Nicola Spallanzani
n.spallanzani@cineca.it
High Performance Computing department

Parallel programming with MPI

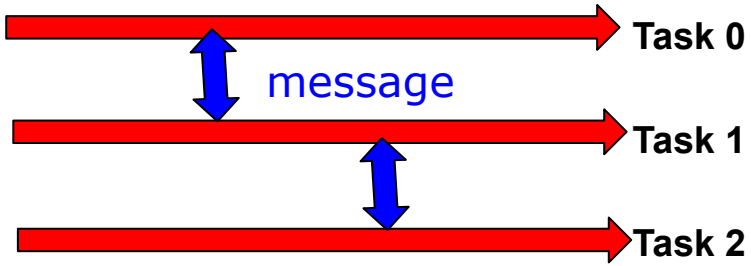
Introduction

Contents

- Introduction to message passing and MPI
- Basic MPI programs
- MPI Communicators
- Send and Receive function calls for point-to-point communications
- Blocking and non-blocking
- How to avoid deadlocks

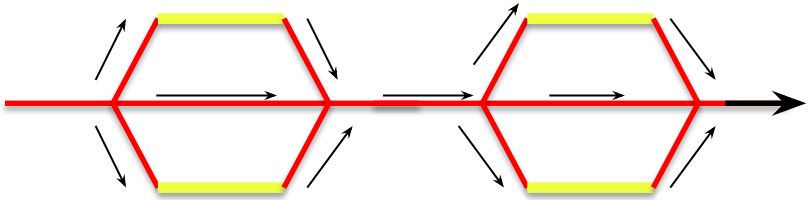
Message passing and shared memory parallelism

message passing



Multiple tasks exchange data via explicit messages

shared memory



Program splits into threads which share data via variables in shared memory

Message Passing

- Unlike the shared memory model, resources are **local**;
- Each process operates in its own environment (logical address space) and communication occurs via the **exchange of messages**;
- Messages can be instructions, data or synchronisation signals;
- The message passing scheme can also be implemented on shared memory architectures;
- Delays are much longer than those due to shared variables in the same memory space;

Advantages and Drawbacks

- **Advantages**
  - Communications hardware and software are important components of HPC system and often very **highly optimised**;
  - Portable and scalable;
  - Long history (many applications already written for it);

- **Drawbacks**
  - Explicit nature of message-passing is error-prone and discourages frequent communications;
  - Most serial programs need to be completely **re-written**;
  - High memory overheads.

The most important concept in message passing is...

...to minimize message passing as much as possible!

To maximise performance, the program should spend as little time as possible communicating data or waiting for other processes.

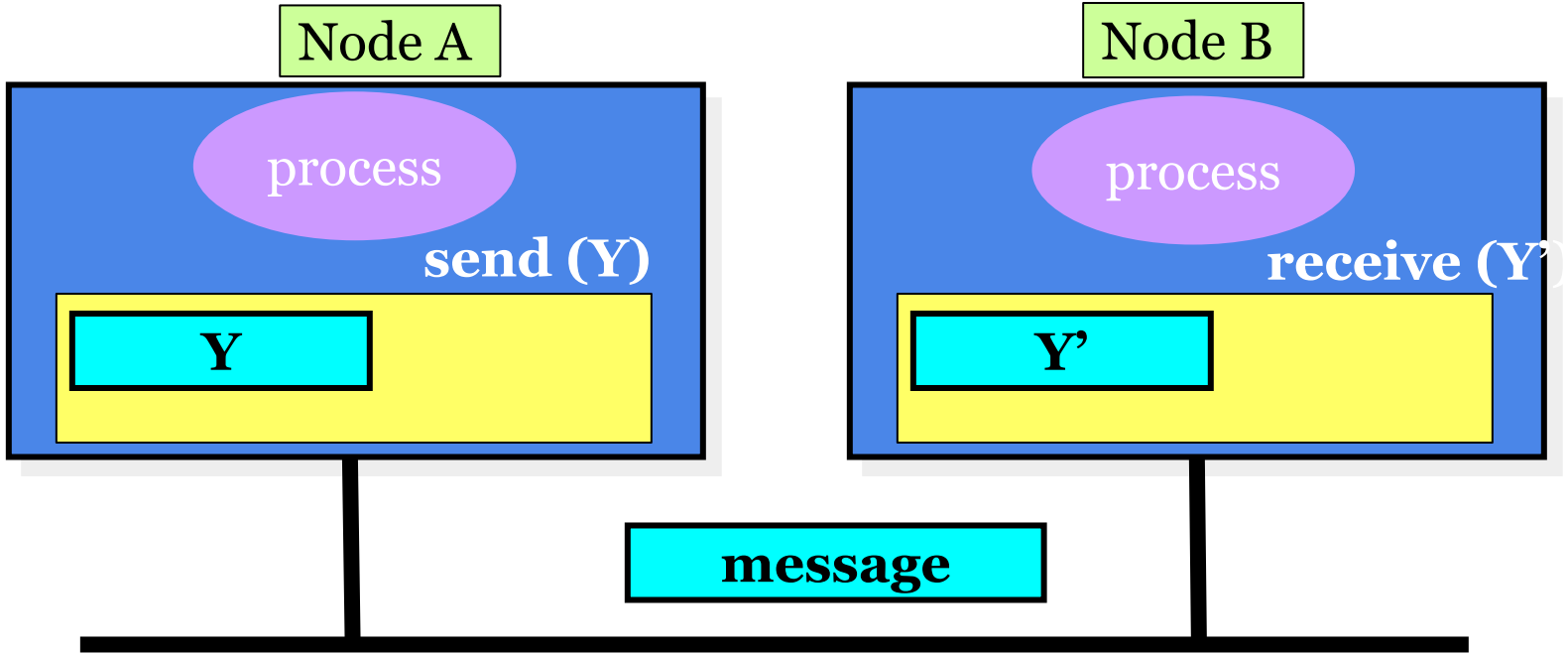
Data transfer and Synchronization

The sender process cooperates with the destination process

The communication system must allow the following three operations:

- send(*message*)
- receive(*message*)
- synchronization

MPI Programming Model



The Message Passing Interface MPI

- MPI is a **standard** defined in a set of documents compiled by a consortium of organizations: http://www.mpi-forum.org
- In particular the MPI documents define the APIs (application interfaces) for C, C++, FORTRAN77 and FORTRAN90.
- The actual **implementation** of the standard is left to the software developers of the different systems
- In all systems MPI has been implemented as a **library of subroutines** over the **network** with drivers and primitives

Goals of the MPI standard

MPI's prime goals are:

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

MPI also offers:

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

MPI2 further extends the library power (parallel I/O, Remote Memory Access, Multi Threads)

MPI3 aims to support exascale by including non-blocking collectives, improved RMA and neighbourhood collectives.

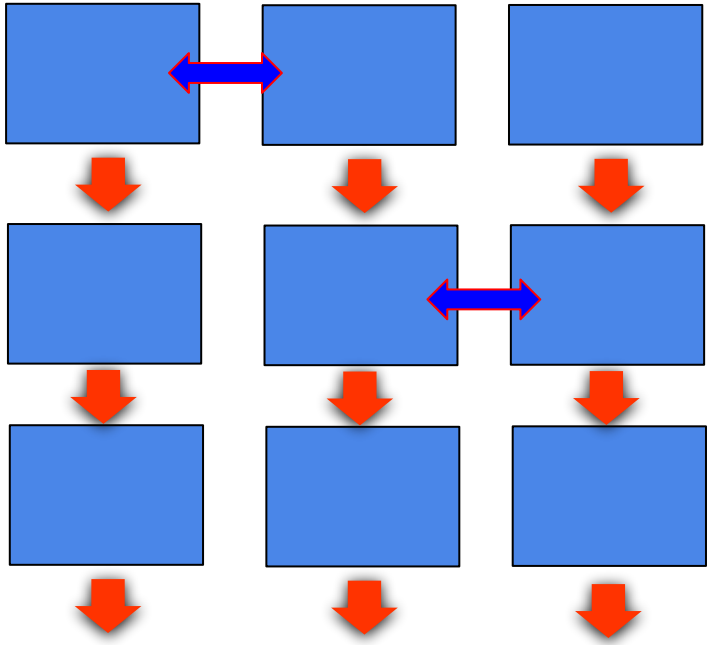
Basic Features of MPI Programs

An MPI program consists of multiple instances of a serial program that communicate by library calls.

Calls may be roughly divided into four classes:

1. Calls used to **initialize, manage, and terminate** communications
2. Calls used to **communicate between pairs** of processors. (point to point communication)
3. Calls used to **communicate among groups** of processors. (collective communication)
4. Calls to create **data types**.

Single Program Multiple Data (SPMD) programming model



Multiple instances of the **same** program.

A note about MPI Implementations

- The MPI standard defines the functionalities and the API, i.e. what the C or FORTRAN calls should look like.
- The MPI standard **does not define how the calls should be performed at the system level** (algorithms, buffers, etc) or how the environment is set up (env variables, mpirun or mpiexec, libraries, etc). This is left to the **implementation**.
- There are various implementations (IntelMPI, OpenMPI, MPICH, HPMPI, etc) which have different performances, features and standards compliance.
- On some clusters (e.g. Galileo, Marconi) you may choose which MPI to use, on other systems you have only the vendor-supplied version (IBM MPI for FERMI).

Compiling and Running MPI programs

- Implementation is system dependent but it is usual to use the “**wrapped**” version of the compiler to include the **MPI headers and link** in the MPI libraries. Wrapped compilers tend to be called mpif90, mpicc, mpic++, etc.
- On HPC systems MPI programs are run via the **batch system** with appropriate settings. For debugging sometimes it is possible to open interactive sessions (e.g. SLURM on GALILEO).
- a program such as **mpirun** or **mpiexec** is then used to launch multiple instances of the program on the assigned nodes. For MARCONI and GALILEO you can also use the **srun** command.

Compiling and running MPI on GALILEO

Compile+link using Openmpi

```
module load autoload openmpi
mpicc -o mpi_prog mpi_prog.c
```

Compile+link using Intelmpi (recommended)

```
module load autoload intelmpi
mpiicc -o mpi_prog mpi_prog.c
```

Job script

```
#SBATCH -N 1
#SBATCH --ntasks-per-node=16
#SBATCH -t 1:00:00

module load autoload intelmpi
mpirun -np 12 ./mpi_prog
(for SLURM: srun -n 12 ./mpi_prog)
```

A First Program: Hello World!

Fortran

```
PROGRAM hello

  INCLUDE 'mpif.h'
  INTEGER err

  CALL MPI_INIT(err)
  PRINT *, "hello world!"
  CALL MPI_FINALIZE(err)

END
```

C

```
#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();
    }
```

Header files

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

C:

#include <mpi.h>

Fortran:

include 'mpif.h'

Fortran 90:

USE MPI

Fortran 08 (MPI-3):

USE MPI_F08

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

FORTRAN note:

The FORTRAN include and module forms are *not equivalent*: the module can also do type checking. Some compilers gave problems with the module but it is now highly recommended to use the module, particularly for FORTRAN 2008 (most rigorous type-checking)

MPI function format

C:

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

FORTRAN:

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

Initializing MPI

C:

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

FORTRAN:

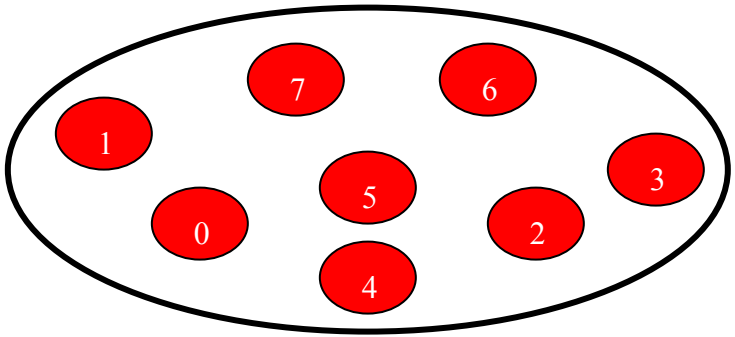
```
INTEGER IERROR
MPI_INIT(IERROR)
```

Must be first MPI call: initializes the message passing routines

MPI Communicator

- In MPI it is possible to divide the total number of processes into groups, called *communicators*.
- The Communicator is a variable identifying a group of processes that are allowed to communicate with each other.
- The communicator that includes all processes is called MPI_COMM_WORLD
- MPI_COMM_WORLD is the default communicator (automatically defined)

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



MPI_COMM_WORLD

Communicator Size

How many processes are associated with a communicator?

C:

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

FORTRAN:

```
INTEGER COMM, SIZE, IERR
OUTPUT:  SIZE
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
```

Process Rank

How can you identify different processes?

What is the ID of a processor in a group?

C:

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

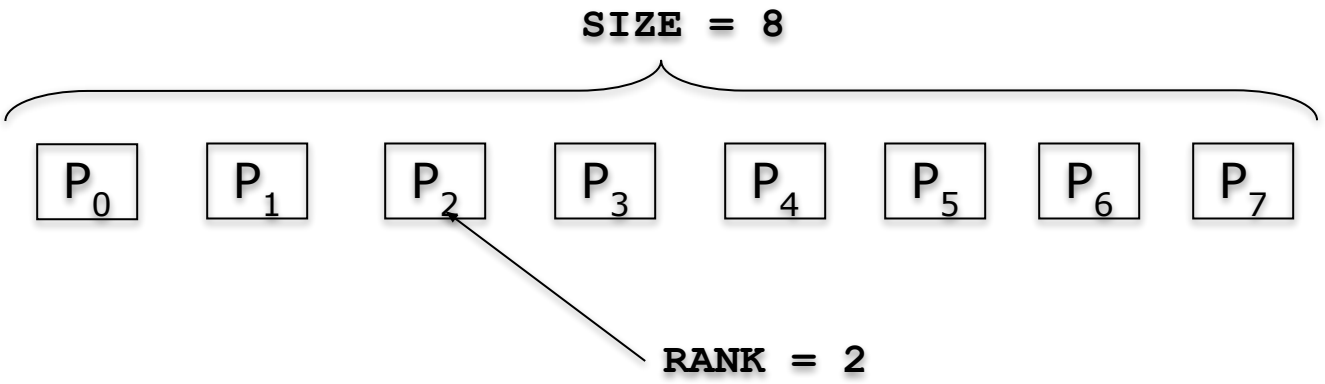
Fortran:

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

- *rank* is an integer that identifies the Process inside the communicator *comm*
- MPI_COMM_RANK is used to find the rank (the name or identifier) of the Process running the code

Communicator Size and Process Rank / 1

How many processes are contained within a communicator?



size is the number of processes 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

Exiting MPI

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**. Before and after these calls all instances execute the same code (as in serial execution)

MPI_ABORT

- Usage

```
int MPI_Abort( MPI_Comm comm, int errorcode );
```

- Description

  Terminates all MPI processes associated with the communicator comm; in most systems (all to date), terminates *all* processes.

A Template for Fortran MPI Programs

```
PROGRAM template

    USE MPI
    INTEGER :: ierr, myid, nproc

    CALL MPI_INIT(ierr)
    CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
    CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

        !!! INSERT YOUR PARALLEL CODE HERE !!!

    CALL MPI_FINALIZE(ierr)

END PROGRAM
```

A Template for C MPI programs

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

void main (int argc, char * argv[])
{
  int err, nproc, myid;

  err = MPI_Init(&argc, &argv);
  err = MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  /*** INSERT YOUR PARALLEL CODE HERE ***/

  err = MPI_Finalize();
}
```

Example

```
PROGRAM hello

    IMPLICIT NONE
    USE MPI
    INTEGER :: myPE, totPEs, ierr

    CALL MPI_INIT(ierr)
    CALL MPI_COMM_RANK( MPI_COMM_WORLD, myPE, ierr )
    CALL MPI_COMM_SIZE( MPI_COMM_WORLD, totPEs, ierr )
    PRINT *, "myPE is ", myPE, "of total ", totPEs, " PEs"
    CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

```
MyPE is 1 of total 4 PEs
MyPE is 0 of total 4 PEs
MyPE is 3 of total 4 PEs
MyPE is 2 of total 4 PEs
```

Output (4 Procs)