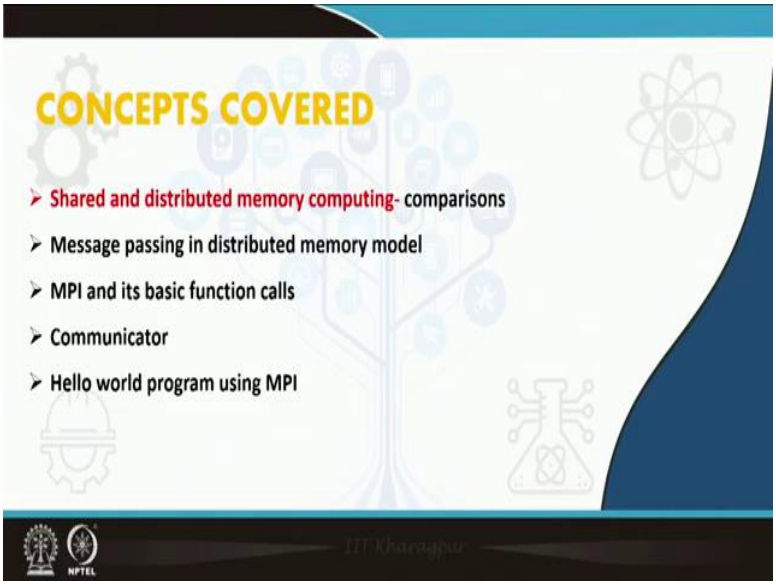
High Performance Computing for Scientists and Engineers
Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module - 03
Message Passing Interface (MPI)
Lecture - 20
Introduction to MPI and Distributed Memory Parallel Programming (continued)

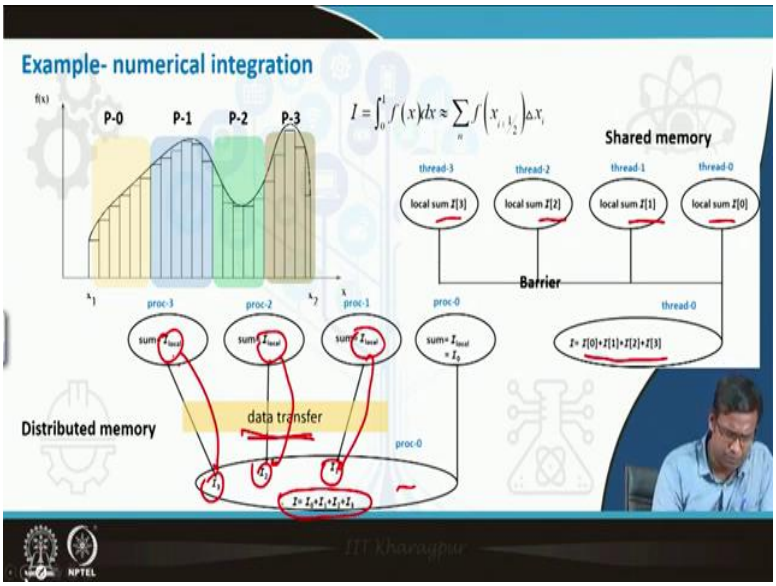
Hello everybody, we are discussing the topics in the course High Performance Computing for Scientists and Engineers. We are in the 3 rd module and 20 th lecture of this course, which is Introduction to MPI and Distributed Memory Parallel Programming. We are basically continuing from the last lecture.

(Refer Slide Time: 00:42)



In the last lecture we discussed shared and distributed memory computing ,tried to get some comparisons in between them, and identified the important features of distributed memory computing, because when doing MPI we are basically working on distributed memory parallel programming. So, we will start with comparing distributed and shared memory programs at least algorithm wise.

(Refer Slide Time: 01:07)



One of the examples which we have worked on while learning OpenMP and shared memory programming as numerical integration is that we have to integrate a function between certain values 0 to 1 or x 1 to x 2 here. We distribute into many small finite trapezoids and try to find out the area of the trapezoid and sum it up.

So, if we try to get a very accurate estimation of the integration the number n(the intervals over which we are summing it up) must be a large number. So, as it is a large number if we try to write a simple C program there will be many number of iterations. So, we try to ask multiple processors to take care of that.

Each processor will execute part of the integration and finally, they will sum up the value. So, say, there are four processors each trying to do the integration for certain for the portion given to it, for a certain number of steps given to it and they are doing it concurrently, and finally, everything will be summed up. So, in shared memory program we can see that there are four threads each processor is running 1 thread and each one is finding the local sum I [0], I[1], I[2], I[3] and after each thread has finished their work there is a barrier and only thread 0 is active after the barrier. All other threads are destroyed.

However, the very value I[1], I[2], I[3], I[0] being shared memory variables are available to the main RAM. So, thread 0 gets all these values and sums them up and finds the I .We have seen that while writing these sums there can be false sharing issues, you have to use padding or critical construct, etcetera. This is the basic model that each processor computes the local

sum stored in a shared memory array and after the parallel part the final summation must be done sequentially.

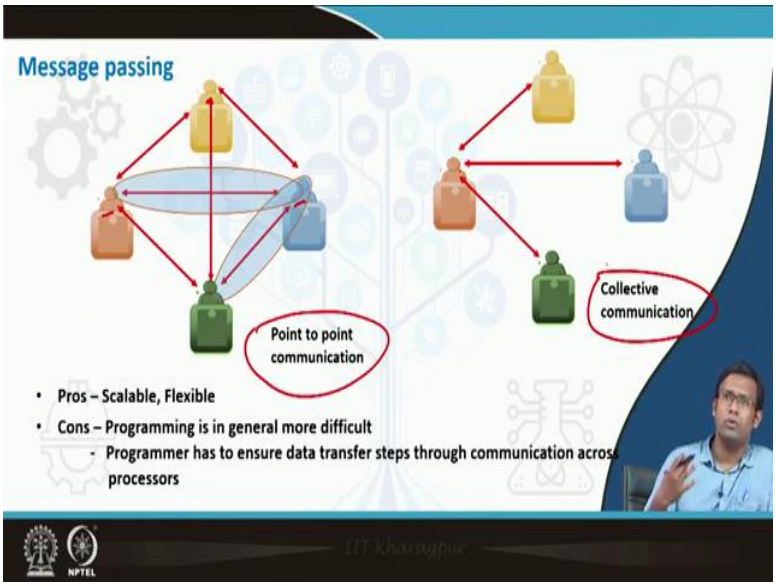
So, there is a barrier where all the threads are destroyed and only the thread 0 is active who reads all these variables from the shared memory and sums them. What happens in distributed memory? In distributed memory the memory units are distinct. Each processor finds its own local sum, processor 0 finds its own local sum I[0], for certain purpose, we can see that. Processor 1 finds the local sum which is again I local, because there is no meaning of writing I[1], I[2], I[3] each processor the processor has its local sum which is not being seen by the other processors.

So, how to find out the summation that after this part when the final sum has to be found out other processors have no work, because this is one sequential step. So, processor 0 retains the local sum as I[0] and other processors send their local value to processor 0 as I [1], I [2], I [3]. The data transfer part where I local of processor 0 is copied as I[1] here, I local, the local integration of processor 2 is copied as I [2] , the local integration of processor 3 is copied as I [3] here. This is the data transfer step.

So, these processors have their own remote local data. They send this information that this is the value calculated by them ,through network cable to the 0 th processor and now, 0 th processor sums up and gets the final integration.

So, that is the main difference, here, they do not need to send anything, because everything is being written to the same shared memory address space, but the obtained summations are written into the local address spaces which is not visible by the address space of processor 0. They are physically different units most likely. So, they have to send this value, they have to send this message, they have to pass this message to share the values. Therefore, there is a requirement of data transfer across the network cables here and that is the essential component and the unique demarcating component of distributed memory programming compared to shared memory programming.

(Refer Slide Time: 06:28)



So, in message passing all the processors must be connected via network cable and they should be able to send and receive data in between each other. This has to be both ways; one sense data other receives it, the other can send data and it can receive it.

The advantage is that this is scalable and flexible ,you can send data across these two processors when others are unaware of this. So, you do not need to interrupt the working of the other processors, it has a lot of flexibility. You can identify which processor gets data, which data it gets from which of the processor etcetera without disturbing .

It is scalable, you can add as many processors as possible depending on your network. Basically, you have to spend more for the network cable and send data across the processors. Of course, data transfer has its own overhead, and there is no hardware restriction for doing that.

The cons are; programming is in general more difficult compared to shared memory, because you have to identify the instances for data transfer and send the right data which has to be mapped to the processor. Programmer has to ensure data transfer steps through communication across the processors. There is another con that data transfer has its own overhead. So, they will add up to certain latency which is more than shared memory programming.

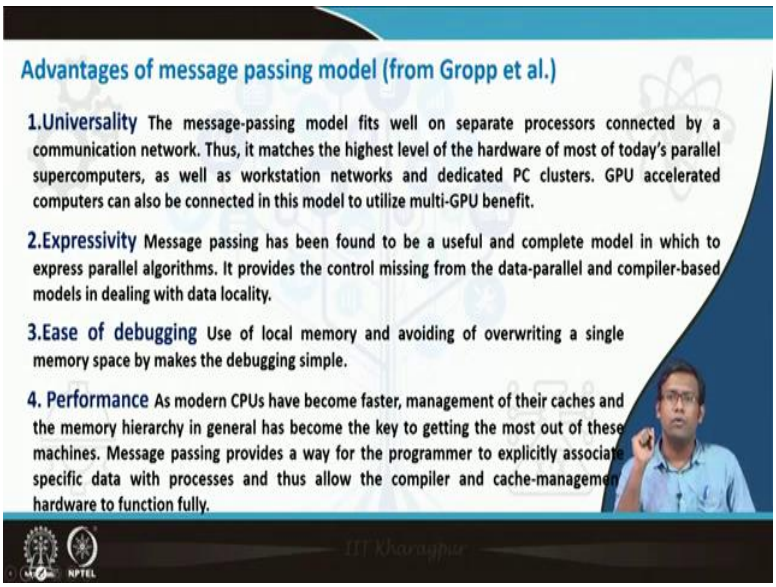
This is called a point to point communication in message passing, where each of the points or each of the nodes or processors can share data to any of the nodes in between one point to another point.

There is another mode of data transfer here which is a collective communication where one processor can send data to all other. It is like one person standing with a mic and speaking. So, everybody gets the information. This is possible in the message passing paradigm that 1 processor can send data to all other processors or 1 processor can get data from all other processors.

Like in the previous example the 0th processor needs to get data from all other processors 1, 2 and 3, which is also possible ;instead of ensuring that it has to one by one interact with all the processors it can collectively get data from all the processors.

So, these are the two distinct communication modes in message passing; point to point communication (one talks to another) and the collective communication(when one gets data or gives data to a group of processors).

(Refer Slide Time: 09:24)



The advantage is that universality; message passing model fits well on separate processors connected to a by a communication network. Therefore, it matches the highest level of hardware which is the supercomputer and also for dedicated PC clusters or workstation networks. It basically requires different processors which are connected by network switch and

this communication network can be 1000 gigabyte per second or even faster than that. GPU accelerated computers can be connected and specially, we will discuss about GPU programming (but we will probably not spend time on multi GPU programming).If you want to ask multiple GPU's to solve a program,GPU's cannot do anything independently, the different CPUs will solve the program ,and each CPU is connected with the GPU .You can use something like a message passing model across the CPU so that they break the problem. If you are using say four GPU's you can use a message passing model where the job is broken down to four CPU's and each CPU offloads part of the job assigned to it to the connected GPU's. So, multi GPU benefits can also be taken using message passing systems.

Expressivity, it is a useful and complete model to express parallel algorithms. It provides very good control over the programming. The programmer can control many aspects of the program which are missing in data parallel or compiler-based models like OpenMP. It gives you good data locality and more expressivity. Programmers can express themselves more, programmers can be more innovative with the parallel algorithms and do a lot of things more and have more control over the program.

Ease of debugging; as it is using local memory and not all the processors not trying to overwrite over a single memory space the debugging becomes simple. When you have written the program, you can look into one particular processor's behavior and probably find out what are the bugs in the program.

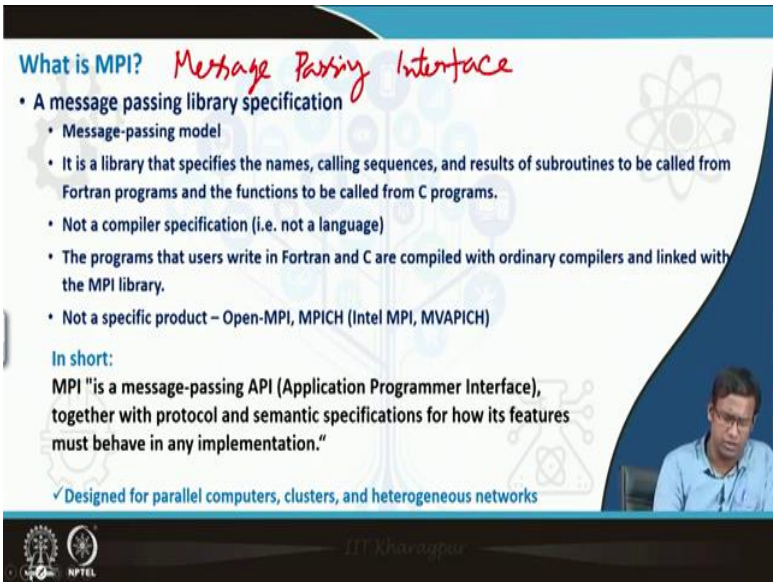
Performance is usually better here, because in modern CPU's the management of cache and memory hierarchy are very important issues to get the right performance.

When you use OpenMP or shared memory-based parallelization cache coherence kills this. You have very fast cache, but in order to establish cache coherence the advantage of using cutting-edge CPU's is done, because the fast cache speed is killed down by the cache coherence in a shared memory system.

However, because you do not need to worry about cache coherence and false sharing in a distributed memory system, you can get the benefit of using high performance computers. You are explicitly associating the specific data with the process. You do not need to worry about how the other processors work and therefore, the compiler and cache management hardware can function fully.

So, there are certain advantages over shared memory programs in distributed programming and these advantages are very obvious. When you are trying to solve a very large problem with say millions of floating-point operations to be done in the problem over a large data set, these advantages become very obvious.

(Refer Slide Time: 14:24)



So, shared memory can be good for working on relatively smaller problems, it is very simple to operate on and it has less overheads etcetera, but distributed memory models are the right thing for working on a large problem.

Now we come to the message passing interface or MPI and as we said that all the different components of the distributed memory programming model will be taken care of in MPI.

MPI is message passing library specification, it is called message passing interface and this is a message passing model. The full name of MPI is message passing interface. It is a message passing model, it is a library that specifies the names ,calling sequences and results of subroutines to be called from the Fortran programs or functions of C programs.

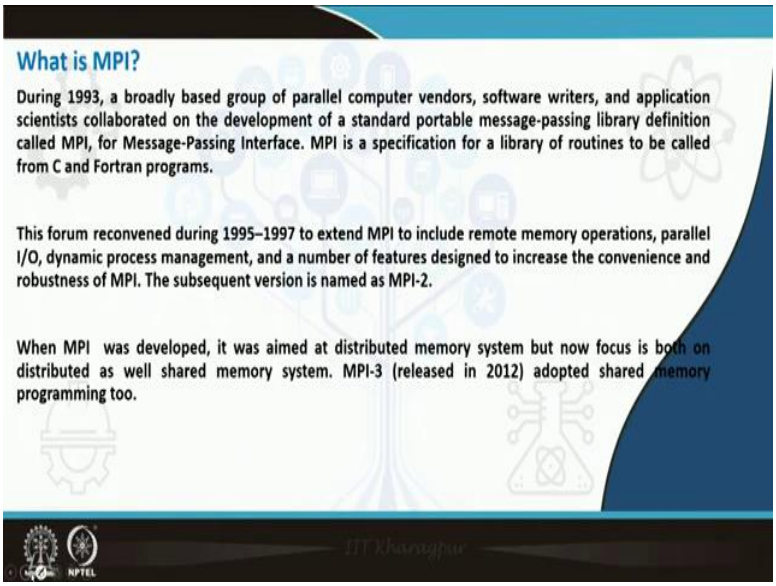
In a Fortran it's like a subroutine and in C it is a function , but they are basically library calls. It is not a compiler specification so; it is not a language. It is a library basically. The programs that users write in Fortran and C can be compiled with ordinary compilers and compiled with MPI libraries to get the benefit of MPI.

It is not a specific product; there are multiple versions of MPI and multiple releases of MPI from different forums. In short MPI or message passing interface is a message passing application programming interface together with protocol and semantic specification for how its features must behave in any implementation.

So, these are library calls which are written over standard C ,C ++ or Fortran programs (many other languages also support MPI) by which you can ensure that the essential elements of distributed memory parallel programming can be taken care of. This is specially designed for message passing across different processors.

Again, it is an API,so it is something similar to OpenMP. OpenMP works for a shared memory system, MPI will work for a distributed memory system. These are designed for parallel computers, clusters, and heterogeneous networks.

(Refer Slide Time: 17:14)



In 1993 groups based on parallel computing vendors, software writers, and application scientists collaborated and developed a standard portable message passing library definition called MPI for message passing interface. MPI is a specification for a library of routines to be called from C and Fortran programs for message passing across different processors.

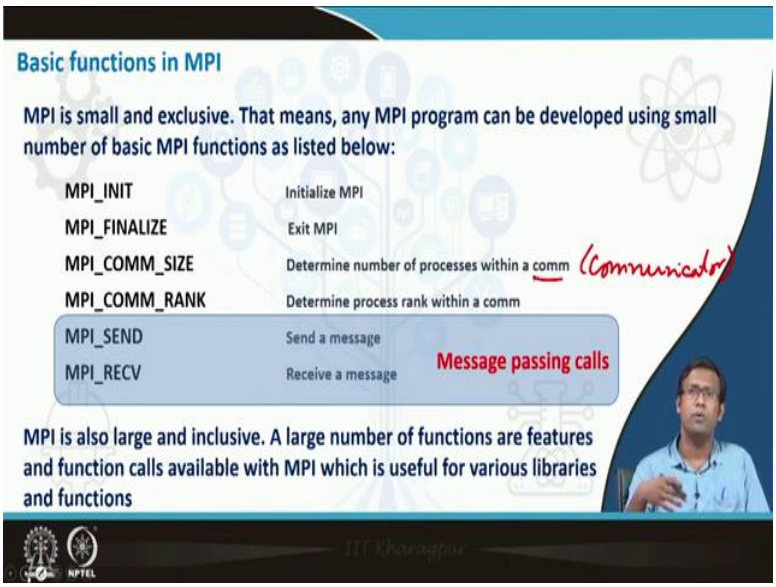
These forums again reconvened in 1995 to 97 and extended MPI to remote memory operations, parallel input output, dynamic process management and a number of other features which we many times use for writing robust programs in MPI and this version is named MPI 2.

When MPI was developed it was dedicatedly developed for distributed memory systems, but later people found out that adding some shared memory applications will be very helpful. So, MPI 3 which was released in 2012, adopted shared memory programming too.

So, in a sense MPI is a or message passing interface is a library specification by which you can make different computers talk to each other, communicate across each other through message passing in a distributed memory system; however, there are certain developments later in MPI which enables it to work it on this shared memory system also.

Again, in this particular course we will look into distributed memory applications of MPI only, which was the main driving factor in development of MPI.

(Refer Slide Time: 19:03)



Basic functions of MPI. MPI is small and exclusive; that means, like if you know only a few MPI functions you can take any program and make it parallel and these MPI functions are these six functions.

MPI_init which initializes MPI environment,

MPI_finalize which exits MPI, it says that MPI environment is over and processors can do something else.

MPI_comm_size which determines the number of processors within a communicator.

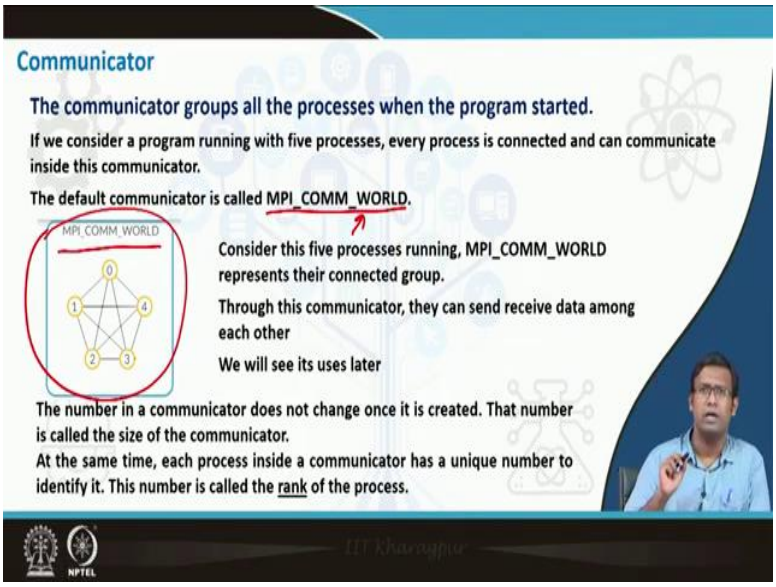
MPI_comm_rank which determines the rank of the process within the communicator, MPI_send which sends the message.

MPI_receive which receives the message.

With these six functions, any program can be parallelized in a distributed memory system. So, we call it a small and exclusive library system that, with only a small set of functions you can work with MPI.However, MPI is large too. So, these are the message passing calls MPI_send is for sending the message, MPI_receive is received for receiving the message. These are the main message passing calls.

MPI is large too; that means, a large number of functions and features are available in MPI useful for more optimized programming.

(Refer Slide Time: 20:48)



Now, the communicator is a very important part of MPI ,it is the group of all the processors which will work in a distributed memory system. In a shared memory system, it is easy, all the processors connected with the RAM, share the same memory space and therefore, they are part of the shared memory system. Also, you can define the number of threads, so, few of the threads will be identified and they will work. In a distributed memory system, physically this is difficult, there are multiple processors with their own rams and they are connected through a network switch. Now, when you run a program you have to identify few of the processors and they are part of this particular MPI program.

In shared memory it was easy in that sense that there is always a processor 0, when other threads are launched it asks other threads to work for it.

In MPI though there will be some processor named processor 0, but in a sense, there is no master. First the scheduler is scheduling the program ,then when the scheduler allocates the resource MPI knows that this resource is allocated to it.

So, it groups these processors and tells them that you are working for the same problem, you are working to execute the same executable and you need to communicate among them using the MPI calls and this group is called a communicator.

If we consider a program running in five processes then every process is connected so that they can do point to point communication and collective communication and they can communicate inside them within this communicator. The default communicator is called MPI_comm_world.

When you identify that these processes are connected and they can communicate across themselves the program identifies this group of processes as MPI_comm_world. So, if you write MPI_comm_world it will identify that these processes are involved in running the program. So, whatever I am writing it should be within these processes. Physically, it is the computers which are connected via the network cable or the CPU's which are connected via the network cable and communicating across themselves.

If there is no network cable you are doing multi threading or you are using a SMP machine to work in the distributed memory systems. These processors which are executing that particular program (it is always a single program multiple data model which is executing the program),they are grouped and this group is called the communicator.

So, when they have to communicate ,they have to communicate within the group; that means, via the communicator. MPI_comm_world is the default name of the communicator. So, all of the MPI calls which require access or which require coordination among multiple processors must mention that they are working on the through the communicator MPI_comm_world.

So, if there are five processors working in between them ,all of them can share data in between each other or they can do a point to point share and they are connected in that way.

There is a network cable, however, virtually for the program they are connected directly in between each other and this connection group, this group of all the processors and their

connection is called the communicator MPI_comm_world. This is the elementary block of the parallel program ,any data message passing any data transfer works within this group or within MPI_comm_world.

So, when a parallel program is launched the processors are identified, connections are established in between them so that they can do point to point data transfer as well as collective data transfer and this group that these processors are executing this program through MPI and they are doing data transfer through MPI across themselves with all the flexibilities.

Any processor can send data to any other processor. The entire group is identified by MPI_comm_world. This is the default communicator, you can get derived communicator also communicator within a communicator which we will not discussing here, but any MPI call which requires synchronization or communication across the processor must mention MPI_comm_world so that it identifies that in the whole network these processors and their connectivity is being taken care of through this call.

Consider these five processes running MPI_comm_world will represent the connected group. If processor 0, 1, 2, 3, 4, these 5 processes are running the MPI_comm_world will identify their group and the connected group will be identified as MPI_comm_world.

So, it is like when I made the slide in PowerPoint what I did? I drew these circles; they are the processors; I drew the lines connecting in between them and then I selected everything, made a right click and grouped them. Now, I can move this group as I wish and this particular group is communicator or MPI_comm_world. So, what do we do in PowerPoint? We get many many elements and group them.

Similarly, there are many processors, there are communications in between the processors, we group them, this group is called the communicator and the default name of the communicator is MPI_comm_world. Through this communicator they can send and receive data among each other.

The number in the communicator does not change once it is created. So, once we write MPI_init or initialize MPI,it identifies the resource allocated to it and builds a communicator .Once this communicator is built in, the number of processors will not change .In OpenMP you can remember you can change the number of threads, but in MPI you cannot change the number of

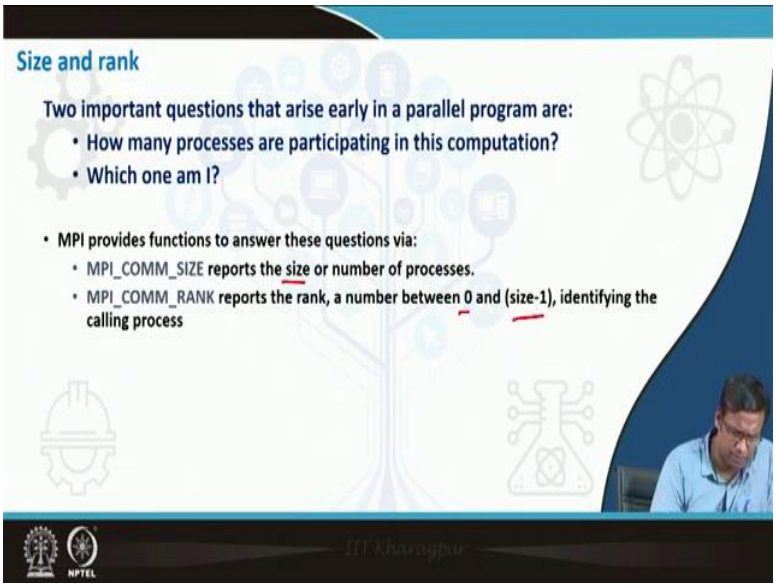
processors once the communicator is built, the number of processors are identified and the entire program will be executed in this processors.

If you do not try to run part of the program in one of the processors you have to do something as a programmer, you have to write some if else statement etcetera.

However, in general this entire program will be executed in all the processors assigned to that communicator and once done it cannot be changed. The number of processors inside the communicator is called the size of the communicator and at the same time each processor inside the communicator has it's as a unique number to identify it.

We can see 0 to 4 , each processor has its unique number and this number is called the rank of the process. Each process has its unique number, I sometimes mention processor in terms of process, but the correct word is process here. Each process has its unique number and this number is identified as the rank.

(Refer Slide Time: 28:43)

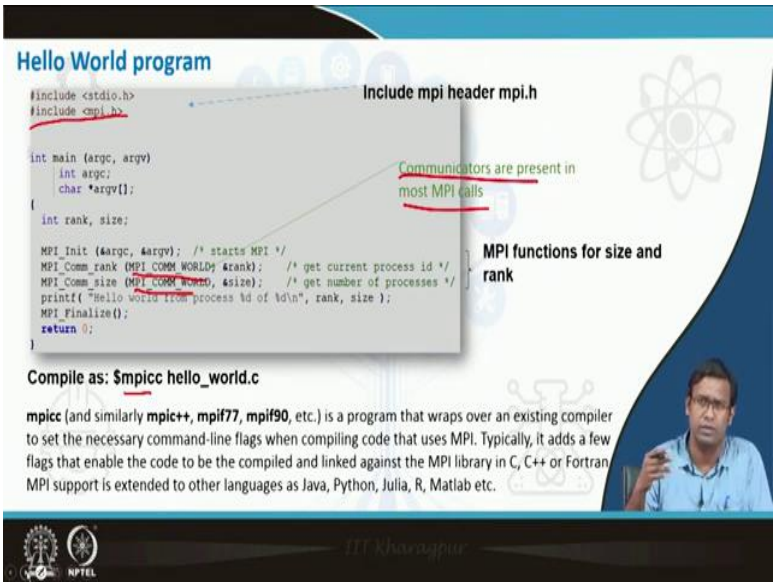


So, two important questions arise in a parallel program, how many processes are participating in the computation; that means, what is the size of the communicator and who am I, what is the identity of the particular process in that.MPI provides functions to answer the questions, if you write the function MPI_comm_size, its output is the size of the communicator or the number of processes in the communicator.

If you write MPI_comm_rank, each processor or each process will return a unique id, which is the id of that process. The rank of the processes starts from 0 and ends at size - 1, because size is the total number of processes here.

So, size and rank are two important features here and many times we have seen in shared memory programming also, you need to know about the rank of the process and the size of the computation also. So, these will be required many times.

(Refer Slide Time: 29:54)



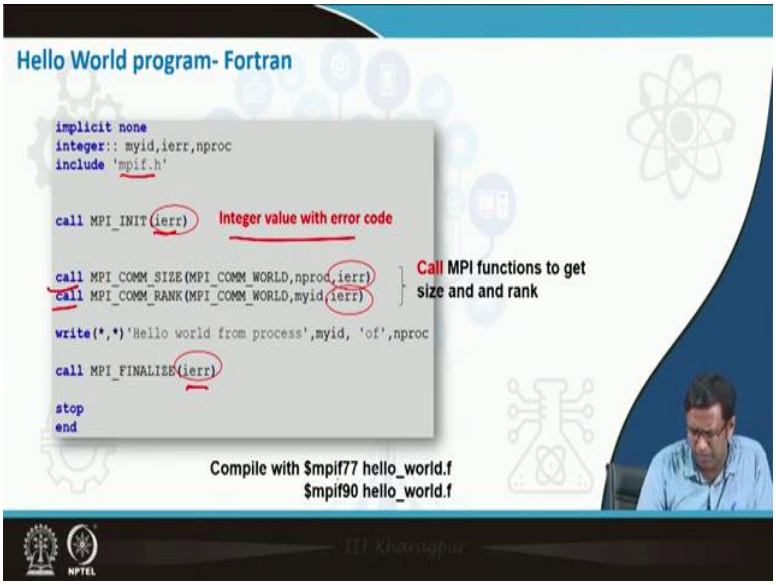
So, now we come to our first MPI program which is again the same as the first OpenMP program ,we are discussing Hello world program.

It starts with (Refer Time: 30:07) # include stdio .h then # include mpi .h; that means, this is the MPI library header function which has to be included. We write MPI_init with , integer argument and character argument as in input of the functions, it initializes the MPI. MPI_comm_rank gets the current process id, MPI_comm_size gets the number of processors and then it prints Hello world from each process and writes the rank and size. MPI_finalize finalizes it and the return is 0. So, this is the MPI header mpi .h. We can see that when we need to synchronize or coordinate or communicate across many processes, we accept initialize and finalizing, we have to use MPI_COMM_WORLD which is the communicator that tells you are working within this group, what is the size of this group and what is your id in that particular group. So, both COMM rank and COMM size as well as send ,receive (we will see later) request the communicator MPI_COMM_WORLD which is present in most MPI calls

We can compile this with mpicc Hello world. C, mpicc is the compilation command. So, this is the wrapper over the GCC compiler. In GCC it is , mpicc similarly mpic++ in c + + ,mpif 77, mpif90 in Fortran 77 and 90 respectively is a program that wraps over an existing compiler to set necessary command line flags when compiling the code which is using MPI.

So, in OpenMP it was a compiler option, but here it is a program which is the wrapper over the compiler. Typically, adds few flow flags to enable the code to be compiled and linked against MPI libraries in C,C + + and Fortran. MPI is also supported by other languages like Java, Python, Julia, R, MATLAB etc.(Julia MPI works very well).

(Refer Slide Time: 32:51)



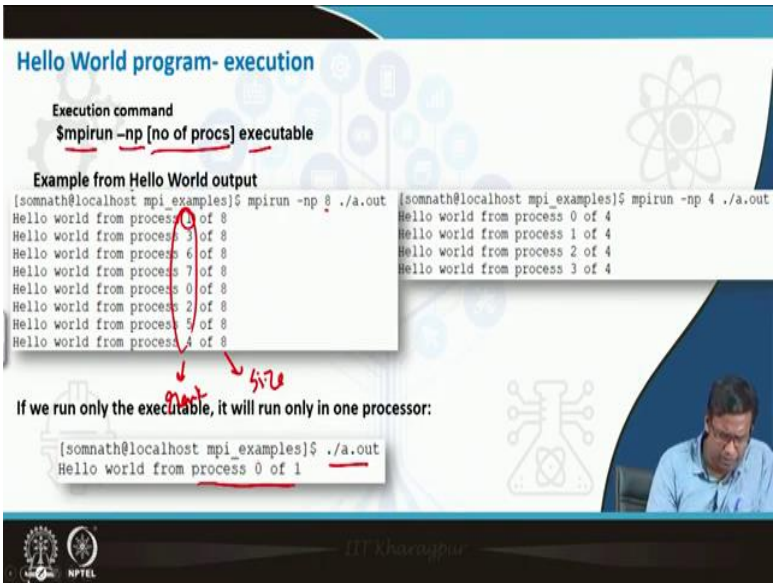
Well so, if we write this program in Fortran, we have to include mpif .h, for Fortran the header file is mpif .h and all the functions are now subroutines and they are called and the input is not void or some address space, but rather an integer ierr. So, there is an integer value with error code which will be input of all the functions.

If the MPI call is successful, this error code will return non garbage integer value usually, it is 0, if it is unsuccessful, it will run something garbage. So, there is an error code which comes with all the MPI function calls and these are not function calls, they are subroutine calls, because it calls these functions.

Well this is the same Hello world program and instead of just writing the function ,the function call is written and the function call requires an error code ierr that is the difference in Fortran

and it has to be compiled with mpif 77 hello_world.f or mpif 90 if its Fortran 90 .This is Fortran 77, but can also be compiled in Fortran 90(mpif 90 Hello_ world. f). So, again I am using another wrapper here.

(Refer Slide Time: 34:17)



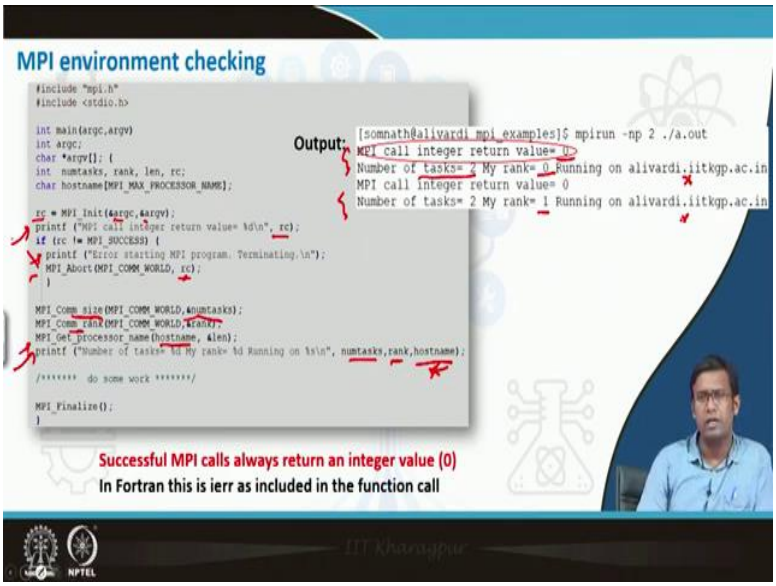
Now, the execution is MPI run - np then number of processors that you want to run and the executable file. So, MPI run - np 8 a.out and the output will be Hello_world from each process with its rank and the size. So, each of the processes is writing its rank and that size of the processes.

If I run it in 4 processes, then 0, 1, 2, 3. Now, you can see though it is written sequentially, but it is not ensured that sequentiality will be maintained. There is no specific order in writing by the processors, because it's different processors which are writing to the screen and they can do it in their own way, in order. So, the order is not maintained when you are writing MPI.

Output from MPI is very similar to OpenMP. In OpenMP still you can control it using the ordered construct, but in MPI this synchronization across the processors is not possible. So, you cannot make an order (Refer Time: 35:21).

Well, if you simply run the executable it will run on a single process. If you do not write mpirun - np then an integer value of how many processes you want to run and the executable, it will simply run the executable this will be running as a sequential code and it will run in a single process.

(Refer Slide Time: 35:46)



It can be important to check the environment in which an MPI program is running. So, we write a program which will write down the environment. What will it write? First what is the output of MPI_init? We say in Fortran MPI_init comes with an error code ierr in, c it comes with an argument argc and argv and the output is rc. It is a function, there should be some output of the function what is the output of the function.

So, this output is an integer and will write the integer value .This value is same as the ierr in Fortran, then if this value is not equal to MPI success(MPI success is assigned to be a value 0), then there is no success, and it will write “error in starting MPI program” and then it will call MPI abort and the program will come out of the MPI.MPI is not finalized, it is not finish finished, but the communicator will be destroyed, because MPI is not running correctly.

If not destroyed, then it will find out the COMM size, the COMM rank and MPI_get_ processor name. This MPI_abort and MPI_get_processor name are not part of our six function calls which are sufficient to write an MPI program. So, though MPI is small, but MPI is also large we can add more function calls in, I mean we can use some features which are not those six basic calls.

Out of that one of them is MPI_abort similarly one is MPI_get_processor name which will identify the name of the processor which is executing the job and write it and then you can physically see which computer is executing that program.

If you write MPI rank it will tell you which process within the communicator, which process id is executing the job. If you use MPI_get_ processor_name so in host name it will be written which processor is actually executing the job. You will see the name of the processor, the host name, and this will help you to check the MPI environment.

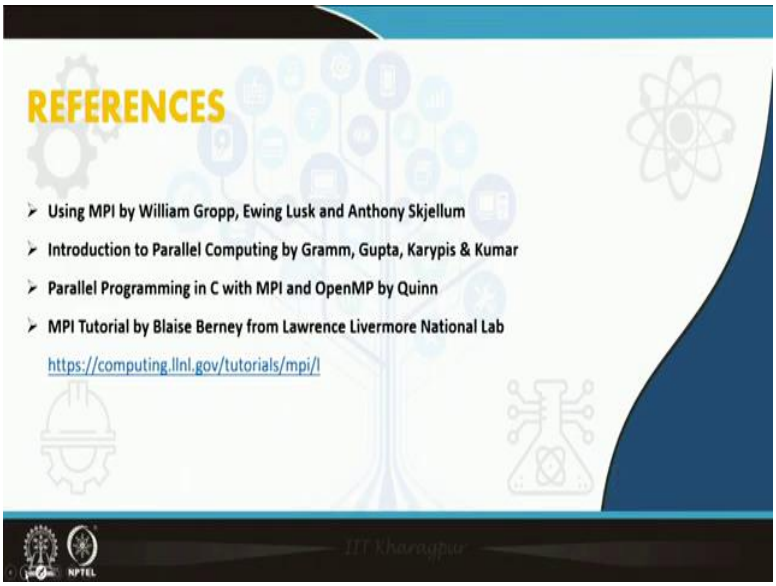
So, once you execute that it runs in two processors. Now you can see each process writes printf .MPI rc is equal to MPI success, so MPI has worked fine. So, each process is executing one of the all the printf statements. So, it is MPI called return value rc, rc gives the value 0, the return of MPI_init call is 0 and it writes number of tasks 2, 2 is the MPI_comm_size that is num tasks is 2, the rank is 0 for this processor and running on the host name, which is running on a machine called Alivardi from IIT Kharagpur ,alivardi. iitkgp . ac. in.

In the same machine another process is running in another(it is a multi-core machine) processor of the machine. So, the second one is also running the same, writing the same name, but it is the same machine in which multiple processors are executing different processes and the rank is 1.

So, successful MPI calls will always return the MPI_init or , all these functions can be associated with one integer value and this integer value, if the call is successful , is 0. In Fortran, we can probably go back and see Fortran code.

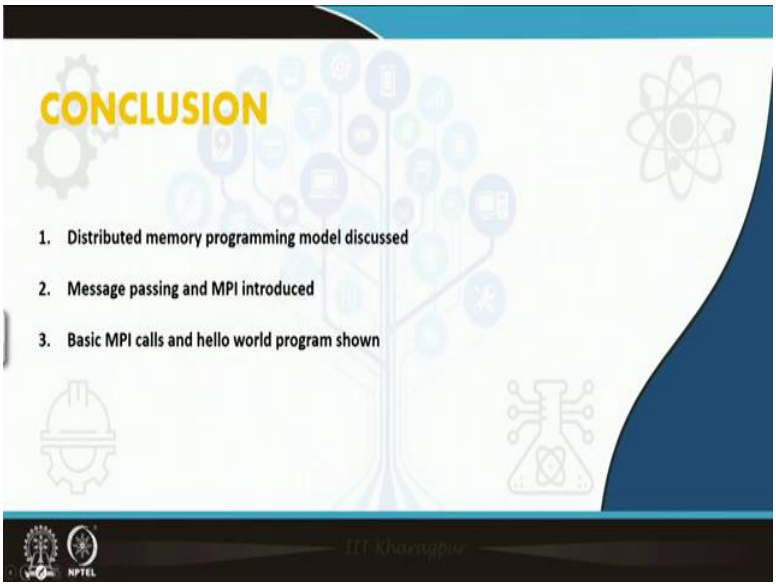
In Fortran this ierr is there with all the MPI calls. In Fortran if it is successful then ierr is equal to 0. This gives the same function call.

(Refer Slide Time: 40:14)



Well in the first basic discussion on what MPI programs do look like and what are the important features including communicator. These are the references. Specifically, William Gropps book is a canonical text in MPI and you can look into Lawrence Livermore National Labs website to get MPI tutorials. We will discuss some of the problems.

(Refer Slide Time: 40:40)



We have discussed the distributed memory programming model, message passing interface and MPI is discussed and basic MPI programs and hello world programs are shown. Based on that we will work on them, send receive operations in MPI in the subsequent classes.