**High Performance Computing for Scientists and Engineers**

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

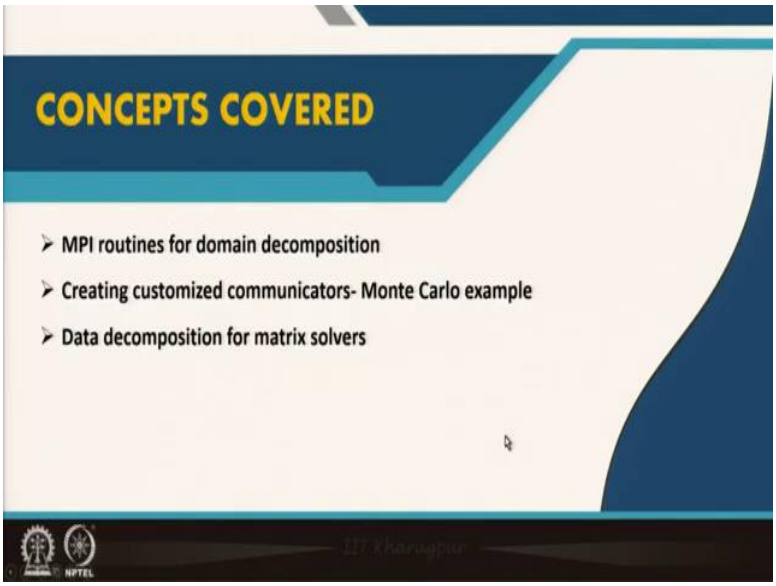
**Module - 03**
**Distributed Computing using MPI**
**Lecture – 29**
**MPI routines for parallel matrix solvers**

Welcome to the class of High-Performance Computing for Scientists and Engineers and this is the 3rd module of the course which is on MPI programming for distributed computing. We have discussed about the preliminaries of MPI programming and we have also looked how domain decomposition method can be utilized for parallelizing matrix solvers for Laplace or Poisson equation type of problems using MPI and therefore, we understand that MPI has a good applicability for parallelizing matrix solver algorithms.

So, in this particular lecture we will further explore MPI's applicability and we look into some of the features little advanced from the basic features which you have looked at till the last few classes.

This is kind of an intermediate MPI ,not the most advanced features of MPI. However, we are discussing some new features of MPI programming using which we can more efficiently parallelize matrix computing programs and matrix solvers. So, the title is MPI routines for parallel matrix solvers.

(Refer Slide Time: 01:45)




We discussed MPI routines for domain decompositions. We will see some of the features of MPI using which domain decomposition becomes easier to implement compared to the way we have seen domain decomposition earlier. Especially when we have to do domain decomposition in multiple dimensions. Earlier we have only done one dimensional domain decomposition or when we have to do domain decomposition for more complex geometries these features can be used. We have looked at the communicators, MPI communicator is basically a topology combining all the processes working for a program and also defines the inter connectivity across the processes.

So, a communicator is by default setup when we call MPI_init and this is the default communicator. We can also create customized communicators, we can split the communicators, we can take out some of the processes and form a new communicator. We will see these communicators and we will take an example.

Now, this is not a matrix computing example, we will see a Monte Carlo programming example for that. Monte Carlo is a probabilistic method applied to solve deterministic problems and we will see how we can compute the value of pi using Monte Carlo method. If you remember while discussing openMP we have shown you how to calculate the value of pi using numerical integration.

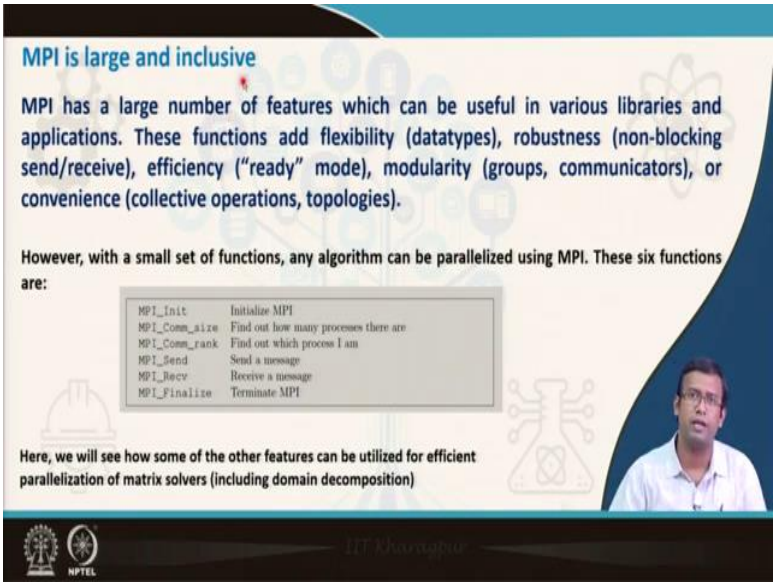
Now, we will see how we can do it using Monte Carlo, but this is not a matrix computing method. This is little shift from what we are discussing on matrix computing, but this will be a

good example and Monte Carlo type of simulations are abundantly used in different scientific computing exercises like molecular mechanics simulation, stochastic model-based simulation etcetera. So, this is a good exercise if we look into the Monte Carlo example, but our focus will be how to see creation of customized communicators using MPI and then we will see a data decomposition method for matrix solver.

What we are doing using domain decomposition is basically decomposing the domain into multiple sub domains and solving some matrices there. This is more of a task decomposition. So, you are breaking down the main task into multiple smaller tasks when solving them.

But, if we try to do a data decomposition; that means, the matrix vector multiplication will be itself decomposed and given to multiple processes, which I have done when we have looked into threaded parallelism using openMP for matrix solvers or matrix vector products, but we will see how that can be done using MPI here.

(Refer Slide Time: 04:47)




So, we have earlier discussed that MPI is small as well as large. Quote like this is given in the canonical textbook for MPI William Groff's MPI book that MPI is small in a sense there are only six functions with which we can write any MPI program. When we have looked into the previous domain decomposition method, we have used 7 MPI clauses apart from the reduction clause .All belongs to these six clauses even reduction can be replaced by send receive type of clauses.

So, there are few basic MPI calls by which you can parallelize any algorithm. However, there are a large number of features which can be used for different libraries and applications using MPI. These are apart from the six basic MPI calls which are basically, MPI_init, Comm_size, rank send and receive, finalize. With this you can parallelize everything, but apart from that there are other MPI features or other MPI functions.

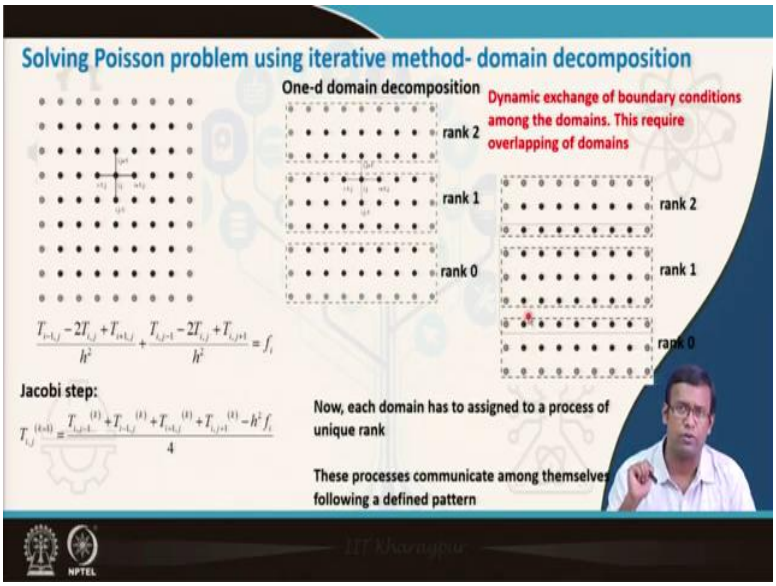
Some of which we have looked earlier. MPI_gather, MPI_Scatterv, etcetera which we can use for simplicity in programming as well as for efficient MPI programming or efficient parallelization. These functions sometimes add flexibility in terms of data types.

They add robustness like you can have non-blocking send receive buffers send receive they are efficient in many cases because they use the grid topology in a more efficient way and talks to compiler and operating system better .They sometimes give you modularity which we can group make groups and make communicators out of it and for collective operations. So, they are convenient for writing the program as well as they are efficient in terms of the parallel overheads. That is why you call MPI large and inclusive; a large number of features exist within MPI. Apart from these six MPI functions with which you can parallelize any program, if you want to do it with more flexibility, convenience, simplicity in programming , less overhead if you want to ensure that some of the wait times and again overheads are smaller, then you can use some other features and these features precisely are what we will try to discuss in today's class.

At the end of today's class when we will discuss matrix vector product parallelization .I will show you an example where a sub subset of these six functions are used. Another feature is reusability, I mean what you have written in 30 lines of program is compressed to 1 or 2 lines of coding and also this is efficient in terms of overhead. So, these are the advantages. For using the other features either you have to be an expert or you should have good experience so that you know which features have to be called upon . With advanced MPI more features are coming you need to look into a MPI manual and find out the feature which will suit there.

Here we will see some of these features which are part of the large and inclusive list of MPI functions that can be utilized for efficient parallelization of matrix solvers. So, we will look into domain decomposition-based matrix solvers first and then we look into data decomposition-based matrix solvers.

(Refer Slide Time: 09:12)




So, we look into a Poisson problem which is Laplacian of a function say Laplacian of T is equal to a source term in the right-hand side and that will be solved using numerical methods. So, use a finite difference method. These are the finite difference grid points and form the discrete difference equations for each of the points see. Then we will use an iterative method to solve it. Say for example, we are using Jacobi iterations to solve it. So, this is the Jacobi iteration step and now we will use domain decomposition for this. We have discussed how we can use domain decomposition for matrix solver problems.

So, instead of writing it in matrix form we have written it in indicial form. We will use a domain decomposition method for this. So, the entire domain is divided into three sub domains and black points are the internal points, gray points are the boundary points and we will solve for the black points in each of the sub domains.

Now, we can see the inter domain boundary points for solving this one point resides in the neighboring sub domain. So, we need data exchange across the processors. Dynamic exchange of boundary condition among the sub domains will be required and that will require overlapping of the domains. These things we have discussed in the last few classes.

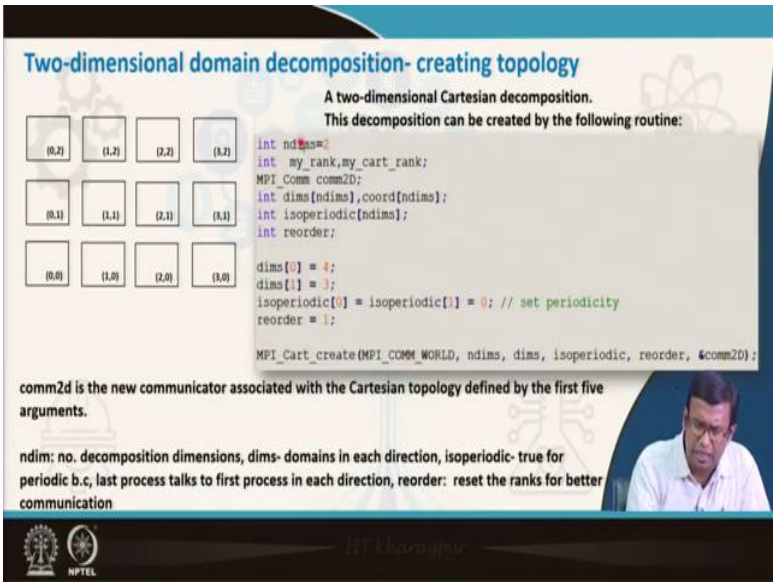
So, this will be an overlap domain and inside this domain we will solve only for the internal points. The outer layer points will come as boundary conditions and they are the values which are copied from the last updated solution of the neighboring sub domains and this is everything. We will follow a Jacobi step here.

While solving these different processors will be responsible for solving different parts of the sub domain. So, we have to assign each domain to a process with a unique rank. So, for each of the sub domains we will assign a process and there will be a rank for this process. This is in a nutshell what we have done earlier also while discussing domain decomposition.

Now, in this class we will bring little more complexity in the problem and see how MPI features can help us. This process will also communicate among themselves following a defined pattern and this is important. Say rank 0 will communicate with rank 1 because it has overlapped with rank 1. Rank 2 will communicate with 1, 2 is never going to communicate with 0.

So, there is a definite pattern by which the processors will communicate among themselves and which data will be communicated to each side where it will be mapped that also follows a pattern well.

(Refer Slide Time: 12:29)




Now, if we look into a two-dimensional domain decomposition problem, so, instead of decomposing the domain into one direction we decompose it into both directions. So, it is a two-dimensional domain decomposition and we get multiple sub domains. Now, each of these sub domains can be identified by an index in the x direction by an index in the y direction and similarly it will be mapped to the processes and the communication pattern will be little more complex than the previous one.

For example, this sub domain (1, 1) will communicate with the neighboring sub domains, but will not communicate with the further sub domain and these neighboring sub domains are in both x and both y directions. This decomposition can be created using the following routine.

So, earlier when we have created the decomposition, we simply divided the number of points by the number of sub processes and created points in x direction by number of processes and created chunks of points in x direction and created different sub domains.

But here we have to create sub domains in different dimensions and this can be created by MPI function. So, early you remember that we took the number of points, found out the remainder, distributed in different processes, assigned each of the sub domains to each of the process, so on; this we have done while doing the domain decomposition through a simple c program. But we can directly use an MPI feature which is MPI_Cart_create for doing that. When we do MPI_Cart_create ,the main geometry is divided into multiple sub domains and each sub domain gets associated with one processor. New sets of processes are mapped on different sub domains, and we create a new communicator comm2D.

MPI_COMM_WORLD was our initial communicator, but we created a new communicator comm2D. Why do we create a new communicator? Because this new communicator uses a topology like this, where it knows (2, 1) will communicate with (2, 2); (3, 1);( 1, 1) processes.

So, the communication pattern is also known to the communicator ,not only is it only groups all the processes, but it also keeps a track of the communication pattern or it keeps a track of which process is neighbor to which of the processes. So, a new communicator is created. Now, we will see this new communicator is basically a communicator which is mapping of the sub domains. So, it divides the domain into multiple sub domains and maps them to respective processes.

This entire set is stored in this communicator. So, the output of cart create is that we get a sub domain divided geometry with the mapping itself. Now, this output is comm2D the new communicator and the input is old communicator. ndims tells us that in how many dimensions we are doing the domain decomposition.

Here we are doing in 2 dimensions. So, what are the number of blocks in x direction, what are the number of blocks in y direction? 4 and 3; these are the input of dims.

Then there is something called isoperiodic. This is also important. Many times we use a periodic boundary condition. Say we have a circular geometry. In theta direction there is a periodicity. So, what goes to theta 0 that is the same as theta 360 degree. Similarly, sometimes you use periodic boundary conditions there in 2D also, Cartesian geometry also; in that case 3, 1s neighbor will be 0, 1. So, there will be a data exchange between (3, 1) to (0, 1).

If we set isoperiodic to be 0 then there is no periodicity that means, these are physical boundaries. It can be done in both x and both y. If we set isperiodic in any of the directions to be 1, there will be a periodicity. (2, 0) will be mapped as an if there is a periodicity in y direction the 0 line will be mapped as a neighbor of the 2 line.

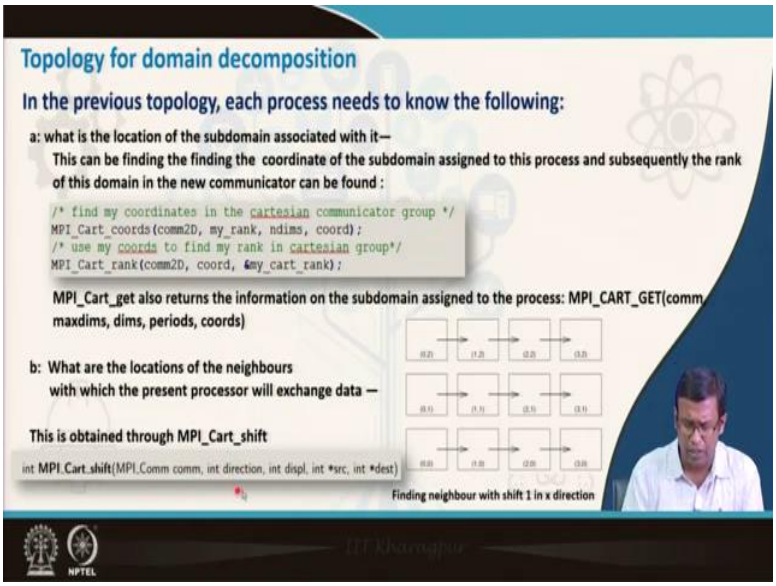
So, there will be another set of communication across 0 and 2 by which we can also control that. There is another term called reorder. Reorder reorders the ranks of the processes. Earlier the processors ranks are obtained by MPI comm rank whatever they are, but once you create this then different processes say (1, 1) has a neighbor (1, 0),( 2, 1),( 0, 1), (1, 2). So, if based on the hardware of the system if they can be mapped well so that (1, 1) and (1, 2) goes to neighboring processors, neighboring physical processors data transfer will be faster.

So, keeping them in mind if we put reorder is equal to 1, the ranks are reassigned, so that processes are considered to be mapped with the sub domains and the ranks of the processes are so arranged that the communication is faster.

So, what is taken away from this entire exercise is distributing into multiple sub domains, giving each sub domain to different processes and looking into the communication pattern between them, finding out who is a neighbor of what; this entire exercise is done by one feature one function MPI_Cart_create.

This is not part of the elementary MPI function calls, but if we can call them, we can very easily and simply distribute the geometry into multiple sub domains and map them on to the right processes.

(Refer Slide Time: 18:55)




This is called a topology or a virtual topology. We have created a new communicator and how these processes are virtually connected with these neighboring processes combining the interconnected is called a virtual topology. So, when we define MPI_Cart_create and new communicator we define a new topology.

Now, in the previous topology we have these different processes and their mappings and the neighboring processes. The communicator has the entire map of the sub domains and their mapping into the processes. Now, each process needs to be aware of certain facts for executing the program. First is who am I; what is the rank of the process, what is the location of the domain assigned to that process.

So, what is the location of the sub domain associated with it? This can be done by finding the coordinate of the sub domain assigned to the process and subsequently, the rank of this domain within the new communicator. Like, using MPI_Cart_coords we give the communicator id, the new communicator name , the rank of the process within the old communicator and coordinates of the sub domain can be found out. So, using the rank of the process in the old communicator and the new communicator id, the coordinate of each of the processes can be found out. Using this coordinate and the new communicator one can find out what is the rank of the process in the new reordering. In the new communicator what is the rank of this particular process and this rank will be given as my_cart_rank.

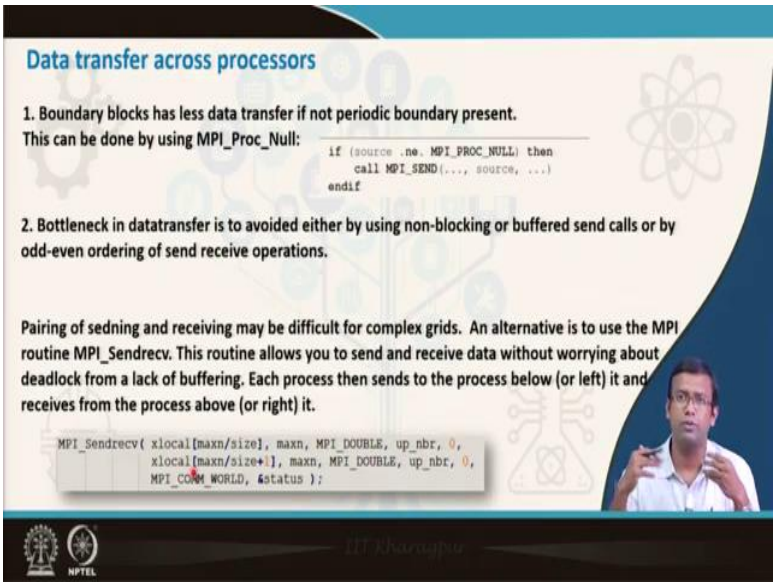
So, by my_cart_rank one can identify a particular process mapped on the particular subdomain and what is its location. Also, my_Cart_get can return the information on the sub domain, if MPI_Cart_create is executed then the topology has been created. Now, if you get my MPI_Cart_get then each processor knows where it lies with respect to the topology and what is the overall topology that goes to each of the processors.

Now, next important part will be what are the locations of the neighbors, with which the present processor can exchange data. So, each processor will exchange data with the neighbors, what is the location? That can be obtained through MPI_Cart_shift. What is MPI_Cart_shift? That in each direction will be the displacement of 1; so, in x direction displacement of 1. So, what is the location of the processor and what is its rank with plus 1 coordinate and minus 1 coordinate? So, we can find out that say for processor (1, 1), what is its left and right neighbor.

Similarly, if we put the direction to be y or the second direction and put a displacement 1, we can find out what are the rank of the processors, what is the process we are looking into and what are the destinations where the data should go in plus y minus y direction. So, using MPI_Cart_shift a processor can know what its neighbors are. So, with this information available it now becomes simple to write down the domain decomposition program.

So, the creation of the topology domain distribution of the domain into multiple sub domains has already been taken care of by MPI_Cart_create. Now, with these functions you can distribute into you it is already distributed to multiple processors. Each processor can know what is its coordinate, what is its rank and what are the ranks of the neighboring processor, so that it can do data exchange.

(Refer Slide Time: 23:30)




But another important thing is that in case the geometry is complex it is difficult to use simple MPI programs and write your own algorithm to do domain decomposition and the mapping. These types of functions are very useful then. So, data transfer across the processors is the next important part. First you do domain decomposition. Assign different processes to different sub domains and then you have to write the solver and do data transfer across the processes.

Here you know the source and destination you got from MPI_Cart_shift and using that for data transfer. Now, in case the boundary is not periodic boundary then the destination is not found out. There is nothing in the x direction for the last boundary last sub domain. So, then the data transfer can be done using MPI_Proc_Null that if you found that there is no process after that in x direction then that the id will be MPI_Proc_Null.

In case the source or destination is not MPI_Proc_Null, then only MPI send and receive will work. So, if nothing is found that rank will be given as MPI_Proc_Null. Bottleneck of data transfer can be avoided by non-blocking or buffer send receive or odd even ordering of send receive operations.

Now, if the geometry is complex, if you have a complex grid it might be difficult to find out the exact odd even number. There may not be odd even numbers. The mapping can be done in a very unstructured manner. So, one alternative is using buffers, but there are certain issues while using buffers because send data goes to a buffer and waits for the right receive, but other alternative is to use a MPI routine called MPI sendrecv.This routine at one instance launches

sending from a call to a processor and also ask the processor to receive data from this other one because whenever there is a domain overlap one domain is sending data to other the other domain is also sending back data to here. So, a sent boundary is also a boundary through which sending is happening from one processor to another through that the processor which is sending he is also receiving something.

So, send receive does not put a deadlock due to lack of buffering, rather launches the send and receive process across the processors at one instance. This is a send receive call through which data can be sent from one particular processor to the up neighbor and as well as data can be received from the processor from the up-neighbor processor to this particular processor. So, this is one way out here to avoid bottling neck that use a combined send receive.

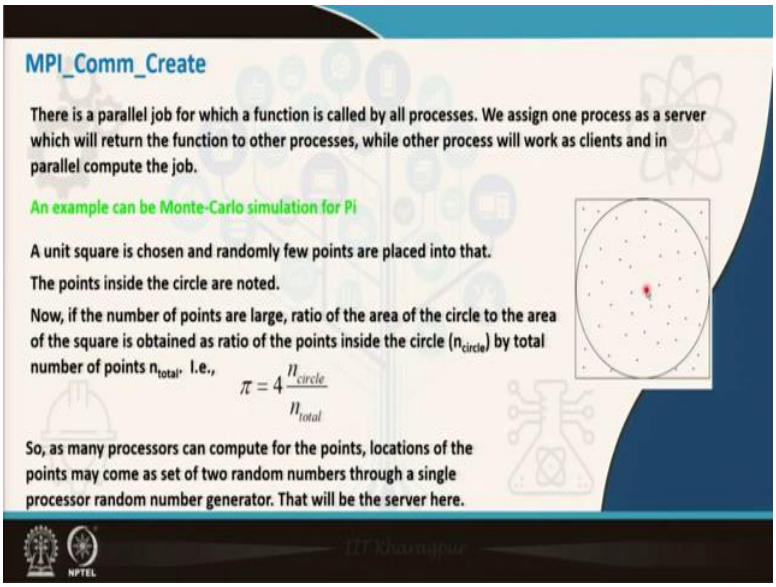
(Refer Slide Time: 27:00)


Next important part will be creating communicators in MPI. MPI_comm_world is the default communicator in MPI. However, a group of processes may be separately taken out and combined to form a new topology. A topology means the collection of the processes and the interconnectivity among them and a topology is defined within a communicator. So, a new communicator can be formed. MPI_comm_world is the default communicator.

However, from MPI_comm_world you can separate out some processes and create a new topology and a new communicator. This can be done by forming a group of processors by MPI_Group_excl .Some of the processors will be excluded from the original number of processors and a new group will be formed and then a communicator is created within the new

group using MPI_Comm_Create.Also we can split the original total number of processors and build up multiple communicators within the main MPI_comm_world; this is done using MPI_Comm_Split.

(Refer Slide Time: 28:13)




Say there is one job which is done in parallel by multiple processes. Now, all of this process calls a function. This function is serially computed or one particular process can give this function. This function is not a very compute intensive job, but there is only one process who can do this function. So, all the other processes will work as clients of this process and it will return the value to the other processes. Whereas, the other processes will be working in parallel for this particular job and do the parallel computing. The example can be Monte Carlo simulation. Monte Carlo is famous for casinos. People go there. There are a lot of risks involved in waiting in casinos.

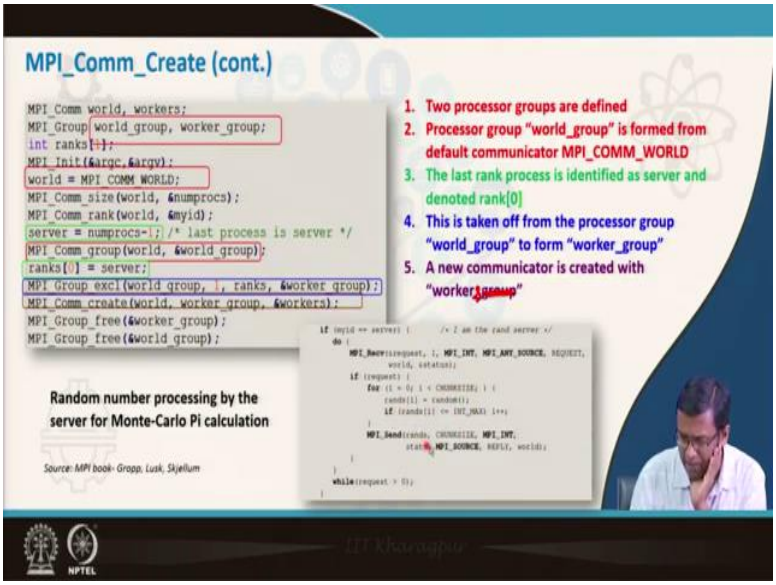
So, these people came up with some ideas involving taking risks and probability. This Monte Carlo method is based on a probabilistic method for calculating a definite answer. These probabilistic methods for calculating deterministic solutions are often named as Monte Carlo simulations. One example is calculating pi,this is like a dart board playing.

You have a square dart board and you throw darts on it and then these dots randomly hit the board. Then calculate what are the number of the darts inside a unit circle within that and this number divided by the number of the total dots represents the area of the circle divided by the area of the square in case the number of throws is large.

So, units square is chosen and few points are placed into that. The points inside the circle are noted. Then if the number of the points are large, it can be shown that the ratio of the area of the circle to the area of the square is obtained as the number of the points inside the circle by the total number of points. That gives us that pi is equal to 4 times the number of the points inside the circle by the total number of points; provided the total number of points is large enough.

So, one has to do computing over a large number of points and these points come as the random numbers within 0 to 1; with both x and y coordinates. Then find out what are the number of points inside the circle by measuring the distance from the center and do this calculation. The location of the points may come as a set of two random numbers. As many processors will be there the total number will be large. Again, generation of random numbers in parallel is difficult, as sometimes they give copies of the same number. So, even if there are multiple throws all throws by different processors come to the same location. So, it is usually done that a single processor is asked to generate the random numbers, this acts as the random number server and the other processes gets the information out of it.

(Refer Slide Time: 31:35)




So, we have to create a new communicator with the processes with which we will calculate the value of pi and there will be one process which is taken out that will work as the server process and that is done in the following way. We first create two processor groups: world group and

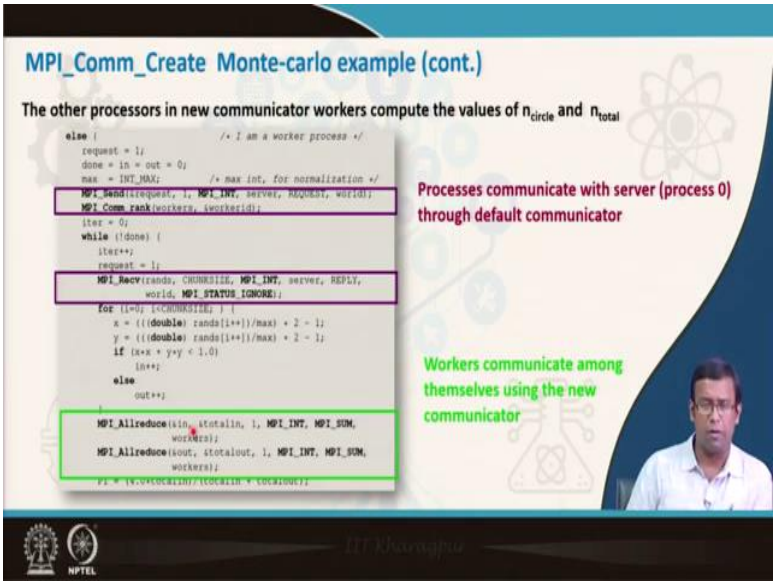
worker group. These are the processor groups. They are not communicated, not topologies, they are simply processor groups.

What is the difference between a communicator or topology or the processor group? Processor groups only have a defined group of processors defined. When I create a communicator, topology is created which considers that these processors are now connected to do a job concurrently.

So, the interconnection between the process is a part of the communicator, but not the part of the processor group. So, you create two processor groups. Now, what is coming from the MPI_COMM_WORLD this communicator is named as world and these processors are grouped into world_group processor group. From world_group, now the last rank processor numprocs - 1, is identified as a server. It is given as a rank 0 and this rank 0 is excluded from world_group and the new group is created as in worker_group. Within the worker_group, so, the new group worker_group is formed. Within the worker_group MPI_Comm_create is given and a new communicator named workers is created. We create a new communicator named workers out of this worker_group.

Now, if we are doing random number processing the server that is getting requests from the other processes and generating a random number and sending the random number back to the source the process which has sent it .

(Refer Slide Time: 34:04)

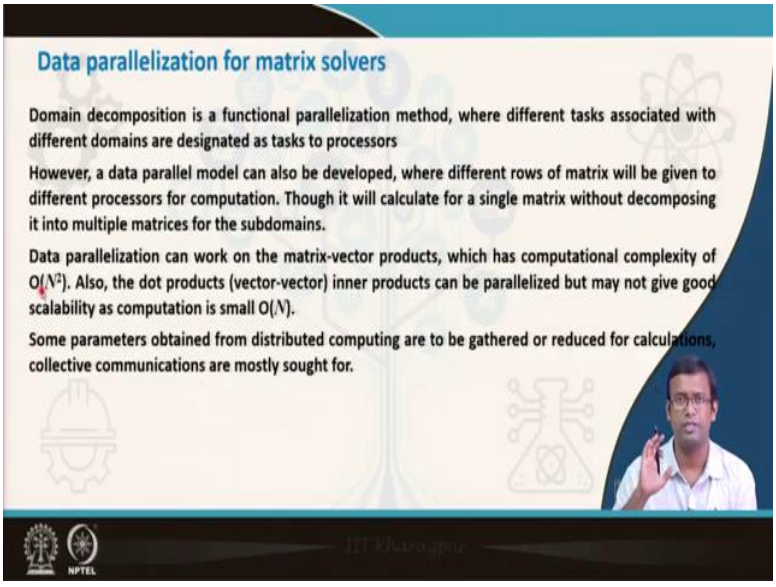


The other processors send requests to the server and then they get the random number from the server and puts it as x y axis, sees that whether this is within the radius is less than 1 then considers it within the circle otherwise outside the circle. Then use all reduce to find out the total number of points inside the circle and outside the circle and finally, pi is calculated.

Now, you can see that when the processors in the worker_group communicate with the server they are using the default communicator MPI_COMM_WORLD. When they are receiving from the server, they are also using MPI_COMM_WORLD. When they are finding rank within the new communicator, they are using the new communicator id workers. When the client processors are communicating among themselves, they are using the new communicator worker. So, two different communicators are created to ensure two different working patterns. We do not want to bring the server into allreduce. So, we have excluded it and ask the other processors to do it.

So, using simple if else look this could have been done, but with MPI send MPI receive type of comments, it will be more complex in terms of programming. As well this implementation is more efficient.

(Refer Slide Time: 35:37)




Now, we come to the last part of the discussion quickly. We will go through data parallelization for matrix solvers. Earlier we have looked into task parallelization or domain decomposition. Domain decomposition is a functional parallelization where different tasks associated with

different domains are designated as tasks to different processors. Each processor is doing a task and this is coming from the domain decomposition.

However, a data parallel model can be developed. Well instead of breaking down the geometry into multiple sub domains and making small matrices out of each sub domain we ask we assign different rows of the matrix to different processors and so, we do not decompose the problem domain into multiple sub domains. We have the main matrix. We have a large matrix. We decompose only the rows of the matrix into small rows of the matrices, but do not decompose into sub matrices. We only decompose the data and give different rows to different processors to work.

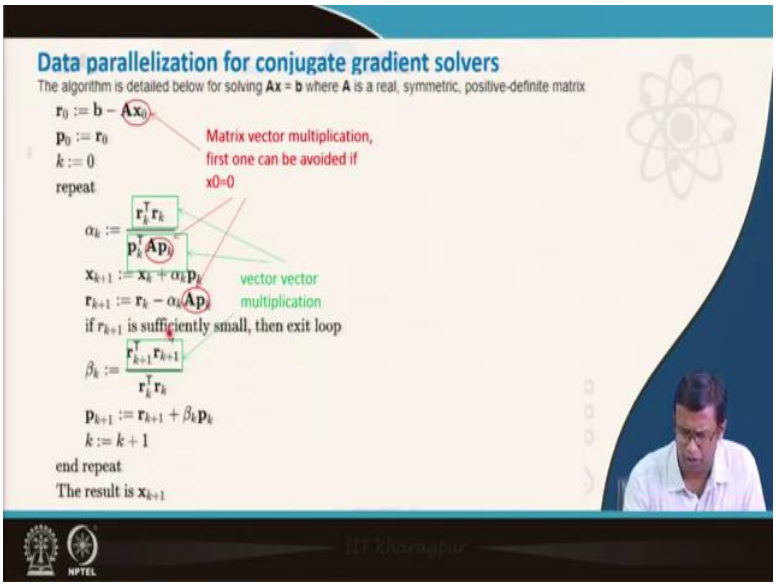
This is simpler in terms of the implementation and in terms of iterations also maybe because we do not need to exchange data across the boundaries and reiterate it.

Data communication is also less here. Data parallelization can work for matrix vector products, which is complexity of the order of $n^2$. We have seen that. The matrix vector product is part of any iterative matrix solver. So, that part can be taken care of by data parallelization. The dot products or vector products can be parallelized, but we have seen in openMP programming that vector dot product parallelization is not very computationally efficient.

In case the number of points in the vector is small then we do not need to do the dot product parallelization. It is not very worthy, but this part for a large matrix can be done, but for medium size matrix, the matrix vector product parallelization is also efficient.

Some parameters obtained from distributed computing are to be gathered or reduced for the computation. Like, if you are doing dot product or if you are doing matrix vector product different processors generating different parts of the vector. So, they can be communicated by using collective communications. You can use send receive type of operations. We will show some examples, but collective communications are better.

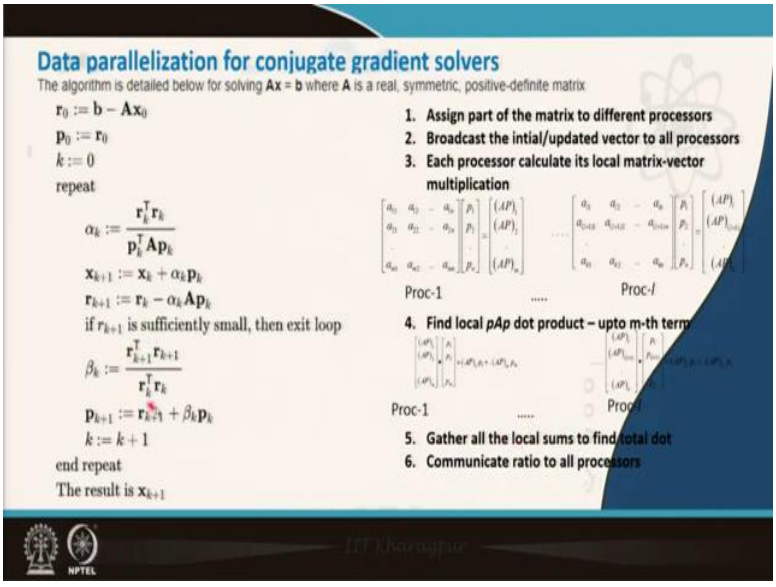
(Refer Slide Time: 38:03)




So, we see a conjugate gradient solver here. We can see that it starts with a guess and then calculates the value alpha. While calculating the value alpha, (b, r ,P, x are vectors) there is a matrix vector multiplication, this multiplication is dotted with a vector and there are vector multiplication, dot products.

So, you can see that there are three matrix vector multiplications. One is done only at the first time at the initial guess, but if the initial guess is 0, we do not need to do it. The other is another matrix vector multiplication which is used twice. There are many vector dot products.

(Refer Slide Time: 38:50)



Now, specially the matrix vector multiplication is computationally costly. So, we need to parallelize it. What will we do? Assign part of the matrix to different processors. Broadcast the initial or updated vector to all the processors.

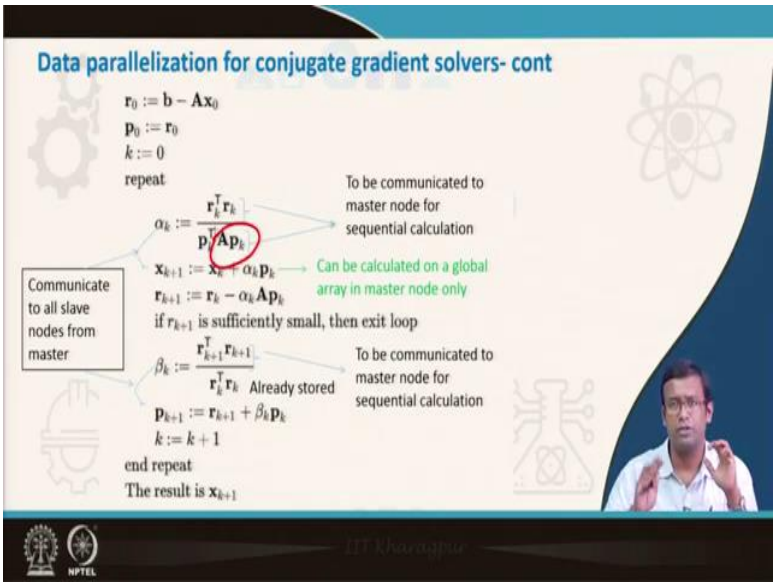
So, take the vector and broadcast it to all the processors because vector is small, broadcasting it will not be difficult. Matrix is large. So, take parts of the matrix and give it to different processors. Each processor calculates local matrix vector multiplication, like Processor 1 calculates matrix vector multiplication up to m rows.

Processor 2 will take from m +1 row and calculate, so on and Processor L will take from lth row and calculate up to kth row matrix vector multiplication. Each processor is doing a part of the matrix vector taking a few rows of the matrix and getting the matrix vector multiplication. They have the entire vector and few rows of the matrix available to them.

So, they are getting few rows of the matrix vector multiplication. Finally, $p^T Ap$ product is required. So, in each of the local vectors you take the p vector and get the local dot product . Then this part of the dot product is calculated in all the vectors. Gather all the local sums to find the total dot because finally, we need a dot product out of it. We do not need the matrix vector, but we need the dot product. Because p is a small matrix we can distribute into multiple processes and or keep the copy of p to all the processors and then get the local dot product also. It is not much communication here.

Finally, you get the local sum and, you have to calculate the ratios in one processor and then it has to be communicated to all the processors.

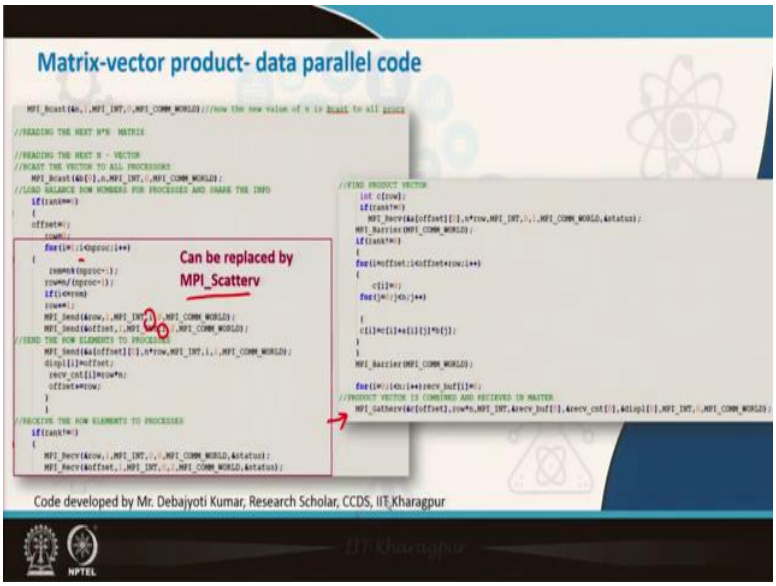
(Refer Slide Time: 40:35)




We can see that these dot products have to be communicated to master nodes for sequential computing . So, instead of communicating it you can take the entire vector to the master node and the master node can do it itself also.

But only this Ap product is a computationally complex one that has to be parallelized; there is no other way of that. Then once these computations are done then alpha and beta are to be informed to all slave nodes or all other nodes for the rest of communication because alpha and beta will be used for local calculation in all the nodes. This can be calculated in a global array in the master node. Instead of communicating many times only Ap can be calculated in slave nodes and finally, using gather the Ap can be calculated by the master and the master node or one of the nodes can take care of the rest of the computing because these are computing only on a vector; they are small computing. The only computationally complex part is this matrix vector multiplication which has to be parallelized.

(Refer Slide Time: 41:59)




So, you will see a quick example of matrix vector multiplication parallelization. The vector is broadcasted to multiple processors. Now, the matrix is read or obtained from some other source; this matrix is.

So, the part of the matrix has to be given to different processors. So, first is the vector, the number of rows n is broadcasted to all the processors. The vector b is also broadcasted to all the processor because all the processor needs the entire vector, only a few rows of matrix is needed.

Then we find out how many rows and from which row each processor will start. We send the number of rows and the offset from which each processor will get from the master node which has the entire matrix to the other processors. Then we send the row elements from which offset that the rows will start and all the rows like after 5th row next three rows will go to Processor 2.

Similarly, we set the location of the first row offset and the number of the rows and the size of the data that will go. Similarly, the processors will receive the row and the offset and the matrix the actual matrix from this processor.

Now, this can be replaced by Scatterv. Instead of using send receive this becomes a little complex we can efficiently use Scatterv here. Here we can do this. So, it will receive this

receive also. This will come in the part of Scatterv. It receives from the master node that these many rows will come.

Then each process will do its own matrix vector calculation and there are barriers for synchronization and the product vector you are using a single call gatherv where this product vector is coming to the master processor and then master processor if it wants to do any dot or any other calculation can do it using Gatherv.

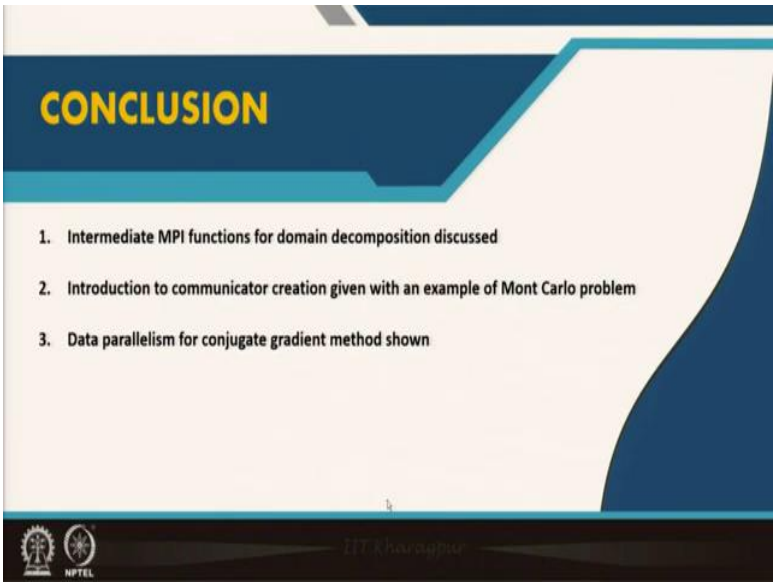
So, instead of using receive and send from each processor to the master processor, we can use a collective call Gatherv and collect the entire vector. What is the difficulty in doing a send receive operation here ,that there is a sent from the master node to all.

There are three sends from the master node to all the processor and it follows this send is from master node to ith node. So, if this send operation is happening over a loop, so, 1 by 1 master node is sending data to all the processors. There is some sequentiality here which can be avoided if we use collective communication. So, collecting communications is very good. Only you have to remember more functions or look for more functions and use it efficiently.

Here just by one call we remove the sequentiality in receiving by master node and not looking into the offset's displacements, difference and receive etcetera ;collecting all the vectors and putting it into the master node. So, gatherv is an efficient communication here. This is developed by one of the research scholars from Center for Computational Data Science IIT Kharagpur, Mr. Debajyoti Kumar.

(Refer Slide Time: 45:32)


(Refer Slide Time: 45:34)




Well, we come to the end of MPI discussions. We have discussed some intermediate functions for MPI for domain decomposition methods. Given an introduction to communicator creation and shown an example of Monte Carlo pi simulation problem here, looked into data parallelism for conjugate gradient method and showed how matrix vector products can be parallelized using MPI. We also see that using the advanced features or intermediate features helps us to do with more simplicity as well as more efficiently, especially data parallelism type of tasks.