

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

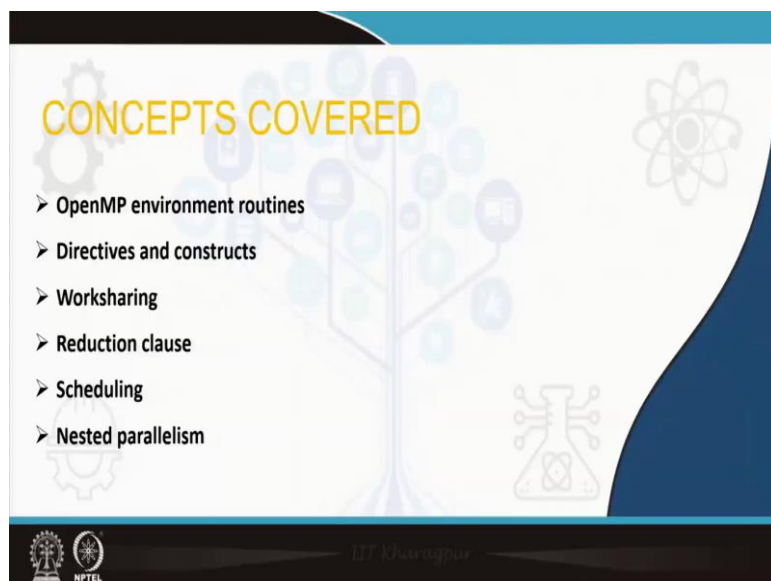
Module - 02
OpenMP programming
Lecture – 15
Essentials of OpenMP Programming

Welcome, we are discussing the topics in the class High Performance Computing for Scientists and Engineers. We are now discussing the second module of this course OpenMP programming, and this is the part of ongoing lectures on OpenMP programming and today we will discuss Essentials of OpenMP programming.

In the last 3 classes, we have discussed some important features of OpenMP programming. I tried to introduce; what OpenMP is to the students and then we discussed how OpenMP works at system level, and what are different types of memory handling in OpenMP specially shared and private variables.

We will discuss today as the name suggests the essentials of OpenMP programming that are some of the constructs and directives which are essential to write an OpenMP program and the idea is that you can start writing your own OpenMP program. ;also are aware about this particular construct and directives and the associated clauses .

(Refer Slide Time: 01:49)



Work sharing; in OpenMP while discussing work sharing I will again touch upon this point that OpenMP gives us a parallel paradigm which is extremely portable; that means, you can go to large supercomputers with say NUMA type of access even multi processor system or you can look into your own desktop computer which has multiple processors and use the OpenMP program.

Therefore, when you are using simple systems to run your OpenMP program it becomes also important that the resources are effectively used by OpenMP so that you get right parallel performance. That is, how the parallel distribution will be done, how the mapping will be done, how the load balancing will be done, these are important, but fortunately because OpenMP is also written for people who are beginners in parallel programs.

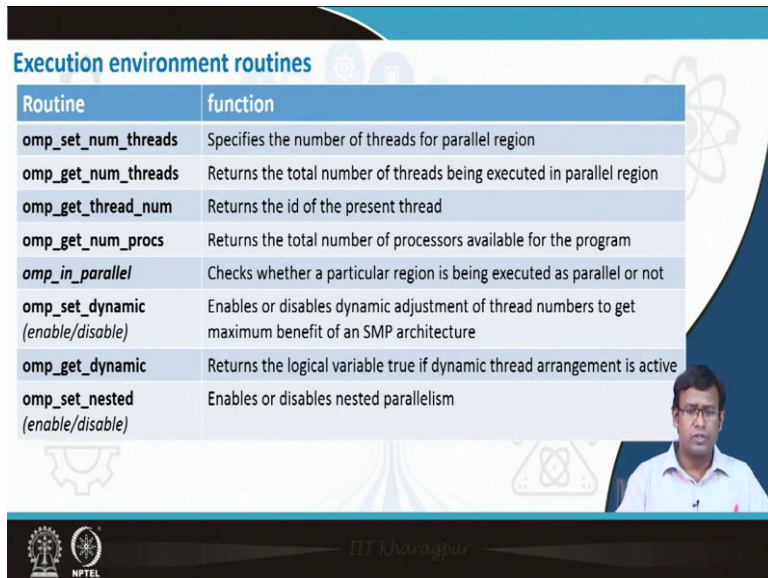
They usually have their own scientific computing algorithm and they want to utilize parallel efficiency using different sources of machine and most of the time they are using their desktop or PC or a small workstation they are having.

So, for them learning the details of load balancing and finding out which processors are free at which instant is can be difficult. Also for this type of systems which are not dedicatedly developed for parallel computing like your PC or like an workstation ,so, in order to understand which processors are relatively free and how the jobs will be distributed that requires some different level of expertise which is not always expected from a person who is working on OpenMP because OpenMP is mostly for scientists and engineers and people from other domain who have some application and they want to get the parallel performance out of it.

So, while we look into work sharing or while we look into the scheduling also ;we will see that the load balancing part is some way automatically taken care of by your OpenMP compiler and we also have certain flexibility on load balancing. So, it is not that it is a black box for us, so we have some flexibility also on load balancing through different scheduling causes.

Recursion is an important clause in OpenMP, you will see about recursion and nested parallelism; that means, within a parallel loop you are again launching another level of parallelism.

(Refer Slide Time: 05:23)



Routine	function
<code>omp_set_num_threads</code>	Specifies the number of threads for parallel region
<code>omp_get_num_threads</code>	Returns the total number of threads being executed in parallel region
<code>omp_get_thread_num</code>	Returns the id of the present thread
<code>omp_get_num_procs</code>	Returns the total number of processors available for the program
<code>omp_in_parallel</code>	Checks whether a particular region is being executed as parallel or not
<code>omp_set_dynamic</code> (enable/disable)	Enables or disables dynamic adjustment of thread numbers to get maximum benefit of an SMP architecture
<code>omp_get_dynamic</code>	Returns the logical variable true if dynamic thread arrangement is active
<code>omp_set_nested</code> (enable/disable)	Enables or disables nested parallelism

So, if you go to OpenMP environment routines, these are the routines which are certain different functions. These routines do not give you parallelized output. If you have a code block with this routine you cannot parallelize them. However, these routines help you to set up the parallel environment as well as to control certain features in the parallel part ,also to understand how the parallel environment works.

So, you will see some of the examples like the few we have said; `omp_set_num_threads`. This is a function, if you call this function the output will be the number of threads in the parallel regime. So, while the OpenMP program is running many times it will be important to know actually how many threads are launched and especially for a large program running for a substantial amount of time you really cannot scroll back and see what has been set as default. So, this function can be important.

Similarly, what is the number of a particular thread ;you might have to run if you loop over a particular thread that a particular thread will do something else than the other thread. So, you need to know what is the number of the thread `omp_get_thread_num` is what it will return the id of the particular thread or if we go to say `omp_get_num_threads` it will return the total number of threads being executed in the parallel regime.

So, one can probably foresee the situation that it is a huge code and at different instances just by using `omp_set_num_threads` or by using the number of threads as a clause in the parallel directive ; you are changing the number of threads at different instant, but it is a huge code and

the execution is also lengthy. So, at different regimes how many threads are running ,to know that it , you can get `omp_get_num_threads` which will return the number of threads.

Also, the number of threads is not the same as the number of processors. We have seen that the number of threads are many times more than the number of processors. You have written an OpenMP job, you go to some platform, you run that OpenMP job. Maybe some of the processors are occupied, some of the processors are not being allotted to your job. When you run the job, what is the number of processors that you are using? Threads are the units of parallelism and processors are physically the CPUs which are crunching numbers for you. So, for that `omp_get_num_procs` will return you the number of processors.

Similarly, another important environment routine is `omp_in_parallel` and we know that the OpenMP program has both parallel and sequential components. There is a master thread which is active throughout. When you come into the parallel part it forks to multiple threads ,once the parallel part ends the threads join and again the master thread is active. Again, we consider the case that we are really working with a huge code with a large code with 10,000s of lines code and the execution is also taking days.

So, to see whether one particular instance is being executed in parallel mode by multiple threads or only by the master thread it is not always possible to go into the thread program and find out on that instant that what are the number of threads there whether it is a parallel part there. So, this function `omp_in_parallel` can return you that.

Similarly, `omp_set_dynamic` ; if you provide a non-zero value as its clause it will say that the number of threads will be not pre-decided by the environment set or by the default call rather the number of threads will be dynamically allotted by the SMP architecture. What is the architecture? How many threads are there? Based on that OpenMP will dynamically alert the number of threads.

So, even if the user who has given the number of threads goes to a different computer the number of threads might be different. So, I discussed it a few minutes back that OpenMP is really written for people who are using different platforms for doing their scientific computing job and are not many times well aware of the exact hardware they are using and what is the best way to utilize that hardware.

Therefore, OpenMP compiler itself does that job that it can look into the program and look into the hardware and set that what is the best number of threads so that the maximum benefit can be obtained and that can be done by `omp_set_dynamic`. `omp_get_dynamic` will return in true or false whether the dynamic arrangement of thread is initiated or not; what is the present environment at the threads statically defined by the user or the dynamic number of threads being used .

`omp_set_nested` it enables or disables nested parallelism. Nested parallelism is a parallel loop within a parallel loop it is. So, it is one more order of parallelization. We will discuss these things later.

So, with these environment variables we can check which environment the parallel program is actually utilizing. Also, we can tune some of the environment parameters to get better performance; to say ,that is the hardware we are using, and by using these environment parameters we can get best performance. We can also do something in the code , though we are running single instruction multiple data code, but there can be some issues where some of the threads will run a different instruction.

So, it will be more towards the multiple instruction multiple data code and that can be done by identifying those threads which will run different instructions and that identification can also be done through the environment variable. These environment variables are not executable. These variables will not the instructions which will execute the program in parallel mode, but rather these are the environment variables which will help us to set the environment as well as to know about the parallel environment.

(Refer Slide Time: 12:39)

Execution environment routines- examples

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Present thread id %d \n",id);
    }
}
```

Output

```
[somnath@aliivardi OMP_Programs]$ ./a.out
Present thread id 0
Present thread id 1
Present thread id 3
Present thread id 2
```

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int nthread = omp_get_num_threads();
        printf("This thread id %d from total %d threads \n",id,nthread);
    }
}
```

Output

```
[somnath@aliivardi OMP_Programs]$ export OMP_NUM_THREADS=8
[somnath@aliivardi OMP_Programs]$ ./a.out
This thread id 1 from total 8 threads
This thread id 5 from total 8 threads
This thread id 0 from total 8 threads
This thread id 7 from total 8 threads
This thread id 3 from total 8 threads
This thread id 4 from total 8 threads
This thread id 6 from total 8 threads
This thread id 2 from total 8 threads
```

IIT Kharagpur

So, some of the examples are; first is we are running a program which will write from each thread that this is the present thread id and to run it in multiple threads we have to call pragma omp parallel. So, it will launch a number of threads.

While running this if we can try to set the number of threads inside the program. We can do it as a clause of parallel or we can write the environment variable omp_set_num_threads threads. So, the number of threads is set and this runs over this particular number of threads.

And what is the number of identities of one particular thread that will be obtained by the environment function call that omp_get_num_threads that will return the id which is the thread id and we will read this ;we have seen this example in last class also discussing the hello world program .

Similarly, if we need to know that we do not set the number of threads inside the program. We execute this program and the number of threads is set by the default or by the environment variable outside the program. While running the program we need to know how many threads are running and that will be done by omp_get_num_threads ,that will get us to give us the number of the threads.

Earlier omp_get_thread_num was giving us the number thread id of a particular thread. Here, it will give us the total number of threads. So, even if this is running in parallel for all the processors get_num_threads, the total number of threads is fixed and omp_get_thread_num

,what is the id of this particular thread, is different for different processors. So, this will write that the total number of threads are 8 and thread 0 to 7 are running and each of this thread will write their own thread id.

So, these are few environment routines example, similarly so we have to call the environment variables as function or we have to take the output of the environment variable call as an integer ; if you are trying to set something this is just a function call with the integer input which is the environment that we are going to fix. If we want to get some output of this thread, we have to call the function and associate it with an output.

(Refer Slide Time: 15:17)

Important OMP routines- directives

Important terminology

OpenMP Directive- In C/C++, a #pragma and in Fortran, a comment (!), that specifies OpenMP program behavior.

OpenMP directive syntax in C: #pragma omp directive-name [clause[[:clause]]...] new-line
in Fortran: !\$omp directive-name [clause[[:clause]]...]

The different types of directives are:

- (i) Declarative directive: The directive that is placed only on declarative purpose. threadprivate is the only declarative directive.
- (ii) Executable directive: A directive that is not declarative, but placed in an executable context.
Example: #pragma omp parallel num_threads (10) *Construct*

```
{  
/* do work here */  
}
```
- (iii) Stand-alone directive: An OpenMP executable directive that has no associated user code
Example: #pragma omp barrier
- (iv) Loop directive: An OpenMP executable directive whose associated user code is a loop nest in form of a structured block

NPTEL

IIT Kharagpur

Video inset: A man speaking.

So, apart from the environment routines, important routines are which executes the parallel program, which instructs the compiler that this is a parallel part and it should be executed in multiple threads and this is called a directive. Some important terminologies are required here to know what is directive and what are constructs.

These two are very important terminologies. Omp directive in C, C++ statement starting with # pragma or in a Fortran statement starting with a comment(!) or say colon dollar. It specifies the OpenMP program behavior.

OpenMP directive syntax in C is #pragma omp then the directive name, then the clause, clauses can have few other clauses and the new line starts. So, this specifies that this is a part of the OpenMP program and this will be executed in parallel.

Anything we started with `# pragma omp` with the directive name and directive name. Depending on the directive name what will you do in parallel mode that will be directed. In Fortran this is colon dollar and directive name and there are multiple clauses. We will look at the clauses there.

There are different types of directives. One is a declarative directive. This directive is not instructive. This directive is not executing anything parallel, but it is declaring something which is parallel and will have certain value for all the threads. Till today's date the directive `threadprivate` is the only declarative directive. We have looked into `threadprivate` directive earlier in last class, that some variable which is private to a thread and after the thread is destroyed and again the same thread will be launched this variable in the private memory will remain for that particular thread.

There can be more declarative directive which openMP may add later, but right now there is on one directive which is declarative and `threadprivate`; it starts with `# pragma omp`.

Executable directive a directive that is not declarative, but placed in an executable context. So, when there is an executable directive it executes. It does something and what it does it parallelizes the job. It asks multiple threads to work on certain part of the program in parallel and use the concurrency.

One of the examples is `pragma omp parallel num_threads (10)`; that means, this parallel part will run in 10 threads and do work here. So, each of the threads will do their work. Whatever we asked, write `a = a + 1` or something `a = i` or right `printf hello world` something like that.

So, all the threads will do the same thing. If we have something in parallel whatever given will be executed by num multiple threads. The directive `pragma omp parallel num_threads parallel`, `pragma omp parallel` this is the directive and `num_threads` part is known as clause of the directive. So, we have seen that directives come with the clause. So, this is the clause that executes this parallel part in 10 threads.

Standalone directive; this directive has an associated work with this . A part of the program is associated with this directive. In case there is no program associated with the directory, it is still an executable directive and we call it to be a standalone directive .Only the particular statement of directive `# pragma omp something` that is sufficient for that directive. It does not require any associated code.

#pragma omp barrier; that means ,at this point all the threads should wait until there is a barrier on the thread till all remaining all the threads finish their work. This is called a synchronization directive, also this is a standalone directive. So, for this directive you do not need anything like this after this directive if you do not need anything like this it will execute on its own.

Loop directive: OpenMP executable directive who's associated user code is a loop or a loop nest or in form of a structured block. So, if the structured block is a loop say for loop or a do loop something like that then we call this directive to be a loop directive. We write a directive #pragma omp for and then we write a for loop; that means, this for loop will be executed by multiple threads. Multiple threads will take care of different iterations of the for loop and they will execute it, which is called a loop directive.

(Refer Slide Time: 21:11)

Important OMP routines- constructs

Important terminologies

Construct: An OpenMP executable directive (and for Fortran, the paired end directive, if any) and the associated statement, loop or structured block, if any, not including the code in any called routines. That is, in the lexical extent of an executable directive.

Example:

```
int dequeue(float *a);
void work(int i, float *a);
void critical_example(float *x, float *y)
{
    int ix_next, iy_next;
    #pragma omp parallel private(ix_next, iy_next)
    {
        #pragma omp critical (xaxis)
        {
            work(ix_next, x);
            ix_next = dequeue(x);
        }
        #pragma omp critical (yaxis)
        {
            work(iy_next, y);
            iy_next = dequeue(y);
        }
    }
}
```

Parallel construct

Critical constructs

Region: A region may also be thought of as the dynamic or runtime extent of a construct or of an OpenMP library routine. Region includes any codes on called routines as well as implicit codes from OpenMP.

IIT Kharagpur

NPTEL

So, a few more important terminologies.

Construct ;as we said after the directive especially for executable directives there is a block of code which it executes. So, an OpenMP executable directives (and for Fortran the paired end directive if any)and the associated statement, which is either a loop or a structured block if they are not including any code which is called as routine; this is called a construct. This is the lexical extent of the executable directive.

So, what this directive is doing, the next code block associated with the directive written after the directive till the next brace ends is called the construct. So, we can see an example that this

is a program omp. Some function is called as by omp critical inside omp parallel and if you see `pragma omp parallel x,y shared` (this is a clause) `private ix_next, iy_next` this is also a clause. So, these are the clauses. So, at this point we know that some jobs will be executed in parallel and this part of the job will be executed in parallel. So, this is the entire thing called the parallel construct. This line is the directive and directive plus the associated code block, the associated structured block of the code is called the parallel construct; that means, when I am trying to execute this line the entire construct will be executed.

So, if we call the function what is inside the function if we write somewhere the function works what is inside the function that is not part of the directive. Only these lines are part of the construct and we can see also 2 other directives `omp critical x-axis`, `omp critical y-axis`.

The critical directive tells that though it will be executed in parallel, at one instant only one thread will execute this line. Though this entire block will be executed by multiple threads, but at one instant only one thread will be allowed to execute this. This is called a critical directive and these are also constructs because this directive and the associated line that `ix_next` will be executed one by one by the threads. This is part of the critical construct; the directive followed by it is a statement. Similarly, there is another critical construct.

So, the `# pragma` line that is the directive and the next part of the code which is a in form of structured block or in form of a nested loop associated with the directive ; through the directive the next part of the code will be associated in parallel in the in the manner that the directive is suggesting; the entire thing is called the construct. A region may also be thought of as the dynamic or runtime extent of the construct or an OpenMP library.

Now, if we see here that when we run this construct, we call a function. Here, this function we say that it is what is inside the function is not part of that construct. The construct is only these lines, but this function is also being executed when we call the multiple threads ;all the threads are also executing this function. So, if we consider what is inside the function also, the entire thing is called a parallel region.

So, whatever the runtime extent of the construct, the function call and what the function is doing or if any other implicit code or implicit functions are being called, the entire thing is called a region . So, this complete part with this work will be called the parallel region.

(Refer Slide Time: 25:57)

Parallel construct

C: `#pragma omp parallel [clause[,clause]...]`
(structured block)

Fortran: `!$omp parallel [clause[,clause]...]`
structured block
`!$omp end parallel`

Clauses supported by parallel construct

<code>if (scalar-expression)</code>	(C/C++)
<code>if (scalar-logical-expression)</code>	(Fortran)
<code>num_threads (integer-expression)</code>	(C/C++)
<code>num_threads (scalar-integer-expression)</code>	(Fortran)
<code>private (list)</code>	
<code>firstprivate (list)</code>	
<code>shared (list)</code>	
<code>default (none shared)</code>	(C/C++)
<code>default (none shared private)</code>	(Fortran)
<code>copyin (list)</code>	
<code>reduction (operator : list)</code>	(C/C++)
<code>reduction ({operator intrinsic_procedure_name} : list)</code>	(Fortran)

So, one of the parallel constructs is in C it will look like `# pragma omp parallel` that is the parallel construct. Parallel construct means when we write this with certain clauses whatever part of the program in this structured block, that will be executed by multiple threads. When we write `pragma omp parallel` whatever the structured block of the program we write after that that will be executed by all the threads concurrently.

In Fortran it will be `!$omp parallel` then the block and then we have to end the parallel block.

The clauses will be `num threads`, `private`, `shared`, `firstprivate`; `if` are the logical clauses; that means, if this is true only then this parallel part will be called. If this is not true then this parallelization will not be done.

Similarly, `num threads`, `private` first ,`private` `shared` type of variables. If there is any default if there is any `copyin` all the variables will be copied from master thread to the remaining other threads; `reduction` will look at the `reduction` call clause in detail later. So, all these can be clauses of the parallel.

So, we can say that when this parallel loop will be launched when the construct is working this will take care of this particular clause. If some of the clauses are true, then this construct will work. If the clause specifies these are shared and these are private variables when the construct is working these variables will be considered like that manner.

(Refer Slide Time: 28:01)

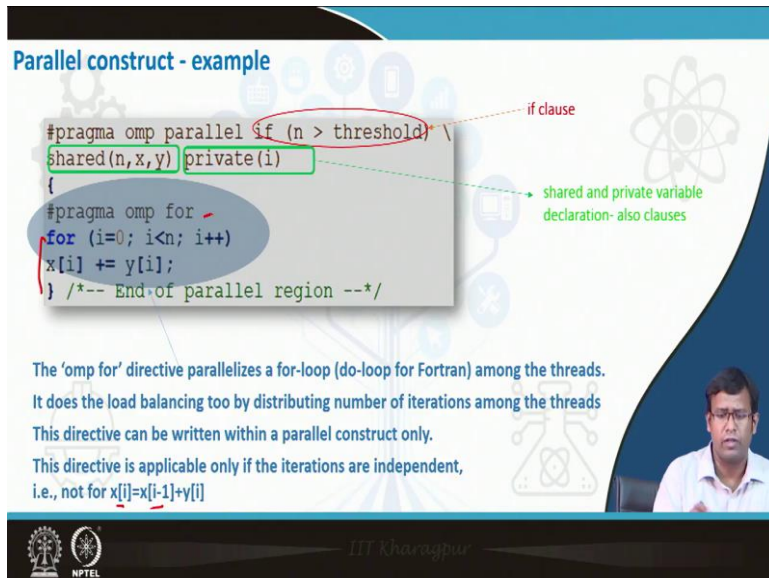
Parallel construct - example

```
#pragma omp parallel if (n > threshold) \
shared(n,x,y) private(i)
{
  #pragma omp for
  for (i=0; i<n; i++)
  x[i] += y[i];
} /*-- End of parallel region --*/
```

if clause

shared and private variable declaration- also clauses

The 'omp for' directive parallelizes a for-loop (do-loop for Fortran) among the threads. It does the load balancing too by distributing number of iterations among the threads. This directive can be written within a parallel construct only. This directive is applicable only if the iterations are independent, i.e., not for $x[i]=x[i-1]+y[i]$



So, you will see another example. So, `pragma omp parallel if (n > threshold)`. If n is greater than threshold, then only this parallel part will work. If this parallel part is working, n , x , y these variables are shared variables and i is a private variable and then after that we can see there is another directive which is `pragma omp for` and then there is a for loop.

So, this is the construct. Inside this construct there is another directive there; this entire thing is the construct. In the construct, there is another directive and this directive is associated with another set of constructs. What is this directive associated with? With a for loop. So, when we execute this first is this if clause.

If the clause is true then only this part will be executed. Shared and private variable declaration there are also clauses. The `omp for` will parallelize a for loop if it is Fortran a do loop among the threads.

If we launch say $n = 8$, total n iterations. So, total 8 iterations 0 to 7. If we launch 4 threads 2 iterations will go to each thread and that way the parallelization will be done so that each of the threads will take care of 2 iterations. So, what is inside this for loop that will be automatically parallelized and distributed across many threads following some load balancing.

We really do not need to take care of the load balancing. OpenMP is doing load balancing for us. Only before a for loop we write `pragma omp for` it will be parallelized and the number of

threads that has been launched through the omp parallel part; all these threads will take care of this parallel loop and it will be accordingly executed concurrently.

It does the load balancing too by distributing a number of iterations among the thread. This pragma omp for directive which is for parallelizing a for loop, we really do not need to do anything else only just to call an OpenMP directive. OpenMP directive will itself take care of parallelization of the loop and this will also do the load balancing .

There is also something called as schedule clause, when calling the parallel loop or the for-loop clause of the directive. With this clause we can have certain control over the load balancing, but otherwise if we do not call anything by default it will also do some load balancing which is not bad.

This load balancing is most of the time quite useful and distributes the iterations among different processors. It will be parallelized in that way. The pragma omp for directive can be written within a parallel block only.

You have a for loop and pragma omp for has parallelized, it has distributed the for loop into multiple threads. When can it be done? When multiple threads are there. To launch multiple threads, you need a parallel construct. So, we need a parallel directive. So, this pragma omp for should be a part of the parallel construct only.

This directive is not applicable if the iterations are dependent. This directive is applicable only if the iterations are independent. In case the iterations are dependent that x_i is dependent on x_{i-1} we cannot parallelize it. There is no concurrency if x_i requires x_{i-1} , then x_i and x_{i-1} cannot be calculated concurrently. So, there is no concurrency and we cannot parallelize it .

So, only if each of these iterations are independent and they can be run concurrently then pragma omp for is meaningful.

(Refer Slide Time: 32:39)

Parallel for - example

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    int i, id;
    int a[10];
    omp_set_num_threads(3);
    #pragma omp parallel private(i,id) shared(a)
    {
        #pragma omp for
        for (i=0; i<10; i++)
        {
            id=omp_get_thread_num();
            a[i] = i+1; // updating private a
            printf("a[%d] is %d from thread: %d\n", i, a[i], id);
        }
    }
}
```

output

```
[sommath@localhost OMP_Programs]$ ./a.out
a[0] is 1 from thread: 0
a[1] is 2 from thread: 0
a[2] is 3 from thread: 0
a[3] is 4 from thread: 0
a[4] is 5 from thread: 1
a[5] is 6 from thread: 1
a[6] is 7 from thread: 1
a[7] is 8 from thread: 2
a[8] is 9 from thread: 2
a[9] is 10 from thread: 2
```

locally updated by each thread in its own part of do-loop

garbage

output

```
[sommath@aliivardi openmp]$ ./a.out
a[4] is 1 from thread: 1
a[5] is 2 from thread: 1
a[6] is 3 from thread: 1
a[1] is 1 from thread: 0
a[2] is 2 from thread: 0
a[3] is 3 from thread: 0
a[7] is 4196534 from thread: 2
a[8] is 4196535 from thread: 2
a[9] is 4196536 from thread: 2
```

If the iterations are dependent, then it fails:

```
#pragma omp for
for (i=1; i<10; i++)
{
    id=omp_get_thread_num();
    a[i] = a[i-1]+1; // updating private a
    printf("a[%d] is %d from thread: %d\n", i, a[i], id);
}
```

IIT Kharagpur

We can see an example. So, what we are doing here that `omp_set_num_threads = 3`, `pragma omp parallel private (i, id) shared (a)`. There is a loop `i = 0` to `9`, for that `id` is `omp_get_num_threads` and there is a variable `a` who's each of the element is the `i`th number + 1. So, `a0` is 1, `a1` is 2 so on and it will write that the value of `a` from the thread number.

What is called inside `pragma omp parallel`, that is executed by multiple threads and this is a construct and then we call `pragma omp for` and what is called within `pragma omp for` this is another construct.

So, this `omp for` is always part of another parallel construct and this here `i` and `id` is private, `a` is shared; these are the clauses. Now, there is a for loop for `i = 0` to `9` `ai` is `i + 1` and `printf` the values of `i` and this for loop is parallelized by `pragma omp for` in the number of threads. The number of threads is set by `set num thread`.

We see, 3 threads and 10 iterations. We have not done any load balancing, but OpenMP has balanced the load in some way. That zeroth thread gets 4 iterations, the first thread gets 3 iterations, the last thread gets 3 iterations.

So, it is the best load balancing possible. We write that the first 4 are done by zeroth thread, then they are done by first thread and then this is done by second thread. This is possible because this calculation is independent for any iteration and all the iterations are independent.

We do not need to know the value of a_2 to calculate a_3 or we do not need to know the value of a_0 to calculate a_7 . All of them can be calculated independently.

If the iterations are dependent, then what happens. Then we have written a_i is $a_{i-1} + 1$ and this is written in the same place. We have the same code only the for loop is replaced by this part and what will happen if we execute it.

When the first thread is operating a_0 's $i - 1$ is undefined, it will probably pick up 0, though it is outside the memory bound the stack will help.

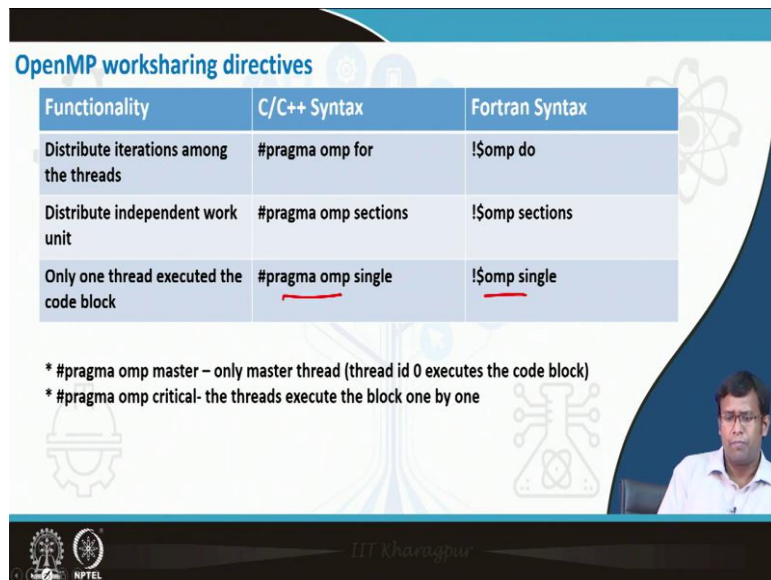
For the second thread which starts from our thread 1 which starts from $i = 4$, a_3 is not known because a_3 has gone to the first thread. Though a is a shared variable first thread has not calculated a_3 by that time. So, what is the initial value of a_3 . It is by default initialized by 0, it will pick up 0. So, the second thread will start calculating from 0. It might not pick up anything, as the values are not available.

So, the execution is quite interesting. The first thread picks up everything initial value 0 and then keeps on adding 1 with that, because it somehow, I thought that it is initialized with 0, though the initializations were not done.

The first thread was also done, initialization was by default 0. So, it again picks up 1 a_4 is the first iteration of thread 1 it picks up as value 0 and adds 1. This is also the first iteration of 1. It picks up 0 and adds 1 and then for the next values of a it keeps on adding 1 for, but this is a shared memory variable which is being accessed by multiple variables and this is not rightly updated. This is an invalid for loop.

So, the last thread actually picks up garbage values. So, these are locally updated by each of the do loop, but the last thread is working on garbage values because these values are not rightly updated and the last threads are picking up garbage values. One has to be careful to see that the iterations are independent in order to parallelize it. If the iterations are dependent you cannot parallelize it like this.

(Refer Slide Time: 37:59)

The slide is titled "OpenMP worksharing directives" in blue text. It features a table with three columns: "Functionality", "C/C++ Syntax", and "Fortran Syntax". The table has three rows of data. Below the table, there are two bullet points explaining specific directives. The slide also includes a small video inset of a man in the bottom right corner and logos for IIT Kharagpur and NPTEL at the bottom.

Functionality	C/C++ Syntax	Fortran Syntax
Distribute iterations among the threads	<code>#pragma omp for</code>	<code>!\$omp do</code>
Distribute independent work unit	<code>#pragma omp sections</code>	<code>!\$omp sections</code>
Only one thread executed the code block	<code>#pragma omp single</code>	<code>!\$omp single</code>

* `#pragma omp master` – only master thread (thread id 0 executes the code block)
* `#pragma omp critical` – the threads execute the block one by one

With this we come to discuss the different work sharing directives. Because we have seen that if there is a parallel loop it will be shared across many threads by the right directives. So, one is `pragma omp for` or in Fortran this is `!$ omp do`. This will do a distribution of the loop. If there is an iterative loop after this, the iterations will be distributed among the threads.

But many times, the jobs are not like an iteration, not like a do loop or for loop. Many times, it will be seen that there are different parts of the jobs which can be run concurrently. Think of a task dependency graph that we can find out that these are the concurrent jobs which can run, but they are not parts of iteration. They are something else, but they can be done concurrently.

We can use `pragma omp sections` and define different sections and each section will take care of each part of the job. Then we can use `pragma omp`, if there is another work sharing directive, if only one thread has to take care of that part inside a parallel regime because these are called within the parallel region. Inside the parallel region we understand that for cannot be called outside the parallel construct. It is part of a `pragma omp for` it is part of a parallel region and when we are doing work sharing already the threads are activated. Within that if a certain portion is required to be executed only by one thread, we can use `pragma omp single` that one of the threads will be asked to execute that.

There are some similar things `pragma omp master`. Only master thread that is thread id 0 will execute that code block and `pragma omp critical` we have seen that. It is in the within a parallel region one particular part is actually not parallelized; that means, one by one threads we are

executing in this part and this is usually done to avoid contention and the false sharing or shared data accessing issues; that one thread will do something next the other thread will come and work on that same variable like that.

So, this is done by `pragma omp critical` and these are the work sharing directives in OpenMP. We will look into the work sharing in OpenMP in more detail in the next class.