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

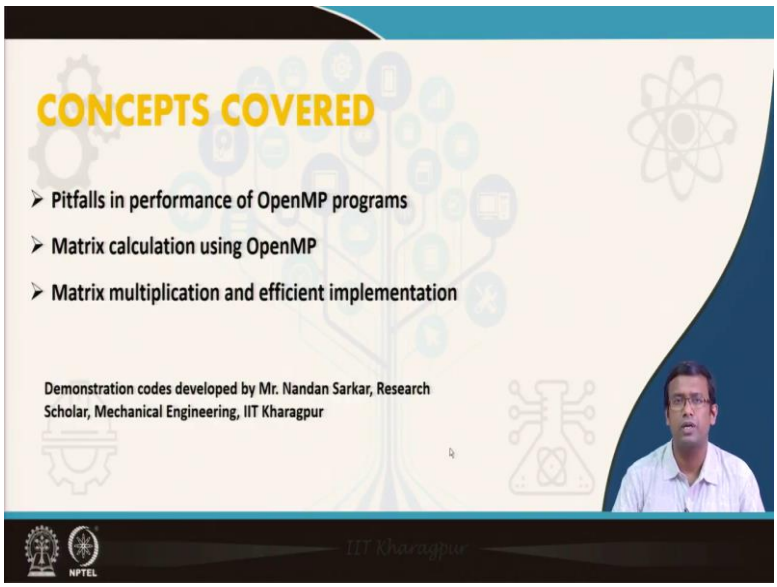
Module – 02
OpenMP Programming
Lecture – 18
Efficient OpenMP programming for matrix computing

Hello, welcome to the class of High-Performance Computing for Scientists and Engineers and we are discussing OpenMP Programming through this module. This is module 2, while discussing OpenMP programming we have mainly looked into different syntax and semantics related with OpenMP and also looked into memory management issues and task scheduling issues using OpenMP.

Now, we will take a step forward and try to see how in scientific computing applications we can efficiently use OpenMP and as we have discussed earlier. We will also see later that many of the scientific computing applications depend on matrix algebra, looking into matrices, transforming matrices, rotating matrices, and very importantly solving matrix equations.

So, the focus of this lecture will be applying OpenMP for parallelizing matrix solutions or matrix multiplications, matrix product operations and looking into the efficiency while doing so.

(Refer Slide Time: 01:32)



What we will discuss in this class is: pitfalls in the performance of OpenMP programs, because when we will write OpenMP programs the main goal behind that is we want fast computing, we want efficient solutions.

If there are certain issues for which performance of the program can degrade, we need to be aware of that, so that our program efficiency increases. Then we will see how we can do matrix calculations, while saying matrix calculations we will focus on three different types of matrix related calculations here or linear algebra calculations here.

One is vector dot product, matrix vector product, where the solution is a matrix and this matrix vector product we will again discuss that this is one of the very important applications in many scientific computing tasks .Especially, when we are solving matrix equations using iterative methods, matrix vector products are unavoidable and these are computationally complex problems also and we will see further computationally complex problem ,which is matrix matrix multiplication and we will see how we can efficiently implement the algorithms. So, it is not only the OpenMP implementation, but we will also see from the serial computing point of view, what is an efficient algorithm to handle matrix multiplication programs.

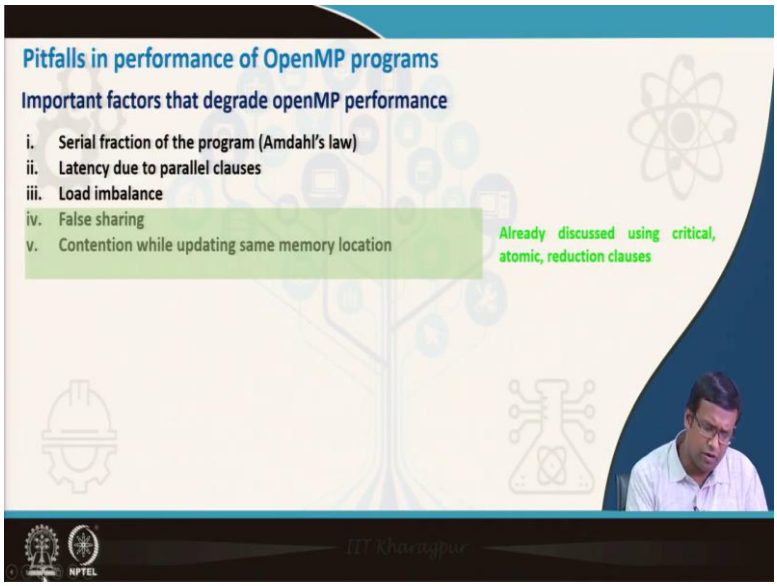
We must understand that using OpenMP, using parallelization (we will later see MPI and CUDA) to get fast solutions, if the stem of the algorithm is not efficient enough, we will not be able to utilize all the high-performance computing infrastructure efficiently.

So, it is also important that we look into different other perspectives of the program like are we handling the cache correctly; are we storing the matrix correctly issues like that. We will go through this particular issue also in this class.

And I must mention that as we discussed OpenMP programs also I will show some demonstration codes to present the case where certain methodology has to be adapted to parallelize the program or to augment the efficiency of the program and this demonstration codes , I must acknowledge that they are developed by Mr. Nandan Sarkar, a research scholar in Mechanical Engineering Department, IIT Kharagpur.

So, the course I will see most of them are developed by Mr. Sarkar and also later we will start discussing MPI. He has contributed a lot in terms of code development for this course.

(Refer Slide Time: 04:24)



Well so, what are the issues for which OpenMP performance can degrade. Some of these we have already discussed and some of this we will discuss. One is that there is always a serial fraction of the program. If you remember Amdahl's law any program cannot be completely parallelized. So, there will be certain serial parts of the program.

So, when the serial part of the program will be executed, the other processors are sitting idle, I mean that you cannot avoid that there is a serial part of the program, but in case if we try to parallelize this part also or if we deploy multiple processors or multiple threads to repeat the serial job, there will be no benefit out of that, but we will probably little overdo it, because when we say there is one particular part of the program which has to be done serially say reading a file this has to be done serially.

So, if we ask all the processors to do, that might add overheads or that might add serial computing time also, rather we will identify one processor and ask it to do this part. While parallelizing a job we have to be quite cautious about the serial part of the program and do the right synchronization so that only a designated processor works on that.

If this has not been handled correctly, this can degrade the parallel performance and also we need to look into the algorithm and see that the serial part of the program is not large enough; if a program has mostly serial part and a very small parallel component, running it in a parallelized mode will probably not help, because parallel overheads is going to kill other

benefits. So, we have to be cautious about that; we will discuss some of these issues also in subsequent slides.

Latency due to parallel clauses; so, when we call a parallel clause there is latency. The latency in the sense, if we have not done the proper load balancing, some of the processors will be done with their work, some will be waiting. So, there will be latency for the processors who have finished their job, but working for the other processors. There will be some other parallel overheads also in terms of distributing the job in terms of synchronization, etcetera. So, this is another part which can degrade the performance.

Load imbalance; this also can contribute to latency, this is of course, an important issue.

False sharing: we have looked into that though two processors are working in two different data address addresses of the two different data locations of the same memory address of the same array, but if they are sufficiently close then they are probably shared in both the processors cache and therefore, the cache coherency protocols are working over them which gives a sense of false sharing and the performance degrades.

We have looked into it when looking into the pi calculation in previous lectures and for that you need to use padding or critical constructs, etcetera.

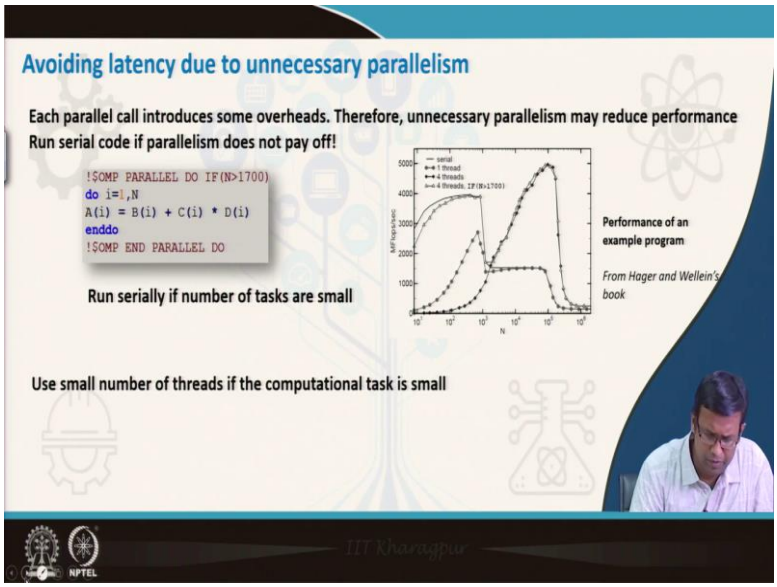
The other can be contention while updating the same memory location and this is, we will show you an example; this can be very bad. If multiple processors are working together and they are trying to update one particular memory location, then the garbage values may come out, because one has updated another and has not taken care of that update and further overwritten that.

So, this is another issue , where performance can come down, because the processors are fighting together to write on the same memory space whereas, one processor can overwrite another's write and you can get a garbage result or result without meaning.

In this particular case when multiple processors are trying to write to the same memory location, we know that reduction type of operations or atomic operations are helpful. So, we have these two clauses issues we have already discussed that both false sharing and contention can be avoided using critical or atomic constructs or using reduction clauses, etcetera.

We will rather focus this part of the discussion in understanding this part than what to do when a part of the program is serialized or what to do when there is certain latency due to parallel clauses, etcetera.

(Refer Slide Time: 09:0 9)



Avoiding latency due to unnecessary parallelization :For example, I have a program which is mostly serial, but there is some small parallel component or I have a program which is a small program. I mean whose computational cost is small. So, if we want to divide it up across multiple processes, then the distribution and synchronization overhead will be so high that the benefit will not be much. Whenever we are writing a parallel clause, it is adding some overhead. So, if we are unnecessarily parallelizing a job say, I have to operate over five numbers and I have five processors with just A + B type of operations. The distribution synchronization, the overheads will be so high that, (because each processor is just crunching one number) this benefit will be overshadowed by the parallel overheads. So, unnecessary parallelism may reduce the performance. We need to see whether it is necessary to parallelize the program and when it will be necessary to parallelize the program. If not then we will serialize the serial program. When will it be necessary to parallel the program; when the number of jobs are large.

In case, I have a very small number of jobs say, I am doing some operation for 1700 numbers well, (I will show you some examples in some in after few slides only) that this operation like a is summation of three numbers, any processor for a small number of iterations any processor

can do it very fast. So, if we try to divide across multiple processors then it will take a large amount of time.

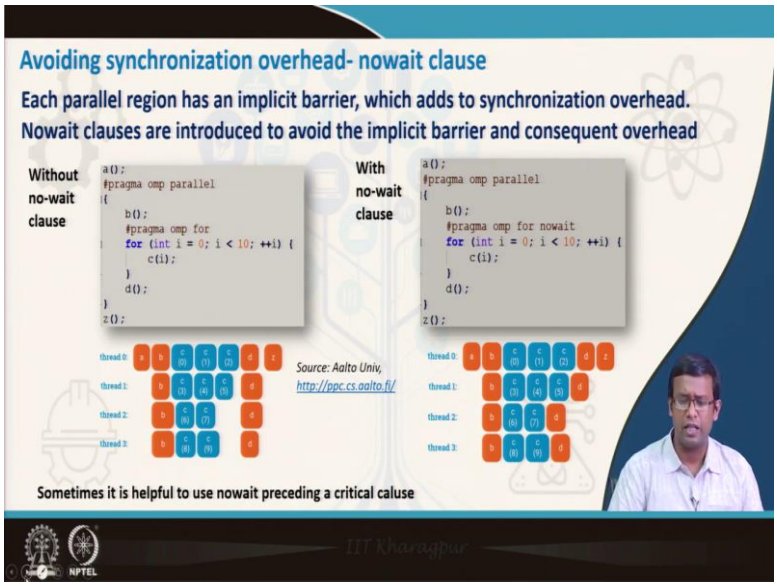
So, if we say we are doing something here that we are using a clause, the number N if the number of iterations is greater than 1700. This 1700 is kind of arbitrarily chosen, but it tells you that the number should be large enough and also when each processor is reading it does not overflow the cache of the processor. There should not be many cache hits or cache misses. If the number N is less than 1700 then this parallel loop will not be executed then this job will be executed as a serial process. If the number is more than 1700, only then the multiple threads will be launched and different threads will take part of different iterations .This is taken from Hager Welleins book ,some example addition of three numbers, three arrays is done, and we can see as the number of increases, serial performance initially increases, because it can utilize the entire cache line and then around 1000 serial performance drastically drops and then(because the number of iterations are large takes a large amount of time)it keeps falling, but in case we do that up to 1700, where serial performance is good, we will run it in serialized mode, after 1700 then it will go and four threads will be launched.

Then we will probably get the best performance, because when the number is substantially small it is running as a serial program, but now one can tell that still why there is little difference in performance, because we are using an if loop here. If it is a serial program if these two lines are really commented out it's a serial program then this if loop would not have been there and there would have been more calculations then this.

There are some small issues that let us not focus on that, but the idea is that if we run this program throughout in 4 threads right, it gives much smaller performance, certainly a number less than 1. The distribution synchronization thread, forking thread, destruction, etcetera takes a substantial amount of time and adds overhead. So, this performance is not right.

So, here the part up to which running in multiple threads is not beneficial, we run it as serially and when running multiple threads are not beneficial; when the task is small. So, if the given job is small, it is more judicious to run it serially if the given job is large then we can launch multiple threads.

(Refer Slide Time: 14:13)



If the computational task is small you either run parallely or use a small number of threads. Then another part is latency due to synchronization issue. Each parallel region is somewhat synchronized; that means, once threads are launched, they will finish their work and then there is a barrier each thread will wait till others have finished till that part of the job and then threads will be destructed and the next parallel region will start or next part of the codes will follow.

So, each parallel region has an implicit barrier, where each thread has to wait for others to finish their work and this adds a synchronization overhead; that means, this is adding a latency, which is an overhead to the threads.

In case I have a program where output of one parallel region is not important for the next parallel region, I can use a no wait clause which will break this synchronization. Threads do not have to wait till the end of that particular parallel region and go and go to the next part of the work and that will consequently reduce the overhead.

Say this is an example which is without no wait clause, a simple parallel regime. Initially the parallel regime is launched. So, each thread updates their private memory b, a is a shared memory variable and then each thread updates the variable b as given to them. They call a function say and update some variable and now, in the for regime there is a loop in which sum variable c is updated and then this for loop ends then a variable d is updated by the threads and then this parallel region ends at this point only all the threads are free and they can work on the part z.

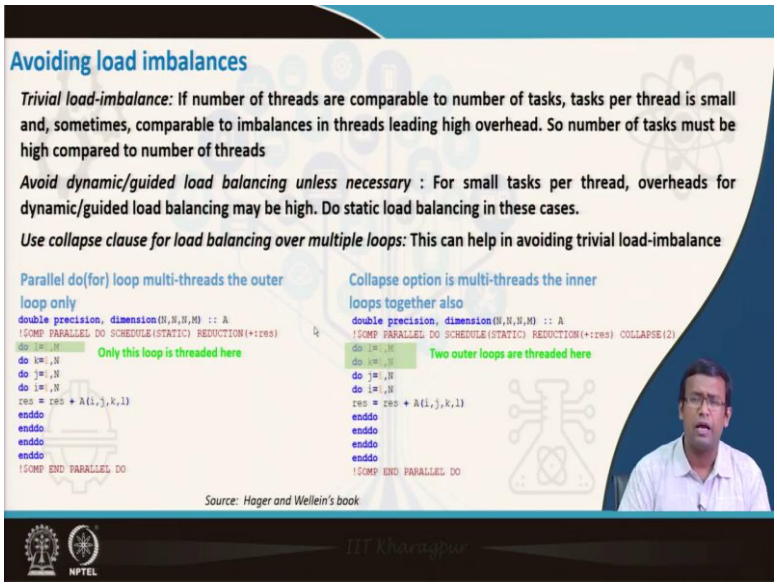
If we look into the execution pattern say for thread 0, because this is outside the parallel region thread 0 is only active here, a is updated at thread 0, then the very the part b is done by all the threads, then for the part c this is an iteration and say 4 threads are active over 10 number of iterations. So, few threads work for 3 iterations, few threads work for 2 iterations, but this is a particular parallel construct. So, unless this construct is over, unless this parallel region is over the other threads are waiting, till the threads which have more work they have finished their job and then once they finish everybody does the part d and then the again all the threads are destroyed and part z is done.

Now, we assume a case where the output of this function c is not required by the functions d and z. What we can do instead of allowing this wait time by these two threads we can use a no wait clause here; that means, once these two threads jobs are done, they can directly pick up the task d. So, thread 2 and thread 3 is not waiting for thread 0 and thread 1 rather they are finishing their own job, we are destroying the barrier there.

So, this is done to reduce the synchronization overhead ; if we can efficiently design the algorithm, in next part probably we will give more tasks to say thread 2 and thread 3 something like that these things can be planned or we can put another for a loop where more tasks are given to thread 2 and thread 3. So, they can pick up and we can say save in terms of time.

This figure is taken from Aalto Universities, high performance computing web page. Sometimes when a variable is updated using critical this no wait clause can help, why? Critical means this is inherently a sequential part each thread is executing that part of the job and other threads are waiting for that. In case we use a no wait and make them free, they can go into the critical part and these two threads are still working, but by the time they have finished writing the critical variable, they will be finished and they will take care of the job. So, we can save more in terms of time.

(Refer Slide Time: 19:0 8)



Avoiding load imbalances: if the number of threads is comparable to number of tasks, task per thread is small. We are doing load balancing and we are trying to give almost an even number of tasks to each thread. But if I say I have 10 tasks and 8 threads, so each thread is getting 1 task and there will be 2 tasks. So, two of the threads we will get 2. So, the amount of tasks given to each thread is small and the wait time for each thread, because others have not finished their job some threads have 2 tasks is also comparable to that which is not good.

In case you have a smaller number of tasks compared to the number of threads, I mean you do not have a large number of tasks compared to the number of threads sometimes it is comparable to the imbalance between the threads. Therefore, the latency and overhead will be high as compared with the computational time. A thread is doing one unit of task ,it is waiting for another unit which is bad, but in case I have 8 threads and 731 tasks then each thread is getting 91 tasks or 92 tasks.

So, the thread that less tasks they have got 91 tasks and waiting for 1 task by some of the other threads. The compared waiting time is small, the overheads are small in that particular case. Trivial load balancing we will try to avoid the fact that the number of threads is comparable to the number of tasks. We will try to assign a number of threads, which is much less than the number of tasks.

Avoid dynamic /guided load balancing unless necessary; for small tasks per threads if we use static load balancing ,load balancing is once done and fixed, you do not need any complicated

algorithm to do load balancing, but if you are using dynamic and or guided load balancing there is some algorithm there , the compiler has to do something else to do the load balancing and that will add to overhead.

In case we have a smaller number of tasks, then this overhead can be large. So, if it is not absolutely necessary do not do dynamic or guided load balancing. In certain cases, for handling large jobs they can be very helpful, but in general if you are working with small or medium complex problems do not use these load balancing techniques.

Use collapse clauses for load balancing over multiple threads. This can help in avoiding trivial load imbalance and this is an important aspect. We will see some of these issues later on matrix calculating. In case we are using a parallel do loop over multiple loops there are multiple loops and we have we are not doing nested parallelism, we are doing simple parallelism with one parallel do loop.

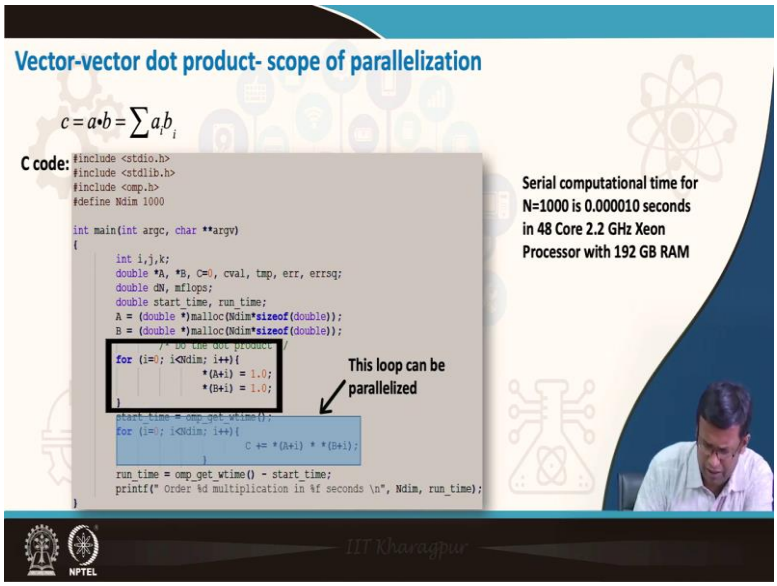
So, what happens when these nested loops are parallelized? OpenMP by default takes the first outer loop and parallelizes the iterations in the first outer loop. The internal loops are the same for each of the threads. So, threads are assigned only to the first iteration loop.

Now, here the number 1 to M can be not a very high number, can be small say 1 to M is 10, 1 to N is also 10,therefore, there will be total 10 to the power 4 iterations; however, if we try to parallelize it, each thread will get a fixed load balancing on the outer loops, which can give less number of tasks to each thread. The tasks are heavy; however, the imbalance can be more, because the outer loop tasks are only threaded therefore, there can be more imbalance there.

So, one idea can be an imbalance case is not more, rather an imbalance is large comparable to the given number of tasks, that if we use a collapse of level two what will happen? That first two outer loops will be collapsed and both of them will be threaded. So, threads will be launched not only in the outermost loop, but also the inner loop will be threaded; that means, 1 thread it will take say first ten, first five of the outer loops inside each of the outer loop there will be threads which will take care of say first set of the inner loops.

So, collapse can do threading over the first two outer loops and this can sometimes give you less overhead, because it can avoid the trivial load imbalance problems.

(Refer Slide Time: 24:20)



These are the few things apart from that we know about false sharing and trying to write in the same memory, parallel write issues contention while writing to the same memory location issues, but apart from that this load imbalance problems, specially trivial load imbalance and the serial part of the program and latency due to some threads waiting for some other threads who have not finished their work can also degrade the performance.

Now, now we have to be cautious and utilize OpenMP constructs efficiently and smartly so that we can avoid the overheads there and can get better performance out of the program. We come to the linear algebra applications and one of the basic applications is a vector vector dot product. If you can understand that if we do a vector vector dot product each element of the vector is multiplied with corresponding element of another vector and they are summed up and finally, we get the sum.

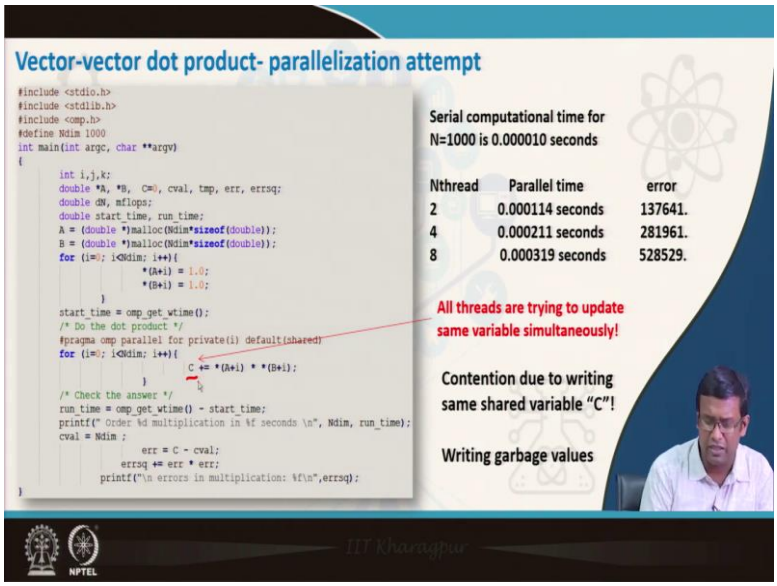
So, in case we have a vector of dimension 100, then there are 100 into 100 and addition of 100 times so 200 operations. So, usually the number of operations in a vector dot product is not large.

Therefore, in some cases where you are dealing with vectors which are of the order of millions or 100 million or billions , apart from those cases you really do not like to parallelize the vector dot product program simply due to the fact that it is not computationally much complex. Sequentially, it can be handled quite well and we will see some of the examples, but in case you have a very large dimension you may like to parallelize this.We will look into the

sequential program. What is happening in the sequential program? See, here the dimension is given 100. So, two vectors of size 1000 dimension are declared and they are initialized to be 1 and then we do the dot product of each of the vector elements. A + i gives different ith elements of the vector and they are added and written to the vector variable C. So, in a 48 core 2.2 gigaHertz Xeon processor with 192 GB RAM it took 10 to the power - 5 seconds for N = 1000. It is a serial computational time in a single processor.

So, you are not using 48 cores, we are only using 1 core here when we execute this program, because this is not an OpenMP program I have added omp. h only for finding out the time we are using OpenMP functions, but it is not an OpenMP program. We are neither using 192 GB RAM, because it is a very small program. If we have to parallelize what we will do, we will parallelize this for loop ,this for loop is the dot product calculation. So, this is iterating over i = 0 to i = the dimension of the vector and in case we want to parallelize it we will include omp parallel for construct here and that will launch multiple threads and parallelize. This loop can be parallelized. This also can be parallelized, but I am not looking here, because it is initializing.

(Refer Slide Time: 28:14)



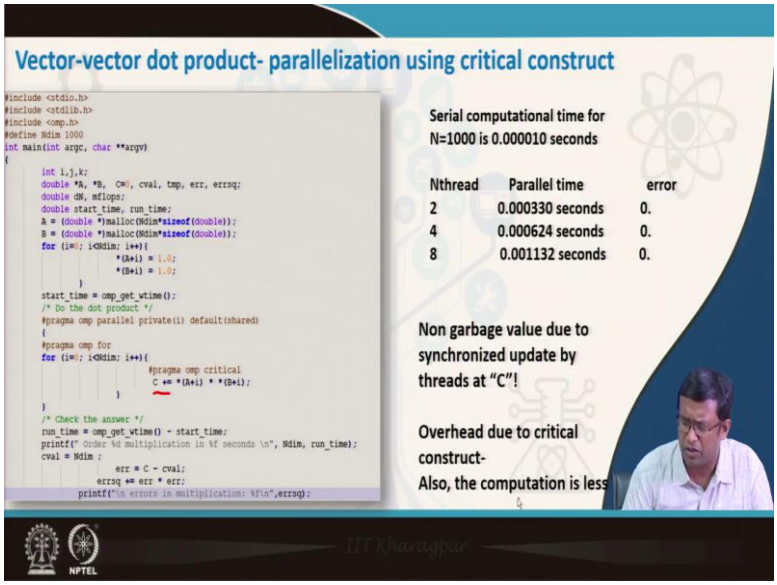
So, now if we try to parallelize it what we will do we will write pragma omp parallel we will look into the dot product part pragma omp parallel for private variable is i here and all other variables Ndim, C ,A all are shared (default is shared) and parallelize it .To check our parallelization right for that what I do, because this these vectors have valued 1, so, their dot product is basically 1 *1 + 1 *1 so on. If the number of elements is N value of the dot product

is N, we will check whether the difference between the dot product calculated by the program in a parallelized and the actual dot product C actual value which is N (dimension of the vector) are the same and what is the error we will write that.

So, when we parallelize it and serial computational time is 10 to the power - 5 seconds. In 2 threads, the computing time is 10 to the power - 4, the fourth thread's computing time is 2 into 10 to the power - 4, 8 threads computing time is more. As we are increasing threads and our computational time is increasing and error is huge, we are getting some garbage value, because the error from the what actual value should be C val the error between C Val and C is huge and as we are increasing the number of threads the error is increasing.

So, what happened wrong? Well, we try to update the variable C by all the threads concurrently. It is the same variable location where all the threads are trying to write. So, all threads are trying to update the same variable simultaneously, therefore, contention due to writing the same shared variable C happened and it ended up in writing garbage values well. So, what will we do? We will not try to update it simultaneously; we will put a critical construct here.

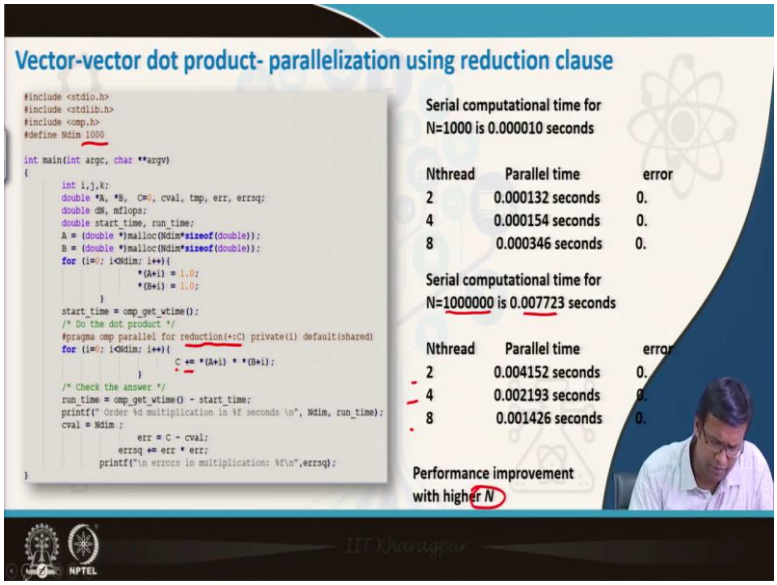
(Refer Slide Time: 30:21)



So, that is exactly what we have done here. Here, this is parallel with the for loop, we have launched it separately (you can do it other ways also) and then when we are updating the variable C, we put pragma omp critical. So, each of the threads can execute this particular operation serially.

What happens the error reduced to 0, but the parallel time is (earlier the parallel time was 10 to the power of - 4 and now the parallel time for 2 threads is 3 into 10 to the power – 4) increased, the computing time increased as we are increasing number of threads the computing time is increasing and why is it so; because again critical construct tells each of the threads to wait till others can write it. So, the time is increasing.

(Refer Slide Time: 31:39)

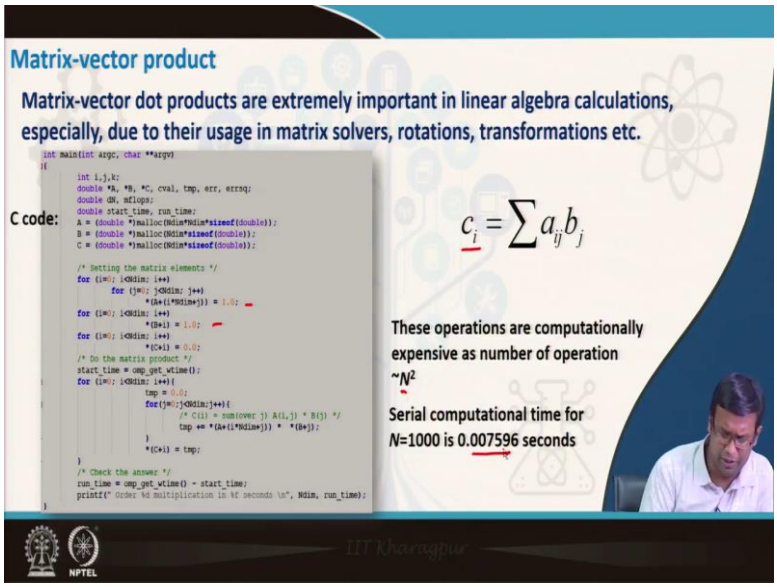


However, error is reduced, because the contention has been reduced. No garbage values due to synchronized update threads at C, but overheads due to critical construct. Well, the other way also, because the computation is less compared to the overhead the time is increasing. I told you that this I mean for 100 size vector computation will be less, so, overheads can really kill the problem. Well, the other way out will be, because we are updating the variable C and its addition, we just add a reduction clause. Reductions are very optimized clauses.

So, what happens? When we run reduction clause computing time is reduced, but again it is not comparable to the serial computational time, it is more than serial computational time, error is 0 which is fine. Still computation time is more than the serial time in parallel, because the vector is a very small, the computational task given to it is quite small, but if we increase the vector to be 1 million and then serial computational time is 7 into 10 to the power - 3 and we can see with the reduction clause with increasing number of threads computing time is reducing.

So, because the task is more here, parallel performance is better. If the task is small, do not parallelize it, if the task is large, because we are doing it for 1 million size vectors then there is some benefit of parallelizing it, and program performance improves with higher values of N.

(Refer Slide Time: 33:14)

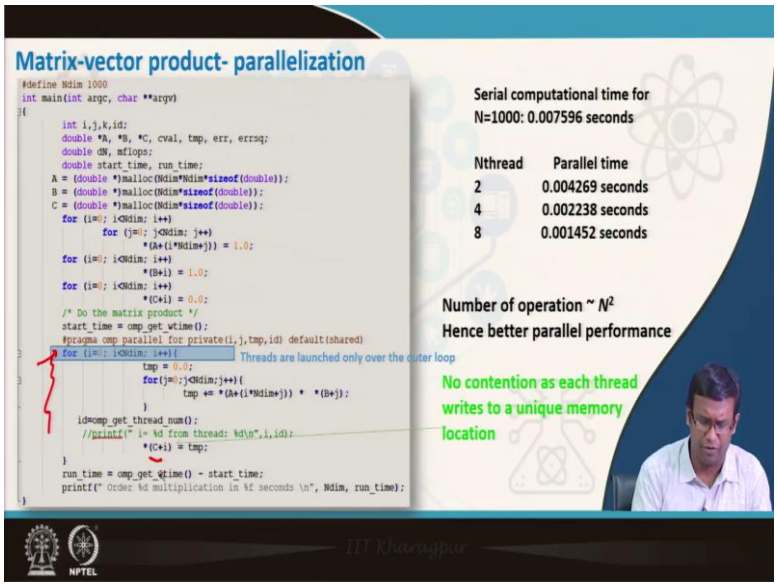


Now, we come to a more important issue that is matrix vector products. Matrix vector products are extremely important in linear algebra calculations. In matrix iterative solvers, we know about it, in preconditioning in many applications we use matrix vector product several times, in matrix rotations, matrix transformations, the domain of scientific computing is infested with matrix vector products. These are unavoidable and, if we look into matrix vector products a product between a matrix and vector gives us a vector each element of the vector comes as a summation of all the elements of the row of the matrix multiplied by the corresponding elements of the vector.

So, for each row if there is a N X N size matrix and N dimensional vector, each row there are N operations. If there are N rows there are total N square operations. So, numbers of operations are large therefore, parallel parallelization benefits will be good here for large values of N.

So, we have written a matrix vector product here it initializes the matrix with all elements 1, vector with all elements 1 and then multiplies them and writes it here and the computational time is 7.5 into 10 to the power - 3.

(Refer Slide Time: 34:50)



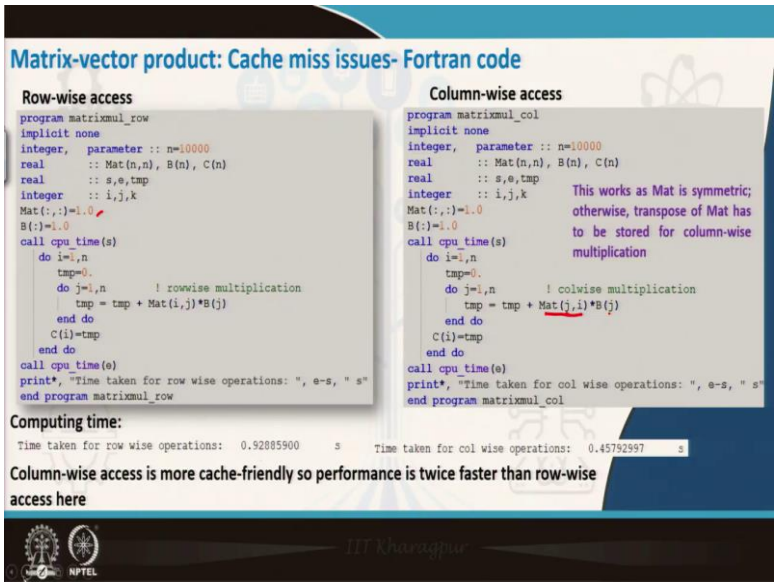
Now, if we only have to write a parallel construct before the matrix vector product output and which of the iteration loops it will parallelize? It will take the outer iteration loop and parallelize it. Also, you do not need to worry about false sharing or even you do not need to worry about contention while writing it, because each of each of the threads will update a different element in the product row.

There will be some false sharing, because you are always doing a large number of computations only at the ends of the iterations of each thread. There might be some sharing ,might not be also. This is not severe here.

Well; so, threads are launched only over the outer loop and if we see the serial computing time which is 7.5 into 10 to the power - 3, parallel computing time improves as we are increasing the number of processors the computing time reduces.

As the number of operations is of the order of N square therefore, the total tasks are more therefore, we get better performance here. If we increase the size from 1000 to something more, we will get even enhanced performance and there is no contention, because each thread is writing to a unique memory location. They are not trying to update the same memory value.

(Refer Slide Time: 36:27)



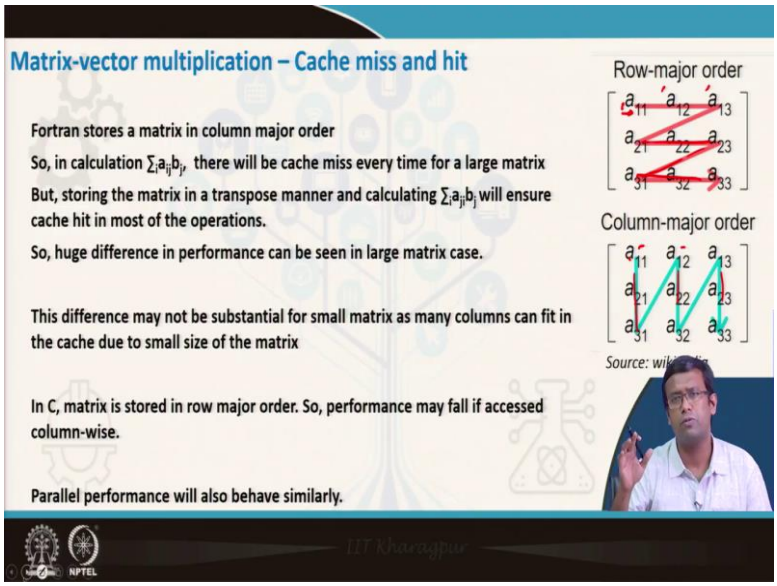
Now, we look into a Fortran program. So, what was in the C program that each element of the matrix row is multiplied with the corresponding element of the vector. Now, if we do the same thing that each row of the matrix is multiplied by the corresponding vector element, we will see computing time for this matrix is 10 to the power 5 is 0.92 second.

Instead of doing the row wise multiplication, if we do it column wise that all row elements will be multiplied by the column and, because this matrix we are defining everything one there is a symmetric matrix this will be equivalent here if not symmetric then instead of b we have to store it as b transpose.

We will see the time is much smaller 50 percent of that and why is it and this, there comes certain cache miss issues. When we are accessing the matrix column wise, in Fortran the matrix is stored in a column wise manner.

We are utilizing the cache more efficiently, so there are less cache misses. When you are accessing it row wise as the matrix is stored column wise, every operation is giving you a cash miss therefore, the computing time is almost double for row wise multiplication.

(Refer Slide Time: 38:02)



If we focus more , the 2 D matrix is a 2 D array, but when it is stored it is stored only by the address and the size. So, in C the compiler stores the matrix as the first location and then maps everything as a contiguous array.

So, the first row is stored, then the second row is stored, then the third row is stored ,so on, it's stored like a contiguous line and this is called a row major order. In Fortran it is rather stored in column major order that first column is stored, second column is stored, third column is stored, so on.

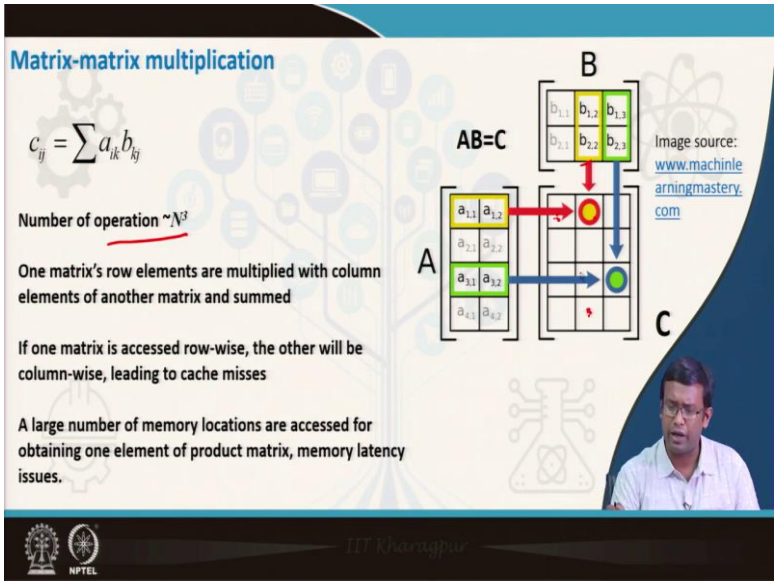
So, when we access the first elements of the first row they might be in the same cache, but in Fortran when we are trying to access this and if there is a large matrix, they are not in the cache line as the cache line contains the columns only.

Therefore, as Fortran stores the matrix in column wise in calculation of this, there will be cache miss every time for a large matrix, because if this is the cache line and cache starts from a_{11}, a_{21}, a_{31}, a_{N1} up to the cache, a_{12} is never on the cache line. So, every time there is a cache miss therefore, while looking into Fortran we have to look into the column major order or we have to go column wise. By storing the matrix in transpose manner and calculating $a_{ji} b_j$ instead of storing a transpose that will ensure in Fortran that we are going column wise direction and we are ensuring cache hit every time.

Therefore, a huge performance difference can be seen if the matrix is large, if the matrix is small the entire matrix might fit into the cache, if you are just talking about three by three matrix or if you are talking about million by million matrix then the performance issue will be significant. Now, for a small matrix it will not be substantial as the columns can fit in cache.

In C it is stored in row major order. In case we use the try to store it as b transpose and multiply it in C then there will be a problem, because the matrix is stored in the row major order. Therefore, the parallel performance will also behave similarly. In C we have to access as a row major order, we have to access the matrix row wise, parallelize it similarly in Fortran, you have to access the matrix column wise and parallelize it similarly, else there will be degradation of performance.

(Refer Slide Time: 40:46)



Now, we look into matrix matrix multiplication. These are not as abundant as matrix vector multiplication in scientific linear algebra applications; however, there are many cases where you have to do matrix matrix multiplication and the number of operations are quite high (N^3).

If the matrices of the size N x N matrix matrix multiplication will take N^3 operations which is huge; if we think of direct solution of matrix equations, we have to use matrix matrix multiplication like; Gauss elimination or inverse of matrix then we need matrix matrix multiplication the operations are huge.

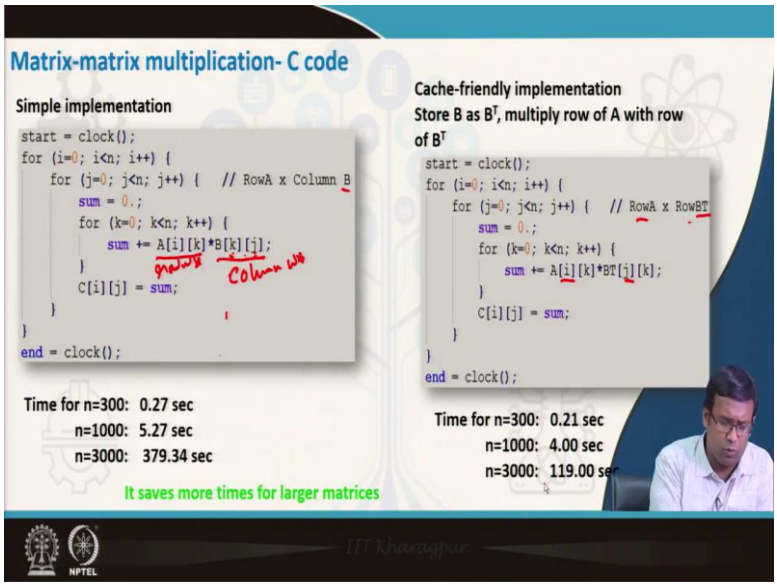
So, it is always beneficial to parallelize it, what happens is that we pick elements of one row, multiply it with the corresponding elements of the column of the other right, multiplier and sum it up and find the product.

So, always one matrix is accessed in the basic algorithm row wise, another matrix is accessed column wise. So, be C, be Fortran ,it is either row major order or column major order is a storage. So, if we try to access row wise and the other one column wise, the other one will always give you a cache miss and so, we have to do something for that .

Also, while finding out one element in the product matrix we have to take one line of the right matrix, one line of the left matrix and multiply them and find the product.

So, a number of memory locations are accessed and only one element of the product matrix is obtained, but we can see if we take this line and this row of a matrix, we will get this value. So, this row is accessed many times for different elements, but reading a row means taking a number of elements from the main memory to cache. We are doing this exercise several times and not reusing that part, so there will be memory latency issues, because a large number of memory locations are accessed for getting one element. Again, for getting another element many of the same memory locations will be accessed and this element will be obtained. Again, for getting this element these memory locations will be accessed and we will get this element, but this and this they are two different iteration loops, may be different threads are responsible for that. But essentially the same exercise for fetching the memory is done several times for the same memory location for getting different computing. This is one issue in terms of memory latency.

(Refer Slide Time: 43:38)

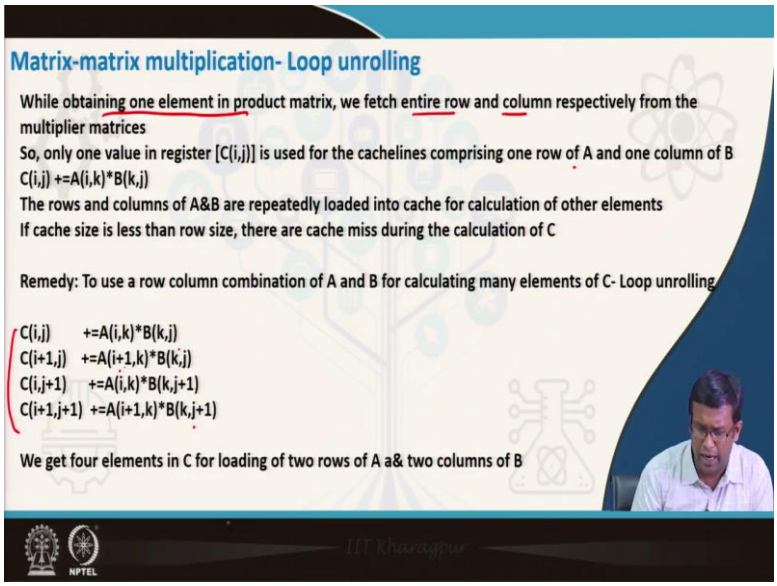


Well, this is the simple; this is the simple implementation that each element is the c_{ij} product matrix is obtained by multiplication of the row of the matrix A and column of the matrix B and the time for n= 300 is 0.2 second and 1000 is 5.27 seconds and 3000 is 379 second. So, you can see it is almost cubically increasing. This is not cache friendly, because this is a C program and A is accessed row wise which is cache friendly but B is accessed column wise. So, for one particular row element different columns of B are being accessed. So, j is fixed, column id is fixed, and different elements of the columns are being accessed and they are not in the same cache line. So, this is a non-cache friendly example.

What we can do instead of B we can take B^T and multiply it. We can store B as B^T or rather we can store the B^T directly and multiply row of A with row of B^T.

So, we have stored B^T instead of B and multiplying row of A and row of B^T (row of B^T is column of B doing the same calculation, but storing it differently so that both the matrices are now accessed column wise).We can see the difference in time for 300 its 0.21 second. For a large matrix more than one third of time has been saved while doing a column wise access. It saves more time for larger matrices.

(Refer Slide Time: 45:26)



This is one part that we have to access in C we have to access both the matrices row wise, in Fortran we have to access both the matrices column wise and even if we parallelize or not parallelize it this performance difference will be there ,even with parallelization we will get better performance if we do cache friendly access.

The other part of what I was telling is that for obtaining one element in the product matrix we fetch the entire row and column respectively from the multiplier matrix. So, each element requires one row reading, one column reading.

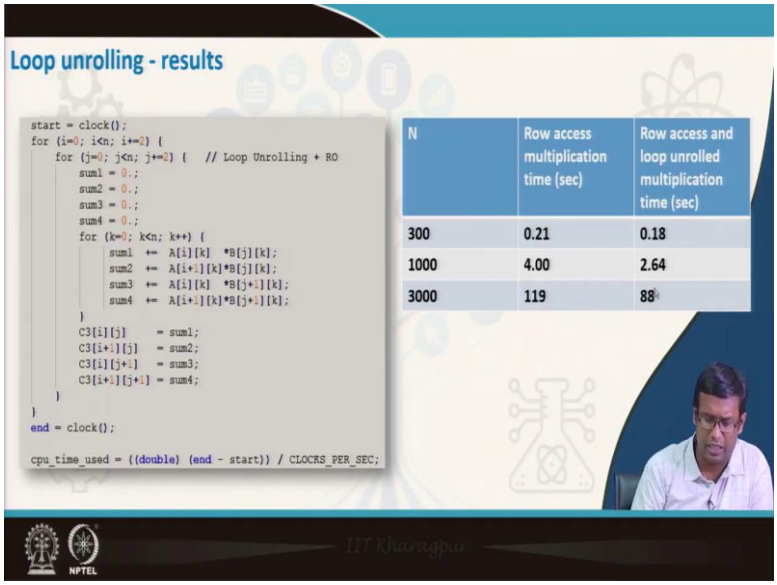
One value of the register is used for the cache lines comprising one row of A and one column of B. Let us see that we are calling it in a cache friendly manner instead of B, we are calling B^T. The rows of A and B are repeatedly loaded into cache for calculation of other elements. If cache size is less than row size there will be cache misses during these calculations. Well, if we can unloop the row ,that row column combinations ,few rows and few columns are read and all the calculations using these row columns are done once.

So; that means, instead of only calculating one element in the loop of the C matrix we will calculate multiple elements of the C matrix and we will do that C (i ,j),C(i + 1, j), C(i, j + 1), C(i + 1, j + 1). So, what are we doing here? We are reading ith row of A Bth column of j ,i + 1th row of A and B j + 1th row of k .

So, we are reading ith row ,i + 1th row, jth row, j + 1th row. We are reading two rows of A two columns of B and doing four calculations of C elements here. We are reading one row of A, one row of B, and only doing one calculation.

Here, we are reading two rows, two columns, four cache readings we are doing and doing four calculations. So, more calculations are done as compared to the memory access.

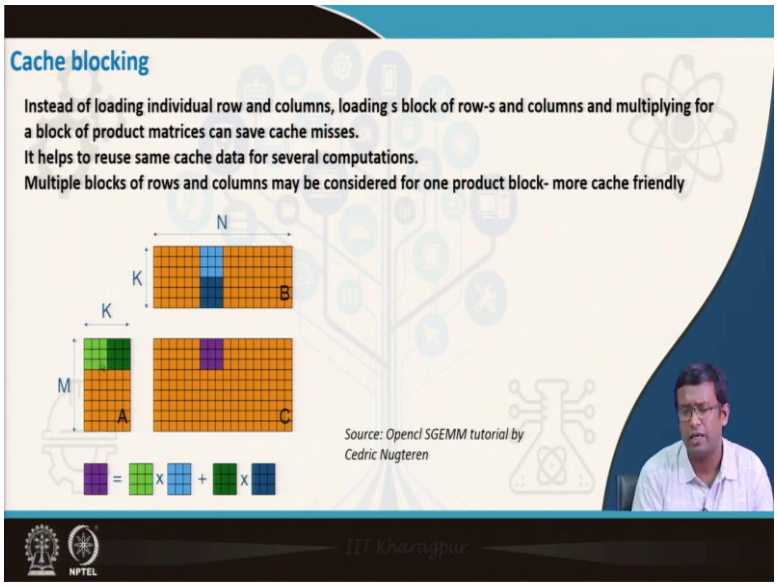
(Refer Slide Time: 47:46)



We can see the performance that when we run this code without simple row access that we are accessing the matrix row wise. That is what we are doing for both the cases. It was point 2 1 second.

Now, once we unroll the loop for, we are doing it for four elements in the C matrix the time is less than 0.18 second. For larger matrix row access time simple calculation time using row access pattern is 119 seconds, for loop unrolling it is 88 seconds.

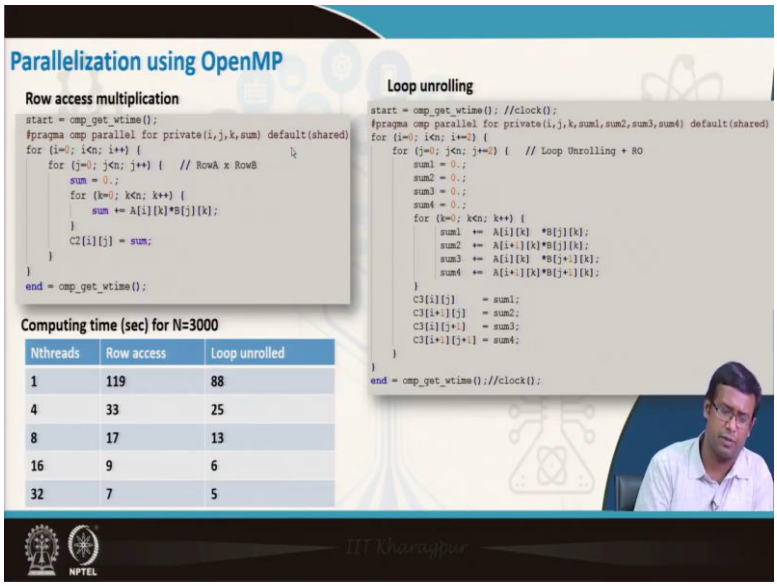
(Refer Slide Time: 48:23)



There is another idea of cache blocking that instead of reading four row, four column and calculating only for four elements read a block of rows, read a block of columns and do all the calculations pertaining to that and it helps to reuse the same data instead of loading individual row column ,load a number of blocks of rows and blocks of columns and multiply for a block of the product matrix.

Instead of loading the full row block, full column block, you can load part of the row block, part of column block depending on the cache size. So, fill the cache with contiguous memory and you have to map the memories contiguously so that you can do it correctly and then calculate one part of the product matrix. You have to map the memory continuously if the size of the block fits the cache correctly then you can get much faster performance, but now with simple row access multiplication which is accessing both A and B row wise.

(Refer Slide Time: 49:30)

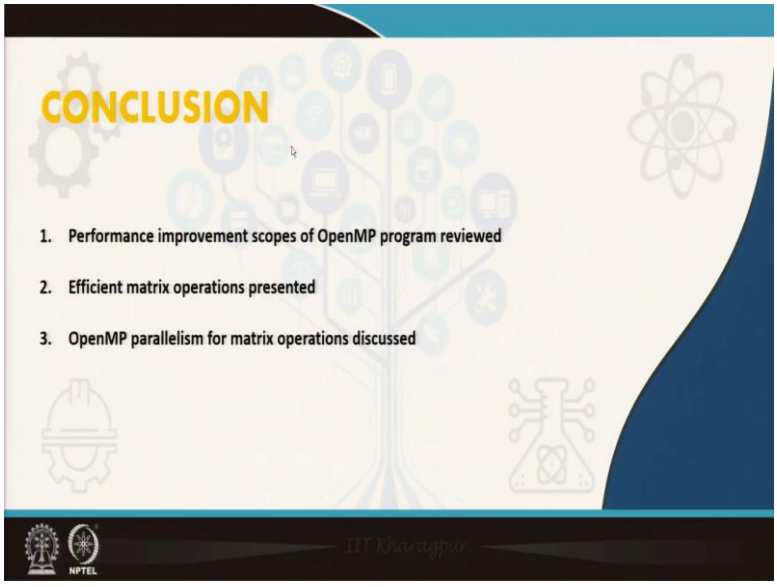


So, instead of B we are doing with B^T and here we are doing row access along with that we are doing loop unrolling and we can see the performance with parallelization. Again, parallelization will be only in the outer loop ;it will be simple as a matrix vector product.

So, the computing time which was for 1 thread 119 second, for 32 threads it comes to 7 seconds, for loop unrolled it comes down to 5 seconds. So, parallel performance is also my better when you do loop unrolling part and parallelization is very simple. Here, you only have to put a parallel for pragma mentioning the right private variables and the shared variables and this is one of one way of using the registers efficiently and using cache friendly access, getting efficient implementation of matrix matrix multiplication.

(Refer Slide Time: 50:41)

(Refer Slide Time: 50:46).



So, these are a few references which we have used for these slides, this lecture. We looked into performance improvement scopes of the OpenMP program by avoiding trivial, load balancing, and avoiding synchronization overheads, etcetera.

We looked into how efficiently matrix operations can be done, especially looking into the cache friendly access pattern. Row wise access for C matrix, column wise access in Fortran programs, these are cache friendly access patterns and also loop unrolling helps in matrix matrix multiplication and looks into OpenMP parallelization of matrix operations.

Basically, for dot product OpenMP parallelization is little critical you must not do it unless the vector is large, but for matrix vector and matrix matrix product parallelization is straight forward. Your main algorithm must be cache friendly and efficient enough. Parallelization is only by calling a pragma parallel for loop. This for loop will be active as it will launch threads only over the outer iterations and it gives good results.

So, these are some scientific computing related applications we have discussed here and now, we will from the next class we will look into the MPI based program and go to the third module of this course.