MESSAGE PASSING INTERFACE - 1

Dr. Emmanuel Pilli

What is MPI

- Message Passing Interface
- What is the message?

DATA

- Allows data to be passed between processes in a distributed memory environment

Goals and Scope

- MPI’s prime goals are:
  - To provide source-code portability
  - To allow efficient implementation
- It also offers:
  - A great deal of functionality
  - Support for heterogeneous parallel architectures

MPI Program Structure

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

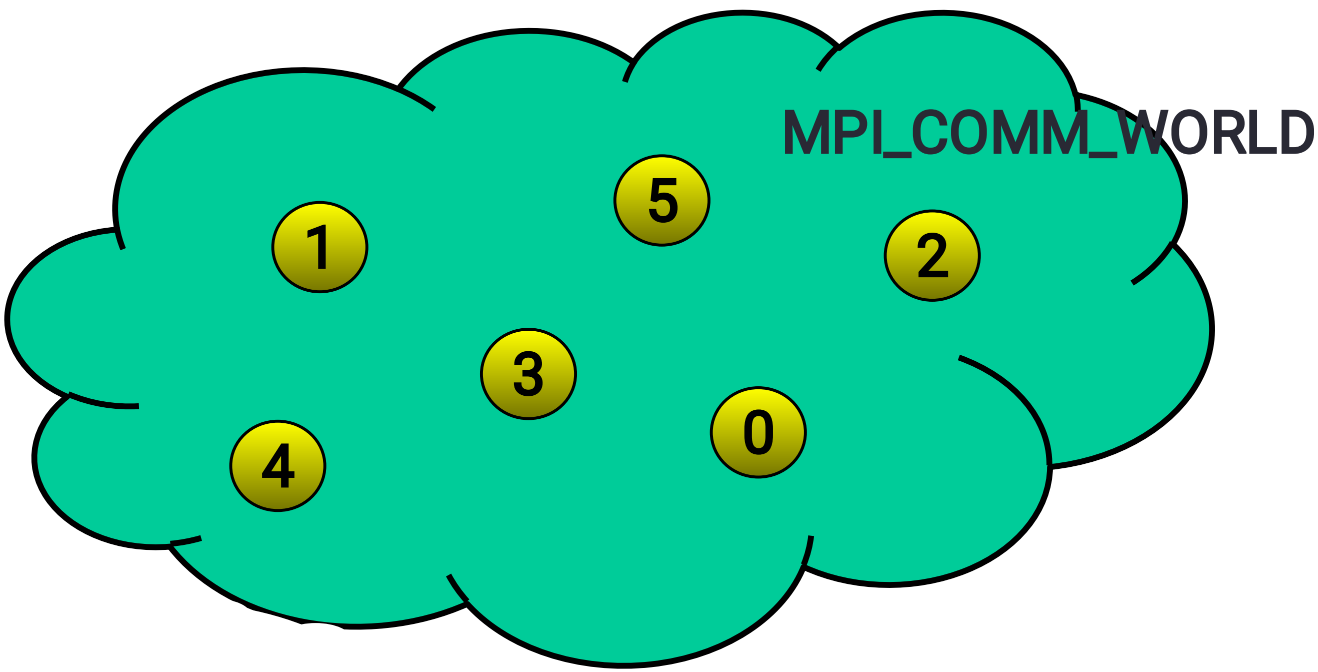
Handles

- MPI controls its own internal data structures
- MPI releases "handles" to allow programmers to refer to these
- C handles are of defined typedefs

Communicator

- Programmer view: group of processes that are allowed to communicate with each other
- All MPI communication calls have a communicator argument
- Most often you will use MPI_COMM_WORLD
  - Defined when you call **MPI_Init**
  - It is all of your processors...

MPI_COMM_WORLD Communicator



Header Files and Function Format

- MPI constants and handles are defined here

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

- Function Format

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

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

Initializing MPI and Communicator

- Must be the first routine called (only once)

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

- How many processes are contained within a communicator?

```
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

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

Basic MPI Program

```
#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);

/* … your code here … */

   MPI_Finalize ();
}
```

What’s in a Message

- Messages
- MPI Basic Datatypes
- Rules and Rationale

Messages

- A message contains an array of elements of some particular MPI datatype

- MPI Datatypes:
  - Basic types
  - Derived types

- Derived types can be build up from basic types
  - Covered Later …

Datatypes

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	

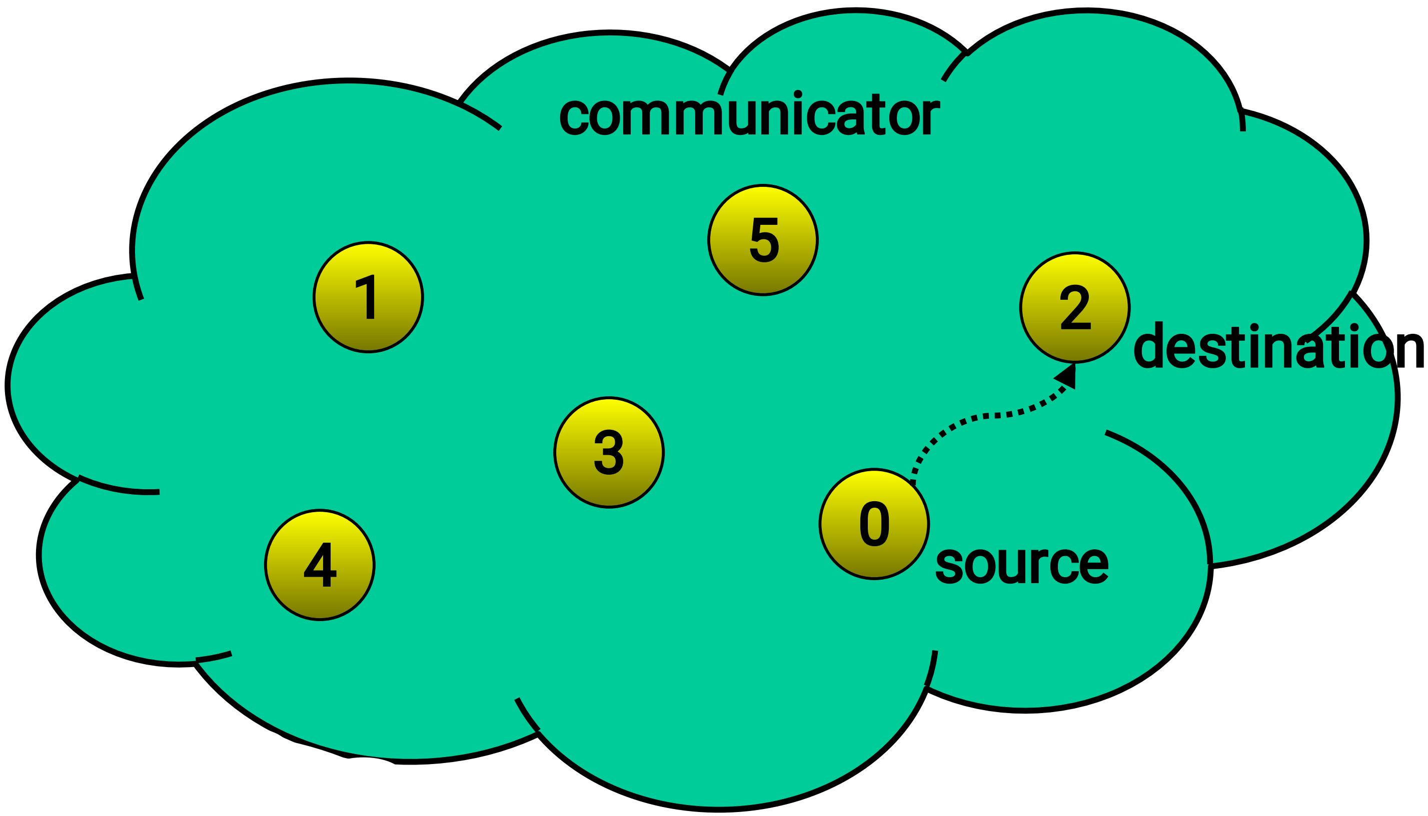
Rules and Rationale

- Programmer declares variables to have “normal” C type, but uses matching MPI datatypes as arguments in MPI routines
- Mechanism to handle type conversion in a heterogeneous collection of machines
- General rule: MPI datatype specified in a **receive** must match the MPI datatype specified in the **send**

Point to Point Communications

- Definitions
- Communication Modes
- Routine Names
- Sending a Message
- Receiving a Message
- Wild Carding
- Timers

Point to Point Communications



Point to Point Communications

- 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

Definitions

- "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

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

Routine Names (Blocking)

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

Sending a Message

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

Arguments

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_REAL,6,33,MPI_COMM_WORLD,IERROR)

Synchronous Send MPI_Ssend

- Completion criteria: Completes when message has been received
- Use if need to know that message has been received
- Sending & receiving processes synchronize
  - regardless of who is faster
  - processor idle time is probable
- "Fax-type" communication method

Buffered Send MPI_Bsend

- Completion criteria: Completes when message copied to 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

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)

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

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

Successful Communication

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

Wildcarding

- Receiver can wildcard
- 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

Using Status Handle

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

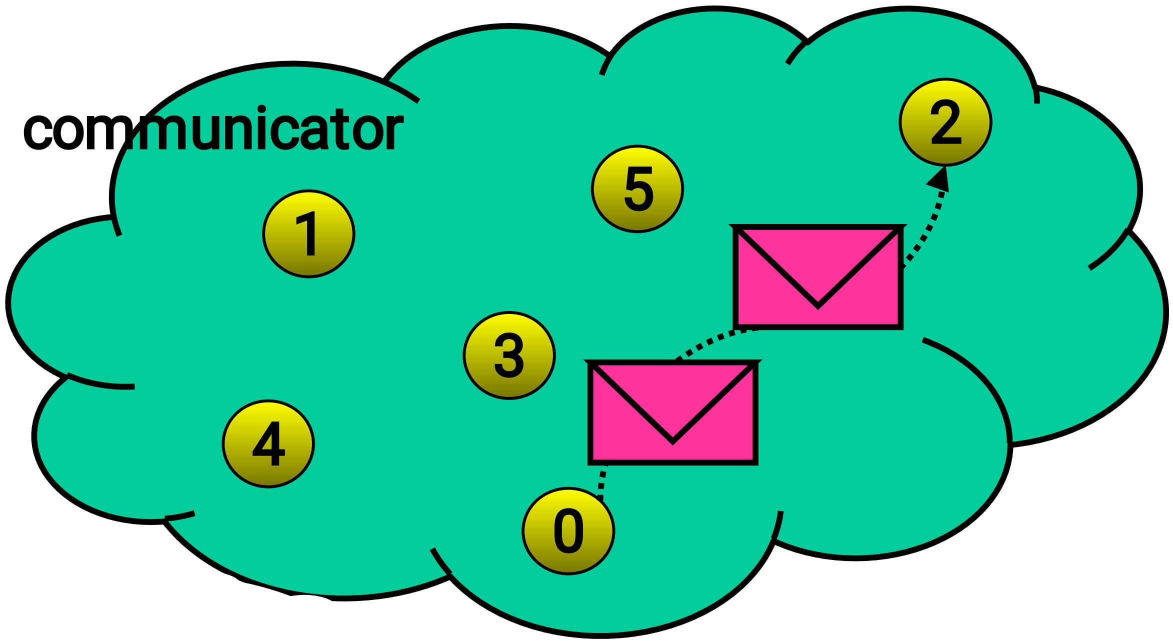
Information	C
source	status.MPI_SOURCE
tag	status.MPI_TAG
count	MPI_Get_count

Received Message Count

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

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

Message Order Preservation



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

Sample Program

```
#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();
 }
```

Timer

- Time is measured in seconds
- Time to perform a task is measured by consulting the timer before and after

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