High Performance Computing for Scientists and Engineers

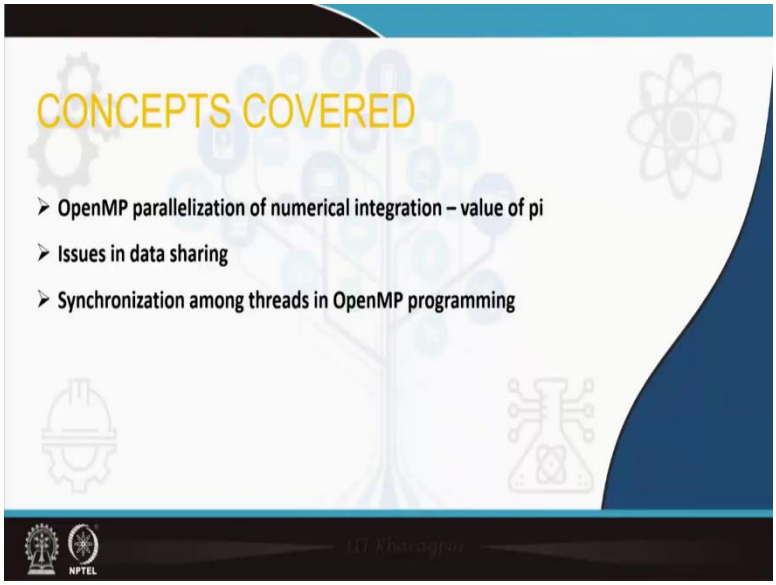
Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 02
Open MP Programming
Lecture – 17
Data Sharing and Synchronization

We are discussing High Performance Computing for Scientists and Engineers courses, 2nd module which is Open MP Programming. We have already covered the basics of open MP programming and now, we will see some of the important issues in open MP programming; without resolving these issues we will never get the right parallel performance.

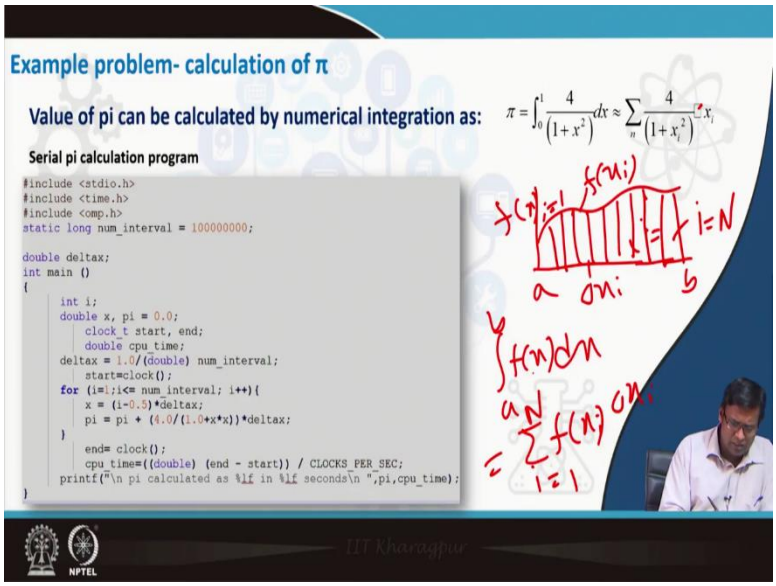
We will sometimes see that though we are using multiple threads and multiple processors to compute certain jobs , the performance is the same as the serial processing calculation., if we do not take care of this particular issue .These issues are related with data handling. So, we will see data handling and synchronization issues. This is the topic of this lecture in open MP.

(Refer Slide Time: 01:22)



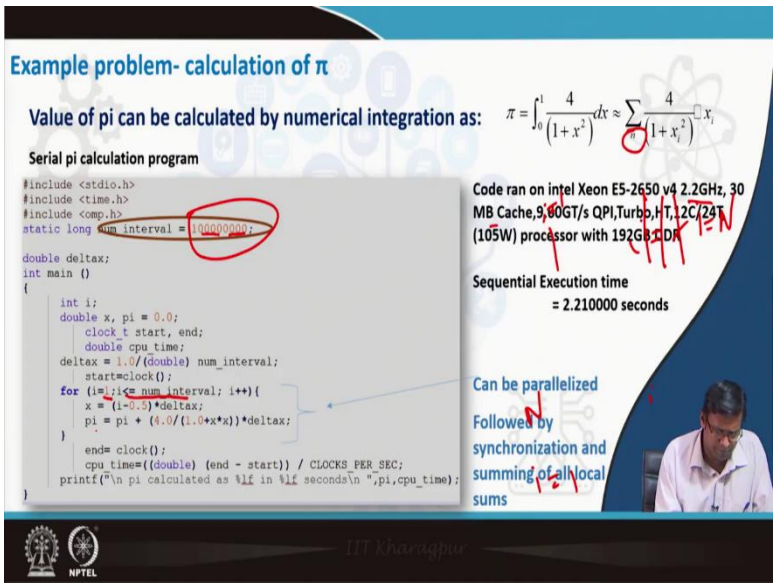
So, we will we will first take an example problem on numerical integration ,which is calculation of the value of pi by $\int_0^1 \frac{1}{1+x^2}$,numerically and see how we can parallelize it.We will see that there is an issue in accessing shared memory variables by multiple processors, and then how we can synchronize it and can get the right parallel performance.

(Refer Slide Time: 01:55)



So, value of pi can be calculated $\int_0^1 \frac{4}{1+x^2} dx$. This is also a summation of multiple small interval values of this function. We know this type of thing, if we have to integrate f (x) in between this region a to b, this is equivalent to taking different chunks and finding out the values of f (x_i). So, $\int_a^b f(x)dx$ will be $\sum_{i=1}^{N} f(xi)\,\Delta xi$ So, finding out summation of the values of f(x)Δx from 1 to N, and this is exactly what we are doing here.

(Refer Slide Time: 03:21)

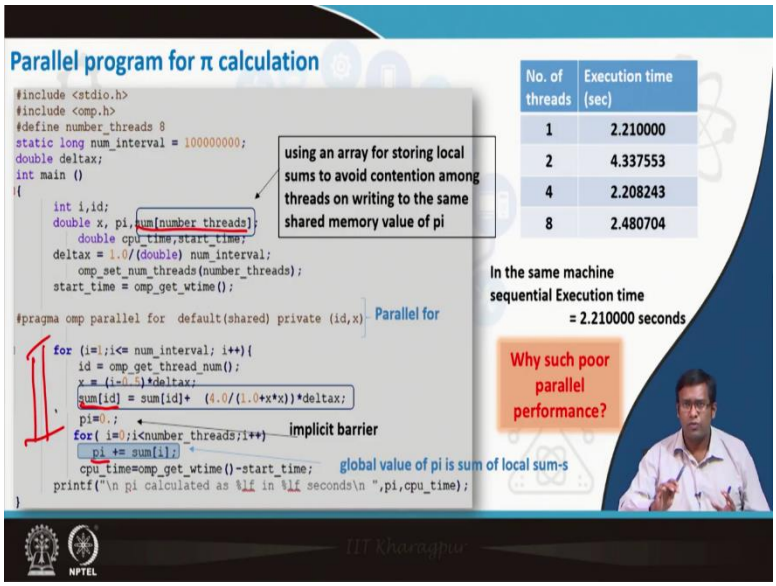


We wrote the program ,it basically takes this number of interval n to be, 10 to the power 8 and takes this loop that i = 1 to i is <= number of intervals. So, this 10 to the power 8 loops. x calculates the value , $x=(i - 0.5)\Delta x$ and $pi = pi + (4.0/(1.0+x*x))\Delta x$ and the time finds the time in between them and then writes that the value of pi,(we have seen the value of pi is 3.1459) and what is the time computed for that.

We run this in Xeon E5-2650 v4 2.2 gigahertz 30 MB cache, system with 12 cores which multi-threaded to 24 threads 192GB RAM. So, this is a sequential program, and will essentially run on a single thread.

So, we can see that the sequential execution time is 2.21 second. Once we execute this in a single processor, we get 2.21 second time. This is the for loop which does calculation from i = 1 to i = 10 to the power 8. So, 10 to the power 8 iterations are there and this is the most compute heavy part of the programs. Therefore, this loop can be parallelized. We will see the parallelization later. Once we parallelize it, the value of pi in each thread will get the local value of pi, they have to be summed up. So, some synchronization is also required.

(Refer Slide Time: 05:30)



So, the parallel program for pi calculation is as simple as we can put a parallel construct over the for loop # pragma omp for default(shared) private (id, x). Each thread gets its own ID number, it gets some parts of the iteration by default by default mapping and knows the ID number, calculates the x assigned to it and finds a local sum. We can see that the local sum is put into an array sum id, so we define another variable. The number of threads can be changed,

this is run in omp set number of threads by number of threads we can change the number of threads. We define a variable array sum with the size number threads and each thread is writing to the location of the array defined by its local ID. After this parallel for loop each of the threads; so, this is the iteration loop which is parallelized, each of the threads goes to another region and adds their local value to the global value of the pi.The value we calculate is exactly the same which we are getting from sequential computing. So, basically this parallel for is taking care of the parallelization of this program.

We are using this sum [number threads] array so that each thread writes to a different location. Each thread does not directly try to write the same shared memory variable because there are 10 to the power 8 iterations .We have launched 10 threads each thread is doing 10 to the power 7 iteration.

So, if there are 10 to the power 7 access to a shared memory variable concurrently by 10 threads there will be contentions, there will be false sharing, etcetera. So, in order to avoid this contention among the threads we ask each of the threads to write the values locally to one of the locations assigned to them in the array sum [number threads].

So, thread 0 will write sum 0, thread 1 will write at sum 1, thread 2 will write at sum 2 so on. So, after we finish this particular parallel for part there is a barrier and all threads should finish because there is an implicit barrier here after the parallel for loop, all threads should finish their work.

In the next part is not in the parallel region only the master thread will be active here you will take the values of sum 0, sum 1, sum 2 and add it to pi.

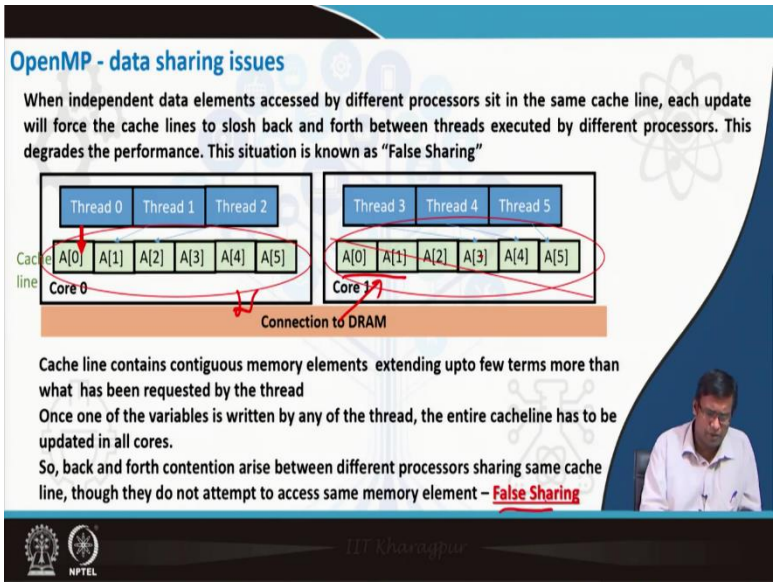
Now, if we execute it only the master thread is active at this part. So, the shared memory variable which is pi is not being accessed by multiple threads at the same time also, it is good. Global value is pi sum of local sums.

Now, if we run it in multiple threads the result is quite disheartening. We have seen single thread it takes 2.21 seconds the same as our single processor execution time in 2 threads it takes more time 4.33 seconds, in 4th thread it takes less time; however, the time is more than the sequential processing, then in 8 threads it also takes more time than the single processor time.

So, you are not getting any parallel performance; we are running it in multiple processors by multiple threads, but the execution time is more or at least comparable to the single processor execution even when we are using 8 threads. So, why is such poor parallel performance a question? Also, we can assure that we are not trying to access the same variable by all the threads are the same instance, because the threads are writing to their own local sums and these local sums are added by a master thread.

This is a barrier all the threads finish their work, it comes out of the parallel region pragma omp for, this for loop is only the parallel regime. It comes out of the for loop, next for loop is there which is launched only by a single thread. This for loop is not launched by multiple threads, only the master thread has launched this for loop and it is adding up all the elements of sum i. This is a single processor work and is not an issue.

(Refer Slide Time: 10:58)



So, why is the poor performance? When independent data elements are accessed by different processors sitting in the same cache line, each update will force the cache lines to slosh back and forth between the threads executed by different processors. This degrades the performance which is known as false sharing.

So, what is that? That these are independent data elements each thread is trying to do. Say, we have 2 processors; each processor is executing three threads. In this case we talked about 8 processors; each processor is executing one thread .

So, what happened we told that we are working in a 12 core multi thread to 24 threads processor; that means, virtually 24 processors are active there and when we launch 8 threads each processor can get 1 thread. But if we are in a 2-processor system and launch 6 threads each processor we will get 3 threads.

What happens here is that there are 2 processors and some threads are given to each of the processors. Now, each of the threads is writing to one of the memory elements and they are part of the same array. These memory elements are part of the same array. So, there is no contention among the threads.

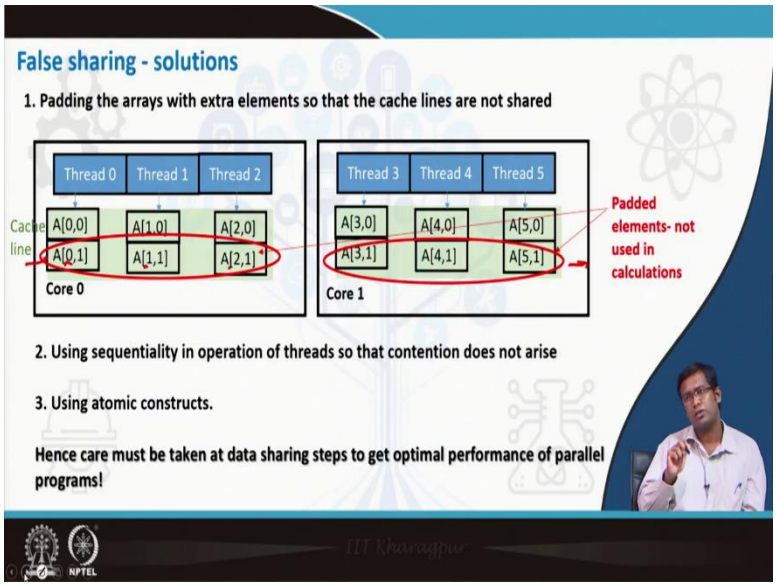
However, a particular core or processor has a cache line and the cache line not only comprises the elements accessed by the threads, but some more data. Similarly, this is the cache line for core 0, and this is the cache line for core 1, the second processor. So, you think about the processor: each processor has its own cache line.

So, though A[0] is not being accessed by thread 3 to 5, they sit in the same cache line of these threads of the core 1,and they are connected to the DRAM. In case A[0] is accessed by thread 0, the entire cache line becomes invalid because A [0] has been changed. Therefore, this entire cache (means this is the chunk of data) the core 1 is taking for the computation, this becomes invalid.

Cache line contains contiguous memory elements extending up to a few terms more than what has been requested by the threads. Once one variable is written by any of the threads, the entire cache line has to be updated for all the cores. For this thread this cache line becomes in invalid.

Therefore, thread 3 cannot even operate, this has to be updated. So, this update has to come to the RAM and this has to flow to core 1. So, it essentially introduces a great amount of latency in the program. Therefore, back and forth contention arises between different processors though they are not accessing the same data, but sharing the same cache line and this is known as false sharing.

(Refer Slide Time: 14:26)



Well, so, what can be the solution that pad the arrays with extra elements so that different processors do not share same cache line; that means, that though instead of making A to be a 1-dimensional array make a 2-dimensional array and add things which are not being accessed, but you are padding it.

You are adding more elements because the cache line has a particular size depending on the architecture of the processor, it is saying 32MB ,48MB some size is required by the cache line. Fill the cache, once one thread or one processor reads from the memory, fills its cache so that its cache does not share the same thing with the cache of the other processor.

Well, so, pad it with some extra element and now even if it tries to update A [0, 0] this cache and this cache is not common, there is no common element. So, this cache and this cache there is no common element and therefore, there is no cache contention or false sharing. These are called padded elements. They are not used in calculations. They are only given, so that caches having the same common elements in cache of different processors do not arise.

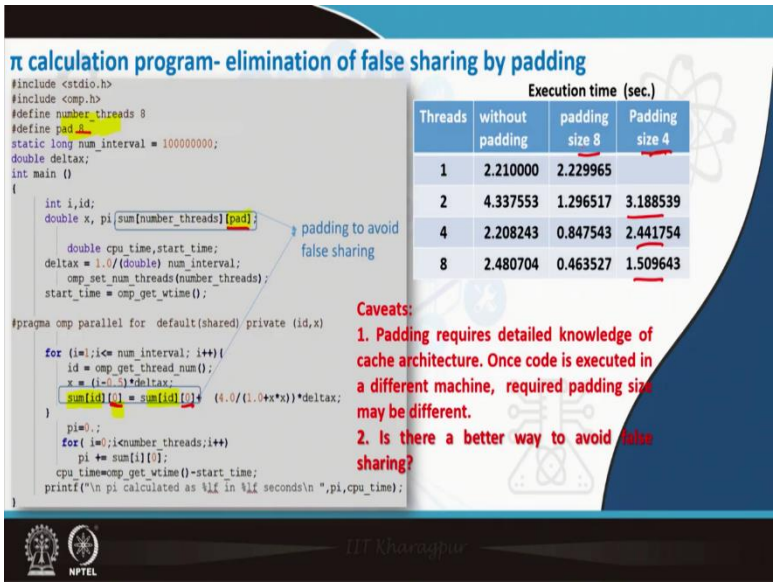
The other is using some sequentiality in operation of threads, so that contention does not arise. Ask one thread to take care of that part, another thread will come and take care of the other. If you are accessing a shared memory do not ask all the processors to access the shared memory at the same time, add some sequentiality; processor 0 goes and writes there, then processor 1 goes and writes there ,so that they do not try to update same variable or variables with the in the same cache line at same instance.

Use atomic constructs: we will look at atomic constructs, but this is again if there is some variable being updated in the memory, the memory location is not accessible by multiple processors at same point of time. There is something using the hardware and system support, it can be ensured that the memory location is not being accessed by many processors at the same memory location at same instant and that also helps here.

Also, you can use reduction clauses. We have seen reduction earlier because if you are trying to do the same operations by all the processors on the same shared memory data you can use a reduction process and these can give you optimal performance.

So, care has to be taken for data sharing steps to get optimal performance, even if they are not sharing the same data, but if they are sharing contiguous data, so that they can end up in sharing the same cache line there can be false sharing and that can dictate the performance.

(Refer Slide Time: 17:15)



Now, what is done is that instead earlier sum was a 1D array instead making it 1D array make it a 2D array and add padding of size as 8. This padding size is given an 8 with an idea that if we have 64-byte cache, with double precision variables then 8 X 8=64 spaces then these 8 variables can eat up 64 variables and can take up the entire 64-byte cache .

The disadvantage is that what will be this number, this will be 8 or something else we need to know about the cache size. L 1 cache size of the processor cache size is given in the hardware

of the processor as the processor is configured cache size is given. So, you need to know about that.

Cache is usually very important because it prefetches some data for operations, but here we can see that sharing the same data in the cache of different processors in a shared memory system can degrade the performance severely. So, you try to fill the cache of the processor with some data which will not be used by other processors, we need to fill the cache. There is no common data in between cache of different processors. This is the padding you add another pad with the local sum variable, so that when each of the processor is writing to the writing the local sum ,none of the processors or none of the threads because threads belong to independent processors here share the same cache line because they are writing only in the first location or zeroth element of that array and rest is not used.

So, this padding is used to avoid false sharing. We can see padding size 8 ; in a single processor the time is almost the same 2.229 second which is the same as a single processor calculation. In 2 processors it is reduced, almost by half not exactly half because there will be certain overhead; in 4 processors it is further reduced; in 8 processors it is also further reduced.

So, we are getting parallel performance. It is reducing almost by half not exactly by half because we know that as we increase the number of processors or number of threads overheads are there.

Well, so, padding with extra elements gives us good performance. The caveat is while doing, so, padding requires the detailed knowledge of cache architecture. Once code is executed in a different machine the required padding size may be different because these 8 elements which we have put here is sufficient to fill up the cache line.

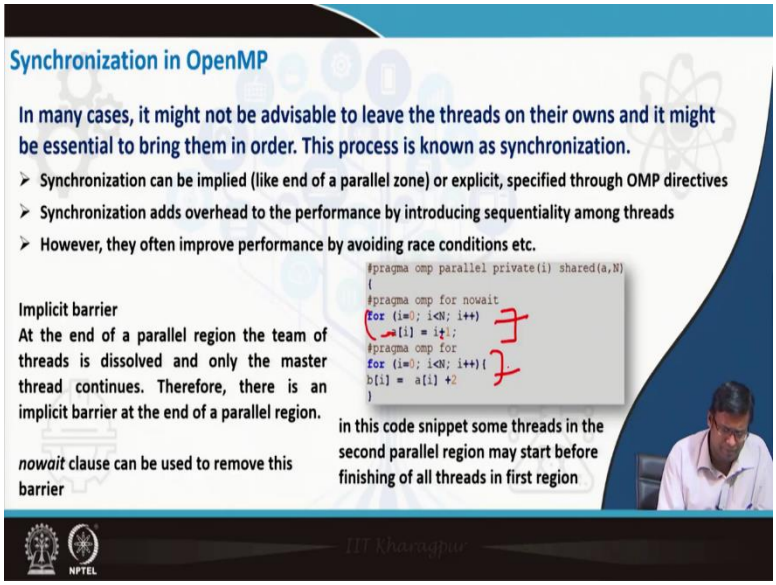
In a different computer the cache line size might be different and with 8 it might not be able to fill up the space, it might require more elements. So, it is not a portable code anymore. It is an architecture hardware dependent program that is the cache size we need to know and then we can specify the padding. We used a padding size 4 and we saw that the performance did not improve much because still there is some false sharing here.

So, the question is that is there a better way to avoid false sharing? Is there a better way to avoid writing to the same shared memory location? One way is that you do not write on the same variable by all the processor, but even if you are not writing the same variable if you write

contiguous elements of the same array there is false sharing. Is there a better way; one is padding. But padding is very very much dependent on the padding size and how the padding size compares with the cache size of that particular processor.

So, is there a better way? Is there a more portable way for that? that means, you go from one processor to another processor one system to another system still it will work any other way.

(Refer Slide Time: 21:59)



For that we need to look into synchronization so that we can ask different processors to synchronize among themselves and write to the same shared memory location, so that they do not end up in having contention among them; they do not end up in having false sharing among them. In many cases it might not be advisable to leave the threads on its own that this thread is trying to write, this thread is also trying to write though different locations they are some way contiguous and make some sense of wrong use of cache coherence protocol by false sharing .

So, it might not be advisable to leave the threads on its own rather it might be essential to bring them in order that thread 0 will do it first. Well, once the cache is updated thread 1 will come and do it. This process is known as synchronization. So, one part is using padding and brute way of filling the cache and avoiding false sharing another part is if we can do some synchronization, we can probably get better results.

Everything again occurs because we are using a general statement which will be executed by different threads. That is why you need to do it in some contiguous variable because we will

define an array and each thread will write to the different locations of the array. In case we had the flexibility of not writing a general program, the first thread will write to a location called A; the second thread will write to a location called B and we can add all A B etcetera we have put into this issue.

But we are developing parallel programs of legacy codes we already have for loops. We have seen that our old pi program had a sequential for loop which we are trying to parallelize, so, you will end up in this type of situation. So, we need these treatments like avoiding false sharing using padding, or using synchronization in discussing synchronization here.

Synchronization can be implied like at the end of parallel region we do not need to write anything, but all the threads get destroyed except the zeroth thread will be active and once all threads have finished that part then the zeroth thread will go and take care of the remaining part the following subsequent part.

So, it is an implied synchronization when a parallel zone ends ,there is a synchronization that every thread will wait till all the threads have finished up to this part. So, this is a synchronization or it can be also too explicit through some open MP directives.

Synchronization adds overheads to the performance because it adds sequentiality that one thread has to wait till others have finished the job or one thread has to wait till one specific thread finishes the job it will follow it. So, it adds some sequentiality across among the threads.

Threads cannot really work in parallel when there is a synchronization construct. At the synchronization statement there is some sequentiality, some order among the threads and they lose their parallel execution mode and follow one by one. So, it adds to overheads. However, they often improve performance even when they are adding to some overhead, but there are some other issues like false sharing which can be avoided using synchronization or race condition and we can get better performance.

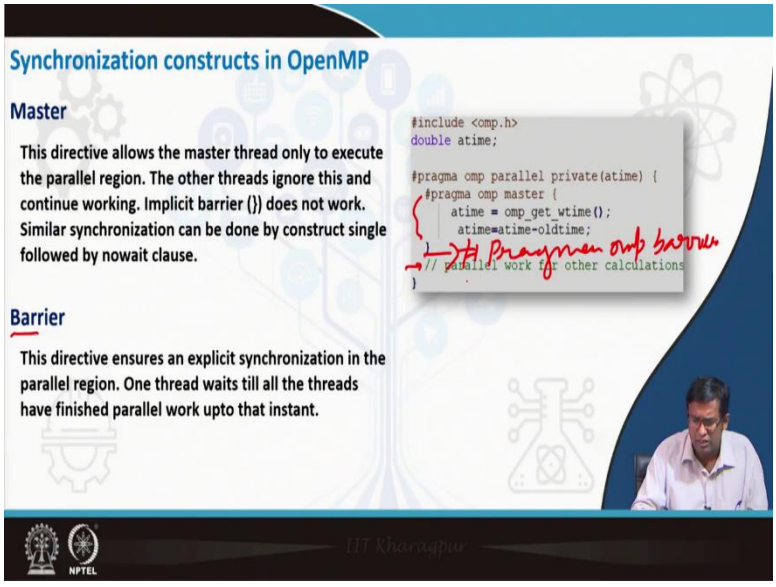
One of the synchronization statements as I said is implicit barrier. At the end of parallel regime, the theme of thread dissolved and only the master thread continues. Therefore, it is an implicit barrier that at the end of parallelism master thread is waiting till all the threads are destroyed, till all the threads have finished their work, so that they can be dissolved and the master thread will take care of the remaining part. So, this is also a synchronization step.

In our case, we do not need that. These are really parallel independent jobs; master thread does not need to carry anything any output of the other threads from the parallel region etcetera we can put a nowait clause. Then when we are calling the parallel regime we can write # pragma omp parallel nowait and then the loop.

That means, that once a master thread has finished its work it can go and start next work and once even other threads some of the threads have finished their work, if there is another parallel regime the other threads can go and start that work. They do not need to wait for all remaining threads to finish the work. So, there is no need to wait.

When this particular loop is launched the threads will be active, one particular thread is working here till it finishes its work; once it is finished it can go and go into the next regime. It does not need to wait for all threads or for the finish of this parallel entire for loop till all other threads are finished. It does not need to wait for that. So, in the code some of the threads who finish their job in the first for loop can go to the second for loop here.

(Refer Slide Time: 27:48)

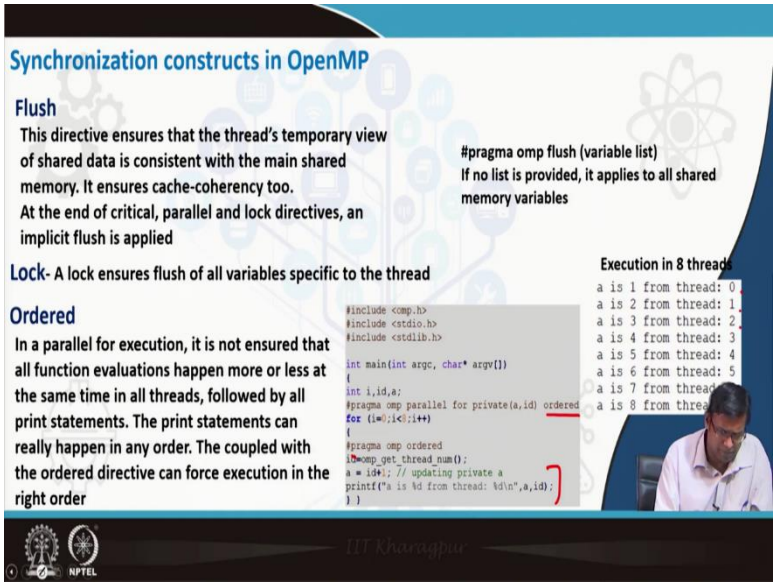


If I give the clause master this directive allows only the master thread to execute that region and other threads ignore this and continue the work. There is no implicit barrier to this regime ; that means, if we write this only master thread is active here. If there is something else the other threads this particular part is only executed by master thread, the other threads can go and work there.

In case we need to ask other threads to wait for the result of the master thread, we have to put an explicit barrier. Here we have to put a dollar pragma omp barrier in case we need a synchronization here, otherwise it is not synchronized only master thread will work here the other threads will carry with their own work.

Barrier: this directive ensures that explicit synchronization is put in the parallel regime all threads will wait till the other threads have finished their work. Here there is no implicit barrier. So, if you have to put the barrier you have to put an explicit barrier here.

(Refer Slide Time: 28:58)



Flush: this directive ensures that the thread's temporary view of shared data is consistent with main shared memory and cache consistency too. So, if some variable is updated by one thread this is known to all the threads. So, you put a flush; that means, it is not being updated by the other thread and other threads do not know that one thread has updated it and are still trying to write it.

Here we have seen that when we are in omp parallel Fortran program when all the threads are trying to access the same shared memory data there is some garbage written with some contention among them. In a flush that once one thread has written it, it will flush this. All threads cache will be flushed and the new data will come here. It ensures cache coherency and end of critical parallel and lock directive there is an implicit flush.

If pragma omp flush (variable list) if variable list is given then flush occurs only on those variables and if variable is not given all shared memory variables are flushed.

If we say lock then all the shared memory variables specific to the thread in a particular region will be flushed. One thread is working on certain shared memory variables, if we write a flush here that variable specific to that thread if we write a lock here the variable specific to that thread will be flushed. It will not act over all the shared variables, but in the specific variables that are being operated by the threads.

The order is a very important issue. In parallel execution it is not ensured that all function evaluations happen more or less the same time in all threads, followed by print statements. Many processors have their own speed and own latency, so each thread is kind of picked up randomly and operating there in one parallel regime. Thread 0 then it might be followed by thread 8, then thread 1 might come depending on how the CPUs are active in different cases.

If they print something that is not in order, we have seen that first thread 0 is written, then thread 1 is written, then thread 4 is written, then output from thread 2 is written. So, they do not write in order, but if we put the ordered synchronization clause then the print statements really happen in order and the order directive can force the execution to be in the right order.

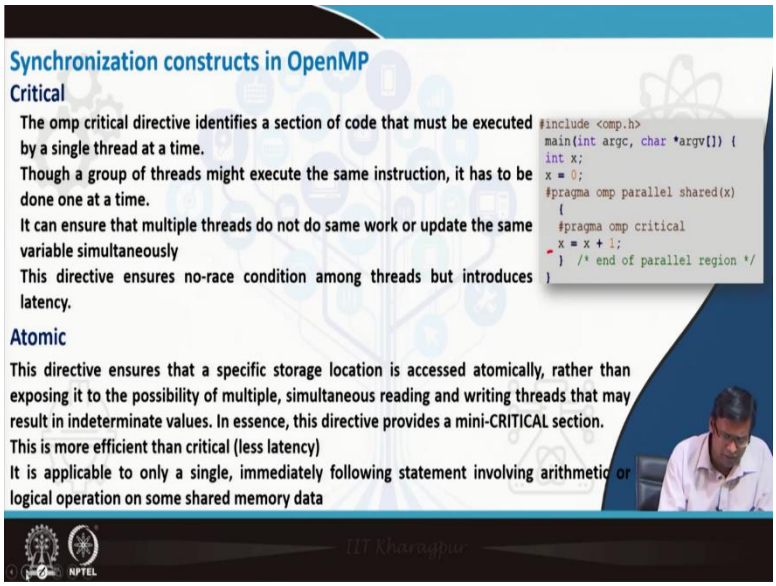
For example, if we have pragma omp parallel for private with the ordered clause and then we write this # pragma omp ordered location what will happen that all the threads here will which is updating the private variable a and they will execute in order.

So, if we see the output will be from thread 0, for thread 1, for thread 2, thread 3, so, the threads will follow in order. How can that be achieved? Of course, that can be achieved by some sequentiality. So, there is certain overhead when you do order that all the threads really cannot work in parallel.

The 0th thread works first, then the 1st thread first and to 7 thread they are waiting till the 0th thread has finished their work . 2nd to 7 thread is waiting till the 1st thread first finishes their work. So, there is some sequentiality and we will use the right parallel performance.

So, however, this can be important in some cases especially when we are trying to write some output from the threads that can be important.

(Refer Slide Time: 32:47)



Critical ;this is extremely important. omp critical identifies a section of the code which has to be executed by a single thread at a time. Therefore, we discussed the shared data update issue. If we use the critical construct here pragma omp parallel pragma omp critical, that particular section will be worked on by one thread at a time. So, if there is a shared data access each thread will access one by one that data. Though a group of threads might execute that same instruction it has to be done one at a time. So, pragma omp critical means thread 0 first writes x = x + 1= 0 + 1 =1 if in this particular code x = x + 1,a shared memory variable is being shared x by all the threads. So, thread 0 will do first, thread 1 will do first they may not be order , but not two threads will try to update it at the same point of time and end up in some wrong result or some contention.

It ensures that multiple threads do not do the same work or update the same variables simultaneously. They have done one by one that particular part. This directive ensures no race condition among threads while writing to the same shared memory location, but includes latency. Of course, this is understandable because we lose the parallelism in true sense, threads operate one by one.

Another construct is atomic which says that if there is critical work for doing some work x = x + 1 this is a work which critical is doing or x there can be a particular loop involving some of the private variables also which critical is doing it. Here it is updating a memory, but it can be something else than updating a memory, can do some other work also. But, when it is only

associated with shared memory update and we want the processors not to have contentions, processors do one by one; one processor will do only at a time, we can use the atomic construct.

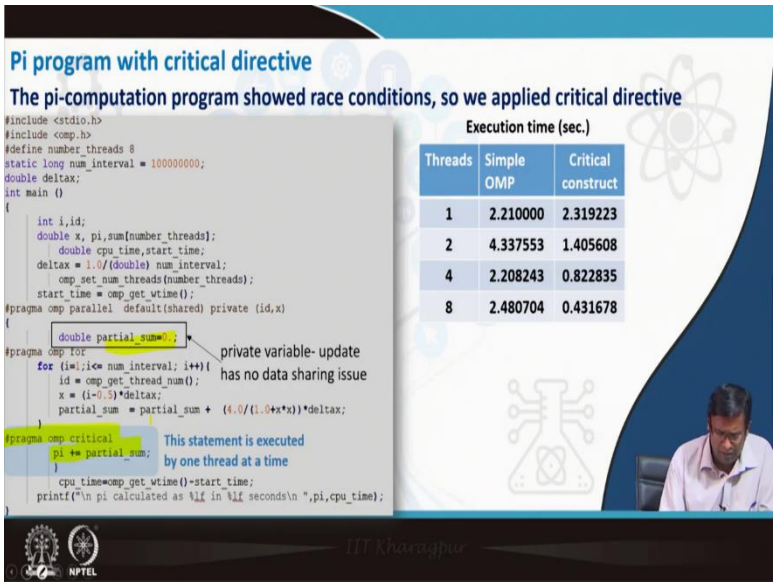
This directive ensures that a specific storage location in the shared memory is accessed atomically rather than exposing it to possibility of multiple simultaneous reading, writing by threads which may result in indeterminate values ,one of the threads will access that particular memory location and do something read, write update on that particular memory location.

It provides a mini critical section; mini critical in a sense that it is only associated with shared memory variables. One particular shared memory variable location the threads will operate one by one. It cannot do anything else other than shared memory update and this is kind of compiled by system level operation on the memory and hardware level operation and it is also seeming to be more efficient than critical. It has less latency and is a very useful construct also.

So, when we are updating a shared memory variable by multiple threads, we can use critical or atomic constructs to avoid race conditions and false sharing. It is applicable only to a single immediately following statement involving arithmetic or logical operation and some shared memory data.

For other cases we have to use critical if you want sequentiality among threads if you only have one thread to be active at one region, though all threads will perform that region. But if we want if it is on a single shared memory statement it is the atomic construct.

(Refer Slide Time: 36:57)

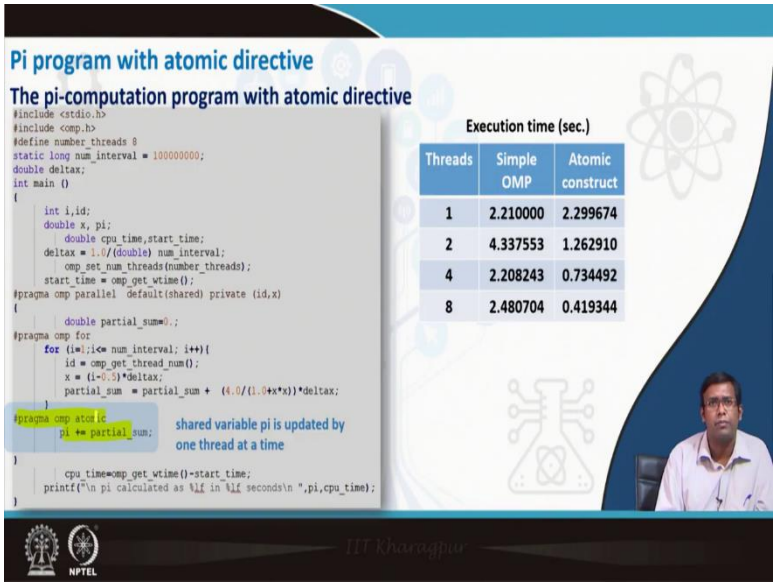


So, we can see here it is the same open MP function and we have introduced the critical construct here. So, we have written the output this is the same pi calculation to a pi partial sum. Partial sum is the private memory variable for each thread and each of this thread is calculating the new summation to the partial sum. In critical all threads one by one update pi = pi + partial sum.

So, this is the shared memory access same shared memory location access by multiple threads, but we use a critical construct and therefore, we get the right parallel performance that as we increase the number of processors the speed is almost increasing not exactly linearly, but close to that. So, you are getting the right parallel performance.

The same thing so, this is the private variable which has no data sharing issue and this statement the critical statement is executed which is on the shared memory variable updated by the private variables which is executed one by one.

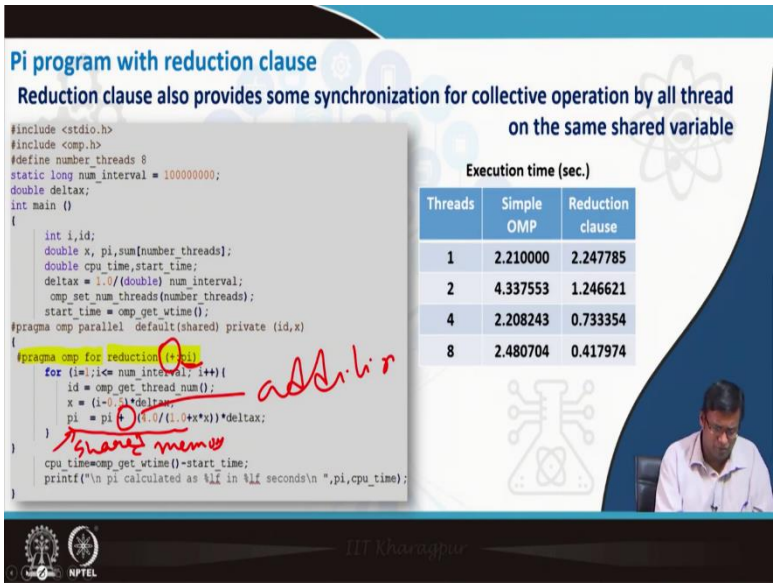
(Refer Slide Time: 38:08)



The next one is atomic. Instead of critical we put an atomic that the private variable is there, but the shared memory variable is updated in an atomic manner we see performance is actually little better than critical construct. This is a simple code for more complex calculations we will see performance is quite actually better than critical construct in atomic construct. We get the same parallel performance.

So, the right parallel performance is obtained when you are using critical and as well as atomic. Well, this is again a shared memory variable updated by one thread at a time because we have introduced the atomic construct here well.

(Refer Slide Time: 38:51)



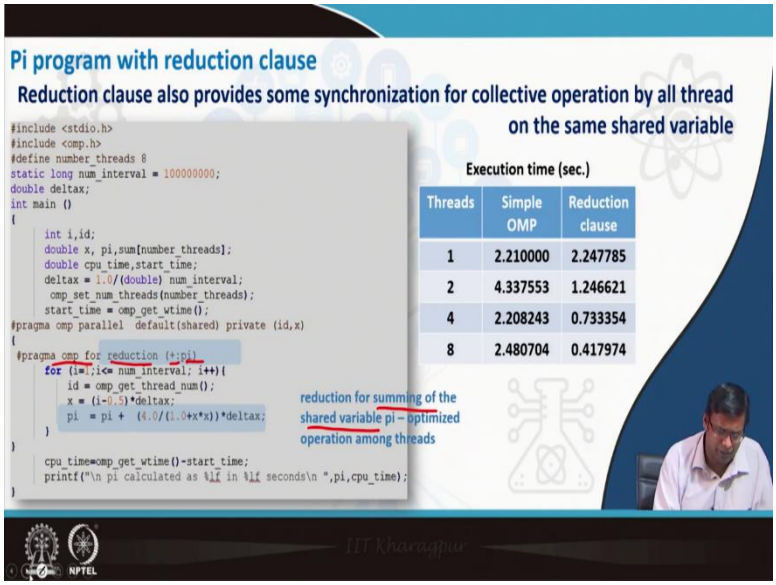
Also we can use reduction clauses. That all the threads are adding up the elements of the numerical integration and finally, the local sums will be added to the global sum. So, it is a summation operation which is done by multiple threads. Each thread is summing it's up and giving its local sum finally, but we need a global sum finally. The final goal is to get a global sum.

We can very easily use a reduction operation here that the directive is pragma omp for loop will use the reduction and what is the reduction? Reduction operation is addition and the reduction variable is pi. So, the main for loop can be retained as it is, pi = pi + which was in the sequential program.

We do not need to introduce any shared memory variable, any private variables separately because all the private local sums are taken care of by the compiler itself. So, backend these variables are; but the programmer has to only write that this calculation pi = pi + something ,is a summation which will write to the shared memory pi . An additional operation will be done over the shared memory variable pi.

We will see that the performance is as good as critical or atomic . It is the right parallel performance we are getting out of it.

(Refer Slide Time: 40:32)



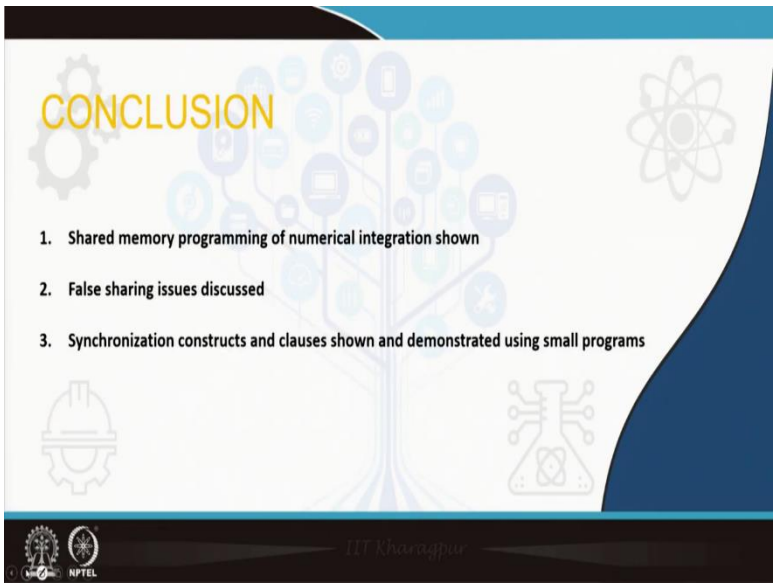
So, reduction for summing of the shared variable pi. So, you have to write the operation is summation and the variable is pi. This is optimized. We have discussed the reduction clause before, this reduction is a clause on the parallel for loop on the directive. This optimization for that, because shared memory is being accessed by multiple threads or multiple processors is done already by the compiler and the system level support.

Programmers only have to specify that it requires an optimized operation for that which is a reduction operation. We can see that it is more beneficial than finding out local sums and adding it by the programmer himself in the program itself in a simple way, because if the programmer tries to do it without reduction, he has to use atomic or critical or use some other clauses. But, reduction itself takes care of the optimization. It has extreme good utility in scientific computing calculations.

(Refer Slide Time: 41:46)

So, these are the references Chapmans OpenMP book, Tim Mattson tutorial and Hermann's Fortran95 and there are some tutorials given by Eijkhouts in Eijkhouts website from Utexas.

(Refer Slide Time: 42:06)



We looked about shared memory programming of numerical integration. It is very important that we understand false sharing in order to get the right performance. We have to write programs which will avoid false sharing and synchronization constructs and clauses are shown and demonstrated also. You have demonstrated how reduction clauses can be used to get the right parallel performance.

So, this gives us a good background on programming using open MP in shared memory systems and also gives us some idea about optimization of the program looking into the overheads, looking into the synchronization issues and looking into the false sharing and contention on accessing the shared memory issues.

With this background, we will see some of the matrix calculation algorithms in the next class and finish the open MP discussion. Then, we will go to distributed memory MPI discussions.