

**Parallel Computing**  
**Prof. Subodh Kumar**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology – Delhi**

**Module No # 03**  
**Lecture No # 15**  
**MPI**

Alright let us look at to this very simple example do you mean to ask the data can get clause no MPI is going to use whatever mechanism the underline network provides for you to do lossless data signing.

**(Refer Slide Time: 00:50)**

**Example**

```
#include <stdio.h>
#include <string.h>
#include "mpi.h" /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv); // start MPI

    MPI_Comm_size(MPI_COMM_WORLD, &numProc); // Group size
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank); // get my rank
    doProcessing(myRank, numProc);

    MPI_Finalize(); // stop MPI
}
```

So it is guaranteed I think I have a slide on that it is guaranteed that data will get there it is guaranteed that it will get there in the order you sent that okay. But whether the receiver is ready to receive or the whether the receiver as went through receive column I have got somebody else data is that should profile and hence your data never got received it will keep waiting that can happen.

But thus end you made will to a given recipient will get there in the order you made so the only real way to check performance is look at all clock time I have got a job to do and you start when the job is done and look at the wall clock done. It can be different and as a matter of fact that is one of the reasons that especially context of parallel programming scalability becomes an

important matter very say what happen if I run it on 2 processors, 4 processors, 8 processors, 16 processors what happen if I run in on a bigger data set okay that is kind of bringing back the notions of complexity.

We will talk about the complexity that will be in the context of the PRAM or something but you will see that there is a correspondence so that if an algorithm is as a lower lemma complexity then its going to be fast on open MP given big enough problem size as going to be fast as MPI as well. Alright so here is the main program it includes MPI.H so basically have to include MPI dot H and later do MPI CC to compile though it is greater things.

And when you do a MPI CC it knows where to get MPI dot H so dash I is added in you do MPI in it okay an MPI it takes arguments we are not going to talk about that. It is checking for the size of it is communicator or the world communicator and it is rank within it okay. So it is a (()) (03:39) to saying in open MP my rank = OMP thread num okay. Here you have to make the function called give the pointer to my RAM and it is going to get written locally.

Similarly it is also get into the group size how many this is all being determine dynamically right though the way MPI run was executed will determine how many elements there are rank is somehow it is going to get assign a unique rank will get assign who is going to be one who is going to be two is in determine. And then (()) (04:21) process after processing is done MPI finish.

**(Refer Slide Time: 04:25)**

## Example

```
doProcessing(int myRank, int nProcs)
{
    /* I am ID myRank of nProcs */
    int numProc; /* number of processors */
    int source; /* rank of sender */
    int dest; /* rank of destination */
    int tag = 0; /* tag to distinguish messages */
    char mesg[MAXSIZE]; /* message (other types possible) */
    int count; /* number of items in message */
    MPI_Status status; /* status of message received */
}
```

And so do processing as a few local variables that the names are clear enough what they do so am not going through them when they used it will be without clear.

(Refer Slide Time: 04:41)

## Example

```
if (myRank != 0) { /* all others send to 0
    // create message
    sprintf(message, "Hello from %d", myRank);
    dest = 0;
    MPI_Send(mesg, strlen(mesg)+1, MPI_CHAR,
             dest, tag, MPI_COMM_WORLD);
}
else { // P0 receives from everyone else in order
    for(source = 1; source < numProc; source++){
        if(MPI_Recv(mesg, MAXSIZE, MPI_CHAR, source, tag,
                   MPI_COMM_WORLD, &status) == MPI_SUCCESS)
            printf("Received from %d: %s\n", source, mesg);
        else
            printf("Receive from %d failed\n", source);
    }
}
```

And this is really main part of what each executable is doing okay it says if my rank is not 0. So everybody whose rank is not 0 so somehow said let 0 become the master everybody else is ((04:59). So everybody else says destination is 0 okay everybody is going to send something T2 the 0<sup>th</sup> process and then it sends the message. Message only says hello from my whoever is sending it what is tag? We do not care as long as everybody using the same tag so we are setting it to 0.

World is being communicated we are sending string length of message + 1 character array and so that is what being used okay. Now P0 says I have got so many different people who may be sending me something say source = 1 one number of processor receives something from that source is not the only way to receive but it is serialize that says am going to read 0 once first then two's then three's which means if three came first better it is going to have way.

And then it is going to print what is got from the message ahh and if MPI receive did not succeed it is going to fail. All MPI methods written success code returns to MPI success which is 0 MPI success will be non-zero then it is succeeded otherwise something went wrong. All this computers open different ports that is not even clear ports are very specific to CPI this may not be running on CPI. Exact details of it I do not know but it would be unnecessary to use different for different fraction okay you can everybody can be connecting on same port and you simply read.

So those are some of the details that remain and will take them off on Monday I want to give you second quiz on memory consistency which is essentially the same thing just a different sequence okay.

**(Refer Slide Time: 07:48)**

**What's the strictest memory consistency model being supported? Why?**

Initial values are U		
<u>thread 1</u>	<u>thread 2</u>	<u>thread 3</u>
x = 0	read y [2]	read x [1]
..	..	..
x = 1	z = 5	read z [3]
..	..	..
z = 3	read x [0]	y = 2

So it is a same question different detail different thread sequences you will have to figure out which is the strictest sequences and you got to figure out which is the strictest consistency model

any questions we are talking about MPI the here is basic chunk of MPI code which essentially tells you all there use to know at high level there are lots of details that you still want to talk about but any questions before we move further from there.

Okay So there is a send so in this case if you rank is my rank is not equal to not always you can create new communicators but my question is if I have two programs both use the MPI code so are they are MPI code. All the once they were started with MPI run okay get their com world and in fact there was question about why some different ports cannot be used actually MPI does use large port number internally.

It use as I said only to get started only to validate you to authenticate you so that your process can begin on the remote machine but then later the communication happens through if it is the CCPIP machine through sockets and there is some discussion on MPI forum about whether should be able to control the port that communication happen that currently there is such control. So it automatically determines it tries sockets ports and if that port is been used as somebody else try some other port it has some juristic which port is going to use.

There is some discussion about whether should have the same user can say use range from 5000 to 5000 in ten something so they do use sockets when you start yes. If you look at the starting program the command when we have an example of starting it said MPI – MP4 said that four times but you can give at the list of machines you can also give host fine okay. You can list machines as well as how many times you want it to run on each machine in a file also.

But yes it is going to do a look up on a host name in order to get to it so MPI does not have default always when it is not or that kind of environment lots of note are there going up and down which means you have to put some layer on top of MPI to allow that to happen and does a matter of fact in any very large for example top 50 or top 100 HPC machines would have something build on top of MPI or they are own from scratch communication layer.

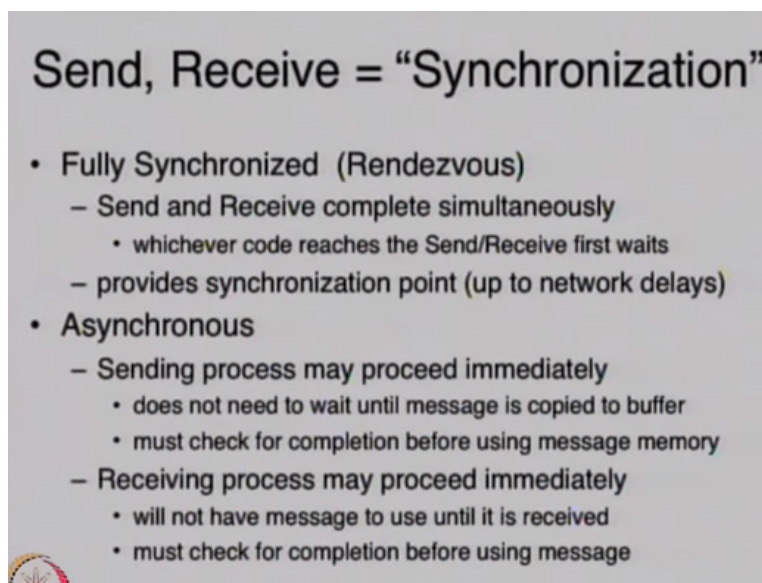
Because you know that once the number of notes come to go very high then you do need to worry about notes going down and come back. But MPI is of course blind to that eventually

everything will stop so things can go on because you are sending someone to host but at some point there is internal handshake that people are waiting for and definitely if you do barrier or things are that sort definitely it is going to hang okay.

So there is send MPI receive and in this case the protocol is that in fact in this example you go through of receipts if you either you are sending to one master or you are receiving from all slaves and so you say for every slave receive from that slave which means that guys sent an your receive are going to matched and then you are going to potentially proceed okay. So there is level of synchronization associate with it and you have to worry about whether we can control that level of synchronization question.

That would require pretty much lock step execution on both sides that is little too stringent just a fact I cannot proceed from this send until somebody received is enough of performance problem right synchronization wise.

**(Refer Slide Time: 14:43)**



**Send, Receive = "Synchronization"**

- **Fully Synchronized (Rendezvous)**
  - Send and Receive complete simultaneously
    - whichever code reaches the Send/Receive first waits
  - provides synchronization point (up to network delays)
- **Asynchronous**
  - Sending process may proceed immediately
    - does not need to wait until message is copied to buffer
    - must check for completion before using message memory
  - Receiving process may proceed immediately
    - will not have message to use until it is received
    - must check for completion before using message

And that is why that is next of things we will be looking at do we need to have full complete synchronization it send and receive and as a matter of fact even that the default and then receive does not provide for that complete synchronization of for that matter wait for complete synchronization what it is though is blocking okay. So we got to look at various aspects if you would have to do what would you do?

I want to send around the data sent to receive somewhere I do not necessarily want to have to wait for. So every send you want to thread that could be lot of threads if you doing lots of communication then potentially locked however there is lots of threads that are waiting you do a send you do compute you do send you do compute another send. So it is computing and keeps generating the sends and so every time if you generate a sends and another thread starts.

What is it give to that thread? However you still need to worry about the synchronization between this two threads now yes so for one is that the communication thread does not have one communication thread that is talking to another remote side where lots of done and so that problem got postponed right. So there is some buffer based management that is required so that you can go on and this buffered data will get sent what was one the receive side you keep receiving something into the buffer.

So some receiver of something keeps receiving it into the buffer and at some point you say am going to take what are the things that can go on buffer overflow. In the sense that you are the original reader so if you might have outsourced your reading and said tell me when you done I have come back and actually read it or you just have to wait because it is not there in the buffer it is not under flow possibility to read it.

So could not find somebody else to buffer it for you or may be it is not there then you wait and read it which was the original thing anyway. So there is a buffer overflow issue which means you may want to control or size the buffer is are there other ways to reach. So somehow somebody needs to be able to receive from multiple right places that's one thing and that somebody may be that communication thread in fact I will have a communication thread ahh that you are suggesting.

So you would not want to respond the thread but MPI will internally have communication thread to which you will be talking but you do need to understand that there is buffer with that communication thread are you provide a buffer every time you send or you say am not going to touch this buffer until you have sent. So there are lots of different ways you might control this

level synchronization one at the highest level of synchronization you say I am going to proceed if that receiver has received it say it is not just matter of communication but a complete synchronization.

Because now that is out of my I know the receiver has happened you can make assumptions about what code as executed on the other side okay. The next may be that I can continue if that guy has started receiving is to a different it take long time to receive but once it has started receiving means it has allocated it has figured out there is so much in order to information coming it has allocated the buffer space for that much communication and it is going to receive it guarantee unless something crashes which we are not contact for.

No these are semantics inside it no need to do that so do that the different variance that it provides. So in this case you want to guarantee that receiving started there is underlined reception that it start only when it can finish many it has enough buffer it has figured out how much data you are sending because it is typically on the header and it is allocated that much buffer it found some much that much buffer somewhere and it is going to receive there are other situation where you say here is this piece of information send it okay.

And some time later come back and tell me that it was received in that case it may never get this because that other machine just could not get buffer that it needed for. So it is at the granularity of the same okay meaning that you said am going to send you one Giga bytes right. So MPI is going to tell the other machine that you are going to receive one Gigabytes you need a buffer of one gigabytes okay.

If that machine is not going to get one gigabytes it is not going to receive it is not going to be able to receive it right it is not even going to start that is end of the story automatically you can be done an opportunity based where you do not allocate buffer you got buffer you start am going to send you one gigabyte you start reading okay and when you run out of buffer we have to wait for the buffer to get empty done read by the or you stop okay.

**(Refer Slide Time: 23:35)**



# MPI Send and Receive

- **MPI\_Send/MPI\_Recv** is blocking
  - MPI\_Recv blocks until message is received
  - MPI\_Send may be synchronous or buffered
- **Standard mode:**
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- **Buffered mode**
  - If no receive posted, system must buffer
  - User specified buffer size
- **Synchronous mode**
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- **Ready mode**
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance

So those all those are possibilities okay what you see here are those possibilities okay there is so one way to differentiate or one axis along which you may differentiate them is that either the function cause send and receive or blocking or there small block. Blocking means something has to happen in terms of safety message being received before you will return before that call will return okay.

Non-blocking is it will return right away so to speak right away means that the underline system will figure out you want to send this you want to send so many bytes you want to copy all those things the information that you are saying send so many bytes of not bytes so many hints from this memory location to that destination it is going to copy those pieces of information and return so it is as if there is not waiting because you are not waiting for some condition to happen there is some local variables the parameters to the send or being copied locally and then the process of sending will begin in the background okay.

That is one access the other access to begin with buffering alright MPI by default has buffers meaning that second third that you are talking about the communication thread has a buffer so when you make a function called that buffer is going to copy from your buffer. Remember when you say something you are providing a pointer you are providing an array full of data and when you provide this array full of data that array full of data will ultimately get copied into an array full of data on the recipient side that is what you go when that is what you want.

When the receiver cause it is receive function with the pointer your data is going to appear there it may go through a sequence of buffers along the array okay. And one buffer one of those will be MPI buffer and if MPU have space it is going to keep in that if does not have space is not going to return your blocking card okay so it so it is blocking in the sense that if MPI is able to copy from your memory location to its buffer area you have done it is safe that your data is going to get send from here you can proceed and reuse that array to write more data whatever you may wish.

And when you do that there is also a need to know that this information has been received because that because the data was copied to your buffer does not mean is going to eventually get it may run out buffer somewhere around somewhere else and not just whether reaches there at some point you may also want to know whether it reach there now okay. So you may want to say high and latency saying this am going to send this piece of data then I going do computation and then am going to wait because I need to make sure that they need that data gone before I proceed at this point.

So at this point there you can you need some to check okay there are two possibilities that this communication thread somehow tells you that yes you have sent something it is gone through or you query. You query saying I have sent this goal if it did not go through either no wait or you go to something else and comeback and check if it go through okay.

**(Refer Slide Time: 28:15)**

# MPI Send and Receive

- **MPI\_Send/MPI\_Recv** is blocking
  - MPI\_Recv blocks until message is received
  - MPI\_Send may be synchronous or buffered
- **Standard mode:**
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- **Buffered mode**
  - If no receive posted, system must buffer
  - User specified buffer size
- **Synchronous mode**
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- **Ready mode**
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance

We will look at how you do those test but the modes that are provided is the standard send and receive it is until the message has been received there is something called standard mode which is implementation stress like the chefs choice okay whatever is going to work best at that moment the implementation is going to chose for you dynamically it can change from cord to cord.

Standard means if you want to you do not want to it is default is going to figure out what it can be otherwise there is buffered mode where you provide the mode this is always some buffer even when it is not buffered mode MPI is going to copied in buffered mode of itself. But you do not know how big that buffer is so you may say am going to sent gigabyte of data so here is one gigabyte array you copy into this and then am going to do non-blocking you are going to say send.

And am going to want to go to something else you return immediately make a copy into there is one gigabyte array that bring you and let me go to something else there are again MPI is the standard and implementation the standard does not anything and all but an implementation choose to optimize and I do not know whether open MPI does or not if you in the provide the same address that you have provided in the send copy.

So buffer says here is the buffer when you can make that copy if the buffer address is the same as the address called you provided in the send then you know that there is no copy here you want to copy to the same place where it is. So it might do the optimization so the standard does not say anything for that it does not going to get rid of its internal variables completely and so need not when you say this much buffer you expect it to use all of it for your business.

There is a synchronize mode which as an N suggest it say something about the receive on the other end the receive is completed and only then it returned will also something called ready mode which is the receive has start receive knows how much data is coming it is going to complete so if I do not need to know that actually has received all the data then I can proceed because I know that because that receive as started everything that was before that received has been done by now.

Okay if that is all the level of synchronization I needed I can say please say the receive is ready to receive then I can go and do something else now okay. If it is ready to receive a and it is buffered right only then MPI is not going to guarantee it is going it is going to have an enough buffer to receive it at only says that to receive function as called MPI s receiving the received call it is looked at the parameters of the received call that is it.

**(Refer Slide Time: 32:52)**

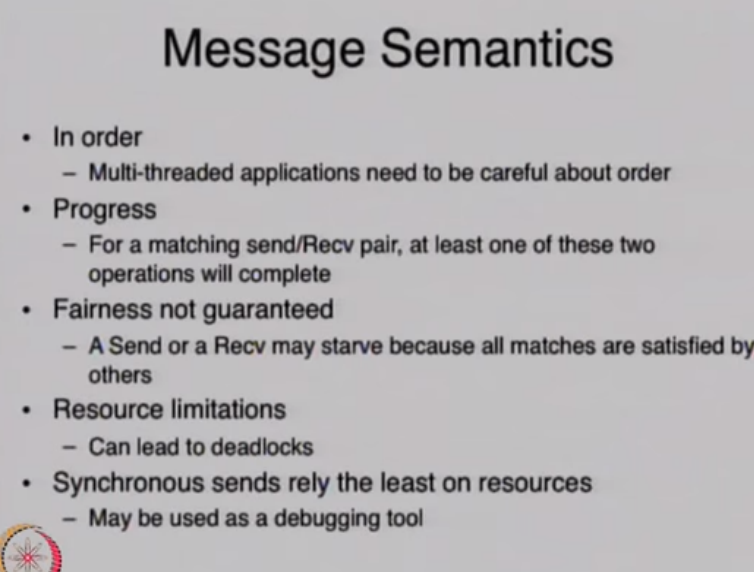
## Function Names for Different Modes

- MPI\_Send
- MPI\_Bsend
- MPI\_Ssend
- MPI\_Rsend
- Only one MPI\_Recv mode

Okay we mean you send has been matched that receive call has been made okay the function names say what they are send is the blocking send we say earlier there is a which is the standard send there is a buffer send for B send the same name and again only giving you the C version the C++ version were it will be MPI\_B send S send is for synchronization R send is for ready send and receive is only one because all of this control is happening on the sender side okay.

Receive can also be blocking and unblocking but the buffering is synchronize is not business of receiver okay blocking and non-blocking will get into one more level of detail one more interns of buffering synchronizing and ready it receive can only say receive now when the data is here return to me or am ready to receive I can go to something else but I have this point and I am giving you buffer where to give me the result.

**(Refer Slide Time: 34:29)**



### Message Semantics

- In order
  - Multi-threaded applications need to be careful about order
- Progress
  - For a matching send/Recv pair, at least one of these two operations will complete
- Fairness not guaranteed
  - A Send or a Recv may starve because all matches are satisfied by others
- Resource limitations
  - Can lead to deadlocks
- Synchronous sends rely the least on resources
  - May be used as a debugging tool

The messages are guaranteed to not be lost again this depends on servers not crashing and all that their guaranteed to be in order meaning that if you send a stream of messages to a given recipients they will be received in order okay there is no global clock there is no global synchronization that I send and some other sender sent then our sends are order in anywhere okay.

There is also progress that if I called send you call receive when the data will get through as long as you providing enough buffer to read it into fairness is not guaranteed what that mean is that if

two you can match sends and receive you can say send to this processor ID process ID 3 and somebody else in the same group same communication may also be sending to process ID 3 and 3 does receive.

This is kind of like race condition in the shared memory context to senders sending to one recipients with exactly the same time right. And the recipient says receive from anybody if the recipient receive from sender number one then one is going to get matched perform and if the recipient says receive from any sender then one of these senders calls will get matched and the other will have to wait for another match.

So starvation is possible in the sense that i do a send and this other processor does 50 sends and that fellow does 50 receives and all the 50 receives came from that one okay even though I did it first is not guaranteed because there is no ordering between us okay it has to do with ensuring that there is enough buffer on the it results. At least one meaning that the sender says I am giving you this much data right and there may be all this different modes and one mode says you do standard do whatever you can.

And on the other side there is not enough buffer to receive everything okay if I keep sending what is going to happen that the user is going to give you an array right so the recipient does not have buffer but the user is given you that one gigabyte buffer okay. So you are going to keep taking more and more off the data and putting it into in your buffer into the recipient buffer right. So at some point either the sender will sent all data and the recipient either run out of buffer right or the recipient has so in this case the sender will end.

When will the recipient end? No I am trying to think why I wrote this it is matching pair is there is matching. The matching pair has been found okay so no now I know I remember what I mean when you send and receive here recipient says I have got the same tag but receive from any right so you send would match with that receive. But has not necessarily been matched to the content right.

Your send also matches that other send also matches if the same receive right which means you too match with each other so either you will be able to send or he will be able to receive he may be able to receive from somebody else which case you are send did not go through but one of these two will go through okay at least one we just talked about the case we are receiving right not go through right.

And send are also brought us although there are more condition some broadcast will talk about that also. Dead locks can happen because there is no direct or no underlined methodology to say that I are going to be before you. You do a send I will get as a receive and when the two things match the data transfer takes place okay. If you send to this fellow and that guy is receiving from that fellow and that fellow is sending to you and you are all blocking then you are waiting for each other okay.

So it is the user it is up to the user to make sure that every thread or process can proceed with means that user need make sure the no dead locks okay.

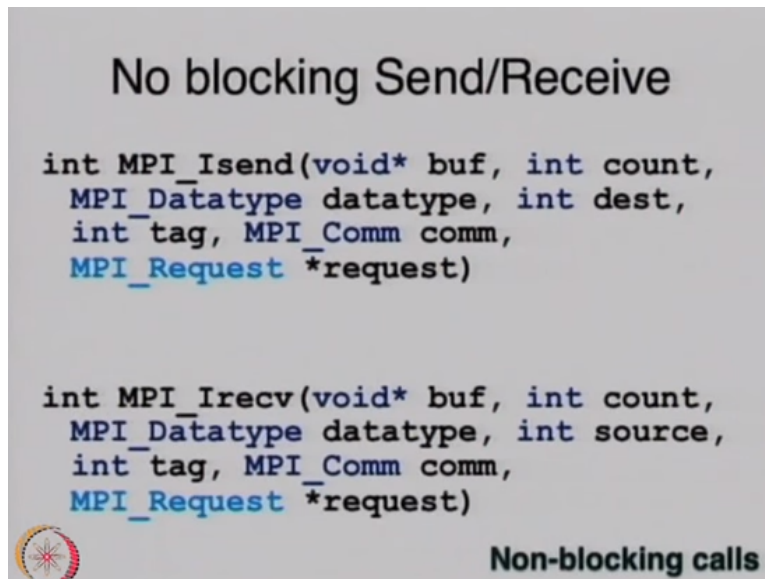
**(Refer Slide Time: 40:57)**

**Asynchronous Send and Receive**

- **MPI\_Isend() / MPI\_Irecv()**
  - Non-blocking: Control returns after setup
  - Blocking and non-blocking Send/Recv match
  - Still lower Send overhead if Recv has been posted
- All four modes are applicable
  - Limited impact for buffered and ready modes
- Syntax is the similar to Send and Recv
  - MPI\_Request\* parameter is added to Isend and replaces the MPI\_Status\* for receive.

Here is the synchronizing version and again all those variants will apply so can say MPI I send MPI IB send MPI IS send MPI IR send for the buffered synchronize and ready send ready version okay. So those all go with I and will look at in fact probably make more sense to look at the function call.

(Refer Slide Time: 41:24)



So interactive send you send a buffer you provide a buffer just like before you give a count say what type of data you are sending you say here is the destination there is a tag there is a communicator and then there is a MPI request type variable right. You allocate a variable send it address and that becomes an identifier handle for this specific request that you are made.

So you can do lots of sends it want to pipe line sends each one will return to you one request and it is request that we use later on to test whether this request has succeeded or not okay. And similarly on the receive side if you want to do non-blocking receive then you must get this request back you want to check whether you receive is done you make a function call will talk about shortly on this request variable okay.

(Refer Slide Time: 42:39)



# Detecting Completion

- **MPI\_Wait(&request, &status)**
  - **status** returns status similar to **Recv**
  - Blocks for send until safe to reuse buffer
    - Means message was copied out, or **Recv** was started
  - Blocks for receive until message is in the buffer
    - Call to **Send** may not have returned yet
  - Request is de-allocated
- **MPI\_Test(&request, &flag, &status)**
  - does not block
  - flag indicates whether operation is complete
  - Poll
- **MPI\_Request\_get\_status(&request, &flag, &status)**
  - This variant does not de-allocate request
- **MPI\_Request\_free(&request)**
  - Free the request

And the function call you make or wait where you say I have done sending now let me see whether my receiver my call that i have made is done or not if it is done good it is not done then wait for it to get done okay. So MPI wait is blocking variant is sent so instead of you blocking at send you do did some other things now you are blocked until your send over there are completed.

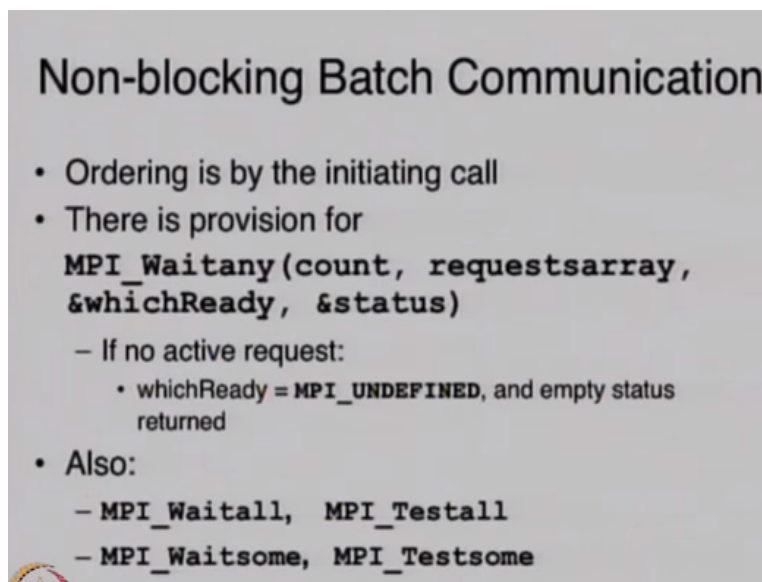
Alternatively you can simply test so test is non-blocking version where it simply returns saying not done yet or say yet it has get done now. Yes it is polling versus sleeping on it somehow we have associate that lock with that send how is it any different there is no of course what can go wrong if I give you these two wait and test that means it probably a dead lock right. You may be start but eventually everybody else will get through and then your if you are not dead locked then your request to received but you would I said that you will be pipeline case.

So how can you wait on one of them so test is whole based you have to keep testing so if you have nothing else to if you have something else to do then testing make sense if nothing else to do then done a loop right We will MPI test is equal to 0 semi colon. But you were not using a resources your process put in a sleep mode and it will be scheduled only when the system as determined that you can be woken up right so it you are not using up any resource at that time you are using memory and all.

But we are not being scheduled to run any of your instructions so it is much more efficient especially if you do locks on and specially if you have lots of processes runs on your version may be doing something else. Even if you do not have anything to do and we have test is been locked. So in MPI wait specially you can do MPI test also there is variant where you can wait for more than one.

And not going to talk about those variants there is MPI request gets status and MPI request is not an variant we have to do it when you created a variable column request that request is filled with information and system is monitoring something about that request. So once you have done you must free that request otherwise it is keep taking MPI memory resource inside MPI and so you will be able to get few or more request done.

**(Refer Slide Time: 46:54)**



**Non-blocking Batch Communication**

- Ordering is by the initiating call
- There is provision for  
`MPI_Waitany(count, requestsarray, &whichReady, &status)`
  - If no active request:
    - `whichReady = MPI_UNDEFINED`, and empty status returned
- Also:
  - `MPI_Waitall, MPI_Testall`
  - `MPI_Waitsome, MPI_Testsome`

You can also separately get the status of the request which is slight variant on MPI test but the function that is little more useful is MPI wait anywhere you say how many things you are waiting for and here is the array of request. So if you want to away for everything is MPI wait all but if you want to wait for some subset then you create a request array and get only waiting done for the course okay.

Wait all means every request that you made is done okay there is some way some which is very similar to wait any in wait any you give it an array and it will return if any of those requests has been satisfied in wait some so you get to say some of them some number of them satisfied okay.

(Refer Slide Time: 48:01)

## Receiver Message Peek

- `MPI_Probe(source, tag, comm, &flag, &status)`
- `MPI_Iprobe(source, tag, comm, &flag, &status)`
  - Check information about incoming messages without actually receiving them
  - Eg., useful to know message size
  - Next (matching) Recv will receive it
- `MPI_Cancel(&request)`
  - Request cancellation of a non-blocking request (no de-allocation)
  - Itself non-blocking: marks for cancellation and returns
  - Must still complete communication (or deallocate request) with `MPI_Wait/MPI_Test/MPI_Request_free`
- The operation that 'completes' the request returns status
  - One can test with `MPI_Test_Cancelled(&status, &flag)`

On the recipient side you also get some information about the message that you are receiving so you can peek into a message that you have not fully received yet okay. and so there is an MPI probe peek into a message and it has got some information that would have about a message and it will tell you something about the size of the messages the message that being receive it and all that you can cancel the request also.

Just in case you do not want it anymore made a request cancel does not necessarily mean it would not happen may be too far down but it is just has not been completed so its request done flag has not set yet but if you see that is not done you can try to cancel it okay it is more request than an ensuring it direct will get answered but then you can test whether it got cancelled or not. Whether it is still went through after you cancel or your cancelled took an effect okay.

I think we probably need to stop at this point take a small break and have a little bit more on MPI send and receive before looking at the other aspects of it.