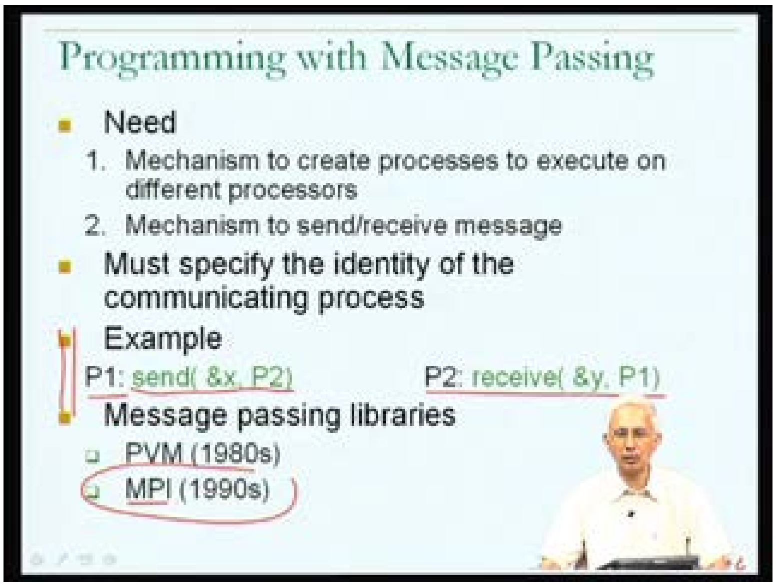
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No # 09
Lecture No # 40

This is lecture forty of the course on High Performance Computing. For the past few lectures, we have been looking at parallel programming. We had a brief look at characteristics of parallel machines. We understand that a parallel machine is a computer system in which there is more than one processor.

(Refer Slide Time: 01:01)



Programming with Message Passing

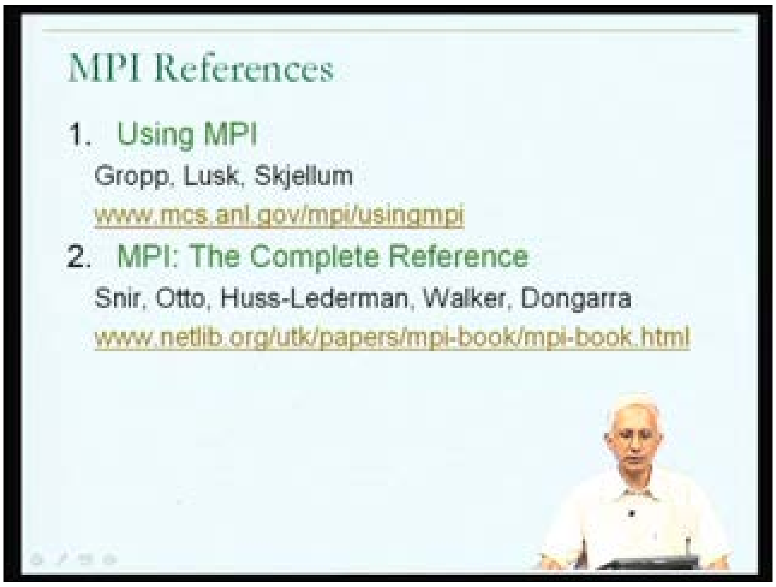
- Need
  1. Mechanism to create processes to execute on different processors
  2. Mechanism to send/receive message
- Must specify the identity of the communicating process
- Example

  P1: send(&x, P2)   P2: receive(&y, P1)
- Message passing libraries
  - PVM (1980s)
  - MPI (1990s)

This could range from something like a cluster of workstations, a cluster of PCs, in other words, a network of computers to a specially designed machine which has multiple processors sharing physical memory. In the process of trying to understand more about how to program a parallel machine not using shared memory ideas, which we saw during our discussion of concurrent programming, but rather the programming of parallel machines using the message passing idea which we had mentioned earlier. But we are now looking at, in more detail. Now, in order to program, to write parallel programs in which the individual processes communicate with each other through message passing, there must be support provided by the underlying operating system which runs on the individual processors of the parallel computer. And this must take the form of what I will call functions provided or supported by the operating system for the explicit sending of data from one process to another such as in this notation which I am using here, this particular notation is not standard to any particular system, which is unfortunate that it becomes necessary for the programmer to learn the specific details of sends and receives as these functions are often called, on the different operating systems or systems on which the parallel programs are to run.

Fortunately, we have message passing libraries which abstract that way making it unnecessary for the programmer to learn about the individual systems, but rather just has to learn the library functions of that particular message passing library. And we are talking about MPI- Message Passing Interface, which is a currently a fairly popular library for the message passing programming of parallel computers.

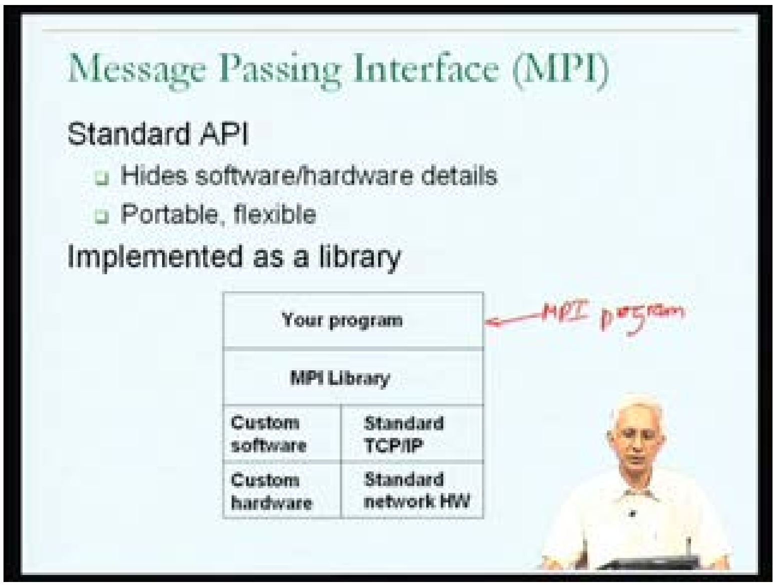
(Refer Slide Time: 02:13)



MPI References

1. Using MPI
   Gropp, Lusk, Skjellum
   www.mcs.anl.gov/mpi/usingmpi
2. MPI: The Complete Reference
   Snir, Otto, Huss-Lederman, Walker, Dongarra
   www.netlib.org/utk/papers/mpi-book/mpi-book.html

(Refer Slide Time: 02:18)



Message Passing Interface (MPI)

Standard API

- Hides software/hardware details
- Portable, flexible

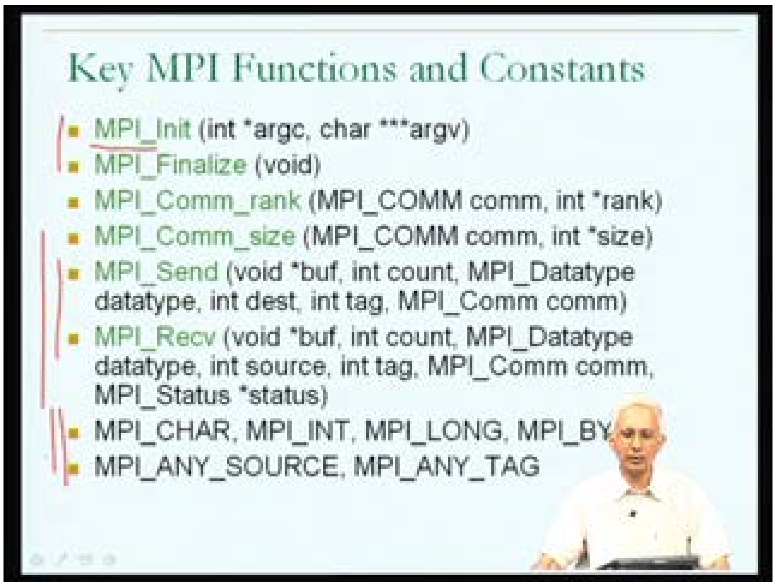
Implemented as a library

Your program	
MPI Library	
Custom software	Standard TCP/IP
Custom hardware	Standard network HW

These are two good references to read regarding MPI. And as I mentioned, MPI is a application programmer interface for writing message passing parallel programs which hides the hardware and software details of the underlying system, thereby making it possible to write portable programs- parallel programs which you could run on one machine and then without any change to the program, run it on another parallel machine as long as both the parallel machine support the message passing interface MPI.

MPI is implemented as a library. So, as far as you are concerned, you write your, what I will call MPI program. Make sure that the system on which you are going to run it, supports MPI. In other words, that the MPI library is available and subsequently regardless of whether the system uses very standard hardware and software or custom designed hardware and software, your program will be able to run on the system which is the portability advantage.

(Refer Slide Time: 03:10)\



Key MPI Functions and Constants

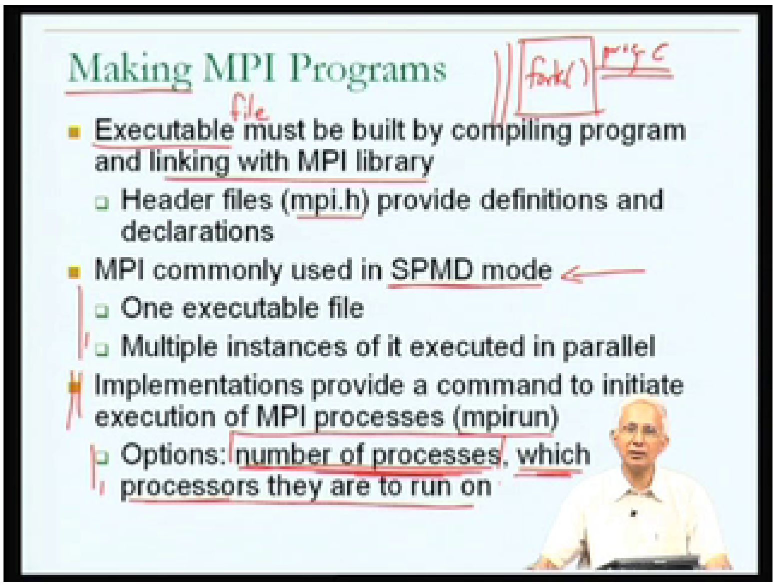
- MPI_Init (int *argc, char ***argv)
- MPI_Finalize (void)
- MPI_Comm_rank (MPI_COMM comm, int *rank)
- MPI_Comm_size (MPI_COMM comm, int *size)
- MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
- MPI_CHAR, MPI_INT, MPI_LONG, MPI_BY
- MPI_ANY_SOURCE, MPI_ANY_TAG

I had briefly looked at some of the key functions and constants of MPI in the previous lecture. We noted that all the MPI functions and constants start with the prefix MPI underscore making it quite easy to recognize them. A program, an MPI program must be written to start by initializing itself using the MPI init function, and as you would possibly imagine, what this function might do is to set up the infrastructure. It is possible that its need to set up some buffers or it needs to set up processes on the individual compute nodes in order to allow the message passing communication to happen.

Subsequently, the programmer must know how to do sends and receives and we must understand the details of the send and receive functions more carefully and in these function calls, one may need to make use of certain predefined constants which are given rather than having to value of the constants 0 1 2 or 3. One could use a name for the constant which gives the program a better readability.

So, we do need to understand more about these functions. We will manage with approximately with an understanding of about ten functions overall which should be adequate to write most of the parallel programs that may be of interest to us.

(Refer Slide Time: 04:24)


Making MPI Programs

- Executable must be built by compiling program and linking with MPI library
  - Header files (mpi.h) provide definitions and declarations
- MPI commonly used in SPMD mode
  - One executable file
  - Multiple instances of it executed in parallel
- Implementations provide a command to initiate execution of MPI processes (mpirun)
  - Options: number of processes, which processors they are to run on

Now, one question which will arise before we move to the individual functions is how does one construct an MPI program. With the discussion that is going to follow in this lecture, we will know how to write an MPI program. But obviously, that MPI program will subsequently have to be compiled or something of that kind before it can be executed on a parallel machine and that is what I am referring to by the word making- how does one construct and write a parallel program and then cause it to be a transformed into version that can be executed on a parallel computer.

Now, the thing which must be noted is that, in compiling an MPI program, one must of course make sure that it is linked with the MPI library functions. Because when you write a MPI program, it will be using the functions that we had seen on the previous slide. And they are not present in your programs, instead you did not write them and they are present in the library which must therefore, be incorporated into the executable version of your program by linking. And this is similar to the way that you had to link the math library if your program was using a math functions.

So, just like there is a math.h header file, there is an MPI.h header file in which the declarations and definitions are contained and subsequently by the linking of the correct MPI.o file, the various functions would be incorporated for use by your program at runtime. Now in the previous class, I had talked about the different versions that a program may take depending on whether one is running the program on an MIMD parallel computer using Flynn’s classification or a SIMD parallel computer. And I had used the notation SPMD standing for Single Program Multiple Data or for Shared Program Multiple Data as a kind of programming model which could be used to program, let us say an SIMD computer.

Now, it turns out that in MPI, people commonly use this SPMD mode. In other words, they will write a single program which generates a single executable; what I mean by an executable is an executable file like an a.out and this executable file will then be run on each of the processors or each of the nodes of the parallel computer. So, it is the same program that is running on the each of the nodes. And we have seen an idea like this when we talked about concurrent programming.

You will recall that we talked about the possibility of writing a program which runs as two or three processes. So, one would write a single C program, let us say, and one would compile it and then one could run the program. In the context of concurrent programming, there was no question of running it on more than one processor because we were talking about a situation where there was only one processor, but the program could run as multiple processes because we may have included a fork system call, which cause a new process to be created when the program was executed. So, in that sense, what we were doing in that form of concurrent programming was to write a single program which could run as many processes. And the idea that we are talking about over here, is similar.

Rather than writing a separate program to run on each of the nodes of the parallel computer, we write a single program which contains information about what all the different processes which are cooperating towards the common objective are going to do. Along the lines of what we did in the case of the concurrent program, you will recall that, in case of the concurrent program, we looked at the return value from fork to determine whether the process which was executing at that point in time was the parent or the child process and depending on the return value we would then move to a different part of the program to do what the child had to do over the parent had to do. And something this similar could be done for an MPI program where, depending on the identity of the process which is running the program a different portion of the program could be executed thereby allowing many different activities to be described.

So, this is one of the more popular modes of writing MPI programs. MPI programs could be written in different modes as well, where you have different executable for each of the processes. The simplicity one gets from this is that, there is a single executable file which could be run on each of the nodes. Now, some multiple instances of it executed in parallel; one instance of this executable running on each of the processors of the parallel computer.

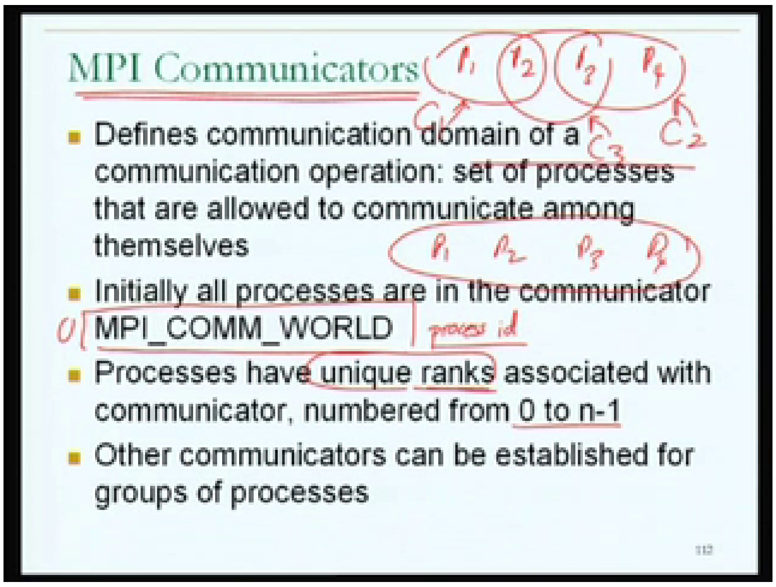
Subsequently, there is the other issue which we have talked about last time. How does one actually cause, let us suppose, I have a parallel computer with one thousand nodes then I would have to cause this executable file to run on each of the thousand nodes which would take a long time for a human being to do. One would have to type a.out on the shell prompt of each of the one thousand processors constituting the parallel machine which would take a long time.

So, fortunately the implementations of MPI will typically provide you with a command which is often called MPI run, through which you would cause the initiation of the thousand processors to happen through the services of MPI run; to the single executable file a.out would be taken as input by MPI run and run on all the thousand processors. Now, depending on how many processors you want use on the parallel computer, you could provide inputs to the MPI run.

So, the options to MPI run will typically be the number of processes that you want the MPI program to run as, and if you have sufficient knowledge about the parallel computer, specifically which processors of the parallel computer you want the different processes to run on. As you can imagine, to specify a processor of a parallel computer, one would have to have processor IDs of some kind and therefore, its more often useful to try to think of scenario if one was writing an MPI program to run on a network of computers in which case each of the computers in the network would have its own network address or IP address and one could possibly specify the individual processors in that form by the IP address of each of the processors in the parallel computer. But these are the two options which MPI run, a typical implementation of the MPI run would take- the number of processes that you want to program to run as and further possibly the specific processors where you want the processes to run.

Now, this is an interesting idea because it suggests that you could write an MPI program without taking into account the actual details of how many processes that is going to run as. So, you could write an MPI program so that it runs as some n-processes and then you could actually run it as sixteen processes one day or as thirty-two two processes the next day without actually changing the program, but by just providing the correct information about the number of processes in the execution of the program through MPI run.

(Refer Slide Time: 11:00)



MPI Communicators

- Defines communication domain of a communication operation: set of processes that are allowed to communicate among themselves
- Initially all processes are in the communicator MPI_COMM_WORLD
- Processes have unique ranks associated with communicator, numbered from 0 to n-1
- Other communicators can be established for groups of processes

112

P1
P2
P3
P4
C1
C3
C2
P1
P2
P3
P4
0
process id

Now, one of the important concepts in MPI is what is known as the MPI communicator and you will recall that, when we talked about send and receive, I pointed out that it is critical that the identity of the sender and the identity of the receiver must be known because, in doing the send, the program of the sender must mention the intended recipient. So, there is this need to have the identities specified within the program and therefore, to distinguish between the different processes constituting the MPI program and central to this is a notion of a communicator.

Basically, a communicator is a mechanism to define a communication domain for a given communication operation, essentially the set of processes that are allowed to communicate among themselves. So, an MPI program could be written to have many different sets of processes depending on what communication requirements of the program are. Let me just give you a very simple example, that suppose if I am writing a parallel program which I know is going to run as four processes, but I know that in achieving the common objective, it is sufficient for process P 1 and process P 2 to communicate with each other. There is some data which must be communicated between P 1 and P 2 possibly both ways. Further, I know that it is necessary for process P 3 and P 4 to the communicate with each other and it might be the case that, I also know that towards the end of the program, process P 2 and P 3 have to communicate with each other.

So, I know that there these three pairs of communicating process that have to be achieved through the MPI program. Now, one way that I could handle this is by setting things up so that, all the processes allowed to communicate with each other. An alternative is that, I could use this idea of the MPI communicator and note that for the earlier parts of the program, I have to have one communicator which I will call- c one, which is defined as the communication for the communication between P 1 and process P 2. I could define another communicator which defines communication to in process P 3 and process P 4 and possibly a third communicator which indicates communication between process 2 and process 3, thereby avoiding the necessity for declaring all the processes as being part of one communicator which could potentially have to communicate with each other. But the communicator itself is just this. They construct within MPI through which the system keeps track of what processes may have to communicate with each other.

Now, the way that things are set up is that, when you write of MPI program by default, initially, all the processes will be assumed to be part of a single communicator. And we can refer to that single communicator as MPI communicator world. The word world here is suggesting that it includes everything; all the processes which will come into existence when this program runs.

So, in in effect, the default will be that the program, an MPI program would run as with a single communicator. In other words, with a possibility that any two processes on the system can communicate with each other, since all the processes form part of single communicator. This would have been the case, in our example, If I had not tried to declare communicators, but I had just used the default, in which case, when I run the program as four processes, they would all be part of MPI comm world, the initial default communicator and therefore, any two processes in this set a four could communicate with each other.

Now, within a communicator, each process has what is called a unique rank. And I had use this term when we looked at the list of functions in the previous lecture. Essentially, what the idea of the rank is going to do is serve the purpose of the process identifier. Recall that, in doing sends or receives, one had to, there was a need to specify the intended recipient by its identity and I had pointed out that, using the a Linux or Unix process ID which is unique on one processor, but not possibly unique across processors; since Unix may or Linux may keep track off on a given processor may keep track of the different processes that were created for execution on that processor, starting with P IDs going from zero upwards and similarly on other processors, the same P IDs might be used for the local processes.

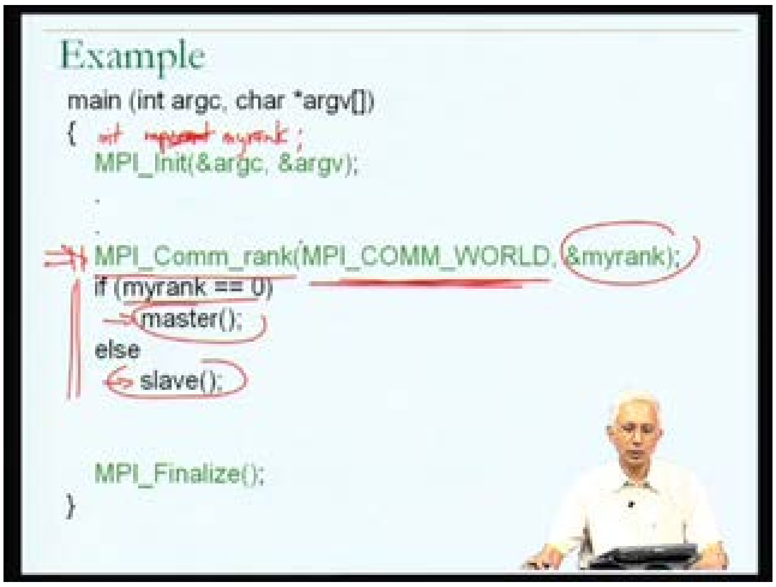
So, the notion of rank, abstracts, process identification to a level beyond the individual processor. And the way to look at this is, on a given, within a given communicator, the different processes which have communicating with each other through the communicator are assigned ranks numbered from 0 through n minus 1.

So, if there is a communicator with four processes, then to have one processes whose rank was 0, another with rank of 1 and so on; assuming that there are n process constituent as parts of the communicator. And as I had mentioned, other communicators could be established for other groups of processes. So, long the lines of the upper example which I had used.

So, the two important concepts we see here are, one is the concept of the MPI communicator with the default of MPI comm world. Secondly, the notion that every process which comes into existence when an MPI program is executed will have the unique rank. Now, I should point out right now, that MPI comm world follows the syntax, the notation that we had seen for the various MPI functions and for the various MPI constants. Its starts with the prefix MPI underscore and from this notation you would understand that basically MPI comm world is one of the MPI constants.

So, just think that it might it might be a constant, let say with a value of 0. So, just think that the different communicators associated with an MPI program each has a unique constant associated with it and the default communicator has the constant, whatever the constant, MPI comm world relates to or corresponds to. The MPI comm world is one of the MPI constants which in fact, was on the list that we had seen.

(Refer Slide Time: 16:56)



Example

```
main (int argc, char *argv[])
{  int ~~myrank~~ myrank ;
   MPI_Init(&argc, &argv);
   .
   .
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   if (myrank == 0)
     master();
   else
     slave();

   MPI_Finalize();
}
```

Now, let us just look at a very simple example of an MPI program, just to make sure that we understand how simple it is to write an MPI program.

So, basically I am going to be using that SPMD. I am going to write an example where there is a single program which is going to be run on all the processors, the one process per processor possibly of the parallel machine.

So, I am writing the single programming in C. So, I am showing you this main of the single C program. You recall that any MPI program must start by doing the initialization by calling MPI init. So, that is the first thing that I include in my MPI program and any MPI program must start by end by finalizing.

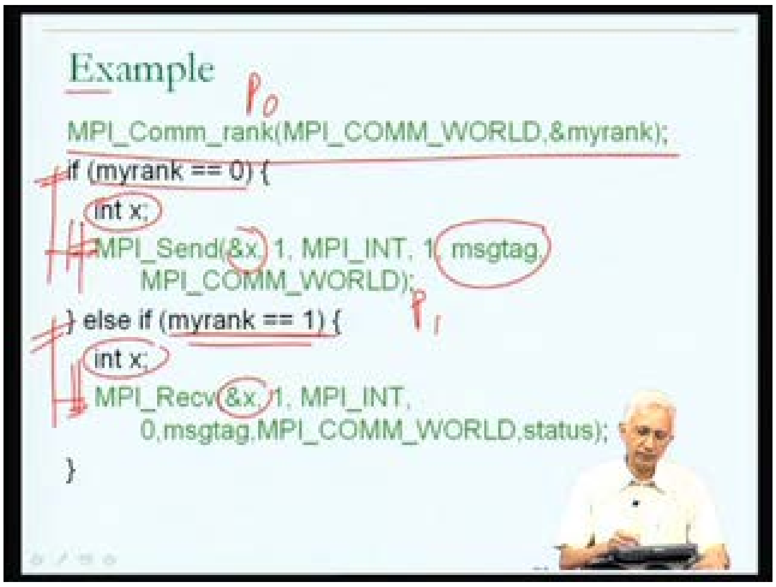
So, I put that is the last thing, the last component of main. In between, there could be various kinds of activities that different processes do, based on their activity as required by the common objective. The same program is going to run on each of the processors which I specified through MPI run. And therefore, very clearly, in the code that follows I have to set up things so that, the word executable as when running on one processor, possibly running as process with a rank 13 will do different things from the same executable running on another processor, possibly running as process number 27; the process number 27 in terms of its rank. Therefore, what follows must first of all, make certain determinations about this particular execution of the MPI program. For example, how many processes are there in this particular MPI program and there is an MPI function through which the program can determine how many processes this program is running as. Further, there is a function which can be used by any particular process to determine what its own rank is. So, we had seen MPI comm rank; a function through which an MPI process can determine what is own rank is. So, in this particular call to MPI comm rank, there is a need to the two parameters, one is the particular communicator that one wishes to know what the rank is and then one passes a parameter into which the return value, in other words, the rank of this process which is running, which is executing this particular function, will be returned in the variable MPI rank. So, MPI rank would have been declared as a variable, I am sorry, my rank would have been declared as a variable, an entity variable in this program. So, after calling MPI comm rank, the value of that variable will be the rank within the communicator MPI comm world of the process which is running this particular instance of the program. And if I am running this program on each of hundred processes or each of each of a hundred processors in a parallel computer, then each of the processes running on the hundred processors would receive a different value as its my rank.

So, you can also see that its possible for one process to be part many MPI communicators and it could determine its rank in each of those communicators using, by putting the correct number of the communicator into the first slot of the call to MPI comm rank. Subsequently, depending on what this particular process find its own rank to be, for in this particular example over here, I am sort of assuming that if there are a hundred processes, then one of them is going to be doing important work which I designate as the work done by the master process and the others, the remaining ninety-nine are going to be doing very similar work which I will designated as the work done by the slave process and it looks like I am setting up this MPI program, so that, after each of the hundred processes has determined what its own rank is, each of them then proceeds to determine if its rank, to check if its rank is equal to 0 and if its rank is equal to 0 it will run the function called master.

Now, only one of the processes will therefore, run the function called master. But if it finds that its rank is not 0, then it runs the function, it executes the function called slave. And therefore, in writing this MPI program, I would write a function called master which contain the functionality to be done by the mater process. and would write a function called slave which contains the C functionality describing what is to be done by each of the remaining ninety-nine processes in order to achieve the common objective.

So, this is the general structure of this SPMD, a single program which we will be run and each of the hundred, in these example, processes of the parallel machine to achieve the common objective. So, this was just to illustrate the notion of rank and communicator and how the rank of a process can be determined by the process.

(Refer Slide Time: 21:22)



Example

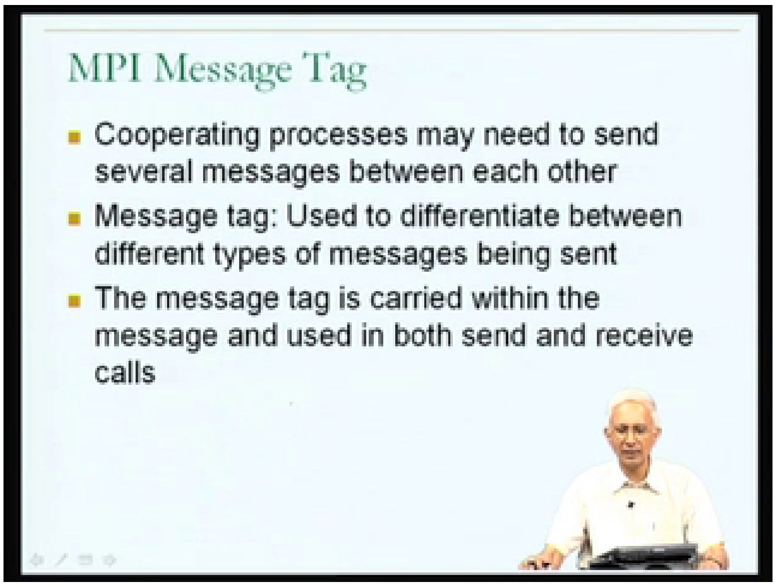
```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0) {
    int x;
    MPI_Send(&x, 1, MPI_INT, 1, msgtag,
        MPI_COMM_WORLD);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT,
        0,msgtag,MPI_COMM_WORLD,status);
}
```

Now, just to complicate the example a bit, very clearly the reason that we want the processes to know the rank is partly so that, they can each end up doing the right work. For example, the master process does a master work and all the slave processes do the slave work.

But at some point in doing this work, there may be the need for one process to communicate with another process and that is where the calls to MPI send and MPI receive will enter the picture. So, in this particular example, we once again have, I am not having, I am not showing you the call to MPI init and MPI finalize. In this example, we are assuming that this is just an extract form a larger program. But in this example what we are doing is, trying to see how two processes could send; one process could send data to another process.

So, once again I am assuming that each process knows its own rank. And if process can find out its rank by calling MPI underscore comm underscore rank, subsequently that the whether I say things up is, I know that the process whose rank is 0 is supposed to send the data to the process whose rank is 1. Therefore, in the program as I write it, there is a part of the code which checks if the rank, if the process finds out that its rank is 0,then it is the process which must do the send. On the other hand if the process finds out that its rank is 1, it is the process that must do receive in connection with this communication. And the parameters of send and receive, you will see about a little bit shortly, but you will notice that in common, if you if you look at the way things are set up, the process which is going to do the send has a local variable called x and it is the value of that local variable which is apparently going to get send. The process which is going to do the receive also has its own local variable called x which is the variable into which it is going to receive the value. Subsequently, if either of these process receives first of the variable x will actually be referring to a variable that had the same value at least soon after the send and the receive were done. But this is how a send and receive pair could be set up for process 0 to send to process 1. And in general, prior to this, we were thinking of process 0 as operating system process 0 and operating system process 1. Now, we just think of them as the process who happens to have a rank of 0 when the MPI program was run and the process which happens to have a rank of 1.

(Refer Slide Time: 23:49)



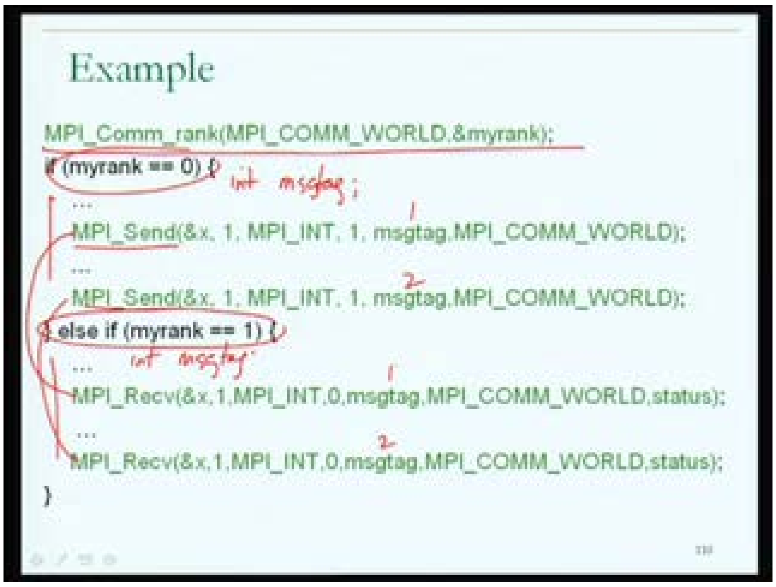
MPI Message Tag

- Cooperating processes may need to send several messages between each other
- Message tag: Used to differentiate between different types of messages being sent
- The message tag is carried within the message and used in both send and receive calls

Now, one of the parameters in call to send and the call to receive, you would have noticed, was a parameter with the name m m s g tag which is actually referring to something called a message tag. This is another important concept in MPI communication, the idea of the message tag. And basically, the reason that a message tag is necessary because it is quite possible if you are writing as non-trivial program that processes which are communicating with each other toward the in terms of needing to corporate may in fact, need to send more than one message to each other. So, it may be necessary to process P 1 process with a rank of 1 to send one message to process P 2 early in the program and send another message to process P 2 a little later in the program.

So, the message tag is one of the parameters in the calls to send and receives. And there is a facility through which it is possible to differentiate between different messages that are being sent between let say, the same pair of processes. So, the message tag value is carried along with the message in both and used in both the send and the receive calls, and provides this capability of distinguishing between two messages that was sent from the same sender to the same recipient and that is what we have illustrated in this example.

(Refer Slide Time: 24:53)



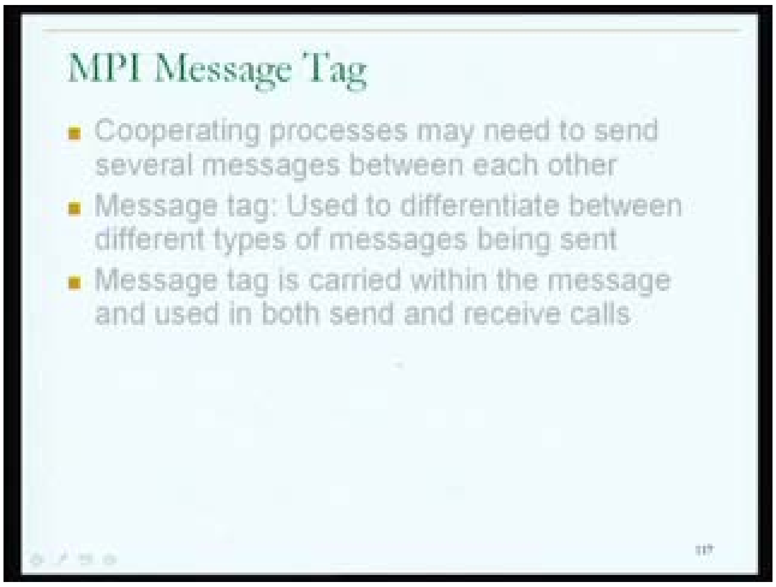
Example

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0) {   int msgtag;
  ...                           1
  MPI_Send(&x, 1, MPI_INT, 1, msgtag,MPI_COMM_WORLD);
  ...                           2
  MPI_Send(&x, 1, MPI_INT, 1, msgtag,MPI_COMM_WORLD);
} else if (myrank == 1) {
  ...        int msgtag         1
  MPI_Recv(&x,1,MPI_INT,0,msgtag,MPI_COMM_WORLD,status);
  ...                         2
  MPI_Recv(&x,1,MPI_INT,0,msgtag,MPI_COMM_WORLD,status);
}
```

So, once again, I am showing an example, an extract from a larger MPI program. Here the objective is that, after the processes have determined what their ranks are, process with rank 0 is sending data to the process with a rank 1. So, process with rank 0 does certain things, the process with rank 1 does other things. In this particular example, the process with rank 0 is doing two sends; first this send and that send to the process with rank 1. And in order to distinguish between these two sends, it could use a different value for the message tag variable and once again, message tag is a variable local to the process with rank 1 which could be set up with an the integer variable called message tag.

So, it is possible that message tag the first message is sent with the message tag of one and second message is send with a message tag of two. Similarly, the first receive could be set up to receive with a message tag of one and the second receive could be set up to receive with a message tag of two and the consequence of doing this is going to be that, whatever was sent by the process with rank 0 in the first message will be received by the process with rank 1 in the first receive. And whatever was sent by the process with rank 0 in the second send will be receive by the process with rank 1 by the second receive, and that will avoid possible confusion about what which data is going to be received by, which data sent is going to be received by which of the receives. And we should notice that there is a possibility and a very complicated parallel machine that the two send messages may actually not be received in the same order, such all the more important to have this message tags to distinguish clearly between, the data sent by the one send and the data sent by another send.

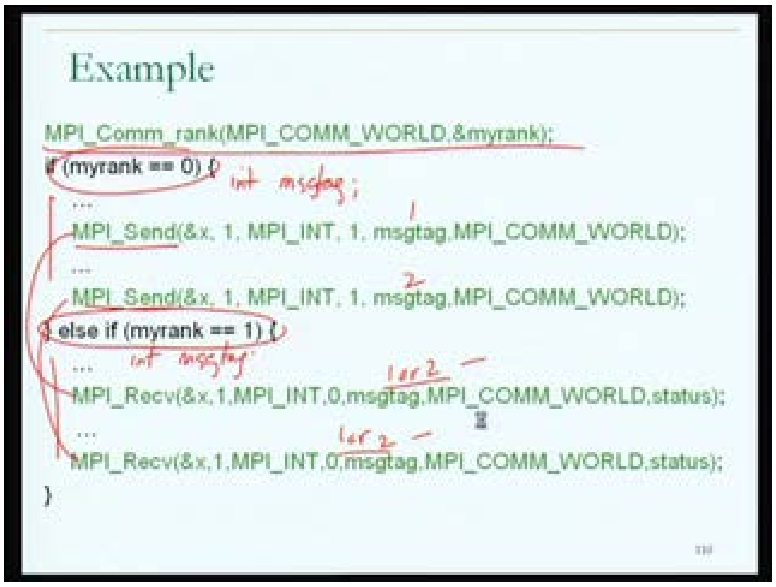
(Refer Slide Time: 26:46)



MPI Message Tag

- Cooperating processes may need to send several messages between each other
- Message tag: Used to differentiate between different types of messages being sent
- Message tag is carried within the message and used in both send and receive calls

(Refer Slide Time: 26:54)

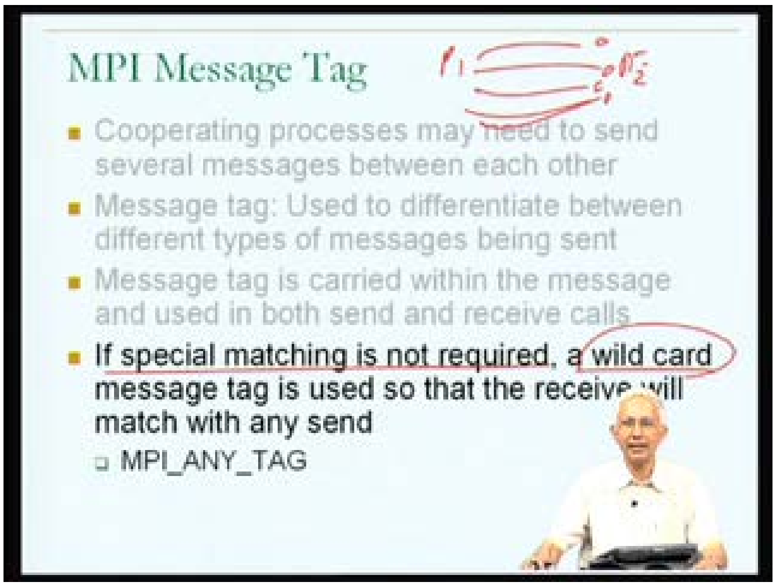


Example

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0) {      int msgtag;
  ...                                    1
  MPI_Send(&x, 1, MPI_INT, 1, msgtag,MPI_COMM_WORLD);
  ...                                    2
  MPI_Send(&x, 1, MPI_INT, 1, msgtag,MPI_COMM_WORLD);
} else if (myrank == 1) {
              int msgtag
  ...                                  1 or 2 —
  MPI_Recv(&x,1,MPI_INT,0,msgtag,MPI_COMM_WORLD,status);
  ...                                1 or 2 —
  MPI_Recv(&x,1,MPI_INT,0,msgtag,MPI_COMM_WORLD,status);
}
```

Now, so, the idea of a message tag, we understand, we should note that it is conceivable that, in a particular program, you may have setup the programs so that it is not critical, that the messages be received in a particular order and that you do not really care whether the first receive receives the data sent in the first send, over the second receive receives the data sent by the second send. So, if you if the application is such that you do not really care, unfortunately there are no option of mentioning message tag one or message tag two in both of these locations.

(Refer Slide Time: 27:24)



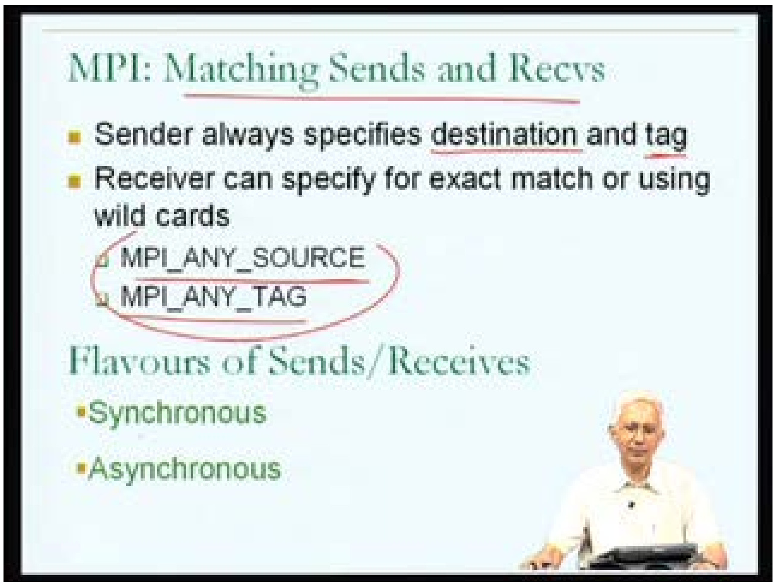
MPI Message Tag

- Cooperating processes may need to send several messages between each other
- Message tag: Used to differentiate between different types of messages being sent
- Message tag is carried within the message and used in both send and receive calls
- If special matching is not required, a wild card message tag is used so that the receive will match with any send
  - MPI_ANY_TAG

And therefore, MPI actually give us an option for message tag which is what is called a wild card. It is basically wild card indicating that it is not a specific value of message tag, but some general kind of a notation for a value. And this is to be used in the special case where we do not really want to match a particular send and a particular receive, but do not mind which data is received by any one of the receives in process P 1 in our example.

So, this case one would include as the message tag, is the MPI constant MPI any tag. MPI any tag is what is referred as a wild card to be used in cases where one does not really care what the value of the message tag is. In another words, for the many communications between process P 1 and process P 2, we do not mind which order they are received by the process P 2 and therefore, all of the receive MPI calls within process P 2, we could use the MPI any tag as the message tag.

(Refer Slide Time: 28:20)



MPI: Matching Sends and Recvs

- Sender always specifies destination and tag
- Receiver can specify for exact match or using wild cards
  - MPI_ANY_SOURCE
  - MPI_ANY_TAG

Flavours of Sends/Receives

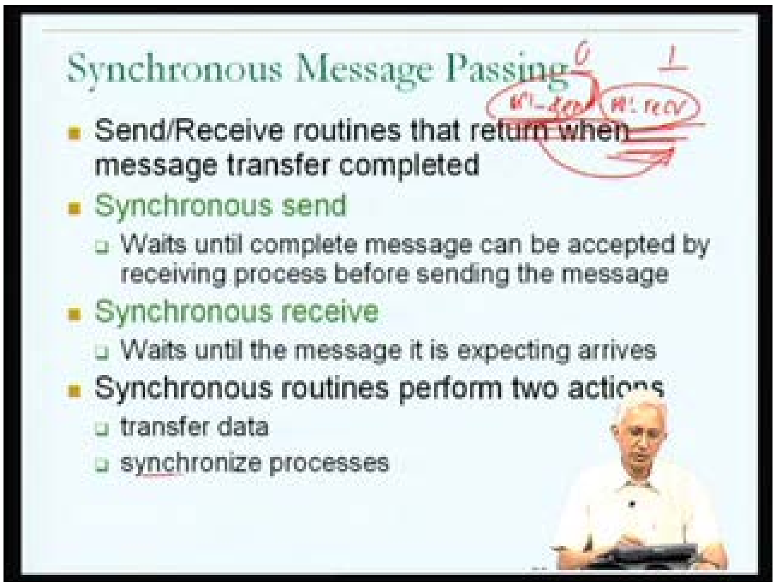
- Synchronous
- Asynchronous

So, the consequence is that they need not be sending matching sends and receives. Now, the default as we understand is that, one may use message tags and that the sender will specify in the MPI underscore send function call. The send ,in our example process P 0 must mention what the destination is. In other words, that you want to send it to the processor with rank 1 and it could also specify the tag. And at the point of receive, we note that once again, the receive function it indicates the recipient not only where the message came from, but also the tag with which it was sent and that this, as part of the function receive, an attempt is made to check whether the value of this these two fields match with each other.

Now, in the case of the tag, we saw that there was the possibility of indicating that we did not care which tag the message came with. The same is also true in the case of the source of the message. So, in the receive system call, it is possible to say rather than saying that this message receive is supposed to receive a piece of data that was sent by process with rank 0, one could say that this receive can receive data sent from any source using the MPI constant, MPI any source.

So, it is possible to overcome the default of matching sends and receives using these two wild card values for the sender and tag message tag options of the send and receives functions. Now, further there are a few flavors to sends and receives. Two flavors to send and receives in MPI which are what are called synchronous and asynchronous. we have seen these two words earlier, when we talked about asynchronous I/O- input and output and there is a similar flavor to what is happening in the case of MPI.

(Refer Slide Time: 30:13)



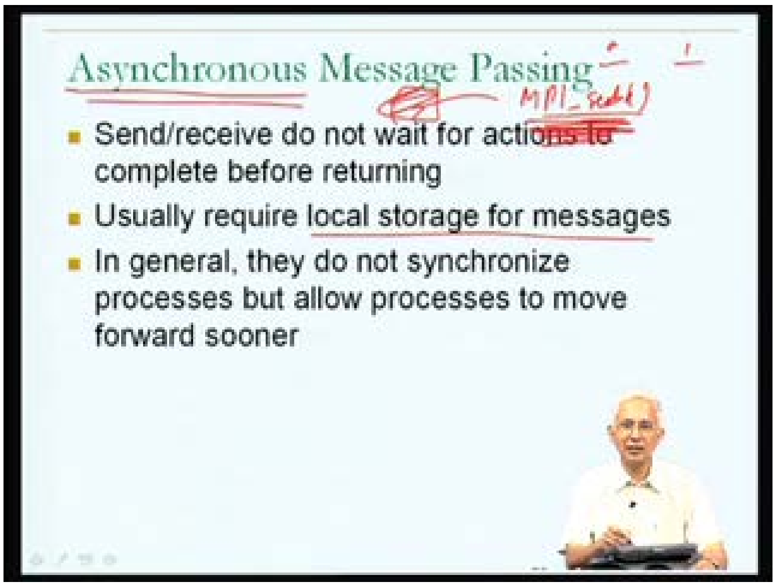
Synchronous Message Passing

- Send/Receive routines that return when message transfer completed
- Synchronous send
  - Waits until complete message can be accepted by receiving process before sending the message
- Synchronous receive
  - Waits until the message it is expecting arrives
- Synchronous routines perform two actions
  - transfer data
  - synchronize processes

Now, the idea of the synchronous message passing is, that after the send and receives, the send and receives routines return when there is a message transferred has been completed. So, the idea is that I could have a process with rank 0 and a process with rank 1. And the process whose rank 0 does a send and the process with rank and MPI underscore send the process with rank 1 does a receive. These are both function calls in those processes if the process. if the functions that I have used are what are known as synchronous MPI sends and the synchronous MPI receive. And the property is that the actual return from the function MPI send will happen only when the data transfer has successfully happened. Therefore, the return from receive similarly will happen, in other words, you will proceed MPI program proceed to the next statement in C program, only when the actual message transfer has happened. So, in some sense, one could say that, in the case of synchronous send, the send; the process which is doing the send will actually wait until the complete message has been excepted at the receiving end. It does not proceed to the next statement in the program just because the local activity relating to MPI send has been completed.

Similarly, with the MPI synchronous receive, it return, it continues execution of the receive only after the data has actually arrived. In some sense therefore, one could say that the synchronous message passing primitives available in MPI do two activities. Because data to be transferred from the sender to the receiver, but in addition, they synchronize the processes because, until the message transmission has been completed, both this the process P 0 and process P 1 will be within the MPI function send or receive and not able to proceed to the next point in their program. Therefore, the synchronous sending and receiving provide us not only with the capability of transferring data from one process to another, but also with the capability of synchronizing processes.

(Refer Slide Time: 32:10)



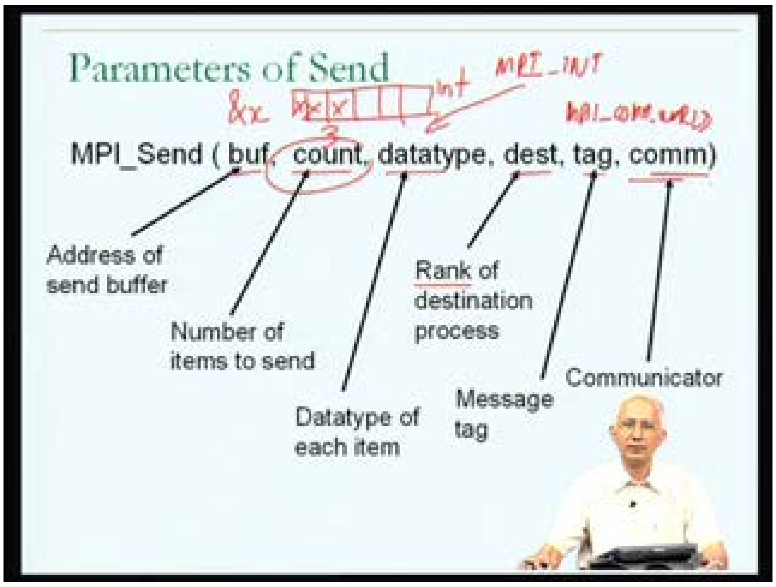
Asynchronous Message Passing

- Send/receive do not wait for actions ~~to~~ complete before returning
- Usually require local storage for messages
- In general, they do not synchronize processes but allow processes to move forward sooner

The asynchronous version, on the other hand, does not provide this facility and may be useful in certain kinds of settings. The bottom line is that, the MPI send and the MPI receive do not wait for the action to be completed at the other end before they return, and allow the individual local process to continue to the next statement in the program. And this had consequences as far as the implementation of MPI send or MPI receive is concerned because it will require that some local storage be used to remember the message and the way to think of this is, if there is a function called MPI send and as soon as the MPI send has transferred the data which is supposed to be send to a local variable of the MPI library, you can go on to the next statement. Very clearly, there must be this local storage, in order to allow the MPI send to quickly go to the next statement in the program which is why I reminded you that we have seen asynchronous I/O where it was possible immediately after a file I/O operation for the process which did the file I/O operation to go on to the next statement in its program, regardless of whether the I/O operation had completed or not. The situation is very similar here. The idea is to allow the programmer to write the MPI program to do other activities after the MPI sent has been finished some of its activity in the interest of reducing the amount of time that the process is blocked waiting for the transfer of data to actually complete.

So, it will try a little bit more careful programming by the programmer to make sure that the receipt of the data by the receiver it not assumed in the statements that immediately follow the send.

(Refer Slide Time: 33:44)



Parameters of Send

&x int MPI_INT MPI_COMM_WORLD 3

MPI_Send (buf, count, datatype, dest, tag, comm)

Address of send buffer

Number of items to send

Datatype of each item

Rank of destination process

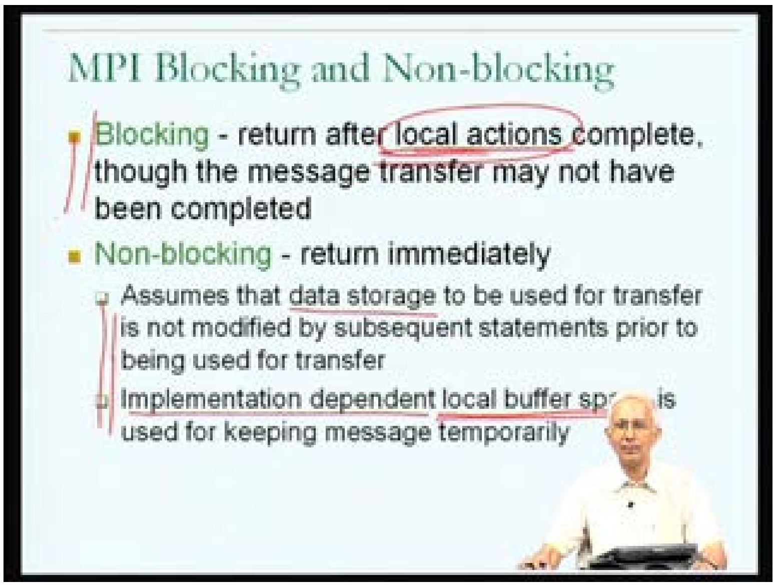
Message tag

Communicator

Now, we can now look at the parameters of the typical MPI send function. Now, the typical MPI send function has six parameters- the first parameter is the address of the buffer which contains the data that is to be sent. The examples that we had for example, there was the address of the variable x which contain the value that process 0 wanted to send to process 1. The second parameter is the number of items that are to be sent. Now, until now we would be assuming that a single integer value had to be sent. But what if this buffer is an array which contains a large number of integer values. It should still be possible for the entire array of values to be sent .That is why there is this notion how being able to specify, how many values from the buffer are supposed to be sent as part of the message. So, if for example, we want the first three values in the buffer to be sent, then we could indicate that the number of values from the buffer that are to be sent is equal to three. And how big each of the values that is to be sent? That is specified by the third parameter which is the data type. The data type specifies how many the type of each of the elements which is supposed to be included in the message that is sent. So, if for example, this is the buffer of integers, then the fact that it is integer is what will become radiant the data type and you will remember that we had MPI declared constants such as MPI int and this is this kind of a situation where one would use the MPI constant MPI int, for example, to indicate that each of the values, which each of the three values in this example which is supposed to be sent as part of the message is of type integer.

Now, as always, we know that the destination intended recipient of the message must be mentioned in the send call by its rank. Finally, we know that the tag may be used. By default, one could use the MPI any tag if one is not too concerned and finally, the communicator within which this particular communication is happening is also a parameter So, here for example, if we are just using MPI comm world, the default communicator, then the declared constant MPI comm world would be the value of this parameter.

(Refer Slide Time: 35:55)



MPI Blocking and Non-blocking

- Blocking - return after local actions complete, though the message transfer may not have been completed
- Non-blocking - return immediately
  - Assumes that data storage to be used for transfer is not modified by subsequent statements prior to being used for transfer
  - Implementation dependent local buffer sp[illegible] is used for keeping message temporarily

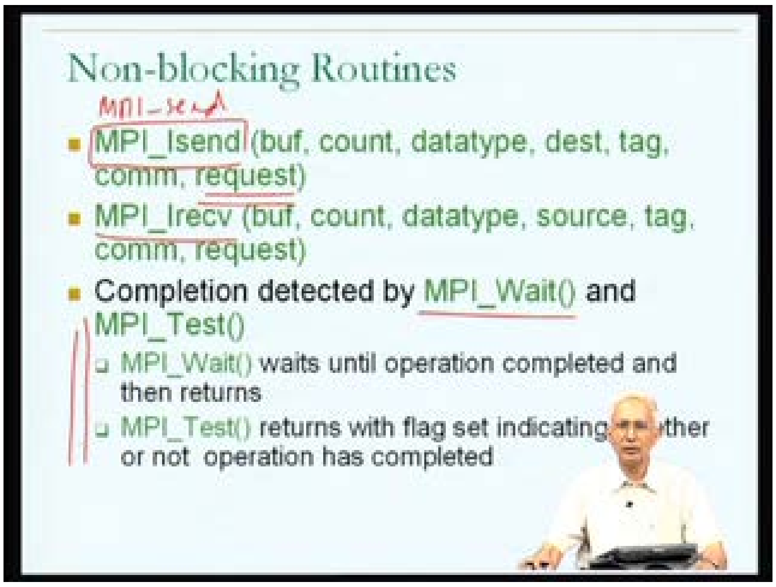
At the receiving end, there is a matching set of parameters which I therefore, would not go to in detail, rather let me spend a little bit of time talking about two other variants on the kinds of sending that can be done in MPI which are known as the blocking and the non-blocking versions of send. Now, the property of the blocking version of send is that the the send call returns immediately after the local variables have. The local actions associated with the send have been completed regardless of whether they message transfer has happened or not.

As opposed to the non-blocking where returns immediately even though, after the local actions relating to the send operation have been completed. And again this would recall using the blocking, in the case of the blocking, notice that it talks about returning of that local operations have been completed; where as in the non-blocking, it returns immediately even if the local operations have not been completed. What do I mean by local operations. Basically, one must understand that ultimately the transfer of data from process P 1, process with the rank 0 to process with rank 1 is going to happen using some underlying operating system functionality and they will be a need to therefore, pass the information from the MPI program to the operating system, possibly by copying it into an MPI buffer as I had mentioned. And there will be various other local operations between MPI, between the your program and the MPI library and also between the MPI library and the underlying networking software on the computer system.

So, these about we are referring to as a local actions. So, in the case of a blocking MPI send, so, only after the local operations are completed that the return will happen from the send, where as in the case of the non-blocking, even if the local operations have not been completing have been completed, the return from the non-blocking send could happen. And this once again, will assume that there is adequate storage, for example, for the buffering of the information contained in a message and this may be something that the programmer has to take care of the allocation of adequate storage for this purpose.

So, it is, as I pointed out in the second bullet, a little bit of a concern to the programmer because the amount of local buffer space which may be available for this local activity may be implementation dependent and this may cut into the portability of the program that we are talking about. And therefore, one must use a non-blocking calls only after carefully understanding the implications in terms of what other, but other kinds of facilities may have to be included in the MPI program as for example, ensuring that adequate local buffers space is occupied. Where as a non-blocking case would be much easier for us to setup, ensuring that the problem any problems with the amount of local buffers space are not present.

(Refer Slide Time: 38:36)



Non-blocking Routines

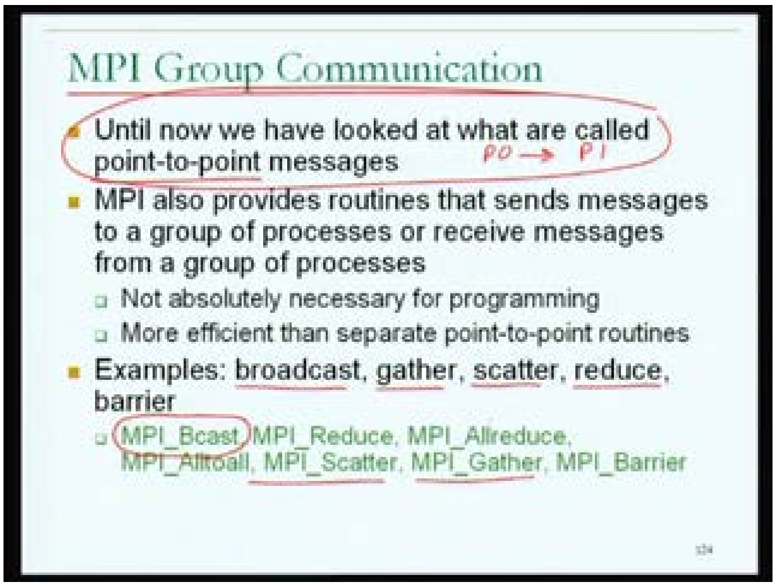
MPI_send

- MPI_Isend (buf, count, datatype, dest, tag, comm, request)
- MPI_Irecv (buf, count, datatype, source, tag, comm, request)
- Completion detected by MPI_Wait() and MPI_Test()
  - MPI_Wait() waits until operation completed and then returns
  - MPI_Test() returns with flag set indicating whether or not operation has completed

So, I had indicated that, each of the facility send and receive is actually a collection of functions because we now understand that there is going to be a separate function for sending, a blocking send, the separate function for a non-blocking send and therefore, one will come across a family of names for example, the non-blocking send is known as MPI underscore Isend and the parameters are similar. There is one additional parameter because there is a need for this synchronization to understand whether at what point in time the operation has actually finished this local activity. Therefore, I included this slide just let you know that in addition to MPI send, one may find other variance in which this this concepts of non-blocking or blocking may also be present as different functions. And that the other end they will be the non-blocking receive which may be which is refer to as MPI underscore Ireceive.

Now, in the case of this non-blocking send and receive there is a need for the sender to actually have some way of finding out when and as and when the send actually completes and hence there additional functions is known as MPI wait and MPI test along the lines would have to be done in the case of asynchronous I/O for the sending program the sending process to check when the send has completed and hence continue to the next part of its functionality.

(Refer Slide Time: 40:00)



MPI Group Communication

- Until now we have looked at what are called point-to-point messages $P0 \rightarrow P1$
- MPI also provides routines that sends messages to a group of processes or receive messages from a group of processes
  - Not absolutely necessary for programming
  - More efficient than separate point-to-point routines
- Examples: broadcast, gather, scatter, reduce, barrier
  - MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Alltoall, MPI_Scatter, MPI_Gather, MPI_Barrier

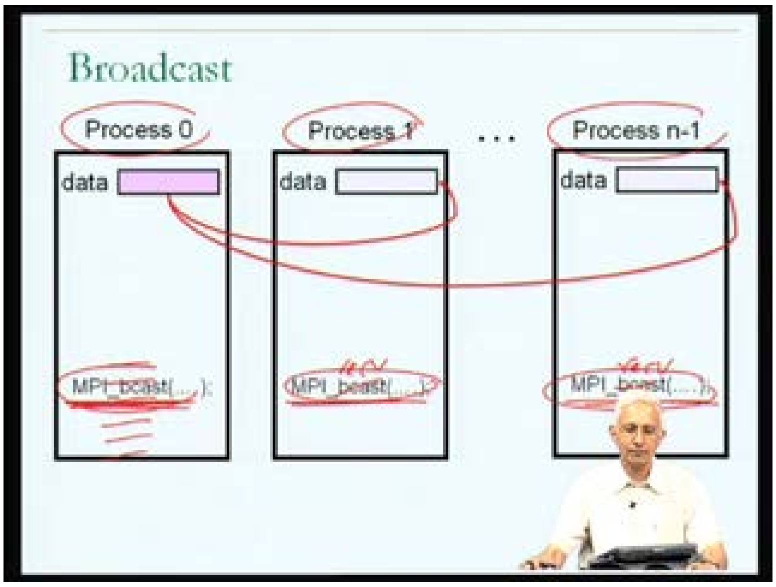
Now, the next important part of MPI communication. Now that we have a rough understanding of the nature of the send and receive basic functions is another kind of communication which is known as group communication. As the name suggest, the idea here is, rather than talking about communication between one process and one other process, we are taking about ways to do communication between groups of processes and as that is as described by disk bullet because until now, the kind of send and receive that we were talking about were functions that could be used for what is called point to point messages. A communication between process P 0 and process P 1 could be done with a send and the matching receive. So, that is what we refer to as point to point message. What we are now going to talk about are communication mechanisms for communication between groups of processes.

So, possibly one process sending a message to ten other processes or ten processes communicating data to a single process. So, these are what is known as groups. a group communication. Now, the group communication as you would understand is actually not essential for programming because if you have a mechanism through which you can communicate between one process and one another process, then you could use the same mechanism to communicate between one process and ten processes, you just have to include that mechanism ten times.

So, the reason that MPI includes group communication is to make thing easier for the programmer. Programmer does not have to include the ten sends when one process wants to send data to ten other processes, but rather can just include one of the group communication calls. So, it is not absolutely essential, but it makes programming a little bit more convenient and it may be possible for the MPI library to do the group communication in a more efficient fashion, then the programmer may think of while, if you were to set it up, he or she was the set it up using point to point communication.

Now, the examples of group communication which I am going to talk about, are the kinds of group communication which are generally known as broadcast, gather, scatter, and reduce. I want to talk about barrier. I will talk about barrier as an example, in the next lecture. So, what are broadcast, gather, scatter, and reduce? Now, they all functions which are available in MPI. For example, the broadcast function is available in MPI by the MPI function, MPI underscore Bcast. Similarly, there is MPI underscore Reduce, MPI underscore Scatter, MPI underscore Gather and so on for the various other kinds of group communication.

(Refer Slide Time: 42:24)

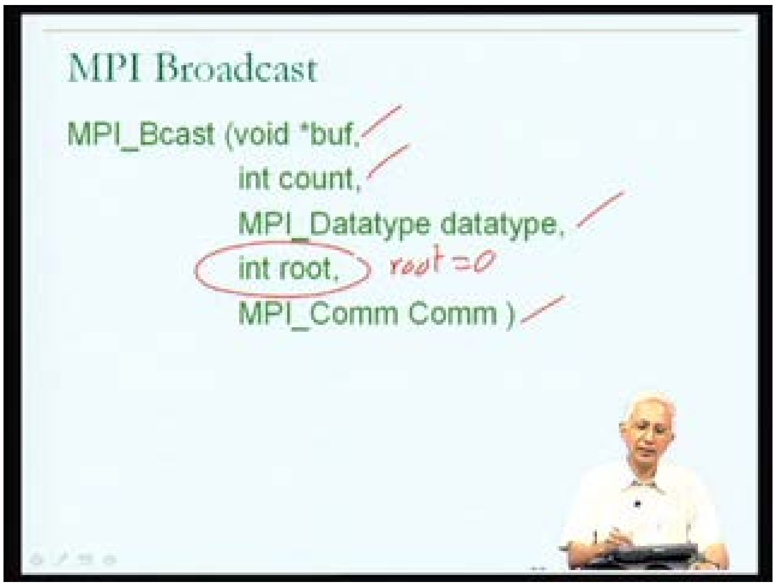


First of all, let us try to understand what the broadcast functionality is. I am going to describe this diagrammatically. So, the situation is, let us suppose that I have a parallel program which is running as n processes and the processes have ranks 0 through n minus 1. Now, each of these processes is going to run, let suppose on a different processor. It is going to run the same program and let suppose that I have a requirement that I want to particular piece of data to be communicated from Process 0 to all of the other n minus1 processes. This is the kind of situation where I can, which I can achieve using a call to MPI broadcast. So, at the appropriate point in the program, I would have to include a call to MPI broadcast. So, that it is executed by all of the processes and what this is going to allow me to do is, going to allow me to communicate a single piece of data which is part of the address space of process P 0 to all of the other processes with a single communication.

So, with the single call, I have been able to broadcast the value to all the other processes. Remember, the alternative what have been that the process 0 would have add to do a send to process P 1 followed by a send to process P 2 and so on. It would have had to do nine MPI sends and each of the processes in turn would have had to do an MPI receive, but the net effect would have been that the programmer would have had to write a more complicated program in the current version with broadcast. The programmer just writes its very simple program in which each of the communicating processes just does in MPI broadcast.

Now obviously, there is going to be some interesting parameters in MPI broadcast because they were all executing the same program which means that the fact that Process 0 was a process which was doing the broadcast and that the other processes what the processes which were receiving the broadcast must somehow be indicated within the function call.

(Refer Slide Time: 44:09)

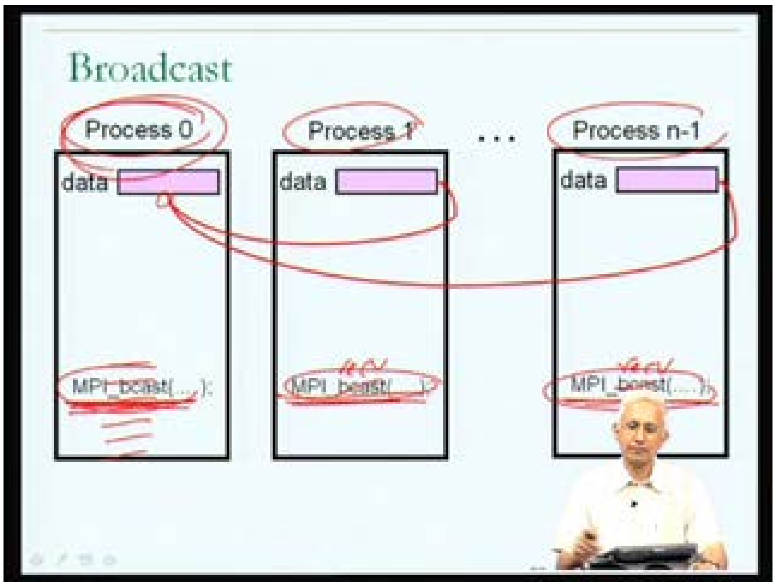


MPI Broadcast

MPI_Bcast (void *buf,
int count,
MPI_Datatype datatype,
int root,
MPI_Comm Comm)

If you look at the parameters to MPI broadcast, that is would we will find we will find out that in addition to the buffer count and data type which was present in MPI send or MPI receive and in in addition to the communicator information which was also present in MPI send or MPI receive, there is one additional parameter which is what is called root and the root is basically the specification of which is the process which contains the data that is going to be broadcast to the other processes.

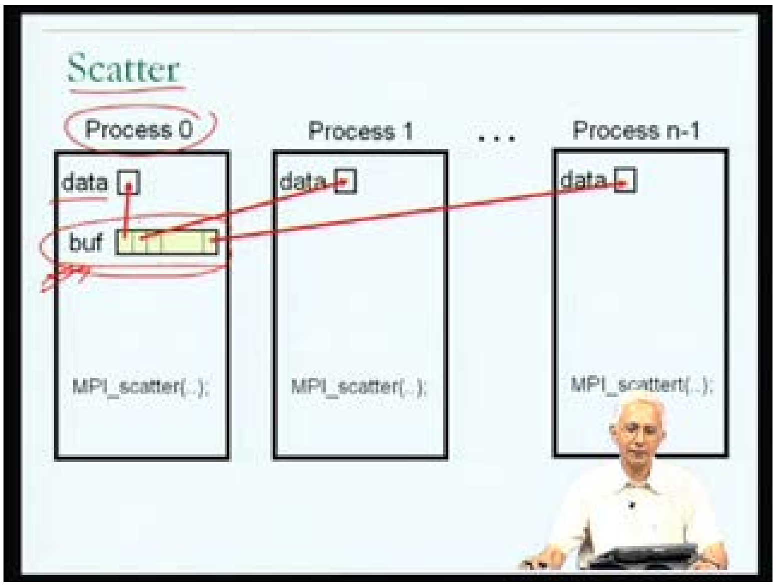
(Refer Slide Time: 44:35)



Broadcast

So, in our example the example which I am referring to as this one, this Process 0 which is the root of the broadcast in some sense, it is the root from which this tree of information is going out to the other processes and therefore, the value of root, we would find would be equal to 0 in all of these calls to MPI broadcast. So, MPI broadcast is a function which is provided in the MPI library for this kind of group communication and as you would imagine, this kind of group communication may be fairly wide spread, may be necessary for a lots of different kinds of parallel programs which is why it is provided by as a basic function of MPI of the MPI library.

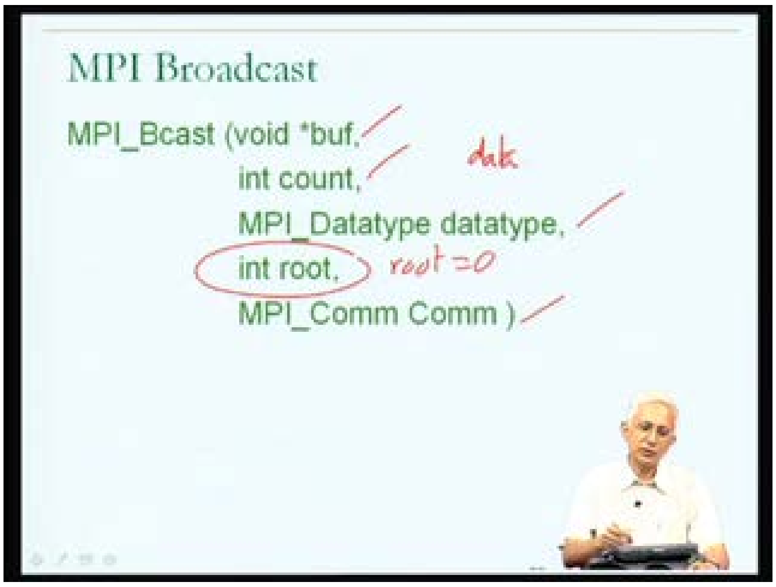
(Refer Slide Time: 45:13)



Now, the next function which I have talked about was a function call scatter. We are talking about the different kinds of group communication functions provided by MPI and once again we will look at a diagrammatically. So, we have a situation where there is a parallel program running as n processes and once again a single call to MPI scatter is going to satisfy our requirement. Now, what is the requirement? The requirement is that I have a collection of data which happens to be present on let’s say process P 0 in a buffer and what I want to do is, I want to distribute the data which is present in the buffer across all the n processes of the parallel program. To be more precise, I want the first element in the buffer to go to process P 0, I want the second element in the buffer to go to process P 1 and so on.

Now, this primitive operation is what is known as a scatter and the name make sense once you understand what it does. Essentially I had this collection of data and I want to scatter it across the different processes of my parallel program. In this particular example, one element is going to each of the processes. So, very clearly once again, if we looked at the parameters of MPI scatter, we should find out that it should mention both the buffer as well as the location into which the data is to be scattered further individual process. In addition, the identity of the root of the scatter must be indicated. In other words, the fact that it is Process 0 which actually contains the buffer which is to be scattered across all the n processes it is important to note that unlike the case of broadcast, it is the scattering of the data includes process P 0 or the root of the scatter within it.

(Refer Slide Time: 46:55)

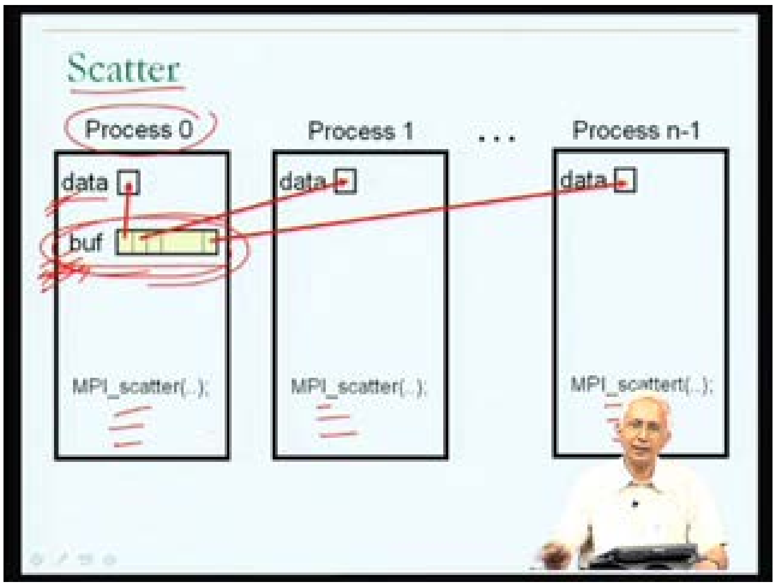


MPI Broadcast

```
MPI_Bcast (void *buf,
           int count,
           MPI_Datatype datatype,
           int root,
           MPI_Comm Comm )
```

So, we expect that the parameters are going to be very similar to what we had in the case of broadcast. In other words, this set of parameters. The only additional parameter that we will have will be the buffer which is going to be into which this scatter is going to be done. Notice that we had both a buffer from which the scatter data came and a buffer into which this scatter data goes. So, one additional parameter in the case of scatter if compared to the broadcast group communication. We will now move to gather. And as the name suggests, this is going to do the opposite of what scatter did. In other words, this is going to gather data from n processes into one into one process.

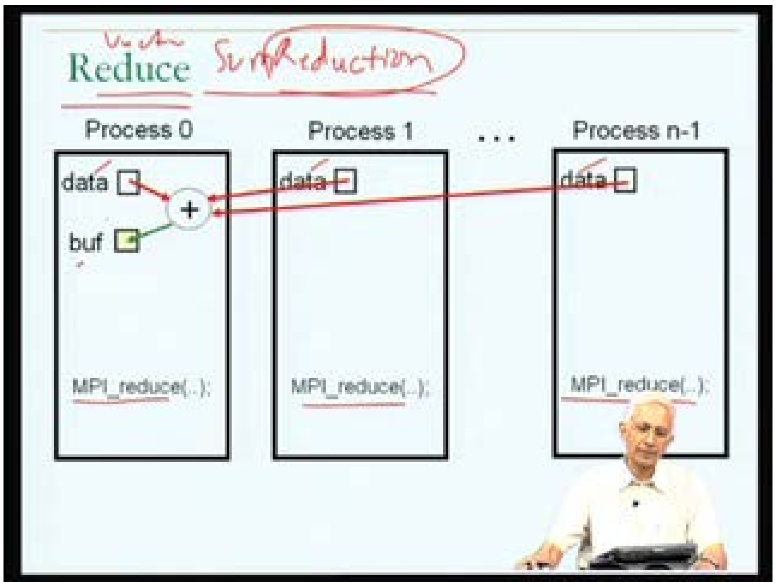
(Refer Slide Time: 47:29)



So, once again in a parallel program running as n processes is suspect that there is data which has been scattered or which is available, one data item on each of the n processors, but that we want to gather it into either a buffer within one of the processes, let suppose that it is Process 0 which contains that buffer. So, the effect of the MPI gather call is going to be that the data is going to come from each of the variables data in each of the individual processes and gets put into the appropriate slot in the gathering buffer. In other words, the data from Process 0 goes into the zeroth element of the buffer, the data from process P 1 goes into the first element of the buffer and so on.

So, this can be, as you can see by scattering data and subsequently by scattering data, it might be possible to for example, distribute work to be done by the individual processes of the parallel program. For example, if I wanted to do some computation and each of the elements of this array, then I could write a parallel program which scatters the data across the processes of the parallel program. A processes could then independently operate on their portion of the data and after that the computation had been completed, they could conceivably send their newly computed value back to Process 0 using a gather. So, one can see that in certain applications these kinds of group communication could be very useful.

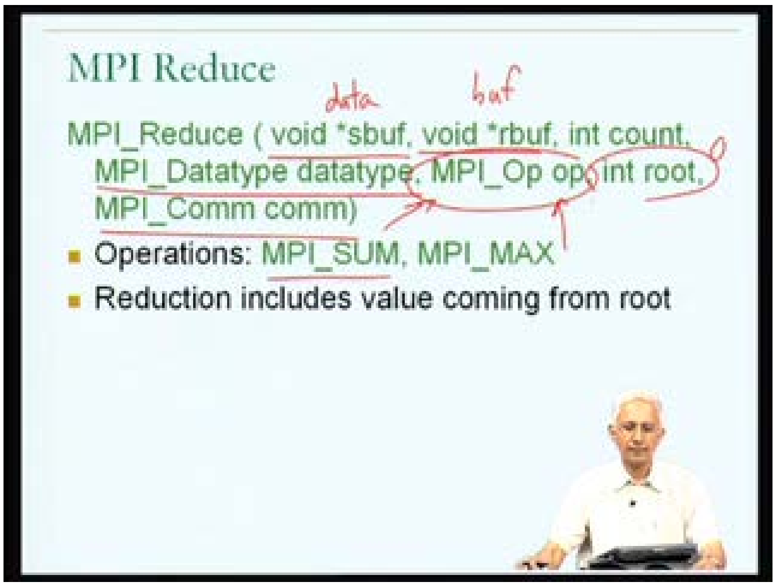
(Refer Slide Time: 48:58)



There was one other group communication primitive that I had included in my list and that was an operation called reduce. And as the name again suggest here, what is going to happen is some large collection of data is going to get reduced into possibly a single value and in our scenario, where there are n processes, we would understand that the objective of reduce must be to take data which is present one value on each of the processes and somehow combine it into a single value which will be available on one of the processes. So, the question is, what kind of reduction operations might be possible. We have come across the notion of reduction operation earlier. We talk I talked about the notion of a vector sum reduction. You will recall, when we are talking about the different, when we were looking at cache optimizations of loops of programs, one of the operations that I talked about was a vector sum reduction where essentially, we were computing the sum of the elements in a vector and that is an example of a reduction operation of the kind that we were talking about over here.

So, the reduction that we were talking about might involve gathering the values from all of the processes in the variable which they individually call data into one of the processes and at that process reducing it by, may be by adding all the values together to produce a single value which might be the result of the MPI reduce call. So, in MPI reduce obviously, the parameters would be like the parameters in scatter and gather, they must be mentioned of the individual data items, the variable which is going to be reduced and also of the buffer into which the accumulation is going to be done or the reduction is going to be stored.

(Refer Slide Time: 50:38)



MPI Reduce

MPI_Reduce (void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

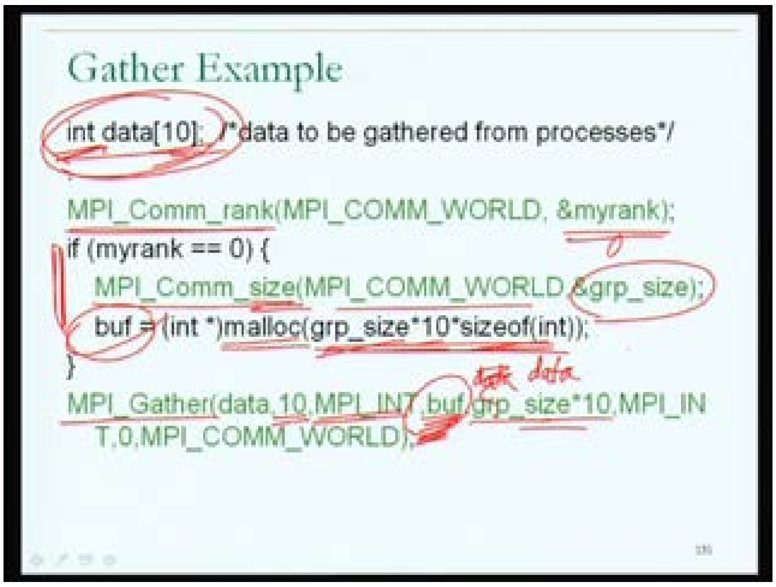
- Operations: MPI_SUM, MPI_MAX
- Reduction includes value coming from root

What are the different kinds of reduction operations that are supported by MPI? Here, I have actually shown you the parameters of MPI reduce, as we always suspect there is going to be the need to specify the communicator in which this communication is going to happen. In this particular situation, there is going to be the need to specify both the buffer into which from which the data is going to come.

So, this is what had been referred to as data in my example, and the buffer into which the reduction is going to happen which was referred to as buf in the example. Some indication of how much data is going to have to be accumulated and of what type from each of the buffers and which is the root of the reduce operation. In our example, the root was zero. And finally, what operation is going to be used for the reduction. So, one has options about a few different operations which could be used in the in the reduction to the two which I will mention. First of all, the possibility of actually doing the sum of the different values of data using MPI sum as the reduction operation. Another possibility which is supported is to actually compute the maximum of all those values and this again may be useful in certain kinds of parallel activities.

So, in short, there is a family of these group communication primitives through which it is possible to improve the performance in MPI program by utilizing the improved version of communication as an alternative to individual processes communicating values by separate sends and receives.

(Refer Slide Time: 52:03)



Gather Example

```
int data[10]; /*data to be gathered from processes*/

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Comm_size(MPI_COMM_WORLD,&grp_size);
   buf = (int *)malloc(grp_size*10*sizeof(int));
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_IN
   T,0,MPI_COMM_WORLD);
```

Now, as the closing example, I mean I will go through one more example after this, but just to sort of put these things together. Here we have let say, a situation, a brief MPI program which you will notice is using three of the MPI functions, is using MPI rank, is using MPI size, which I have not talked about. Let me say a word about that. And it is using MPI gather. Gather is the group communication function which will gather values from each of the n processes into a buffer an array in the root process.

Now, the objective of this particular program as the common suggest is that, there is a collection of ten pieces of data and that we want to gather it into one of the elements of the, one of the processes of the MPI program. So, this is declared in all of the processes which are running. So, in this program each of the processes determines its rank, stores at in its local variable my rank, and depending on whether the process is the root of the group communication or not, if the process is the root of the group communication, then it finds out the size of the communicator.

Now, MPI comm size is one of the functions that we had seen in our list or functions, but I had not talked about. What MPI comm size returns is the size of the communicator in question. And by size, we mean the number of processes involved in the communicator or the number of processes in that communicator and the therefore, the drawings of the processes within the communicator would go from 0 to group size minus 1. And you will notice that what the processs which is the root of the communication is doing, is it is allocating a buffer of adequate size to all the data which is going to come from each of the processes which is going to be involved in the gather.

So, it declares a buffer which is going to be the buffer which is used in the call to gather. Now, the data which is going to be gathered is going to come from the individual processes. So, each of them has data inside its data object data which is going to be another of the parameters. So, data is going to be another of the parameters in the call to gather, not listed here. Each of them is going to provide ten pieces of data from its, I am sorry, ten pieces of data, I am sorry that is the way things are setup. The way things are setup is to do this using gather, but you will remember that, in gather we actually had a piece of data coming from each into a global buffer. So, in this example, the global buffer is of adequate size because it has been declared to be of that size and the individual values are coming from the individual communicators from their buffers called data which must be specified as the other parameter in the call. Now, from each of the. So, buffer was missing in this particular list. From each of the buffers as you will notice that ten values had to be communicated and that each of those values was in integer which is why we talked about communicating ten values each of type integer, from each of the processes to the root process into the buffer. That is why buffer was of size 10 multiplied by group size multiplied by size of int.

So, if there were ten communicating processes, then buffer would have been of size 100 multiplied by the size of int, possibly four hundred bytes. So, this was a slightly more complicated example, but what we will do in the next lecture is to actually look at a specific example of a computation as a MPI program incomplete, incompleteness and then we will proceed to look at some examples of writing parallel programs not specifically in MPI, but from the perspective of understanding how the activity of writing parallel program from specification of the problem until a clear identification of how the different processes might be viewed could be could be conducted. And we stop here and proceed with a more complete example of in MPI program in the next lecture. Thank you.