

**High Performance Computing for Scientists and Engineers**  
**Prof. Somnath Roy**  
**Department of Mechanical Engineering**  
**Indian Institute of Technology, Kharagpur**

**Module – 03**  
**Message Passing Interface (MPI)**  
**Lecture – 21**  
**Communication using MPI**

Welcome to the class of High-Performance Computing for Scientists and Engineers and we are in our 3rd module of this course which is Message Passing Interface or MPI programming. This is the 21st lecture of this course; this lecture is on Communication using MPI. If we can remember in our previous lectures, we have discussed the essential features of MPI which are required to start MPI programming.

We have also looked into a sample MPI program, hello world that shows how multiple processors can be assigned in a distributed memory parallel computing paradigm to write hello world on their own on the screen. So, we have looked into the hello world program in the last class, and we have noted that for initializing a parallel synchronized environment across the processors; we need an MPI initialization command. This MPI\_init is a function for C program and it is a subroutine for Fortran program. So, all the MPI calls like MPI\_init come as functions in the C program and come as subroutines in Fortran programs. However, we need to call a number of MPI functions to write an MPI program. We have seen that there are 6 essential MPI calls by which we can parallelize any program, and there are also some more MPI calls by which we get more flexibility in writing parallel programs.

These 6 functions are very important and these 6 functions are MPI\_init (initiates the parallel MPI parallel environment across the processors), MPI\_finalize (MPI\_finalize terminates that environment). MPI\_comm\_size (which tells us what is the size of the calculation, how many processors are involved), MPI\_comm\_rank (which tells us what is the rank of the particular processor in that MPI calculation) and MPI\_send, MPI\_receive. Another important thing we have also looked into is the communicator which clubs all the processors working in the environment and this MPI communicator is by default called MPI\_COMM\_WORLD.

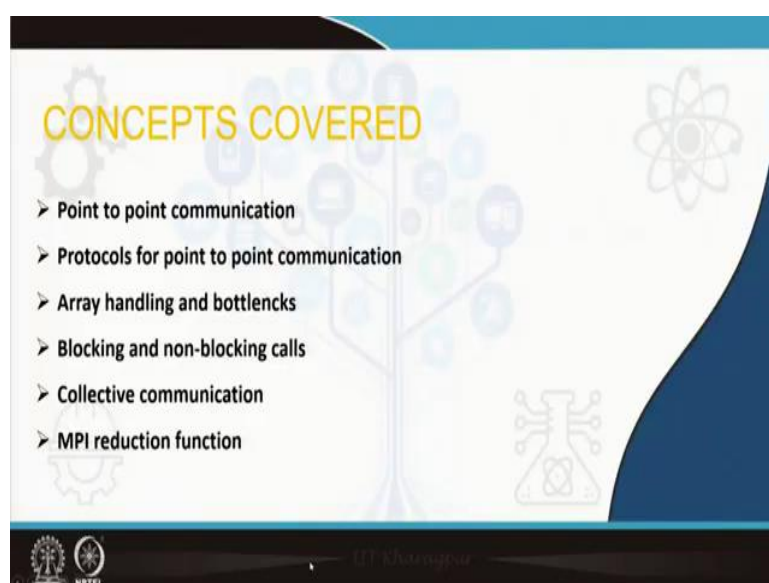
Virtually this is a collection of all the processors which are connected through the network cable and executing the particular parallel MPI program together. This is extremely important and we will see in many cases when you try to communicate across the processors, when you

try to synchronize certain things; we need to mention about this communicator; because this communicator `MPI_COMM_WORLD` which is grouping of these processors which are working in the MPI mode; they will ensure that these processors are working together.

Any message being passed goes through the network which connects these processors. So, if you think of a shared memory program or if you think of a synchronization sequential program, we need to take care of the processors only. But here we need to take care of the fact that there is a network cable or there is a connectivity across these processors, and you need to do certain things using that network cable or the connectivity. I think a better way is to mention it as connectivity because we are looking into software right now.

Now message passing interface or MPI is developed to facilitate certain work which is obvious from its name which is message passing. So, within that communicator, is a group of the processors working together in the distributed memory algorithm combining the connectivity and interconnect between them. So, within that communicator if we have to transfer certain messages, we need to use some of the protocols.

If we have a background on sequential programming or shared memory programming these protocols are very new to us, because this is the first time when you are asking one processor to communicate with another processor, these things we have not seen till now in our parallel computing discussions. So, we will focus on these communications right now. (Refer Slide Time: 05:08)



We will look on the key aspects which are point to point communication and collective communication. These are two modes or two models of communication in MPI. We will look into protocols for point to point communication, array handling in point to point communication.

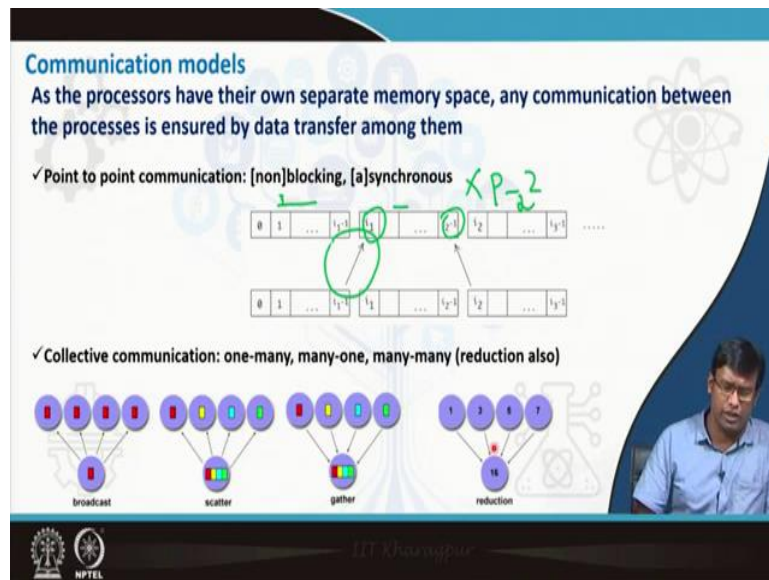
Point to point communication basically means one processor is communicating with another processor. We will see how it can send large arrays from one processor to another processor, how these communications are done, and, what is a blocking and non-blocking mode of communication in this .

Then we will see something called collective communication, that is when one processor is communicating with all the processors in the group or in the communicator. In the communicator group there are multiple processors, when you club them, combining the interconnectivity we call it a communicator. So, within the communicator if one processor tries to send something to all to the entire communicator and all the processors present there and this is through a collective communication.

We will look into MPI reduction functions. Reduction functions are very important in scientific computing. We have discussed that if there is some calculation which has to be done parallelly by multiple processors and each processor comes up with the local value, and finally, all these local values are to be combined to get a global value, reduction is the right operation for that.

We have seen in open MP, reduction is a very optimized communication, we will also see for MPI there are similar reduction functions. But now these reductions are not in shared memory; these reductions are happening across distributed memory. That means, different computers are sending data to one computer who is collecting all and processing the global information out of that, we will see about reduction functions here well.

(Refer Slide Time: 07:15)



So, as the processors have their own separate memory space, any communication between the processor is ensured by data transfer among them. This is the basic fundamental concept of communication in distributed memory processing, that is there are multiple processors and each processor has its own memory space. So, they cannot share any data in between them through the memory connectivity. That means, memories are disjoint and memories cannot communicate with each other.

Memories are connected in the motherboard and memory cannot send any instruction; only the processor can access memory and process some data. So, if there is a requirement that one processor's address space has certain elements which is required by another processor, this has to be communicated in between the processors. Again, we come back to our money sharing example, that I have 100 rupees note in my money bag and somebody needs that money, I can give that money to him. But the money cannot go directly from my pocket to his pocket; I have to take the money and send it to him.

Similarly, information cannot directly go from one memory to another memory, because these are the disjoint memory spaces. The processor has to identify that piece of information and can do a data transfer across the network cable or across the interconnect switch and send the data to the processor, who wants that particular information.

So, any communication in a distributed memory system can only be achieved through a data transfer process. There are two models of data transfer. One is point to point communication

which can be blocking or non-blocking, synchronous or asynchronous. What is point to point communication? We see that these are disjoint memory spaces attached to different processors. These are locations of memory in the different processors.

Now  $i_1$  to  $i_2 - 1$ ; this part of the memory is attached to some other processor 1 say. So, let me write the processor numbers here that processor 0 is attached to this memory space, processor 1 is attached to this memory space and processor 2 is attached to this memory space. So, now, processor 0 needs this information which is given to processor 1. This information is required by processor 0. So, it has to get this data or it has to do a data transfer from processor 1.

Similarly, this information says this is required by processor 2. So, it also needs to take this information from processor 1. Processor 0 needs to talk to processor 2; and processor 2 needs to talk to processor 1. Processor 0 needs some data from processor 1's memory space, processor 2 also needs some data from processor 1's memory space.

They have to distinctly talk to processor 1 and ask for this data. So, in a network system one point is one processor we can think of like the processors are connected through a network and one of the points is one of the processors. So, one point is to send data to another point and it has to receive it, and, again there is another point which needs its data from the other point.

So, point to point communication; means, one point will take data from the other. When processor 0 takes data from processor 1, processor 2 does not know about that, that this is in between these processors. Similarly, when processor 2 takes data from processor 1, processor 0 does not know about that; this is in between the other processors. This is very individual data transfer among two individual processors and we call them point to point communication.

The other one is collective communication which is one to many, many to one or many to many; this also includes reduction. What is that? That one processor will send data to all other processors or each processor has some amount of data and one will collect data from all of them or each processor has some amount of data; this data will be combined and will be given to all the processors. So, this is not transferred across two individual processors. This transfer is visible to all the processors and each processor gets the effect of this data transfer. So, this is called a collective communication.

First one was like I want to give somebody money; nobody else will know that I will do an account transfer and money will go from my account to his account fine. The next one is when

our government announces some plan and everybody gets some money in its Jan-Dhan account, everybody is getting money in his account; who are members of that. So, it is a collective communication from governments account to everybody's account money is coming. It is not a point to point communication and also everybody is aware of the fact. So, when collective communication happens, the address space of all the processors are aware that this particular information is being broadcasted to all other processors from one particular processor or one processor is gathering data from this particular processor.

There are few examples like broadcast when data from one processor is given to all the processors. Scatter, when there is data in one processor and part of it goes to all other processors. Gather, when different data is there in different processors and everything is combined and received by one processor. This can be done by a single processor or it can be done by many processors, so at the same time multiple processors can get a copy of this combined data.

There can be reduction when each processor has its own local value, as we can see for processor 0 has 1, processor 1 has 3, processor 2 has 5, processor 4 has 7 and then everything is combined, summed up and processor 0 gets summation of that which is 16. So, it is like a reduction operation, this is also a collective communication.

(Refer Slide Time: 14:56)

**Point to point communication- message format**

A message is sent/ received as a packet of data, this message has sender's/receiver's address, datatype and size, tag etc.  
 Sending of message is incomplete unless this is received!  
 Sent message must match the received message, as they are executed by two different processes.

Every message (document) has :

Sender End	Receiver End
<ul style="list-style-type: none"> <li>A destination address (rank, like a business address)</li> <li>A message location (starting address, like the document's location)</li> <li>A message datatype (what is being sent)</li> <li>A message size (how big is it in datatype units)</li> <li>A message tag</li> </ul>	<ul style="list-style-type: none"> <li>Sender's address (rank, like another business address)</li> <li>A receiving location (that cabinet in the office of so-and-so)</li> <li>Compatible datatype and size combo in order to fit</li> <li>Matching tag</li> <li>status</li> </ul>

*Handwritten notes in green:*  
 - Under "A destination address": *sender's memory*  
 - Under "A message location": *sender's memory*  
 - Under "Sender's address": *receiver's memory*  
 - Under "A receiving location": *receiver's memory*

NPTEL

We will first focus on point to point communication, and, point to point communication means one processor is sending data from its local address space because this local address space is

the only memory which is visible to the processor. So, one processor is sending data from its local address space to another processor. How can it send data? It will identify a location in its local address space; pass the data through network cable, the other processor will get a copy of this data.

So, it is like taking data and copying it to somewhere else, like if you are using a file transfer protocol; you copy a file and put it in another computer's memory just like that. You have some data, a copy of that will be taken through the interconnect network and will go to another processor's local memory space. This data will be sent as a packet of data. It is not like any amorphous grains of data randomly sent between them.

It is a packet of data, so, one processor will identify that these are the components of that data which has to be sent; some elements of array or something like that, and, this will go to the next processor. When it is sending it, few things are important. Who is sending it? Well, he must know that to whom it will be sent. I want to transfer money to someone's account; I know the account number otherwise I cannot transfer the money. Similarly, the processor who is sending the data must mention that this data will go to him.

Now, he has to also tell that what is the data type, what is the size, he has to tag the message. Similarly, the processor who is receiving the data must know that this data is coming to him and he must be ready for receiving the data. Because sending is an operation which the processor is performing, similarly receiving is another of operation which the other processor, or the remote processor will perform. So, it is not that one processor will send the data and that is that will surely be received by the other processor. The other processor also has to be able to perform the job of receiving; that means, he will know who is sending the data, he will receive the data from him only. What is the data type, data size; that should match, I mean I am trying to send an array and some other processor is trying to receive an integer, there is a data type mismatch, or, one is trying to send an array and other is trying to receive a scalar that will also not happen, there should be matching of the data type and that size of the data. The tag should match, there is a matching tag in the data and a send is incomplete unless this is received. Well, why is it so? If a processor sends data to another processor, the other processor has to receive it. Now, you get back to the idea of account transfer of money; you transfer money to somebody's account; however, there is an error and the money is not sent. So, the money will get back to your account, money will not be deducted. Then who is processing your account, he has to do some process that the send is incomplete, so, it will return back to this account.

Similarly, one computer who is sending data to another processor, if the data is not received; then the operation is not finished, the processor must be aware that it has been finished. If it is not received then there will be a stalemate in the processor, it initiates some job which has not been finished. So, it cannot do anything else probably or even if it does something that there is an issue there.

So, a sending of data will always be incomplete unless it is received. We can think of another example like you are going on a bus and you give a rupee note to the bus conductor; he will give you back the ticket. If you give him the note and he is not giving you back the ticket or he is not receiving the note, you are stalled, you cannot do anything else; or the reverse is giving you the ticket, but you cannot collect the ticket, he cannot go to the next passenger. So, simply sending an information is not sufficient, any send has to be complemented by a matching receive, this is extremely important when we will look into details. Any processor who is sending the data there has to be a matching receive of the data, and the sent message must match the received message.

Well, now what receiver expects? It is the programmer's job, we mentioned earlier to mention what will be sent and what will be received.

So, programmer has to make it sure that the sending message and receiving message must match in terms of the data type, in terms of the size of the data, in terms of the source and destination; also in terms of the tag that this is written on top of that envelope, this data has some certain header or some message, and, the receiver should know that this is particular data what he is trying to receive.

Every message document(the message which is being passed) therefore, must have two components. One is a sending instruction :a protocol through which one processor will send it; there is certain syntax for that. Another is a receiving protocol: a protocol through which the receiver will take the message and put it somewhere in its local memory. So, there has to be similarly another syntax for receiving the message. The senders end there should be a destination address to whom it is being sent, it is like a business address. What can be an address in an MPI parallel system? It obviously will be the rank of the processor, when a processor is sending data to this particular processor, it will be identified by its rank and destination address means rank of the receiving processor. A message location which data will be sent? So, the processor who is sending it in his memory, what is the location of this data? What is the address

of the first element of that array which will be sent, what is the pointer to that? A message data type (which type of data is being sent) A message size (how big is it in that particular data type unit), and the tag. The receiver will have the sender's address, who is sending it? Receiving processor should know from which rank this message has been sent. What is the receiving location? Once the message comes where should he keep it? So, message location is in the sender's local memory. A space in the receiver's memory in which this data will be accepted. It also requires a compatible data type and size combo that this particular data of this size has been sent. When the processor who is receiving it, he should give a compatible data type, that if he is sending an integer, he should be ready to receive the integer, and, also the size should match.

So, in case this data type and size combo matches with the sending (though the data type may not match), the data transfer will be successful. But, the data type real to real, integer to integer, float to float that also has to be matched for the right nature of the data being accepted.

The data type and size combo must match in order to see that this data has been sent and I can receive the entire data. So, the receiving location will be mentioned and the space in the receiving location where it will be received that should have to also be mentioned. A matching tag, this data comes with a tag from the sender set. The receiver end also should have a matching tag. Especially in a program with high complexity, there can be multiple sending and receiving between a couple two processors. Processor 1 is sending data very often and processor 2 is receiving that data very often. So, when one particular data is coming from a location in the sender's end, that data will be received in a particular location in the receiver's end and that is required by the program.

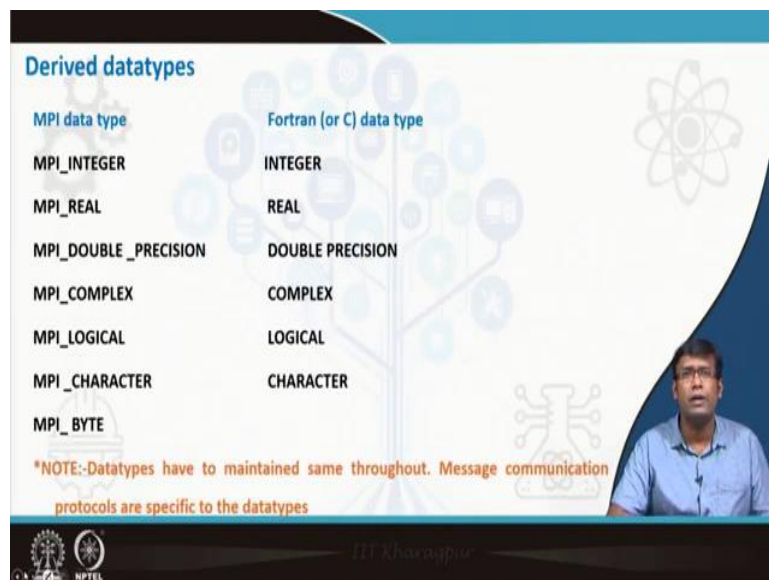
You need to know that this data is coming from the sender's end and this data will be received at this particular location. But if there are multiple sends and receive, there are multiple data coming from the sender's end. Now, each of this data has to be received and located in the designated place by the receiver. Therefore, how will it know which data, which is being sent by the processor, where it should go? That will be identified by the tags. If there are multiple sending of data, multiple instances data is being sent, and the receiver is receiving it; through the tag it will identify that this data is meant for this particular location. So, matching tags is required, like you are getting mails from the same sender, but you identify that these mails are for this particular purpose. Your boss has sent you three emails and one email is regarding a

publication, another email is regarding some information, regarding your last week's work. The third email is regarding some file in your cabinet which he wants readily.

So, all these three emails have three different utilities and you identify which email has to be replied in which manner and what are the relevant jobs corresponding to that email by the email header. Similarly, this tag is like an email header through which the receiver will identify that this data is for this particular location.

The status will identify the correctness of the data transfer. If the data transfer is right you will see certain values in the status message. If the data transfer is wrong you will see garbage, we will look into the status later.

(Refer Slide Time: 27:57)



MPI data type	Fortran (or C) data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	

\*NOTE: Datatypes have to maintained same throughout. Message communication protocols are specific to the datatypes

What is further important here are the data types for message passing. We are aware of this data type in Fortran; INTEGER, REAL, in C it is FLOAT, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER. These are the data types we use for computing in our simple sequential computing environment.

Now, when we are using a message passing interface or MPI we are doing distributed computing, and we are using an architecture which is developed by combining multiple computers together. This can be very heterogeneous, there can be computers with different operating systems or different versions of operating systems, different versions of some other software's and they are connected together. The meaning of data packet, meaning of integer

can vary across the processors. The size associated with the data, how they are stored, how they are accessed, also this data is being sent through a communicator which is not a simple computer.

Therefore, when we are considering about a data in a MPI data transfer; this is a data which is in transit which is being transferred across the processors, and, once a processor receives this data, it must know that it should map to the particular data type; the OS or the compiler is using for its own work.

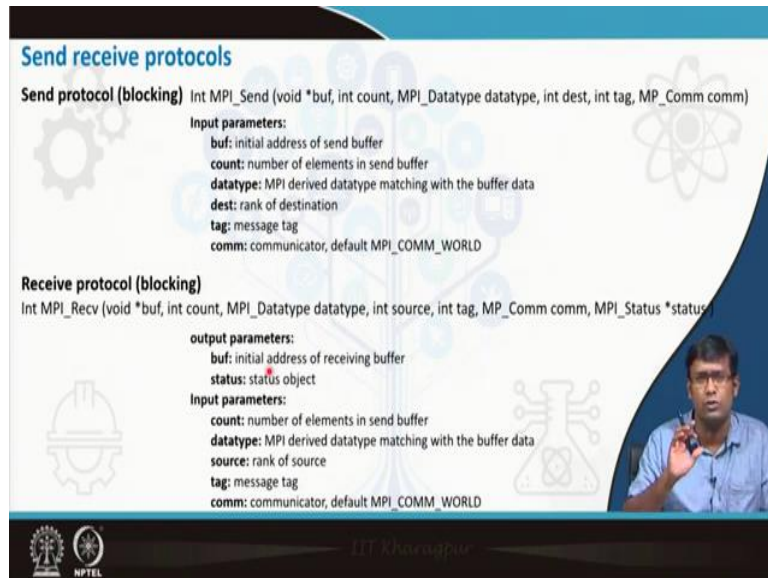
So, the data in transit through the communicator is in nature little different than the data, the compiler considers while working through the program, and therefore, when we are using MPI, a derived data type is used. This data is for data transfer across MPI and for each of the data types in your Fortran or C or C++ program, there is a matching MPI derived datatype.

When data transfer takes place, we need to look into the matching data type. While doing the data transfer, we will not say that integer data will be transferred. We will say that the MPI derived form of integer which is `MPI_integer` will be transferred. Why? Because, this is not like a simple data type which is being addressed by the OS of the computer and the compiler.

This data is a packet of information which is being transferred through MPI protocols. So, there might be heterogeneous systems which are connected through the interconnect and MPI has built a communicator across them. So, the data in transit must follow MPI's own data type. And therefore, we use `MPI_INTEGER`, `MPI_REAL`, `MPI_DOUBLE_PRECISION` or in case of C, we will use `MPI_FLOAT`; corresponding to the Fortran and data C data type. So, when this data type will be transferred, we will use the MPI derived data type. We will not write simple C or Fortran data type here. Message communication protocols are very specific to the data types, if you see that I am transferring a MPI double precision, it will assign bits as per double precision number which will be in transit through the coming an interconnect network using MPI protocol.

It will be different; obviously, than an integer variable which is in transit. So, this is very important that while doing the data transfer, we must mention the MPI data type, not the simple data type coming from Fortran or C compiler.

(Refer Slide Time: 32:07)



**Send receive protocols**

**Send protocol (blocking)** `int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MP_Comm comm)`

**Input parameters:**

- buf:** initial address of send buffer
- count:** number of elements in send buffer
- datatype:** MPI derived datatype matching with the buffer data
- dest:** rank of destination
- tag:** message tag
- comm:** communicator, default MPI\_COMM\_WORLD

**Receive protocol (blocking)** `int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MP_Comm comm, MPI_Status *status)`

**output parameters:**

- buf:** initial address of receiving buffer
- status:** status object

**Input parameters:**

- count:** number of elements in send buffer
- datatype:** MPI derived datatype matching with the buffer data
- source:** rank of source
- tag:** message tag
- comm:** communicator, default MPI\_COMM\_WORLD

NPTEL IIT Kharagpur

The send and receive protocols are important now. So, we know that when sending a data, we need to find out which data is being sent, what is the size of data, what is the data type and what is the MPI data type? We cannot write integer data that is being sent, we have to write if the variable is integer, when that will be sent, it will be during movement considered as MPI integer. Also, what is the destination, the tag and the communicator, that is usually MPI\_COMM\_WORLD. So, the input parameters will be the buffer (which is the initial address of the sent). The count (number of elements that will be sent), the data type (MPI derived data type matching with the buffer data). Buffer data is the data which has been taken from the address space of the sending processor and to be sent to the receiving processor.

Now, what is the data type of that buffer data, the data in the buffer which will be sent to receive through MPI. There should be a matching data type, if it is integer the MPI data type will be MPI\_INTEGER, if it is float MPI data type it will be MPI\_FLOAT. If it is a double precision data, MPI will be MPI\_DOUBLE\_PRECISION so on. The destination or the rank of the destination processor, the message tag and the communicator which is by default MPI\_COMM\_WORLD if you are using default communicator mostly we will use that.

The receiving protocol will be MPI\_receive void\*, MPI int count, MPI\_Datatype, int source, tag, MPI\_Comm and Status.

This is the protocol; when you write the function, it will give you an integer value, but you have to call this function for doing data transfer. The output of doing a receive protocol is

buffer, that is the receive buffer. There is some buffer through which data has been taken by MPI and as MPI sent is there, the processor sends this data to the receiving processor. As the receiving processor receives it, it puts it in a buffer which is in the local memory of the receiving processor now and that is the output. So, as the data transfer takes place, you get some data in its buffer. Well, another output is status, if it is successful the status is success, we call it MPI success, it is a different type of variable (it is a structure variable and the output is MPI success). So, if it is true then the data transfer process has been successful, then the status will be MPI success.

So, these are the outputs ;the data received (that is the output of receive call) and the status (successful or not). The input parameters are ,while writing this protocol, you have to mention the count; what is the number of elements that will be in the same buffer.

The same number of elements should come in the received buffer. The data type, source from which where it is coming, tag and the communicator. This is called a blocking send and blocking receive; that means, as the processor has sent the data it cannot do anything else till the send is being finished. Similarly, another processor which is receiving the data, as it has started to receive the data it cannot do anything else till the receive is done. The processor's execution will be blocked at this stage, and, in this particular case a send has to be matched by a receive because send is not finished unless matching received command in the destination processor has been executed. Similarly, destination processor receiving is not finished unless the sent source processor has sent the data. So, there has to be a matching in between them otherwise there will be a bottleneck. We will see about these things in the next few lectures.