

**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 – 26**  
**Domain Decomposition based parallelization of matrix solvers**

Hello everybody, welcome to the class of High-Performance Computing for Scientists and Engineers and we are in the 3rd module of this course, which is Distributed Computing using MPI. Today we will discuss Domain Decomposition based parallelization of matrix solvers. So, this lecture and the subsequent couple of lectures will be devoted to this particular topic.

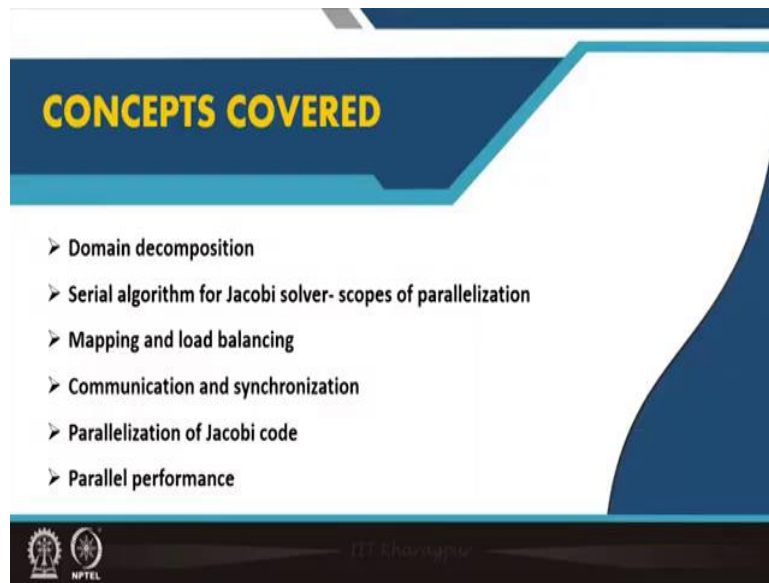
Here we will see the basics of domain decomposition in context of implementing it for a matrix solver and then we will consider a matrix solver; we will consider Jacobi solver for solution of Poisson's equation or Laplace equation in a Cartesian domain

We look into the Jacobi solver C code and later at the final lecture in this topic, we will try to parallelize this Jacobi solver; we will develop a parallel Jacobi solver based on domain decomposition again for solving Laplacian equation in a Cartesian domain, and we will also see its performance in terms of the parallel performance matrices.

Well, these lectures and the subsequent lectures, we will discuss a Laplace equations solution using Jacobi based matrix solvers. However, different equations which are based on spatial geometry, which are based on space can be parallelized using domain decomposition method.

We are taking Jacobi; because this is the simplest solver, but any iterative matrix solver can be taken for domain decomposition-based parallelization.

(Refer Slide Time: 02:09)



Well, we will quickly revisit domain decomposition; we had discussed domain decomposition in the previous few lectures, we will quickly go through domain decomposition once more.

Then we will look into the serial algorithm for the Jacobi solver, we will identify the scopes of parallelization there. We will see how load balancing can be done for that, while we are doing domain decomposition-based parallelization. What are the communication and synchronization aspects of that?

Then we will take a Jacobi solver; it is a C code in which a Laplace equation in a 2 D rectangular geometry is being solved in matrix form.

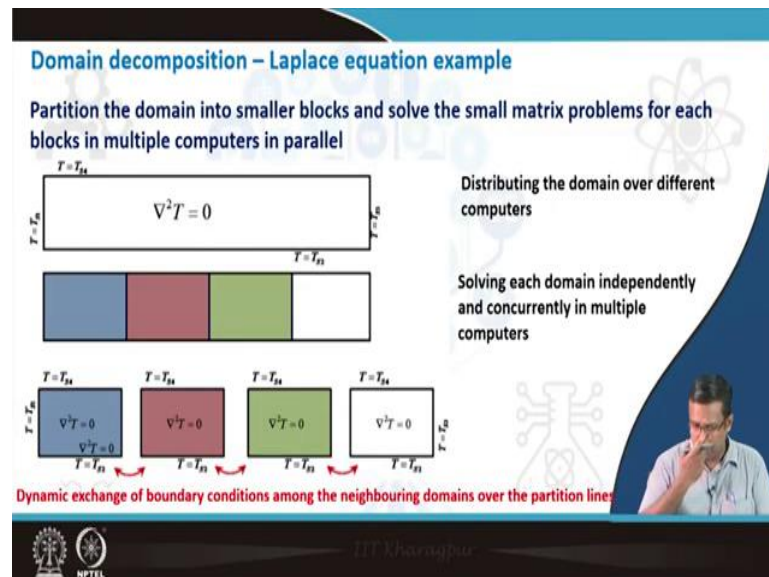
We will see the sequential or single processor Jacobi solver code. Then we will identify the pointers in which we can parallelize it and we will develop a template for a parallel Jacobi solver, and at the final session of this particular topic, we will try to develop a parallel Jacobi solver using domain decomposition-based method.

Domain decompositions are extremely important in terms of parallel implementation for matrix solvers and scientific computing problems. This is one of the simplest examples of domain decomposition-based parallelization, and while looking into parallel performance, we will see certain shortcomings of this method for parallelization also.

There are more efficient methods, there are more complex methods in which we can similarly apply domain decomposition and can get good parallel performance. So, this can be a stepping

stone in order to develop your own parallel solver using domain decomposition for more inward and more complex problems.

(Refer Slide Time: 04:00)



Well, so we will solve this equation  $\nabla^2 T = 0$  in a 2 D Cartesian block. The idea of domain decomposition is that you break the main domain into multiple sub domains; solve the problems independently in each sub domain.

If starting with a differential equation  $\nabla^2 T = 0$  ; it has to be converted into a matrix equation for numerical solution. Solve the sub matrices which are coming for each of the independent domains; break it into multiple domains and solve each of the domains separately. All these solutions are independent solutions.

However, after getting the solution, you exchange the boundary conditions which are the inter domain values which will act as boundary conditions from the inter domain partitions of each of the sub domains and reiterate it. Through that you reach the final converging solution which is a continuous, which ensures continuity and coherence for the entire domain.

So, the idea is distributing the domain over different computers. So, when you distribute the main problem into different sub domains and each of the domains are assigned to different CPUs, each CPU gets a matrix which has much smaller memory than the actual matrix. Therefore, the problem size reduces and also each domain has to be independently and concurrently solved in the multiple computers. Therefore, the problem size has been reduced

as well as these small problems are being in parallel solved in multiple CPUs, therefore the overall performance increases.

However, one of the important points to mention here is that, while solving in order to ensure continuity and coherence of the solution; you have to interchange the inter domain boundary values in between the processors. So, there is always a requirement of data transfer across different processors working in domain decomposition method.

(Refer Slide Time: 06:22)

**Jacobi solver- serial algorithm**

Let us start with the matrix equation  $Ax=b$

Now, let us assume a trial solution :  $x=x^{(0)}$

Obtain updated guess  $x^{(1)}$  as:  $x_i^{(1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(0)}}{a_{ii}}$

For all rows in the matrix

Continue till convergence:  $x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}}{a_{ii}}$

$\max_i |x_i^{(k+1)} - x_i^{(k)}| < \epsilon$

So, we look into a Jacobi solver and look into the serial algorithm first. As I mention Jacobi is one of the basic iterative solvers; we are solving a matrix equation  $Ax = b$  using an iterative method. We consider a trial solution  $x = x^{(0)}$ , as our initial guess.

We will substitute this  $x^{(0)}$ , it is not the final solution; because it is an arbitrary trial solution. So, you substitute this  $x^{(0)}$  into the main matrix equation except the diagonal terms, and evaluate the values of the updated diagonal terms or updated vectors using the diagonal terms only; it is like that you obtain the updated  $x^{(1)}$  by substituting  $x^{(0)}$  in the equation that  $x_i^{(1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(0)}}{a_{ii}}$ .

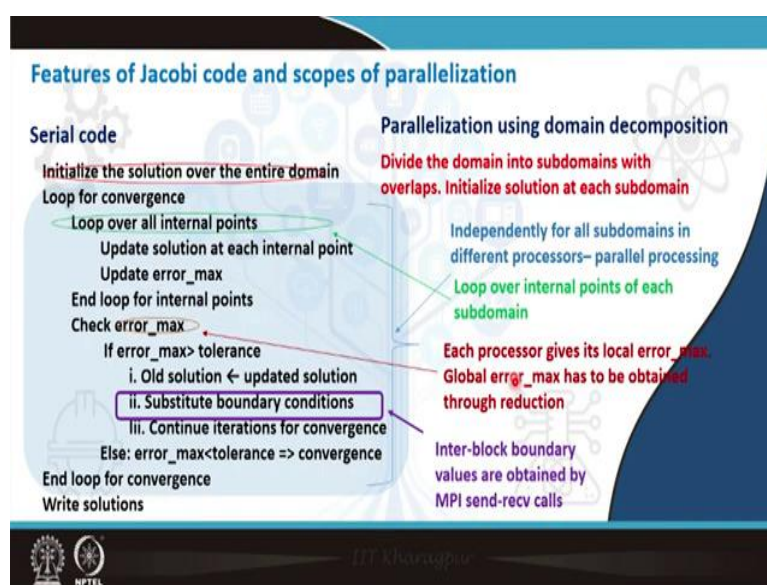
This  $x^{(1)}$  is again probably not the right solution; because it is calculated in after only one iteration based on the guess values. So, you have to calculate  $x^{(2)}, x^{(3)}, x^{(4)}$  so on and continue till you get  $x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}}{a_{ii}}, |x_i^{(k+1)} - x_i^{(k)}| < \varepsilon$ , where  $\varepsilon$  is a small number.

These operations had to be done for all rows of the matrix. So, one thing we can identify here is that, in case we have a large matrix, we have to do it for all the rows of the matrix; we have to do it for a large number of terms. If we are doing it for a million by million matrix, in each iteration, we have to do these calculations for million rows.

Again, for each row, we have to do a number of calculations; the calculations in each row is again of the order of the size of the matrix, because all the columns of the matrix will be multiplied with the RHS vector here. So, if we have a matrix of n rows, in each iteration there are calculations for n rows and in each row, there are at least n operations.

So, it takes n square of operations in one iteration. We have seen that the number of iterations are usually very large, much larger than the dimension of the matrix, so the net computational effort is quite high here. Therefore, this is a good problem which we can parallelize; because the total calculation is high. We can parallelize it and even considering the parallel overheads, we can get good performance. If the problem is not that complex, there would have been less benefit in parallelizing it.

(Refer Slide Time: 09:44)



So, we will look into the features of the serial Jacobi code and identify the scopes which can be parallelized inside the serial code.

First is to initialize the solution over the entire domain, because you work with a guess solution; you use a guess solution for the entire domain.

Loop for convergence: You have to do a number of iterations and similar operations will be done in each iteration. So, there will be one loop for the iterations. Within an iteration loop you have to loop over all the internal points and update the solution at each internal point. While updating the solution at each internal point, again there is a loop over all the elements of the row for one internal point. Then you find out the error max and end the loop for the internal points. After this you get a new updated  $x$  vector.

Check the error max: If the error max is greater than tolerance; that means if the solution is not converged, overwrite the old solution by the updated solution, because your new guess will be what is the updated solution. Then substitute the boundary condition and continue the iterations for convergence. When you see that the error max is less than tolerance, then there is convergence and you end the iteration loops and write a solution.

Now, if we see, where can we parallelize it using domain decomposition? First is that you are initializing the solution over the entire domain; if you are doing domain decomposition, you have broken down the main domain into multiple subdomains. So, you have to initialize a solution for each of the subdomains; because this sub domain will be independently solved in one particular CPU.

So, it is an independent separate problem in each sub domain; therefore, initialization will be required for each subdomain separately. Then this iteration loop had to be run for each subdomain; that means that iteration loops will be in parallelly running in different computers concurrently.

So, this is a parallel processing, over each subdomain one iteration loop is running. All the CPUs involved which are responsible for one subdomain are running one set of iteration loops. However, we have to loop over the internal points and these internal points are the internal points of the sub domain. So, it is not the entire set of points in the geometry over which one particular CPUs operations will loop; but it will loop during one iteration in one particular CPU only over the points which are assigned to that particular sub domain in which this CPU is

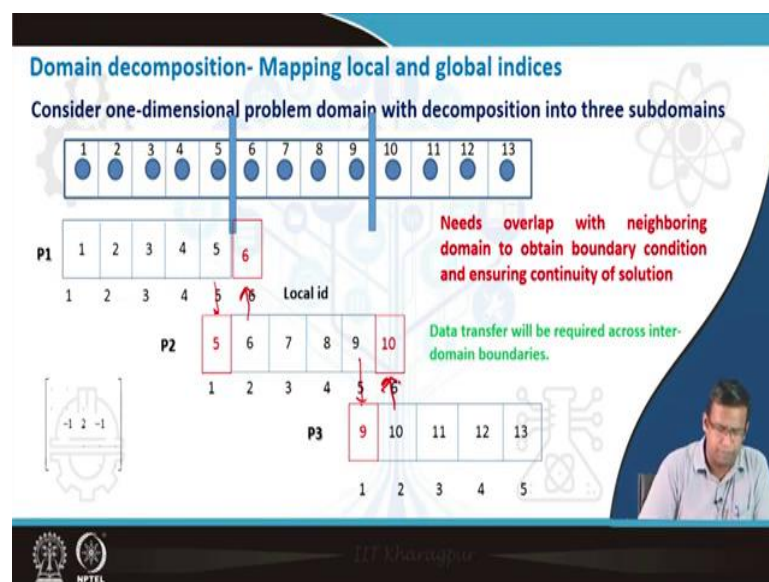
mapped. Then each processor will give a maximum error, because each processor is solving, each CPU is solving an independent problem or independent matrix vector solution problem, so, each CPU will come up with its own local error.

Now, you have to consider all these local errors and can get the global error using reduction operation. We have seen about MPI reduction operation, all the CPUs running in parallel will give their error values and using a maximum MPI max or maximum reduction, we will find out what is the maximum error here.

When substituting the boundary conditions, we also have to look for the inter block boundary values; because if we go to one sub domain, it has four boundaries as we are solving a Cartesian geometry Laplace equation. In the top and bottom boundary, if we can remember physical boundary conditions are there; in the left and right boundary, they are basically inter block partitions which are being shared with the neighbouring block.

So, boundary conditions will come from the updated value of the neighbouring block; therefore, using MPI send receive boundary conditions are to be substituted here.

(Refer Slide Time: 14:17)



We will see about the domain decompositions over a 1 D problem. This particular geometry can be projected and we can think of a 2 D problem. But what we are interested in is that, if we do domain decomposition in only x direction, what will be the steps? And we will also look into the load balancing steps, considering the similar problem later.

There are 13 cells and we use a domain decomposition. Domain decomposition will ensure that almost an equal number of cells will go to each CPU or there will be a load balancing. So, 13 can go almost equal; if the first one will get five cells, the next will get four cells and these will be the inter block boundary, inter domain boundaries.

Processor 1 or CPU 1 will get these five points; processor 2 will get 6 to 9 and processor 3 will get 10 to 13 points. It is important to identify the local indices also; these are the global index; because when one processor will run the job, it will run over the local identifiers only.

Because we are talking about distributed memory system; so, each processor has a different view of memory and each processor can only look into its local memory. So, what are the local values that are important while writing the program, and the other thing is that, once we distribute this into multiple domains in order to ensure continuity, we need overlap with the neighbouring domain. Why? Because if you remember 1 D Laplace equation or simply  $\frac{d^2T}{dx^2} = 0$ ; when we discretize it, we get the points  $-T_{i-1} + 2T_i - T_{i+1} = 0$ . So, in the matrix one particular row equation has three points  $i-1$ ,  $i$  and  $i+1$ . If we have to write an equation for this particular point, we need a point in the right-hand side also which will give us  $i+1$ , or we need this value to be mapped here and that is the requirement of inter block data transfer here to give it the right boundary condition.

When we look into the matrix which is coming out of this particular subdomain; we will consider the overlap also which is acting as the boundary condition here. So, the overlaps are given like that, this 6 will be mapped here it is an overlap here; 5 will be mapped here it is an overlap here, similarly 10 will be mapped here and similarly 9 will be mapped here as an overlap; this will ensure the continuity of the solution.

I told you that local ids are important. So, these first 6 will be the local ids here; then here we include these two overlaps, there are 4 cells only. But now we consider two overlapping cells in both sides; so, there are a total 6 points and 1 to 6 are the local ids here.

Here there are initially five cells, only one cell is coming from the right-hand side processor and one cell overlap. This is only one cell overlap; because this is already a known boundary or a physical boundary. However, for the P2 both the boundaries are inter domain boundaries, therefore we need overlap in both sides. So, this will be the local ids. While running the



program, we will run the loops over the local ids only and data transfer will be required across inter domain boundaries.

How? The data from 5 will come and sit here, similarly this will act as a boundary condition for this particular domain; data from 6 will go and sit here, this will act as the boundary condition for this particular domain. Similarly, data from this 10 will go and sit in the next processor. So, this data will come and act as its boundary condition, this data will go and act as the boundary condition for the P 1.

Similarly, the cell 9 of or cell 5 of P2 will come and come in P 3 and act as the boundary condition, as well as this will go and act as the boundary condition. This will be the data transfer required for doing domain decomposition here.

(Refer Slide Time: 19:42)

**Load balancing and mapping**

The objective of load balancing here is to distribute computations near-uniformly to all processors so that latency is minimized. That can be ensured by ensuring that the sub-domains have near equal internal points.

Cartesian domain: **N number of task**

Sub-domains distributed in nproc number of processor

Global indexing for each subdomain

```

n_process=(N)/nproc
n_remaining=(N)-n_process*nproc
if(rank.lt.n_remaining)then
  n_process=n_process+1 -- distribute the remainders
endif

if(n_remaining.eq.0)then
  istsart=1+(N)/nproc*myid
else
  if(myid.lt.n_remaining)istsart=1+(N/nproc+1)*myid
  if(myid.ge.n_remaining)istsart=1+(N/nproc)*myid+n_remaining
endif
iend=istsart+n_process-1
  
```

Each subdomain has n\_process tasks

NPTEL

The next important part will be load balancing. What I told you earlier is that the main domain is such a way that nearly equal numbers of cells go to each sub domain and nearly equal numbers of tasks are going into each of the CPUs. We have to find an algorithm by which we can give nearly equal amounts of work to each of the CPUs. So, the idea of load balancing is that near uniformly the tasks are balanced to all the processors, so that the latency is minimized.

Latency will happen when less tasks are given to one processor; it finishes its own task; while another processor has more tasks; it is not finished and this processor, who has finished his own task is waiting for the other processor to be free and that will give a latency to that

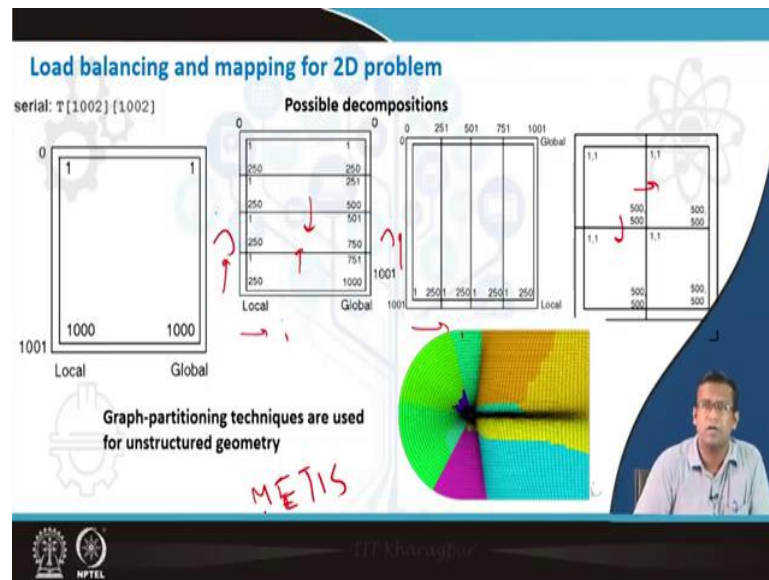
particular processor. This will reduce the parallel performance; so, we want to avoid the latency. To ensure that we need to have near equal number of tasks mapped to each processor.

Now we look into a Cartesian domain. There are  $n$  tasks total and there are  $n_{\text{proc}}$  CPUs and each CPU is taking care of one subdomain. In one subdomain, the number of task if  $n$  is divisible by  $n_{\text{proc}}$ , is  $N/n_{\text{proc}}$ . In case we have 1000 grid points and 10 of sub domains or 10 CPUs, each CPU will get 100 cells. But in case we have 1000 grid points and we have said 33 CPUs; then some CPU will get more, some CPU will get less. However, this more and less has to be done near uniformly and we will show a very simple implementation of that for 1 D domain decomposition. If there are a total  $n$  number of blocks in  $x$  direction and we are doing domain decomposition in  $x$  direction only. If  $N$  is divisible by  $n_{\text{proc}}$ , then each get same amount of work which is  $N/n_{\text{proc}}$ . If it is not, then we have to find out the remainder and the CPUs whose rank is less than remainder; because CPUs rank start from 0, we will add one task only with them. Thus, if the remainder is 3; that means 3 tasks are yet to distribute. So, we will see the CPUS whose rank is less than 3, rank 0, rank 1, rank 2; we will add one task only with them.

So, in case rank is less than the remainder, we add the number of tasks one for them. However,  $n_{\text{process}}$  is the local id. So, for each subdomain the number of tasks are  $n_{\text{process}}$  or the number of grid points in  $x$  direction after which domain decomposition, on which domain decomposition is done is  $n_{\text{process}}$ . The value can be different for different processors, but the variable is  $n_{\text{process}}$ . Then based on this we do global indexing, find out what is  $\text{istart}$  which is the first point we took as 1; in case of  $N \text{ remaining} = 0$  that means, number of tasks is divisible by number of processors, the number of tasks assigned to the previous processors + 1 that is the start id. The end global id is that the start id + total number of processors  $n_{\text{process}} - 1$ ; because we are just indexing it from 1, the initial value is given 1, the last one is  $\text{istart} + n_{\text{process}} - 1$ .

In case there are some remainder, we add the remainder to the processor and this ensures a near equal load balancing.

(Refer Slide Time: 24:06)



The load domain decomposition for a 2 D problem can be done in several ways. We can have domain decomposition in one particular direction. Here we are showing a domain where there are 1002 by 1002 points.

There are four boundaries. So, 0 and 1001 are boundary points, 1 to 1000 are the internal points, and while doing domain decomposition, we have we are working on the matrix containing only the internal points, because the external points are the boundary condition.

We will do domain decomposition only on the internal points here. This domain decomposition can be done in several ways; we can do it in y direction that after 1 to 250 one domain ends, 251 to 500 one domain ends and so on and in each domain, there is the local id 1 to 250, 1 to 250, 1 to 250.

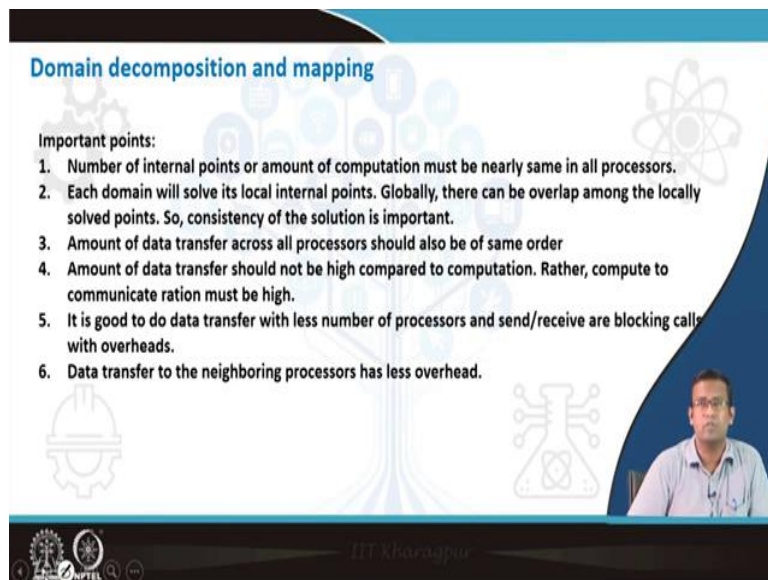
We can do it in the x direction also. If this is my i and this is my j, this is domain decomposition in the y direction or in the j plane. Similarly, I can do it in the x direction also or I can do domain decomposition in both directions using half of x and half of y and can get four blocks.

So, all domain decompositions are possible, but one question might arise : is there one which is good or is there one which is specifically worse than others? In case we have an unstructured mesh, we have to use something like graph partitioning to ensure domain decomposition. One important thing while doing domain decomposition, you need to consider the fact that there will be always data transfer across the inter domain blocks. You have to minimize the inter

domain boundary points; that is one aspect, you have to minimize the data transfer with different processors; if one processor is doing data transfer with ten different processors, there is more communication overhead for it.

Well, so looking into all these things, we have to do a domain decomposition and graph partitioning techniques; there are efficient graph partitioning techniques, there is software named METIS which can be used for unstructured mesh partitioning-based domain decomposition, etcetera.

(Refer Slide Time: 26:54)



**Domain decomposition and mapping**

Important points:

1. Number of internal points or amount of computation must be nearly same in all processors.
2. Each domain will solve its local internal points. Globally, there can be overlap among the locally solved points. So, consistency of the solution is important.
3. Amount of data transfer across all processors should also be of same order
4. Amount of data transfer should not be high compared to computation. Rather, compute to communication ratio must be high.
5. It is good to do data transfer with less number of processors and send/receive are blocking calls with overheads.
6. Data transfer to the neighboring processors has less overhead.

The slide features a blue header with the title 'Domain decomposition and mapping'. Below the title, there is a list of six 'Important points' regarding domain decomposition. The background of the slide is white with faint blue icons of a gear, a network, and a person. In the bottom right corner, there is a small video inset showing a man in a light blue shirt speaking. At the bottom of the slide, there is a dark blue footer with the text 'IIT Kharagpur' and some logos on the left.

The important point in doing domain decomposition and mapping is that we need to have load balancing that is the number of internal points or amount of computation must be nearly the same in all processors, so that latency is minimized. Each domain will solve its local internal points. Globally, there can be overlap among the locally solved points. So, locally each domain is solving the internal points. Globally, even there can be overlap among the locally solved points also. But locally it is not aware of the overlaps; it is solving whatever is told about the internal points among them. However, this overlap is given to them, so that consistency of the solution can be maintained.

So, it is possible that there is an overlap in the sub domain. If one point is being solved by one CPU; the same point can belong to another subdomain and can be solved by the other CPU. So, this is an overlap geometry; locally this overlap will not matter because the sub domain is CPU is solving all the points internal to its sub domain.

Globally, overlap can help in ensuring the consistency of the solution, and if overlap is there, we have to make it sure that if one point belong to two sub domains; two CPUs are computing for that particular point, the value must be same at that point, so that the boundary conditions and data transfers are to be done properly.

Amount of data transfer across all the processors should also be of the same order. If two processors are communicating in between them and another two processors are also communicating in between themselves, the data communication time and the amount of data transfer must be same among themselves; otherwise again there will be latency in data transfer, one set of processors will finish their data transfer, other set of processors are still working on it.

So, along with giving equal numbers of internal points to different sub domains; we also have to ensure that equal amounts of, nearly equal amount of boundary points are assigned to each of the sub domains, so that the data transfer is also of the same order. Amount of data transfer should not be high compared to computation; rather, the compute to communication ratio must be high.

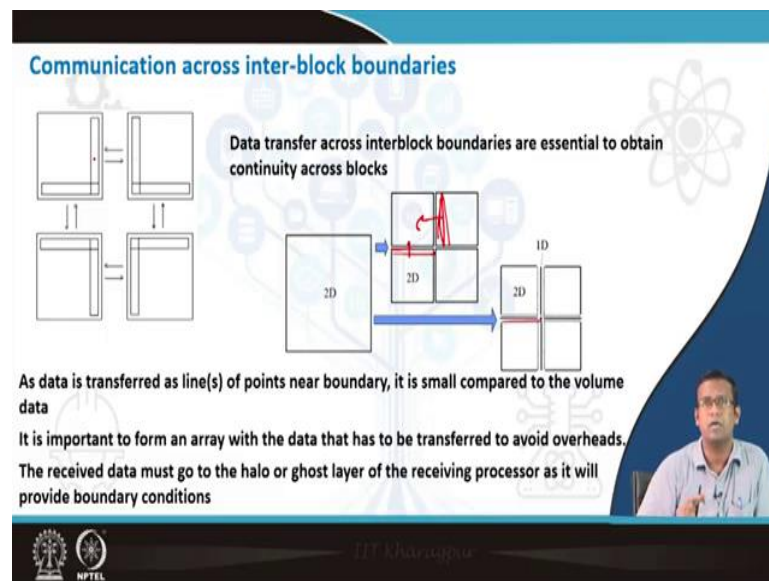
The amount of data transfer should not be high compared to the computation. If it is doing a lot of data transfer, but less computing; then the overheads are too high and the parallel performance will be bad. You have to see the amount of communication compared to the amount of computation, what is that value? And compute the communication ratio that number of computing steps for each data transfer step must be high.

It is good to do data transfer with a smaller number of processors and send, receives are blocking calls with overheads. So, if one CPU is communicating with multiple CPUs; when first communicating with one CPU, the next communication calls are on hold, waiting or in the pipeline. There is a sequentiality of communication steps executed by one particular CPU that will add to over it. If one CPU is communicating with many CPUs that will not be a good option. Rather each CPU communicating with a smaller number of CPUs will be good. It is good that the inter domain boundaries are less for one particular subdomain because the communication overhead will be less in that particular case.

Data transfer to the neighbouring processors has less overhead; if data transfer is done with a far end process, there are multiple hops between them and the communication time will be more, if you remember the part of communication. If data transfer is done to the neighbouring

domains, especially in case of the large hybrid servers, few CPUs are on the same motherboard; while few CPUs are connected to Ethernet switches. So, if data transfer is happening with the neighbouring CPU only, it is within the same motherboard it is faster. But if data transfer is happening with a remote CPU and there can be grid computing, where the CPUs are physically in a different location, then the data transfer time will be more. So, it has to be also checked that the data transfer is with the neighbouring CPUs as much as possible.

(Refer Slide Time: 32:08)



Now, we need to look into the communication across the boundaries. We are discussing data transfer, so data transfer across inter block boundaries is essential to ensure the continuity across the blocks. It is also essential, because while solving the matrix equation; the inter block boundary conditions give the are the boundary conditions for the sub domain, without the boundary condition the matrix will be singular and you cannot solve it.

Hence inter block boundary conditions are essential for solving problems in a sub domain especially when you are doing matrix solvers. To get the right interblock of boundary condition, data transfer steps are essential. So, communication across inter block boundaries is an essential step in domain decomposition; it has to be done multiple times during iteration to ensure continuity of the updated solution.

In case of 2 D geometry when it is broken into multiple sub domains, if we see the data transfer, it is of the points belonging to this particular line, this will go here. Similarly, if we see the data

transfer in between these two domains; it is the points belonging to the boundary line which will come here.

In 2 D data transfer is happening for a line of solution variables; it is not for the entire volume of solution variables, but for a line of solution variables. This is done using MPI send, receives. So, if you remember MPI send, receive command, it sends the address and size of the data.

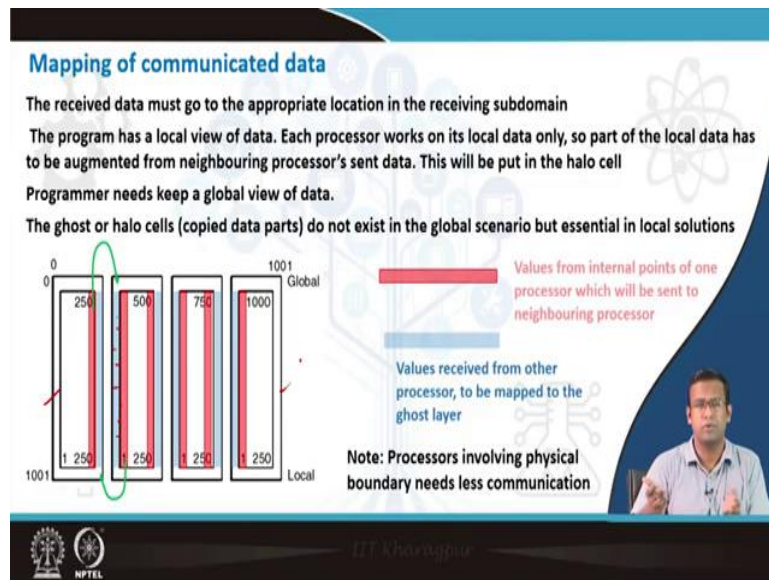
So, when will do the data transfer, one address will come and the data size of the data that will be transferred, that will come. Therefore, it is important that the entire data on that line which has to be transferred, has to be mapped to a single 1D array.

If we can map the data to be transferred into a contiguous array, it is an efficient step. In case of 2 D it is a 1 D vector which will be transferred. In case of 2 D, it is a plane's vector data which will be transferred, so, you have to use an effective mapping and map it into a contiguous array.

As data is transferred as lines or points near the boundary, it is small compared to the volume of data. It is important to form an array of the data and the data has to be transferred to avoid overheads. The data that will be transferred, it is important that if we can put it into a different buffer, make a single array of it and transfer it.

The received data must go to the halo cell; when this data will be transferred, this is going to the halo cell or in the boundary cell which is not an internal point, but coming as the boundary condition for that matrix. This is a halo or ghost layer we can tell; this is not an internal point, but belongs to the sub domain as boundary condition. Therefore, the received data has to be rightly mapped into that halo and ghost layer.

(Refer Slide Time: 35:23)



The received data must go to the appropriate location in the receiving subdomain. The programmer has a local view of data. Each processor works on the local data only, so part of the local data has to be augmented from neighbouring processor's send data, and this will be put in a halo cell.

When we are looking into one subdomain, we are looking into the internal points of the subdomain, which is stored as a local data as a separate matrix for that particular subdomain. When we are getting data from another processor; this data is being received in a received buffer and now it has to be merged with the right local array in which we are doing the solutions. So, this has to be put in the halo cell and efficiently mapped to the local processor's proper data location.

Each processor is looking into having a local view of the data only, only the programmer can have a global view based on which he has done the domain decomposition, again based on that we will collate the data and write the final solution.

The ghost or halo cells do not exist in the global scenario. This is one subdomains part which is sent to another subdomain and received by another subdomain staying there as boundary condition. This is again overlapping data.

In the global scenario there is no overlap, in the global scenario each internal point is unique; in the sense of local domain this overlap or halo or ghost cells exist. But in the global view, it

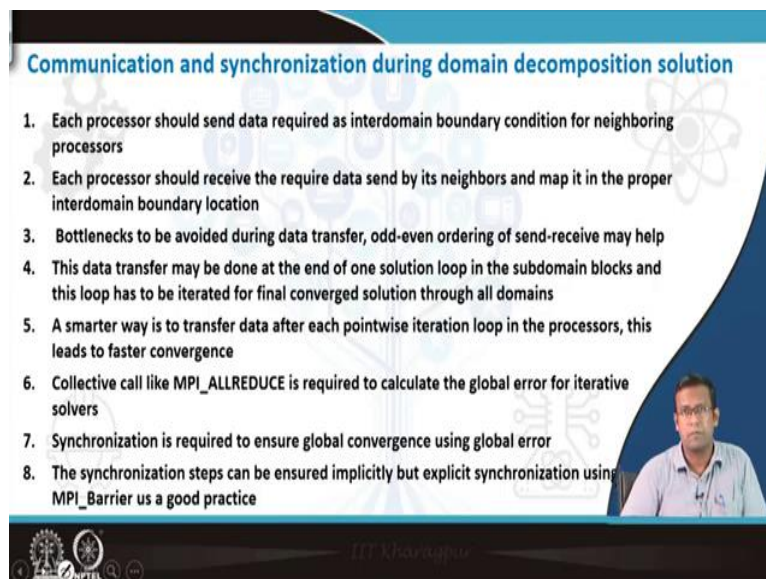


is not there; these are only copying of one data point to another processor's local data point. We can see that; this data is internal to this particular domain; but this will act as the halo layer or boundary condition of the next sub domain. So, this will be sent to the next sub domain and received here and then mapped in this particular location. The green one will take it and use as for the boundary condition. Similarly, this sub domains first cells local data will go to the previous subdomain and see it as the boundary condition. So, values from the internal points of one processor which will be sent to the neighbouring processor.

Greens means, once this is received; this is mapped to the raised ghost level. So, all the neighbouring processors have this red blue, red blue layers arrangement.

The processors involving the physical boundaries require less communication; because if it is a physical boundary you do not need to get data from the neighbour, there is no neighbouring boundary, a neighbouring domain on the physical boundary, it is a physical boundary. So, you do not need to do data transfer for the neighbouring boundary, and it requires less data transfer.

(Refer Slide Time: 38:42)



**Communication and synchronization during domain decomposition solution**

1. Each processor should send data required as interdomain boundary condition for neighboring processors
2. Each processor should receive the require data send by its neighbors and map it in the proper interdomain boundary location
3. Bottlenecks to be avoided during data transfer, odd-even ordering of send-receive may help
4. This data transfer may be done at the end of one solution loop in the subdomain blocks and this loop has to be iterated for final converged solution through all domains
5. A smarter way is to transfer data after each pointwise iteration loop in the processors, this leads to faster convergence
6. Collective call like MPI\_ALLREDUCE is required to calculate the global error for iterative solvers
7. Synchronization is required to ensure global convergence using global error
8. The synchronization steps can be ensured implicitly but explicit synchronization using MPI\_Barrier us a good practice

The slide includes a video inset of a man speaking in the bottom right corner. At the bottom, there are logos for IIT Khargpur and NPTEL, and the text 'IIT Khargpur'.

Each processor should send data required as inter domain boundary condition for the neighbouring processors. How does the neighbouring processor get data from the other processor? It has to be through a send process. So, each processor has to execute the send command to the neighbouring processor.

Each processor should receive the required data sent by the neighbours and map it in the proper inter domain boundary condition. So, each processor has to execute a send command and the receiving processor also has to execute a receive command. This command has to be written by the programmer for the processors in the relevant location and the mapping also has to be ensured by the programmer.

There can be a bottleneck, when there is a boundary between two processors, two processors are sharing a common boundary both the processors are sending data to them and both are receiving that can create a bottleneck. So, odd even ordering type of things has to be done, so that when one is sending other will receive; when the next one will send, the previous one will receive.

The data transfer may be done at the end of one solution loop in the sub domain blocks and the loop has to be iterated for the final convergence through all sub domains. There can be two ways of doing data transfer; one is that you solve the local problem, solve the local matrix equation, then do the data transfer, again do the local matrix equation, again do the data transfer. But a smarter way will be that during iterations, you keep on doing data transfer; run once set of iteration over all the points; do not look for the local convergence. Do the data transfer and go for the next iteration loop, and after one iteration loop, you find out whether global convergence has been achieved.

So, always look for global convergence, do not look for local convergence and in one sweep of iterations over all the internal points do a data transfer that ensures faster solution, because the data is more quickly updated. You need to calculate global errors, like MPI allreduce is required to calculate the global error for iterative solvers.

Synchronization is required to ensure global convergence using global error.

The synchronization steps can be ensured implicitly, even for our reduction operation etcetera there are synchronization steps; but it is a good practice to explicitly specify the synchronization step using MPI barrier, because the overhead is less, but from programming perspective it is easy to debug and to maintain the proper ordering of steps. It's important that you use an explicit synchronization step like MPI\_barrier.

Well, so we discussed a lot about the essential steps of doing domain decomposition. The next class we will take up a Jacobi solver and look into the detailed steps in the Jacobi solver and identify where we can do domain decomposition over it.