High Performance Computing for Scientists and Engineers

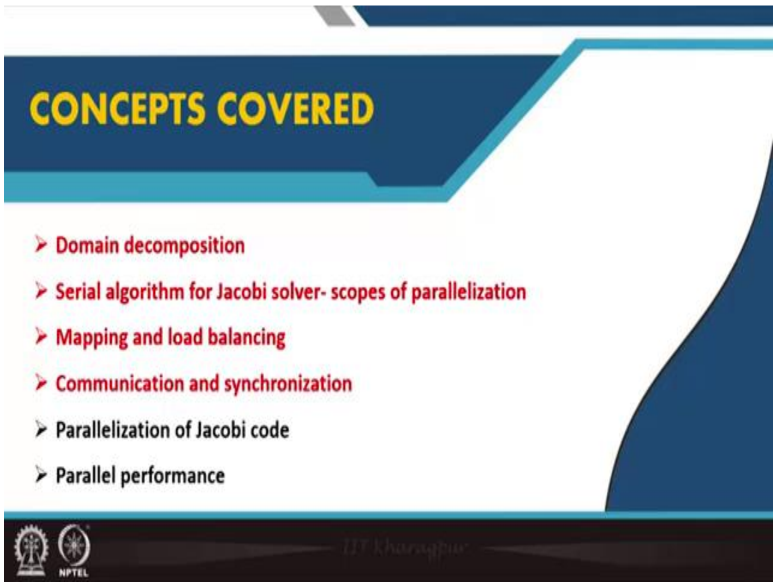
Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 03
Distributed Computing using MPI
Lecture – 28
Domain Decomposition based parallelization of matrix solvers (continued)

Hello everybody, we are discussing Domain Decomposition based parallelization of Jacobi solver in the course High Performance Computing for Scientists and Engineers. In the previous two classes we have looked into domain basic ideas of implementing domain decomposition for Jacobi solver and then we took a sequential code for Jacobi solver or a single processor Jacobi solver C code.

We identified the steps, where we can put right constructs for doing domain decomposition and developed a template for the parallel Jacobi solver. In this particular lecture we will try to develop our first domain decomposition based parallel solver which is solving Laplace equation using Jacobi solver.

(Refer Slide Time: 01:14)

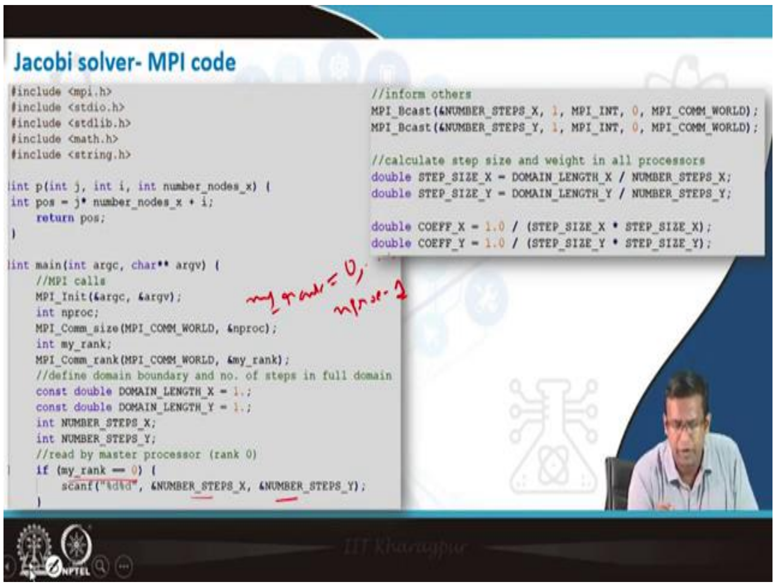


CONCEPTS COVERED

- Domain decomposition
- Serial algorithm for Jacobi solver- scopes of parallelization
- Mapping and load balancing
- Communication and synchronization
- Parallelization of Jacobi code
- Parallel performance

So, we will look into the parallelization of the Jacobi code here, and then we will also see the performance of the parallel code that we have developed here.

(Refer Slide Time: 01:24)



Jacobi solver- MPI code

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

int p(int j, int i, int number_nodes_x) {
int pos = j* number_nodes_x + i;
    return pos;
}

int main(int argc, char** argv) {
    //MPI calls
    MPI_Init(&argc, &argv);
    int nproc;
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    //define domain boundary and no. of steps in full domain
    const double DOMAIN_LENGTH_X = 1.;
    const double DOMAIN_LENGTH_Y = 1.;
    int NUMBER_STEPS_X;
    int NUMBER_STEPS_Y;
    //read by master processor (rank 0)
    if (my_rank == 0) {
        scanf("%d%d", &NUMBER_STEPS_X, &NUMBER_STEPS_Y);
    }
```

my_rank = 0, nproc = 2

```
//inform others
MPI_Bcast(&NUMBER_STEPS_X, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&NUMBER_STEPS_Y, 1, MPI_INT, 0, MPI_COMM_WORLD);

//calculate step size and weight in all processors
double STEP_SIZE_X = DOMAIN_LENGTH_X / NUMBER_STEPS_X;
double STEP_SIZE_Y = DOMAIN_LENGTH_Y / NUMBER_STEPS_Y;

double COEFF_X = 1.0 / (STEP_SIZE_X * STEP_SIZE_X);
double COEFF_Y = 1.0 / (STEP_SIZE_Y * STEP_SIZE_Y);
```

We are solving a Laplacian equation in a Cartesian geometry in matrix form. The matrix has been obtained using a finite difference method and we are using Jacobi iterative methods for solving that.

Now, if you observe you can find that we have considered a Cartesian geometry which is the simplest geometry in the 2D plane, Cartesian square geometry. We have considered Laplace equation which is again the simplest second order equation in partial differential equations and we are looking into Jacobi solver which is again one of the most simple forms of the basic iterative solvers.

So, you try to keep everything simple, but the same methodology which we are discussing here, domain decomposition based parallelization for problems involving matrix equation, over spatial domains; the same methodology can be applied for complex problems for non-Cartesian geometry, for non-square complex geometries, for problems involving higher order differential equations, more complex even nonlinear differential equation,where some linearization has been applied.Say for example, Navier–Stokes equation use the same method and also more efficient solvers like conjugate gradient bi CG we can use the same method.

Actually for those types of problems you do high performance computing, you do not do high performance computing for very simple problems where you can easily get analytical solutions. You do high performance computing for much complex problems, but in the similar methodology you can parallelize the solver required to solve that particular complex problem in complex geometry using domain decomposition method.So, if you understand the basic steps for parallelization of Jacobi solver in Cartesian square geometry for Laplace equation which we are discussing here you can take it and apply it for complex problems, well.

We start with the Jacobi solver MPI code. The first thing that comes here is that you have to include a MPI header file mpi.h. ,for MPI programming.

The indices for finite difference comes in i, j; from i, j it has to be converted into a 1D vector for writing a matrix equation. So, this is the pointer function which gives the ID vector from the i,j index ok.

So, in the main function first there will be MPI calls.What are the MPI calls? MPI_init (MPI has to be initialized), MPI_comm_size that will tell me that, what is the total number of processors working here. MPI_comm_rank ,what is the rank of each processor . So, myrank will start from 0 and end up to nproc - 1. nproc is the total number of processors.

For different processors there will be different myrank; for the first processor myrank is 0, for the last processor myrank is nproc - 1. The domain length in both and X and Y is 1, the number of steps in X and Y that will be obtained from the user.

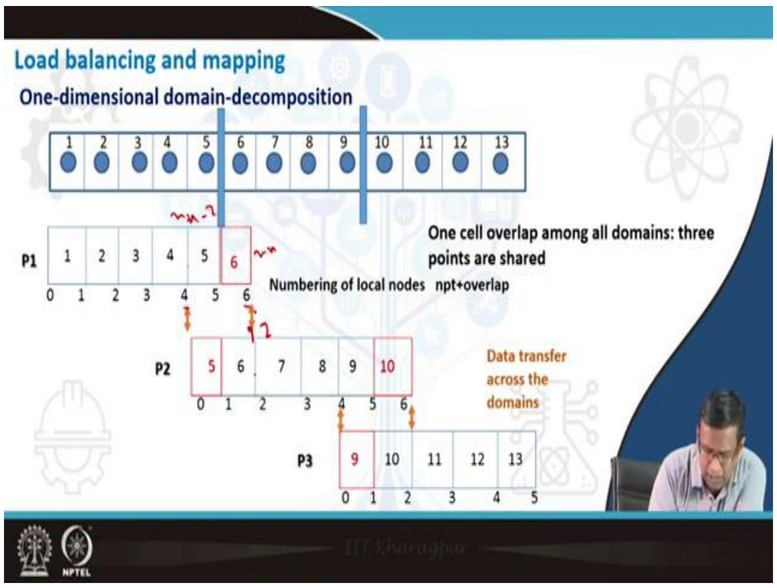
So, the master processor whose myrank is 0, will read it from the user that what is the number of steps in X and number of steps in Y. We kept it variable because we want to do certain tests, we want to change the problem size, we want to change the number of points and see how the solutions are coming, how much time it is taking. As many steps as will take in X and Y direction Δx and Δy will be small and will get a more accurate solution.Then the number of steps in X and Y which has been read by the master processor this number will be informed to others by Bcast. MPI_Bcast number of steps is an integer of size 1, it has been read by processor myrank = 0. So, he knows this the value of the number of steps in X and he will send it to the other processors using broadcast similarly num steps in Y will be sent to the other processors.Each processor will calculate the step size which is domain length by number of steps in X, domain length by number of steps in Y and the coefficients. So, there can be one question that, this is the information which is obtained by the zeroth processor it is sharing with the other processors.

Why are all the processors doing this calculation? This particular calculation is being done by all the processors. It could have been done by a master processor or rank 0 processor who already knows these numbers and then he can broadcast it to all the other processors.

Answer is broadcast is a communication operation, so it is costlier than calculation, and if myrank 0 is doing the calculation , then the other processor would have been sitting ideal there.

So, you want to reduce communication also we want to see that all the professors are doing nearly the same work. Therefore, the essentially sequential part is reading from the processor which will be done by one processor and sent to the others, but this will be done by all the processors.

(Refer Slide Time: 07:59)



Load balancing and mapping

One-dimensional domain-decomposition

One cell overlap among all domains: three points are shared

Data transfer across the domains

Now, we have to do load balancing and we have to map it to the right processors. This problem we have seen earlier, we consider a 1D domain decomposition that there are 13 steps. We have calculated in how many steps the problem is broken. So, in X direction there are 13 steps and we use three processors.

So, the first processor will get five, the next processor will get four, the next processor will get four steps. It is a domain decomposed distribution,and these are the steps in processor 1, processor 2, processor 3. Now, the overlaps will be there in order to give the right boundary condition.There will be overlap of 6 with processor 1; the red ones are the overlap.Wwe have discussed overlaps and load balancing in previous lectures.

Now, we are allowing one cell overlap among all the domains. So, how many points are shared across the domains? If we see processor 1 and processor 2, 6 is in processor 2 which is a hello or ghost layer in processor 1; 5 in processor 1 is an internal step in processor 1 which is a hello layer in processor 2.

What are the points? We are solving for the grid points, these are the cells. Grid points are the points in between the cells points connecting the cells. So, if we renumber the points this is 0 to 6 are the number of points here and for each point we get a value.

Now, here these are the number of points. So, what are the points shared here? If two cells are common then 2 + 1=3 points are shared. So, 4, 5, 6 of processor 1 and 0, 1, 2 of processor 2 are the same points.

Now, processor 1 will solve for the internal points for up to point 5; point 6 will be the boundary condition which will come from processor 2. Similarly, processor 2 will solve from the first point which is its internal point, point 0 is a boundary condition which will come from processor 1.

While renumbering the number of points each processor will get the total number of points plus how many overlaps are there, say here the total number of local points are 5 there are 6 and the 1 overlap is there. Here, the total number of local points is 5 and then there are two neighboring boundaries. So, 2 overlaps are there. So, 0 and 6 ,7 points come .

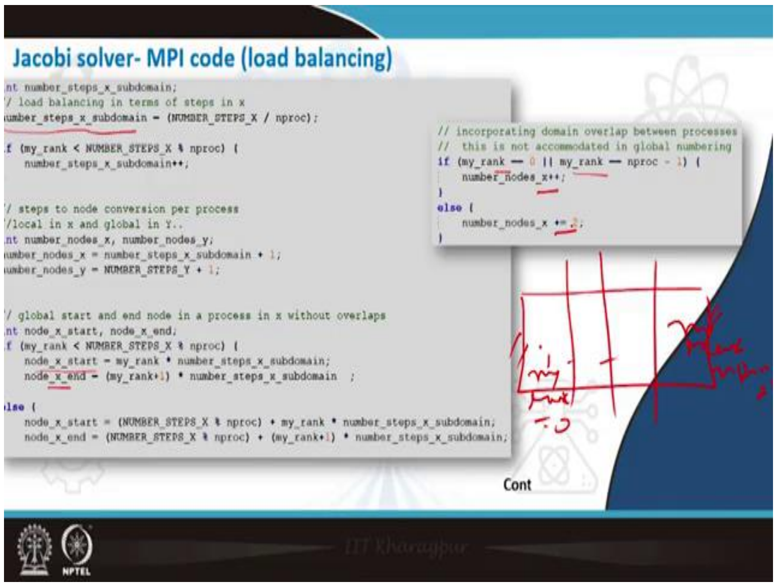
So, if there is overlap from both the sides ,if it has boundary inter domain boundary from both the sides two extra points are added, these two will be the boundary conditions. If there is overlap from only one side, the other side is the physical boundary one point is added, any two contiguous domains share three points in between them.

While solving the domains it will solve up to point 5, this processor will solve from point 1 up to point 5. So, this particular point will be solved by both the processors. By data transfer we will ensure that the solution is the same in both the processor,so the consistency is achieved.

Point 4’s data from processor 1 will come and sit in processor 2s 0 point process. So, this will send the data processor 2 will receive the data. Point 2’s data from processor 2 will go and sit it in point 6 of processor 1; this will send the data, this will receive the data. So, there will be a both way data transfer in these points in between the processors.

So, we have to identify the domain boundaries, we have to add overlaps while calculating the local number of points, because local number of points = number of cells or number of divisions plus 1 and we have to add one point in each for each of the overlaps and we have to do a data transfer. Where will the data go? From the internal point’s last but one points data will come to the next right processor's 0th location.For the next processor it is second points data will go and sit in the last point of the previous or the left processor. These are simple things, you can look and observe how this data transfer is being happen these we are going to implement in our program.

(Refer Slide Time: 13:49)



Jacobi solver- MPI code (load balancing)

```
nt number_steps_x_subdomain;
/ load balancing in terms of steps in x
umber_steps_x_subdomain = (NUMBER_STEPS_X / nproc);

f (my_rank < NUMBER_STEPS_X % nproc) {
    number_steps_x_subdomain++;


/ steps to node conversion per process
/local in x and global in Y..
nt number_nodes_x, number_nodes_y;
umber_nodes_x = number_steps_x_subdomain + 1;
umber_nodes_y = NUMBER_STEPS_Y + 1;


/ global start and end node in a process in x without overlaps
nt node_x_start, node_x_end;
f (my_rank < NUMBER_STEPS_X % nproc) {
    node_x_start = my_rank * number_steps_x_subdomain;
    node_x_end = (my_rank+1) * number_steps_x_subdomain  ;

lse {
    node_x_start = (NUMBER_STEPS_X % nproc) + my_rank * number_steps_x_subdomain;
    node_x_end = (NUMBER_STEPS_X % nproc) + (my_rank+1) * number_steps_x_subdomain;
```

```
// incorporating domain overlap between processes
//  this is not accommodated in global numbering
if (my_rank == 0 || my_rank == nproc - 1) {
    number_nodes_x++;
}
else {
    number_nodes_x += 2;
}
```

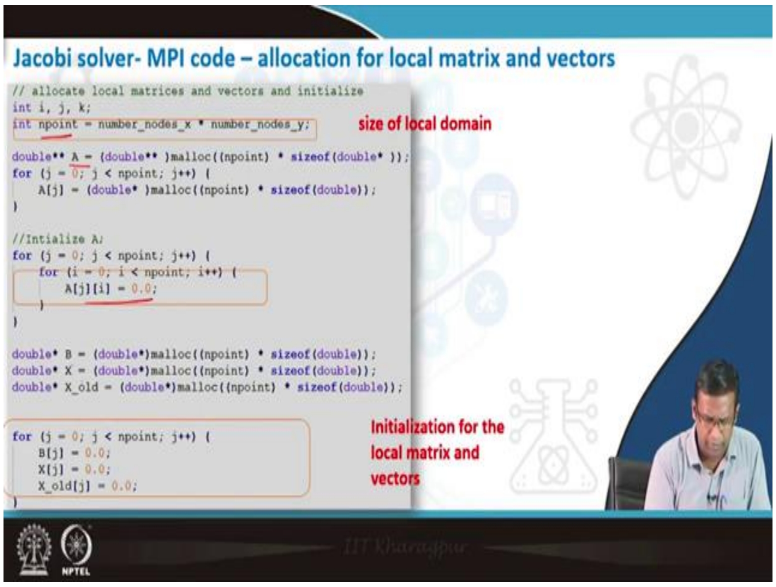
Cont

Number of steps in the sub domain is the number of steps divided by nproc, if it is not divisible by nproc, then will get the remainder and we add the remainder to the processors whose rank is less than the remainder. So, this is the load balancing step; some processors get one more than the other processors. However, for each processor number of steps in x is number_steps_x_subdomain.

Then, the number of nodes in x = number_steps_x_sub domain + 1, these are the internal points without considering the hello layer and the number of nodes in y is number_steps_y + 1. Based on the internal points we give the global ID of the starting point and the global ID of the end point. These are for the processor whose rank is less than the remainder, for the processor whose rank is greater than the remainder the previous processor’s start and end are added.

If this is my domain, if there are multiple subdomains this is myrank 0 , and this is myrank nproc - 1. This is the first processor and last processor, they have physical boundaries. So, this processor has only one inter domain boundary. So,there is only one overlap, the number of nodes is added by 1. The other processor has two inter domain boundaries therefore, the number of nodes in that processor are added by 2.

(Refer Slide Time: 15:52)



Jacobi solver- MPI code – allocation for local matrix and vectors

```
// allocate local matrices and vectors and initialize
int i, j, k;
int npoint = number_nodes_x * number_nodes_y;

double** A = (double** )malloc((npoint) * sizeof(double* ));
for (j = 0; j < npoint; j++) {
    A[j] = (double* )malloc((npoint) * sizeof(double));
}

//Intialize A;
for (j = 0; j < npoint; j++) {
    for (i = 0; i < npoint; i++) {
        A[j][i] = 0.0;
    }
}

double* B = (double*)malloc((npoint) * sizeof(double));
double* X = (double*)malloc((npoint) * sizeof(double));
double* X_old = (double*)malloc((npoint) * sizeof(double));

for (j = 0; j < npoint; j++) {
    B[j] = 0.0;
    X[j] = 0.0;
    X_old[j] = 0.0;
}
```

size of local domain

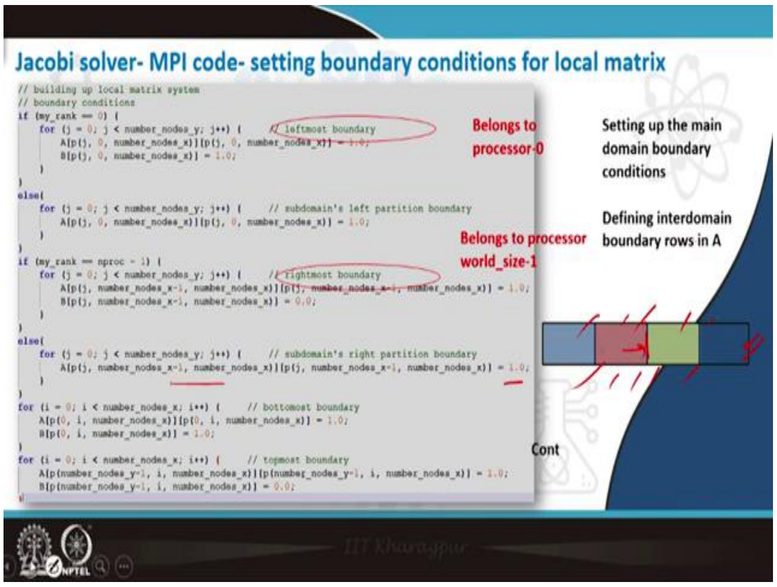
Initialization for the local matrix and vectors

Now, we know that number of nodes in x. So, what we calculate here is the number of nodes in x based on the number of nodes steps in that particular subdomain. Number of nodes in x into the number of nodes in y gives us the end point or total number of points in one particular subdomain.

Now, this endpoint includes the points in the halo layer or the inter domain ghost layers with the data which is coming from the next sub domain being used for the boundary condition to the internal points of this sub domain. Based on this we are to put everything in the matrix including the boundary conditions and only solve the matrix equation.

Based on that we allocate the coefficient matrix A based on this npoint which is for the domain decomposed nodes. We initialize everything to be 0, we allocate B, X X old and initialize everything to 0. This is the size of the local domain, the total number of points in the local domain is num_node_x and this can vary because load balancing ensures that nearly equal amounts of points are there in each domain, but there can be little difference in the number of points. So, this number npoint can be different for the different processors. These are done for the local matrices.

(Refer Slide Time: 17:21)



Jacobi solver- MPI code- setting boundary conditions for local matrix

```
// building up local matrix system
// boundary conditions
if (my_rank == 0) {
    for (j = 0; j < number_nodes_y; j++) {     // leftmost boundary
        A[p(j, 0, number_nodes_x)][p(j, 0, number_nodes_x)] = 1.0;
        B[p(j, 0, number_nodes_x)] = 1.0;
    }
}
else{
    for (j = 0; j < number_nodes_y; j++) {     // subdomain's left partition boundary
        A[p(j, 0, number_nodes_x)][p(j, 0, number_nodes_x)] = 1.0;
    }
}
if (my_rank == nproc - 1) {
    for (j = 0; j < number_nodes_y; j++) {     // rightmost boundary
        A[p(j, number_nodes_x-1, number_nodes_x)][p(j, number_nodes_x-1, number_nodes_x)] = 1.0;
        B[p(j, number_nodes_x-1, number_nodes_x)] = 0.0;
    }
}
else{
    for (j = 0; j < number_nodes_y; j++) {     // subdomain's right partition boundary
        A[p(j, number_nodes_x-1, number_nodes_x)][p(j, number_nodes_x-1, number_nodes_x)] = 1.0;
    }
}
for (i = 0; i < number_nodes_x; i++) {     // bottomost boundary
    A[p(0, i, number_nodes_x)][p(0, i, number_nodes_x)] = 1.0;
    B[p(0, i, number_nodes_x)] = 1.0;
}
for (i = 0; i < number_nodes_x; i++) {     // topmost boundary
    A[p(number_nodes_y-1, i, number_nodes_x)][p(number_nodes_y-1, i, number_nodes_x)] = 1.0;
    B[p(number_nodes_y-1, i, number_nodes_x)] = 0.0;
```

Belongs to processor-0
Belongs to processor world_size-1
Setting up the main domain boundary conditions
Defining interdomain boundary rows in A
Cont
IIT Kharagpur
NPTEL

Now, we have to give the boundary conditions. So, for the physical boundaries boundary conditions will directly come, for the inter domain boundaries boundary condition will come from the data transfer from the next sub domain.

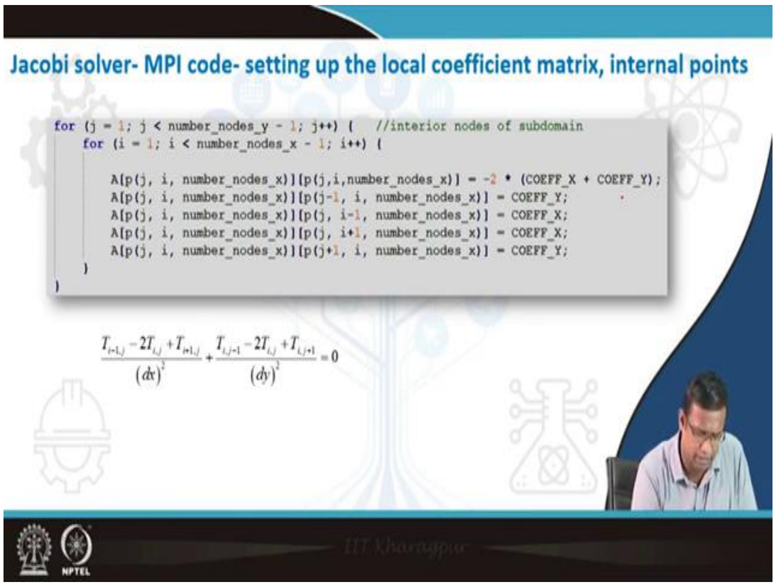
So, when setting the boundary conditions for the physical boundaries we put the boundary conditions in the B vector for the inter domain boundaries, we will not put the boundary conditions right now in the B vector, we will put them after data transfer. For myrank = 0, this is rank 0, this has rank nproc - 1.

For myrank = 0,on the left most boundary we apply the right boundary condition. If myrank is not 0 for the leftmost boundary condition we only make in the A matrix diagonal 1, off diagonal 0. We do not do anything for the right hand side vector. If myrank is nproc - 1, the rightmost boundary we give the right boundary condition here for this boundary also.

In the other case if myrank is nonzero for the rightmost boundary that is this boundary of this domain, we only would make the diagonal of A = 1, do not do anything with the B vector. For the bottom most and topmost boundary conditions these are physical boundaries for all the domains,so, we apply the boundary conditions accordingly.

This belongs to processor 0 leftmost physical boundary ,rightmost physical boundary belongs to processor nproc - 1 and we will apply the boundary conditions rightly. So, we set up the main domain boundary conditions and also we define the inter domain boundary rows in A that are diagonal 1, non-diagonal 0 for the inter domain points.

(Refer Slide Time: 19:49)



Jacobi solver- MPI code- setting up the local coefficient matrix, internal points

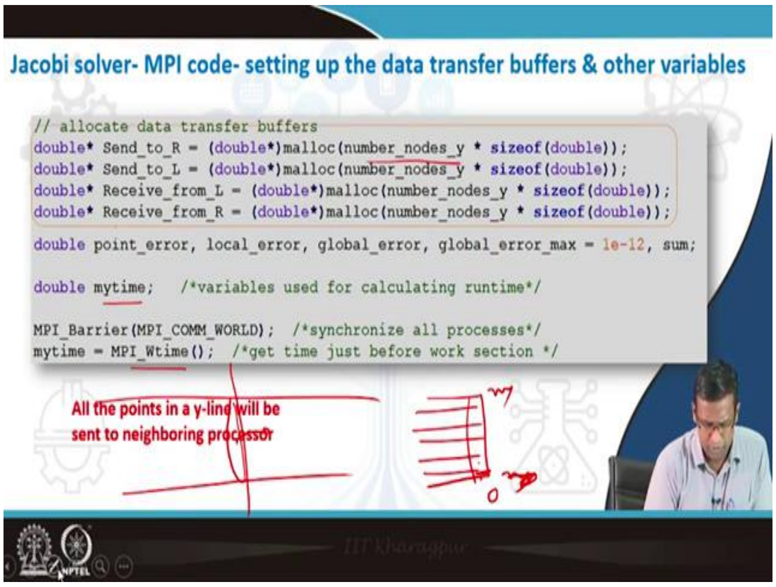
```
for (j = 1; j < number_nodes_y - 1; j++) {   //interior nodes of subdomain
    for (i = 1; i < number_nodes_x - 1; i++) {

        A[p(j, i, number_nodes_x)][p(j,i,number_nodes_x)] = -2 * (COEFF_X + COEFF_Y);
        A[p(j, i, number_nodes_x)][p(j-1, i, number_nodes_x)] = COEFF_Y;
        A[p(j, i, number_nodes_x)][p(j, i-1, number_nodes_x)] = COEFF_X;
        A[p(j, i, number_nodes_x)][p(j, i+1, number_nodes_x)] = COEFF_X;
        A[p(j, i, number_nodes_x)][p(j+1, i, number_nodes_x)] = COEFF_Y;
    }
}
```

$$\frac{T_{i-1,j}-2T_{i,j}+T_{i+1,j}}{(dx)^2}+\frac{T_{i,j-1}-2T_{i,j}+T_{i,j+1}}{(dy)^2}=0$$

Now, for the interior nodes we define the matrix there I give the coefficients, and, this comes from the equation finite difference equation, $\frac{T_{i-1,j}-2T_{i,j}+T_{i+1,j}}{(dx)^2}+\frac{T_{i,j-1}-2T_{i,j}+T_{i,j+1}}{(dy)^2}=0$.

(Refer Slide Time: 20:06)



Jacobi solver- MPI code- setting up the data transfer buffers & other variables

```
// allocate data transfer buffers
double* Send_to_R = (double*)malloc(number_nodes_y * sizeof(double));
double* Send_to_L = (double*)malloc(number_nodes_y * sizeof(double));
double* Receive_from_L = (double*)malloc(number_nodes_y * sizeof(double));
double* Receive_from_R = (double*)malloc(number_nodes_y * sizeof(double));

double point_error, local_error, global_error, global_error_max = 1e-12, sum;

double mytime;   /*variables used for calculating runtime*/

MPI_Barrier(MPI_COMM_WORLD);  /*synchronize all processes*/
mytime = MPI_Wtime();  /*get time just before work section */
```

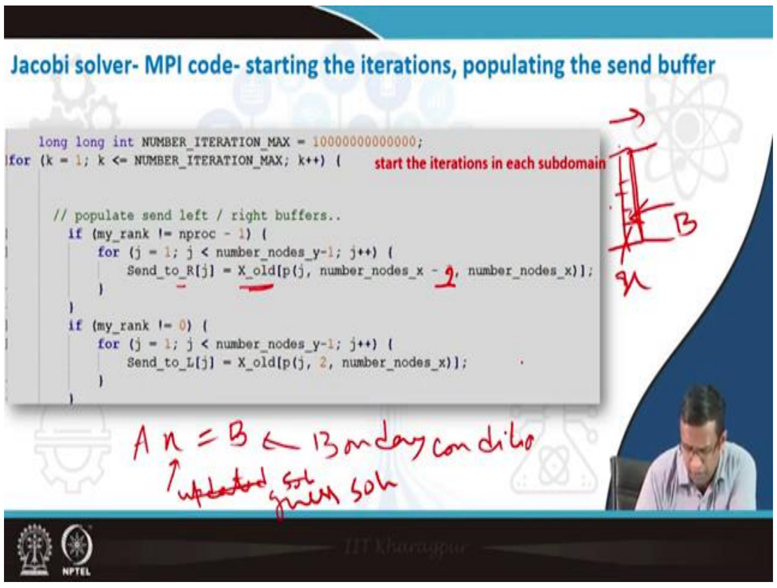
All the points in a y-line will be sent to neighboring processor

Now, the data transfer buffers are to be allocated. We have discussed earlier that if this is my inter-domain boundary this data will go to the next domain. What is this data? If we see one particular domain there are n0 to n y points in this particular domain. So, this ny data will go to the next domain; physical boundary data may not we may not need to send it.

However, the total amount of points in one y plane is num_nodes_ y. So, we allocate the buffers using num_nodes_y. Also, to define the global errors, we also define a variable mytime which will be used to calculate the time and set MPI_Barrier to synchronize all the processors. Using MPI_Wtime we calculate what is the time instant in that program.

We will calculate it again and subtract it between them, and then we will find out how much time has been taken for the solver during this MPI call. So, you calculate MPI time here. All the points in y-line will be sent to the neighboring processor. So, we allocate everything by num_nodes_y.

(Refer Slide Time: 21:38)



Jacobi solver- MPI code- starting the iterations, populating the send buffer

```
    long long int NUMBER_ITERATION_MAX = 10000000000000;
for (k = 1; k <= NUMBER_ITERATION_MAX; k++) {        start the iterations in each subdomain


    // populate send left / right buffers..
    if (my_rank != nproc - 1) {
        for (j = 1; j < number_nodes_y-1; j++) {
            Send_to_R[j] = X_old[p(j, number_nodes_x - 2, number_nodes_x)];
        }
    }
    if (my_rank != 0) {
        for (j = 1; j < number_nodes_y-1; j++) {
            Send_to_L[j] = X_old[p(j, 2, number_nodes_x)];
        }
    }
```

Now, start the iterations. We have to first populate the send and receive buffer. So, what we are doing here we are solving A x = b. x will get the updated solution, and, the boundary conditions will go in b.

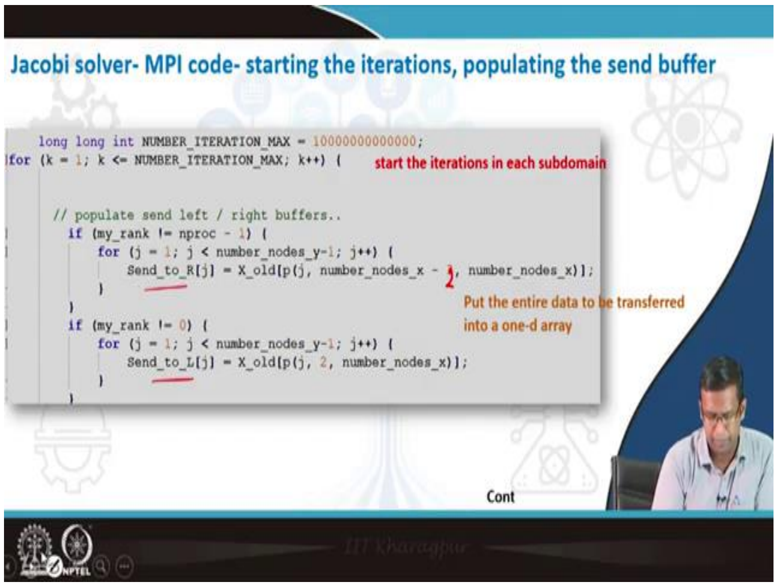
We have an inter domain boundary and we have to send data from the left domain to the right domain. We have to take the updated value the x values and these x values will be received by the next neighboring sub domain as the boundary conditions.

So, this will come to this capital B. This will come to B. So, we put X_old because the updated value after one iteration will be written in the old guess value. So, we are sending guess values here. They are basically the same, we could have done it after the update, we can do it before the update.

We are taking the guess solution and putting it into the send_to_R. This data will go to the right. What will go right? From the left boundaries of the sub domain, the data will go to the right boundary, right. Which data will go to the right boundary? If this data goes to the right boundary, it will come from the right side of the main sub domain and number of nodes x - 2 this will go to the next boundary.

What will go to the left boundary? 0 will not go, 1 will not go, 2 will go to the left boundary. What will go to the right boundary? If the number of nodes is 6, 6 - 2, 4 will go to the next boundary. What will go to the left sub domain? The left sub domain will go to the second value, 2 will go to the left sub domain. So, n x - 2 that will go to the right boundary and this is second from the left side, this will go to the left boundary.

(Refer Slide Time: 24:57)



Jacobi solver- MPI code- starting the iterations, populating the send buffer

```
    long long int NUMBER_ITERATION_MAX = 10000000000000;
for (k = 1; k <= NUMBER_ITERATION_MAX; k++) {


      // populate send left / right buffers..
        if (my_rank != nproc - 1) {
            for (j = 1; j < number_nodes_y-1; j++) {
                Send_to_R[j] = X_old[p(j, number_nodes_x - 1, number_nodes_x)];
                                                          2
            }
        }
        if (my_rank != 0) {
            for (j = 1; j < number_nodes_y-1; j++) {
                Send_to_L[j] = X_old[p(j, 2, number_nodes_x)];
            }
        }
```

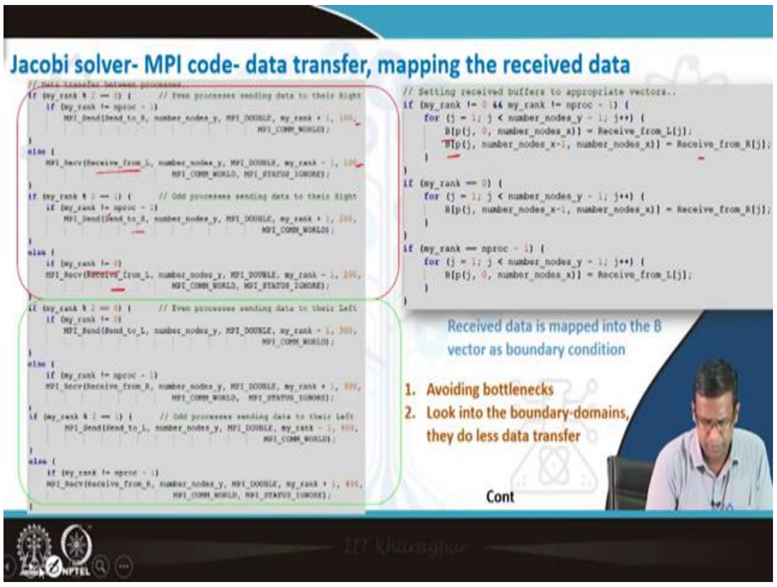
start the iterations in each subdomain

Put the entire data to be transferred into a one-d array

Cont

So, the number of nodes - 2 will go to the right boundary, and x =2 will go to the left boundary. So, we have started the iterations in each sub domain; the entire data that will be sent is put in an array, which is the send buffer array.

(Refer Slide Time: 25:21)



Jacobi solver- MPI code- data transfer, mapping the received data

```
// Data transfer between processes..
if (my_rank % 2 == 0) {      // Even processes sending data to their Right
    if (my_rank != nproc - 1)
        MPI_Send(Send_to_R, number_nodes_y, MPI_DOUBLE, my_rank + 1, 100,
                                                        MPI_COMM_WORLD);
}
else {
    MPI_Recv(Receive_from_L, number_nodes_y, MPI_DOUBLE, my_rank - 1, 100,
                            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
if (my_rank % 2 == 1) {      // Odd processes sending data to their Right
    if (my_rank != nproc - 1)
        MPI_Send(Send_to_R, number_nodes_y, MPI_DOUBLE, my_rank + 1, 200,
                                                        MPI_COMM_WORLD);
}
else {
    if (my_rank != 0)
    MPI_Recv(Receive_from_L, number_nodes_y, MPI_DOUBLE, my_rank - 1, 200,
                            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
if (my_rank % 2 == 0) {      // Even processes sending data to their Left
    if (my_rank != 0)
        MPI_Send(Send_to_L, number_nodes_y, MPI_DOUBLE, my_rank - 1, 300,
                                                        MPI_COMM_WORLD);
}
else {
    if (my_rank != nproc - 1)
    MPI_Recv(Receive_from_R, number_nodes_y, MPI_DOUBLE, my_rank + 1, 300,
                            MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
}
if (my_rank % 2 == 1) {      // Odd processes sending data to their Left
        MPI_Send(Send_to_L, number_nodes_y, MPI_DOUBLE, my_rank - 1, 400,
                                                        MPI_COMM_WORLD);
}
else {
    if (my_rank != nproc - 1)
    MPI_Recv(Receive_from_R, number_nodes_y, MPI_DOUBLE, my_rank + 1, 400,
                            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

```
// Setting received buffers to appropriate vectors..
if (my_rank != 0 && my_rank != nproc - 1) {
    for (j = 1; j < number_nodes_y - 1; j++) {
        B[p(j, 0, number_nodes_x)] = Receive_from_L[j];
        B[p(j, number_nodes_x-1, number_nodes_x)] = Receive_from_R[j];
    }
}
if (my_rank == 0) {
    for (j = 1; j < number_nodes_y - 1; j++) {
        B[p(j, number_nodes_x-1, number_nodes_x)] = Receive_from_R[j];
    }
}
if (my_rank == nproc - 1) {
    for (j = 1; j < number_nodes_y - 1; j++) {
        B[p(j, 0, number_nodes_x)] = Receive_from_L[j];
    }
}
```

Received data is mapped into the B vector as boundary condition

1. Avoiding bottlenecks
2. Look into the boundary-domains, they do less data transfer

Cont

So, first is that even processors will send ,if my id myrank is not nproc - 1. So, if it is not the last processor, then only it will send to the right sub domain, because the last processor has nothing in its right, it is the physical boundary.

So, if the processor id is divisible by 2 it will send the right sub domain; if the processor id is not divisible by 2, it will receive from the left sub domain. So, first the even processors will send and the odd processors will receive. Processor id starts from 0. So,it starts from the even processor; even processors will send at that time odd processors will receive.

And, where will it be received? send_to_R, in n max - 2's y line that we have put here that will be received in receive_from_L buffer. Similarly, and the matching tags will ensure that the same data is which is sent that will be received.

Similarly, if it is not divisible by 1, then it will send to the right boundary , so first it is it will if it is divisible by 2, it will send and if non divisible by 2 it will receive. Now, if it is not divisible by 2, then after this receive it will send data to the right hand side and the even processors will receive data and from the left.If myrank = 0; that means nothing on left.

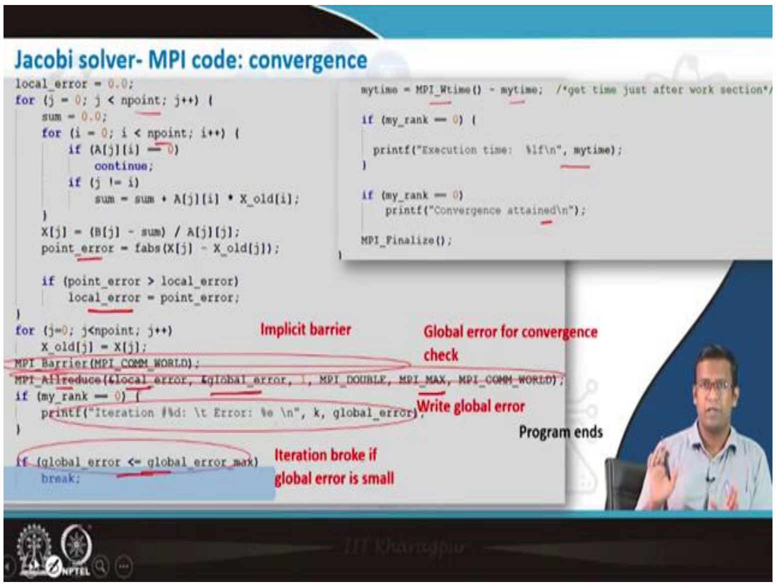
This odd even staggering is done to avoid the bottleneck and we can see that the boundary domains that is myrank = 0, and myrank = nproc - 1, they do less operations. So, boundary domains will do less operations. They do less data transfer.

Received data is mapped into the B vector. So, what is being received from R that will be mapped into the B vector because that is the boundary condition now. So, what is received from a L is mapped into the B vector ,received from right is also mapped into the B vector. All the received data are mapped into the B vector.

If myrank is not 0, my id is non zero, then data is received from both left and right.If myrank is 0 that is the first processor it has nothing from the left side. So, it will receive only from the right side, the last processor will receive only from the left side. So, the first and last processors have physical boundaries; they will do less data transfer.

Now, all these steps have to be thought of by the programmer, put it into the right algorithm and then put in the right positions of the program. In open MP programming we have seen that the compiler itself does a lot of things if you give the right open MP construct, but in MPI programming or in distributed computing everything has to be done by the programming. Especially in domain decompositions, all this data transfer, all the mapping taking care of all the physical boundaries everything has to be well thought by the programmer and implemented here.

(Refer Slide Time: 29:05)



Jacobi solver- MPI code: convergence

```
local_error = 0.0;
for (j = 0; j < npoint; j++) {
    sum = 0.0;
    for (i = 0; i < npoint; i++) {
        if (A[j][i] == 0)
            continue;
        if (j != i)
            sum = sum + A[j][i] * X_old[i];
    }
    X[j] = (B[j] - sum) / A[j][j];
    point_error = fabs(X[j] - X_old[j]);

    if (point_error > local_error)
        local_error = point_error;
}
for (j=0; j<npoint; j++)
    X_old[j] = X[j];
MPI_Barrier(MPI_COMM_WORLD);
MPI_Allreduce(&local_error, &global_error, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
if (my_rank == 0) {
    printf("Iteration #%d: \t Error: %e \n", k, global_error);
}

if (global_error <= global_error_max)
    break;
```

```
mytime = MPI_Wtime() - mytime;  /*get time just after work section*/

if (my_rank == 0) {

  printf("Execution time:  %lf\n", mytime);
}

if (my_rank == 0)
    printf("Convergence attained\n");

MPI_Finalize();
}
```

Implicit barrier
Global error for convergence check
Write global error
Program ends
Iteration broke if global error is small
IIT Kharagpur
NPTEL

Now, we do the iterations and this iteration will be done only for the local number of points. There is an implicit MPI barrier after that there is an implicit barrier is MPI allreduce, but we put an explicit barrier also.

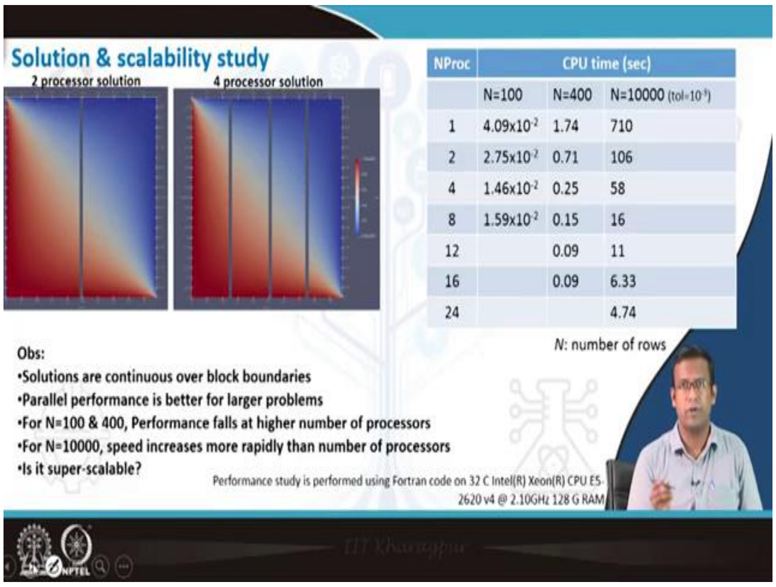
 From each point within each processor we calculate a local error, now we take this local errors called MPI allreduce maximum and maximum of these local errors come as global error.

So only for myrank 0 processor we ask them to print global error, because now we are looking for convergence based on the global error. If the global error is less than the error max, then it is a convergence error. We are not looking into local error or error in each processor, we are only seeing if global error has reduced around within some value.

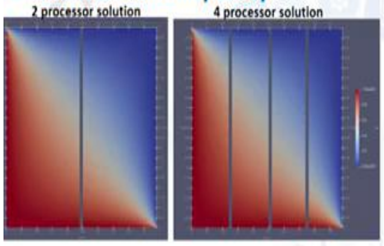
So, we write global error and then if it has not reached convergence level continuing with the iterations. Do global error for convergence check, write global error iteration block if global error is small break and then again call MPI_Wtime and subtract the previous my time. So, we get the time interval within which this MPI calls and the domain decomposition based Jacobi solver has been operated and write the execution time.

Again all these writing operations are done by only one processor myrank 0.It writes that convergence attained and MPI finalize comes out of the program. So, this is our first MPI parallel program for a matrix solver.

(Refer Slide Time: 30:58)



Solution & scalability study



NProc	CPU time (sec)		
	N=100	N=400	N=10000 (tol=10^{-3})
1	4.09×10^{-2}	1.74	710
2	2.75×10^{-2}	0.71	106
4	1.46×10^{-2}	0.25	58
8	1.59×10^{-2}	0.15	16
12		0.09	11
16		0.09	6.33
24			4.74

N: number of rows

Obs:
- Solutions are continuous over block boundaries
- Parallel performance is better for larger problems
- For N=100 & 400, Performance falls at higher number of processors
- For N=10000, speed increases more rapidly than number of processors
- Is it super-scalable?

Performance study is performed using Fortran code on 32 C Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz 128 G RAM

Now, we look into the solutions, we can see that the solutions show the continuity and it looks right visually.We can also calculate and validate with the analytical solution .This data comes from a Fortran code (we have shown you a similar C code)on 32 processor Xeon CPU with 128 GB RAM.

For a matrix of 100 by 100 size for 1 processor the time is 4.09 seconds; 2 processor time is 2.75* 10^{-2} seconds; 4 processor it is further reduced 1.46*10^{-2} second. 8 processor it increases to 1.5 *10^{-2} seconds.

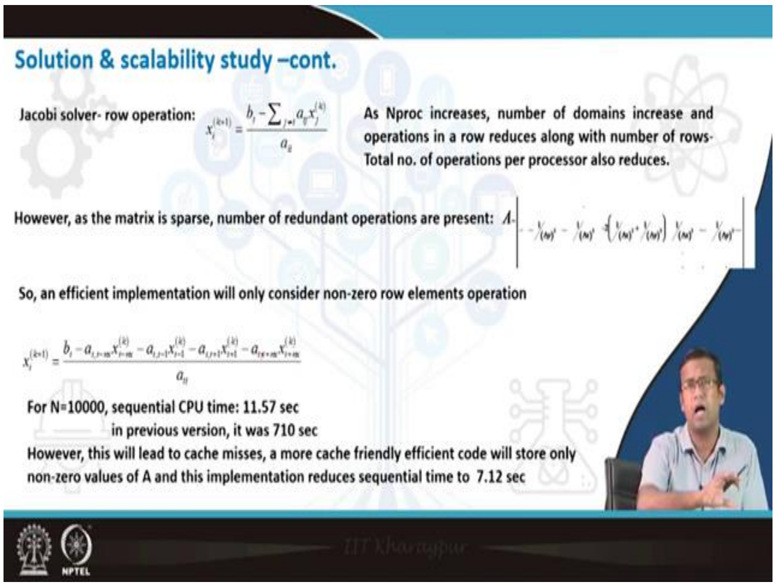
For a 400 by 400 matrix 1 processor is 1.74 seconds ;2 processor 0.71 seconds; 4 processor 0.25 seconds; 8 processor 0.15 second; 12 processor 0.09 seconds; 16 it is also 0.09. So, after a certain number of processors we are not getting the advantage of increasing the number of processors and it is well known to us already looking into the performance matrix of the processors that after a certain extent parallel overheads increase and we do not get better performance.

There is an interesting case for a large 10000 by 10000 matrix ,1 processor 710 seconds; 2 processor 106 seconds; 4 processor 58 second; 8 processor 16 seconds. So, the time elapsed is reducing more than the added number of processors, and it keeps on reducing up to 4.74 second.

The observations are solutions are continuous over block boundaries.Parallel performance is better for larger problems. As we are increasing the problem we are getting more speed up in that sense. For N = 100 and 400 performance falls at a higher number of processors, because the computing to communication ratio falls down and performance falls.

For N = 10000 speed increases more rapidly than number of processors. Is it really giving us super scalability, because ideally if we increase the number of processors, speedup should not be more than the 45 degree curve, but here we can see much higher speed up here. Is it super scalable? So, I will go to the next slide and discuss that.

(Refer Slide Time: 33:43)



Solution & scalability study –cont.

Jacobi solver- row operation: $x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}}{a_{ii}}$

As Nproc increases, number of domains increase and operations in a row reduces along with number of rows- Total no. of operations per processor also reduces.

However, as the matrix is sparse, number of redundant operations are present: $A = \left| \cdots \frac{1}{(\Delta y)^2} \cdots \frac{1}{(\Delta x)^2} \; -2\left(\frac{1}{(\Delta x)^2}+\frac{1}{(\Delta y)^2}\right) \; \frac{1}{(\Delta x)^2} \cdots \frac{1}{(\Delta y)^2} \cdots \right|$

So, an efficient implementation will only consider non-zero row elements operation

$$x_i^{(k+1)} = \frac{b_i - a_{i,i-m} x_{i-m}^{(k)} - a_{i,i-1} x_{i-1}^{(k)} - a_{i,i+1} x_{i+1}^{(k)} - a_{i,i+m} x_{i+m}^{(k)}}{a_{ii}}$$

For N=10000, sequential CPU time: 11.57 sec

in previous version, it was 710 sec

However, this will lead to cache misses, a more cache friendly efficient code will store only non-zero values of A and this implementation reduces sequential time to 7.12 sec

The operation in Jacobi row solver is $x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} \; a_{ij} x_j^{(k)}}{a_{ii}}$. A processor operates within the rows assigned to it. In each row if it is a single processor case it operates over all the column elements. If it is a multiprocessor case it operates over less number of rows and within the sub matrix it also operates over less number of columns.

So, operations are reduced in terms of N square and that is why you are getting speed which is not scaled with N, but rather scaled with N square and this is specially observable for large problems. As Nproc increases, the number of domains increases and operations in a row reduces along with the number of rows. Therefore, the total number of operations per processor also reduces. It is not only operations in a processor that are reduced by a factor of the number of subdomains. Operations also reduce because operations within a row reduces.

However, as the matrix is sparse the lot of operations are redundant. So, if you remember when we talk about scalability, the speedup is obtained by dividing the parallel time by the most efficient single processor time. But, here as we are doing this multiplication we are not doing it very efficiently, because there are a lot of zeros in one row. Only these five are nonzero numbers. Still we are doing operations for these zeroes.

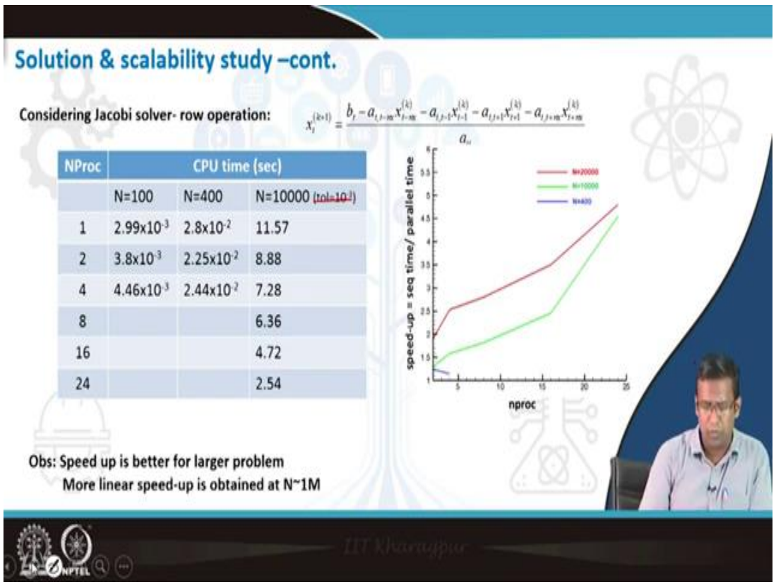
So, the single processor program is also not an efficient program. In order to make an efficient program we only have to consider the nonzero numbers and instead of doing this matrix vector multiplication for all the elements including the zero elements we will do it for only the nonzero numbers. So, the efficient implementation will be like only considering the nonzero numbers and their multiplication with the coefficients.

For N = 10000, the CPU time is 11.57 seconds; where earlier when we are doing it for all elements in the column this time is a sequential program this time was 710 seconds. Now, if we take this program then we will see that it is not actually super scalable, because the number of operations in a row is also fixed, only five operations it is doing in a row.

However, this can lead to a cache miss issue, because it is looking into x $_{i-1}$, then x $_i$ + n x. So, if the matrix is large, then all the x vectors may not sit in the fit in the same cache. So, with an increasing number of processors it might give more cache misses and fall down the performance.

So, one better idea can be that if we can consider all a's is rather in a contiguous matrix and only five elements in a matrix is considered then it will be more cache friendly, and if we can do that the calculation time from 11.5 second ;this is a single process of calculation time reduces to 7.12 second. So, parallel parallelization high performance computing is fine, but we must be sure that we are doing the right thing solving a large problem most efficiently even when we are doing sequential processor process calculation.

(Refer Slide Time: 37:16)



Solution & scalability study –cont.

Considering Jacobi solver- row operation: $x_i^{(k+1)} = \dfrac{b_i - a_{i,i-m}x_{i-m}^{(k)} - a_{i,i-1}x_{i-1}^{(k)} - a_{i,i+1}x_{i+1}^{(k)} - a_{i,i+m}x_{i+m}^{(k)}}{a_{ii}}$

NProc	CPU time (sec)		
	N=100	N=400	N=10000 ~~(tol=10^{-3})~~
1	2.99×10^{-3}	2.8×10^{-2}	11.57
2	3.8×10^{-3}	2.25×10^{-2}	8.88
4	4.46×10^{-3}	2.44×10^{-2}	7.28
8			6.36
16			4.72
24			2.54

Obs: Speed up is better for larger problem
More linear speed-up is obtained at N~1M

So, we are not considering the cache friendly operation, but just considering only the nonzero numbers are taken and five operations are done for each row for matrix vector product. We can see that for N = 100, 2.99 $*10^{-3}$ seconds for 1 processor; 3.8$*10^{-3}$ for 2 processor; 4.46$*$ 10^{-3} .

So, N = 100 parallelization is not beneficial as we increase the number of processors, because computing is very less; only five computations per row and number of rows are also less. As we increase the number of processor communication is more and parallel overheads are reducing the performance.

N = 400 performance is reducing slowly, but not as good as we have seen earlier. N = 10000 we can see that 11.5 second is single processor performance; 8.88 is 2 processors time; 7.28 is 4 processor times so on. So, scalability is linear, speed up is not super linear. Scalability is linear with number of performance.Earlier we have done it for tolerance of 10 to the power -3, but here we converge the matrix because calculation time is less. Now, we can see that for N = 400, if we see sequential time by parallel time which is sped up by N processors ,the speed up actually falls with increasing number of processors. Speed up is greater than 1, but it falls for an increasing number of processes.

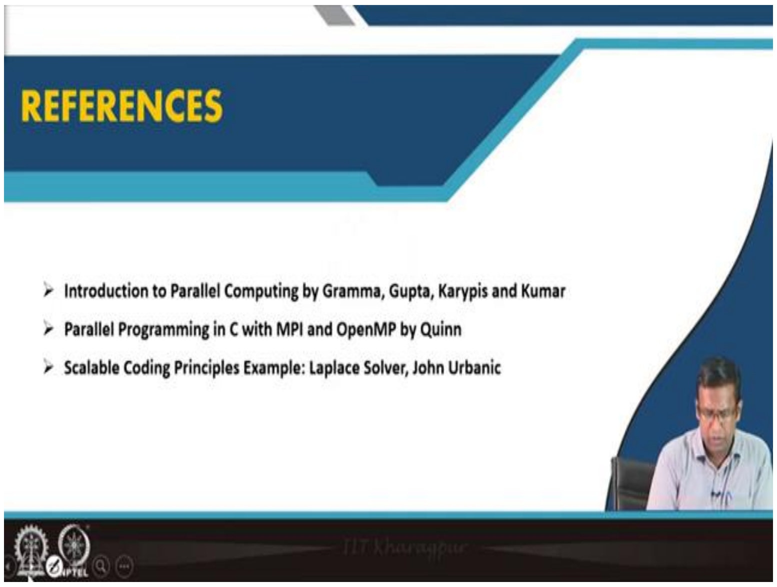
So, it is not beneficial to go to 4 processors; for N = 100 speed up is less than 1 we do not consider that; for N = 10000 speed up increases with the number of processors, but it is less than the 45 degree slope. For N = 20000 speed up is better because with the same number of processors, if we can increase the problem size parallel efficiency will be better, speed up is better.

Speed up also shows an increase in slope after 16 processors and this is probably due to the issue that if you increase the processor local matrix sizes are smaller and the cache miss is less small; if the matrix is large cache miss is more. So, speed up increases here.

So, the observation is speed up is better for the larger problem and if we take a very small problem speed up is bad, if we take a larger problem speed up is good. For N a close to 1 million or even more than that we get more linear speed up and if we take a very large problem like 5 million problems we have tested speed up scalability line is almost close to the 45 degree scope up to a large number of processors.

So, you can get good speed up to a large number of processors if we take a large problem.

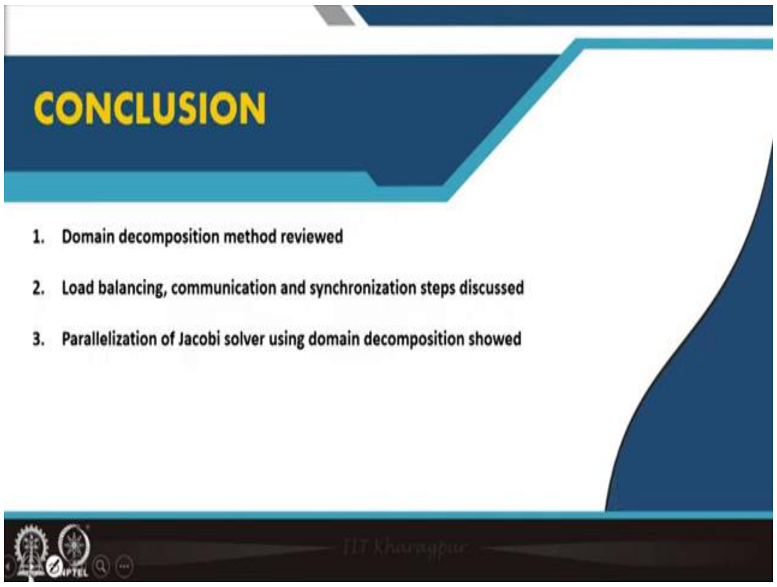
(Refer Slide Time: 40:26)



REFERENCES

- Introduction to Parallel Computing by Gramma, Gupta, Karypis and Kumar
- Parallel Programming in C with MPI and OpenMP by Quinn
- Scalable Coding Principles Example: Laplace Solver, John Urbanic

(Refer Slide Time: 40:29)



CONCLUSION

1. Domain decomposition method reviewed
2. Load balancing, communication and synchronization steps discussed
3. Parallelization of Jacobi solver using domain decomposition showed

Well, these are the references and we have discussed domain decomposition methods; we have discussed load balancing, communication, synchronization.We took up a Jacobi solver parallelized it using domain decomposition method and also looked into the performance of the solvers.