Lecture 07
Message Passing Interface

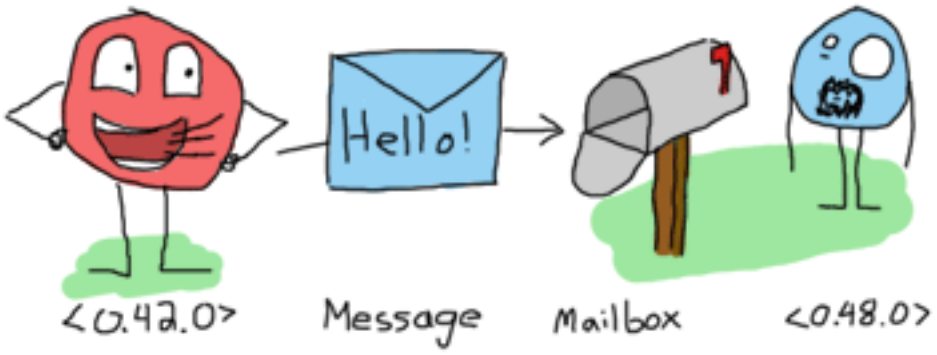
Dr. Ehtesham Zahoor

Distributed Systems - Definition

•« *A **distributed system** is a software system in which components **located on networked computers communicate and coordinate their actions by passing messages**. The components interact with each other in order to achieve a common goal* »

Message Passing

- In this lecture we would focus on patallel programming using a specific implementation of MPI



MPI (Message Passing Interface)?

- Standardized message passing library specification
  - for parallel computers clusters
  - not a specific product, compiler specification etc.
  - many implementations, MPICH, LAM, OpenMPI …

- Portable, with Fortran and C/C++ interfaces.
- *Real* parallel programming

A brief history of MPI

- Writing parallel applications is fun!!

- Initially it was a difficult and tedious task as there was not a standard accepted way of doing it.
- Various implementations existed, all used message passing model with feature differences

A brief history of MPI

- The authors of the libraries and others came together to define a standard interface - the Message Passing Interface.

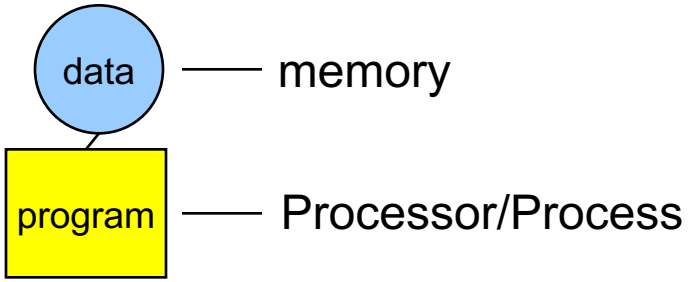
- This standard interface would allow programmers to write parallel applications that were portable to all major parallel architectures.

A brief history of MPI

- By 1994, a complete interface and standard was defined (MPI-1).
- It took another year for complete implementations of MPI to become available.
- MPI was widely adopted and still continues to be the *de-facto* method of writing message-passing applications.

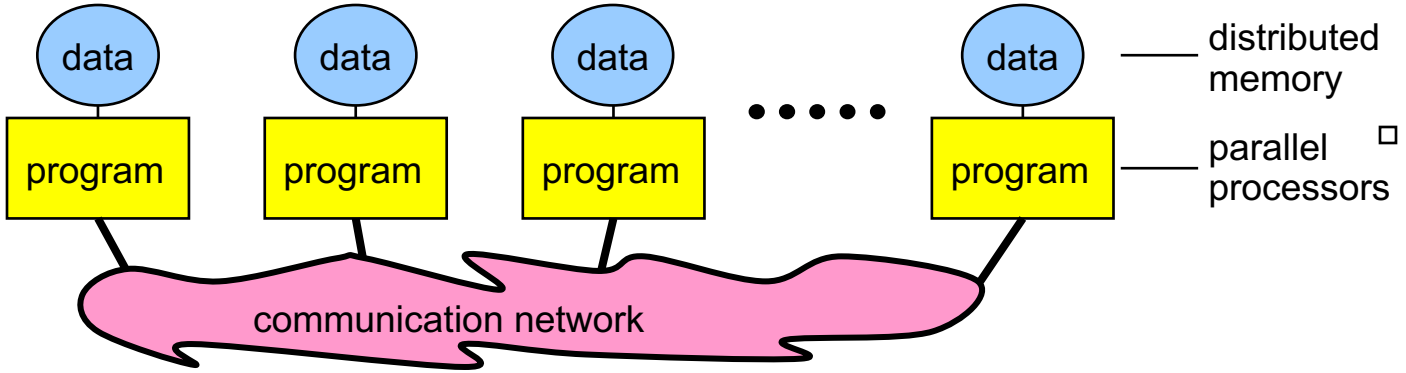
The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**

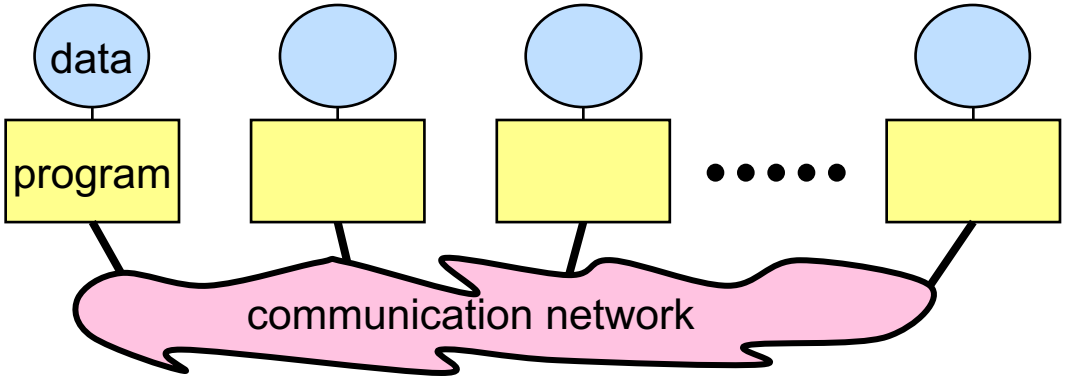


A processor may run many processes

- **Message-Passing Programming Paradigm**



- A **process** is a program performing a task on a **processor**
- Each processor/process in a message passing program runs a instance/copy of a ***program:***
  - written in a conventional sequential language, e.g., C or Fortran,
  - the variables of each sub-program have
    - the same name but different locations (distributed memory) and different data!
    - i.e., all variables are local to a process
  - communicate via ***message passing***



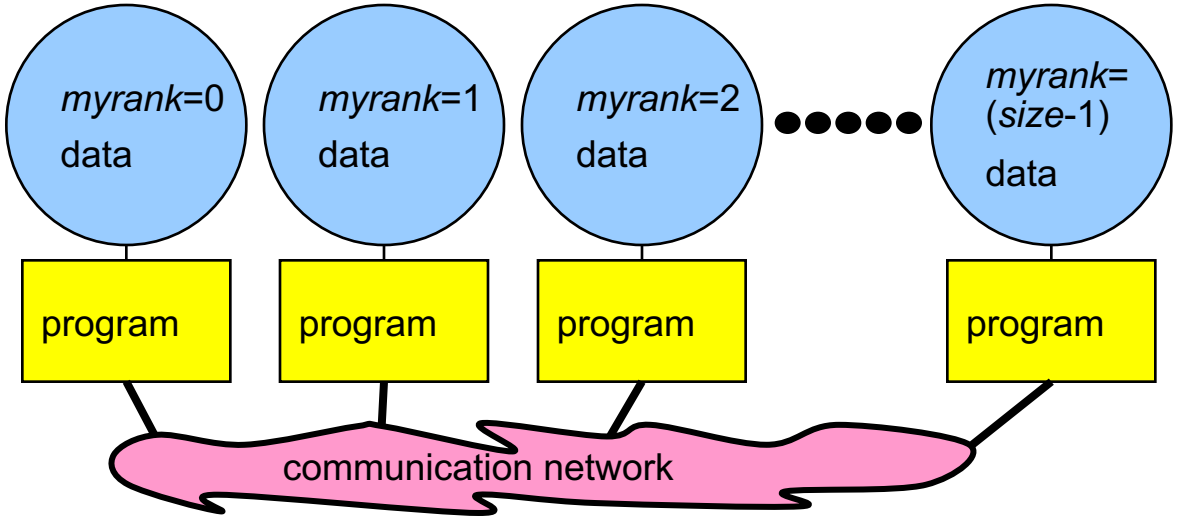
MPI Fundamentals

- A **communicator** defines a group of processes that have the ability to communicate with one another.

- In this group of processes, each is assigned a unique ***rank***, and they explicitly communicate with one another by their ranks.

Data and Work Distribution

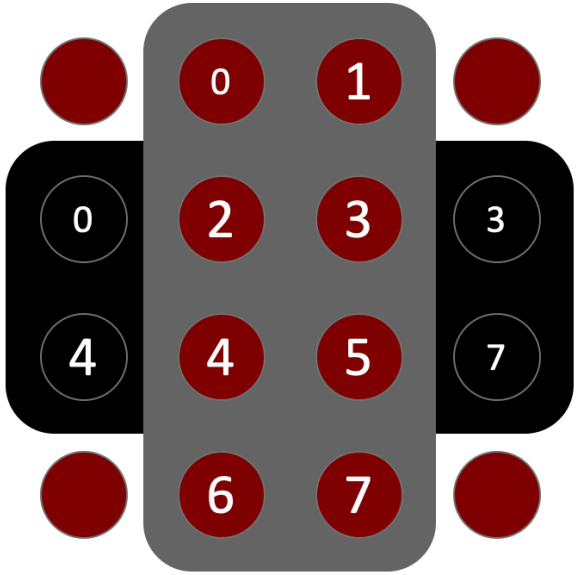
- To communicate together mpi-processes need identifiers: **rank = identifying number**
- all distribution decisions are based on the ***rank***
  - i.e., which process works on which data



Communicators

Communicators do not need to contain all processes in the system

Every process in a communicator has an ID called as “rank”



When you start an MPI program, there is one predefined communicator

`MPI_COMM_WORLD`

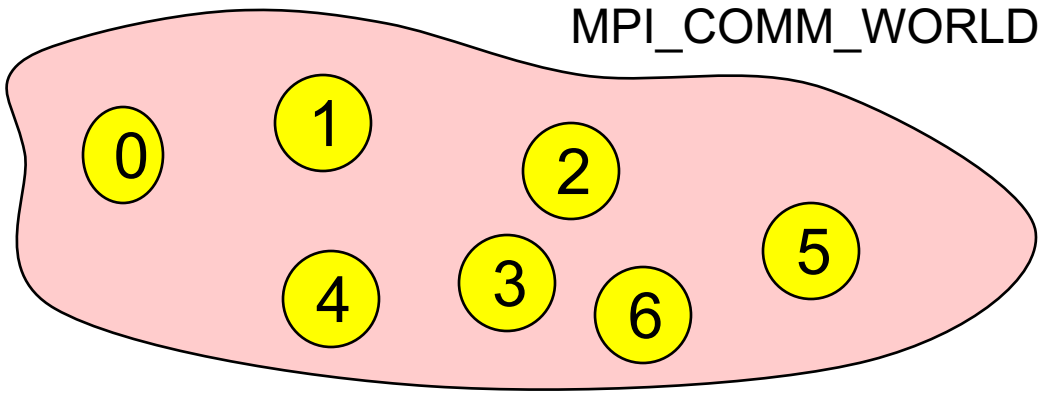
The same process might have different ranks in different communicators

Simple programs typically only use the predefined communicator

`MPI_COMM_WORLD`

Communicator MPI_COMM_WORLD

- All processes of an MPI program are members of the default **communicator MPI_COMM_WORLD**.
- MPI_COMM_WORLD is predefined
- Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)



How big is the MPI library?

- Huge (125 Functions) !!
- Good news – you can write useful MPI programs only using 6 basic functions.

A Generic MPI Program

- All MPI programs have the following general structure:

  include MPI header file

  variable declarations

  initialize the MPI environment

  ...do computation and MPI communication calls...

  close MPI communications

A minimal MPI program(c)

```
#include “mpi.h”
#include <stdio.h>
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  printf(“Hello, world!\n”);
  MPI_Finalize();
  Return 0;
}
```

Lecture 07: Message Passing Interface

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d"
           " out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

MPI header file

Lecture 07: Message Passing Interface

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d"
           " out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

MPI header file

MPI environment initialization, all of MPI's global and internal variables are constructed. For example, a communicator is formed around all of the processes that were spawned, and unique ranks are assigned to each process.

Lecture 07: Message Passing Interface

MPI_Comm_size returns the size of a communicator. Here, MPI_COMM_WORLD encloses all of the processes, so this call should return the amount of processes that were requested for the job.

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d"
           " out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Lecture 07: Message Passing Interface

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d"
           " out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

MPI_Comm_rank returns the rank of a process in a communicator. Ranks are incremental starting from zero and are primarily used for identification purposes during send/receive.

Lecture 07: Message Passing Interface

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_n

    // Print off a hello world message
    printf("Hello world from processor
           " out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

MPI_Get_processor_name obtains the actual name of the processor on which the process is executing.

Compiling MPICH Program

- Regular applications:
  - `gcc mpi_hello_world.c -o mpi_hello_world`
- MPI applications
  - `mpicc mpi_hello_world.c -o mpi_hello_world`

Running MPICH Program

- Regular applications

```
./mpi_hello_world
```

- MPI applications (running with 16 processes)

```
mpiexec -n 16 ./mpi_hello_world
```

Running MPICH Program on a Cluster

- If you are running MPI programs on a cluster of nodes, you will have to set up a host file (named machinefile in our earlier demo).
- The host file contains names of all of the nodes on which your MPI job will execute.

- We earlier setup a host file named, machinefile, with following info

```
slave1:4  # this will spawn 4 processes on slave1
master:2  # this will spawn 2 processes on master
```

- and executed it using

```
mpiexec -n 8 -f machinefile ./mpi_hello
```

Running MPICH Program on a Cluster

- As shown, the MPI program was launched across all of the hosts in machinefile.
- Each process was assigned a unique rank, which was printed off along with the process name.

- The output of the processes is in an arbitrary order since there is no synchronization involved before printing.

Messages

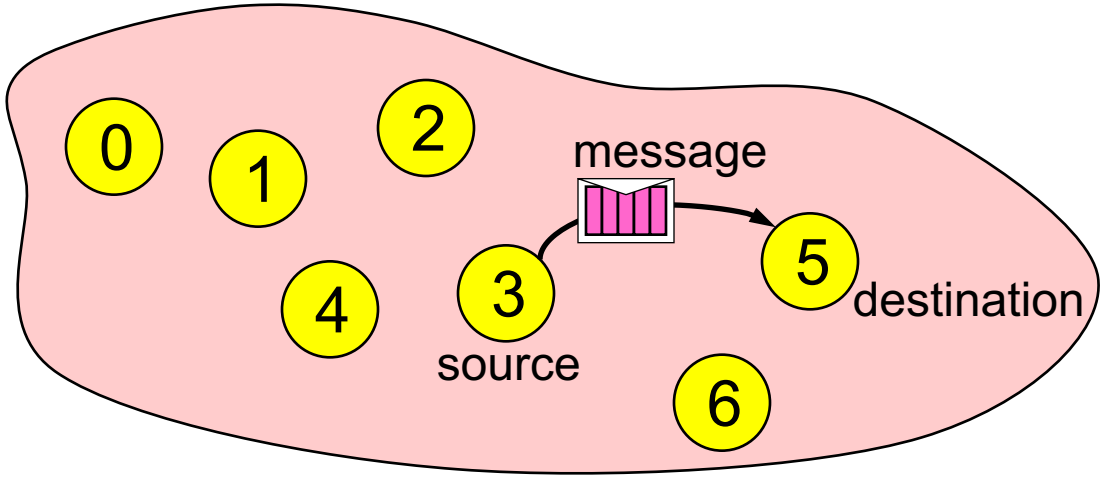
- A message contains a number of elements of some particular datatype.

Example: message with 5 integers

2345	654	96574	-12	7676

Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator, e.g., MPI_COMM_WORLD.
- Processes are identified by their ranks in the communicator.



Point to Point Communication

- Communication is done using send and receive operations among processes.
  - To send a message, sender provides the rank of the process and a unique *tag* to identify the message.
  - The receiver can then receive a message with a given tag (or it may not even care about the tag), and then handle the data accordingly.
  - Two basic (and simple) functions, MPI_Send and MPI_Recv

Data Communication in MPI

- Communication requires the following information:
  - Sender has to know:
    - Whom to send the data to (receiver's process rank)
    - What kind of data to send (100 integers or 200 characters, etc)
    - A user-defined "tag" for the message (think of it as an email subject; allows the receiver to distinguish different messages)
  - Receiver "might" have to know:
    - Who is sending the data (OK if the receiver does not know; in this case sender rank will be `MPI_ANY_SOURCE`, meaning anyone can send)
    - What kind of data is being received (partial information is OK: I might receive *up to* 1000 integers)
    - What the user-defined "tag" of the message is (OK if the receiver does not know; in this case tag will be `MPI_ANY_TAG`)

MPI Send

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

- The target process is specified by `dest` and `comm`.
- The data, its type and amount is described by (`data`, `count`, `datatype`).
- `tag` is a user-defined “identifier” for the message

- When this function returns, the data has been delivered to the system and the buffer can be reused.
  - The message may not have been received by the target process.

MPI Receive

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

- The sending process is specified by `source` and `comm`.
- The receiving data buffer and its type is defined by `data` and `datatype`.
- Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

- Waits until a matching (on **`source`**, **`tag`**, **`comm`**) message is received from the system, and the buffer can be used.

Elementary MPI datatypes

- MPI Datatype is very similar to a C datatype
  - `int` → `MPI_INT`
  - `double` → `MPI_DOUBLE`
  - `char` → `MPI_CHAR`
- More complex datatypes are also possible:
  - E.g., you can create a structure datatype that comprises of other datatypes → a char, an int and a double.

- The "count" in `MPI_SEND` and `MPI_RECV` refers to how many datatype elements should be communicated

Simple Communication in MPI

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

int main(int argc, char ** argv)
{
    int rank;
    char data[] = "Hello World";

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 11, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 11, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    printf("I am a slave and I received %s message
    from my master", data);

    MPI_Finalize();
    return 0;
}
```

So far so good but …

- Let's review the **`MPI_RECV`** function

```
MPI_RECV(buf, count, datatype, source, tag, comm, status)
```

- The receiver may not know the rank of source, the tag the receiver used and the *count* of elements sent by receiver.

- What can the receiver do?

Wildcards

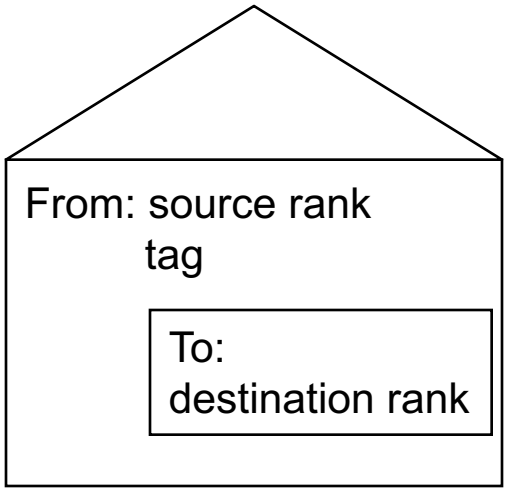
- Receiver can wildcard.
- To receive from any source — source = MPI_ANY_SOURCE
- To receive from any tag — tag = MPI_ANY_TAG
- The count variable is the upper-bound on the number of elements received, it doesn't specify *how many were actually received*.

```
MPI_RECV(buf, SOME_MAX_NUMBER, datatype,
MPI_ANY_SOURCE, MPI_ANY_TAG, comm, status)
```

The MPI_Status structure

- Envelope information is returned from MPI_RECV in *status*.

status.MPI_SOURCE
status.MPI_TAG
count via MPI_Get_count()



The MPI_Status structure

- `status` contains further information:
  - Who sent the message (can be used if you used `MPI_ANY_SOURCE`)
  - How much data was actually received
  - What tag was used with the message (can be used if you used `MPI_ANY_TAG`)
  - `MPI_STATUS_IGNORE` can be used if we don't need any additional information

The MPI_Status structure

- The MPI_Recv operation takes the address of an MPI_Status structure as an argument (which can be ignored with MPI_STATUS_IGNORE).

- If we pass an MPI_Status structure to the MPI_Recv function, it will be populated with additional information about the receive operation after it completes.

The MPI_Status structure

- The three primary pieces of information include:
  - **The rank of the sender**. The rank of the sender is stored in the MPI_SOURCE element of the structure. That is, if we declare an MPI_Status stat variable, the rank can be accessed with stat.MPI_SOURCE.
  - **The tag of the message**. The tag of the message can be accessed by the MPI_TAG element of the structure (similar to MPI_SOURCE).
  - **The length of the message**. The length of the message does not have a predefined element in the status structure. Instead, we have to find out the length of the message with MPI_Get_count.

The MPI_Status structure

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

- In MPI_Get_count, the user passes the MPI_Status structure, the datatype of the message, and count is returned. The count variable is the total number of datatype elements that were received.

```
const int MAX_NUMBERS = 100;
int numbers[MAX_NUMBERS];
int number_amount;
if (world_rank == 0) {
    // Pick a random amont of integers to send to process one
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

    // Send the amount of integers to process one
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
    MPI_Status status;
    // Receive at most MAX_NUMBERS from process zero
    MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0, 0, MPI_COMM_WORLD,
             &status);

    // After receiving the message, check the status to determine
    // how many numbers were actually received
    MPI_Get_count(&status, MPI_INT, &number_amount);

    // Print off the amount of numbers, and also print additional
    // information in the status object
    printf("1 received %d numbers from 0. Message source = %d, "
           "tag = %d\n",
           number_amount, status.MPI_SOURCE, status.MPI_TAG);
}
```

MPI_Probe

- Instead of posting a receive and simply providing a really large buffer to handle all possible sizes of messages (as we did in the last example), you can use MPI_Probe to query the message size before actually receiving it.

```
MPI_Probe(
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Status* status)
```

An example

```
// Probe for an incoming message from process zero
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

// When probe returns, the status object has the size and other
// attributes of the incoming message. Get the message size
    MPI_Get_count(&status, MPI_INT, &number_amount);

// Allocate a buffer to hold the incoming numbers
    int* number_buf = (int*)malloc(sizeof(int) * number_amount);

// Now receive the message with the allocated buffer
    MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

MPI_Isend

Performs a nonblocking send

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

`buf`	starting address of buffer
`count`	number of entries in buffer
`datatype`	data type of buffer
`dest`	rank of destination
`tag`	message tag
`comm`	communicator
`request`	communication request (out)

Non-blocking Send

- Similar to MPI_Send but has an additional request parameter
  - request, which is a handle that is used when checking if the operation has finished

MPI_Irecv

Performs a nonblocking receive

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

`buf`	starting address of buffer (out)
`count`	number of entries in buffer
`datatype`	data type of buffer
`source`	rank of source
`tag`	message tag
`comm`	communicator
`request`	communication request (out)

Non-blocking Receive

- Similar to MPI_Receive but has an additional request parameter and no status parameter
  - request, which is a handle that is used when checking if the operation has finished

Wait routines

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

Waits for `MPI_Isend` or `MPI_Irecv` to complete

`request`	request (in), which is out parameter in `MPI_Isend` and `MPI_Irecv`
`status`	status (out)

`MPI_Waitall`	waits for all given communications to complete
`MPI_Waitany`	waits for any of given communications to complete
`MPI_Test`	tests for completion of send or receive
`MPI_Testany`	tests for completion of any previously initiated communication

```
/* receive termination message asynchronously ... */
MPI_Irecv(buffer, 10, MPI_CHAR, 0, 123, MPI_COMM_WORLD, &request);

MPI_Test(&request, &flag, &status);
while (!flag)
 {
    /* Do some work ... */
    MPI_Test(&request, &flag, &status);
 }
```

Collective Communications

- There are many cases where processes may need to communicate with everyone else.
  - For example, when a master process needs to broadcast information to all of its worker processes.

- MPI can handle a wide variety of these types of *collective* communications that involve all processes.

Collective Communication

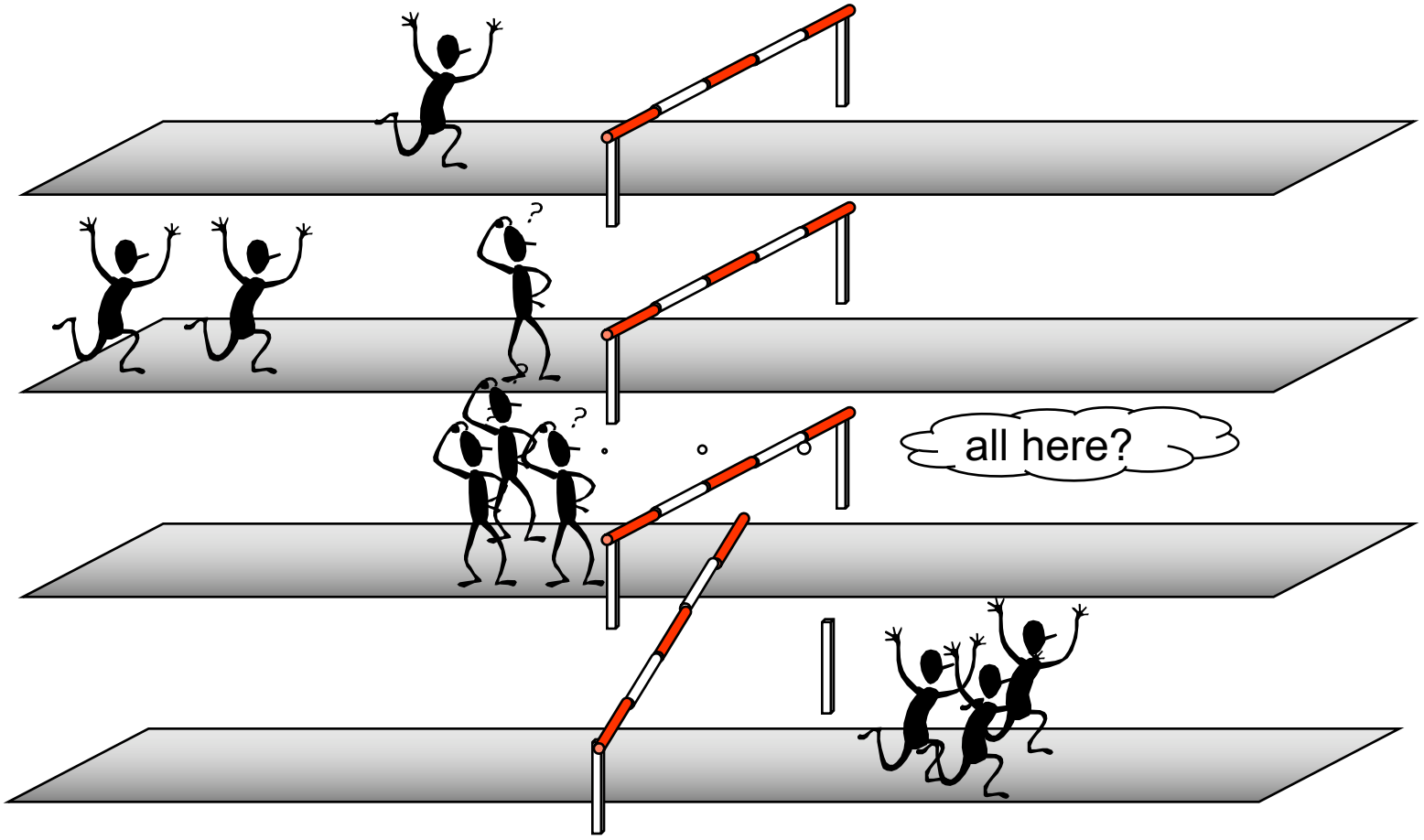
- point-to-point communication- between two processes.

- Collective communication is a method of communication which involves participation of **all** processes in a communicator.

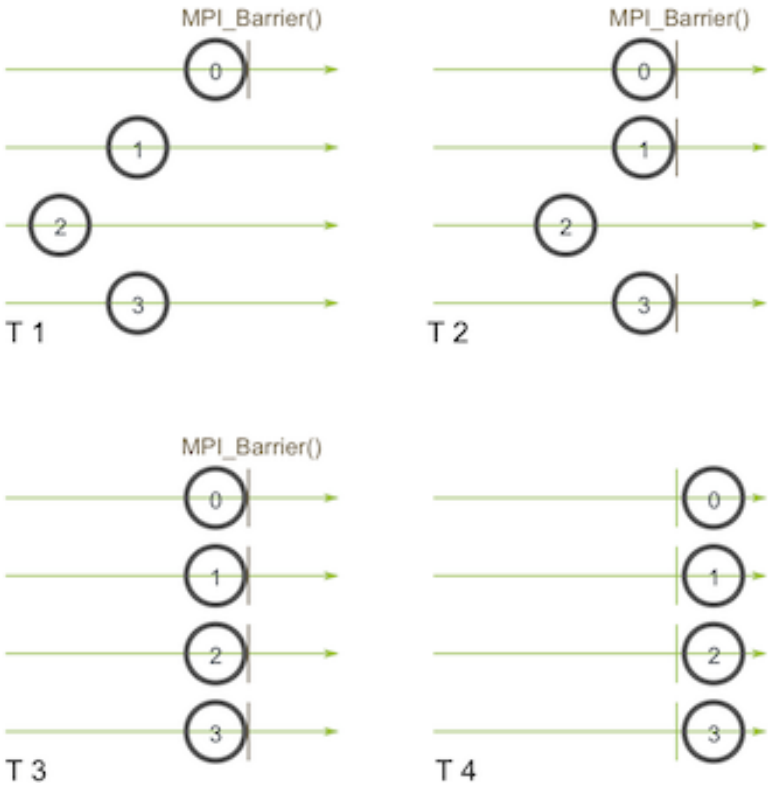
Collective Communication

- Examples:
  - Barrier synchronization.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

Barrier Synchronization



MPI_Barrier(MPI_Comm communicator)



- the function forms a barrier, and no processes in the communicator can pass the barrier until all of them call the function.

Syntax for MPI_BARRIER

C

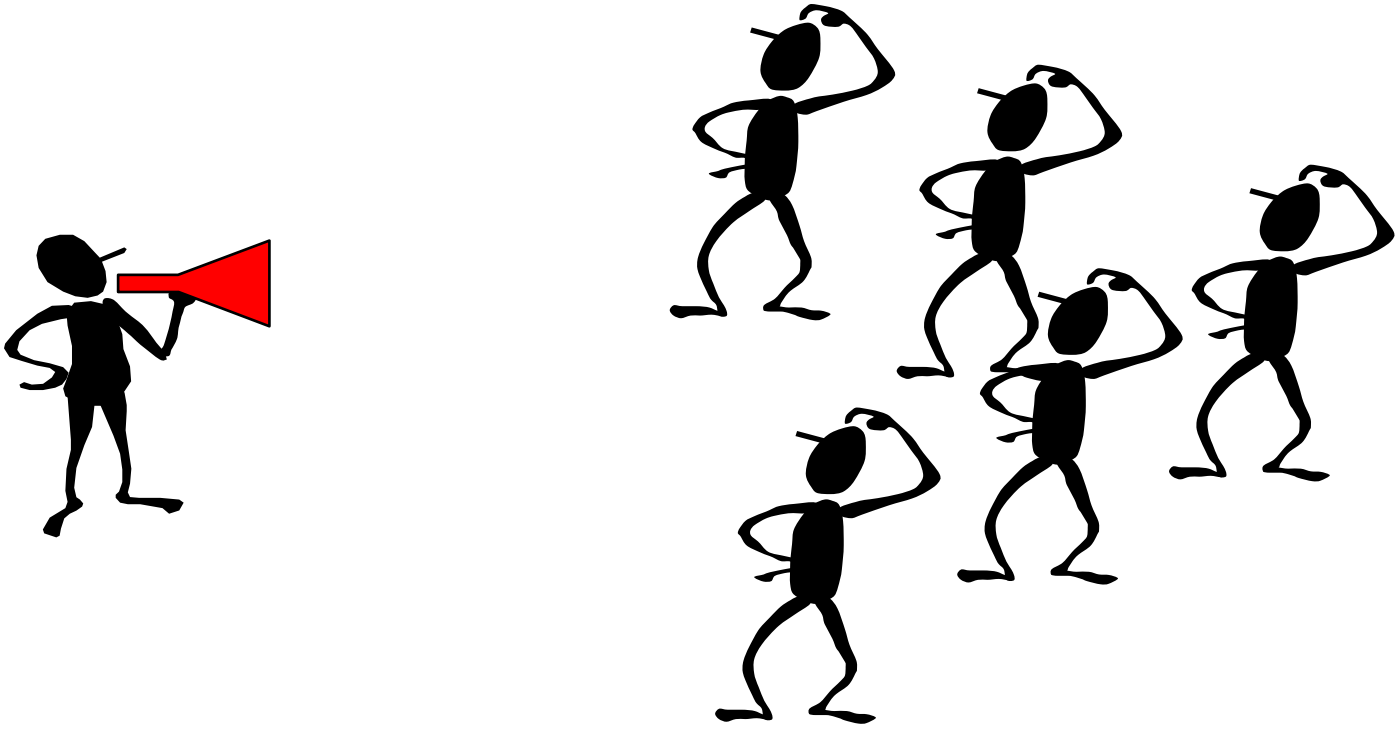
```
int MPI_Barrier(MPI_Comm comm)
```

where:

MPI_Comm	is an MPI predefined structure for communicators, and
comm	is a communicator.

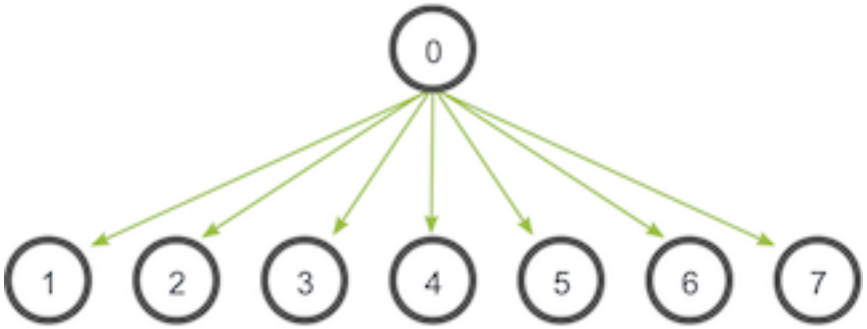
Broadcast

- A one-to-many communication.



Broadcasting with MPI_Bcast

- During a broadcast, one process sends the same data to all processes in a communicator.



Broadcasting with MPI_Bcast

C

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

where:

buffer	is the starting address of a buffer,
count	is an integer indicating the number of data elements in the buffer,
datatype	is an MPI defined constant indicating the data type of the elements in the buffer,
root	is an integer indicating the rank of broadcast root process, and
comm	is the communicator.

Broadcasting with MPI_Bcast

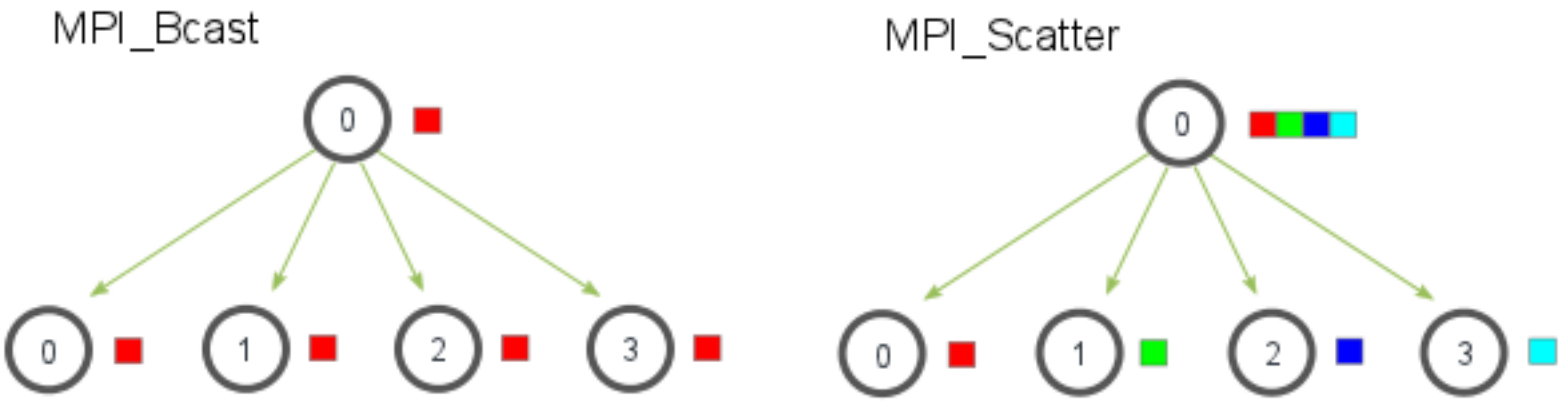
- Although the root process and receiver processes do different jobs, they all call the same MPI_Bcast function.
  - When the root process calls MPI_Bcast, the data variable will be sent to all other processes.
  - When all of the receiver processes call MPI_Bcast, the data variable will be filled in with the data from the root process.

Broadcasting with MPI_Bcast

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size;
  MPI_Status status;
  int root = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == root)
  {
    strcpy(message, "Hello, world");
  }
  MPI_Bcast(message, 13, MPI_CHAR, root, MPI_COMM_WORLD);
  printf( "Message from process %d : %s\n", rank, message);

  MPI_Finalize();
}
```

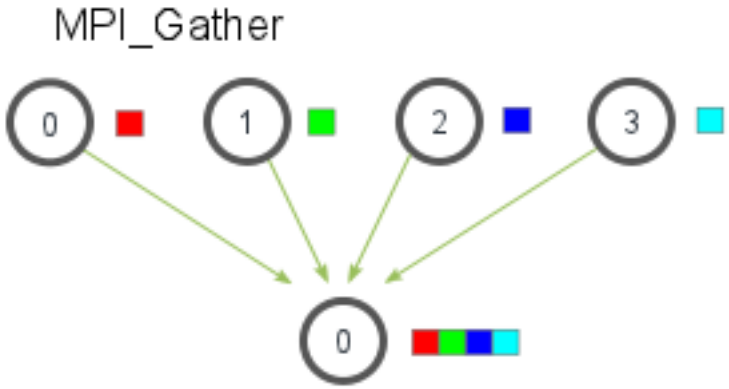
MPI_Scatter

- MPI_Scatter is a collective routine that is similar to MPI_Bcast
- MPI_Bcast sends the *same* piece of data to all processes while MPI_Scatter sends *chunks of an array* to different processes.



MPI_Gather

- MPI_Gather is the inverse of MPI_Scatter. Instead of spreading elements from one process to many processes, MPI_Gather takes elements from many processes and gathers them to one single process.



MPI_Scatter prototype

```
MPI_Scatter(
    void* sbuf,
    int scount,
    MPI_Datatype stype,
    void* rbuf,
    int rcount,
    MPI_Datatype rtype,
    int root,
    MPI_Comm comm)
```

sbuf	is the address of the send buffer,
scount	is the number of elements to be sent to each process,
stype	is the data type of the send buffer elements,
rbuf	is the address of the receive buffer,
rcount	is the number of elements in the receive buffer,
rtype	is the data type of the receive buffer elements,
root	is the rank of the sending process, and
comm	is the communicator.

MPI_Gather prototype

```
MPI_Gather(
    void* sbuf,
    int scount,
    MPI_Datatype stype,
    void* rbuf,
    int rcount,
    MPI_Datatype rtype,
    int root,
    MPI_Comm comm)
```

`sbuf`	is the starting address of the send buffer,
`scount`	is the number of elements to be sent,
`stype`	is the data type of send buffer elements,
`rbuf`	is the starting address of the receive buffer,
`rcount`	is the number of elements for any single receive,
`rtype`	is the data type of the receive buffer elements,
`root`	is the rank of receiving process, and
`comm`	is the communicator.