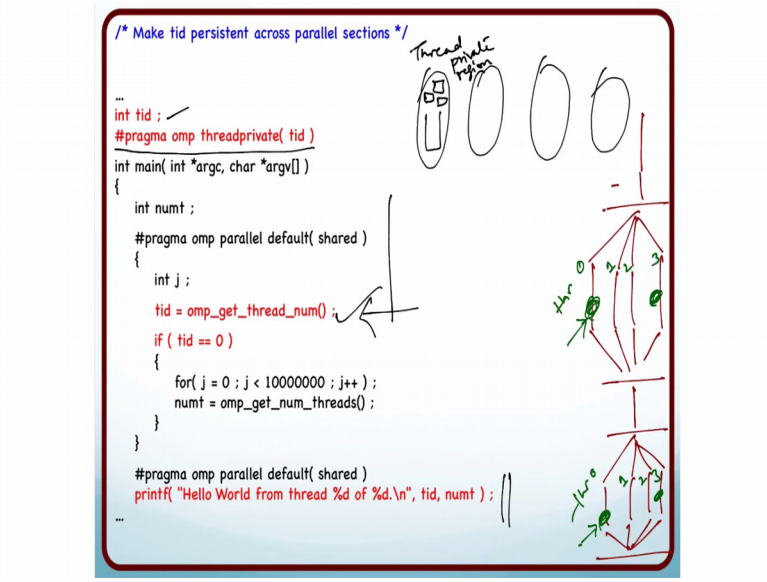
Introduction to Parallel Programming in OpenMP
Dr. Yogish Sabharwal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 20
OpenMP: thread private variables and more constructs

(Refer Slide Time: 00:02)



So here is another way to save those multiple calls to omp get thread num. So, this is something called thread private variables. So, what you can do is; you can make variables persistent across parallel sections. So, here is how you do it, so this is a variable tid and you declare over here that this is a thread private variable. And this is a way to do it you say hash pragma omp thread private and then the variable name. What it is telling the compiler is that; I want this variable to be retained across parallel sections.

So, it is going to allocate this not on the stack because I am in the stack may come and go whenever the threads are launched whenever folks and joins happen. It is going to allocate space for this variable in some thread private region; some region which is there for every thread; maybe the stack is also stored over here. So, maybe it might decide to store the stack and some variables; whatever other thread private variables in the space. So, every thread will have some space, some thread local space and it is going to allocate space for this variable over there.

The important point is that this variable does not lose its value across parallel regions. Now, what happens in this code is that at this point of time I called tid equal to omp get thread num, but now in the second parallel region; I do not need to make a call to tid equal to omp get thread num again. Because tid is declared to be thread private and I have already initialized it over here.

So, this value is going to stack that thread for its lifetime; what does that mean is this master thread that is executing when it encounter this parallel region, it launches 4 threads; when this region ends, it comes together again a single thread executes again 3 threads are launched there are total 4 threads and again they join back right; this is the life cycle of the threads.

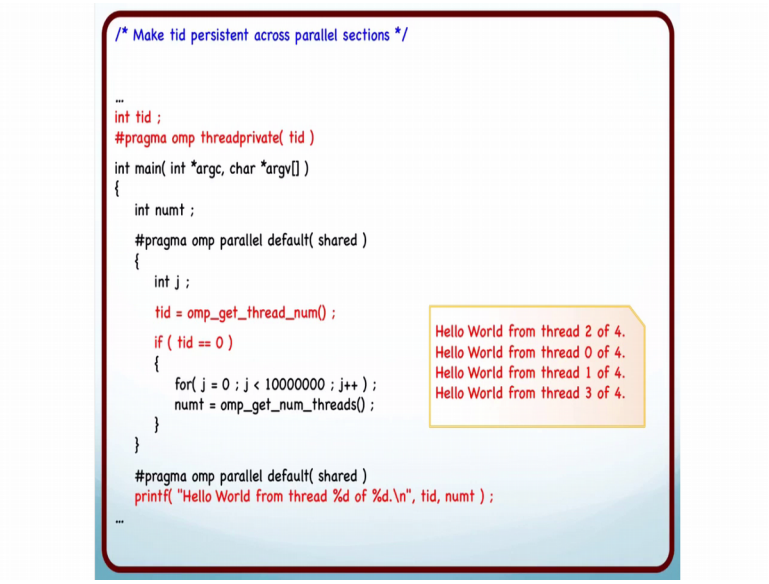
But here is the important part; if this is thread 0; this is thread 1, 2 and thread 3 and this is thread 0, 1, 2, and 3; the value of the variable of the thread private variable over here is the same as the value of the thread private variable over here. Thread 0 will see the same thread private variable; so it is by the thread id; the same thread having the same id; we will see the same value. So, that saves me the trouble of making another call to omp get thread num, I will show it in thread private variable and thread private variables exist to causes multiple parallel regions.

So, this is important because when I set up threads; a lot of time I will start maintaining some state in every thread that what is the work at suppose to do and so on. And then you know I am done with this particular parallel region and later on I want to again start a parallel region, but I know; what is the work I was supposed to do, what was the work I was supposed to do? I am not going to start you know figuring that out again that stored in some thread private variable, I will just take it up and continue my work. Any questions so far?

Student: What happens if we want to add the values of 2 threads into a thread private variables k?

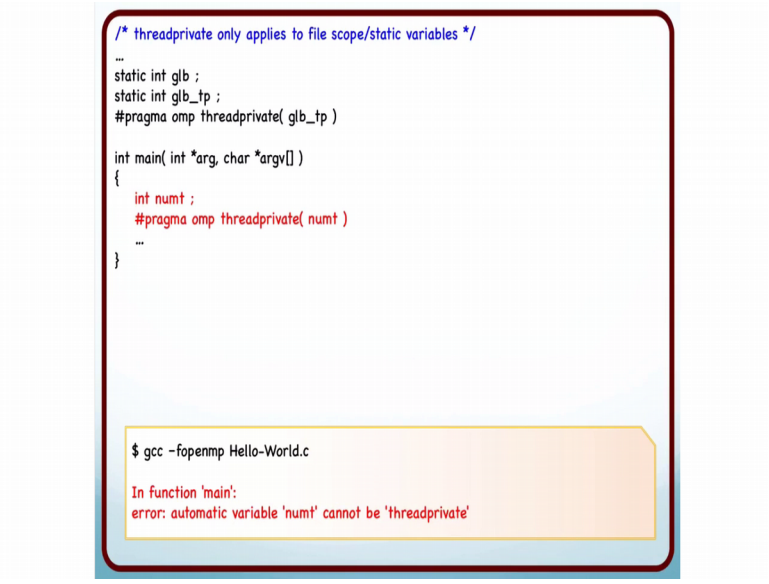
So, the point is that you do not used thread private variables for sharing purposes; you use it for private purposes. Eventually, when you want to accumulate the work across threads; you would use a shared variable for that, not a thread private variable; not a private variable, not a thread private variable, but a shared variable.

(Refer Slide Time: 03:53)



So, this is the output that I get and here is the important part.

(Refer Slide Time: 03:58)



Thread private only applies to file scope of static variables. So, if you defined a variable which is on the stack; if it is local inside a function when you cannot declared to be thread private. So, here is an example this is something I will tried to do int; numt is defining side name and I declared to be thread private.

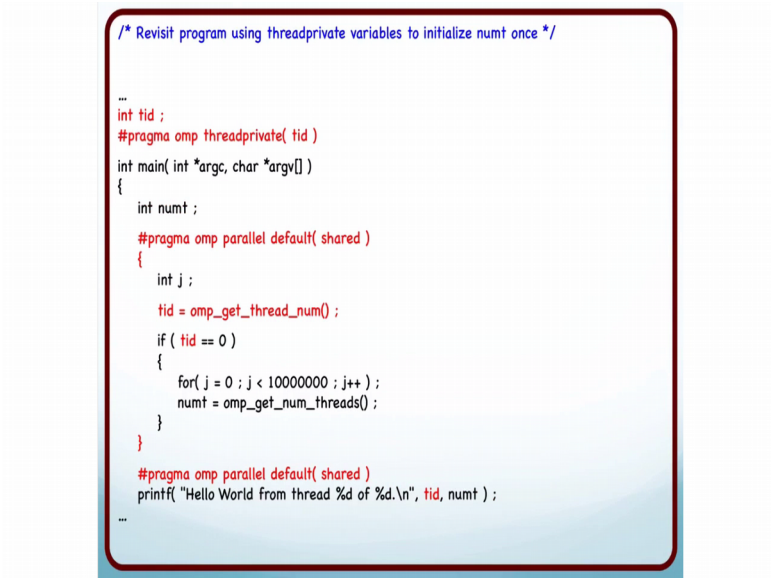
So, when I try to compel it, the compiler will be main error. There is one important thing to point out that; if you access a thread private variable, from outside the parallel region.

What is the value you are going to get? What value should I see in a thread private variable; if I access it outside the parallel region not inside the parallel region? When inside the parallel region each thread will get to see the value that it had set, but what about outside the parallel region.

Student: Value of the main threads.

Main thread; what is the main thread? Thread 0; is the same as the main thread. So, whatever you are doing inside thread 0; that same variable is visible outside the parallel region.

(Refer Slide Time: 05:14)

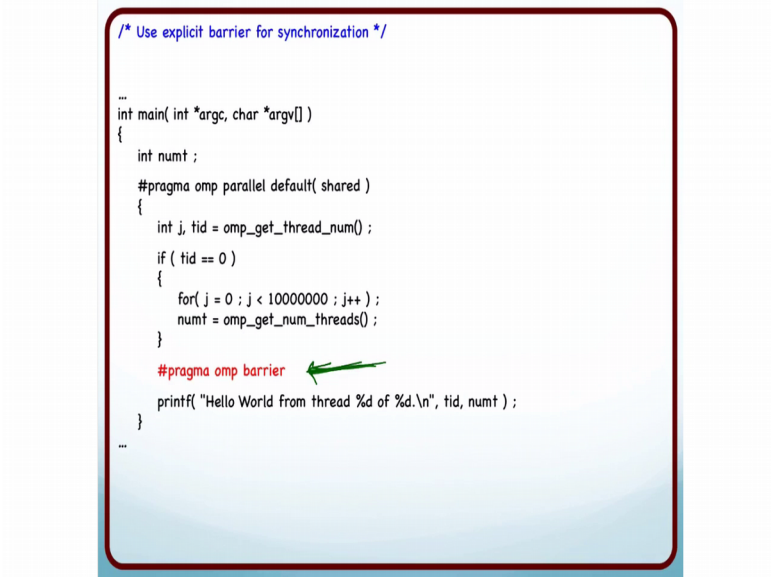


So, this is going back to the same program; where tid is declared to be thread private and then this was the code which executed. So, here what we are trying to do again is something which present make a lot of sense; we are actually using the hash pragma omp parallel region as a point of synchronization between all the threads; to ensure that no thread proceeds ahead beyond this point.

So, this is something call the barrier operation; what is a barrier? A barrier is a point where all the threads must come together before any of them proceeds further. So, all the threads must hit the barrier before proceeding further. So, we are actually using this as a barrier; that is what I am doing. I did not want any thread to print hello world; until thread 0 had initialized numt.

So, what did I do? By defining this parallel region; hash pragma omp parallel and what is the purpose of serving? The only purpose of serving is that no thread can go beyond this point; that is the purpose of serving.

(Refer Slide Time: 06:21)

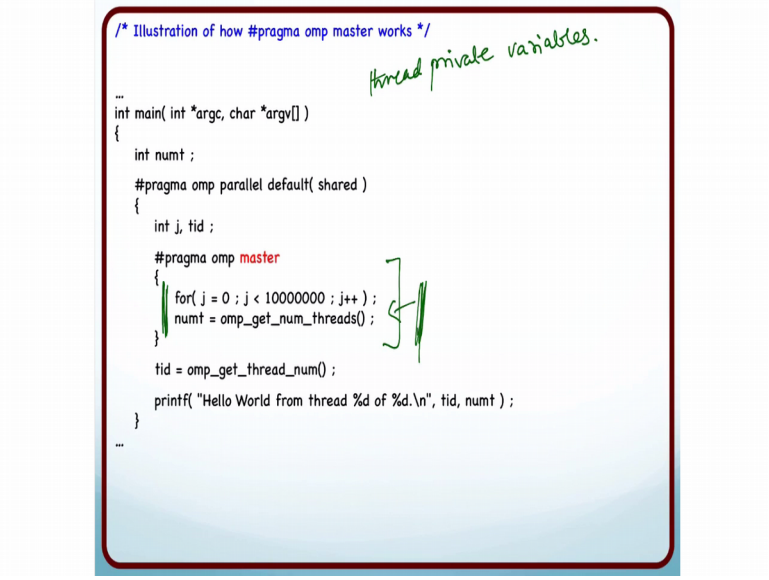


So, there is a; simpler way to do that in openmp; there is this concept called hash pragma omp barrier. So, if I introduce that over here; then it will wait for all the threads to come together at this point in time before any thread to sees further ok.

So, that does the same work that I was trying to do earlier, it serve the same purpose. But now I do not need thread private variables; I mean I was using all these thread private variables and everything because I had 2 separate parallel regions; now I have a single region I do not even need to bother about thread private variables.

So, I can write a simple piece of code; I am just put a hash pragma omp barrier. So, all the other threads will come and weight over here because thread 0 has not come. When thread 0 reaches over here, then all of them will proceed ahead and we know by that time thread 0 would have already called omp get num thread. So, this variable would have been initialized and from this program and I see what I expected.

(Refer Slide Time: 07:24)



Now, instead of using a barrier there is another construct that openmp provides. So, what is so special about thread id 0? Why did I want thread id 0 to initialized the variable? There nothing particular about it; I just had to pick one thread and the only way I need to pick a thread is if I say thread id 0.

So, if you want some work to be only done by one thread is a better way of doing it and that is hash pragma omp single. So, only one thread executes the code inside this construct and it provides the synchronization at the end of the construct. So, at the end of hash pragma omp single, it provides an implicit barrier. So, no thread can proceed ahead until all the things have reached this point. There; obviously, the code inside hash pragma omp single only execute the one thread and it does not have to be thread 0; it can be any thread that is the important part whichever thread comes at first, you will start to you will putting it.

In a parallel program that is want; I have some region which is to be execute by the one thread why do I want thread 0 to do it whoever gets free earlier can come and do it. So, this code also works fine; does what I want to do. Now, there is a option called nowait; so, if you want one thread to do some work and there is no race condition and you do not cared to the threads lower ahead or not, then you can write hash pragma omp single nowait, but with this nowait option what happens is that one of the threads will execute

this code in this construct and all the other threads will proceed ahead without waiting for this thread. And I execute this code and this is what I see again.

There is one other construct this is called hash pragma omp master; it is similar to a hash pragma omp single except that it is always the master thread that executes this part of the code. So, it is always going to be thread 0 the master thread; so, it is like explicitly saying if tid equal to 0.

So, why would I want a particular thread to always execute some code. So, there are lots of times that I am maintaining some state; I will like for instance one of the threads of the master thread which is distributing work to the other threads. Let us say that its keeping track of how much work has already been done, how much is left; maybe there is one thread which is keeping track of all that; that is generally the thread we call the master thread.

So, what will happen is that within this constructed; within this region the variables that you are access are going to be the variables of thread 0 of the master thread. You are not guaranteed that in a hash pragma omp single, then hash pragma omp single the threads that you see maybe different every time you call hash pragma omp single. There will be after thread which is executing that region, but in case of hash pragma omp master. So, if I am calling hash pragma omp master thrice; I will always see the same locations; the same memory locations the same variables.

So, when I say that you are going to see the same variable; I m talking about thread private variables; so, if you have thread private variables you are going to see the same thread private variable which is of thread of 0; every time you write a hash pragma omp master construct to you use that construct and you write code inside that.