Introduction to the Message Passing Interface (MPI)

Themis Athanassiadou

CSCS

Jul 8-19, 2013

Outline

What is Message Passing Interface?

MPI (Message-Passing Interface) is a message-passing library interface specification. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Extensions to the "classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O. MPI is a specification, not an implementation; there are multiple implementations of MPI. This specification is for a library interface; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and Fortran, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers.

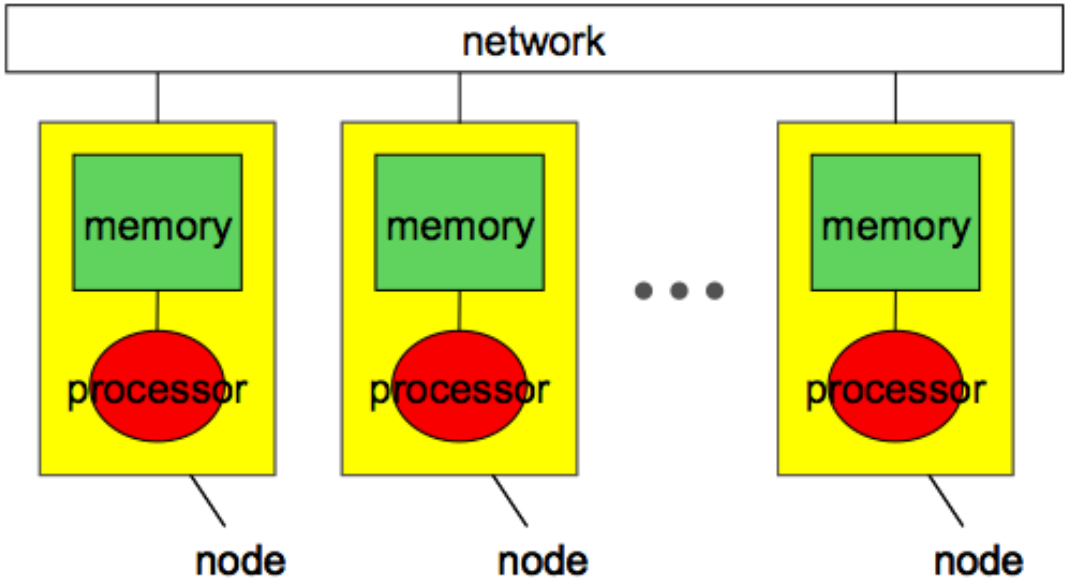
`mpi-forum.org`

In plain English:

- Its not a language or library!
- A set of specifications for communication between parallel processes
  Think of program elements like functions or subroutines
- Different vendor implementations: OpenMPI, MPICH2, ...
  Think of compilers
- Bindings for different programming languages (C, Fortran, ...)
  Think about a functions in C or subroutines in Fortran

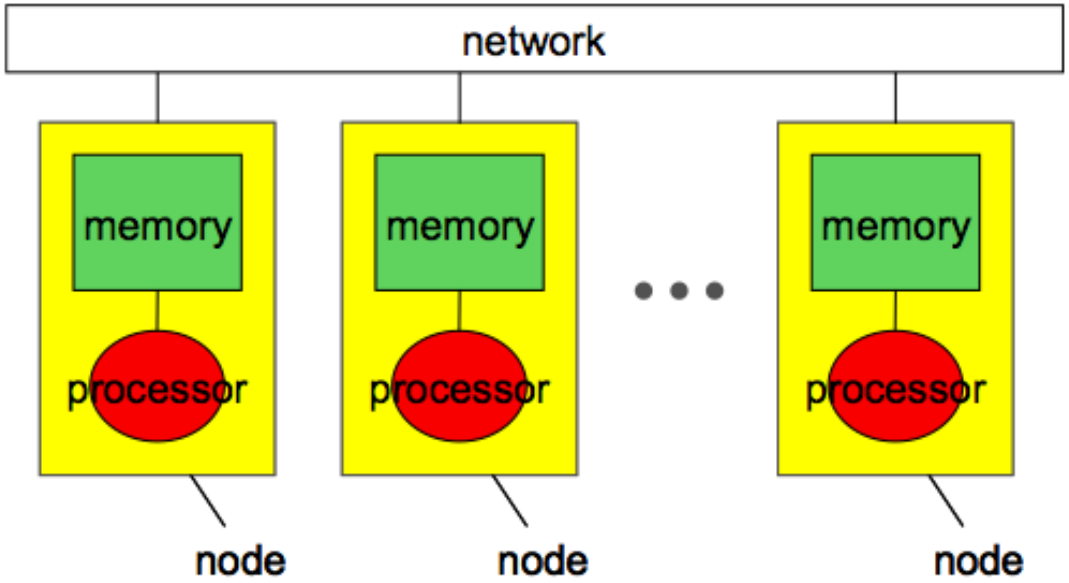
When do you need to use it?

When each process has its own address space, and no (other) way to get at another's
The processes must agree on a way to send and receive data to/from each other.



When do you need to use it?

When each process has its own address space, and no (other) way to get at another's
The processes must agree on a way to send and receive data to/from each other.



PS: You should probably also have a big problem (memory/calculations) that requires a distributed machine!

MPI Program Structure

- include MPI header file

MPI Program Structure

- include MPI header file
- declare all variables

MPI Program Structure

- include MPI header file
- declare all variables
- initialize the MPI environment

MPI Program Structure

- include MPI header file
- declare all variables
- initialize the MPI environment
- do computation as usual + MPI communication calls:
  - manage communication
  - point-to-point-communications
  - collective communications

MPI Program Structure

- include MPI header file
- declare all variables
- initialize the MPI environment
- do computation as usual + MPI communication calls:
  - manage communication
  - point-to-point-communications
  - collective communications
- finalize MPI communications

Header Files

Header files contain the prototypes for MPI functions/subroutines as well as definitions of macros, special constants, and datatypes used by MPI. An appropriate "include" statement must appear in any source file that contains MPI function calls or constants !!

C:

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

FORTRAN:

```
INCLUDE 'mpif.h'
```

Initializing and Finalizing MPI

All processes must initialize and finalize MPI!

C

```
int err;
err = MPI_Init(&argc, &argv);

...program, program, program...

err = MPI_Finalize();
```

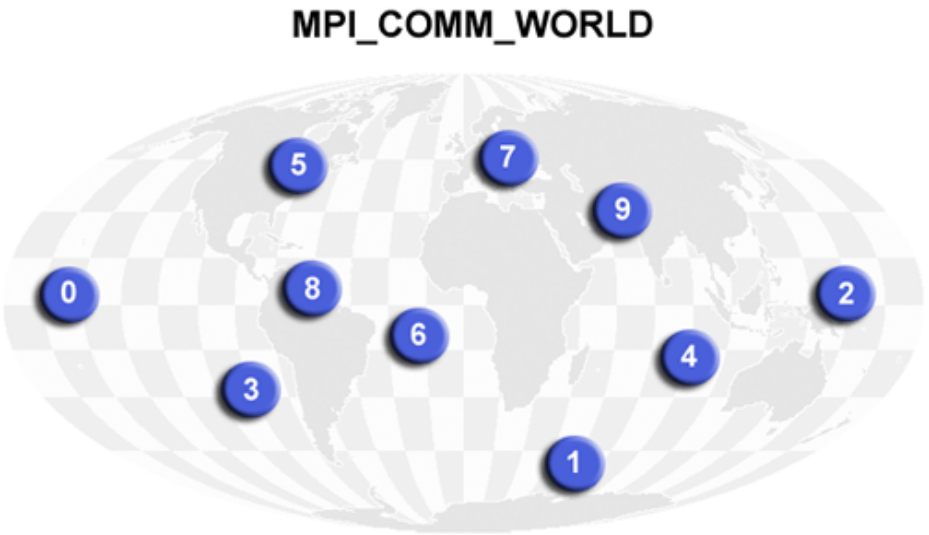
FORTRAN

```
INTEGER IERR
CALL MPI_INIT(IERR)

... program, program, program...

CALL MPI_FINALIZE(IERR)
```

Processes in communicators



- **communicator**: an MPI handle that defines a group of processes that are permitted to communicate with each other!
- The processes have a rank (aka name) within the communicator, so they can talk to each other
- the size of the communicator is the number of processes associated with it
- the communicator encompassing all processes is MPI_COMM_WORLD

Getting the rank and size

Note: For size p, rank ranges from [0,p-1]

C

```
int rank, size
        ...
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

FORTRAN

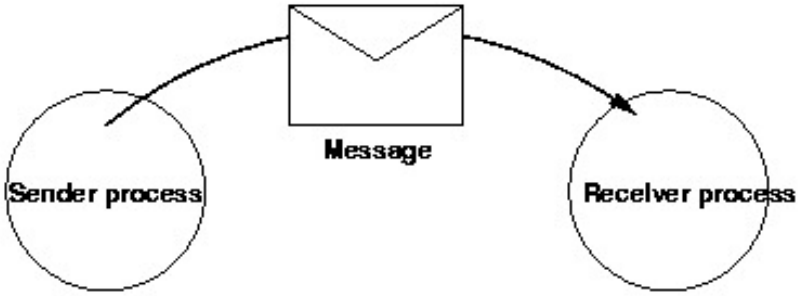
```
integer rank, size, ierr
        ...
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
```

Exercise #1 . Obligatory Hello World, in parallel

Instructions:

- cp -R /project/csstaff/courses/CSCS_USI_School/MPI_Practicals $SCRATCH
- cd $SCRATCH/Ex1 – Choose your computational mother tongue, C or F.
- Add code as per instructions.
- to compile CC *.c or ftn *.f
- grab a node! salloc -N 1 –res=sschool
- to run: aprun -n *number of processes [1-16] executable*
- *Now, rewrite the code, so that only rank 0 prints "hello world"*

Sending and Receiving



- the building block of MPI message passing!
- Address info : Who sends? Who receives? Tags
- Contents : The buffer (array), type (integer, real), size (# of array elements)
- MPI stuff : Error message, Status

Sending and Receiving

C

```
int rank,ierr;
float a10;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0){
  ierr = MPI_Send(a, 10, MPI_FLOAT, 1, 100, MPI_COMM_WORLD);
}
else if (rank == 1) {
  ierr = MPI_Recv(a, 10, MPI_FLOAT, 0, 100, MPI_COMM_WORLD, &status);
}
```

FORTRAN

```
integer rank,ierr
real a(10)
integer status(MPI_STATUS_SIZE)
...
call MPI_Comm_rank(MPI_COMM_WORLD, rank)
if (rank == 0) then
  call MPI_Send(a, 10, MPI_REAL, 1, 100, MPI_COMM_WORLD,ierr)
else if (rank == 1) then
  call MPI_Recv(a, 10, MPI_REAL, 0, 100, MPI_COMM_WORLD, status)
end if
```

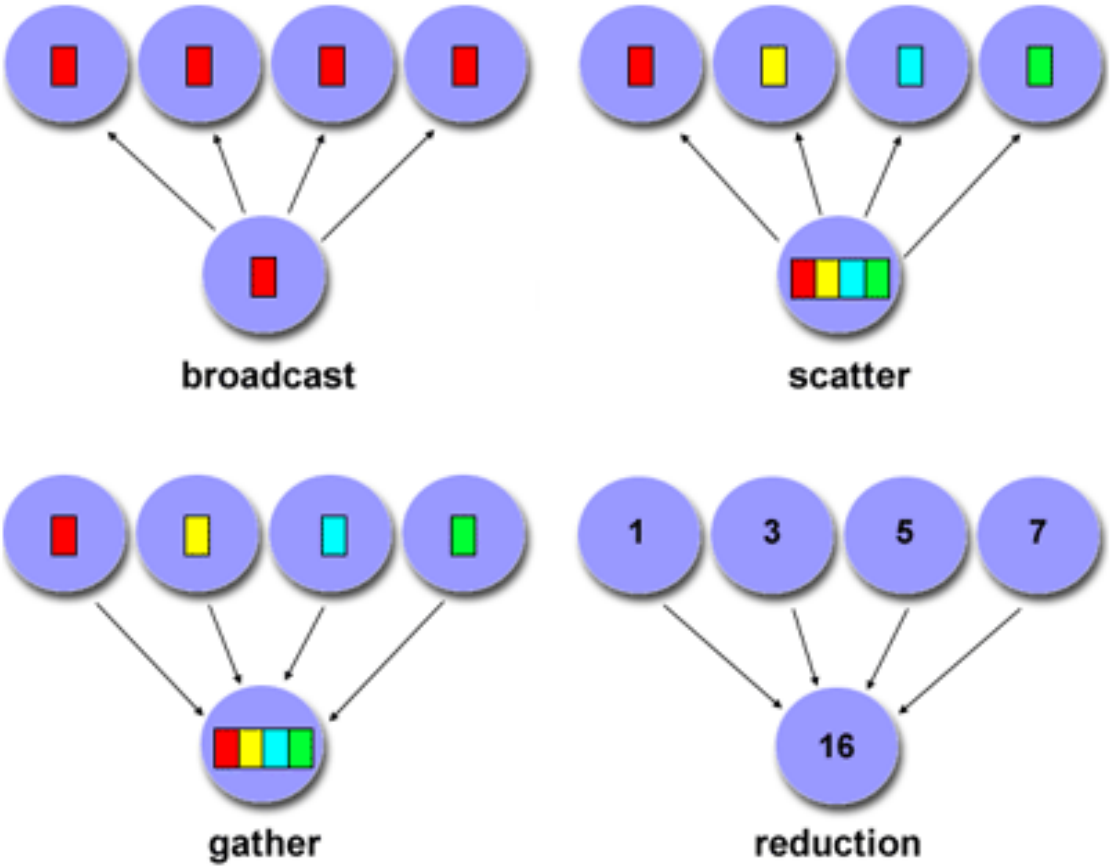
Exercise: Send and Receive a message!

- In MPI_Practicals: Go to MPI_Point_to_Point
- Complete the code and help two processors talk to each other
- when you run, aprun -n 2 *executable*

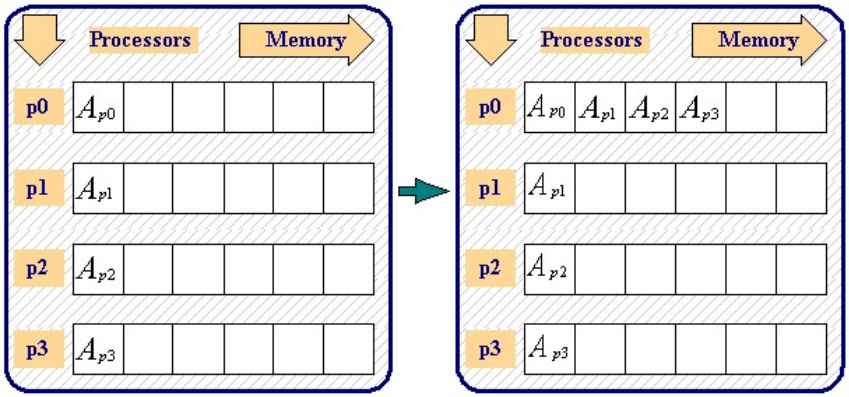
Collective Communication

- Communication pattern involves ALL processes within a communicator
- Three basic types of collective communications
  - Synchronization (i.e. MPI_Barrier)
  - Data Movement (i.e. MPI_Gather, MPI_Scatter, etc)
  - Movement with computation (i.e. MPI_Reduce, etc)

Collectives in pictures



Gathering



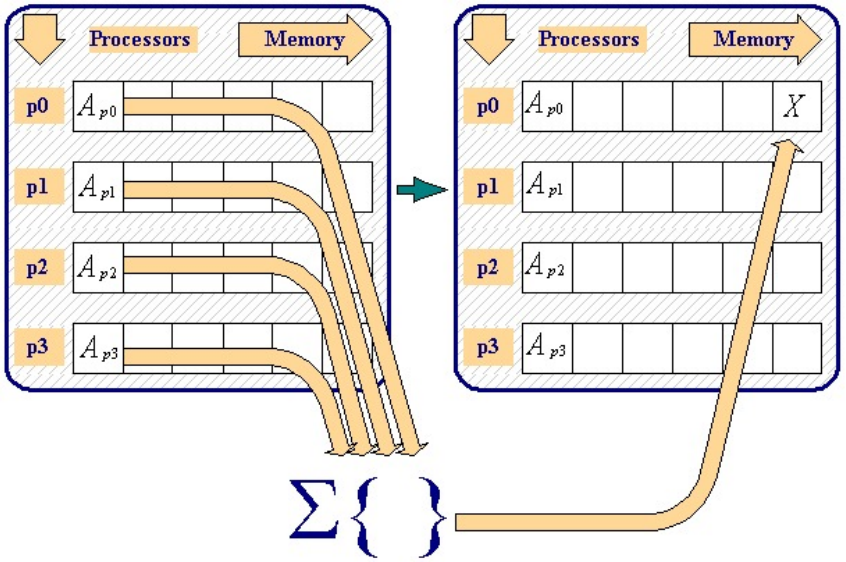
C

```
int ierr;
float a[10], a_gather[160]; /* Assuming 16 processes */
...
ierr = MPI_Gather(&a, 10, MPI_FLOAT, &a_gather, 10, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

FORTRAN

```
integer ierr
real a(10), a_gather(160) ! Assuming 16 processes
...
call MPI_Gather(a, 10, MPI_REAL, a_gather, 160, MPI_REAL, 0, MPI_COMM_WORLD,ierr)
```

Reducing



C

```
int ierr;
float a[10], a_sum[10];
...
ierr = MPI_Reduce(&a, &a_sum, 10, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
```

FORTRAN

```
integer ierr
real a(10), a_sum(10)
...
call MPI_Reduce(a, a_sum, 10, MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD,ierr)
```

Exercise #3: Reduce!

- go again to the MPI_Practicals Folder
- The exercise is in folder MPI_Reduce
- The fanciest calculator you ever used: Add numbers using one core for each number! (Its ok its educational).
- If you are feeling adventurous, you can play around with all the exercises in the folder.

Just the tip of the iceberg!

- www.mpi_forum.org
- www.citutor.org
- https://computing.llnl.gov/tutorials/mpi/