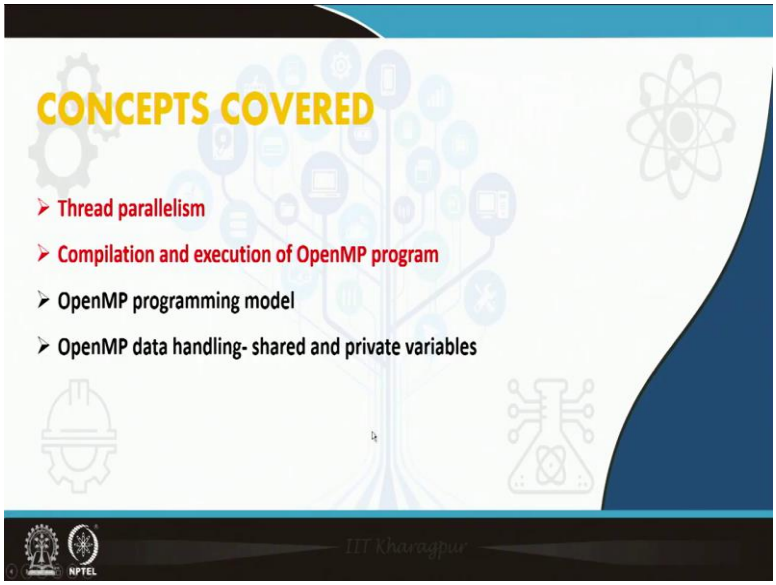
High Performance Computing for Scientists and Engineers
Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 02
OpenMP Programming
Lecture – 14
Introduction to OpenMP (continued)

Welcome, we are going through the lectures of the course High Performance Computing for Scientists and Engineers, and this is the 2nd module, where we are looking into OpenMP Programming and continuing with the lectures on Introduction to OpenMP.

(Refer Slide Time: 00:44)



In the last two classes we have seen, what is thread parallelism and how to compile and execute OpenMP programs. In this class we will discuss the OpenMP programming model and data handling of shared and private variables.

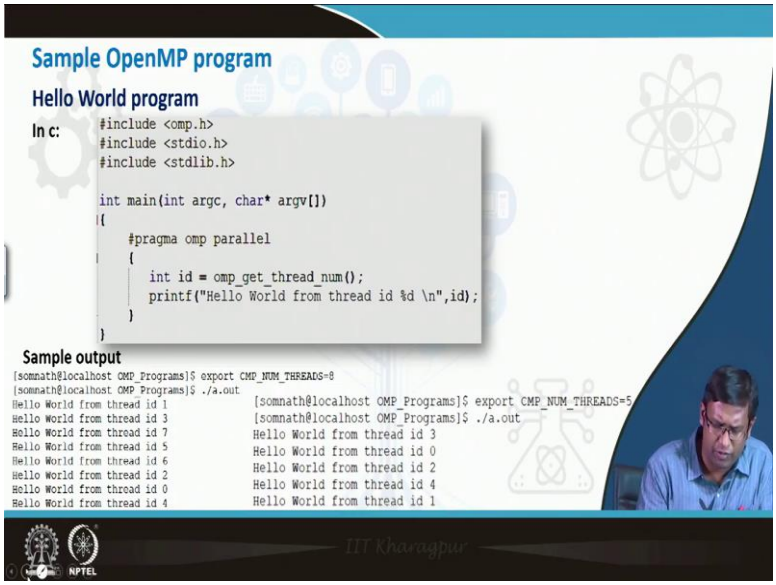
So, while looking into sample OpenMP programs, we have seen that as OpenMP works through different threads, and also all these threads point to a common shared address space. What do these threads do? all these threads basically execute a single instruction with multiple parts of this data.

That means, that the program instance that each of the threads are in executing, this instance is basically a single instance, and we ask multiple threads to execute that instance.

So, a single line is written inside the program which is compiled and given to different threads to execute, and if this line contains some variable which is local to the thread, but as we are using a shared memory system ;these local variables sometimes become out of scope because there is a shared memory. It is the same variable name which is given to all the threads because it is the same programming instance which is being compiled and given to all the threads. So, if you have a variable 'a' which will be used by all the threads differently, it can create a conflict. And we have seen some of this conflict when demonstrating one of the Fortran programs.

So, we will look into shared and private variables in very good detail to see how these variables behave and how to do data handling in OpenMP.

(Refer Slide Time: 02:37)



So, in last class we have looked into two programs; one is a simple hello world program. This is a c program, so we have not separately defined the private variables, but we can see that inside this program this line printf (Hello World from thread). This is the parallel regime within the base so this will be executed over different threads, and each thread will pick up the thread number from the OpenMP library function omp_get_thread_num and it will write the output as Hello World from thread id and the thread number.

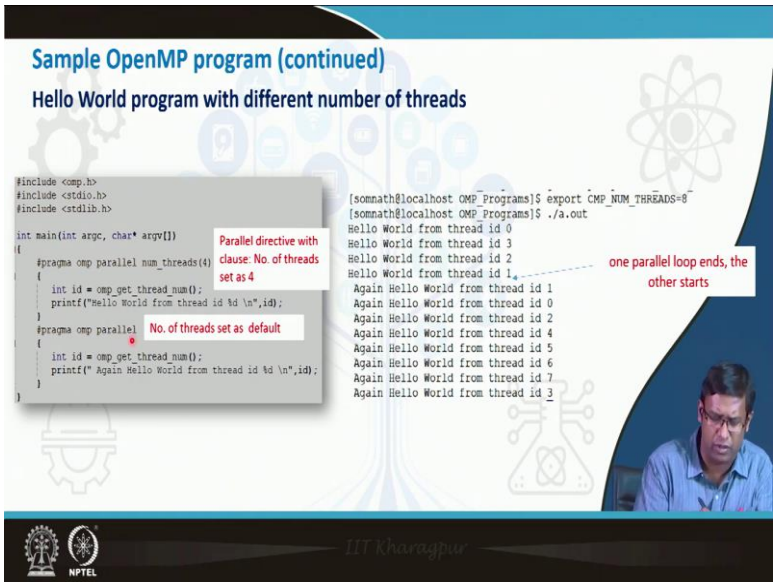
If we launch it in 4 threads in a 4-processor system which will run 1. If we launch it as 8 threads in 4 processor systems, each processor will run 2 threads.

Now, what I was telling about the private variable is that this id is different for different variables; however, this id in general is a single variable which is in the shared memory space. So, these threads are not operating in sequence, they are operating parallel. So, there can be a conflict in the value of id, because many threads are going there and trying to change it.

Good thing in c is that, because id is not used outside the parallel regime, it is treated as a private variable; that means, each thread gets some part of the in some part of the memory which it can keep as private, and these are private to them.

But, if we run a number of threads 8, we execute it, each thread writes the value of id local to it and Hello World from this. So, this line is executed by all threads. It is not that different constructs are to be defined for different threads is the same programming instance which is executed by different threads in parallel.

(Refer Slide Time: 04:43)

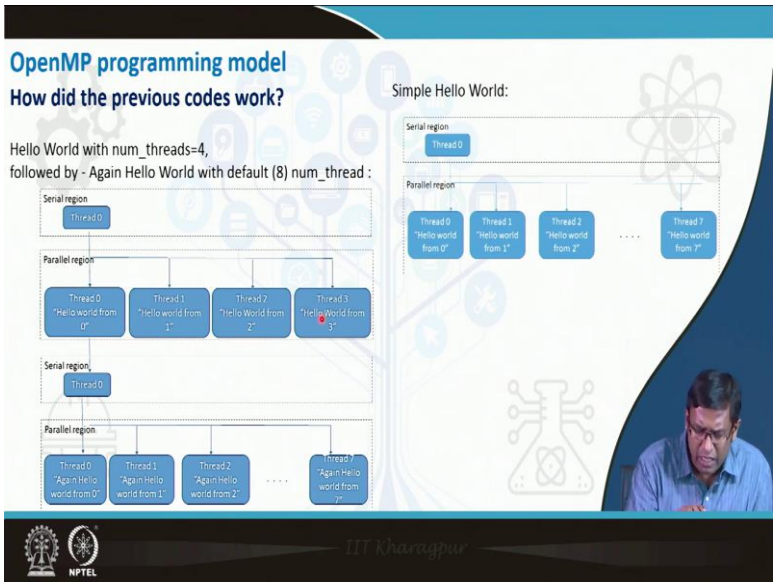


If we see another example of a Hello World program with a different number of threads; that means, we have changed the number of threads, set the number of threads to be 4 here and the number of threads is by default as specified. This is one parallel loop, so there will be one set of threads launched here, then these threads are destroyed. There is another parallel loop; another set of threads are launched here.

So, we can see that, here there will be 4 threads here, the set number of threads and if we run the program, we set the number of threads to be 8. The first region will always run it Fortran the next region will run with 3 threads.

And in between here, we will see that one parallel regime will end here, and after that the other parallel regime will start. So, one brace one parallel regime will be active and then the parallel regime will be ending and another parallel regime will start with the next parallel construct.

(Refer Slide Time: 05:40)



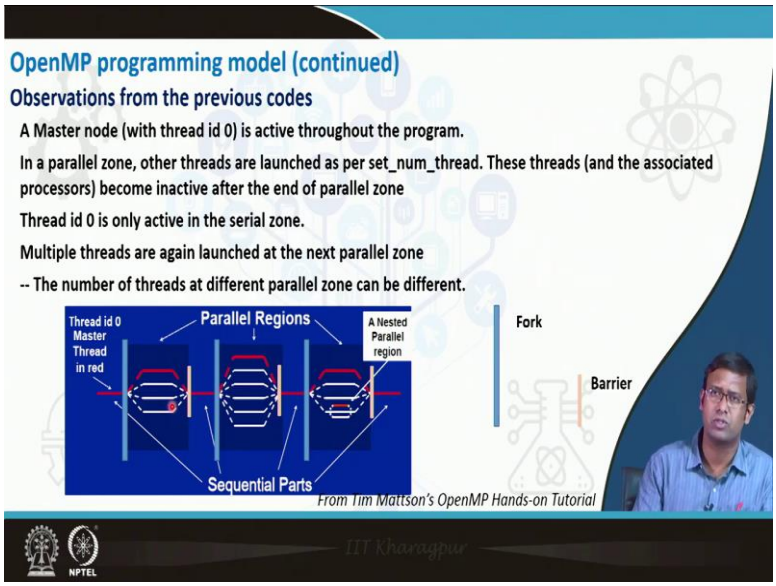
If we see the programming, how does the previous code work? So, in the serial region when everything starts, thread 0 is there. If you do not write the parallel construct, or before writing parallel construct if you write anything that will be executed by this 0 thread, the single thread.

In the parallel regime, when you have to write Hello World from the thread id that is executed by multiple thread regimes. So, thread 0 is active in the parallel regime and it also launches other threads. If it is running over 8 processors, 1 to 7 threads are added over that.

In the second example, where we have hello world from num threads 4 and then again hello world with default thread 8 threads. We will see the first 0 thread 0 is active in the parallel regime there are 3 threads and all the threads are writing Hello World from their thread id.

Then, the parallel regime ends, and it is again another serial regime. If you write anything it will be written only by thread 0, and then again, another parallel regime is launched.

(Refer Slide Time: 06:52)



So, the model of an OpenMP program is like this. That, a master node or thread id 0 is active throughout the program. Thread 0 is active, anything you write outside the parallel regime it is only one thread which is active thread 0, it is executed as a sequential program.

In the parallel zone, other threads are launched as per set num threads. These threads become inactive after the end of the parallel zone. Thread id 0 is only active in the serial zone. Multiple threads are again launched in the next parallel zone. So, after the parallel zone is active, there is another serial zone and thread 0 is again active there. Again, in the next parallel zone multiple threads are launched.

This is taken from Tim Mattson's tutorial, that in there is one thread, the red one is the thread id 0, the master thread, which is active throughout the program at different instances.

A number of threads are launched and these are called a fork, where the threads are launched. And the threads are again killed and destroyed and only the master thread is active through and we put a barrier here, or the joining of the threads and only one thread becomes active.

And again, we launched a thread, and in between the next fork, the previous barrier and next fork there is only master node which is active, master thread which is active or only one processor is active which is launching only one thread, other processors are in dormant state.

So, this is the sequential part of the program. Again, we can launch a set of threads, again we can destroy the threads, the sequential part is there again and through another fork a number of

threads are launched. There is also a nested parallel regime; that means, one of the threads in one parallel regime can be itself another master thread and it can launch a few more parallel threads.

So, inside a thread also can fork into multiple threads and more levels of parallelism can be obtained. We will see some examples of nested parallelism, that one thread is further launching other threads. I mean master thread always launches the other threads, but something which is not master thread within a parallel regime where it is active can launch other threads.

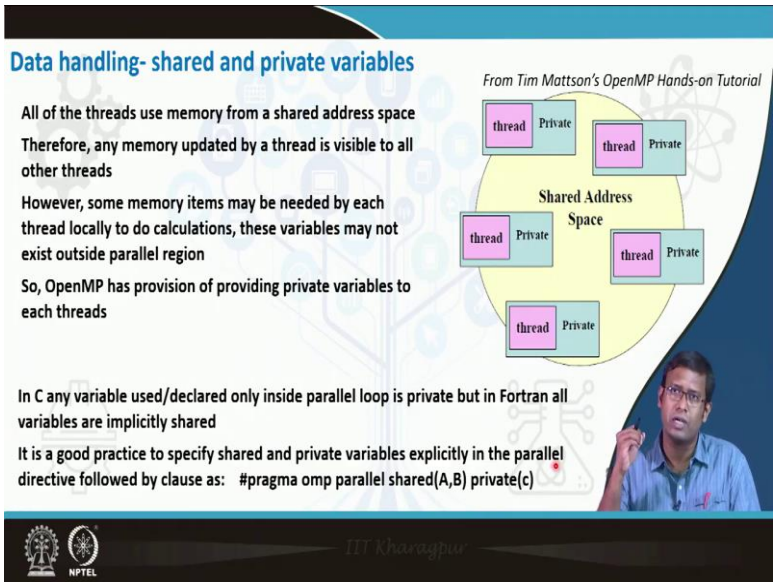
Outside the parallel , any thread outside the parallel regime, any thread apart from master thread is inactive. So, threads are created and launched created and launched, so on.

It is possible that they finish their job at one instance and when the next threads are created, they are completely different from the previous threads. It is also possible that these threads keep some of their own memory, that is when one set of threads are launched, they do something, when the sequential regime goes, these threads are inactive, only the master thread is active.

When the next set of threads are launched, these threads keep some of the previous memory and start operational operation from there. This is also possible, the constructs we will see later.

The number of threads at different parallel zones can be different also, here we can launch more threads, here we can launch less threads. We have seen earlier by set num threads, we can change the number of threads that the or by the clause to the parallel construct at number of threads at different id ok.

(Refer Slide Time: 10:31)



Now comes a very important concept which is shared and private variables which you have seen in the Fortran example, that all the threads use memory from shared address space. Therefore, any memory update by a thread is visible to all the threads. However, some memory items may be needed by each thread locally to do these calculations. These variables may not exist outside the parallel regime.

Say, the thread id itself is a variable, which each thread has its own and this is only active in the parallel regime, outside the parallel regime only master thread is there, its id is 0, but other threads in the parallel regime each thread has a different id. It might be required in some cases.

There can be some other variable which is required by the threads for their own operation in the parallel regime. Inside a thread you want to do a sum up, you are using a variable i for looping. The do loop is through some variable which is specific to that thread and each thread will have different values of this number.

Because it's the same programming instance which is executed by different threads, we are not writing different lines of program for different threads, we are writing a single line of program inside the parallel regime and asking multiple threads to look into that. There can be some variable which is local to the thread, but has the same name as other thread variables.

But in a shared memory system, if the variable name is the same for multiple threads they are pointing to the same location, so one variable changes , if the other looks into the same variable and probably there can be contention in between them.

So, what is the solution? OpenMP has provision of providing private variables to each thread. So, the main address space is like their bank account, is the shared joint account, all can see what is happening there, but everybody is also given with a small money back that they can keep some money within it and do an operation using the private variables.

Because, inside the threads we also do some computations which require some variables say, finding out maximum using the thread id etcetera, and we are not writing different lines in the program for different threads. It is the same programming instance which the threads are executing in parallel.

So, some private variable is required by all the threads, and we see that if this is the shared address space, each thread is connected to the shared address space, but they also have their own private memory where they can keep some variable and change accordingly.

If the variable is changed by this variable or say thread id, id is said by this thread, this is not visible to the other threads. Each thread can do it on their own. And this is again from Tim Mattson's tutorial (Refer Time 13:38).

In C, the variables declared only inside the parallel loop are private. If some variable is not used outside the parallel loop and only used inside parallel loops, it considers this to be a private variable.

But in Fortran, all variables are implicitly shared and that is why we had an issue while translating the same C code to Fortran code; when writing the hello world, the thread id number is the same for almost all the threads in Fortran. Because it was the id which is giving the thread id number was not declared as a private variable in Fortran.

In c it worked, because you do not have to separately declare it in private variable. So, that is why it is a good practice to specify shared and private variables explicitly in parallel directive. Following the clause, say we are running a parallel directive in c it will be # pragma omp parallel; that means, anything after this the next brace will be run as multiple threads; and give

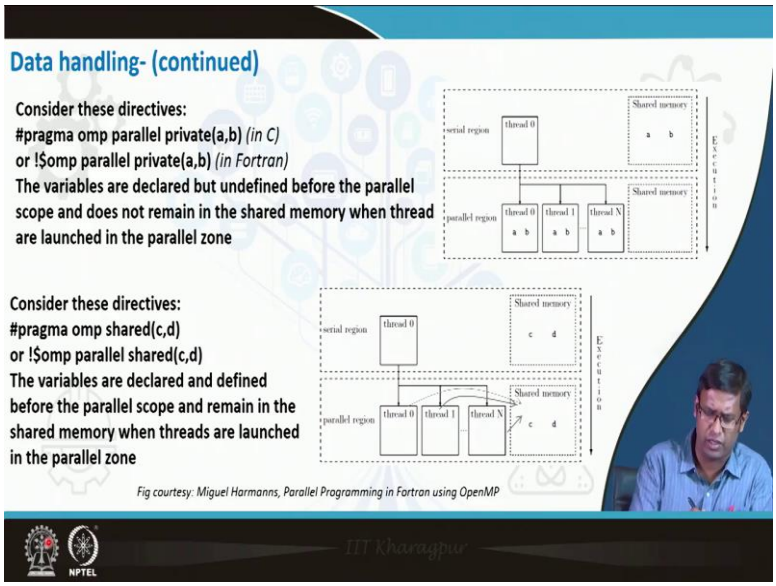
a clause shared(A,B) private(C). So, A and B will be shared variables and C will be a private variable.

So, it is a good practice to explicitly specify shared and private variables, because somebody who is looking into the program can very easily understand what is happening in this code. And as I said, OpenMP is devised to transfer the legacy codes, transfer the codes which are already existing and have been developed over years for scientific computing purposes.

Now, if you look into one of the OpenMP programs, if you are using it, and you have to change something. If private and shared is not explicitly specified and you can see its a 20,000-line long code with many subroutines, you cannot jump into one of the parallel constructs and see and readily say that whether this variable is shared or private.

So, maybe the developer knows it, the next person who is trying to work on it may lose the track. So, it's a very good practice that you specify separately what is shared explicitly; what is shared and what is private as a clause to the parallel construct. That when a parallel loop is there, these variables will be taken as shared variables, this variable will be considered as a private variable.

(Refer Slide Time: 16:03)



We will see these shared and private steps in more detail. Consider these directives, #pragma omp parallel private (a, b) this is a C parallel directive . We start with #, then pragma, then

omp and then what is the directive? This directive says that the next regime will be parallel, parallel with private variables a and b.

Or in Fortran it is exclamation dollar; exclamation is usually a comment in a Fortran program, but if exclamation dollar omp is given ;it's an OpenMP construct or OpenMP directive. Parallel and private a b. The variables are declared, but undefined before the parallel scope. If we do not use these variables before the parallel scope, they are undefined there. Within the parallel score they do not remain in the shared memory. When we declare the variables, they are in the shared memory, but undefined there. When we come into the parallel scope they do not remain in the shared memory, but they are separately copied to the private memory of each of the threads when the parallel zone comes.

You can see it through an example, that if we write this beginning when thread 0 is only a and b while we are residing in the shared memory, though they are not operated, they are not defined there. Inside the parallel regime, each thread gets a b, a b, in the shared memory a, b does not exist there. If you try to write anything a b and check whether it's in the shared memory, there is no a b in the shared memory, a and b these are in the private memory only.

Once this parallel regime will be over, they will again be in shared memory, but these values will be erased and they will be in the shared memory with the previous value, previous shared memory value. Consider this directive # pragma omp shared(c ,d), omp shared c d these variables are declared and defined before the parallel scope, and remain in the shared memory even when the threads are launched.

So, you see, c d are in shared memory, because they are given with clauses shared. When thread 0 is active c d is there in the shared memory. When the other threads are launched in the parallel regime, all the threads are pointing to shared memory and if they are doing with c d, they are doing with the shared memory variable c d. These are not in the private variable.

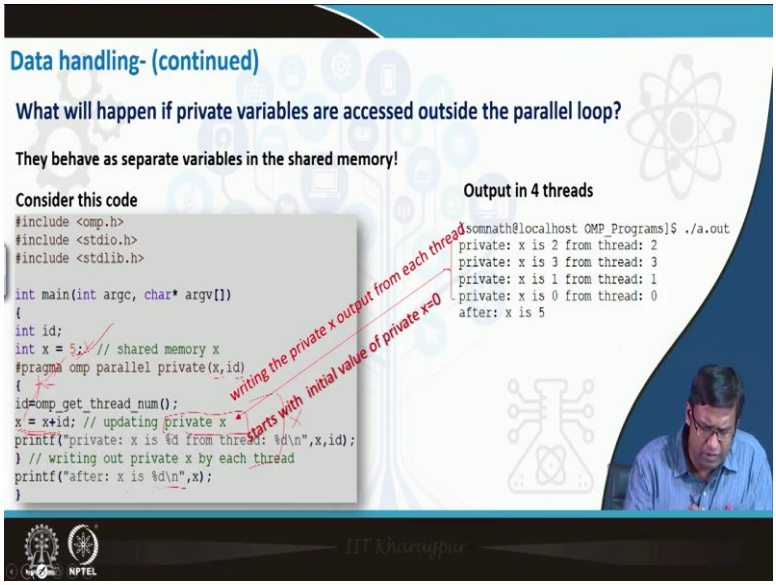
So, if thread 0 writes c =c + 1, initially c = 0,c value has been 1, which is visible to thread 1 up to thread N.

Similarly, if thread N does d = 5. So, all thread 0 to thread 1 everybody says d = 5, but in the other case if thread 0 writes a =5, thread 1 to thread N does not have any idea what is the value of a for updated by thread 0, because this is the private variable.

So, these are the differences and there are private variables which come to the private and local memory of thread, though this is a shared memory system, some variables still can be considered as private variables; that means, multiple copies of that variable are created and allocated to different threads.

If this is shared memory then it does not go to the local allocated memory of the threads. It remains in the shared memory and each thread can directly access the memory and change it. And this is taken from Miguel Harman's parallel programming in Fortran using OpenMP. These figures are taken from there.

(Refer Slide Time: 19:42)



What will happen if private variables are accessed outside the parallel loop? I said that the private variables are defined as inside the parallel loop, so any change done by any thread within the parallel loop is valid for the private variable. Outside the parallel loop, well anything basically lies in the shared memory. So, any variable outside the parallel loop is a shared variable, but whatever has been done in that variable in the parallel loop does not remain in any memory. So, it erases.

In the outside share parallel loop, it is like a shared memory variable; whatever happened to it before the parallel loop will be still valid to it. An example is that, so it will behave like a separate variable in the shared memory.

Consider the code, that you have an integer x which is set to be 5, inside the parallel loop you defined x and id, id is the thread id to be private variables, inside the parallel loop you get id, so for each thread will get its thread id and add with x. So, for thread 0 it is x +0 that is 5, for trade one it is x = x + 1, 5 + 1 6 and write the values.

So, thread 0 and thread 1 will of course, have different values. After the parallel loop is over , this x which is updated as the private variable is not valid. It is erased. So, what will it write? It will write the old x. We will see the output, we have set the number of threads to be 4, when we run it in 4 threads, and it starts with x is equal to 5.

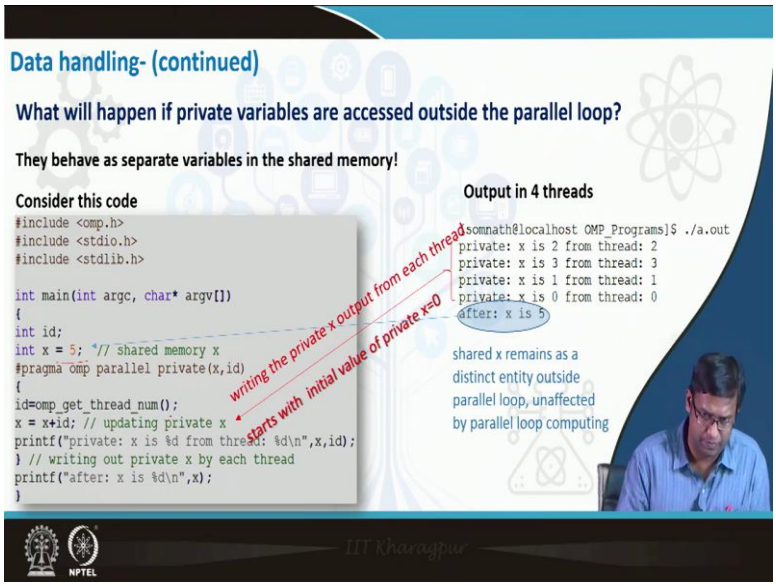
Inside the parallel loop this shared variable is invalid. So, inside the parallel loop it is 0, x goes with uninitialized value, which is by default 0 in c or Fortran. So, x goes with 0 and it is x is equal to for thread 2, 0 + 2= 2 ;for thread 3 x = 0 + 3= 3 for thread 0 x =0 + 0= 0.

So, inside the parallel loop it's the private variable which has no relation. It is not connected with this x ;this was a shared memory outside the parallel loop x was shared inside. As we have declared is as private; it's an entirely different variable.

Again, when we come out of the parallel loop all these things become invalid and it writes x to be what was the value of x before the parallel loop. For what x resides in the shared memory it writes that.

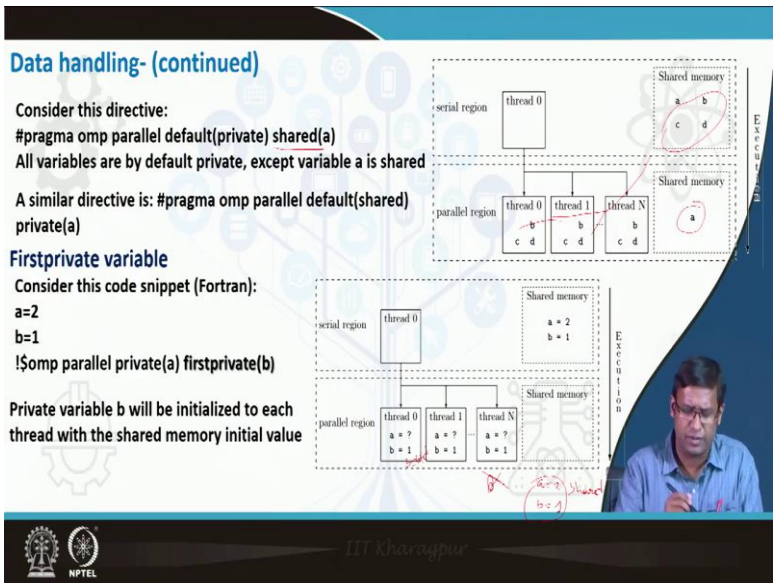
So, if we see the example, that inside the parallel loop x = x + id and writing the value that will be done with initial value of private x = 0. The value 5 will not be accepted inside the parallel loop. Because shared memory will be erased for x for the threads and only private variables will be taken, which will start with initial value x is equal to 0 and add the thread id with it.

(Refer Slide Time: 23:12)



After the parallel loop, x = 5, that is the shared x will remain as a distinct entity outside the parallel loop, unaffected by parallel loop computing. So, a private variable called outside the parallel loop is a shared variable which is unaffected by the parallel loop computing.

(Refer Slide Time: 23:30)



Consider this directive, #pragma omp private parallel default (private) shared(a). This means, by default all the variables are private inside the parallel loop.

Whatever variable we introduce inside the parallel loop is a private variable. Except the variable a is shared. So, all the variables by default are private except a. So, we have a ,b c, d, say we have initialized a, b, c, d before that. When the parallel regime comes, a only resides in the shared memory, all other variables are by default private variables, they go to the private memory of all the threads.

So, when we are dealing with a large number of variables and we know that only a few variables will be private variables or shared variables or vice versa, we can use the default command. If we write parallel default shared private a b c d or any other variable will by default reside in the shared memory; only a will come in the private memory

So, the quick question is, if in Fortran we write dollar exclamation dollar omp parallel and without any construct any clause.

It has a default share, so all the variables are by default in the shared memory in Fortran. So, default shared we do not need to write in Fortran, but it is intrinsically specified in a parallel loop. If we have to write anything in private, we have to specify that in private or in c default, but in default private can be done in Fortran, but in c default shared is not there. So, we have to specify it.

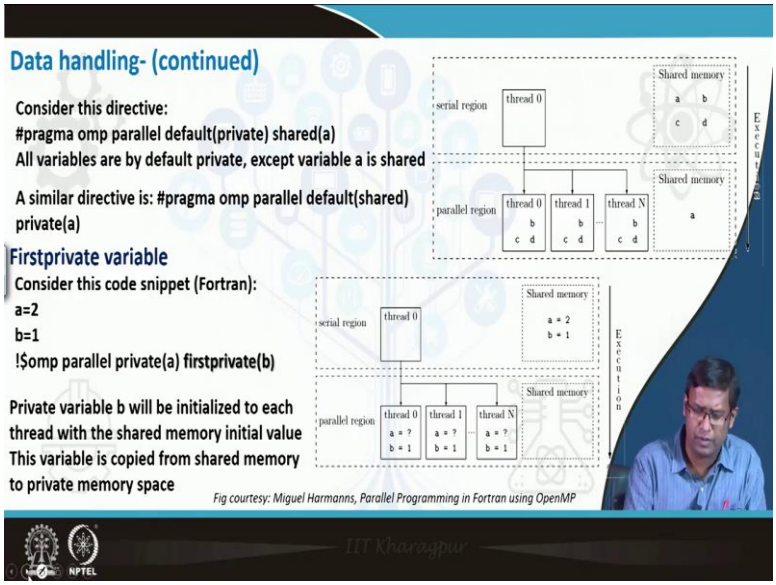
A similar directive is default shared and private (a) the converse. Now, it might be some time important to retain some of the values of the shared memory variable in the private memory in the threads, and this is done to first private, there is another thread private, last private etcetera. Some values which are operated outside the parallel regime might also be required by the parallel regime. So, some of these arrangements might be important.

So, consider the code snippet in Fortran a =2 ,b =1 and we write $ omp parallel private (a) first private (b). What happens here? b was earlier a shared variable, now when we write first private b ,this value of b comes inside the parallel regime and b is initialized with the value which it got last in the sequential part.

So, a = 2 in b = 1 one all the threads when we write private a, all the threads go with an uninitialized there by default a is 0, but for b all the threads get b = 1. So, first private gives the introduces a variable as a private variable which retains its last shared memory updated value.

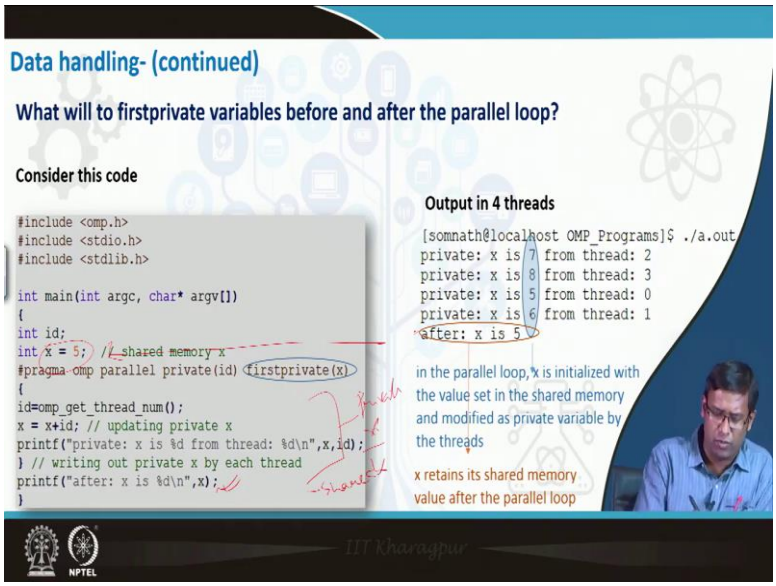
Private variables will be initialized to each thread with the shared memory value, but whatever changes you do with b will not be valid after the parallel regime. b will continue after the parallel regime, you will still get if you write a = 2 b = 1, these are the shared memory values which will be active after the parallel regime. The changes, if you add something say you do b = b +id, that will not be valid after the parallel regime.

(Refer Slide Time: 27:50)



The variable is copied from shared memory. So, here the variable is actually copied from shared memory to the private memory space.

(Refer Slide Time: 28:00)



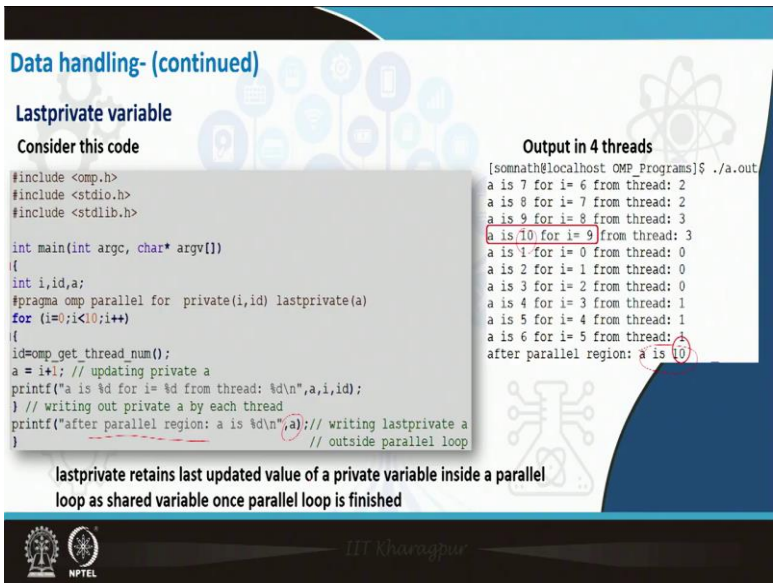
What will happen to first private variables before and after the parallel loop that was the question. Consider the code that x = 5 this is the shared memory value then id is private which will be a thread id and x is a first private variable x = x + id and what happens to x after the parallel regime, we will see.

So, when we write this in our 4 threads x = 5 was the last value in the x shared memory. So, as x goes with the first private the clause x comes with the value 5. So, 5 id is added over 5 for all the threads. So, thread 0 gives 5 + 0 =5 thread 1 gives 5 + 1= 6 thread 3 gives 5 + 3 =8 so on.

After this, the private values which are invalidated, after the parallel loop is over. So, after the loop x writes the shared memory variable 5. In the parallel loop x is initialized with the value 5 and then modified as a private variable, after the parallel loop x retains its shared memory value after the parallel loop.

Whatever happened to this these are done to private x and this is shared x which is unchanged by the updates in private x. Only first private means only the initial value is carried inside the loop.

(Refer Slide Time: 29:39)



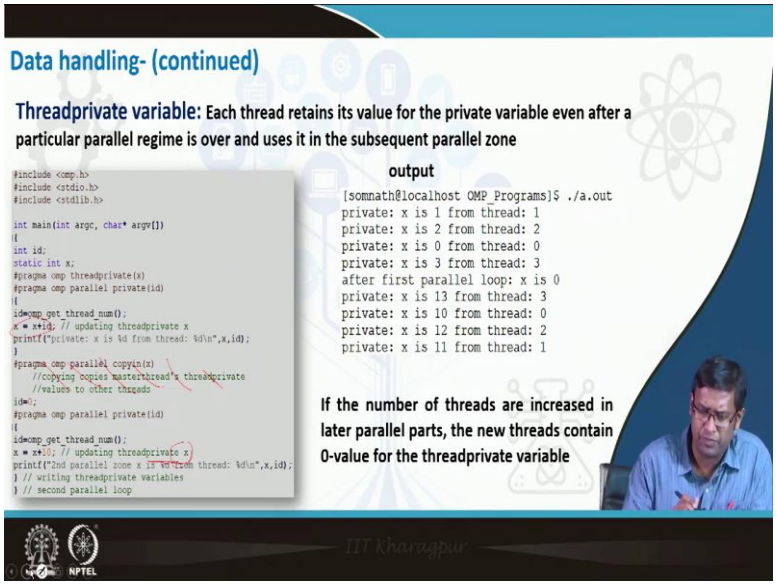
The other example can be a last private variable, now whatever is done within the private within the parallel loop can go out of the loop as a shared variable. So, consider the code and its output. Integer i ,id, a ;i and id are private and a is last private. So, what will happen a = i + 1 for i = 0 to 10.

We have parallelized this loop for 10 threads for a loop of 10 if parallel for means a loop going from i = 0 to 9, this will be given to different threads each if you are running in 10 threads each thread will take care of one of the iterations in the loop.

So, here will be updated. So, the last update will be a =i + 1 for last i is 9, 9 + 1 =10 the last updated value of a in the parallel loop will come out of the loop and that will be retained as the shared variable. So, we see there are multiple threads we have. We have run this in 4 processors there are total, but 10 iterations in the loop. So, each thread will get 2 to 3 that there are 4 threads launched here for 10 iterations first 2 threads get 3 of the iterations and the next 2 threads will get two of the iterations. We look into parallel for later.

And the last updated value is for i = 9 a = i + 1 that is 10 .After the parallel regime when we are writing ,a retains the value 10 . Last private variable retains the last updated value of the private variable inside a parallel loop, as a shared variable, once the parallel loop is finished.

(Refer Slide Time: 31:46)



Thread private variable; now there can be cases where parallel loops are launched one by one. One parallel loop is launched then comes a sequential loop then another parallel loop is launched etcetera. Thread private is a very helpful clause which helps some of the variables to retain their value even when the next instance of threads are launched.

Each thread retains its value for the private variable even after a particular regime ends and uses it in the subsequent parallel regime. Like, we have in we had static it has to be defined a static integer static integer x = x + id, x is a thread private variable.

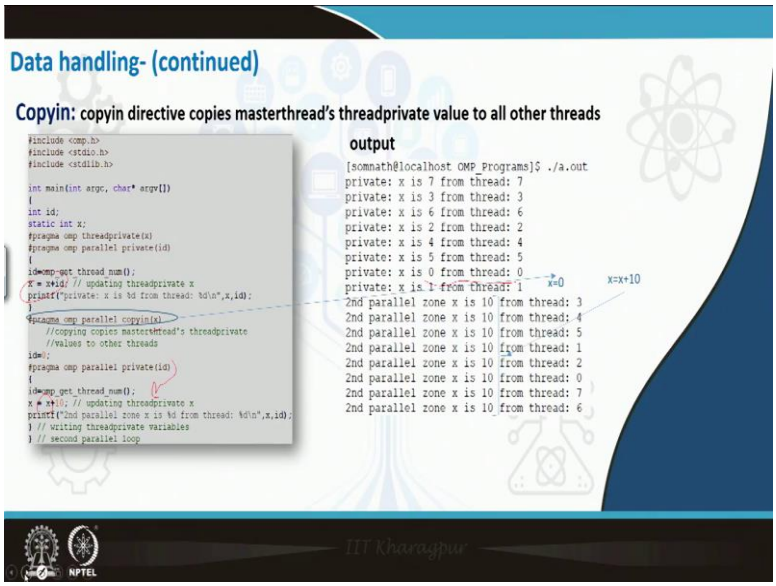
So, each thread updates x on its own. So, it starts with 0th threads writes x = 0 + id is 0 , first thread writes x = 0 + id = 1,so on and then the parallel regime ends . When we will run the threads, we will get the final solutions.

So, for if we are running in 4 threads each first thread will give x = 0. So, second thread x = 1, x = 2 and x = 3, when you go to the next regime these values of x which was 1 from thread 1 ,2 for thread 2, 0 for thread 0, 3 for thread 3 has been retained, and in the next parallel loop it the private variable as we have considered x to be thread private, that private variable is updated and the entire thread private has to be within one pragma. This private variable is updated in two different private regimes. They still keep the up thread private values ;this is initialized with the last initialized value, last value of x on the threads.

So, whatever is the value of the private variable for one particular thread that retains even after the threads are destroyed and the next instance of the threads are launched, but in case if we launch a greater number of threads. In the next instance the new threads will have 0 value in the private thread private variable as a thread private memory.

If the number of threads are increased in later parallel parts, the new threads contain 0 value for the threat private variable.

(Refer Slide Time: 34:47)



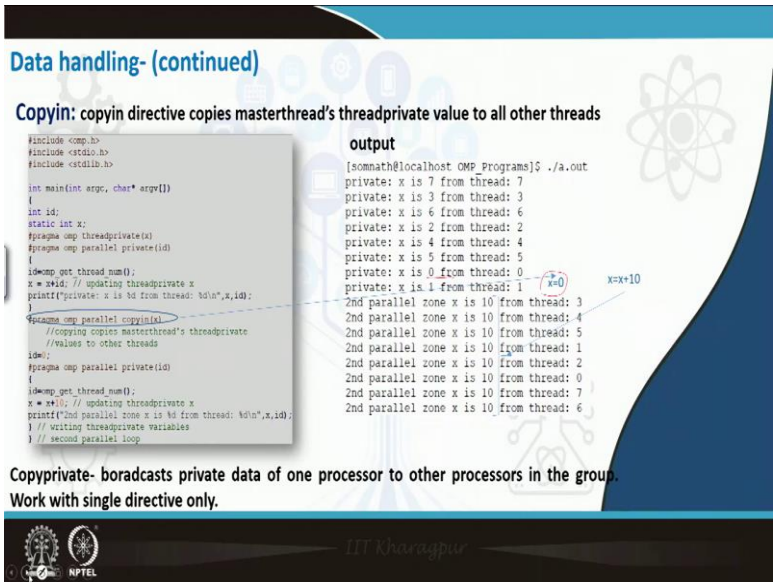
Now, what copyin does is, sometimes it might be happening that each one has different value, but one value has to be given to others everybody should have uniform value after the parallel regime ends. What copyin does? copyin copies the master thread straight private value to all the other threads.

And now if we run the previous program with 8 processors ,with the copyin value.; so, we can see the last master thread has x = 0 ,this x = 0 value, the master threads x will come into the private memory , when we write copyin of all the threads; and this will be initialized in the next parallel loop.

So, the next parallel loop, if all the threads are done with the value 0 and added 10 over that, so we can see that everybody is giving uniform value, because uniform value of x has gone to all the threads. So, though they are thread private, everybody copies the master thread straight private value to all their other threads.

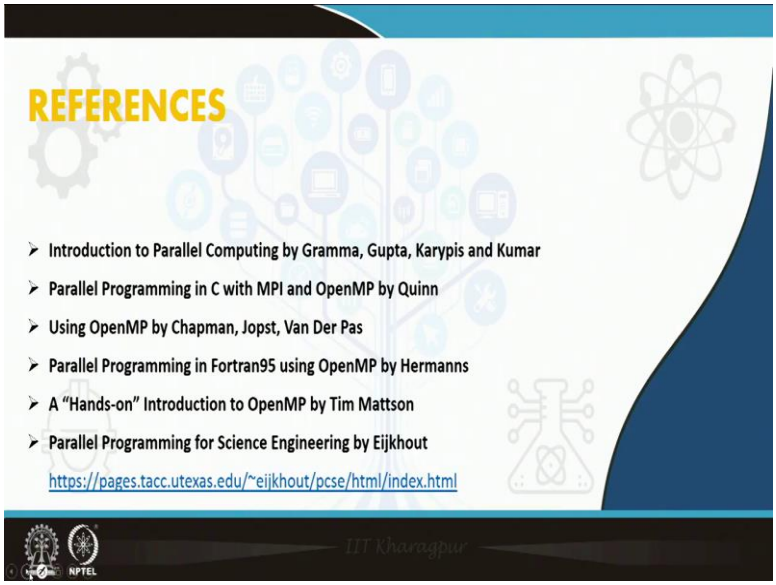
So, this is pragma copyin which x = 0 from the master threads copyin set x = 0 for all subsequent threads and x = x + 10 ,0 + 10 happens for all the other threads.

(Refer Slide Time: 36:12)



Copyprivate broadcast private data of one processor to other processors in the group. This works with a single directive. So, if we give a single directive; that means, though it's a parallel regime only 1 thread will change some value and then we put copyprivate. So, copyin means master thread state private data is given to all the threads, copyprivate means any other threads private thread private data can be given to the other threads in the group, instead of master thread any other thread can do similar work. These are also very useful commands in the program.

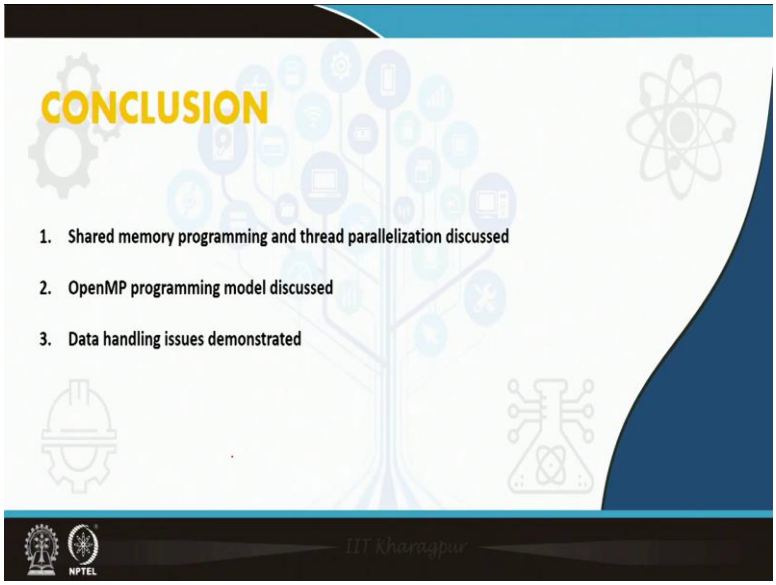
(Refer Slide Time: 36:50)



So, these are the references introduction to parallel computing by Gramma Gupta, Karypis Kumar. Parallel program with C and in OpenMP and MPI by Quinn and using OpenMP is a seminal book on OpenMP by Chapman ,Jopst, Van Der Pas.

Parallel programming in Fortran95 using OpenMP some link to these books can be obtained through OpenMP.org site ;hands on tutorial by Tim Mattson. University of Texas hosts a parallel programming for science engineering, it has many examples and exercises on OpenMP programming you can look into it.

(Refer Slide Time: 37:34)



And we can see that, through these discussions, we have introduced shared memory programming and thread parallelization. We will see the OpenMP programming model in much detail writing OpenMP programming much later, and some of the data handling issues for shared and private variables .

In the next lecture, we will start with the conflict soil data handling and specially the cache coherency and false sharing issues in OpenMP programming.