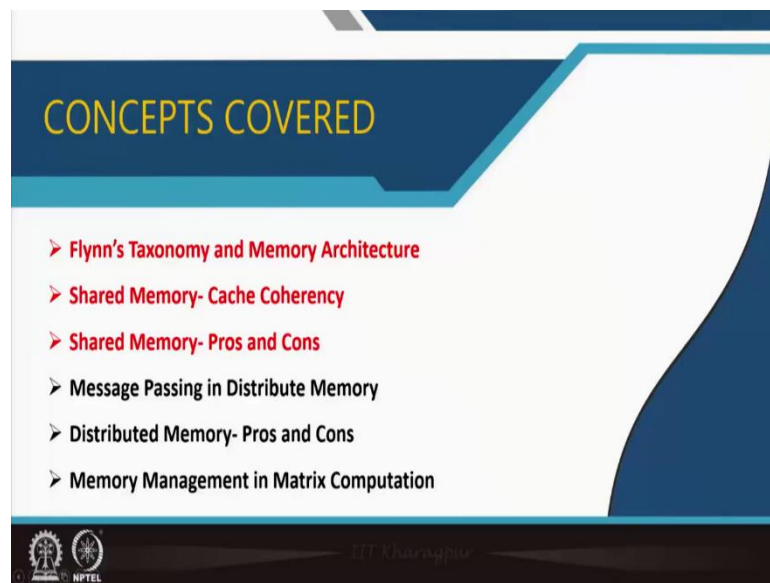


High Performance Computing for Scientists and Engineers
Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module - 01
Fundamentals of Parallel Computing
Lecture - 06
Shared Memory and Distributed Memory in Parallel Computing (continued)

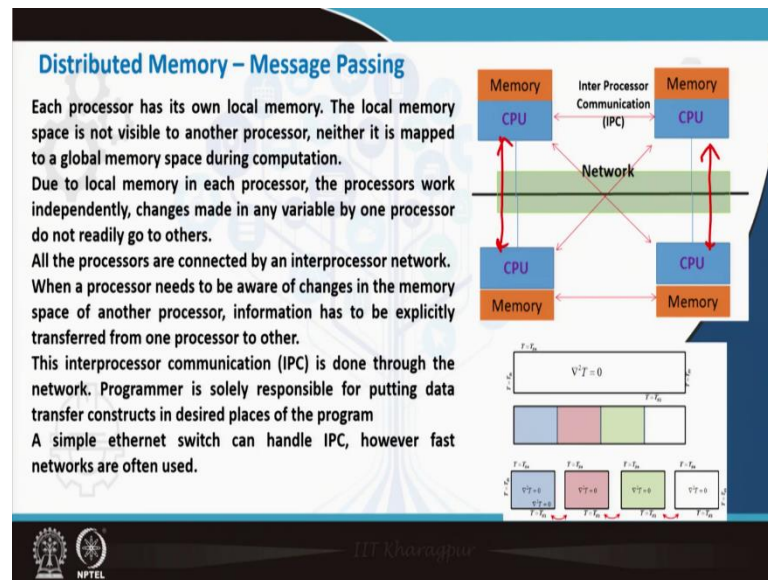
Welcome. We are discussing about Shared Memory and Distributed Memory in our course High Performance Computing for Scientists and Engineers. We have already discussed about shared memory-cache coherence, and the advantages and disadvantage of shared memory systems and this is the continuation of the lecture.

(Refer Slide Time: 00:43)



We will now look into message passing in distributed memory systems the advantages and disadvantages of using distributed memory systems, and memory management, using both shared and distributed memory, with an example focusing with three examples focusing on matrix computing.

(Refer Slide Time: 01:10)



Well, distributed memory means different computers each with their own CPU, own RAM, own cache, connected via network switch to each other. Each processor has its own local memory, and this local memory is not visible to other processors; neither it is mapped to a global memory space during computation. It is a local memory, and it is local to everybody. Everybody has a part of memory; each computer knows that you have to work on this part of memory; you do not know what is with others. This is a distributed memory system. Each has different memory and different computers are doing different memory.

Then what are they doing in parallel? In parallel, they are trying to solve a same problem. Therefore, some of the information is needed from other processor, say you need to establish continuity in between the processors in some of the variable. So, in order to have continuity, some of the neighboring elements of the memory might come from the next processor's memory. So, four computers are trying to do some calculation on different data, but, same type of computation.

Once this is done, they will exchange this and collect it, they find out what is maximum, etcetera. So, they need to have some communication in between them, because there is otherwise nothing common in between them. But as they are solving one same problem, they need to communicate it.

And due to local memory in each processor, the processors work independently, changes made by in any variable of by one processor do not readily go to others. But if they are

working on the same problems, some of these changes need to go from one to other. And this is established by message passing. All the processors are connected to an inter processor network. When a processor needs to be aware of changes in memory space which has been done by another processor. Some memory element which is changed by the other processor, but as it is a same problem which all processors are working on, one processor needs to know what the other processor has done.

Like the first example that chord broken into or dismantled into 4 pieces will be moved from one building to other, and these four pieces are taken care by four different porters. Now, when they will go to the next building, they need to put everything together. So, first they need to see whether they are going to the same place and whether the pieces are coming one after another, what should be their arrangement. One porter not only can take care of his own part of the chord arm, but he also needs to see what others are doing, some information, not all the details, but some information from what others are doing.

So, some information has to come from one processor to other processor. A processor needs to be aware at certain stages the processor needs to be aware by of the changes in memory space, which is done by another processor. Then, explicitly this information has to be transferred from one processor to the other processor. So, the programmer has to identify the locations where one processor needs to get some information from the other processor, and programmer has to specify some construct using message passing interface or MPI. Programmer needs to write some constructs which will say, hey, processor 0, you have to transfer this information this packet of data to processor 3.

So, it has to be explicitly done by the programmer, that this information from this processor will reach to this processor. So, in a sense there will be a inter processor communicator or IPC framework which will allow each of the processor to exchange data with the other.

So, all the processors should be able to communicate among each of them, that means, data from one of the processors can go to any of the processors in the network, that flexibility has to be given. And the programmer will specify which processor at which instance sends what type of data to which processor, and explicitly one processor will send the data, explicitly there will be another processor which will receive the data.

Say, I have to pass an envelope to my friend; if I give him, the envelope and he does not take it, then the message passing is incomplete. So, one gives another has to receive. And

all these things have to be explicitly defined in the program. The inter processor communicated, IPC, is done through network and the programmer is solely responsible for putting data transfer constructs in desired places of the program. In shared memory, you write, you have to update the variable, some changes has been made by the variable, it is the backend, it is the compiler, it is the library functions, it is the message passing infrastructure or parallel infrastructure ,which is parallelization infrastructure which is taking care of taking one packet of data and sending it to network cable or bus from one processor cache and sending it to back to the main memory, and then the copies are going to other cache that is in shared memory.

But in distributed memory the programmer has to write that this information will go from this processor, will go to this processor at this instance, and this processor will receive this information. The compiler or the library functions or the interfaces or the parallel architecture hardware architecture, now no one is going to take care of that; it is the programmers responsibility to ask all the backend things like compilers and interface, APIs, etcetera to ensure that this data transfer is done(Refer Time: 07:40).

We are solving a Laplace equation in a large domain, and it is broken into four sub domains (We will come back to this problem when we will look into MPI programs)and each of this program is solving the same equation on a smaller domain, on the sub domain, this is called a domain decomposition technique. However, in order to solve this variable T , whose Laplacian is called here, $\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$, we need a continuity between T . And while again solving the Laplacian T is equal to 0, here we need a boundary condition. Both boundary conditions or the continuity will be ensured by passing the value of T from this location to this processor, and getting the value of T there to this processor's memory space. So therefore, data transfer will have happened across the processors while solving this equation.

So, the procedure for solving this equation is that you take the last updated data in this processor, solve the equation in this domain. Update this variable, send it to this processor who can solve this equation using this update. Again, the updated value at this processor will come to, this processor while solving this equation. So, a dynamic exchange of data during each stage of calculation is required in different processors. And this is ensured by framework like this, where all the processors can send data to each other.

Now, one thing can be seen that if one is sending data from here to hear the processors are physically close, and these processors are physically distant, so, the data transfer time will be more. So, some data locality, localization, some processor localization, are important. And we will again discuss these, these issues when we look into how to improve the performance of parallel computing is in distributed memory systems.

A simple Ethernet switch can handle this inter processor communication. However, fast networks 10 Gigabit Ethernet, 100 Gigabit Ethernet, mini net, InfiniBand switches they are used, because it is a good amount of data which will be sent, and if the data transfer time is more much more order smooth than the computing time, the performance will be bad. So, good supercomputers, good HPC platforms, use fast networks for data transfer.

(Refer Slide Time: 10:18)

Distributed Memory- Pros and Cons

Advantages

- Memory is scalable with number of processors, as number of processor is increased, memory is geometrically increased
- Each processor can rapidly access memory without traffic or contention from other processors or without overheads of cache consistency
- Cheap- High end motherboards and large RAMs are not required, community machines can be connected by a switch

Disadvantages

- Programmer is responsible for many details associated with data transfer
- Requirement of mapping local memory to a global data structure- complexity!

Required to synchronize
On fly data consistency by programming instances
Final assembly of results

NPTEL IIT Kharagpur

The advantages of distributed memory are that memory is scalable with number of processors. As we increase number of processors, the memories geometrically increased. This is not a common memory space where all the processors are connected. Each processor has its own local memory space. So, if we have a very large problem, we do not need to be constrained by having a same contiguous memory space, it is different small local memory units. As I am putting a new computer its coming with its own local memory, and therefore, then total memory of the problem is increasing.

Each processor rapidly access memory without traffic or contention, because each one is using its own memory, own cache, it is the local memory only. Therefore, the cache update

needs, it is not from some other processor's memory or from some other remote memory's location, cache update is coming to all the computers. And there is no question of having inconsistent cache as each cache is decoupled. So, this overhead can be removed, that is the contention or traffic due to access of the same memory and cache consistency over it.

High end motherboards and large rams are not required. Simple commodity machines some good servers can be connected by a switch, and we can make a large distributed memory platform, it is a cheap system. Another very important thing is that as the memory is scalable, we can break down a large problem into many small memory problems. So, you do not need to think about handling programming ways of sending a very large memory problem. So, the net memory can be also broken down, and each computer is accessing a small part of memory.

Disadvantage is that programmer has lot more duties now. He is responsible for each data transfer. He has to know all the details of the data transfer, and he has to specify when which type of data, what should be the data size will go from one processor to other. He has to ensure that the receiving processor is free to receive the data all these things.

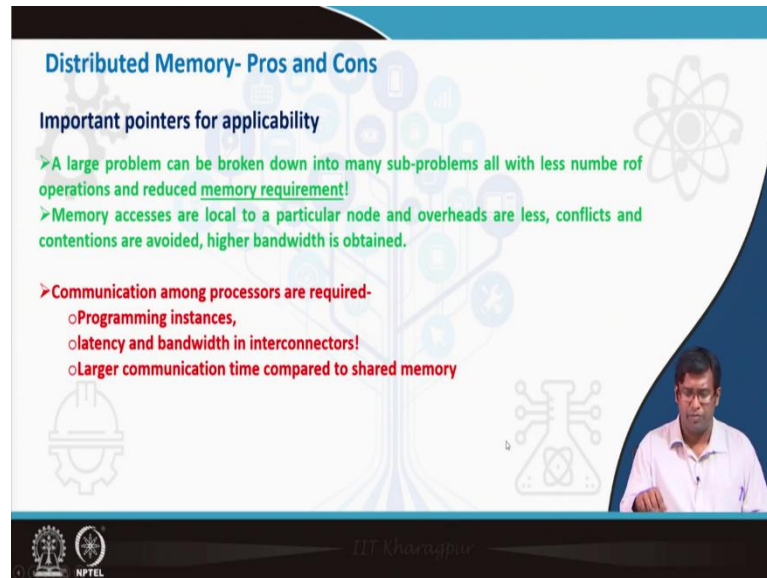
So, programmer has to handle much more complexities compared to shared memory. And in case there is a global data structure, breaking it down to small local memories, and then once the solutions are done, mapping all local memories into a global data structure, is also sometimes a complex problem. So, there are much more complexity compared to the shared memory space problems.

Important steps are that, there is a very heavy requirement of synchronization, because there is more flexibility mode synchronization is required, all the computers are working on different data. The data size may be different, the operations may be different also. But if you are solving a same problem at several stages, you need to see that all the arms of the chords are chord is together, so that you can get back the final chord., so, you can assemble it.

So, synchronization is required. On fly, data consistency may have to be ensured like that solving of Laplace equation that while doing the iterations for the matrix solver for the Laplace equation, you have to see that the temperature, T field or temperature field is consistent. So, you need to do data transfer or check some of this data consistency on fly

at different programming instances. And finally, the results have to be assembled using some of the mapping to the to a local memory.

(Refer Slide Time: 13:54)



The slide is titled "Distributed Memory- Pros and Cons". It lists "Important pointers for applicability" in green text:

- A large problem can be broken down into many sub-problems all with less number of operations and reduced memory requirement!
- Memory accesses are local to a particular node and overheads are less, conflicts and contentions are avoided, higher bandwidth is obtained.

Below these, in red text, it states "Communication among processors are required-" followed by three bullet points:

- Programming instances,
- latency and bandwidth in interconnectors!
- Larger communication time compared to shared memory

The slide also features a small video inset of a man in a white shirt on the right side. At the bottom, there are logos for IIT Kharagpur and NPTEL.

Important pointers are, a large problem can be broken down into many sub problems with a smaller number of operations and reduced memory requirement. As the problem is broken down into small problems, problem requires a small memory. Therefore, a computer with a small memory can handle this sub problem. Therefore, a you do not need to use a large memory, also you do not need to operate over a large address space which is good.

Memory accesses are local to a particular node and overheads are less, contentions are avoided. And one computer is accessing one memory therefore, high bandwidth is obtained. However, communication among the processors are required and at different programming instances are to be written to ensure this communication. Latency and bandwidth in the interconnectors have to be looked up on.

Communication through a PCI bus is much faster than communicating through a very first main at switch. Therefore, there will be more latency, because communication itself has its own latency as well as there will be less bandwidth because now it is going to a network switch. So, communications have to be some way reduced or localized, so that this latency and bandwidth issue can be handled. And it has larger communication time compared to

shared memory. Communication time before a new communication, communication time is much larger compared to a similar shared memory platform.

(Refer Slide Time: 15:41)

Summary of Features *Shared Memory*

Comparison of Shared and Distributed Memory Architectures

Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vxss DEC/Compaq SGI Challenge IBM POWER3	Bull NovaScale SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2 IBM BlueGene
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw Backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	100s ISVs

NPTEL

So, we summarize some of the important architectures. One is a uniform memory access shared memory system, centralized plus computers with uniform memory access. And this can be symmetric multiprocessors some of the IBM machines, SGI challenged machine etcetera. The APIs which can be called to parallelize are MPI threads, open MP.

This machine can be scaled to 10 sub processor, 20, 30, 40, 80, 100 processors can be put in a single motherboard or single bus, or an inter connector which is connected to the same piece of memory. And the drawback is memory CPU bandwidth, because many processors are trying to access the same memory and also cache coherence etcetera there.

A large number of software's, independent software vendors, almost everybody's program software is written for this type of arrangement. The next one is non uniform memory access IBM POWER4, HO Exemplar, SGI Origin, they are examples of that in similar software's API is like MPI, open MP goes there this can be scaled to 100 sub processors. So, many processors 100, 200, 300, 500 processors can be put into an inter connect switch and, though they are computers with different RAMs, but this memory can be mapped into the symbol into the shared memory space.

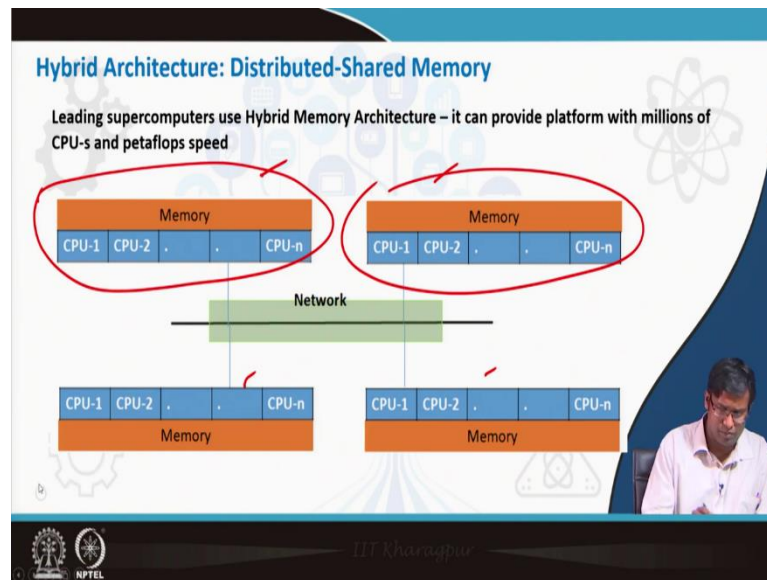
And again, the issue is in on non-uniform access times and memory CPU bandwidth here. However, almost all the software vendors provide software for this type of system. The best thing of shared memory system is that there are thousands of software's for shared memory system. So, you do not need to write anything most likely, you will get a program which will run in your shared memory system.

The distributed memory system, instead of all these different communicated APIs, only one communicator is universally accepted to work on the distributed memory system which is MPI. Good thing is that it can be scaled over a large number of processors, 1000's of processors can be connected with each other.

The drawback is system administration is hard to develop a system and maintain this. Because many computers are connected via a switch what is the stay health of each of the computers how to monitor, when whether one is free, whether one is working fine, you need a good scheduler, so that different SPMD jobs can work together. If there are thousands of processors, maybe you are not your own program is not using 1000 processor, somebody else is also using some of the processors for his own program.

So, a scheduler is needed, so that these jobs run well without hampering performance and bandwidth etcetera. And very a smaller number of independent software vendors actually write supply program for distributed memory architecture. So, many times you have to be a programmer yourself to work on a distributed memory system, because the standard available programs, commercially available software's, many times do not work for distributed memories.

(Refer Slide Time: 19:33)



Another is hybrid architecture. The leading supercomputers use hybrid architecture, where distributed memory platform is given, but each of the distributed memory system is also a uniform memory access SMP – symmetric multiprocessor, that each of the box has multiple processors connected with the same memory. And it can actually scale up to millions of processors. So, it is called a hybrid memory architecture, which is a distributed memory in some sense, because all these are access distributed memory computer, and any of this acts as a shared memory parallel computer. So, it is a distributed cum shared memory, hybrid memory architecture.

(Refer Slide Time: 20:32)

Memory Management for in Matrix Algebra (Dot Product)

Consider Vector-Vector Multiplication

$$\vec{a} \cdot \vec{b} = c \Rightarrow \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \cdot \begin{pmatrix} b_1 & b_2 & \dots & b_n \end{pmatrix} = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

np processors, each processing $n/np=m$ number of elements in a and b :

In Shared Memory:

It is the same variable c , which all the processors are updating simultaneously

do $i=1,m$
 $c = c + a[\text{processor id} * m + i] * b[\text{processor id} * m + i]$
end do

simultaneous access to c by all processors through the do-loop

Overhead due to cache coherence!

Processor 0: $c = c + a_1 b_1 + a_2 b_2 + \dots + a_m b_m$

Processor 1: $c = c + a_{m+1} b_{m+1} + a_{m+2} b_{m+2} + \dots + a_{2m} b_{2m}$

Processor 2: $c = c + a_{2m+1} b_{2m+1} + a_{2m+2} b_{2m+2} + \dots + a_{3m} b_{3m}$

Processor $np-1$: $c = c + a_{(n-1)m+1} b_{(n-1)m+1} + a_{(n-1)m+2} b_{(n-1)m+2} + \dots + a_n b_n$

Now, we look into some of the simple matrix operation programs and see how are the memory management done for that. So, one is a dot product. We have to consider vector-vector multiplication. Vector a multiplied with vector b gives the scalar c, which is c is equal to $a_1b + a_2b_2 + \dots + a_nb_n$. Now, we will assume that these are large vectors, million size vectors. And hypothetically or as a thought exercise we will deploy parallel computing architecture to get the vector-vector product matrix vector product etcetera.

So, if we have np processors, let us assume that n is the number of elements in one vector, the total number of elements is n, therefore each processor is getting n p by n which is m number of elements in a and b. So, what each person will do will take this m number of elements, get the sum, and all the processor will get the sum, and finally they will get a final sum.

If we use a shared memory, we have said np processor, processor id is starts from 0 goes up to n p minus 1. And first processor is doing c is equal to $c + a_1b + a_2b_2 + \dots + a_mb_m$, why it is writing c because all the processors cannot concurrently write the same variable without using some protocol. So, they are using the protocol sum, where everybody is writing a sum. So, they can add up to the variables. Processor 2 write c is equal to $c + a_{m+1}b_{m+1} + a_{m+2}b_{m+2} + \dots + a_{2m}b_{2m}$.

So, as the processors operate in parallel finally, when the operation is done, we get that the c has been updated by all the processors. So, it is the same plus variable c which all the processors are updating simultaneously, using some concurrent right option or some protocol. And each of the processor is some way applying this Fortran programming snippet for m elements write c is equal to c plus the particular location which is allotted to this processor to work on a matrix into b matrix. So, this is the same programming instance which will be running by all the processors. We are not writing different programming instance for different processors; this will be run by all the processors. And how all the processors will work together, that will be ensured by some API like in MPI or open MP.

Simultaneous, access to c is by all the processors through the do loop. So, in this do loop, all the processors are simultaneously accessing c. So, there is an issue of cache coherency, though this is a concurrent write protocol which some way is allowed to handle the cache issue. However, the cache updates are required by all the processors. And during each of the instance of this do loop, this cache coherency is established across all the processors

which is probably adding to overhead, Well, the programming is this simple ,it is the same line which will be executed by all the processors, and it is the same matrix we are going to see.

(Refer Slide Time: 23:59)

Memory Management for in Matrix Algebra- Dot Product

Consider Vector-Vector Multiplication

$$\vec{a} \cdot \vec{b} = c \Rightarrow \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \cdot \begin{pmatrix} b_1 & b_2 & \dots & b_m \end{pmatrix} = a_1 b_1 + a_2 b_2 + \dots + a_m b_m$$

np processors, each processing $n/np=m$ number of elements of local a and b :

In Distributed Memory:

Processor 0: $\hat{a}_1 = a_1, \hat{a}_2 = a_2, \dots, \hat{a}_m = a_m; \hat{b}_1 = b_1, \hat{b}_2 = b_2, \dots, \hat{b}_m = b_m$

Processor 1: $\hat{a}_1 = a_{m+1}, \hat{a}_2 = a_{m+2}, \dots, \hat{a}_m = a_{2m}; \hat{b}_1 = b_{m+1}, \hat{b}_2 = b_{m+2}, \dots, \hat{b}_m = b_{2m}$

Processor 2: $\hat{a}_1 = a_{2m+1}, \hat{a}_2 = a_{2m+2}, \dots, \hat{a}_m = a_{3m}; \hat{b}_1 = b_{2m+1}, \hat{b}_2 = b_{2m+2}, \dots, \hat{b}_m = b_{3m}$

Processor $np-1$: $\hat{a}_1 = a_{(n-m)+1}, \hat{a}_2 = a_{(n-m)+2}, \dots, \hat{a}_m = a_n; \hat{b}_1 = b_{(n-m)+1}, \hat{b}_2 = b_{(n-m)+2}, \dots, \hat{b}_m = b_n$

For each processor: $\hat{c} = \hat{a}_1 \hat{b}_1 + \hat{a}_2 \hat{b}_2 + \dots + \hat{a}_m \hat{b}_m$

Finally $c = \sum_{processor=0}^{np} \hat{c}$

Communicate a single value from each processor to others/master – Gather
Reduction of all local values to a global sum

NPTEL IIT Kharagpur

Now, if we look into the distributed memory, it is same that each processor is working on m number of elements. So, each has its local memory. The first processor has $a_1, a_2 \dots a_m$ and $b_1, b_2 \dots b_m$ on which it will work. It stores this as a part of local memory $\hat{a}_1, \hat{a}_2 \dots \hat{a}_m, \hat{b}_1, \hat{b}_2 \dots \hat{b}_m$. And it operates $\hat{c} = \hat{a}_1 \hat{b}_1 + \hat{a}_2 \hat{b}_2 + \dots + \hat{a}_m \hat{b}_m$. What does the second processor do, it is also working on the local memory which is $\hat{a}_1, \hat{b}_1, \hat{a}, \hat{b}$, but it stores a different element of the main vector as the local memory.

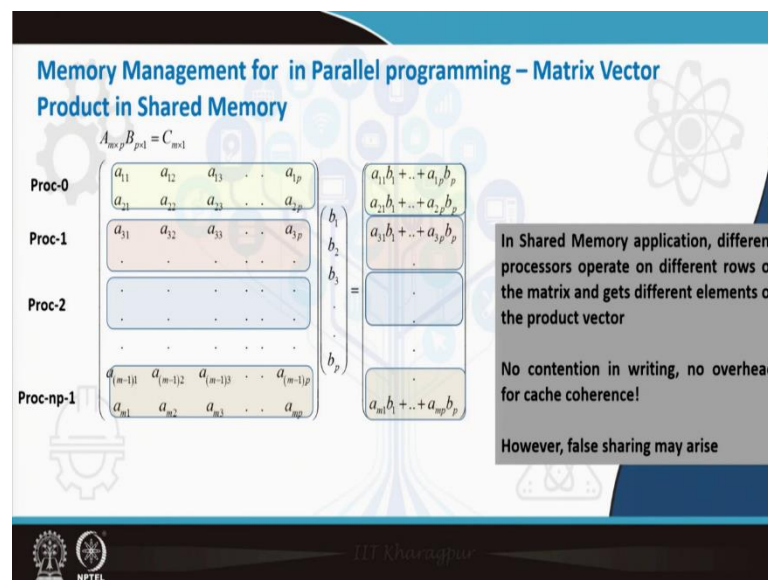
So, some of load allocation algorithm has to be run by each of the processor which can do this, \hat{a}_1 is not a_1 , but a_{m+1} , \hat{a}_2 is a_{m+2} , and so on. And it executes the same right and gets its own \hat{c} . Processor 2 similarly gets different data as \hat{a}_1 , and \hat{a} and \hat{b} and finds its own \hat{c} , and each processor gets their own \hat{c} , up to processor $np-1$.

And once this is done, now all the processor knows that everybody has finished calculating their own \hat{c} , they write the final c as sum of the \hat{c} of all the processors. So, this \hat{c} which is coming from all the processor has to be communicated to one processor or to maybe to other processor.

So, to communicate a single value after the entire calculation, it says that everybody has finished their calculations, and communicate this single value to each processor or to a master node which we will collect all the c hats and sum them up. And use the reduction algorithm like gather all the values and find their sum which is very simple distributed memory communication protocol and find the sum.

So, they are operating separately. The first one operates directly into the address space and there is an issue of cache coherence. Second one does not need to do any cache coherence since distributed memory system, but needs to map the global memory into local memory, and then do synchronization that everybody has finished calculations, and communicate the output from all of the local memories to one computer local memory and sum them up.

(Refer Slide Time: 26:38)



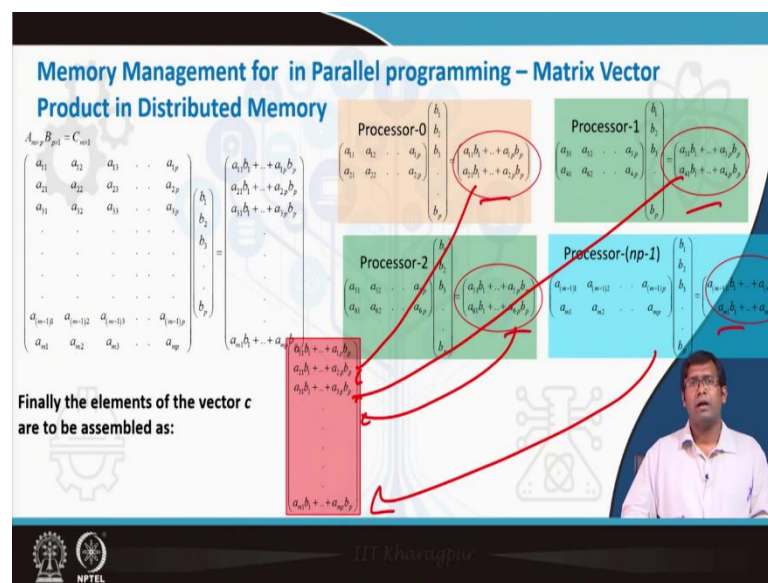
Well, we go to, one order more complexity; it is a matrix vector multiplication. So, $a_{11}, a_{12}, \dots, a_{1p}$ m x p matrix, m rows and p columns multiplied with a p row vector, and we get a m vector as the product. If in shared memory, the first processor operates on the first row of a matrix get the first row of the product vector. Second processor works on the second row of a matrix and gets the second row of the right-hand side vector and so on.

So, all the processors are working using the same b vector operating only on the rows as located to the processors and writing the rows here. In shared memory application, different processors operate on different rows of the matrix and get different elements of

the product vector. No contention in writing because they are writing at different locations of the memory. And the cache coherence is also not required if we know that we are not going to access the cache which is an access any of the others memory, we do not need to make it cache coherence, it is disjoint cache program.

However, false sharing may rise because when writing these, it might take the next line in its cache. So, the second processor may not be able to work on this, because this is the first processor is writing here. Though there is no actual sharing of the output variables among the processors, but as they are in the same address space, there can be somewhere the cache lines are being shared, not exactly the cache element, but the next elements in the in the same cache address may be shared which is called the false sharing and that can act to over it.

(Refer Slide Time: 28:32)



If we come to distributed memory system each processor has been given few rows of that, and each processor is getting a part of the vector independently. And finally, these elements of the vectors have to be communicated to one of the processors, and it has to assemble the entire result.

So, each processor is operating on its own address space. It is looking at first two rows only the processor knows that it has been given two rows it has to operate from these two rows. What is the location of these two rows, it does not know? Some processor knows

that, there is some mapping for that, but while operating it does not know it has two rows it is operating only on these two rows and writing two rows of the vector.

Finally, this data has to be communicated to one processor, and then they will be assembled all these will come to their respective positions and assembled together to form the final matrix. So, you need communication of these elements, agglomeration and mapping to the final matrix.

(Refer Slide Time: 29:54)

Memory Management for in Parallel programming – Matrix Vector Product in Distributed Memory

Processor-0: $\begin{pmatrix} a_{01} & a_{02} & \dots & a_{0p} \\ a_{11} & a_{12} & \dots & a_{1p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)1} & a_{(m-1)2} & \dots & a_{(m-1)p} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix} = \begin{pmatrix} a_{01}b_1 + \dots + a_{0p}b_p \\ a_{11}b_1 + \dots + a_{1p}b_p \\ \vdots \\ a_{(m-1)1}b_1 + \dots + a_{(m-1)p}b_p \end{pmatrix}$

Processor-1: $\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ a_{21} & a_{22} & \dots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)1} & a_{(m-1)2} & \dots & a_{(m-1)p} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix} = \begin{pmatrix} a_{11}b_1 + \dots + a_{1p}b_p \\ a_{21}b_1 + \dots + a_{2p}b_p \\ \vdots \\ a_{(m-1)1}b_1 + \dots + a_{(m-1)p}b_p \end{pmatrix}$

Processor-2: $\begin{pmatrix} a_{21} & a_{22} & \dots & a_{2p} \\ a_{31} & a_{32} & \dots & a_{3p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)1} & a_{(m-1)2} & \dots & a_{(m-1)p} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix} = \begin{pmatrix} a_{21}b_1 + \dots + a_{2p}b_p \\ a_{31}b_1 + \dots + a_{3p}b_p \\ \vdots \\ a_{(m-1)1}b_1 + \dots + a_{(m-1)p}b_p \end{pmatrix}$

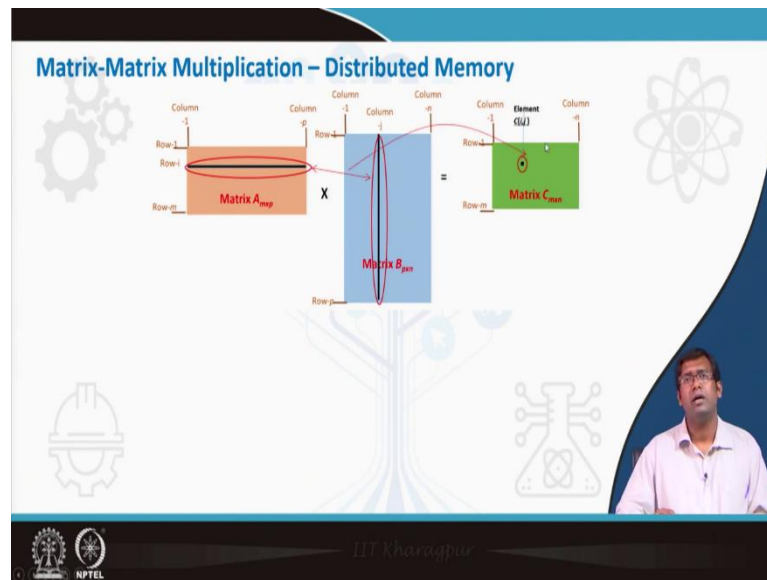
Processor-(np-1): $\begin{pmatrix} a_{(n-1)1} & a_{(n-1)2} & \dots & a_{(n-1)p} \\ a_{(n-1)1} & a_{(n-1)2} & \dots & a_{(n-1)p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1)1} & a_{(n-1)2} & \dots & a_{(n-1)p} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix} = \begin{pmatrix} a_{(n-1)1}b_1 + \dots + a_{(n-1)p}b_p \\ a_{(n-1)1}b_1 + \dots + a_{(n-1)p}b_p \\ \vdots \\ a_{(n-1)1}b_1 + \dots + a_{(n-1)p}b_p \end{pmatrix}$

Finally the elements of the vector c are to be assembled as:

Elements of the vector c (few numbers) are to be transferred from each processor
Final assembly is required
Vector b is stored as local also in all processors
Rows of matrix A ($o \times p$, $o < m$) are stored in all processors

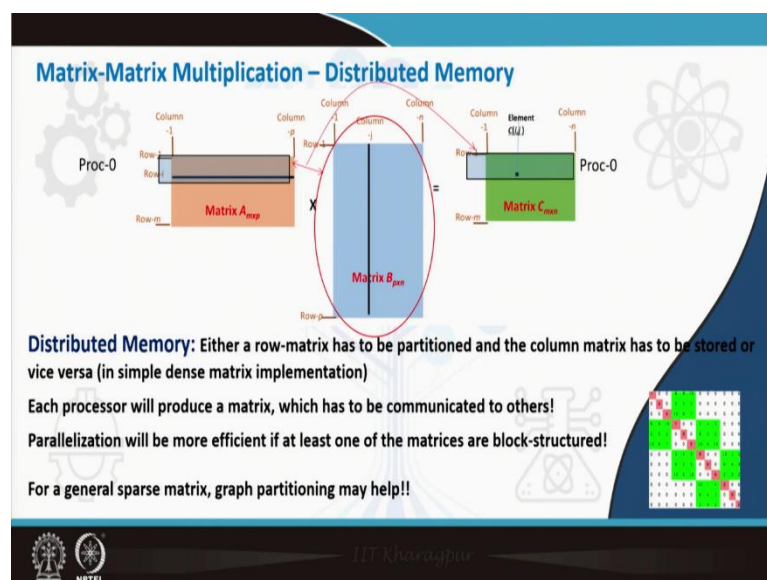
Elements of vector c which are few numbers, are to be transferred from each processor, and a final assembly is required. Interestingly this vector b is known by all the processors. So, this is though this is a common vector, but this is stored in local copies this is stored as local to all the processors. Rows of matrix A instead of being an $m \times b$ matrix, small o into p rectangular matrices are stored in all the processors. So, all the processors are storing a part of memory not the entire memory.

(Refer Slide Time: 30:31)



Now, the more complicated issue can be a matrix-matrix multiplication. So, there is a matrix m rows, there are p columns; and another matrix with p rows and n columns multiplied to get a m row n column matrix. So, for any i th row of this matrix all the elements are sequentially or coherently multiplied with the columns of the matrix B , j th column and you get the i, j th element of the product matrix C .

(Refer Slide Time: 31:04)



So, in distributed memory either a row matrix has to be partitioned or the column matrix has to be stored or vice versa. So, part of the matrix is in the row form, the left matrix is

partitioned, a part of that given to one processor; each operates on the entire column matrix, and we get a part of the right-hand side matrix. So, this matrix has to be stored, this matrix which is the right-hand side matrix that has to be stored in all the computers.

Processor 0 will do that; the next processor will get a part of the matrix and do that. So, each processor will produce a matrix, you need to now communicate a large amount of data which is not a scalar, not few elements of the vector, but it is a small matrix. So, the communication over it will be more. And each processor has to also store the entire column matrix.

So, parallelization, means, that even if you are paralyzing it, but really you cannot do anything with respect to the column matrix here, at least in the in a simple form of matrix-matrix multiplication distributed memory algorithm, because you have to operate over all the columns here and that is in a sequential manner .

But if this column is block-structured, say there is some element like this, some elements are nonzero in the column ,some elements are nonzero and most are 0 in the column, you can much easily paralyze it that some of the blocks are allocated to some processor, it knows that these two will be multiplied with this block only. And if it is not block-structured, but it has many 0 numbers which is called a general sparse matrix, a graph partitioning can give a better parallel algorithm.

(Refer Slide Time: 32:59)

Matrix-Matrix Multiplication – Shared Memory

Shared Memory:
All the products are adding to common memory space.
Contention and Cache coherence may slow down the process.
However, accessing row may be cache friendly but column accesses may lead to poor cache hit ratio!

A better algorithm can allot row and column to processors workspace and product of these row and columns may be written to the common product matrix space.

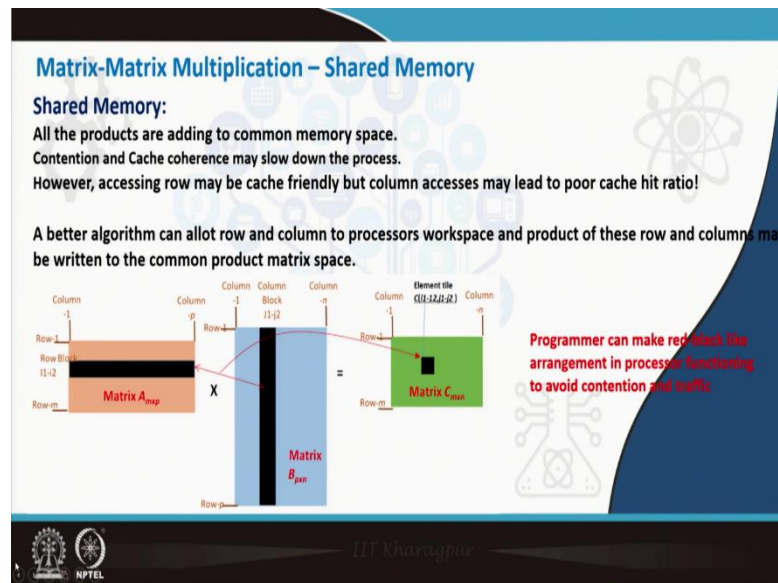
The diagram illustrates the matrix multiplication process. It shows Matrix A (rows 1 to m, columns 1 to p) multiplied by Matrix B (rows 1 to p, columns 1 to n) to produce Matrix C (rows 1 to m, columns 1 to n). A specific element tile C_{ij} is highlighted, showing it is the sum of products of rows from Matrix A and columns from Matrix B. The slide also features a small inset image of a person in the bottom right corner and logos for IIT Kharagpur and NPTEL at the bottom.

And if we think of a shared memory, all the products add to the common memory space and there can be contention in cache coherence, because there will be same cache lines which will be shared across many processors and they are writing to the same memory space. However, accessing a row, if the matrix is the cache, reads the matrix row wise, when one matrix will be accessed and on column wise of other matrix during multiplication. So, we take one column, one row multiply them, take a dot product and get one as element.

So, one column will be accessed, it is not a contiguous memory. So, there will be something related with the cache hit ratio, it will trash the cache size, because some elements of the memory have to be read and different cache lines has to be created. And across many processors it will give us an undesired performance.

Now, we can see both distributed memory as some issue in terms of parallelization though we are paralyzing it, we cannot do anything with the right-hand side matrix, in a simple version. We need some algorithm to parallelization for the right-hand side matrix also. Similarly, shared memory it leads to some issues with cache. So, one of the ideas can be that you allowed some rows and some columns to each of the processor's workspace. And when multiplying it will write back a group of elements or a tile of elements which has number of rows and number of columns here. So that this cache is filled with this element, this cache is filled with this elements they are writing it here, and in some sense what will happen like this two are multiplied and all the rows and columns are multiplied and this group of numbers will be obtain. And instead of operating in the next row and next column, subsequent row and column which will give a false sharing operate in another row give some gap take another row, and take another column get a product here. So, maybe you can take some of the rows here, some of the rows here, some of the columns here and the product will be here. So, there is no cache line sharing, and there is no false sharing, cache coherence as disjoint caches are used, and you can get better performance there.

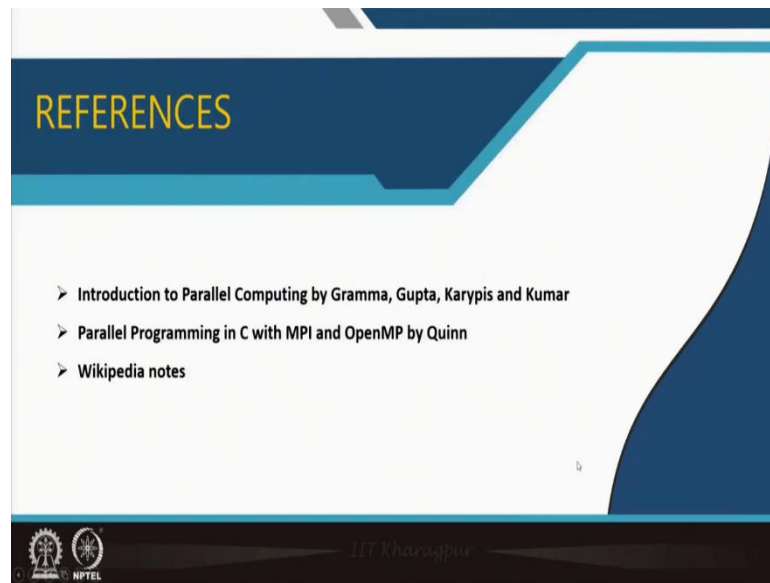
(Refer Slide Time: 35:24)



Programmer can make red black arrangements that few reds will be multiplied at one instance then the blacks will be multiplied and avoid false sharing contention and traffic, and can get better solution. So, for more complex problems, the programmer needs to think of algorithms which are taking care of some of the issues we discussed in computer architecture, parallel computer architecture and parallel computer memories. Which once kept in mind and the program is written looking into these aspects then you can get much better performance using parallel computer.

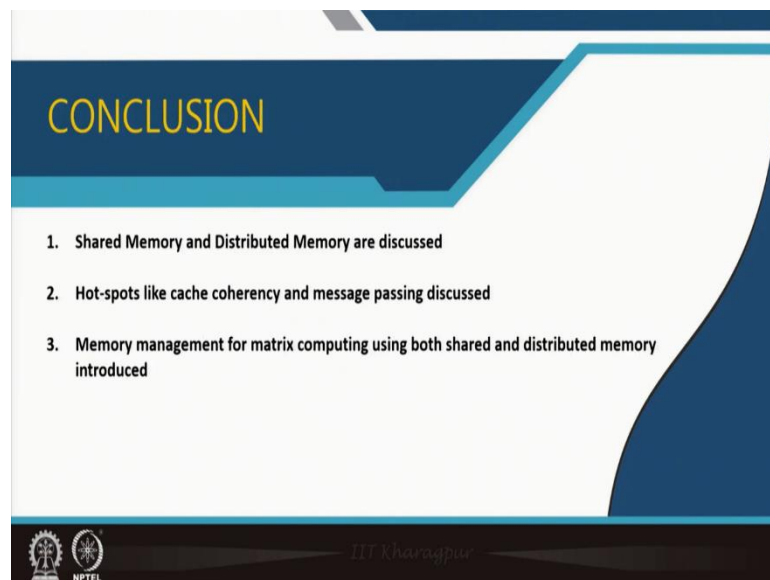
So, using a supercomputer does not ensure that you are getting best performance. You need to think from the architecture point of view and need to know the right things by doing which you can get the best performance.

(Refer Slide Time: 36:13)



Well, so we finished shared and distributed memory discussions. And these are the books along with Wikipedia notes I have followed.

(Refer Slide Time: 36:21)



We have looked into shared memory and distributed memory in detail hotspots like cache coherence and message passing which reduces the performance are discussed. And memory management for matrix computing using both shared and distributed memory, I had introduced today's class.

Thank you.