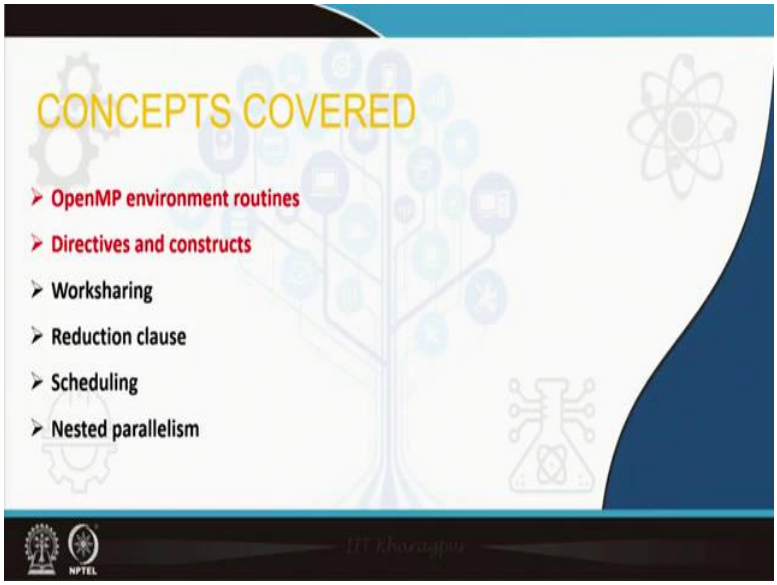
High Performance Computing for Scientists and Engineers
Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 02
OpenMP Programming
Lecture – 16
Essentials of OpenMP Programming (continued)

(Refer Slide Time: 00:40)



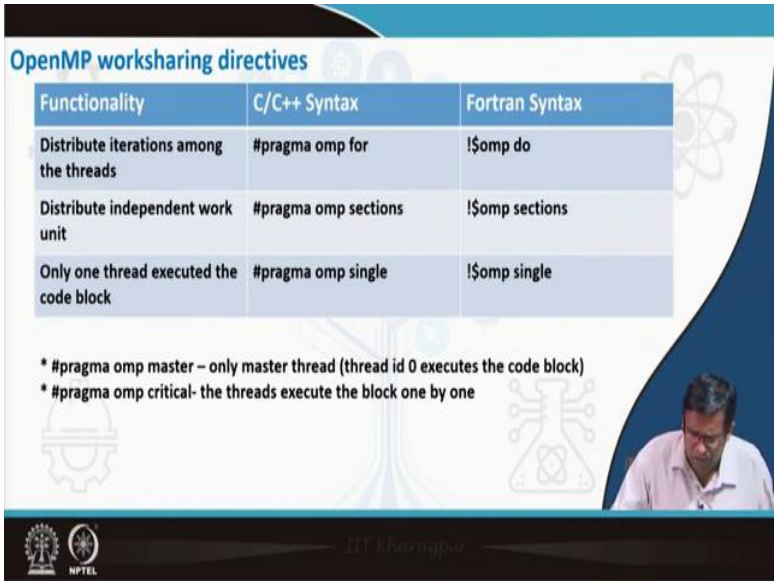
Welcome to the class of High-Performance Computing for Scientists and Engineers. We are in the 2nd module, which is OpenMP Programming and we are discussing Essentials of OpenMP Programs. This is the second discussion on essentials of OpenMP programming.

In the previous lecture, we covered the concepts on OpenMP environment routines and the directives and constructs of OpenMP, and we have specially focused on parallel directives in OpenMP and inside the parallel construct we can use other constructs like omp for etcetera. We have also discussed different types of variables in OpenMP shared and private data; how data handling is taken care of in OpenMP.

So, we look into the remaining important aspects which are work sharing, reduction clause, scheduling, and nested parallelism in OpenMP. The idea is that once we are aware of this particular semantics and syntaxes of OpenMP program, and also the functionality of different

OpenMP constructs and clauses will be able to develop our own OpenMP program. Some of them will see in the subsequent classes well.

(Refer Slide Time: 01:52)



So, we are actually discussing work sharing directives in OpenMP. We have earlier seen that whenever, we need to do something in parallel in OpenMP, the first important directive is #pragma omp parallel or in Fortran !$ omp parallel. That specifies that the construct which has been launched in the OpenMP program, this construct will work in a parallel mode; that means, multiple threads will be active and these threads will take care of the remaining part of the calculations; some of these threads will work in parallel depending on the number of processors.

What will these threads do when they are launched in parallel? There can be certain cases because OpenMP is specifically designed for some class of computation, which we generally call scientific computation; which we are discussing here. So, it is a priori known to the developers that these are the type of programs or these are the type of algorithms that OpenMP is trying to parallelize.

What is the region of the program which requires parallelism? one part may be that there is an iteration, there are a number of iterations; there is a do loop or a for loop which will take care of the number of iterations. These iterations are mutually independent; that means, for i = 1 the calculation is independent for i = 10 in the same loop.

So, this do loop can be parallelized. The other issue can be that there are different tasks and this task can work concurrently, these are not most likely part of what I will say part of a same iteration loop. But there can be say for example, we are looking into solving some equations, equations of motion; and equation of motion in x direction and equation of motion in y direction and in z direction are independent of each other. So, if you know the force and mass the acceleration, it can be independently calculated in each direction

So, in order to calculate acceleration in each direction independent threads can be launched. So, the second part of parallelism can be that there are independent works, but at one instance, you want to know that acceleration in all 3 directions, at a next instance again you need to know acceleration in all 3 directions.

So, there can be independent works which are large pieces of works, but they will be probably repeatedly solved and the solution of each unit of this independent work will take a certain amount of time. So, you can launch one thread for x axis calculation, one thread of y and another thread of z. So, this type of work distribution that if we can identify, that these are the sections of work which can be executed concurrently and launch different threads for them.

So, within the parallel region we will say that now, multiple threads will be there. So, each of the threads can look into different sections, and this is the next way of OpenMP parallelization that is launching sections and distributing independent works.

Again, we are discussing solving momentum equation in x y and z direction and in certain cases, we see that there is no force therefore, no acceleration in y and z direction we only need to calculate about x direction. So, we will not launch any thread in y and z direction. We will ask only one thread though everything is in the parallel region, but we will ask only one thread to do that particular calculation. So, it’s also possible that only one thread will execute that part of the code block the other threads will not do that.

When we see this work sharing directives, we will appreciate the fact that when somebody is developing a parallel program, he is not developing, he is not writing different lines of the program for different threads?

It's developed in us with a sense of generality, that one particular line within a parallel region will be executed by all the threads and these threads, they are operating something like a single instruction multiple data model. So, each of these threads will execute the same instruction, but

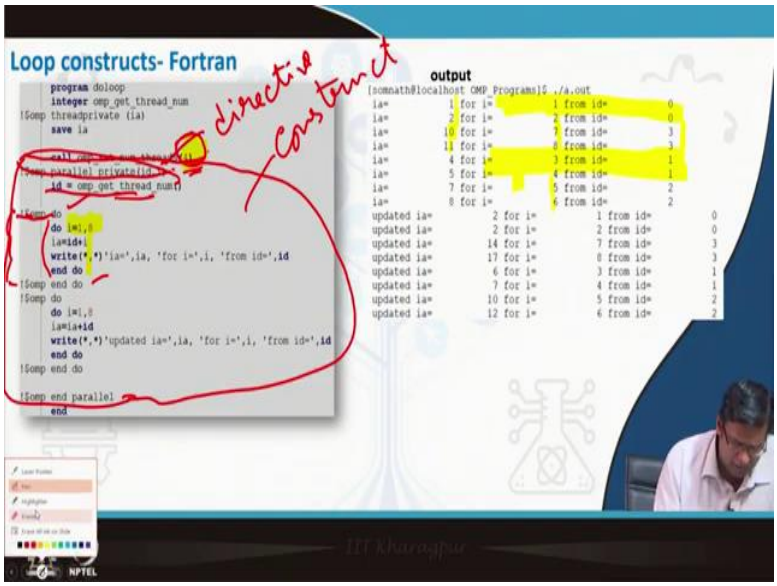
they will pick up the relevant data for their work. They will get connected to the relevant data but, they will essentially execute a similar type of instruction for them; in case they have to do different instruction, which is multiple instruction multiple data it can be also via section model. That different thread will do different work and pick up data differently.

However, when the program is written the programmer does not really try to write instances of the program specific to a thread. So, what its role is is to rather write it in a general format, and when these different threads are launched, each thread using the algorithm will pick up its relevant work and execute that. Therefore, if there is some work which is specifically to be done by one particular thread, we have to specify that other threads will go in inert mode and only one thread will do that. So, this is also one part of work sharing. There are other parts of work sharing like pragma omp master, when only the master thread or thread id 0 will execute that block of code. So, you are controlling mode and this particular thread will work here by pragma omp master.

There is another interesting work sharing directive, which will see later in detail; pragma omp critical. It shows that though the works are independent and mutual threads are concurrently active, but at one particular region of the work each of the thread will follow some sequentiality.

So, one by one threads will do that not all the threads will attempt to execute that particular instance of the program at same point of time concurrently, this is particularly done when you have a single shared memory variable and multiple threads will try to update that variable. If you put an omp critical; that means, one by one the threads can update the variable, so that there is no contention, less condition, no overwriting of the shared memory etcetera. This is also one important work sharing directive.

(Refer Slide Time: 09:19)



So, one of the works sharing directive is a loop construct; that means, there is a do loop, and you have parallelized this do loop using # pragma omp for or !$ omp do in Fortran.

So, what is done here? So, if you see this particular Fortran code, we have a loop do i = 1 to 8 id inside the parallel region of the program. This parallel private is the parallel directive, and we call the entire part, it starts at omp parallel private in Fortran, it is easy to identify the end of the parallel construct by omp in parallel.

This entire part is called the construct, the centre part is the parallel construct, that means, at this region multiple threads are launched and they are active concurrently. This entire part from omp parallel omp end parallel, is the parallel construct.

So, this is the construct we have seen earlier and this part is called the construct, and this particular line omp parallel is called the directive. Now, inside the construct, 4 threads are launched; first it finds out the id of the thread each. Then, there is a do loop do i = 1 to 8 ia is some variable which is id + i, and each of the thread will take care of some parts of this i loop because, we have launched omp do; omp do, and this is another construct which starts at omp do and ends at omp end do.

Within these 4 threads which are launched, will take care of these 8 iterations in the do loops. what will they do? They will update a variable ia ; a thread private variable.

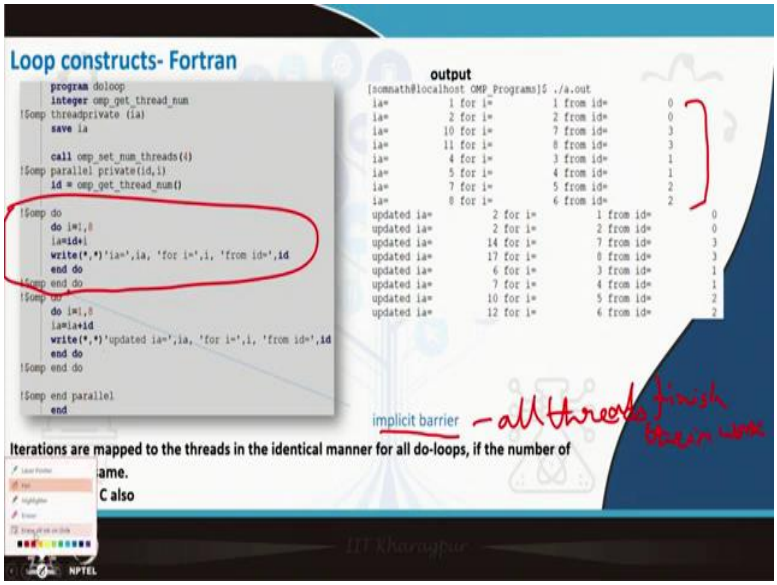
So, for each variable ia is each thread ia is somewhere private. ia = id + i, i is the iteration number and id is the thread number. So, for i = 1, we can see because we there are 8 iterations which is distributed into 4 threads, each of these threads are picking up 2 of the iterations, thread 0 has picked up 1 ,2 ,thread 1 has picked up 3 and 4, thread 2 has picked up 5 and 6 ,thread 3 has picked up 7 and 8.

They have written that ia which initially started from ia = 0 Fortran initializes it automatically to be 0, i = id + i. So, for thread 0, id is 0 therefore, ia will be 0 + i the values of i, for thread 3 id is 3, but i was that goes here 7 and 8 last 2 iterations, and they will pick up 10 and 11.

So, we can see that there are some iterations and this number instead of 8 can be anything else also the thread number could have been anything else. OpenMP compiler itself distributes different iterations to the number of threads; here we can see there is an even distribution; that means, there is a load balanced distribution. But there are cases when the number i the number of iterations is not exactly divisible by the number of threads. However, it ensures some sort of load balancing.

Also, the programmer has some control over the distribution of threads, distribution of iterations over the threads through schedule clauses, we will look into schedule clauses later. But right now, we can see, there is a for loop or, there is a do loop in Fortran, and if we can just define do open omp do construct then, this is automatically parallelized and distributed among states well. So, this is what is called a loop construct? And this same thing will happen Fortran.

(Refer Slide Time: 14:53)



Iterations are mapped to the threads in the identical manner for all do loops, if the number of iterations is the same. We see that there is also another do loop, do i = one to eight same number of iterations, iterates updated ia = i + id. What is the value of ia, that is again added with ia. We can see here iteration 1 and 2 were going to id 0. Here also this is again the same way parallelized, so this loop ends another loop starts here. This is also iteration 1 and 2 goes to id 0, iteration 7 and 8 going to id 3 thread 3 iteration 7 and 8 here also goes to thread 3.

So, the mapping of the iterations to the threads are identical for all do loops, if the iteration numbers are the same. If the iteration number changes it happens differently, but this gives a very good control for the programmer over the work distribution in OpenMP because; if he is trying to solve something over a large number of iterations and the iterations happen over and over, he knows that these iterations are specifically assigned to this particular thread.

Another point is that we have said that the variable ia is a thread private variable. So, the copy of ia remains in each thread at different do loops. From the first thread, id = 0 has the values of i 1 and 2 and the last updated value of ia is 2.

So, this value 2 stays with the thread 0, because, it's a thread private variable, and it starts ia = i a + id when it writes updated ia; that means, this value 2 stays here the value 2 comes here, 2 + 0 = 2.

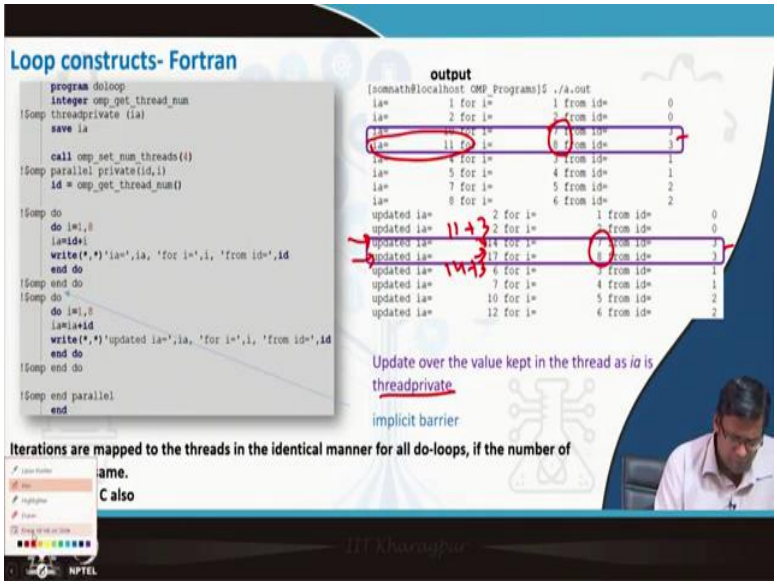
For thread 3 ia was 10 and 11. So, the last updated value was 11, these 11 values stay with thread 3. Now, at the next loop 3 thread does ia = ia + id. So, it is ia + id =11 + 3 and again it adds another 3, because id is 3 here, 11 + 3 =14, 14 + 3 =17 so this stays here. Because there are thread private variables these values stay with the thread.

So, these are certain features of do looping and thread private variables. The main point is that if there is any do loop in Fortran or for loop in C, if you use parallel for C or if you use omp do or parallel do for Fortran, it will be automatically parallelized and some of the iterations in the loop will be mapped to some of these threads. This will be identical, if the number of iterations remain the same for the same number of threads, this is the same for C also.

We also can see, if we write anything after the end of the construct, this will be operated sequentially; that means, this thread ends here, everything executed by all the threads finishes and the next instances of the program are launched. Therefore, this is called an implicit barrier;

that means, all threads finish their work. In case some thread has not finished the work, the other threads will wait for it.

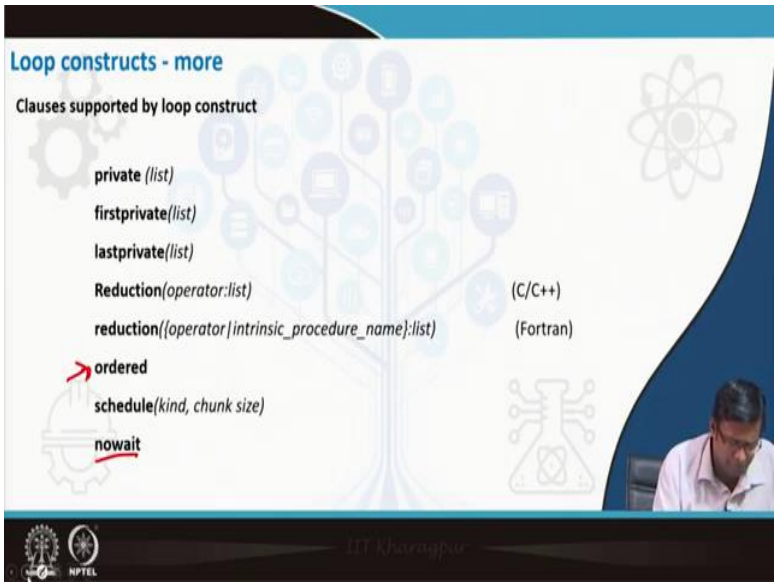
(Refer Slide Time: 19:18)



We can see that when the new loop is launched the number of iterations specific to that is the same to that particular thread, and the last updated value of ia, which is ia = 11, goes to this thing, this is 11 + 3 = 14 and then 14 + 3 = 17. The value of the variable comes here, as it's a thread private variable well.

Also, we can see that this is same in C also; in C that the loop construct is # pragma omp for you can parallelize, the particular for loop using # pragma omp for. But it is important that this parallel do loops must be part of a parallel construct. They should follow a directive parallel omp parallel or # pragma omp parallel, directive and inside that it should be inside the code block associated to the directive or inside that construct.

(Refer Slide Time: 20:31)



The clauses supported by the loop constructor private, first private, we have discussed about the clauses, reduction; we can do reduction using the loops also; order, schedule. So, if we see orders if we go back to the previous slide, we can see that these outputs are not in order 0 8 thread iteration 1 and 2 is written first, then 7 8 then 3 4, then 5 6. Here 1 2 again 3 7 8 3 4 5 6 is not written 1 2 3 4 5 6 7 8, it's randomly written by the threads.

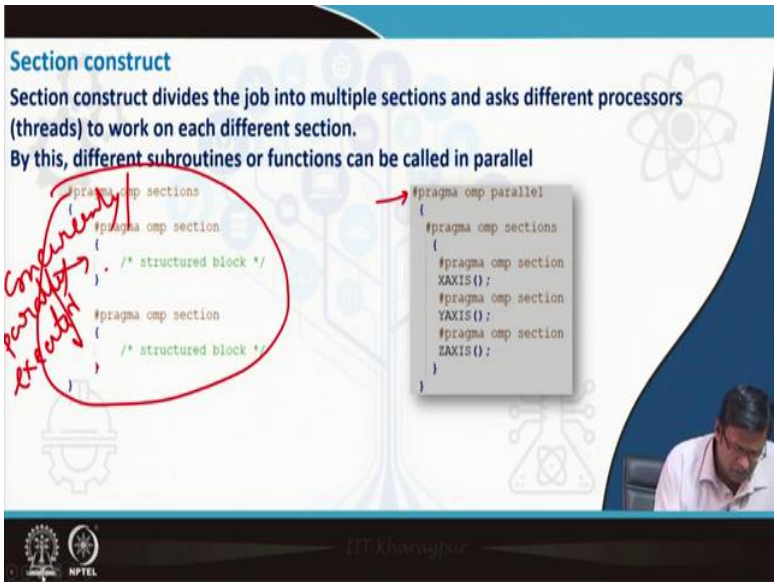
Now, if we put the order clause here, they will be written in an order. How can they be written in order? If there are multiple processors and each of the processors is executing one of the threads. This processor can work with their own latency and they can write in any order. So, you need a synchronization in between them; that means, first the thread 0, will write then thread 1 will write so on.

This synchronization requires some sequentiality of execution and that will also add to the overhead. So, when we put this order, there is certain overhead, and another point we have seen there is an implicit barrier after the threads; that means, there is already some synchronization that unless all threads finish their work, the threads who have finished their work have to wait.

In case we do not want that synchronization because, synchronization means wait time for certain threads, it means latency, it means overhead, we can use the clause no wait that the threads will not wait for other threads to finish the work and start working on there.

For in this particular case because, they are operating on a thread private variable this could easily have been done, we could have put nowait, and if you execute the program you will see that the program speed is faster because threads are not waiting. They are finishing their job, and going to the next step when done, no wait.

(Refer Slide Time: 22:41)



The other important construct is section construct when there is not a do loop, but there are multiple sections of the job which are independent tasks. And they can be assigned to different processors or different threads.

So, section construct divides the jobs into multiple section and asks different processors to work in different section and therefore, the different subroutines are functions that can be called in parallel different parts of the program, which has no dependency among them can be executed in parallel.

So, it is like # pragma omp sections is again a C construct, then you put # pragma section and put a section there. # pragma omp section and put another part of the job there. So, these two parts will be concurrently or parallelly executed.

So, if you write pragma omp sections and then pragma omp section one-part pragma omp section another structured block, these two blocks of the code will be executed in parallel. Again, this entire construct which is inside pragma omp sections must be part of a construct

pragma omp parallel. Because, unless you would say that this is already a parallel region using pragma omp parallel, multiple threads are not launched.

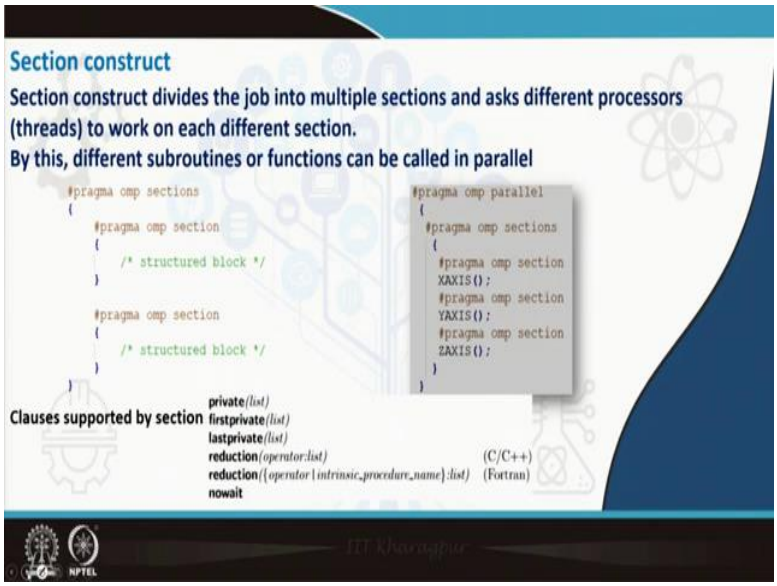
Once multiple threads are launched through pragma omp sections, you can ask if different sections will be executed by different threads in parallel. So, this is a way in which you can call different functions also, different subroutines can be called which can operate in the act independently and they can be called in parallel.

So, pragma omp parallel launches the parallel section, launches the parallel construct inside the parallel construct, you have to use pragma omp sections. That means, multiple threads will be taking care of different parts of the code block, and each pragma omp section, each of the subroutine or the code block you can also write some part of the code, will be executed by different threads.

This is very important construct in a case; you are not working on a do loop or you are not working on a strict SIMD architecture, where same instruction is being operated over different parts of the data, but, you want to do a MIMD type of architecture, there will be different data and different instructions will be processed ,in parallel the omp section is a very useful construct.

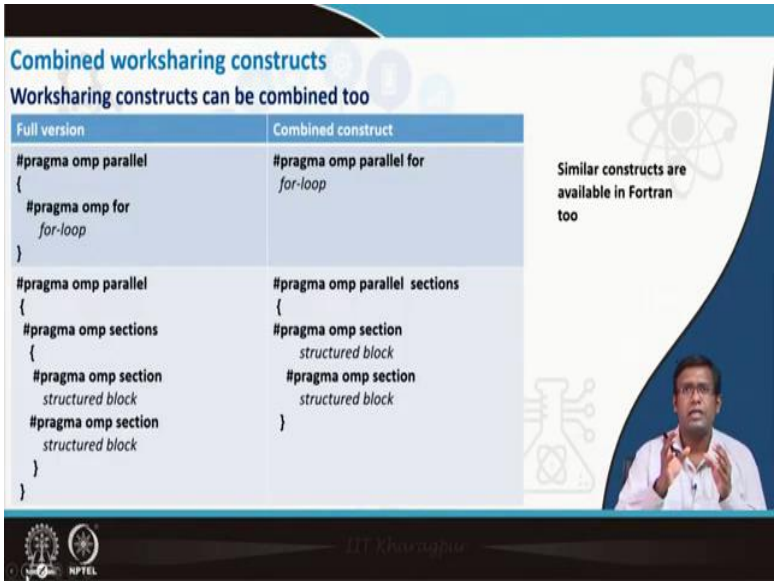
Again, here you also need not worry about how the threads will be mapped? Which thread will pick up which part of the sections etcetera? omp will automatically pick up one section and assign a thread for that, pick up another section and assign another thread for that. So, this optimization is as well as the load balancing is automatically done.

(Refer Slide Time: 26:01)



The clauses are private, first private, last private, thread private as we can see is not supported here, because, this is not do loop these are different data which each processor is working on, and these are different instructions also. So, if we call through sections next time there is probably nothing, which is specific to a thread as a private variable and go with that thread again.

(Refer Slide Time: 26:33)

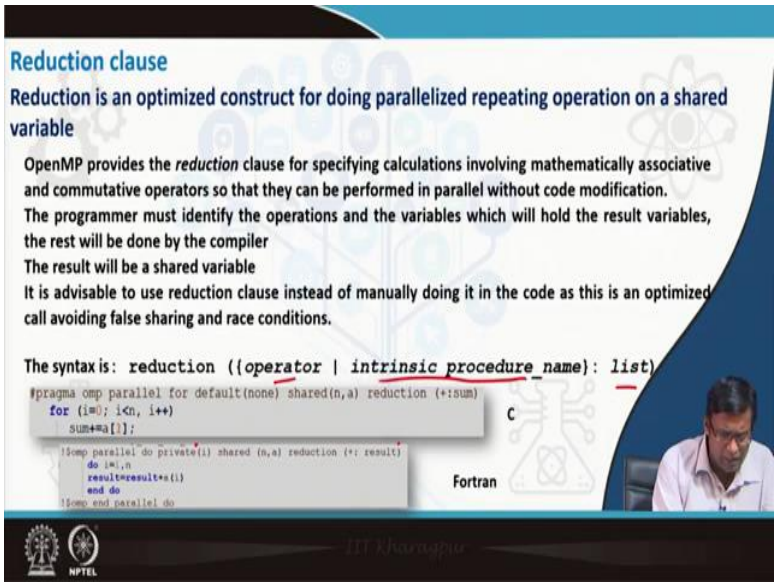


Now, we have seen that section and for they are called inside a parallel block, they have no utility outside the parallel construct, because, these constructs are by definition part of parallel

constructs. Only when parallel construct is launched, multiple threads are launched, you can do a # pragma omp for or you can write # pragma omp sections. You can execute the threads for parallelization of a for loop or for a sections block, only when its already these threads are available. Already it is a part of parallel constant. So, combined work sharing constructs are there which says that for is always parallel for, there is no non parallel for.

So, in certain case if there is only one for loop inside one particular parallel construct, you use # pragma omp parallel for. Or if there is only one sections loop inside one parallel section you use # pragma omp parallel sections. But, if there are multiple parallel regions inside one parallel construct you cannot use that, but in other cases there, they make the programming simple.

(Refer Slide Time: 27:52)



Another very important clause is the reduction clause. Reduction is an optimized construct for doing parallelized repeating operation on a shared variable. It is often required we will see an example later that, one particular shared variable is there ,multiple threads are trying to get do summation of some data, and everybody is getting its local sum and finally, this local sums up to assemble and a global sum has to be obtained; or you need to find out maximum of large number of integers, and many threads are given and they are looking into certain part of the do loop some of the numbers, they are taking and trying to find out what is the local maxima.

Finally, you need to get a global maximum. So, there are many cases when you need to find the global value, but you are trying to do it over many small local values. And reduction is the

clause when this operation that you have a shared memory variable, you are trying to find out maxima or addition or something on that shared memory variable and doing.

Basically, same operations by all the threads on that data, on the shared variable and the optimized way to do it because, it's a shared memory access variable which is being accessed by multiple threads, there can be contention, there can be rest conditions, there can be false sharing. So, to establish cache coherency, there will be certain latency in the threads also, to do it in an optimized way reduction clause is important. It's a very important clause. We will see some of the examples later.

OpenMP provides a reduction clause for specifying calculations involving mathematically associative and commutative operators, so that they can be performed in parallel without code modification. So, we will see some of the examples later. The programmer must identify the operations, and the variables that will hold the result variables, and the rest will be done by the compiler. Say you need to find out the sum, the programmer has to find out that the operation is sum and the summed variable is say total, the where is the sum after the summation it will be written in a variable name total and the operation is sum.

Specify that there is a reduction operation by all the threads, and the local totals will be added to the final total through the sum operation. Compiler will take care of the rest; the compiler will divide it among threads and get the local values to sum it up. The result has to be always a shared variable. The idea of reduction is that the same shared memory variable is being calculated by multiple processors, through the same operations that means, if I say the variable total it is the sum of some variables, it is not the sum of some variables and multiplication of some other variables. It is the sum of some variables and this sum summation process is over a large number of integers, which given a distributed amount of different threads reduction is taking care of the summation.

It is advisable to use reduction clause instead of manually doing it in the code, because, this optimized call avoiding false sharing and race condition, and , it is a shared memory variable and this shared memory variable is coming from, several private variables which are doing the local sums or local maxima or local multiplication, and then finally, assembling everything bringing them together and getting the shared variable.

So, manually it can also be done that you get all the local sums and you write that the global sum is some of these local sums. However, if you try to do it manually because it's a shared

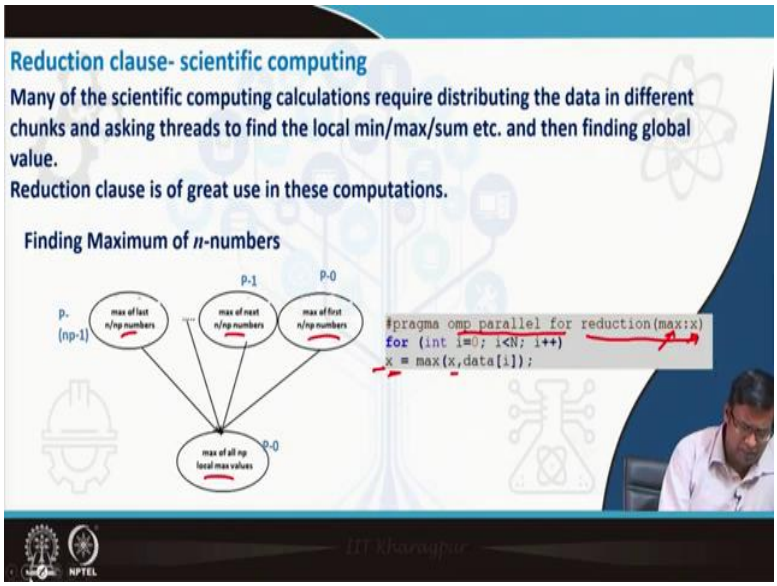
memory variable accessed by all the processors, it can give you some of the issues related to the cache memory.

Therefore, it's advisable that instead of doing it manually instead of writing a code for taking local sums and finding global some is not specifically sum it can be a different procedure; use a reduction operation.

The syntax is just write reduction and then the operator name, the intrinsic procedure name (operator is summation, multiplication, type of things and the intrinsic procedure is maxima, minima average), and then list which are the variables, what is the array that has to be taken for the reduction operation

So, one of this pragma omp for default shared and in this for loop, sums all the variables in a matrix. There is an array and summation all the elements of the array will give you a sum. So, sum + = a[i] is the for loop and you write reduction. So, if we just write this each thread will try to find its own sum, and then we have to do something because they are trying to all threads will try to write to the same shared memory location, we have to do something to take care of that. But, if you just say that because multiple threads are doing this sum, this is a reduction operation with addition of the data of a[i], multiple threads are doing a[i] and the output variable is sum, then this is already taken care of .In Fortran also shared its result all the variables are writing result = result + a[i]. So, all the elements of the matrix a[i] are written as summed up and result. This do loop is parallelized, but this is operating over the same variable and doing basically addition so, reduction + result, that will take care of it.

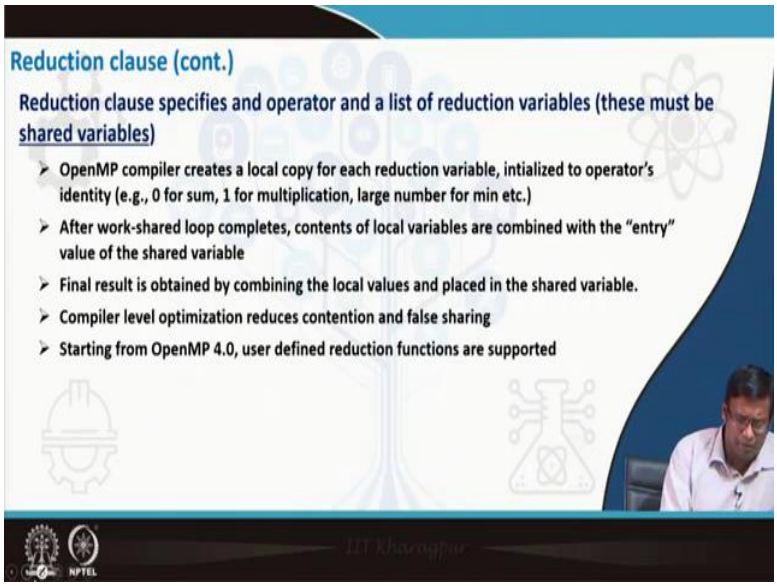
(Refer Slide Time: 33:55)



Many of the scientific computing calculations require this type of work, that distribute data in different chunks, ask threads to find the local minimum, maximum, sum etcetera and then find the global value. Reduction is of great use in these computations, we can see that if we have to find out the maximum of n numbers and each processor gets their n /n p. So, each processor has one thread which gets n/ n p numbers and finds their local maxima. All these local maxima are taken care of by one of the processors and it finds the global maxima of all the local maxima. So, this operation can be easily done by a reduction operation that x is maxima of x, data[i]this is finding out maxima in case it's a single processor thread.

In, multi-processor execution writes that its pragma omp parallel for and this pragma omp parallel for, has the clause that finds maxima, this is an intrinsic operation of the variable x. It is finding the maximum of a certain variable and writing it in the location x. So, you will get the output. This is extremely useful in many operations. We will see some examples later.

(Refer Slide Time: 35:14)



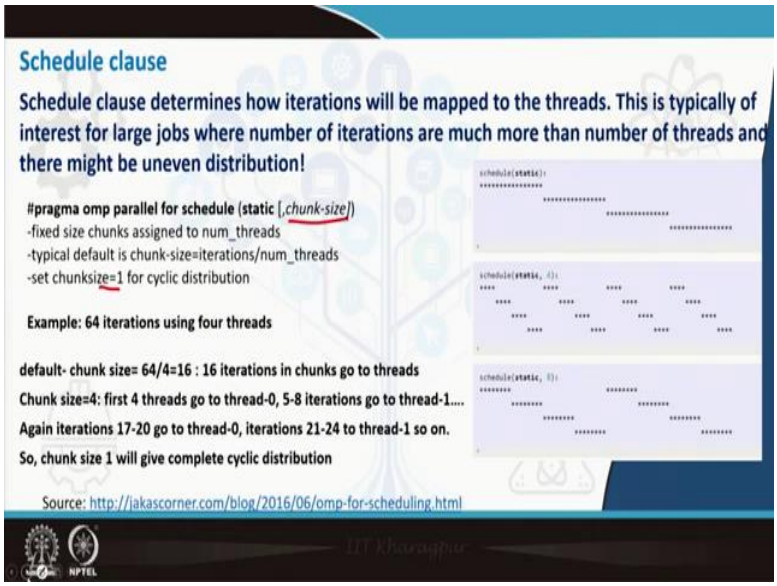
Reductions clause specifies operation and a list of variables. When reduction is done, you have to specify which operator will be active and what are the variables on which it will work.

When you have to use a reduction operation OpenMP first compiler first creates a local copy for each reduction variable, initialized to the operator's identity say, it starts with sum, it starts with 0, if it is multiplication it starts with 1. If it is finding out minima gives a large number and then finds what is the minima?

So, this is basically how you do the sum or find minima ;in a single processor, reduction asks all the processors to get local copy of these initialized variable, and then after work shared loop completes all local variables have their local output, and this combines with the entry value and goes to the shared variable, and the final result obtained by combining all the variables and placed in the shared variable. Compiler level optimization is there which takes care of the contention reducing and false sharing, in the next lecture we will see an example.

Before OpenMP 4.0, for certain operations only reduction was allowed. But now, user defined functions and reduction are also supported you can say that, this type of function I want to operate in a reduction mode, that all the threads will perform this operation and then their results will be taken care of by master thread, and the same operation will be executed over the results of all the threads over all the local results. You can do it in your customized way from OpenMP 4.0.

(Refer Slide Time: 37:06)



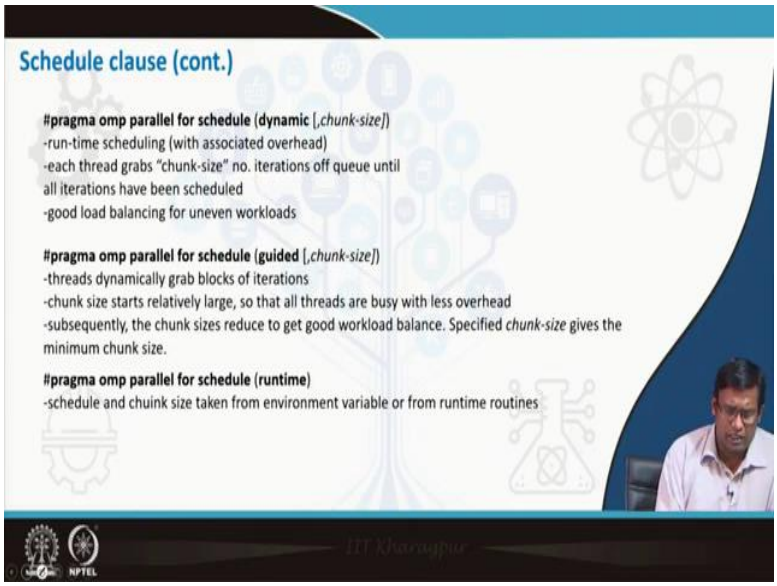
Well another important clause is schedule clause. Schedule clause determines how iterations will be mapped to threads. In case we have a large number of iterations, we have seen that OpenMP does some mapping, OpenMP distributes some of the iteration to some of the threads in a load balanced way. But, if there are a large number of threads and the OpenMP can result in uneven distribution, you need to control it, you can use shared variables.

Pragma omp for parallel for schedule static is one way of using that and then the next is followed by chunk size. So, once this is a static schedule there are fixed chunks of iterations assigned to all the threads, and typically default (if you do not specify anything), the total number of iterations divided by the total number of threads that will be the chunk sizes.

We can see some examples in case we have 64 iterations using 4 threads. We say that pragma omp schedule static, we do not specify any chunk size ,16 will go to each of the threads. In case we say that schedule is static if chunk size 4, so, first 4 will go to thread 0, thread 1, thread 2, thread 3, and then, there will be some cyclic distribution of the threads. It's not first 16 will go to thread 0 first 4 will go to thread 0, then 5 2 8 will go to thread 1 so on.

In case we use chunk size 1 it is really a cyclic distribution 1 to 16 will go to thread 0 to 15, then 17 to 32 will go to thread again go to thread 0 to 6 15 so on. So, there is a cyclic distribution of iterations which is sometimes important especially, if you want to avoid some less condition and sharing of data by the threads.

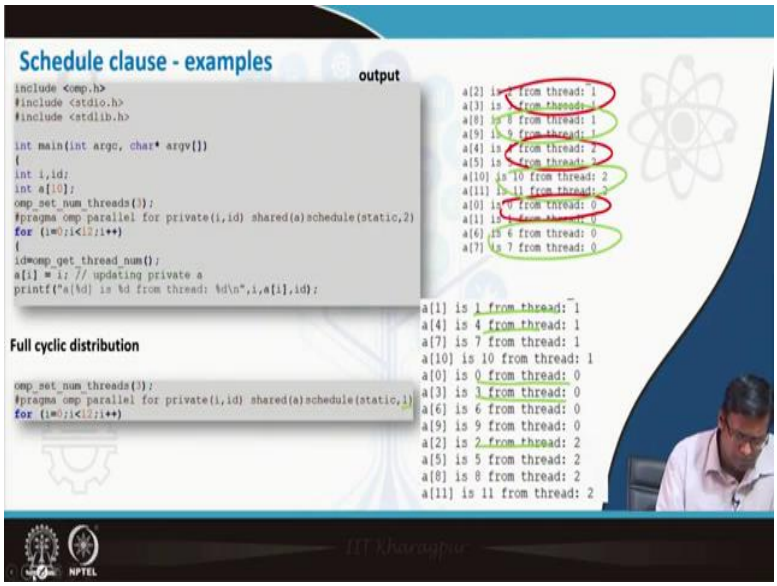
(Refer Slide Time: 39:16)



There are other parts of the schedule clause pragma omp, parallel schedule dynamic which dynamically allocates the threads. And each thread grabs some of the chunk size and depending on, what is the workload it dynamically allocates the thread. But, as it does run time and dynamically there are certain cases, when at every iteration you need to reallocate the number of threads, and there is some associated overhead.

The other way is schedule guided, where initially some of the blocks are given to some of the threads, and then it keeps on reducing the number, because you got a reduced number of threads after, you initially assign some threads to some block, you start distributing them. The guided and dynamic has one good advantage that they take care of the resource you were actually using. That means, if you are using some machine where some of the nodes are occupied by some other job, static will still give a number of iterations and assign some number of iterations to those nodes. However, guided and dynamic can understand that there is some extra loading on these nodes and put some of the jobs in some other nodes. There is a runtime which looks into environment variables and runtime associates the job.

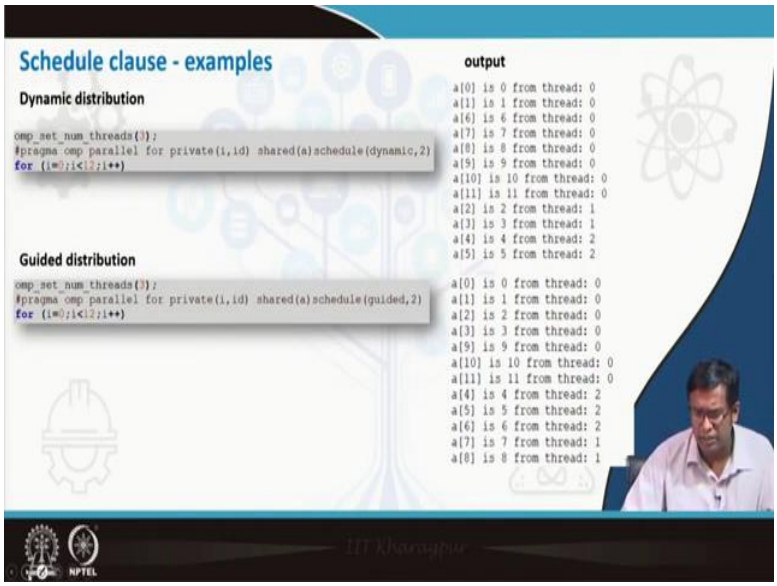
(Refer Slide Time: 40:41)



There are some examples like in case of schedule static 2, there are 12 iterations to be distributed in 3 threads. Each thread gets a uniform amount of work and if this is static 2; that means, 0 and 1 goes to thread 0, 2 and 3 goes to thread 1 sorry 4 and 5 goes to thread 2. Then we can see that 6 and 7 again goes to thread 0, 8 9 again goes to thread 1, 10 11 to thread 2.

In case of cyclic distribution static 1, we can see 0 goes to 0, then 1 goes to 1, 2 goes to 2,3 again goes to 0 ,4 again goes to 1. So, there is a cyclic distribution of the threads. There is no chunk going to any of the threads, 1 thread iteration to 1 thread, another iteration to the next thread, the subsequent iteration to the next thread and so on.

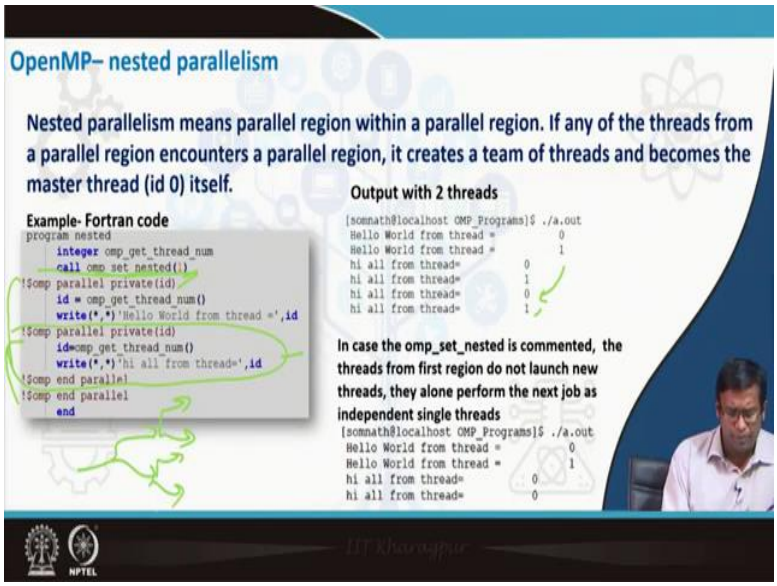
(Refer Slide Time: 41:45)



Also, we can see a dynamic and scheduled distribution that, there is actually uneven distribution if you use dynamic and guided comments. Guided is more uneven in dynamic it is less uneven, they are trying to estimate; however, the loads on different processors and allocating the threads accordingly. IN dynamic it makes chunks of 2 and distributes it to different threads. In guided minimum 2, it's assured that minimum 2 threads will go to minimum 2 iterations will go to 1 thread. But there can be different number of iteration 3 iterations going to this thread, different number of iterations going to thread.

They do not assure load balancing in bookkeeping sense, but they assure load balancing looking into the hardware capacity of different systems.

(Refer Slide Time: 42:44)



So, there is another thing called nested parallelism called parallel region within a parallel region; that means, multiple threads will be there and the threads are further launching sub threads. This has to be done by per nested environment has to be set by omp set nested. We can see that first there are threads launched which will launch inside a parallel construct threads will write their own id, and then, there will be another parallel section within the parallel construct. This is a parallel construct within the parallel construct, there is another parallel section.

So, now each of the threads will become the master thread and launch the slave threads. So, thread 0 and thread 1, in the next section which was the thread 1 in the first parallel construct in the nested parallel section, it got a thread 0 number and it launched another thread.

So, if there are multiple parallel loops, we can call nested parallelism and do that. In case, we turn up the nested parallelism, they do not launch any further threads. They simply launch 1 thread in the sub nested parallelism, multiple threads do not act there. But each thread again becomes a master in the nested parallel part. Whether, it will be nested parallel or not that has to be set up by the environment variable omp set nested, if its clause is non-zero it will be a nested parallel region. Nested parallel means there are multiple threads launched and each thread is further launching another set of threads ok.

(Refer Slide Time: 44:29)

(Refer Slide Time: 44:30)

So, these are the references and we looked into through these discussions' directive constructs and clauses, recursion and scheduling clauses and nested parallelism is introduced. We will see some important applications of the OpenMP program, with a focus to critical issues in shared data handling. How if multiple threads are trying to write to a shared data, which is often done in a practical scientific computing problem, how that is done by OpenMP; based on the basics we have discussed so far, we will see it in the next class.