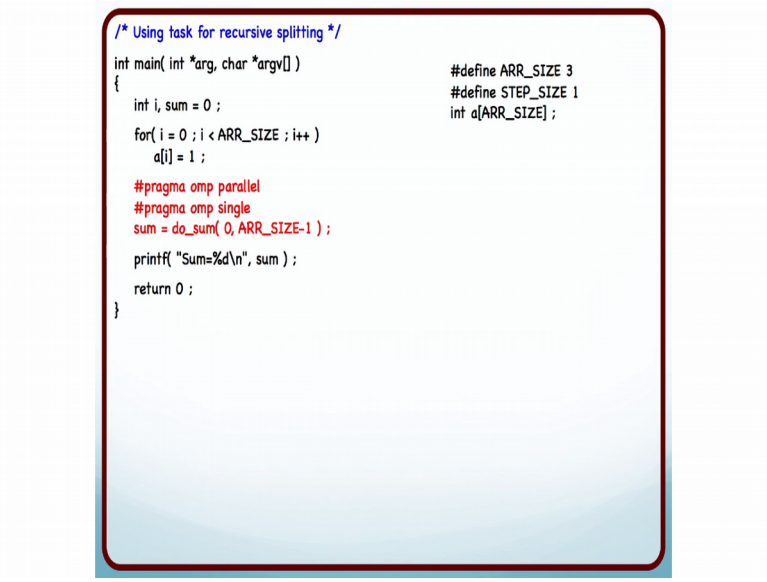
Introduction to Parallel Programming in OpenMp

Dr. Yogish Sabharwal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

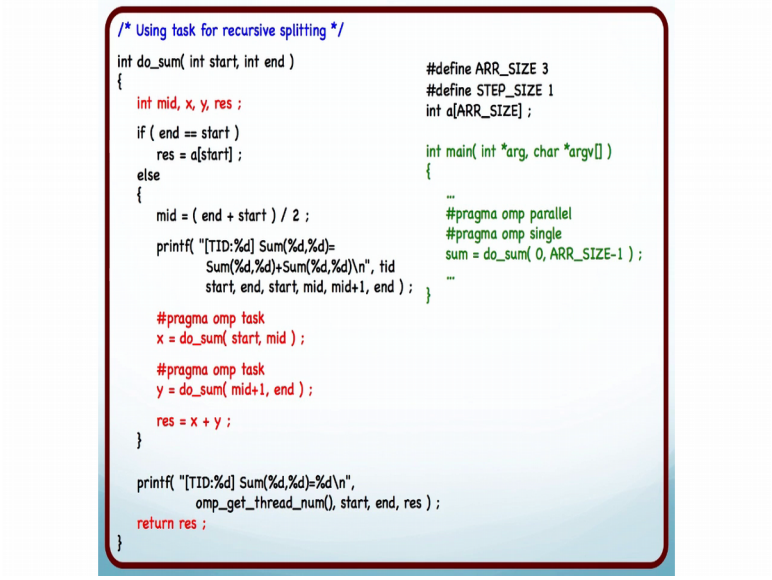
Lecture - 31
Recursive task spawning and pitfalls

(Refer Slide Time: 00:01)



So you can also use tasks for recursive splitting. So, it is a very very simple example; what you do here is you want to compute the sum of an array first you initialize the array with all ones. So, you say hash pragma omp parallel. So, multiple threads are going to get launched over here and then one thread goes and calls sum equal to do sum do sum is a function which actually computes the sum of the elements.

(Refer Slide Time: 00:32)



And this is the implementation of do sum what does do some look like. So, what it is going to do is it is going to do recursive splitting is going to divide the array into half and create 2 tasks. One to compute the sum of the right hand side one to compute the sum of the left hand side. So, these are some boundary condition handling. Now here what it does is, it computes the midpoint and it prints who is doing the summation and finally, it calls hash pragma omp task here and assigns it. The work from start to mid and then it creates another task over here and it assigns; it work from mid plus 1 to end; seem simple and finally, it will say result is equal to x plus y and that is what is returned over here.

So, there are several issues with this code; what is the issue?

Student: There should be a barrier before (Refer Time: 01:33).

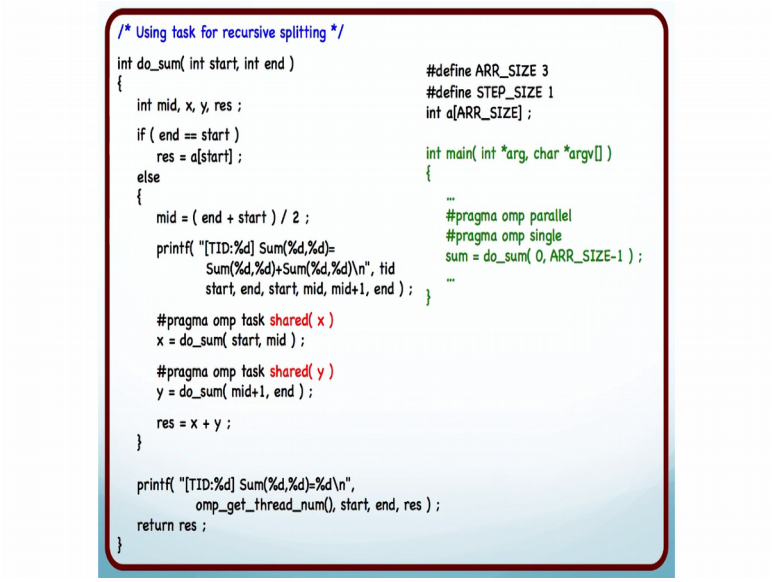
There should be a barrier before res is equal to x plus y; anything else, what is the value of x at this point?

Student: (Refer Time: 01:53).

Returned by the function; so, the problem is that by default variables are treated as first private which means a separate copy is created, right. So, at this point of time for x a separate copy has been created and the moment this task gets completed that copy is gone that data is gone it is not referring back to the same variable, right. So, that is the

first bug. So, if I actually run this code; what do I see? I see some garbage right as expected.

(Refer Slide Time: 02:36)



So, this is what I need to do. So, I need to say here that x and y are both shared in these tasks. Now what will happen is once this task is completed when it sets the value of x; it is actually setting the value of the delegator where it was called from, right.

And similarly this y is setting the value of the delegator where it was called from. Now, what is the problem? So, now, when you encounter res is equal to x plus y, you are actually adding up these 2 values, but you have set them correctly from the tasks. So, the shared issue is resolved there is one more issue.

Student: It has to wait for the tasks.

It has to wait for those tasks to complete.

Student: (Refer Time: 03:17).

The problem is that this thread is going to create this task here create this task here and then carry on doing its work it is not going to wait for them to complete right those tasks are spawned off created I do not know when they will execute. So, how do I know the value of x and y at this point of time I have no idea what these values are right they are not filled from these function calls. So, I need to wait for these tasks to complete. So,

how do I wait for these tasks is to complete; I cannot use barrier because what is the barrier do a barrier waits for all the tasks across all the threads to complete.

Student: Yeah.

That is not what I want here.

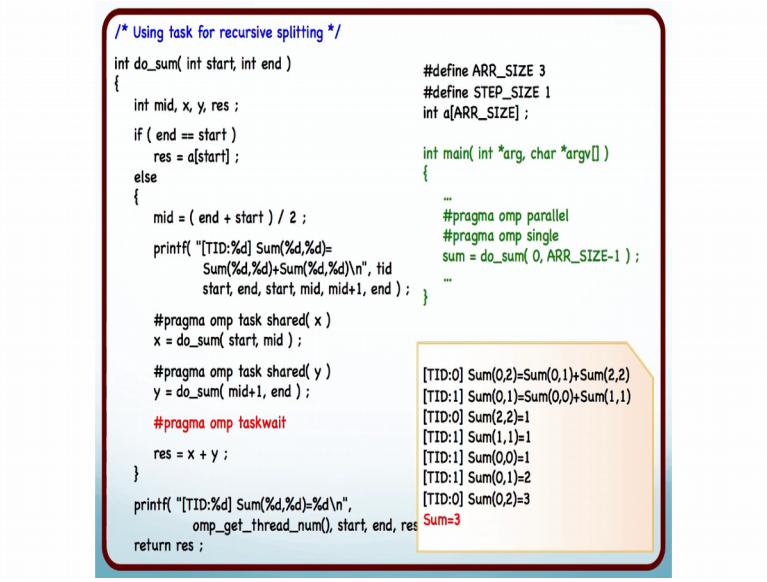
Student: I want for only tasks spawned from this.

Correct I want to wait for all the tasks that are spawned not by this thread also; which are spawned by this task.

Student: (Refer Time: 04:12).

So, when a task executes it may spawn more tasks, I want those tasks is to complete and then I want to proceed further. Now what this is showing is that when I run this code, right, I still get the wrong result.

(Refer Slide Time: 04:31)



And I have this is a particular directive hash pragma omp task wait which says that wait for all the tasks that have been spawned by this task so far to complete.

Student: (Refer Time: 04:43) tasks are being spawned by a thread num (Refer Time: 04:48).

So, actually, in openMP; when you encounter a parallel region what you spawn is not for threads, but for tasks.

Student: (Refer Time: 04:57).

So, we will not get into that. At the top level think of it as that either the set of tasks that have been spawned by this task or if this is not a task, but a thread at the top level, then all the tasks spawned by this thread, right it waits for all of them to complete.

Student: What is barrier (Refer Time: 05:18).

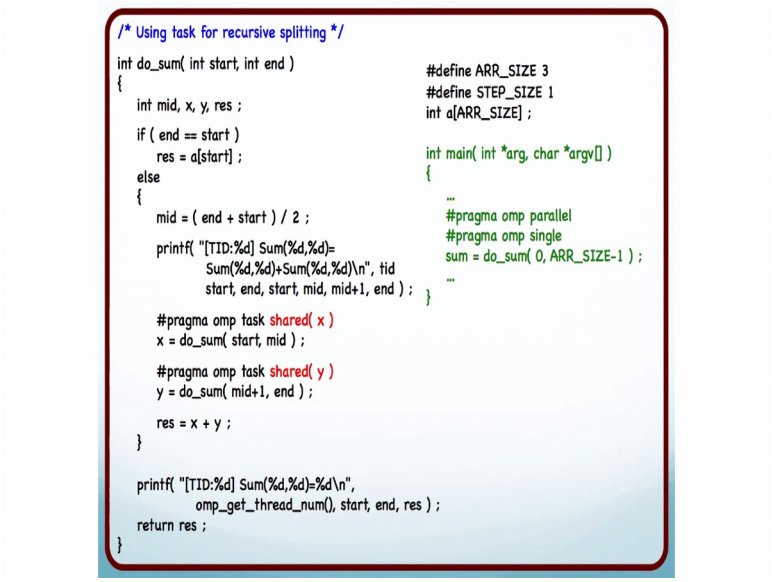
So, a barrier is going to be quite problematic over here, it is dangerous to use barrier inside tasks very very dangerous suppose that you spawn 2 tasks, there are 4 threads, but only 2 tasks are spawned those 2 tasks call a barrier they get stuck in the barrier the other 2 threads do not even call a task they go straight there is no single barrier that all of them are encountering they never even encounter a barrier. So, you will get stuck there are other cases also even if all the threads were spawning tasks how do you know how many tasks they were spawning or how many tasks they were executing all the tasks may be executed by 1 thread or 10 by 1; 5 by another, but if you are calling a barrier inside that how can you be sure that all the threads are going to call barrier equal number of times you cannot be sure about that and if you call the threads are not calling barrier equal number of times then you are bound to get stuck your code will hang.

So, a bottom line use barrier carefully there is a simple use case for barrier that in the main code wait for all the tasks to finish that is the way you should use it right not to wait for some tasks to complete that is very dangerous do not use it inside tasks; barrier inside tasks is very dangerous. So, the right call is hash pragma omp tasks wait that waits for these to complete. Now it is possible that after this I spawn more tasks and I again call hash pragma omp task wait right. So, within a task, I could call it multiple times that is.

But at this point of time this thread is going to wait and it will not execute the remaining statements until these tasks are not done that does not mean that this professor is going to be idle, it is going to put this aside and start looking for if there are any tasks there it can pick up if there is any task it can pick up a team to start executing that yeah. So, now, if I

execute the code I see the correct result that is what I expected you have to be very careful with this. So, let us go back to this slide.

(Refer Slide Time: 07:20)



What is the danger here? I have said that x and y are shared which means that these tasks are pointing to these locations. These; where are these locations? These are on the stack; this is a function when are these going to get removed from the stack when this function ends.

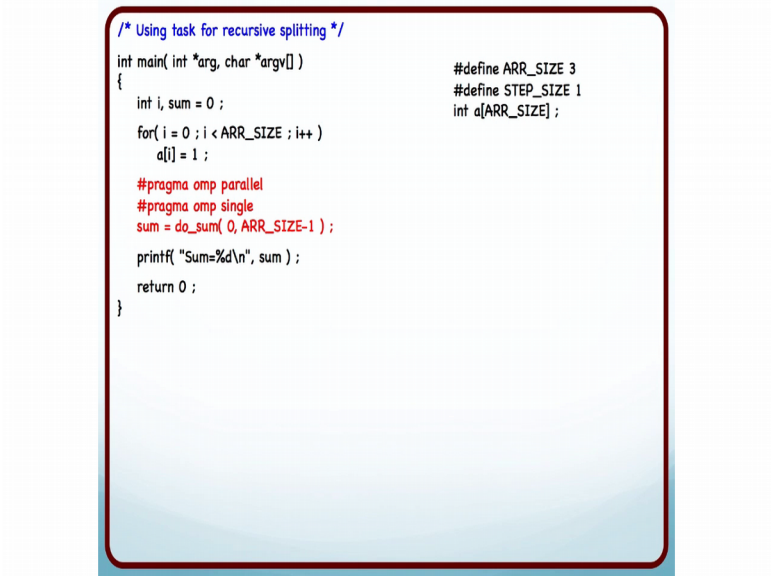
Right and in this case because I do not have a task wait the moment you reach here and this function ends this data is gone. This memory location is no longer valid, it may be used by some other function call or something else some other data could come on the stack over there, right. So, that memory location is not does not belong to this x and y anymore. So, you have to be very careful when you are using shared inside tasks that make sure that data its scope is valid until the task is complete.

Student: So, in this case, there is no implicit barrier at all.

The implicit barrier is associated with certain omp directives. Here this is a function call.

Student: So, you have a function on this calling task and it is not even inside a parallel.

(Refer Slide Time: 08:34)



No, no, remember, this is where we started.

Student: Yeah, but how does a function know (Refer Time: 08:38).

Function does not need to know; you have to know as a programmer

Student: A task needs to be inside a parallel.

Yeah.

Student: You cannot spawn a task without a parallel.

Yeah.

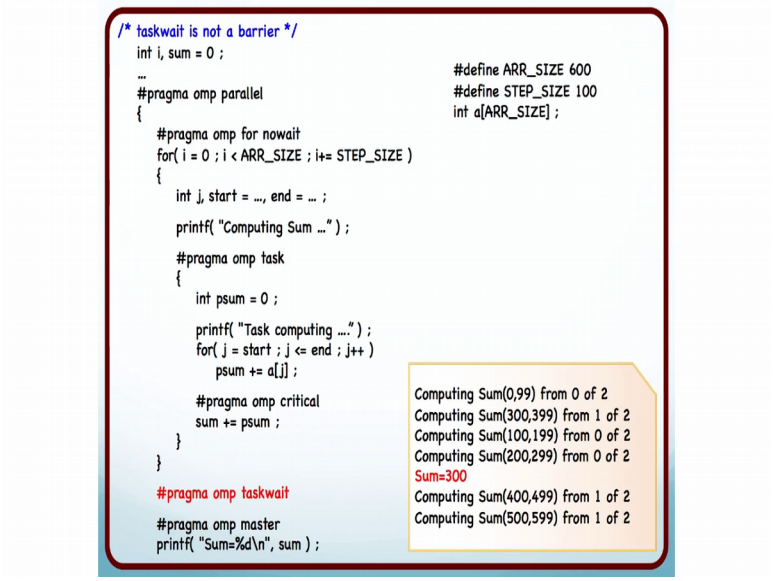
Student: So, there is the function must be telling the task this is this is your parallel.

That is something done at runtime I mean the compiler is going to replace that code with something that may be accessing variables that were initialized at the time the parallel region started and so on.

So, you will run into trouble if you call that function from now outside the parallel region then you will run into trouble, but that will be a runtime error not a compile time bug a compile time it I mean; how does the compiler know compiler will just assume that this must be inside a parallel region, but it does not need to know which parallel region; you

can invoke the same function from different parallel regions what is the harm; why cannot I do the summation twice or trice from different parallel regions that is why; it should not be associated with the parallel region.

(Refer Slide Time: 09:39)



So, remember that task wait is not a barrier, right, I hope that is clear there is a difference between barrier and task wait we have seen that barrier does not function as a task wait, but task wait also does not function as a barrier, right.

And here is a simple example to see that. So, here we have this no wait with hash pragma omp for and I have said hash pragma omp tasks wait at the end of this for loop right and then I am printing sum from hash pragma omp masters. So, only one thread is printing sum. So, what is going to happen here? So, in this case; what happens is in this particular run if we see the output. So, the master thread; thread 0, it came out, it encountered hash pragma omp task; wait, but whatever it had spawned has already been computed in the tasks that it had spawned they are completed. So, it is going to proceed ahead and it is going to print sum and you see that at that point of time the value of sum is 300; it is not 600, right. So, just be careful about the difference between task wait and barrier, right.