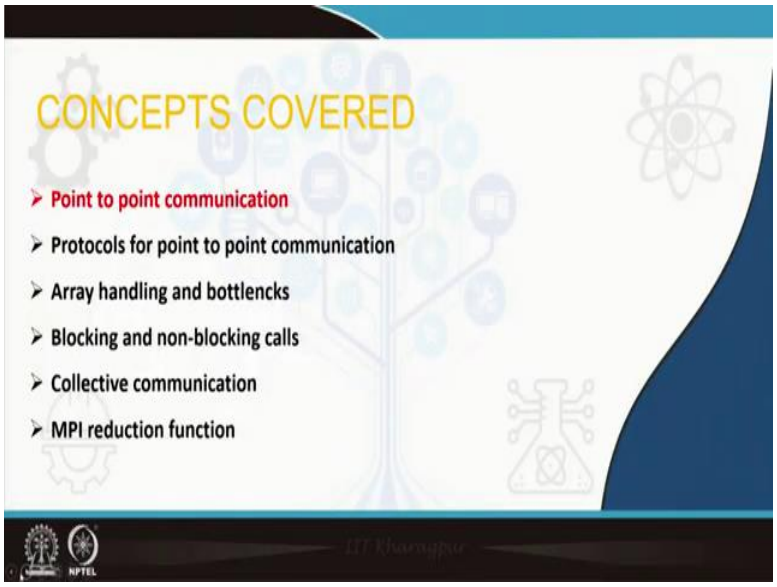
High Performance Computing for Scientists and Engineers

Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 03
Message Passing Interface (MPI)
Lecture – 22
Communication using MPI (continued)

We are in the class of High-Performance Computing for Scientists and Engineers, and we are in the 3rd module of this course which is Message Passing Interface or MPI programming, for distributed memory computing. We are continuing with the lectures on Communications using MPI.

(Refer Slide Time: 00:46)

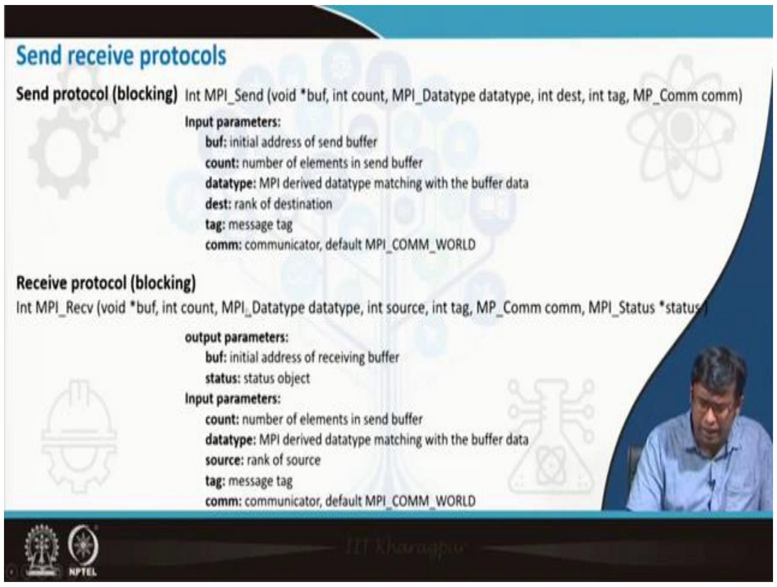


CONCEPTS COVERED

- Point to point communication
- Protocols for point to point communication
- Array handling and bottlencks
- Blocking and non-blocking calls
- Collective communication
- MPI reduction function

We have looked into point to point communications and protocols for point to point communication which is MPI_send and MPI_receive, how one processor can send data to another processor, and through which protocol the other processor can receive data from here.

(Refer Slide Time: 01:03)



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

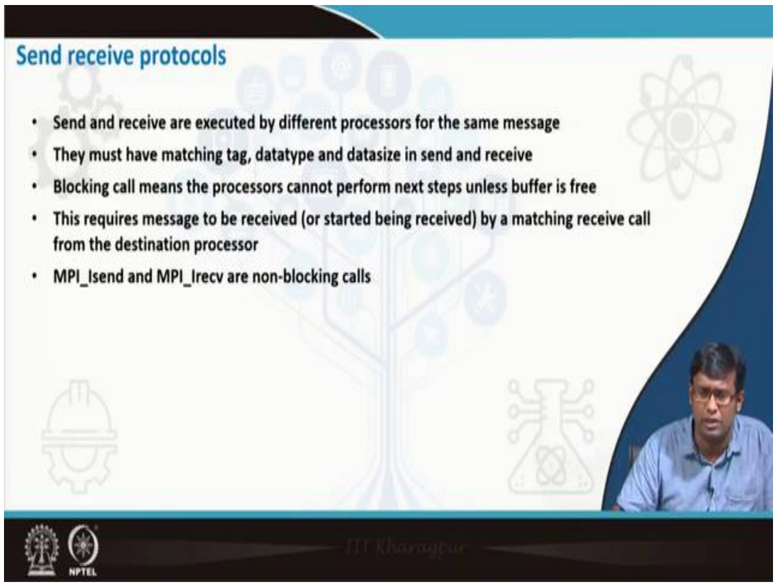
Input parameters:
- **count:** number of elements in send buffer
- **datatype:** MPI derived datatype matching with the buffer data
- **source:** rank of source
- **tag:** message tag
- **comm:** communicator, default MPI_COMM_WORLD

We have looked into the send and receive protocol across the processors, and we saw that data from one processors local memory(each processor has local memory, there is no global memory in MPI distributed memory programming),will be sent to another processors local memory, and, the data from the send buffer will be sent and we have to mention the size of the data, the location of the data, size of the data, the data type, the id of the destination or id of the receiving processor, tag and the communicator.

While receiving it we have to say that where it will be received, the location at the receiving end, and the size of the data, the data type, the source id of the source, there will be a matching tag communicated and the status message will be required. These are called blocking calls because the processors cannot do anything if it has initiated the sending and it cannot do anything unless this send receive is completed.

There are non-blocking calls, there are synchronized and asynchronous calls for data point to point communication. This is called point to point because these communications are very private. One processor sends data to another processor and the other processor receives data from this particular processor, and none of the other processes in the same communicator are aware of that. So, call this to be a private way of sending data and therefore, this is a point to point communication only from one-point data goes to another point.

(Refer Slide Time: 03:01)



Send receive protocols

- Send and receive are executed by different processors for the same message
- They must have matching tag, datatype and datasize in send and receive
- Blocking call means the processors cannot perform next steps unless buffer is free
- This requires message to be received (or started being received) by a matching receive call from the destination processor
- MPI_Isend and MPI_Irecv are non-blocking calls

Some of the important features are :send and receive are executed by different processors for the same message. By now we are aware of the fact that the processor who is sending the data from its local memory is executing the send command and or the send function and the processor who is receiving that data from the sending processor and keeping it in its own local memory is executing the receive command. So, for the same data there has to be a matching send and receive call executed by two different processors.

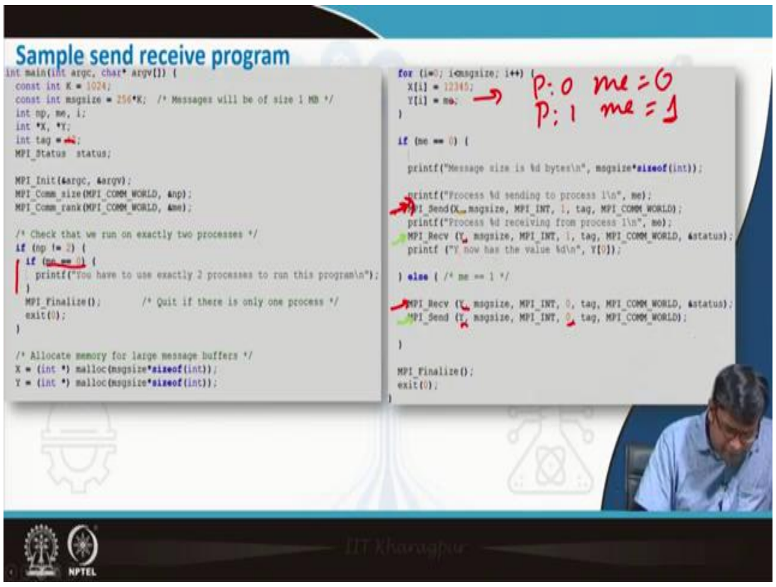
So, for one particular message which will be transferred across two processors, there will be a send, executed by one processor and received by another processor. For send message there has to be two different processors who will execute send and receive, respectively, and there must be a matching of the tag, data type, and data size and send receive. If any of them fails to match there will be a problem in data transfer, it will be incomplete and as these calls are blocking if there are failures the processors cannot do anything. There will be a stalemate case in the execution.

Blocking call means processors cannot perform next steps unless the buffer is free. When the buffer is free? Typically, when the data transfer is complete then the send buffer will be free and the receive buffer is full, so the send buffer is free and the receive buffer is full. This requires messages to be received or started by being received by a matching receive call from the destination processor. Thus, sending processor work is also blocked unless receiving processor has received the message or at least started to receive the message ;in certain cases, if the data is very small, just starting to receive the message might work.

Usually a send executed by one process means that it has put some data from its local buffer to MPI communicators buffer and it will reach the receiver; unless this data transfer is free, the communication buffer is free the send is not complete. When this communication buffer will be free; when the send message is received by the receiving processor, till then this communication is not complete and therefore, the processors are blocked.

If we need to do something in a non-blocking manner ;like data transfer is a different job than doing floating point operations, so, the processors will be allowed to do its next steps till the data is being transferred. Specially for a very large message we do not like to wait for the data transfer rather we ask the processors to do their next steps before finishing up data transfer. There is a way out ,that is MPI_isend an MPI_Irecv they are non-blocking calls; that means, data transfers are not finished, but the processors still keep on doing their own work. We will see some of these examples later as I told you earlier.

(Refer Slide Time: 06:23)



Sample send receive program

```
int main(int argc, char* argv[]) {
  const int K = 1024;
  const int msgsize = 256*K;  /* Messages will be of size 1 MB */
  int np, me, i;
  int *X, *Y;
  int tag = [illegible];
  MPI_Status  status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);

  /* Check that we run on exactly two processes */
  if (np != 2) {
    if (me == 0) {
      printf("You have to use exactly 2 processes to run this program\n");
    }
    MPI_Finalize();       /* Quit if there is only one process */
    exit(0);
  }

  /* Allocate memory for large message buffers */
  X = (int *) malloc(msgsize*sizeof(int));
  Y = (int *) malloc(msgsize*sizeof(int));

  for (i=0; i<msgsize; i++) {
    X[i] = 12345;
    Y[i] = me;
  }

  if (me == 0) {

    printf("Message size is %d bytes\n", msgsize*sizeof(int));

    printf("Process %d sending to process 1\n", me);
    MPI_Send(X, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD);
    printf("Process %d receiving from process 1\n", me);
    MPI_Recv (Y, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
    printf ("Y now has the value %d\n", Y[0]);

  } else { /* me == 1 */

    MPI_Recv (Y, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    MPI_Send (Y, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD);

  }

  MPI_Finalize();
  exit(0);
}
```

P:0 me = 0
P:1 me = 1

So, this is the sample send receive program. Message size is identified as 265* 1024, this is basically for the fact that we want to have 1-megabyte data which has to be sent.

This integer of this message size is x and y .If you see, we start an MPI program through MPI_init, MPI_comm_size MPI_comm_rank. This particular program will run in only two processors. So, MPI_comm_size gives us the size of the processor and if we run it in more than 2 processes then the number of processors will not be equal to 2. So, it will say that you have to run this in exactly 2 processors.

Now onwards, the examples on send receive will consider this particular way of programming because we want one process to send data another process to receive it. If we run this job in multiple processes there will be some haziness, some of the processes will not have some work and it will be difficult for us to identify which one is doing what.

So, as we try to demonstrate only sending from one process and receiving by another process, we will try to run our job in 2 processes. If the number of processors is more than two it will cancel the program and it will come out of the program .

In case we have two processes, we have identified message size of 256*1024. So, we allocate memory for 2 integers x and y of this particular size, and all the elements of x are 12345, all the elements of y are me.

What is me? me is the rank of the process. This is running in 2 processes. So, we can write that for processor 0, me = 0; for processor 1 me = 1. So, x is 12345 for both the processors. For processor 0, me is 0 for processor 1 m = 1 .

Everything in this program will be executed by both the processors, because there is no if else loop here.This particular line if I come here if np != 2, if I run it in more than 2 processors if me = = 0, then it executes this particular printf that you have to use 2 processors.

This will be executed only by one particular processor because we have written that only one processor will execute this line. This is hard coded else. Everything will be executed by all the processors. This is not like openMP that only thread 0 will execute everything unless a parallel construct is given, all the processors will execute all the lines here.

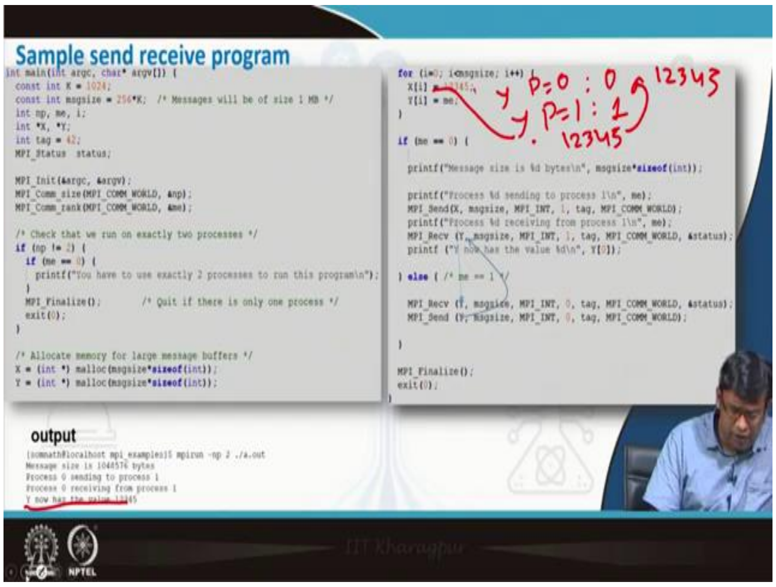
So, now, we ask if me = 0, the process with rank 0, so write that the message size is this process me is sending to process 1, process 0 is sending data to process 1.What it is sending? Sending x which is 12345 of the message size we have mentioned, this is MPI_int, we are sending integer to process 1 with a tag. What is the tag? Tag has an integer value 42 through MPI_comm_world. Then, it will write that process 0 will send something to process 1 and will receive something from process 1. What will it receive? It will receive y from process 1. What will happen at process 1? It will first receive x and is being sent by process 1. What is in x? 12345. Process 1 or if else that is me = 1, it will receive this (the same tag data 42 will match) in the buffer y, y is the location where it will keep it. That y was 1 because me = 1 for process 1. But now that will be overwritten by this data transfer. What is coming through the MPI_send that is being received by MPI_recv,so, instead of one it will be overwritten by 12345. So, all the y elements will be 12345. It will receive this message which is integer from source 0 with a matching tag and the status is there for data communication.

This y which is now 12345 will be sent by process 1 and where it will be sent; it will be sent to process 0. So, process 0 will now receive it because first it will execute process 0 will execute the send, and process 1 will execute the receive. Then, process 1 will execute the sent , and that will be finished once process 0 executes the receive.

So, what will it send? It will send y and this y is already 12345 because x was 12345 which is being received as y is 12345 for all the elements of this message size., and this will go to process 0. Process 0 will receive it as y. So, process 0's y which was 0 earlier, because it is equal to the rank 0. This will be written by y to 12345. So, it will write that y has now the value 12345 if we execute it.

So, let us see what happens here.This send is matched by this particular receive, and which is sent from id 0 to process 1 is being received in the process 1, and process 1 is receiving it from what is been sent by process 0, and this x is being sent to y. So, what was in x? 12345 that is being sent to y, and this y will be sent back to process 0. So, the output will be if we run it in 2 processors, message size is nearly 1 MB. Process 0 sending to process 1, process 0 receiving from process 1, and the received y is 12345.

(Refer Slide Time: 13:29)



Sample send receive program

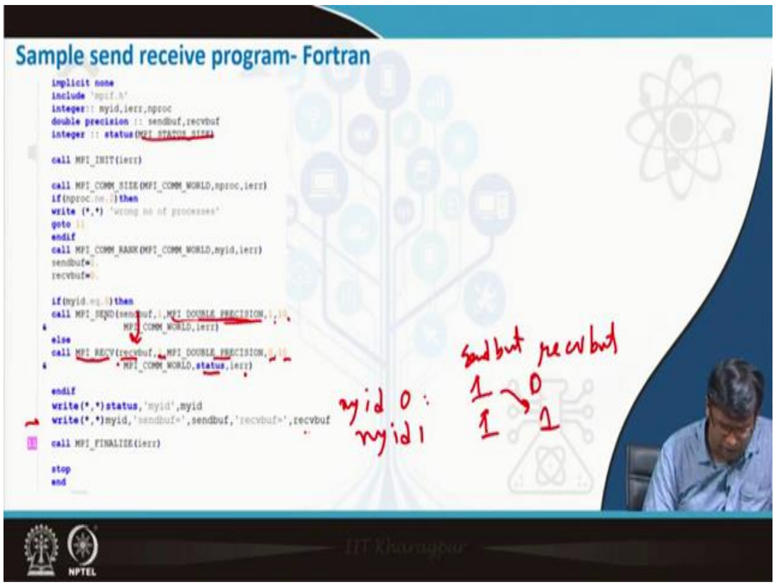
```c
int main(int argc, char* argv[]) {
  const int K = 1024;
  const int msgsize = 256*K;  /* Messages will be of size 1 MB */
  int np, me, i;
  int *X, *Y;
  int tag = 42;
  MPI_Status  status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);

  /* Check that we run on exactly two processes */
  if (np != 2) {
    if (me == 0) {
      printf("You have to use exactly 2 processes to run this program\n");
    }
    MPI_Finalize();        /* Quit if there is only one process */
    exit(0);
  }

  /* Allocate memory for large message buffers */
  X = (int *) malloc(msgsize*sizeof(int));
  Y = (int *) malloc(msgsize*sizeof(int));

  for (i=0; i<msgsize; i++) {
    X[i] = 12345;
    Y[i] = me;
  }

  if (me == 0) {

    printf("Message size is %d bytes\n", msgsize*sizeof(int));

    printf("Process %d sending to process 1\n", me);
    MPI_Send(X, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD);
    printf("Process %d receiving from process 1\n", me);
    MPI_Recv (Y, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
    printf ("Y now has the value %d\n", Y[0]);

  } else { /* me == 1 */

    MPI_Recv (Y, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    MPI_Send (Y, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD);

  }

  MPI_Finalize();
  exit(0);
}
```

output

```
[somnath@localhost mpi_examples]$ mpirun -np 2 ./a.out
Message size is 1048576 bytes
Process 0 sending to process 1
Process 0 receiving from process 1
Y now has the value 12345
```

So, initially y for process 0 was 0, y for process 1 was 1. This y is rewritten as, x came here, this y becomes 12345,and now this has been communicated back to process 0 and its y becomes 12345. So, we will see that the y now has the value 12345. A better way to appreciate this is that you copy this program and try to run it on your own computer with MPI and see what is happening. You can play with the program also.

(Refer Slide Time: 14:10)



Sample send receive program- Fortran

```fortran
      implicit none
      include 'mpif.h'
      integer:: myid,ierr,nproc
      double precision :: sendbuf,recvbuf
      integer :: status(MPI_STATUS_SIZE)

      call MPI_INIT(ierr)

      call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
      if(nproc.ne.2)then
      write (*,*) 'wrong no of processes'
      goto 11
      endif
      call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
      sendbuf=1.
      recvbuf=0.

      if(myid.eq.0)then
      call MPI_SEND(sendbuf,1,MPI_DOUBLE_PRECISION,1,10,
     &              MPI_COMM_WORLD,ierr)
      else
      call MPI_RECV(recvbuf,1,MPI_DOUBLE_PRECISION,0,10,
     &             MPI_COMM_WORLD,status,ierr)

      endif
      write(*,*)status,'myid',myid
      write(*,*)myid,'sendbuf=',sendbuf,'recvbuf=',recvbuf
11    call MPI_FINALIZE(ierr)

      stop
      end
```

Well, so this is a sample send receive program. We will see a similar program in Fortran. We are mainly focusing on c and Fortran because legacy codes in scientific computing are written either in C or in Fortran.

Of course, there are codes written in other languages, specially many open source solvers are written in C++, there are programs in Python, people are coming up heavily with Python programs, programs in Java also, in other languages like MATLAB also. But, a large number of programs are in C and Fortran and C is probably the language which is very universal across different communities ,if you are introduced to numerical methods and scientific programming you know C that is assumed. So, initially focus on C, and we also understand the fact that many legacy codes, many groups have programs written in Fortran. So, scientists from different fields working on computational methods are convergent with the Fortran program and deal with the Fortran program. So, we will put some emphasis in Fortran too.

In Fortran we can see the program starts with mpif .h, in C it is the MPI header is MPI .h, in Fortran it is added that is a Fortran program mpif .h. Now, you have an integer myid ierr(ierr is the error message in the Fortran, in any Fortran call the output is an error message), nproc, and we call a status which is an integer of size MPI_status_size (it is an integer structure it has particular MPI data size). Then, MPI_init initializes, then finds comm size and if the size is != 2, then you write the wrong number of processors; this is designed to run in two processors. Then find the comm rank, and send buffer is 1 for both the processors, in send buffer they have 1 and in receive buffer they have 0. This is an integer only, this is not an array, it is a single number.

Processor 0 will perform send and process whose processor id is not 0 (processor 1) will perform receive. It will send the send buffer of size 1 which is now a double precision variable, so, the data type is MPI_double_precision, and this will be received by processor 1,tag is 10, MPI_comm_world ,ierr. Therefore, processor 1 will execute a receive with the same tag 10, it will get the data in the receive buffer, the data size is 1, data type is double precision ,it will receive, from processor 0 and MPI_comm_world ,status and ierr. We will see what is in status. We said that if it is true, status is something meaningful.

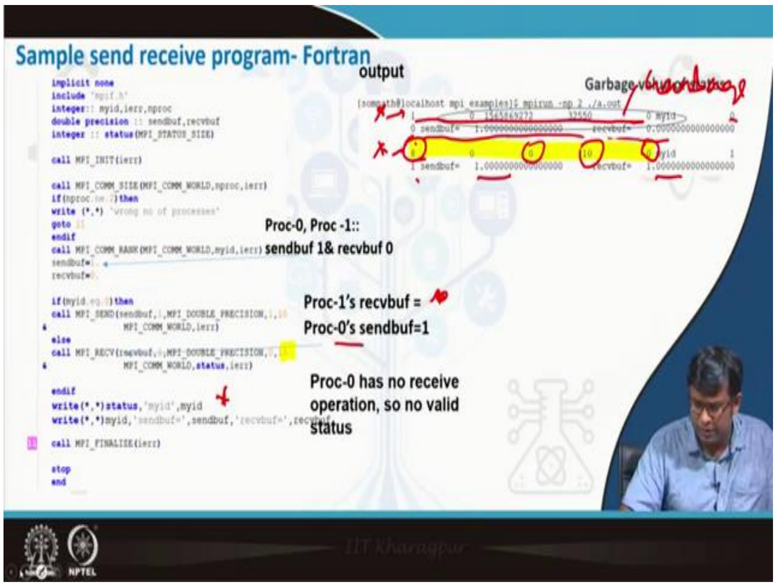
Now, processor 1 only executes the receive command therefore, it has a valid status. Processor 0 does not execute any receive command therefore, there is no status updated in processor 0. It will give us a garbage status, and then it writes what is in the send buffer and receive buffer.

So, let us try to guess apart from the status what will be the next output. My id is 0, rank 0 has send buffer 1, receive buffer 0. My id 1 had initially sent buffer 1, received buffer 0, but my id 0 sent buffer is received as receive buffer by my id 1.

So, it's received buffer will be sent to the buffer of my id which will be 1. So, for my id 0, send buffer is 1, receive buffer is 0; for my id 1 send buffer is 1 ,and receive buffer is the send buffer of my id which is being sent here, so receive buffer will also be 1. So, let us see the output of this program.

So, my id 1 will have two 1s, my id 0 will have 1 and 0. Let us see if we get the same output.

(Refer Slide Time: 18:45)



Sample send receive program- Fortran

```fortran
      implicit none
      include 'mpif.h'
      integer:: myid,ierr,nproc
      double precision :: sendbuf,recvbuf
      integer :: status(MPI_STATUS_SIZE)

      call MPI_INIT(ierr)

      call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
      if(nproc.ne.2)then
      write (*,*) 'wrong no of processes'
      goto 11
      endif
      call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
      sendbuf=1.
      recvbuf=0.

      if(myid.eq.0)then
      call MPI_SEND(sendbuf,1,MPI_DOUBLE_PRECISION,1,10,
     &              MPI_COMM_WORLD,ierr)
      else
      call MPI_RECV(recvbuf,1,MPI_DOUBLE_PRECISION,0,10,
     &              MPI_COMM_WORLD,status,ierr)

      endif
      write(*,*)status,'myid',myid
      write(*,*)myid,'sendbuf=',sendbuf,'recvbuf=',recvbuf
 11   call MPI_FINALIZE(ierr)

      stop
      end
```

output

Garbage values

```
[somnath@localhost mpi_examples]$ mpirun -np 2 ./a.out
           1           0  1565869272       32550           0 myid           0
           0 sendbuf=   1.0000000000000000      recvbuf=   0.0000000000000000
           0           0           0          10           0 myid           1
           1 sendbuf=   1.0000000000000000      recvbuf=   1.0000000000000000
```

Proc-0, Proc -1:: sendbuf 1& recvbuf 0

Proc-1's recvbuf = Proc-0's sendbuf=1

Proc-0 has no receive operation, so no valid status

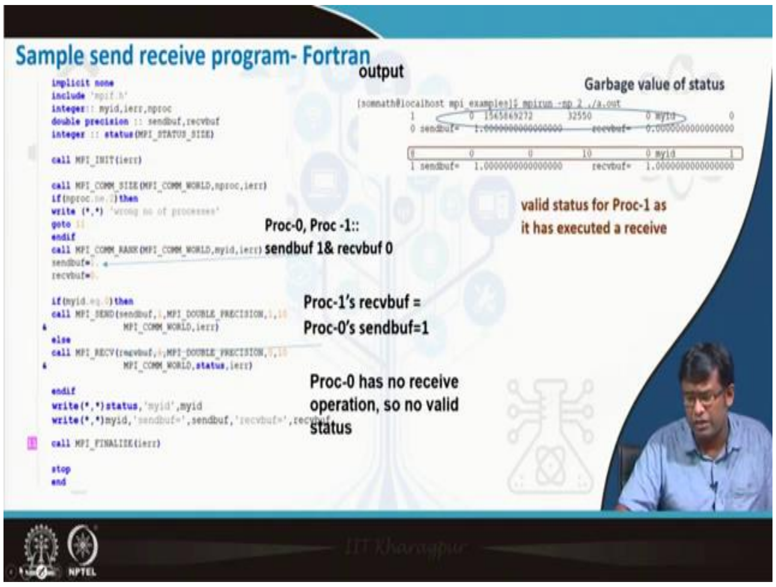
Processor 0 sent buffer is 1, processor 0 and processor 1 both had the same buffer 1 received buffer 0. Processor 1 got a receive buffer = 1 = send buffer of processor 0 which is 1.

The next one is processor 0, send buffer is 1 receive buffer is 0, processor 1 send buffer is 1 receive buffer is also 1 as we have predicted earlier. So, this is simply what will happen. It will send some data, another will receive it, and the receiving processor will overwrite what was initially declared in its buffer, its initial values will be overwritten by the received values.

Now, what is in status? This is the status call. For processor 0,my id 0,status has some garbage number, because it has not executed any status, status is garbage. So, this part of the status which is garbage as status is not executed, but this part of the status must not be garbage, these are valid status. What is in there? (My id is not part of the status) 10. What is 10? You can very easily see 10 is the tag. So, in status it tries the tag. Then, some of the 0s is the source id from where the message has been sent. There are many 0s, so you cannot identify it, but one of the 0s most likely the last 0 is for the source id.The first one is 8. What is 8? Size of the variable in a double precision, 8-byte variable in a double precision system.

So, this is the size of the data that has been sent, this is the tag, and some of the 0s, (other 0s we will we do not need to worry about) is the source id.This will be in the status message. If the status is successful it will write something meaningful here. If the status is unsuccessful then this 0 might also be the receiving the destination id. Then, it will write garbage and we will not write anything, that is not of use here. Process 0 has no receive operation, so no valid status. Garbage value of status is written here as well.

(Refer Slide Time: 21:48)



Sample send receive program- Fortran

```
      implicit none
      include 'mpif.h'
      integer:: myid,ierr,nproc
      double precision :: sendbuf,recvbuf
      integer :: status(MPI_STATUS_SIZE)

      call MPI_INIT(ierr)

      call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
      if(nproc.ne.2)then
      write (*,*) 'wrong no of processes'
      goto 11
      endif
      call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
      sendbuf=1.
      recvbuf=0.

      if(myid.eq.0)then
      call MPI_SEND(sendbuf,1,MPI_DOUBLE_PRECISION,1,10,
     &              MPI_COMM_WORLD,ierr)
      else
      call MPI_RECV(recvbuf,1,MPI_DOUBLE_PRECISION,0,10,
     &              MPI_COMM_WORLD,status,ierr)

      endif
      write(*,*)status,'myid',myid
      write(*,*)myid,'sendbuf=',sendbuf,'recvbuf=',recvbuf
11    call MPI_FINALIZE(ierr)

      stop
      end
```

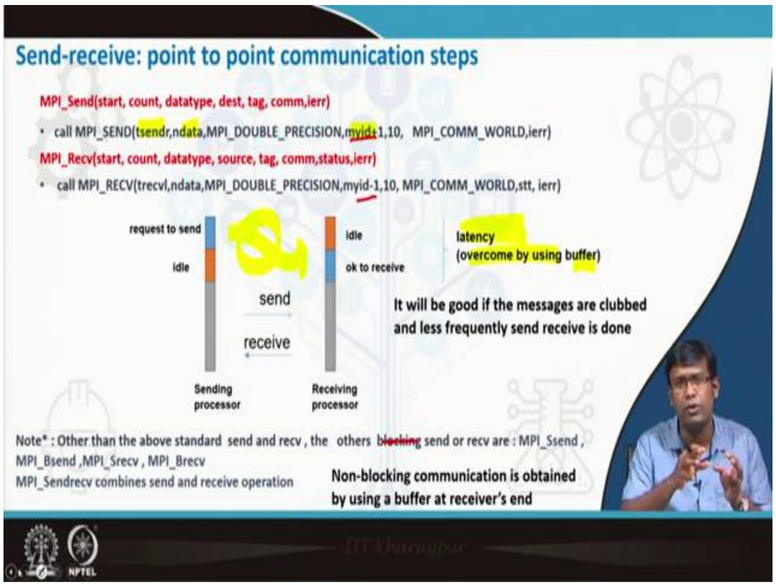
output

```
[somnath@localhost mpi_examples]$ mpirun -np 2 ./a.out
           1           0  1565869272       32550           0 myid           0
           0 sendbuf=   1.0000000000000000      recvbuf=   0.0000000000000000
           8           0           0          10           0 myid           1
           1 sendbuf=   1.0000000000000000      recvbuf=   1.0000000000000000
```

Garbage value of status
Proc-0, Proc -1:: sendbuf 1& recvbuf 0
Proc-1's recvbuf = Proc-0's sendbuf=1
Proc-0 has no receive operation, so no valid status
valid status for Proc-1 as it has executed a receive
IIT Kharagpur
NPTEL

Valid status for processor 1 as it has executed and received.

(Refer Slide Time: 21:52)



Send-receive: point to point communication steps

MPI_Send(start, count, datatype, dest, tag, comm,ierr)

- call MPI_SEND(tsendr,ndata,MPI_DOUBLE_PRECISION,myid+1,10, MPI_COMM_WORLD,ierr)

MPI_Recv(start, count, datatype, source, tag, comm,status,ierr)

- call MPI_RECV(trecvl,ndata,MPI_DOUBLE_PRECISION,myid-1,10, MPI_COMM_WORLD,stt, ierr)

request to send
idle
send
receive
Sending processor
idle
ok to receive
Receiving processor

latency
(overcome by using buffer)

It will be good if the messages are clubbed and less frequently send receive is done

Note* : Other than the above standard send and recv , the others ~~blocking~~ send or recv are : MPI_Ssend , MPI_Bsend ,MPI_Srecv , MPI_Brecv
MPI_Sendrecv combines send and receive operation

Non-blocking communication is obtained by using a buffer at receiver's end

IIT Kharagpur

Now, we have seen that, if we are writing in a Fortran, anything is a subroutine call MPI_send, the send buffer, the data size, double precision (as we are sending to the next processor)this is my id + 1, processor 0 is sending to processor 1, the tag communicator error. Processor 1 is receiving it, so, the receiving location, the size, the data type, my id 1 processor 1 is receiving from my id 0, so my id – 1, that matching tag and the communicator status error.

So, when we are writing a program across many processors, we cannot hard code and say processor 1 will take data from processor 2, processor 5 will send data to processor 11. We have to write an algorithm through because many processors will execute the same command, same send command, same receive command. Based on their my id they will find out which process they will send the data to or from which processor they will receive the data. So, that particular part has to be written as a function of my id,so, that you can use a generic command which is executed by all the processors.

Now, when send and receive happens, what happens? As the sending processor first sends data, it first puts a request to send to the receiving processor. The sending processor looks into the receiving processor's location, and the request goes to the receiving processor, a single piece of information that is okay to receive, does this processor exist in the communicator and then is it okay to receive it.

Then this receiving processor tells that, yes, it is okay to receive. This data till that time when the sending processor sends the data, the receiving processor is not doing anything, it is sitting idle, if it has finished its previous job it is sitting idle.

When the receiving processor says that it is okay to receive by that time sending processor cannot do anything, it has just pinged the receiving processor, is it in the communicator, is it in the network and, is okay to receive the data, and then the data transfer happens.

So, there is a substantial latency even when we give the data transfer process, till then the processors cannot do anything. Even they cannot do data transfer. Data transfer has its own overhead. We have looked about the data transfer overheads, but this is like a startup line. Before, actual data transfer there is a latency across, because it is finding out the processor to which it will send the data, asking whether you are okay to receive the data and then the next processor who will receive the data is replying to that and this particular period gives latency to both the processors.

One way to overcome this latency is use a buffer, that there will be some buffer in between the network, there is some space in between the network or some buffer located in the processors which is actually designated to the network.

So, when the send call has been given , some data already goes and sits in the buffer. So, it is not ideal. It has started sending the data and from this buffer data will go to the receiving processor. So, this is a way of reducing the latency using a buffer.

Now for every data transfer you have to go through the latency, that some data will come, some information there will be some latency because some the processor will ask the receiving processor whether it is okay to receive the data, the receiving processor will answer, and later data transfer will take place.

So, if you are very frequently doing data transfer and sending small packets of data every time this latency will be encountered, every time this latency will be added to your computation time. With every send receive message there is a startup time that the processor asks whether it is to send the data and the other processor replies. So, if you can club a lot of data and try to send it a single packet, then only one ping and response will be sufficient to initiate the actual data transfer.

So, you can reduce latency per word of data transfer. But if you do not club the message and send receive calls multiple times, break the data into multiple small pieces and send receive calls multiple times there will be more latency.

This is one important point while discussing send receive point to point communication, that when you have to send large pieces of data do not try to send this data as multiple packets, try to pack entire data in a single packet and send, that is a more efficient process.

Another part is that when this data is being sent, it is being sent from a buffer; the send address and the size that is given. So, if the data is discontinuous, if you are sending is not contiguous, if you are sending one location and then there is some gap and then sending another location, then it will be more expensive to send the data because this buffer will be made by the compiler and MPI wrapper, so, that one pocket of data will go and then preparing this buffer from this non-contiguous pieces of the memory will be difficult.

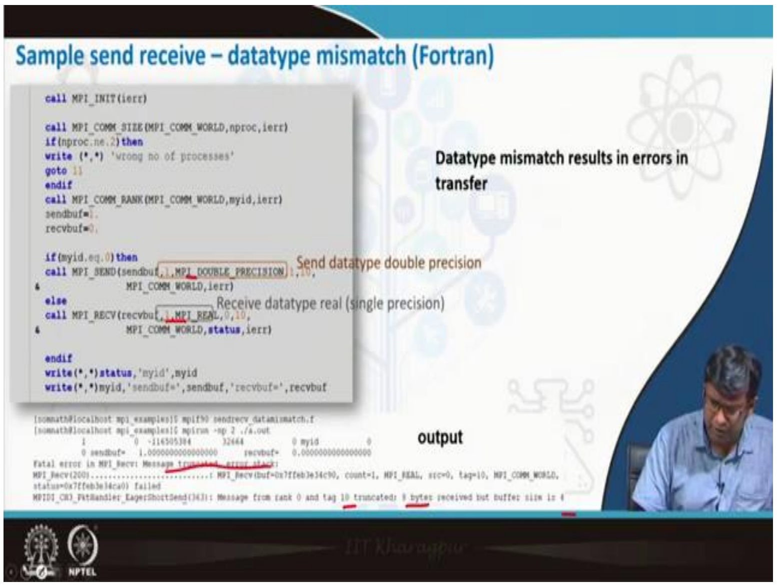
So, if you are using a low major storage in say C and try to send a column out of that, it will be difficult. So, it is good to pack entire data in a contiguous series. In C taking a (Refer Time: 28:21) row vector, in the Fortran you put the entire data in a column vector and try to send it. So, sending the buffer address and the size will be sufficient, and while doing the data transfer the compiler does not need to spend time on preparing the right buffer.

Other than the above send receive(blocking),there are MPI_Ssend and MPI_Srecv receive which are synchronous send receive, that means, if there are multiple processors, they are some way synchronized, even if there is buffer one send is finished only when the corresponding receive is finished. In standard send receive though they are blocking there is a buffer, and once the send buffer is free the same process can be finished, but in synchronous then there is more overhead. MPI_Bsend is again a blocking send, blocking receive, and also, there are MPI_isend Irecv which allows you to get a non-blocking mode of send and receive. There are multiple variants of send receive commands and you can look into any MPI manual, especially the one I have given Lawrence Livermore’s national labs MPI tutorial website. You can go in and look into the send receive commands from there. There are multiple variants of send receive, and you can have different functionality from each other.

If you use MPI_sendrecv command, that combines the send receive operation; that means, one processor will send to another and at the same time it is receiving from that processor. So, you can combine both of them and write a function MPI_Sendrecv which is present in MPI protocols. But with MPI_send an MPI_receive you can get your work done. But for more flexibility and more functionality you can use different other functions of versions of send receive.

When you use MPI_isend Irecv which are non-blocking commands, not only during the network, also in the receiver end you use a virtual buffer on which data is being stored. Even if the entire data transfer is finished in that sense that the receiver has received entire data, there is a buffer where this data goes and a sense of freeing the send buffer happens, and that is in the non-blocking communication; so, that the through the data transfer is not actually finished, but this buffer helps the processors to get a sense that the data transfer is done and they can go and do their own work, and that is in the non-blocking communications.

(Refer Slide Time: 31:52)



Sample send receive – datatype mismatch (Fortran)

```
      call MPI_INIT(ierr)

      call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
      if(nproc.ne.2)then
      write (*,*) 'wrong no of processes'
      goto 11
      endif
      call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
      sendbuf=1.
      recvbuf=0.

      if(myid.eq.0)then
      call MPI_SEND(sendbuf,1,MPI_DOUBLE_PRECISION,1,10,
     &              MPI_COMM_WORLD,ierr)
      else
      call MPI_RECV(recvbuf,1,MPI_REAL,0,10,
     &              MPI_COMM_WORLD,status,ierr)

      endif
      write(*,*)status,'myid',myid
      write(*,*)myid,'sendbuf=',sendbuf,'recvbuf=',recvbuf
```

Send datatype double precision

Receive datatype real (single precision)

Datatype mismatch results in errors in transfer

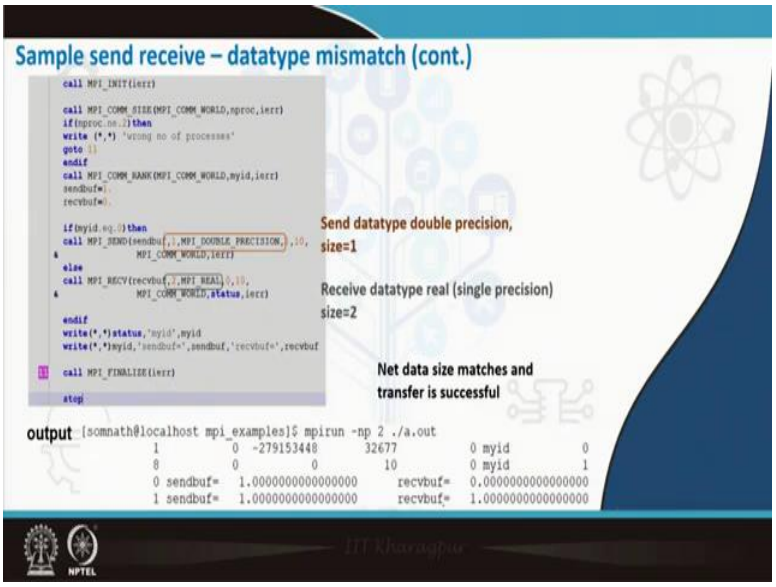
output

```
[somnath@localhost mpi_examples]$ mpif90 sendrecv_datamismatch.f
[somnath@localhost mpi_examples]$ mpirun -np 2 ./a.out
           1           0  -116505384       32664           0 myid           0
           0 sendbuf=   1.0000000000000000      recvbuf=   0.0000000000000000
Fatal error in MPI_Recv: Message truncated, error stack:
MPI_Recv(200)...........................: MPI_Recv(buf=0x7ffeb3e34c90, count=1, MPI_REAL, src=0, tag=10, MPI_COMM_WORLD,
status=0x7ffeb3e34ca0) failed
MPIDI_CH3_PktHandler_EagerShortSend(363): Message from rank 0 and tag 10 truncated; 8 bytes received but buffer size is 4
```

It is also important that we have right matching of data type while doing send receive operations, and if the data types do not match, we can see that a processor is sending double precision and the other processor is sending real data or a single precision data. Sending data is double precision, receiving data is real. So, double precision typically has double size of the real variable. So, 2 packets of information are being sent and if it is sent as MPI real or single position 1 pack of information is received. So, the data transfer will be incomplete and we can see that the output will be that the send is not finished and you will get some wrong output that 8 bytes received, but the buffer size which is in the receive buffer size is 4. But 8 buffers are received because double precision very variable is sent. So, 8 bytes are received. So, the message is truncated and you will get a message truncated error stack here. So, the data transfer will typically not be complete.

Now we can understand that even if it is a mismatch of data type it is not a matter of type in double precision and real precision it is a matter of the size. The data sizes are different.

(Refer Slide Time: 33:41)



Sample send receive – datatype mismatch (cont.)

```
      call MPI_INIT(ierr)

      call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
      if(nproc.ne.2)then
      write (*,*) 'wrong no of processes'
      goto 11
      endif
      call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
      sendbuf=1.
      recvbuf=0.

      if(myid.eq.0)then
      call MPI_SEND(sendbuf,1,MPI_DOUBLE_PRECISION,1,10,
     &              MPI_COMM_WORLD,ierr)
      else
      call MPI_RECV(recvbuf,2,MPI_REAL,0,10,
     &              MPI_COMM_WORLD,status,ierr)

      endif
      write(*,*)status,'myid',myid
      write(*,*)myid,'sendbuf=',sendbuf,'recvbuf=',recvbuf

 11   call MPI_FINALIZE(ierr)

      stop
```

Send datatype double precision, size=1

Receive datatype real (single precision) size=2

Net data size matches and transfer is successful

output
```
[somnath@localhost mpi_examples]$ mpirun -np 2 ./a.out
           1           0  -279153448       32677           0 myid           0
           8           0           0          10           0 myid           1
           0 sendbuf=   1.0000000000000000       recvbuf=   0.0000000000000000
           1 sendbuf=   1.0000000000000000       recvbuf=   1.0000000000000000
```

 This can be handled if we try to receive, send one double precision data and try to receive 2 single precision data. So, then you can see that the net data size matches and the transfer is successful, and you can see that what is being sent data has been received correctly.

So, this is an important aspect that though we talked about the data type and size of the data and said that they must match, but this is information which is being sent through the interconnect in an MPI_send receive call; so, even there is possibility at the data size and data type is not matching, but their combination, the size and type combination matches the net size of the data that has been sent through the processor.

So, the processor 0 is sending some data, the net amount of data is which is being received by the process 1, if that matches with the sent amount of data then this data transfer will be successful and we will get the right data. So, this is another important point that matching is not on in terms of the data type and data size only, but matching is in terms of the combination of data type and data size which is required in MPI, because we are not exactly copying the variable from one place to other place, so that we require the types to be matched. We are taking some information from one computer and sending it to another computer, the other computer is receiving the information. So, in terms of the information size it should match.

We will finish this particular class here. And we will look into some more aspects of send, specially when sending large amounts of data because we have talked about data size, when the data size is big and a large amount of data is being sent. We will see what are the issues and especially how bottlenecks can happen, and then we look into collective communications using MPI.