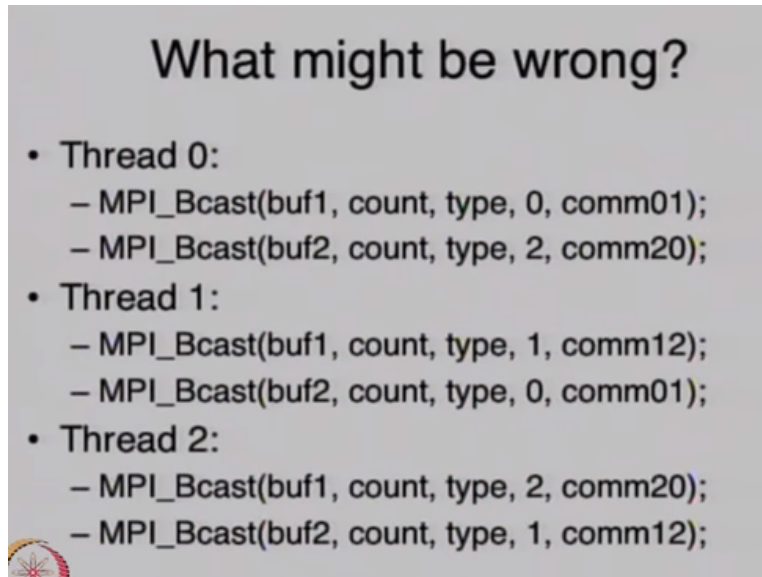


**Parallel Computing**  
**Prof. Subodh Kumar**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology – Delhi**

**Module No # 04**  
**Lecture No # 17**  
**MPI (Contd..)**

(Refer Slide Time: 00:26)



**What might be wrong?**

- Thread 0:
  - MPI\_Bcast(buf1, count, type, 0, comm01);
  - MPI\_Bcast(buf2, count, type, 2, comm20);
- Thread 1:
  - MPI\_Bcast(buf1, count, type, 1, comm12);
  - MPI\_Bcast(buf2, count, type, 0, comm01);
- Thread 2:
  - MPI\_Bcast(buf1, count, type, 2, comm20);
  - MPI\_Bcast(buf2, count, type, 1, comm12);

So what would happen if we send broadcast send is there any additional complication versus just send and receive not that is not right now. So in Bcast to see now we have the ability to say I want to receive from anybody who is broadcasting in my group which we could not happen by simply saying broadcast because there has to be an asymmetric send and receive because send does not have to receive anything right it is look like a send I said Bcast and I do not have to specify a receiver right.

I just say this is the buffer this is that is more so in the current scheme I wish I Bcast then I know that I am going to receive the broadcast from number 3. So those two are exactly the same. So the level of synchrony will go down right okay that is design decision to make. Now when we say we are allowing receive from multiple sources busy multicast broadcast from multiple sources that means potentially I can be broadcasting right more than one member in group may be broadcasting.

So that somebody can receive from him and him so now look that in my point of view somebody has broadcast which I should be receiving and I also have to broadcast which everybody should be receiving should I do broadcast receive for him first or my broadcast send first it does not matter. Because when I say broadcast send that better match with that guy broadcast receive it is little different.

No it does not have to be blocking but the fact that there is a level of order that it gives to you is useful. So right so you can basically raised more and more problems to the programmer and at some point will stop using it is. So basically you are almost guaranteeing a data he have the send and receive and I cannot have a send and receive this can get solved if you have non- blocking issue and you allow to reorder some things.

However if you do have some level of synchrony that you expect from broadcast then it does not make sense from the users point of view to have multiple people broadcasting and you ready accept from anybody right. Because there will have to an order imposed because I am receiving that same broadcast I have to receive that before I send mine so that impose that order being imposed essentially says you do your broadcast and I do my broadcast and then everybody knows that this comes before my broadcast.

So there is no select involved in send and receive you do not have to be receiving the same thing that thing you are sending that the other fellow sending some third fellow can be comparing your send and his send and deciding to choose one of them here you must order yourself with that send also because you have to receive that send. You have all the send because it is a broadcast that other fellow send you have to receive because you are also part of the group.

This is not even that this is that there is third recipient was ready to receive both of us and it does is select but between the two of us it has to be guaranteed that either he sends first or my broadcast happens the first the programmer knows. And if the programmer knows it why would the programmer select it will always say broadcast from that route first broadcast on this route next.

That is the problem that you cannot use this feature if you have this feature why is this is a non-blocking it could have been a non-blocking it is just reducing the level of synchrony that you get it can be implemented with N log in fact that is that may happen in one of the MDS in the feature.

(Refer Slide Time: 08:25)

## MPI\_Gather

```
MPI_Gather(sendbuf, sendcount,
sendtype, recvbuf, recvcount, recvtype,
root, comm);
```

- Similar to non-roots sending:
  - MPI\_Send(sendbuf, sendcount, sendtype, root, ...),
- and the root receiving n times:
  - MPI\_Recv(recvbuf + i \* recvcount \* extent(recvtype),  
recvcount, recvtype, i, ...),
- **MPI\_Gatherv** allows different size data to be gathered
- **MPI\_Allgather** has No root, all nodes get result

Gather so the issue now apply with all we are not going to re-discuss the issues being that you basically want a common primitive that everybody calls that gets matched with each other with the similar type of parameter that broadcast had okay and so you will see some of these example when you are broadcasting you are simply saying send this piece of data everywhere when you are gathering then you are send this piece of data and that is gets filed in these places.

So there is more information to be provided but beyond that same parameters are going to remain for example in MPI gather there is send buff and a receive buff send is what is being sent and gathered at the recipient receive but is only for the recipient right so everybody is sending it is kind of the opposite of broadcasting right everybody is sending and when is receiving the send buff must be have some value for everybody the receive buff can even me serve at the non-gatherers.

At the gatherer it has to have the enough space to keep everybody's data why is then not this is not doubling as received in broadcast there is only one buffer much more space right because you have getting that many copies of send so that is why there is a separate need for a receiver it is opposite of scattered but because we are coming from broadcast right now it is will also be called by every thread.

Every thread will agree that there is the root meaning this is where the gathering is happening and then there is a communication receive buff, receive count and receive type can be junk at everybody except the route. Now it will match only if receiver is ready to receive enough because the buffering is not being provided or it is not going to keep the data in the buffer so that somebody else maybe we read that later on okay.

So just like an broadcast you have to know how much data you receiving here also the route has to know how much data is receiving that means that the sends root sent is important because root is also a sender one of the sender. The call that root makes on its send side meaning the send buff and send count and send type does not have to be the same as all the other members of the group what does have to be? This in fact buffer will not be same as a local pointer what does have to be the same is the size times the type size.

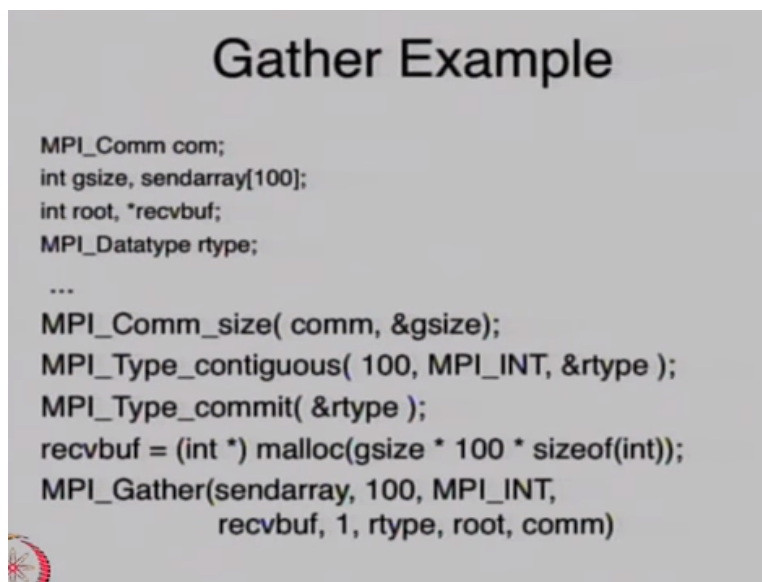
The total amount of information being sent by everybody has to be the same but there is something at the bottom of the slide you can do that there is a gather V where you can gather different amount of data from different senders. But the default gathers you everybody has to send the same amount it can be called a different type and have a different count but first add up to the same thing and the receive type and count has to be match with it okay.

And so I am going to show you an example no because when you built a type you are building it from the primitive type it knows that it is sending you MPI cap and you are getting it in MPI int right. So it is going to collect well it is not going to interpret the hint for you. In fact in that example you may have trouble since you 4 MPI cache and that guys expects 1 MPI int okay. However if it is an a primitive you have built a strut or array based on primitives and

so this guy am sending you 100 ints and you say I am sending a strut which has 100 int fields this that is going to work fine okay.

And all gather is a version where root is not specified because everybody is the root. Everybody still has to make the call which is going to match but the data is going to collected at everybody so everybody must have enough receive also in that case the receive parameters become important and here is the example

**(Refer Slide Time: 14:52)**



```
MPI_Comm com;
int gsize, sendarray[100];
int root, *recvbuf;
MPI_Datatype rtype;
...
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
recvbuf = (int *) malloc(gsize * 100 * sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT,
          recvbuf, 1, rtype, root, comm)
```

The idea for this functions is primarily optimization okay there is slight bit of synchrony that it gives you because of the non-blocking nature of it. So you have to actually compare it realistically with the non-blocking send and receive in which case this is supposed to be faster with non-blocking send and receive you do not know. It depends on whether that non-blocking things allowed you to do something else.

In fact now a days the major installations are on just Ethernet type networks but people still have cube and tree type networks and people do implement MPI on those networks. Because those networks are specially constructed for high performance they are optimized gather an all group communication are optimized there okay. So the top four lines here I have just declaration you declarer a communicator you have in fact it might as well have been the

world comp you get the size of communicator so that you can allocate enough space to hold every members result.

You create an array type so MPI type contiguous creates an array a type which is an array of 100 MPI hints okay. And then you commit the type if you recall we discussed that and the receive buffer is going to be made of enough size to hold G size the membership of group okay. So there is going to be some of data coming from everybody times 100 because you know 100 ends are coming from everybody and size on hint so you allocated this much space.

And now in the MPI gather call in receive buff and buff we are not talking about send array is just array of 100 ends in a receive buff you say I am going to be receiving one data element of our type that means I am going to be receiving one data element of our type from every member of them of the group. And then the root everybody has to be on the route this is the route because other people no need to allocate on that receive buffer so that is what it is any questions?

In MPI hint and hint array no in this case this is guarantee to work perfectly MPI is not a type MPI hint is actually a constant okay it is a constant type for internally so it is not C type I should say it is not a C type it is an MPI struct as defined which says something about how this type has been built and that is those fields really are used only for when you built the type and then for the default type all those fields are not use.

It just internally says this is this type and so do not cast from MPI type to C type for the non-roots just to complete that. They would be send array allocate it will be filled with hundred integers and that is it. The call will look basically the same with receive buff potentially null and those other parameters whatever you care nobody needs to know one element of our type okay from every member two copies of our types from every member which might which could have been done it would have just matched with send of 200 hints.

Committing it because you can build type incrementally so there so every type there is a commit called that you should not need commit you are saying in some cases that is true but

internally it has to be basically set enough resources to make sure that it does not need any more information for this particular type. In this case because you do not have any more to do you are basically saying this our type is done it is only 100 things and nothing more okay.

(Refer Slide Time: 21:45)

## MPI\_Reduce()

```
MPI_Reduce(dataArray, resultArray, count,  
           type, MPI_SUM, root, com);
```

<b>dataIn</b>	data sent from each processor
<b>Result</b>	stores result of combining operation
<b>count</b>	number of items in each of dataIn, result
<b>MPI_SUM</b>	combining operation, one of a predefined set
<b>root</b>	rank of processor receiving data

- Multiple elements can be reduced in one shot
- Illegal to alias input and output arrays

Reduces similar by the way in gather there is a notion of in place where for the recipient there is no send okay. So you physically put your data in the correct slot and receive buffer and so you do not have to allocate a send and everybody else data just comes and lands at the appropriate slot just a minor optimization MPI reduce similar group communication which will have the similar kind of parameter at the end everybody will agree where it is being reduced.

The communicator of course have to be there you will have to provide what is that you are reducing where is the result going to be and the reduction type not necessarily you do something to the type and then you come it. Instead of it keeping track that let us figure what you have done in the last moment it is going to make a copy in its in turn memory about whatever this data type consist of and then you do whatever you want with it right at that point you do not have ensure that this data type is going to remain consistent with.

Because you can incrementally built things in it I think it would be undefined if we did that without committing I should verify but that is what guess is that if you make some change to

it commit or do not commit there may be some hanging things which may be not properly set one. No somebody has to track of what all you are adding in this type right and it is we can it say open we can take a look at what open MPI doing it is probably using this our type as a pointer into something right.

Where things are being modified and once if commit it knows that piece of information is fine now you start modifying our type then either it is modifying that piece of information without actually keeping it in a consistent state in which case something wrong may happen or our type is type which has enough field to store everything about this type okay. I suspect that can never be because our type is an incrementally being constructed.

So when you when you clear our type you do know how many what kind of strut it has right so would have defined in MPI type MP data type this is strut which is a field type which is a field this and the field that okay you need to actually store some piece of information that can be later interpret dynamically and so it would not if you could do that you could say this is an MPI type who was strut says that it has 100 feet 100 hints.

If you could do that you could not then you would not need a commit right but you cannot so it is going to have to something some piece of information and because it is being dynamically there has to be some consistency within it which is probably what it is setting up at the commit time okay. So no the question is why would not able to simply say that does not answer the question for the following reason I keep adding things to this strut that is MPI out of this type which is MPI which is type MPI type MPI later type.

Whenever I call whatever are added to it is what I mean right what is the need for commit and suspect it is because MPI type itself cannot store all the pieces of information because it is dynamically changing you are constructing it incrementally. So that piece of information is stored somewhere and it has to be in consistent state and it is not internally so if you if you looked at MPI type it would not be a type with these specific fields which have all the information about the type you are going to build could not be.



Because you do not know what type you are going to be build if that where the case the it would be enough to simply modify the type because it is right there can call with the given our type okay. In this example that is what it supposed to be one question is should stop answer that and stop one question is I have there is a piece of memory where I can incrementally building this data okay and whenever you say send something with this data type I go and provide this piece of information to their.

So that it can be interpreted but it is not going to be interpreted now it is going to be interpreted at wherever it gets used many times. And so in order for that to become efficient it is probably going to compact way to store all the pieces of types that it has one invisible to the user area of memory.

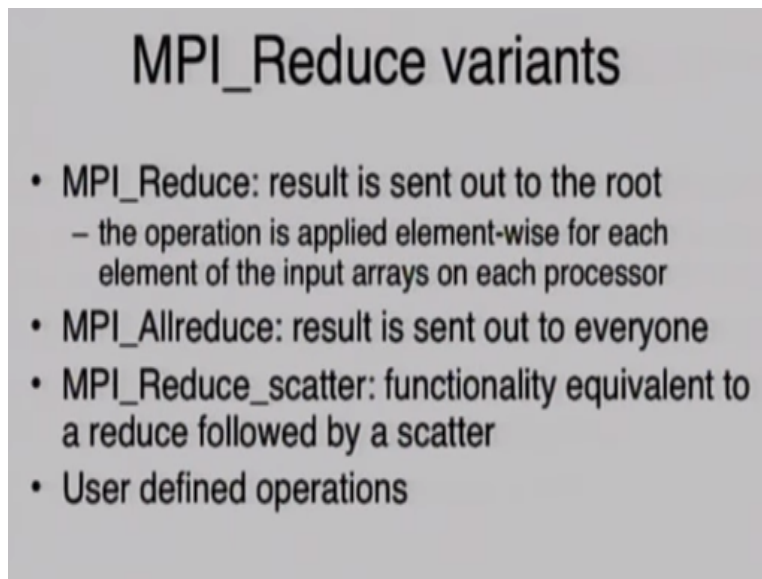
So now when you say I am commit you say I am going to have one more field it is going to be allocating space so that i can keep adding these no pieces of information now that it knows this way this type definition is this big it can figure out how much space this is going to take in mean time you are building some other type also. So you are building many types and whenever type gets committed it is going to be snap shorted over there right that is the guess but that seems logical to me alright.

So let me stop here and the spend some more time I hoping to finish MPI today okay. So are there any other questions on the type okay now admittedly those questions are not particularly important there may be variety of reasons why come it is needed it does not make difference to the programming paradigm programming style learning to program and all that way.

In MPI reduce will provide the array which gets reduced and array where the reduced is result is going to go and how many things are getting reduced the count tells you how many so you can reduce multiple things at the same time. And something that is of type MPI operation okay the predefined operation like MPI sum MPI multiply at saw and stuff of that sort you can also create your own you can write a function and register it as a type.

So whenever it sees that type it is going to it essentially becomes a function pointer and if you do MPI all reduce. There is no array of operators that is being provided and as it says you cannot provide the same variable in the result as well as the input part okay. However there is an MPI in place constant also so instead of the data array you simply say MPI in place and put your result in the result type and everybody else is result is going to get combined in the real array.

**(Refer Slide Time: 32:41)**



And there are MPI all reduce gathered by the way MPI all to all is what I had mentioned all to all is not mentioned of everything to everybody or to all was correct in that diagram that I have shown there is a reduce scatter which is reduce followed by scatter.

**(Refer Slide Time: 33:07)**

## User-defined reduce operations

```
void rfunction(void *invec,  
              void *inoutvec, int *len,  
              MPI_Datatype *datatype);  
  
MPI_Op op;  
MPI_Op_create(rfunction, commute, &op);  
...  
  
Later:  
MPI_op_free(&op);
```

And then if there is also a quick example for how you can create your own function you write a function and you create a OP giving this function as a pointer okay. And later on you can free that up.

(Refer Slide Time: 33:27)

## Prefix Scan

```
MPI_Scan(sendbuf, recvbuf, count,  
         datatype, op, comm );
```

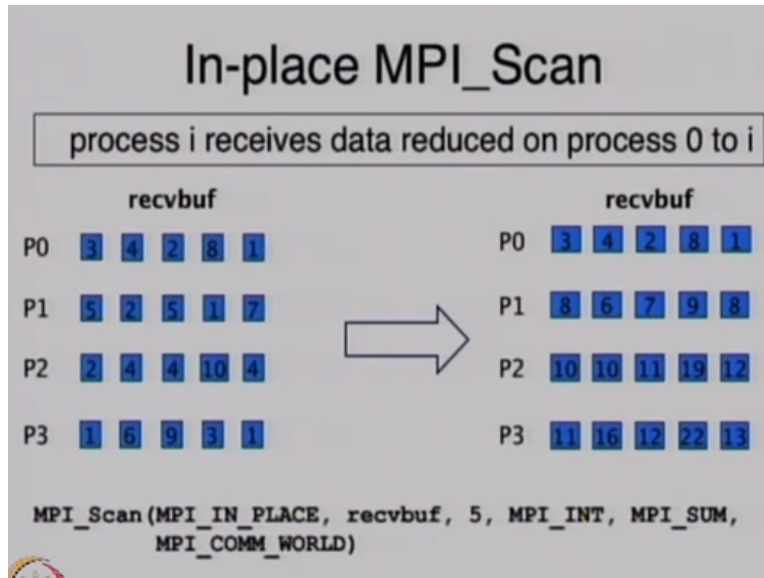
- Performs a prefix reduction on data in sendbuf
  - Multiple prefix ops in one shot
- Returns in the receive buffer of the process i:
  - reduction of the values in the send buffers of processes 0,...,i (inclusive)
- All must agree on op, datatype, count
- There is also exclusive scan:

```
MPI_Exscan(sendbuf, recvbuf, count,  
           datatype, op, comm );
```



There is a scan which we had discussed is the prefix sum again not just sum you can any operation which you can again create and there is an MPI e scan an exclusive where the elements before you strictly before you are reduce to get your result and in MPI scan elements before you including you it the inclusive scan are reduced to get your result.

(Refer Slide Time: 34:11)



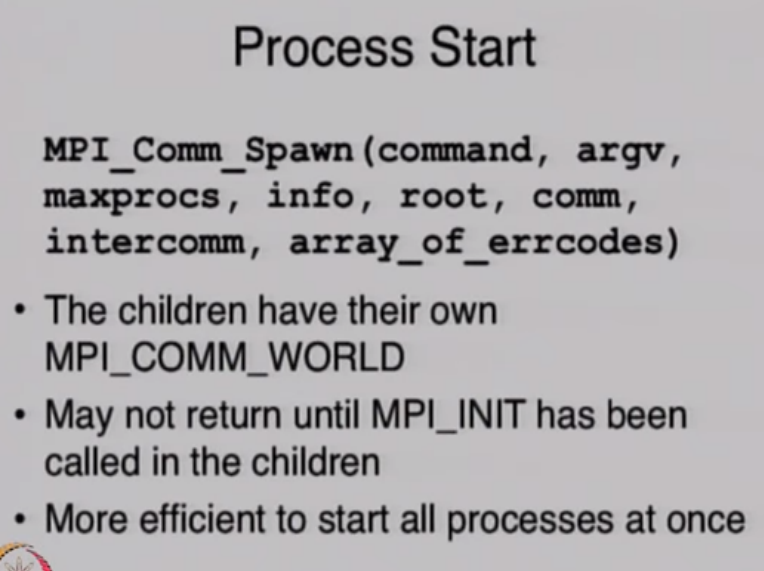
Here is an example in this case it is in place so you how in place is called at the bottom instead of send buff it is just sending MPI in place it is see buff is where the data is going to go and says am going to define elements scanning of 5 elements each is a type INT and the operation is sum so it is prefix sum.

And this is for my entire group which if a it contains 4 processors as shown in this threads shown in this example then receive buff in the beginning everywhere will be what is on the left hand side and receive buff at the end everywhere will be what is on the right hand side away not everywhere. Because prefix sum is up to you so everybody has to get the result of up to you. So P is the processors so P1 in its first position gets the sum of it is first value and P0 first value.

So if you see in so each column is being reduced scanned and so the first column in which started off is 3, 5, 2 and 1 and has turned into 3, 3 + 5, 3 + 5 + 2, 3 + 5 + 2 this one and similarly everywhere all. We will let us talk about that is one of the important algorithm or basic algorithms in basic parallel algorithms that we do need to discuss in more detail later ahh. Because it is instructive and illustrative there are also so we have talked about the basic communication operation so far.

There are also higher level process management operation right so you can say that MPI run at 5 places and it is going to create 5 of them and what if you later decide if you need more you can create more.

(Refer Slide Time: 36:57)



### Process Start

```
MPI_Comm_Spawn(command, argv,
maxprocs, info, root, comm,
intercomm, array_of_errcodes)
```

- The children have their own MPI\_COMM\_WORLD
- May not return until MPI\_INIT has been called in the children
- More efficient to start all processes at once

So you can MPI spawn processors when you create set of processors from one of the threads of from so when I say threads we are actually jump mean unix suppose X threads it means 1 unit of execution so you can spawn multiple threads from a single thread and those threads that you spawn will get when they say MPI would they mean that group okay so the MPI would does not include everybody you can you have create that communicating.

In fact this may be a good time to point out that there also so there is two types of communicator one is called intra communicator one is called inter communicator. Intra communicator is basically what we have been discussing for far we have a group and everybody is doing operation within the group. Inter communicator has two groups it communicates from one groups to another and then it is the group communication is still are defined for that but the semantics is slightly different right.

So when you broadcast inter communicator broadcast means group is in one of the root is in one of the groups and the other group every member is a recipient okay. Similarly for gather and scatter also semantics are slightly different there is also a feature in MPI to think of remote

memory or shared memory and you can this tends to be extremely inefficient but you can create a memory like operation on remote so you can there is something called window with type MPI win and every thread can create a window.

Window means it is creating a window into its own local memory and through that window other process other threads can access this thread local memory okay.

**(Refer Slide Time: 39:35)**

## Remote Memory

```
MPI_Win win;
MPI_Info info;
MPI_Win create (basemem, size, disp_unit,
info, MPI_COMM_WORLD, &win);
...
MPI_Win_free (&win);
```

- Weak synchronization
- Collective call
- Info specifies system-specific information
  - For example, memory locking
  - Designed for optimized performance
- Using `MPI_Alloc_mem` for basemem could be faster

That window is shared right so when you can say I am going to you see how this is created you declared something of type window and you create and everybody in this communication word is probably creating its own it is a base memory some pointer this is how many I am trying to generate info says something about permission and how to access this memory.

And the disp unit is if I recall similar to the memory type size the MPI type size where you use to say 100 hint rather say how many bytes okay. So it is basically saying that i have got base memory is over here it is so many of this type I think it may have to do with slight a generalization on top of MPI types. So that you can simply say that my types are this size this big. So when you want to access something from my memory you have to tell me how many elements you want from which location and you want to multiply form disp unit to get to that position okay.

So instead of you giving me precise offsets you giving me offsets in some unit size okay the unit size maybe the size on N MPI the data type or may not which is a generalization on top it. And this is something that everybody as to so that window when you get your own window variable you have got through that window variable access into all the remote locations okay. So now you can simply say right into somebodies that window will tell you or through that window you will say write into this window that rank ID that piece of information okay.

(Refer Slide Time: 41:54)

### MPI\_Put, MPI\_Get

```
MPI_Put(src_addr, src_count,
src_datatype, dest_rank, dest_disp,
dest_count, dest_datatype, win);
```

- Written in the dest window-buffer at address
  - $\text{window\_base} + \text{disp} \times \text{disp\_unit}$
- Must fit in the target buffer
- `dest_datatype` defined on the `src`
  - should match definition on `dest`
- **MPI\_Get** reverses
  - For the same call: data will be retrieved from 'dest' to 'src'
- **MPI\_Accumulate** performs an "op" instead of replacing at `dest`

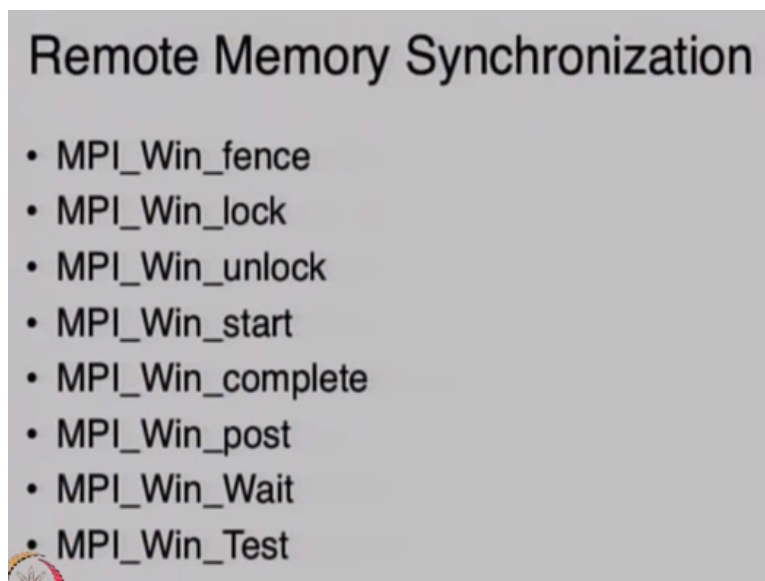
So let me here MPI put is how you would write into somebody else data and in fact I see here it is display unit times display size which is precisely what I was saying. The source address and source count which says here from right from here so many thing to this type to that destination rank in my route destination display is that display or something else I have to take a look displacement yes that would make sense destination displacements the count how many on the destination side and a destination data type.

I am writing hints so many units away from the base memory base offset on that window that rank. So that because you have everybody has called with the same window earlier MPI elopement system is able to figure out which address of which processor are you taking about okay. There is also an MPI get where you can read from another processors memory and other threads memory and there is also an MPI accumulate which is similar to MPI reduce and so except it is one to one so you can say that.

You can write something into a destination memory I can say add this value into this value I have into that guy memory. So instead of read and write with you can expect will be more expensive you simply send it one way saying I am sending it you the performance operation will write it. There is not matching now if you want that thread to know that is programmers business the yes and everybody was doing that right everybody was making this call MPI when create.

And everybody got in response a handle win okay implementation details can vary and through your local variable called win that has enough information to which it was matched the shared memory model is very new model right. Memory consistency model is not well defined for us so the standard does not specify right. We can figure out what open MPI does I suspect it is weak consistency model it may be also be something called release consistency which is similar to be consistency.

**(Refer Slide Time: 45:34)**



But a little easier to implement there are also variety of synchronization because as I said programmer must make sure that if write something there and I that process need to know that I have modified that value I need to be able to synchronize between the two of them. The lock would be within the communicator for the specific memory item like so that we

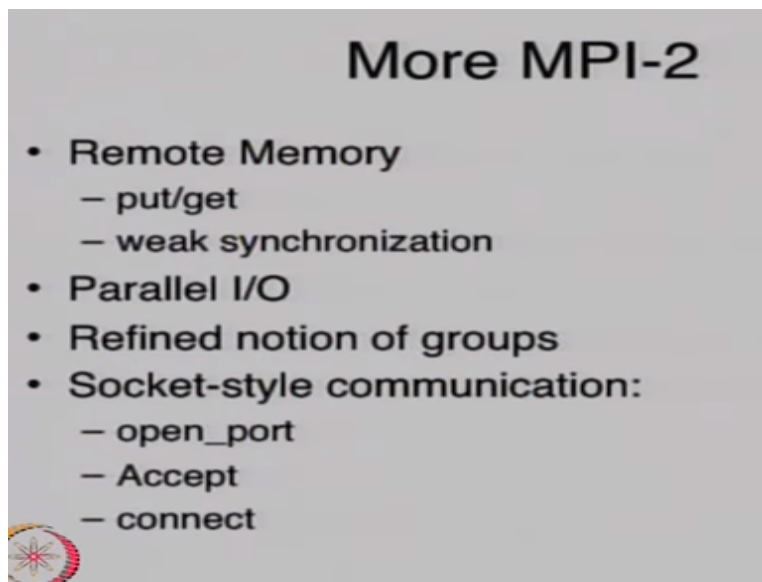


can synchronize it is like shared memory I want to lock this memory so that nobody else can read until I have updated data something of that.

So you are locking a specific shared memory item fence is similar to barrier it says that I have done some shared memory operation MPI puts MPI fence and then do a MPI get that means my put has been everything before the fence has been completed yes it is very high okay. Anyway there are lot more details you can look it up there is little bit more to MPI 2 there is synchronization between their relative reads and writes default read write says what when I print something the standard out goes to the master whoever started it.

Standard ints become null that is a very unsatisfactory model for many situations moreover for file I/O when I am also reading the same file which may be shared using NFS or whatever and the other process other thread is also reading the same file we need to be able to synchronize in some way either you can lock something or just be file operations just like memory operations can be synchronized can be individually synchronized that can be done.

**(Refer Slide Time: 47:47)**



There is also a socket style direct communication if you want something more general than what is being provided then you can open stop okay. That is as much as MPI we are going to talk about MPI is the widest used multi computer because it has so many things and those two are related right this is an open source product why does it had so many things? Because so

many people are building it and so many other people need it that is the thing with open source if there is enough interest then it gets done.

So MPI one did not have any process it was basically a communicator send and receive and then all this meta level features started to get added that is one of the reasons why MPI was not clear winner at MPI one stage. Because it did not have away for you to program multiple things it only had a way for you to tell somebody else to do something now you can actually say create this process run them in parallel produce the result reduce the result and done okay.

So now although not everything is as efficient as it could be it feature wise it can help you to do a lot of things so for shared memory open MP is by far the methodology of choice although there are competitors which are similar in nature many ways and for distributed computers open MP is the methodology of choice okay alright we will stop here we will start talking about algorithms and one of the first we will talk about is this can prefix sum.