  int rank;
  double param;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  if(rank==5) param=23.0;
  MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
  printf("P:%d after broadcast parameter is %f\n",rank,param);
  MPI_Finalize();
}
```

```
P:0 after broadcast parameter is 23.000000
P:6 after broadcast parameter is 23.000000
P:5 after broadcast parameter is 23.000000
P:2 after broadcast parameter is 23.000000
P:3 after broadcast parameter is 23.000000
P:7 after broadcast parameter is 23.000000
P:1 after broadcast parameter is 23.000000
P:4 after broadcast parameter is 23.000000
```

Scatter

- One-to-all communication: different data sent to each process in the communicator (in rank order)
  - send arguments are significant only at the root process
  - example: Matrix-vector multiply of matrix distributed by rows



Scatter

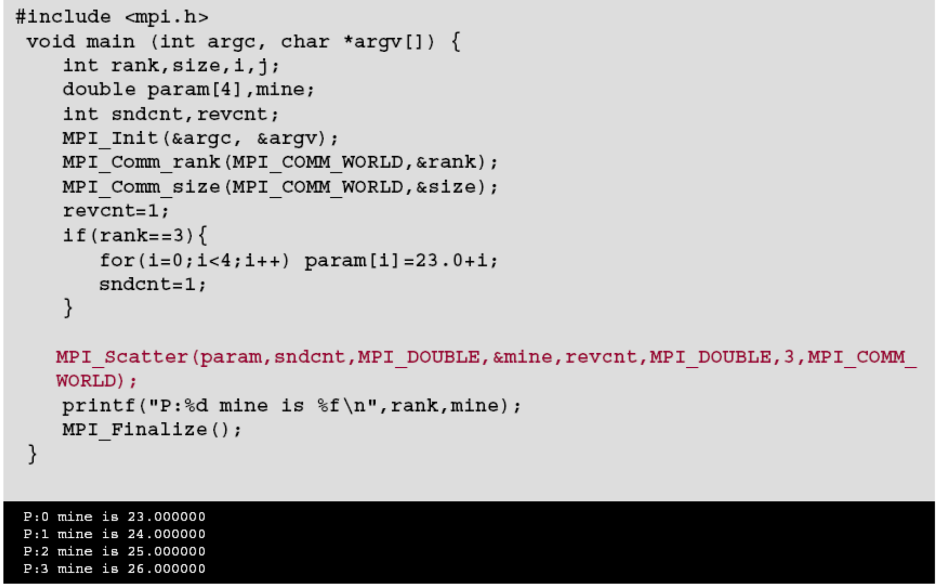
C:

```
int MPI_Scatter(void* sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                     void* recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                     int root,

                MPI_Comm comm)
```

- `sendcount` is the number of elements sent to each process, not the "total" number sent

Scatter - Sample Program #5



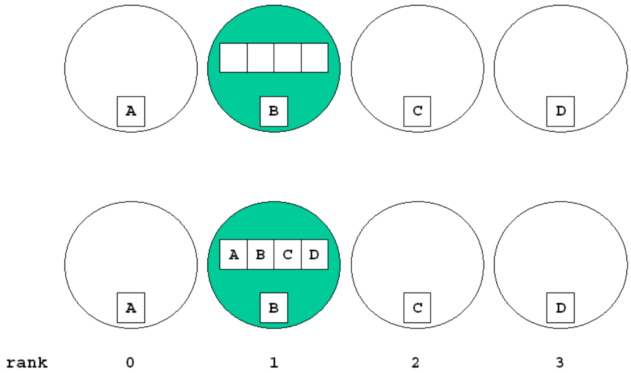
```
#include <mpi.h>
 void main (int argc, char *argv[]) {
    int rank,size,i,j;
    double param[4],mine;
    int sndcnt,revcnt;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    revcnt=1;
    if(rank==3){
       for(i=0;i<4;i++) param[i]=23.0+i;
       sndcnt=1;
    }

   MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,revcnt,MPI_DOUBLE,3,MPI_COMM_
   WORLD);
    printf("P:%d mine is %f\n",rank,mine);
    MPI_Finalize();
 }
```

```
P:0 mine is 23.000000
P:1 mine is 24.000000
P:2 mine is 25.000000
P:3 mine is 26.000000
```

Gather

- All-to-one communication: different data collected by root process in the communicator (in rank order)
  - receive arguments only meaningful at the root process
  - output vector needed in entirety by one process



Gather

C:

```
int MPI_Gather (void* sendbuf,
               int sendcount,
               MPI_Datatype sendtype,
               void* recvbuf,
               int recvcount,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm)
```

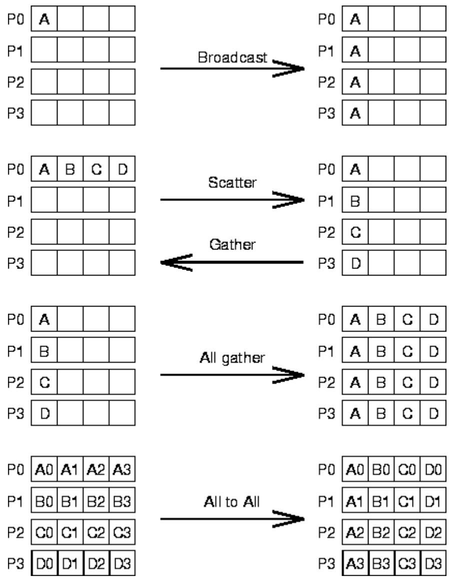
- It has the same arguments as matching scatter routines

Gather/Scatter Variations

- `MPI_Allgather`
- `MPI_Alltoall`

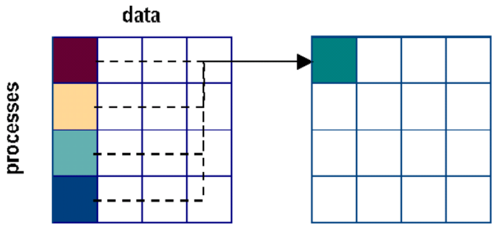
- No root process specified: all processes get gathered or scattered data
- Send and receive arguments significant for all processes

Collective communications



Reduction Operations

- Used to compute a result involving data distributed over a group of processes
- Collective operation in which a single process (the root process) collects data from the other processes in a group and combines them into a single data item
- Examples:
  - – Global sum or product
  - – Global maximum or minimum
  - – Global user-defined operation

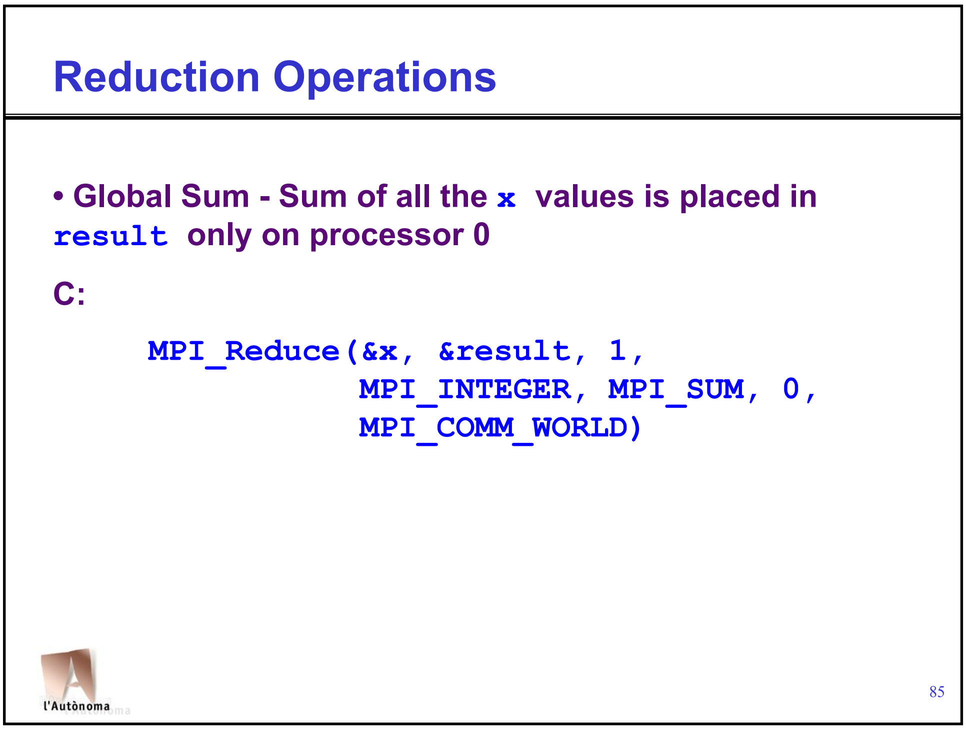


Reduction Operations

C:

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

- `count` is the number of elements in send buffer (integer)
- `op` is an associative operator that takes two operands of type `datatype` and returns a result of the same type

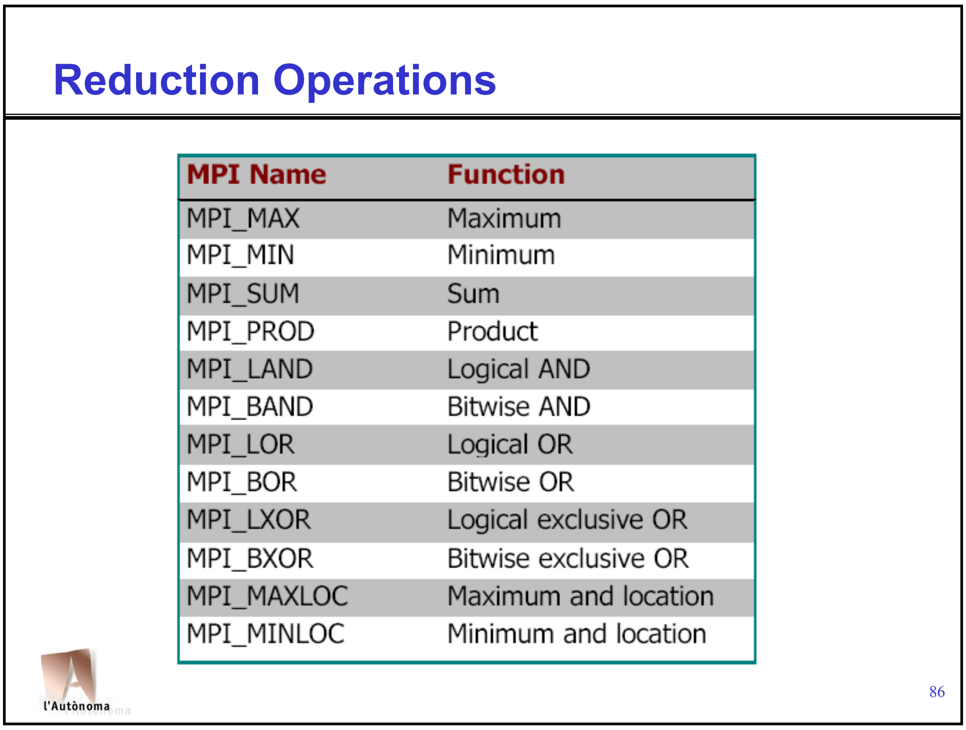


Reduction Operations

- Global Sum - Sum of all the `x` values is placed in `result` only on processor 0

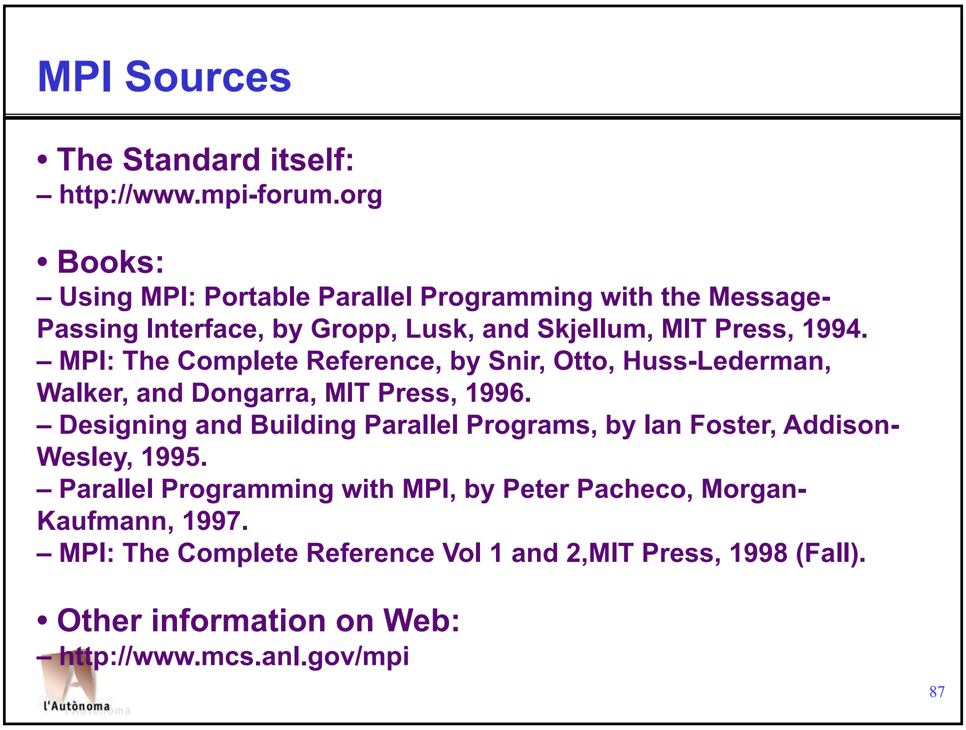
C:

```
MPI_Reduce(&x, &result, 1,
           MPI_INTEGER, MPI_SUM, 0,
           MPI_COMM_WORLD)
```



Reduction Operations

MPI Name	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 Sources

- **The Standard itself:**

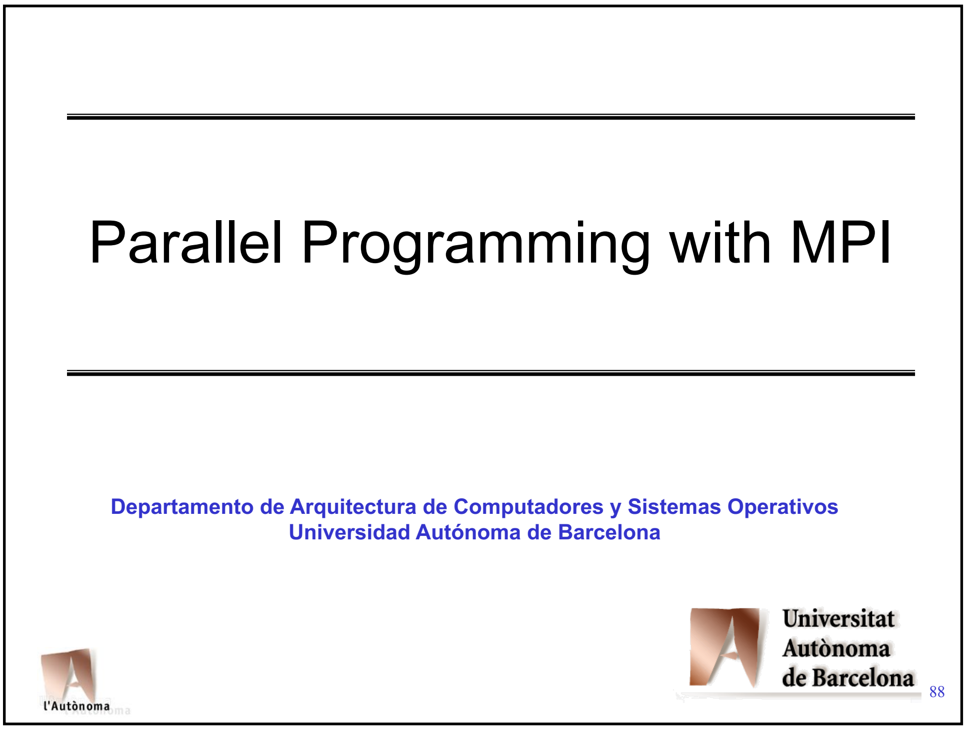
– http://www.mpi-forum.org

- **Books:**

– Using MPI: Portable Parallel Programming with the Message-Passing Interface, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
– MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
– Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.
– Parallel Programming with MPI, by Peter Pacheco, Morgan-Kaufmann, 1997.
– MPI: The Complete Reference Vol 1 and 2,MIT Press, 1998 (Fall).

- **Other information on Web:**

– http://www.mcs.anl.gov/mpi



Parallel Programming with MPI

Departamento de Arquitectura de Computadores y Sistemas Operativos
Universidad Autónoma de Barcelona

Universitat Autònoma de Barcelona