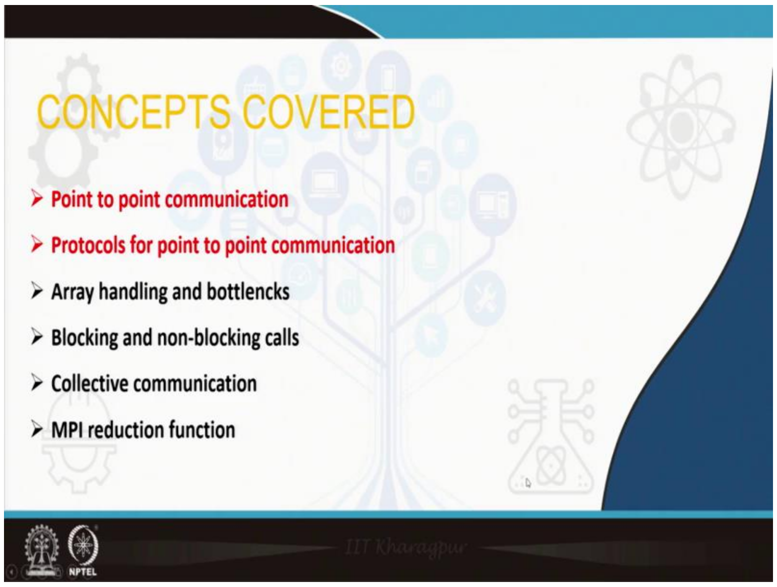
High Performance Computing for Scientists and Engineers

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

Module - 03
Message Passing Interface (MPI)
Lecture – 23
Communication using MPI (continued)

Welcome to the class of High-Performance Computing for Scientists and Engineers and we are discussing MPI programming where, which is the 3rd module of this course. We are discussing Communications using MPI.

(Refer Slide Time: 00:38)



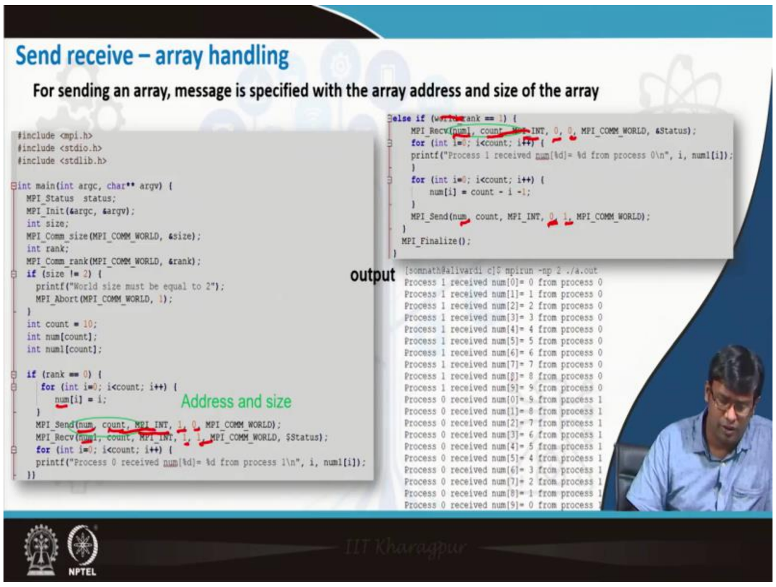
CONCEPTS COVERED

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

IIT Kharagpur
NPTEL

So, continuing from our previous lectures where we have discussed point to point communication and protocols for point to point communication, we will see the other aspects like array handling, blocking, non-blocking calls and then we look into collective communications and its protocols.

(Refer Slide Time: 00:55)



Send receive – array handling

For sending an array, message is specified with the array address and size of the array

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

int main(int argc, char** argv) {
  MPI_Status  status;
  MPI_Init(&argc, &argv);
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (size != 2) {
    printf("World size must be equal to 2");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }
  int count = 10;
  int num[count];
  int num1[count];

  if (rank == 0) {
    for (int i=0; i<count; i++) {
      num[i] = i;
    }
    MPI_Send(num, count, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(num1, count, MPI_INT, 1, 1, MPI_COMM_WORLD, $Status);
  for (int i=0; i<count; i++) {
    printf("Process 0 received num[%d]= %d from process 1\n", i, num1[i]);
  }}
else if (world_rank == 1) {
    MPI_Recv(num1, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &Status);
    for (int i=0; i<count; i++) {
    printf("Process 1 received num[%d]= %d from process 0\n", i, num1[i]);
    }
    for (int i=0; i<count; i++) {
        num[i] = count - i -1;
    }
    MPI_Send(num, count, MPI_INT, 0, 1, MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```

Address and size

output

```
[somnath@alivardi c]$ mpirun -np 2 ./a.out
Process 1 received num[0]= 0 from process 0
Process 1 received num[1]= 1 from process 0
Process 1 received num[2]= 2 from process 0
Process 1 received num[3]= 3 from process 0
Process 1 received num[4]= 4 from process 0
Process 1 received num[5]= 5 from process 0
Process 1 received num[6]= 6 from process 0
Process 1 received num[7]= 7 from process 0
Process 1 received num[8]= 8 from process 0
Process 1 received num[9]= 9 from process 0
Process 0 received num[0]= 9 from process 1
Process 0 received num[1]= 8 from process 1
Process 0 received num[2]= 7 from process 1
Process 0 received num[3]= 6 from process 1
Process 0 received num[4]= 5 from process 1
Process 0 received num[5]= 4 from process 1
Process 0 received num[6]= 3 from process 1
Process 0 received num[7]= 2 from process 1
Process 0 received num[8]= 1 from process 1
Process 0 received num[9]= 0 from process 1
```

When we are sending an array through MPI we have to specify the array address and the size of array and we understand that in any practical program we are not just going to send one small information or an integer or a float. We will send a pack of information, we will send a number of variables, vectors, matrices, etcetera. So, you have to send an array and say what is the size of the array. It can be understandable that this send means that one dimensional array will be sent. As this is a pack of information, we will send the location and the size. So, as I have mentioned earlier also when sending data, it can be optimized in terms of less overheads and the efficiency that if we can pack everything in a contiguous single dimensional array and tell the address and the size of that array.

This is something we will again discuss in the matrix algebra programs using MPI later in this class. We will see that if we have to transfer a matrix from one computer to another computer, one processor to another processor, we will use a single pointer and put everything in a single dimensional array and try to transfer it ,that is the most optimal way.

In looking into the same thing, if we are sending an array, we have to specify the location, the address of the array, size, type of the data type, where we are, sending it and the matching tag and communicator.

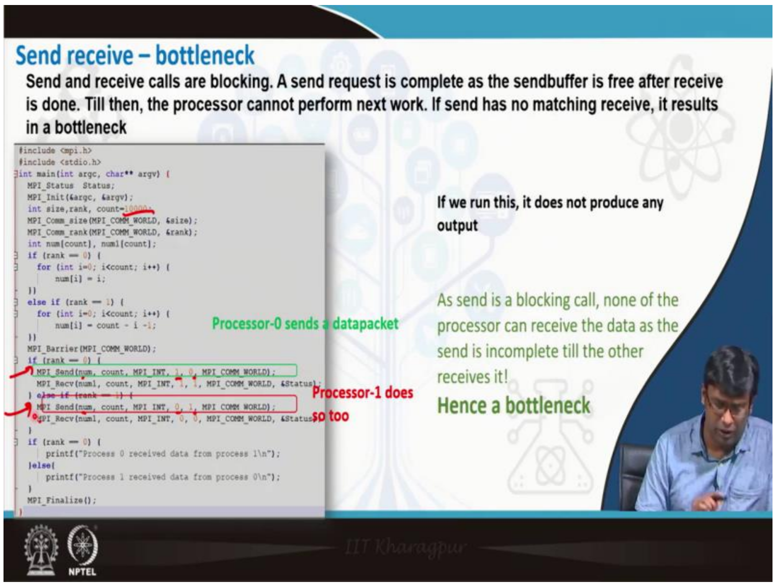
So, process 1 rank 0 will send to process 1 with a tag 0 and process 1 whose rank is given as 1 that will receive information with tag 0 from process 0 and it will put it in num 1.

What is being sent from num that will be put in num 1 and the size is important and the data type should match ,then it will execute another send and send data from this processor with tag 1, to process 0 and process 0 will receive this data with tag 1 from process 1 and store it in num 1.

So, process 0s num will be process 1s num 1 and process ones num will be process 0s num 1, we can see that the address and the size we mentioned earlier ,that it is important to mention the address and size of the of the array that will be sent and that size and type combo should match and the receiving address should be given, that is a way of transferring of array. Then we can see what process 1 has received from process 0 and what process 0 has received from process 1.

Within the process, because this is a loop, it will write from 0 to 9, the loop will execute accordingly. These are the ways how an array can be sent, that you specify the address ,size of the array and while receiving, you specify the address of the received buffer and the size, and accordingly number of elements will be received. But these arrays when it is being received , the malloc has to be performed and that particular array dimension has to be declared otherwise there will be segmentation fault or stack overflow.

(Refer Slide Time: 05:22)



Send receive – bottleneck

Send and receive calls are blocking. A send request is complete as the sendbuffer is free after receive is done. Till then, the processor cannot perform next work. If send has no matching receive, it results in a bottleneck

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  MPI_Status  Status;
  MPI_Init(&argc, &argv);
  int size,rank, count=10000;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int num[count], num1[count];
  if (rank == 0) {
    for (int i=0; i<count; i++) {
        num[i] = i;
  }}
  else if (rank == 1) {
    for (int i=0; i<count; i++) {
        num[i] = count - i -1;
  }}
  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0) {
    MPI_Send(num, count, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(num1, count, MPI_INT, 1, 1, MPI_COMM_WORLD, &Status);
  } else if (rank == 1) {
    MPI_Send(num, count, MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Recv(num1, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &Status);
  }
  if (rank == 0) {
     printf("Process 0 received data from process 1\n");
  }else{
     printf("Process 1 received data from process 0\n");
  }
  MPI_Finalize();
}
```

Processor-0 sends a datapacket

Processor-1 does so too

If we run this, it does not produce any output

As send is a blocking call, none of the processor can receive the data as the send is incomplete till the other receives it!

Hence a bottleneck

Send and receive calls are blocking. A send request is complete as the send buffer is free after receive is done. The send buffer can be made free even before receive is done for certain cases, but if you are sending a larger array send receive is complete for normal receive operations.

So, what one processor is trying to send, another processor has received it ,the senders' buffer is free and the process is complete, else the process cannot complete, and the process cannot do next work. If send has no matching receive it results in a bottleneck and that is exactly what is happening here. So, in this program the same num variable will be send and num 1 variable will be received .What will happen in that rank 0 is trying to send num at the same time rank 1 is also trying to send num,so, both the processors are trying to send at same instant and this is very interesting. Now, process 0 is sending to process 1 process 0 is sending num to process 1 with tag 0. Each process will execute the comments as it follows one by one. So, when it comes to process 1, at the same time process 1 is trying to send a variable name num address num to process 0 with tag 1;so, both are sending. I am giving my friend an envelope, he is also giving me an envelope, but none of us can receive it (we assume that we are working with one hand), because the processes are only executing one instruction at a time. So, both of our hands are full and none of us can receive therefore, what will happen? We will stop here. This will go on, as both are holding the envelope and standing in front of each other.

So, process 1 is trying to send data, process 0 is at the same time trying to send data, but nobody is receiving at that instance and this will give us a bottleneck. If we run this, it will not produce any output. We will see something like an infinite loop ,the executable is running and nothing is happening, because the same call from process 0 cannot be completed and process 1 send call cannot be completed.

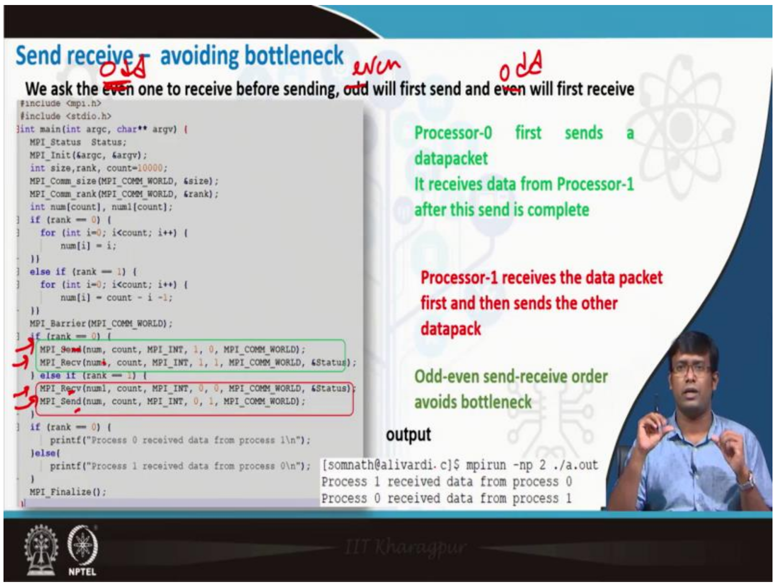
Unless, process 1 will complete its send call it cannot do the receive operation or unless, process 0 send is complete it cannot do the receive operation. So, none of them can execute the receive and therefore, it will be a bottleneck. So, this is a situation which might happen especially, if you have large data. Large data means; if the data size is small it might still work, because we see that some of the data is waiting at the buffer in the interconnect. If some data is sitting at the interconnect buffer it still might work, but if there is large data, then this data is not small to sit in the interconnect buffer. So, the entire data is in wait, none of this data is being received , none of the send is complete, and it will not produce any output. So, the process says 0 sends a data packet, and process 1 does too. No one is free to receive it both are doing send. Send is a blocking call that means, if process 0 is blocked unless, this send is done process 0 cannot execute the next statement.

Similarly, process 1 is blocked, unless this send is done ,it cannot execute the receive statement. So, none of the received statements cannot be executed. Therefore, the data transfer cannot take place across the processors and send is incomplete till others can receive it and others cannot receive it, because others are also hanging at the same place it has another same command which is incomplete, because the present processor cannot receive this data.

This is something which we will try to avoid in our MPI program that if data transfer happens between two processors at the same time both of them should not send data or if one is sending data other will receive first and if one is receiving data the other will send first.

So, there has to be a staggering of send receive in between the processors who are communicating in between each other and there are many instances when both the processors are interchanging data .A is giving some data to B,at the same time B is giving some data to A. So, one will send, another will receive and send receive should have a standard way hence, a bottleneck. This is a very serious issue in MPI programs and all the programmers must be aware of that and as I said in MPI, it is the programmer who is writing these instances. It is programmer who is writing the send receive commands which will be send, where it will be send, which place we will store it, what is the tag, at which location it will be sent, what is the type of the data, etcetera; programmer has lot of responsibility in MPI program. So, you must be very well aware that if he does something wrong in a writing send receives command there will be a bottleneck.

(Refer Slide Time: 11:14).



Send receive – avoiding bottleneck

We ask the ~~even~~ odd one to receive before sending, ~~odd~~ even will first send and ~~even~~ odd will first receive

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  MPI_Status  Status;
  MPI_Init(&argc, &argv);
  int size,rank, count=10000;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int num[count], num1[count];
  if (rank == 0) {
    for (int i=0; i<count; i++) {
        num[i] = i;
  }}
  else if (rank == 1) {
    for (int i=0; i<count; i++) {
        num[i] = count - i -1;
  }}
  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0) {
   MPI_Send(num, count, MPI_INT, 1, 0, MPI_COMM_WORLD);
   MPI_Recv(num1, count, MPI_INT, 1, 1, MPI_COMM_WORLD, &Status);
  } else if (rank == 1) {
   MPI_Recv(num1, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &Status);
   MPI_Send(num, count, MPI_INT, 0, 1, MPI_COMM_WORLD);
  }
  if (rank == 0) {
     printf("Process 0 received data from process 1\n");
  }else{
     printf("Process 1 received data from process 0\n");
  }
  MPI_Finalize();
}
```

Processor-0 first sends a datapacket
It receives data from Processor-1 after this send is complete

Processor-1 receives the data packet first and then sends the other datapack

Odd-even send-receive order avoids bottleneck

output

```
[somnath@alivardi.c]$ mpirun -np 2 ./a.out
Process 1 received data from process 0
Process 0 received data from process 1
```

How to avoid a bottleneck? The most standard way is when the even one receives the data odd one will be sending. So, an odd processor or an even processor is first sending it, another is receiving it and the other processor is first receiving it, another is sending it. I have actually written it reversely. The even one is sending the data and receiving the and then receiving it. The odd one is first receiving it; the odd one is first receiving it and then sending the data.

So, rank 0 will first send the data and the odd one the rank 1 will first receive it .Also rank 1 to first receive and then do send; that means, rank 0 will first do send and rank 1 will first do receive. So, rank 0 send is complete as rank 1 has received the data, then rank 1 the odd one will send the data and the even will receive it.

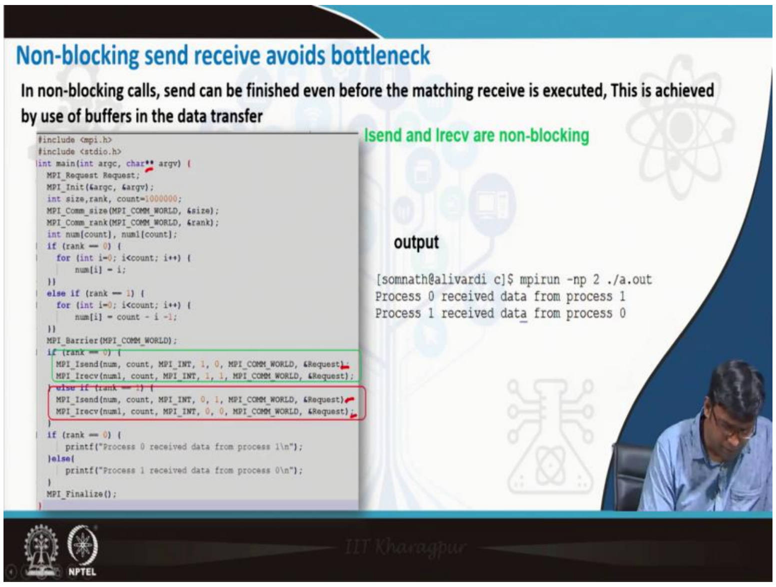
So, if you have a large number of processors doing an odd even for send receive across different processors that helps. We have just staggered the send receive order, 0 is first sending then receiving, 1 is first receiving then sending. So, one send is finished by 0 ones received and then one sends then both are free. So, one can send and 0 can receive it .

First, 0 sends the data packet. It receives data from processor 1 after the send is complete. Processor 1 receives the data packet and then sends the other data pack .Odd-even send receive avoids the bottleneck .If we run the program, we will see that processor 1 has received data from processor 0 and processor 0 has received the data from processor 1.

So, though in MPI, multiple processors are there and they can execute the job in their own time, but there is some synchronization, because when two processors are sending and receiving data both of them will be free at one instance once the send is complete. So, first processor 1 and processor 0 are free once processor 1 has received the data and the next operation will be processor 0 is receiving data from processor 1.

These data transfer sending and receiving by 1 processor cannot be done parallelly. So, when two processors are sending and receiving data between each other, there will be some sequentially in that ,one send and receive will be done, then another receive and send will be done and that helps us to avoid the bottleneck.

(Refer Slide Time: 14:21)



Non-blocking send receive avoids bottleneck

In non-blocking calls, send can be finished even before the matching receive is executed, This is achieved by use of buffers in the data transfer

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  MPI_Request Request;
  MPI_Init(&argc, &argv);
  int size,rank, count=1000000;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int num[count], num1[count];
  if (rank == 0) {
    for (int i=0; i<count; i++) {
        num[i] = i;
  }}
  else if (rank == 1) {
    for (int i=0; i<count; i++) {
        num[i] = count - i -1;
  }}
  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0) {
   MPI_Isend(num, count, MPI_INT, 1, 0, MPI_COMM_WORLD, &Request);
   MPI_Irecv(num1, count, MPI_INT, 1, 1, MPI_COMM_WORLD, &Request);
  } else if (rank == 1) {
   MPI_Isend(num, count, MPI_INT, 0, 1, MPI_COMM_WORLD, &Request);
   MPI_Irecv(num1, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &Request);
  }
  if (rank == 0) {
     printf("Process 0 received data from process 1\n");
  }else{
     printf("Process 1 received data from process 0\n");
  }
  MPI_Finalize();
}
```

Isend and Irecv are non-blocking

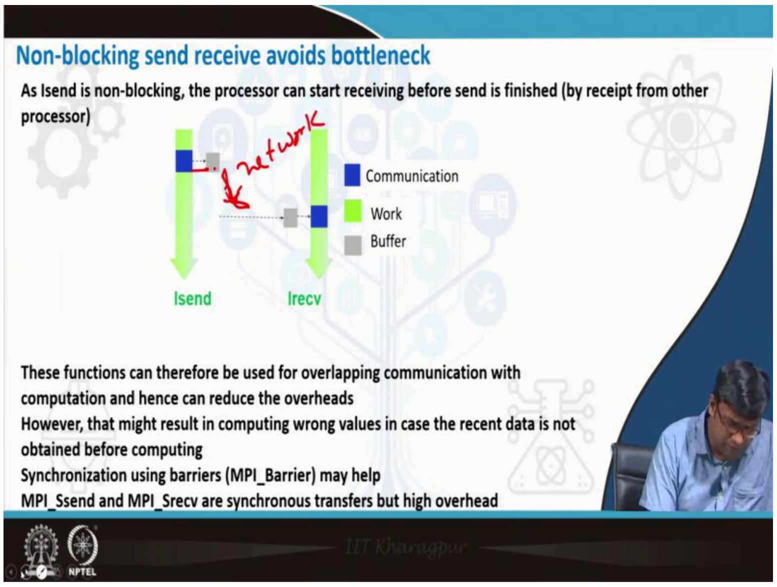
output

```
[somnath@alivardi c]$ mpirun -np 2 ./a.out
Process 0 received data from process 1
Process 1 received data from process 0
```

The other way he is using a non-blocking call. If we are using non-blocking call like Isend, Irecv, send can be finished even before matching received is executed, this is achieved by use of buffer ,there is some buffer in between the processors in the data transfer process and as we execute Isend then the data goes to the buffer. So, processor 0 first executes Isend then Irecv , processor 1 also executes, Isend, and Irecv . So, this data is sending somewhere in the buffer, so, though processor 1 has not received the data sent by process 0, once it has sent that entire data (we have increased data size to 10 to the power 6 million size array), everything can go and sit in the buffer. So, as processor 0 has send the data it is sitting in the buffer, it can do something else as its buffer is free. Similarly, processor 1 has send the data ,it is also in the buffer processor, so it can do something else. Therefore they can both send the data at the same instance ,the data is waiting in the buffer and they can parallelly receive the data. It is more flexible and more parallelized.

We can see both the processors send the data and the status message is not present here rather there is a request message of MPI request is given there. So, it is a request of sending data which finishes the job processor 0 wants to send data, it does not finish the send it sends a request to send the data to processor 1 and it is free it puts the data in the buffer sends a request puts the data in the buffer does its next work. Processor 1 similarly looks into the request and receives the data, but before that it has also made a request, put its data in a buffer and when to receive the data.

(Refer Slide Time: 16:26)


Non-blocking send receive avoids bottleneck

As Isend is non-blocking, the processor can start receiving before send is finished (by receipt from other processor)

These functions can therefore be used for overlapping communication with computation and hence can reduce the overheads
However, that might result in computing wrong values in case the recent data is not obtained before computing
Synchronization using barriers (MPI_Barrier) may help
MPI_Ssend and MPI_Srecv are synchronous transfers but high overhead

IIT Kharagpur

These are non-blocking, so processors can start receiving before the send is finished from another processor. This happens like during communication data is sent to a buffer and for the processor through Isend processor 0 and say processor 1 has executed Irecv it will receive data from the buffer and it can do something else after that.

So, once processor 0 has send the data in the buffer it is free to do the next computation and as the data goes through the network cable it takes some time, the forward data transfer time that we have discussed earlier as it goes through the network .But that particular time processor 0 is still active and it can do its own computation. So, this function therefore, can be used for overlapping communication with computation. Communication has started, it has put this data in a buffer and it can do its own computation. So, though there is certain communication overhead, at the same time computations are going on. So, these overheads can be minimized. So, this is a more optimized and efficient way of transferring data.

However, it might result in computing wrong values, because many times you while computing in one processor it needs data from the other processor, but if Isend and Irecv is giving this data transfer is happening through buffer it is computing something, but it has not received the right data in between that.

So, there is less synchronization than the earlier simple send receives. It was very synchronized, one send is not complete before the other has received it, but as this is less synchronized you have to be careful. You have to put barriers or put different synchronization calls, so that there is no error in the computation or you have to be sure that the data that is being communicated is not being required during the computation, because computation and communication are overlapped in this particular case.

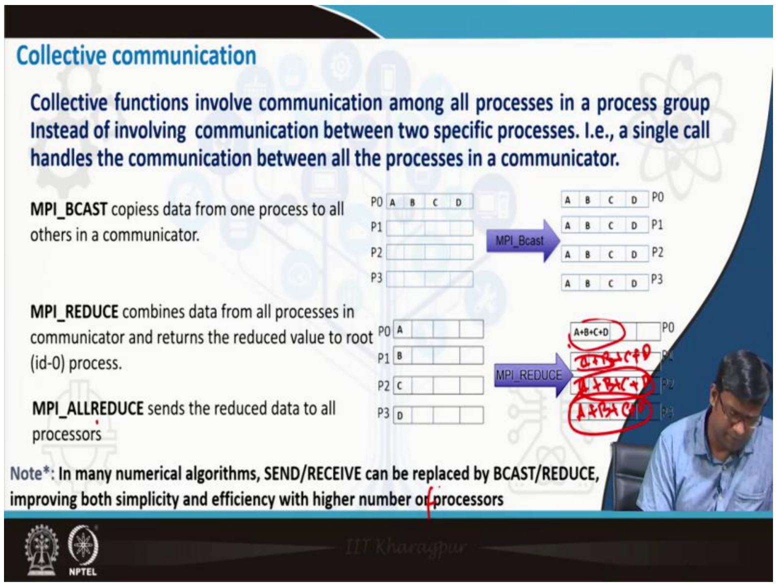
Synchronization barriers might help and sometime for more synchronous transfer you can use MPI Ssend and Srecv which says that the matching receive and matching sent has to be matching receive has to be finished whatever be the data size before the next communication it is more synchronous than simple send and receive, but it has high overhead, but this synchronization is also sometime important.

This, Irecv, Isend are helpful and optimized also, but you can avoid them by programming with simple send receive; you just stagger the send receive arrangement then you can avoid bottleneck. Irecv and Isend simply help you to avoid the bottleneck and increase your efficiency, but even without using Irecv Isend, with simple send receive you can do that.

MPI is small, only the six functions which includes simple send and simple receive you can write any parallel program, but if you want better performance or if you want more flexibility you can get some other MPI calls like Isend and Irecv and can get better performance.

So, that is why MPI is small, it has only six functions by which you can write any MPI program, but it is large, because you can pull other functions into the program and make it more flexible and more optimized.

(Refer Slide Time: 20:00)



Collective communication

Collective functions involve communication among all processes in a process group Instead of involving communication between two specific processes. I.e., a single call handles the communication between all the processes in a communicator.

MPI_BCAST copiess data from one process to all others in a communicator.

MPI_REDUCE combines data from all processes in communicator and returns the reduced value to root (id-0) process.

MPI_ALLREDUCE sends the reduced data to all processors

Note*: In many numerical algorithms, SEND/RECEIVE can be replaced by BCAST/REDUCE, improving both simplicity and efficiency with higher number of processors

Now, we come to another important part of the model of communication which is collective communication. Collective functions involve communication among all processes in a process group .Instead of involving two specific processes to communicate in between them, collective communication gives us a single communication call which helps communication of data across many processors or across all the processors in the communicator. Like if I write an email it goes to the person I am intending and he receives the data. If I announce something with a hand mic, it goes to many people; many people get this information at one go.

So, collective communication uses the topology of the network and sends data in one function to all the processors. One is the MPI_Bcast ,it is like taking data from one processor and communicating it to all the processors in the communicator. It's simply like speaking with a microphone and using a loudspeaker, so that everybody can listen to it. Say processor 0 has some data ABCD ,if it uses MPI_Bcast this data is copied to all the processors in the communicator.

So, inside MPI_comm_world how many processors are there everybody gets this information. You do not need to do point to point communication from processor 0 to processor 1, process 0 to processor 2, and processor 0 to processor 3 and do send and receive in between them, because all the processors will get the data. This is more optimized, because this is using the topology of the network ,the network is itself propagating the data to all the processors ;taking data from one process and propagating it to all the processors. Another can be MPI_reduce which combines data from all the processors and returns the reduced value to root or we mention the processor id, usually it is, rank 0 process.

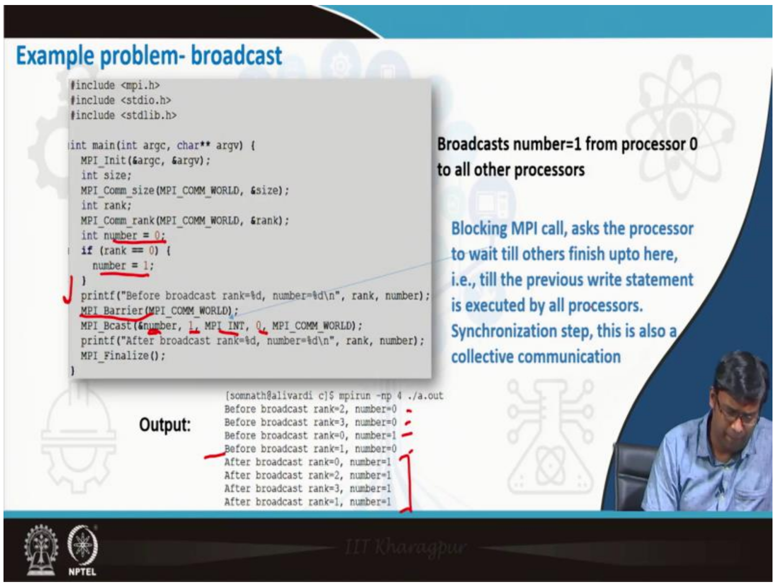
So, there are A B C D in different processes ;you write MPI_reduce with a MPI operation sum A + B + C + D goes to processor 0.You can do it using send receive also. I said like these six functions; we have discussed earlier, MPI_init, MPI_finalize, MPI_comm_size, MPI_comm_rank, MPI_send, MPI_recv, you can do anything with that. But if you want to do it in a more optimized way, you use collective calls or like earlier, we said Isend Irecv and other calls.

What is the optimization here? If we have to do this using MPI send receive, all the processors can send their data to processor 0 and then processor 0 will add up the data, but that is using some sequentiality in data transfer, some send receive operations, multiple operations by the processors, etcetera. It is more complex in terms of writing the program as well as less optimized, but when you are using a collective call it is more optimized, because it takes the advantage of the behavior of the network in propagating data across processors or taking data from one processor and throwing it to multiple processors. So, less overhead is there also it is much simpler in terms of programming.

Instead of MPI_reduce, if you do MPI_allreduce, all the processors get this value. So, in MPI_reduce only processor 0 gets the sum in MPI_allreduce all the processors will get A + B + C + D that is propagated to all the processors. If you use any operation with reduce it goes to only the 0th processor or the master processor. If you use all reduce the reduced data goes to all the processors, and we get much less overhead ;it's a more optimized call.

In many numerical algorithms send receive can be replaced by Bcast and reduced and these improve both simplicity and efficiency with higher number of processors. So, if we have a large number of processors instead of doing many point to point communication a collective communication is an efficient way to do it, because it's optimized using the characteristics of that particular network .It is optimized as well as a simple program.

(Refer Slide Time: 24:59)



Example problem- broadcast

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

int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int number = 0;
  if (rank == 0) {
    number = 1;
  }
  printf("Before broadcast rank=%d, number=%d\n", rank, number);
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);
  printf("After broadcast rank=%d, number=%d\n", rank, number);
  MPI_Finalize();
}
```

Broadcasts number=1 from processor 0 to all other processors

Blocking MPI call, asks the processor to wait till others finish upto here, i.e., till the previous write statement is executed by all processors. Synchronization step, this is also a collective communication

Output:

```
[somnath@alivardi c]$ mpirun -np 4 ./a.out
Before broadcast rank=2, number=0
Before broadcast rank=3, number=0
Before broadcast rank=0, number=1
Before broadcast rank=1, number=0
After broadcast rank=0, number=1
After broadcast rank=2, number=1
After broadcast rank=3, number=1
After broadcast rank=1, number=1
```

So, these are the right calls for a programmer to write. One example is that there are multiple processors and the integer number is 0 for all the processors.

Processor 0 has given that integer to be a number and then all the processors are asked to write the number before doing broadcast. There is only one broadcast, which takes the number

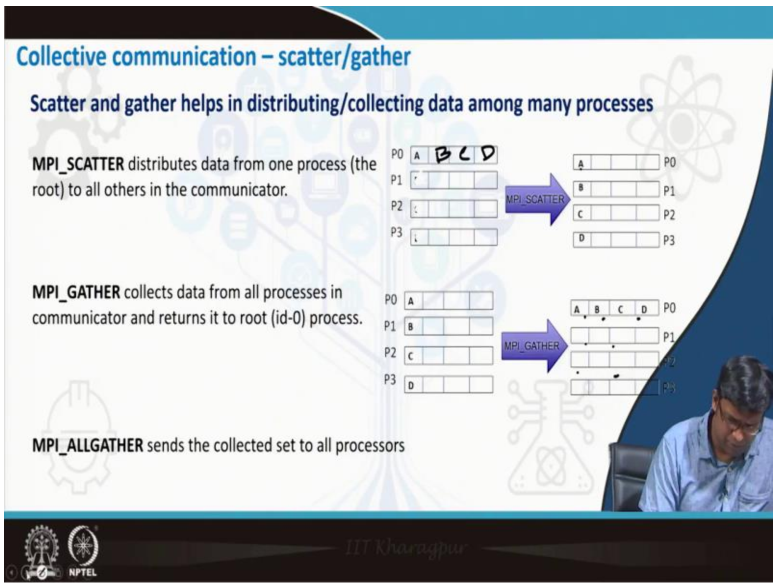
,address, the number size , MPI integer from processor 0 it will be broadcasted and asks all the processors to write it.

If we do send receive, you have to take the number one and write four send commands and they would have been executed sequentially, because processor 0 is sending number 1 to processor 1, processor 2, processor 3;one by one they would have operated and he should have received it.But here it’s more parallelized version of sending and receiving. Then you will see that before broadcast processor 0 has number = 1 others have number = 0 after broadcast everybody gets number = 1.

We can see another function called MPI_barrier. MPI_Barrier is a blocking call. It is for synchronization across the processors. When I put MPI_barrier all the processors have to come up here, write this and then only Bcast can be executed.

Whenever we put MPI_barrier all the processors will come up to that state and everybody is waiting (it is like an OMP barrier), till the others have finished up to that step and then the next step will happen. Interestingly, this is a collective synchronization that the entire communicator will wait till other processors have finished, so it also goes in the basis of a collective communication.

(Refer Slide Time: 27:25)



Collective communication – scatter/gather

Scatter and gather helps in distributing/collecting data among many processes

MPI_SCATTER distributes data from one process (the root) to all others in the communicator.

MPI_GATHER collects data from all processes in communicator and returns it to root (id-0) process.

MPI_ALLGATHER sends the collected set to all processors

There are other interesting collective communications which are scattered and gather; scatter gather helps in distributing and collecting data among many processors .MPI scatter distribute data from one processor to all in the communicator .Like processor 0 has this four data A ,B ,C, D. If we use the scatter then processor 0 retains A and other processors evenly get part of the data, B C D goes to other processors.

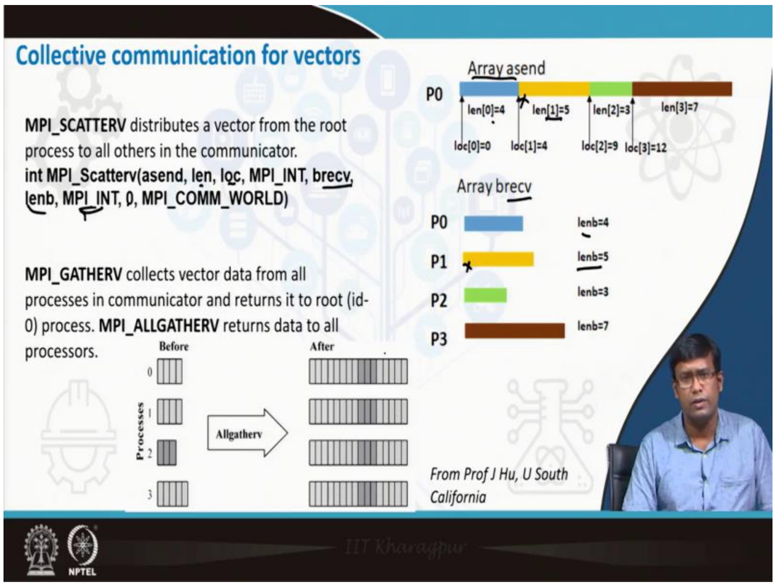
So, if you have an array and you want to share the array to different processors using a scatter, it will be scattered to all the processors. MPI_gather will do the opposite; it will collect data from all the processors. The processors have ABCD and MPI_gather will do the opposite; it will collect data from all the processors and send it to the root or id = 0 process.

You can appreciate that they can also be done using send receive command, but this is a more optimized and simple way of doing it.

So, parallelization specially in this API's MPI as well as OpenMP where doing it simply also helps you to do it efficiently. However, you need to know the right commands, you need to go through the manuals, find out the right protocols/ semantics and do it; but if you can do it in an optimized way it is also simple to write the program. Whereas, in sequence single processor programming the simplest way is not optimized, we have to do something else.

There is something called MPI_allgather. If you do MPI all gather this copy will go to all the processors. So, any collective communication with all means that not only the root process, but all the processors will get a similar copy of that. These are very helpful protocols; we will see some examples in subsequent classes.

(Refer Slide Time: 29:51)



Collective communication for vectors

MPI_SCATTERV distributes a vector from the root process to all others in the communicator.
int MPI_Scatterv(asend, len, loc, MPI_INT, brecv, lenb, MPI_INT, 0, MPI_COMM_WORLD)

MPI_GATHERV collects vector data from all processes in communicator and returns it to root (id-0) process. **MPI_ALLGATHERV** returns data to all processors.

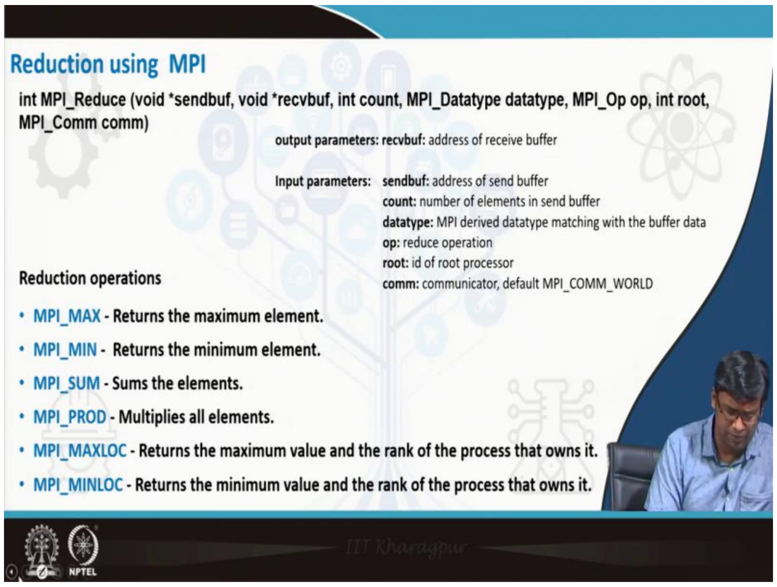
From Prof J Hu, U South California

Now, many times you do not have four valued arrays and give it to four processors. Many times, you have large vectors which are to be distributed among a few number of processors so each processor gets a part of the vector, and the call is MPI_Scatterv. It distributes a vector from the root process to the others in the communicator. We have an array asend and this is a long array and it has to be distributed among four processors. The first processor will get four elements. So, the length that will go to the first processor is 4, then the next processor will get 5 elements ,the length is 5, the third second processor will get 3 elements the length is 3, and the four third processor has length 7. So, this is a length variable which is also send with the array and the displacement or the location of the first element of that array which will be the first element on the corresponding processor. So, loc and lengths are the arrays which are also to be called in the Scatterv and the main array is asend .All the processor will receive and will get a local array of brecv and their local lengths will be output of this. So, len b has to be matched with this len or this will be the output of it. So, the total size is the summation of all the lengths , the data type is MPI int, processor 0 is sending all the processors inside MPI_comm_world.

Similarly, there is MPI_GATHERV which will do the reverse, it will collect vectors from all the processes and send it to one the 0th process or root process. In MPI_ALLGATHERV all the processes will get the combination of the vectors. These are very useful commands in matrix vector operation

So, in GATHERV each process has a different part of the array that their name should be the same for each in local process and then with all GATHERV each process will get a sequential combination according to the processor id of the array.

(Refer Slide Time: 32:42)



Reduction using MPI

int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

output parameters: recvbuf: address of receive buffer

Input parameters: **sendbuf:** address of send buffer
count: number of elements in send buffer
datatype: MPI derived datatype matching with the buffer data
op: reduce operation
root: id of root processor
comm: communicator, default MPI_COMM_WORLD

Reduction operations

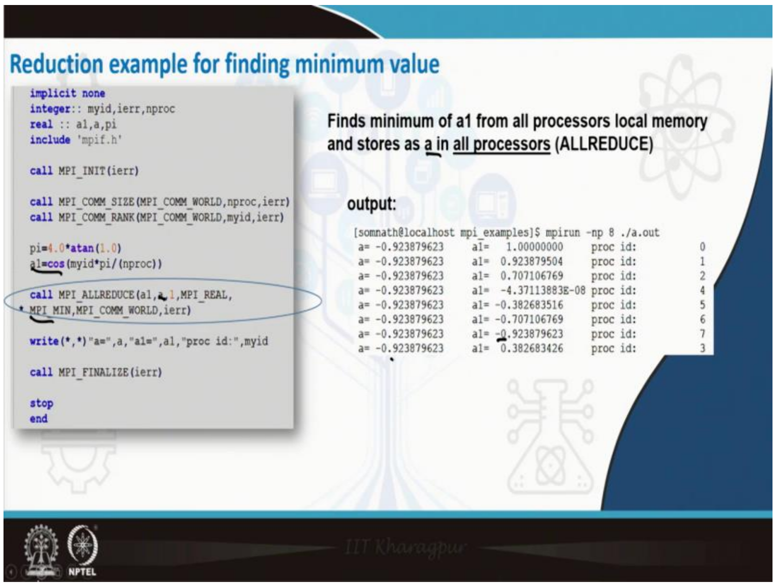
- MPI_MAX - Returns the maximum element.
- MPI_MIN - Returns the minimum element.
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it.
- MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.

Reduction is an important call. It starts with a send message, send buffer ,returns a receive buffer, takes the count , number of data there, what is the data type, what is the MPI operation and the count means, what is the count of the data in each processor, MPI operation and what is the id of the root processor or which should get the reduced value through a communicator.

Output parameters are the address of the receive buffer. Input parameters are the send buffer, the number of elements in the send buffer ,the MPI derived data type, the reduction operation, addition summation, addition multiplication, finding, minimize, finding maxima, etcetera and the id of the root processor will get the reduced value. If you use all reduce like ALLGATHER, all Scatterv, ALLGATHERV, entire data will come and the reduced value will be copied in the entire set of processors , all the processors will get the same reduced value.

These are few reduction operations, MPI MAX; returns the maximum element, MPI MIN; returns the minimum element across the local value, MPI SUM; sums the elements, MPI PROD; multiplies the elements, MPI MAXLOC; finds out what is the maximum value and which processor has the maximum value MPI MINLOC; returns what is the minimum value and which processor has the minimum value.

(Refer Slide Time: 34:15)



Reduction example for finding minimum value

```
implicit none
integer:: myid,ierr,nproc
real :: a1,a,pi
include 'mpif.h'

call MPI_INIT(ierr)

call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)

pi=4.0*atan(1.0)
a1=cos(myid*pi/(nproc))

call MPI_ALLREDUCE(a1,a,1,MPI_REAL,
* MPI_MIN,MPI_COMM_WORLD,ierr)

write(*,*)"a=",a,"a1=",a1,"proc id:",myid

call MPI_FINALIZE(ierr)

stop
end
```

Finds minimum of a1 from all processors local memory and stores as a in all processors (ALLREDUCE)

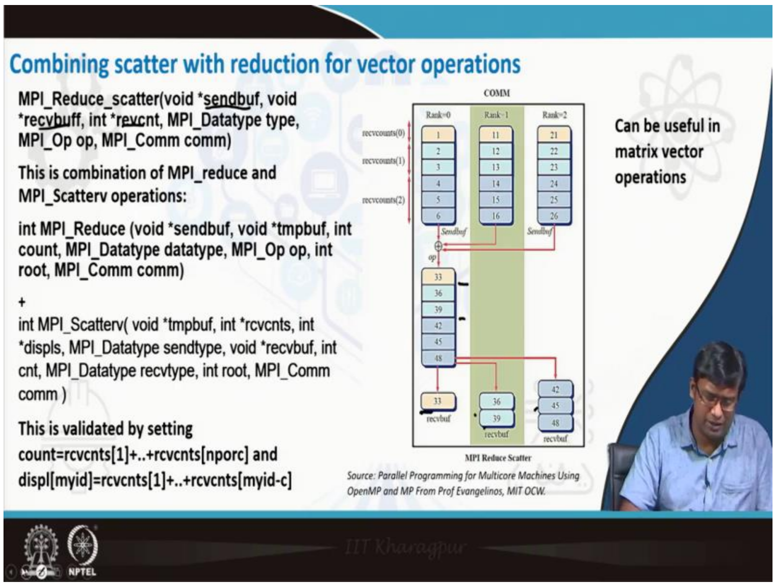
output:

```
[somnath@localhost mpi_examples]$ mpirun -np 8 ./a.out
 a= -0.923879623     a1=   1.00000000     proc id:           0
 a= -0.923879623     a1=  0.923879504     proc id:           1
 a= -0.923879623     a1=  0.707106769     proc id:           2
 a= -0.923879623     a1=  -4.37113883E-08 proc id:           4
 a= -0.923879623     a1= -0.382683516     proc id:           5
 a= -0.923879623     a1= -0.707106769     proc id:           6
 a= -0.923879623     a1= -0.923879623     proc id:           7
 a= -0.923879623     a1=  0.382683426     proc id:           3
```

So, there is an example that a1 is cos(myrank* pi/nproc)and all reduce will take these a1 values and find out what is the minimum of that. a1 is different for different processors, all reduce will find out the minimum of that and all the processors will get the minimum value. So, it finds a minimum of a1 for all the processors local memory and stores it as a.

So, if we execute it in 8 processors, we will get the a1 is different in different processors. It starts maximum 1, minimum is - 0.92 and a which comes after all reduce is - .92 for all the processes.

(Refer Slide Time: 35:23)



Combining scatter with reduction for vector operations

MPI_Reduce_scatter(void *sendbuf, void *recvbuff, int *revcnt, MPI_Datatype type, MPI_Op op, MPI_Comm comm)

This is combination of MPI_reduce and MPI_Scatterv operations:

int MPI_Reduce (void *sendbuf, void *tmpbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

+

int MPI_Scatterv(void *tmpbuf, int *rcvcnts, int *displs, MPI_Datatype sendtype, void *recvbuf, int cnt, MPI_Datatype recvtype, int root, MPI_Comm comm)

This is validated by setting count=rcvcnts[1]+..+rcvcnts[nporc] and displ[myid]=rcvcnts[1]+..+rcvcnts[myid-c]

Can be useful in matrix vector operations

Source: Parallel Programming for Multicore Machines Using OpenMP and MP From Prof Evangelinos, MIT OCW.

So, from all local ids we can find out what is the minimum, what is sum etcetera and these are very helpful in scientific computing exercise. We have earlier discussed pi programming; finding out the value of pi using numerical integration. So, all local sums can be summed up to get a global sum using MPI_reduce or MPI_allreduce. There is another interesting data transfer protocol which is a combination of scatter with reduction and this is MPI_reduce_scatter, this is a combination of MPI_reduce and MPI_Scatterv.

So, what does it do? The different processors have different arrays and then it uses a reduction operation to get a sum of the arrays of all the processors and the reduced array is distributed in the processors using a scatter.

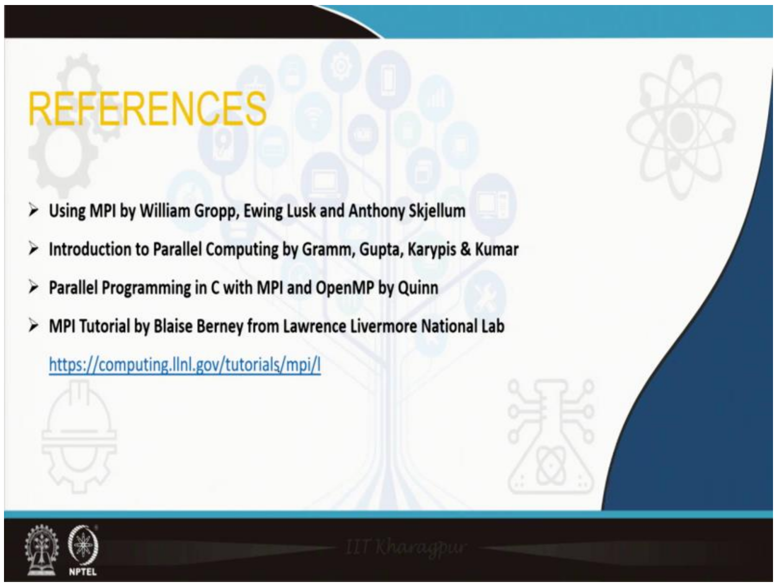
It first uses a reduction, it uses MPI_reduce type and then this reduced type is distributed into multiple processors. Then you can go through it and find out that the number of inputs like send buff is the send array and receive buff is the output which will be given to all the processors ; and these sizes in different processors are different also.

What is the receive count? Receive count is an input, what will be the size of the arrays in different processors that you have to specify that each processor will get this array and based on the received count it sums up the receive count and finds out what is the total size of the array and one by one. So, first processor will get what is receive count for processor 1, then the processor will get what is receive count for processor 2.

So, using that it also finds out the displacement that when first processors data will be finished and next processors data will start. So, this is also a very optimized call and quite well used in matrix vector operations and in matrix algebra calculations.

So, reduction operation can be also combined with scattered operation therefore, multiple MPI collective calls can be combined and final output can be obtained. It becomes simpler as well as anything which is simple to write is more optimized in MPI programming. So, it makes it simpler as well as it is more optimized.

(Refer Slide Time: 38:07)



REFERENCES

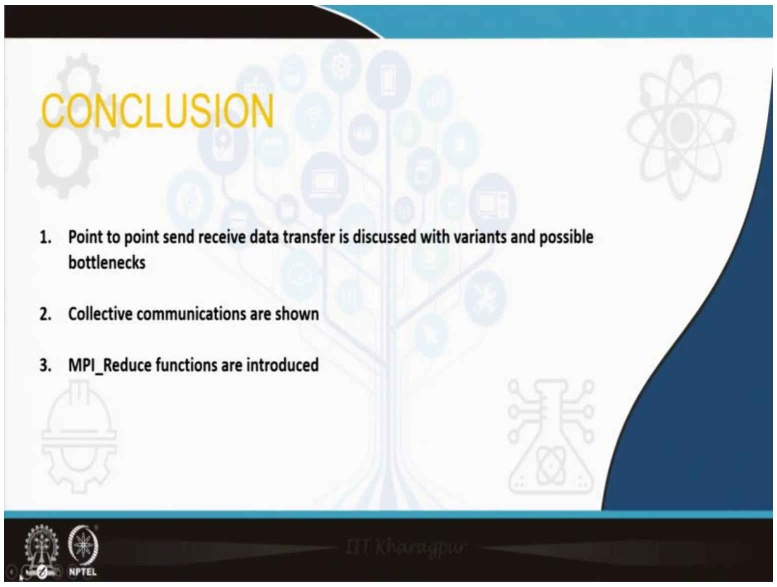
- Using MPI by William Gropp, Ewing Lusk and Anthony Skjellum
- Introduction to Parallel Computing by Gramm, Gupta, Karypis & Kumar
- Parallel Programming in C with MPI and OpenMP by Quinn
- MPI Tutorial by Blaise Berney from Lawrence Livermore National Lab
  https://computing.llnl.gov/tutorials/mpi/l

IIT Kharagpur
NPTEL

We have discussed about a number of collective communication calls also and I said that you can go through the ah Lawrence Livermore Lab tutorial or William Gropps book or any good reference books and you can see number of functions exist on both point to point and collective communication using MPI and all of them has their own utility in optimizing the program and writing the MPI program in a simpler way.

However, again with simple send and receive you can also parallelize any algorithm in distributed computing architecture and write any MPI program.

(Refer Slide Time: 39:05)



CONCLUSION

1. Point to point send receive data transfer is discussed with variants and possible bottlenecks
2. Collective communications are shown
3. MPI_Reduce functions are introduced

IIT Kharagpur
NPTEL

But it is advisable that you do it in an optimized way and use the right collective communication operations. So, we come at the end of communication using MPI. We looked into point to point send receive data transfer with different variable variants and we looked into possible bottlenecks. We looked into collective communications and reduction operations using MPI.

So, I believe that I have given you enough background on starting to write your own MPI program. So, in subsequent classes we will discuss about writing MPI program on simple exercise like numerical integration and then we will go to little involved exercises like matrix vector products and then we will see how we can efficiently write , iterative matrix solvers for large problems using MPI that is that will be the goal while learning MPI for distributed computing

Thank you.