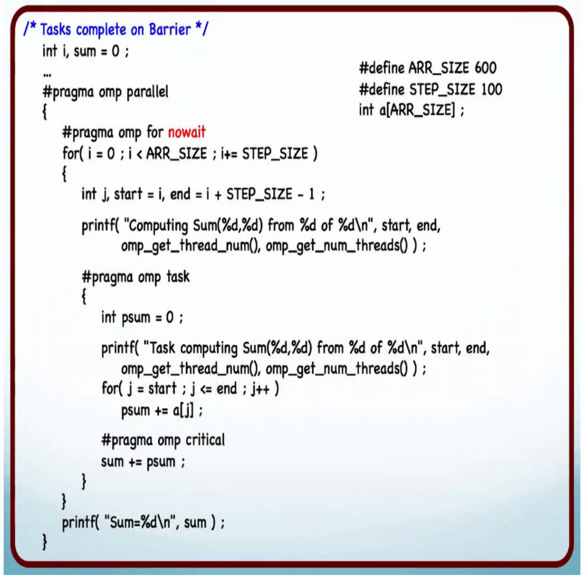
Introduction to Parallel Programming in OpenMP

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

Lecture - 30
Completion of tasks and scoping variables in tasks

(Refer Slide Time: 00:01)



```
/* Tasks complete on Barrier */
int i, sum = 0 ;
...                                          #define ARR_SIZE 600
#pragma omp parallel                         #define STEP_SIZE 100
{                                            int a[ARR_SIZE] ;
    #pragma omp for nowait
    for( i = 0 ; i < ARR_SIZE ; i+= STEP_SIZE )
    {
        int j, start = i, end = i + STEP_SIZE - 1 ;
        printf( "Computing Sum(%d,%d) from %d of %d\n", start, end,
                omp_get_thread_num(), omp_get_num_threads() ) ;
        #pragma omp task
        {
            int psum = 0 ;
            printf( "Task computing Sum(%d,%d) from %d of %d\n", start, end,
                omp_get_thread_num(), omp_get_num_threads() ) ;
            for( j = start ; j <= end ; j++ )
                psum += a[j] ;
            #pragma omp critical
            sum += psum ;
        }
    }
    printf( "Sum=%d\n", sum ) ;
}
```

Now, when do these tasks complete? How do I know that all these tasks are done? I have launched all these tasks. Suppose I want to do something at the end once all the work is over, I want to let us say print that you know how many matrices did I inverse or something on the other or maybe the sum of the elements that are added. So, how do I know that the tasks have completed, right. So, tasks complete on barrier. So, this is something that I have just added on this slide, right. So, suppose this is not there, this nowait is not there, in that case, what happens is there is an implicit barrier at this point when the for ends, right, all the threads proceed beyond this point only after they have completed their part of the for loop.

So, this is an implicit barrier that openMP has whenever you encounter a barrier at that point if there are any tasks that have been created all of them have to complete before you go pass the barrier.

Student: is it only that brace or if I put any brace, it will become a barrier?

No, no, no, this is nothing braces, we saw earlier, right hash pragma omp single even that has an implicit barrier right a lot of openMP constructs have an implicit barrier at the end master does not hash pragma omp master does not have an implicit barrier at the end, but single for all of these have an implicit barrier lot of constructs have an implicit barrier at the end task; obviously, does not.

So, now what happens if I put nowait over here; what happens if I say hash pragma omp for nowait and I execute that same code.

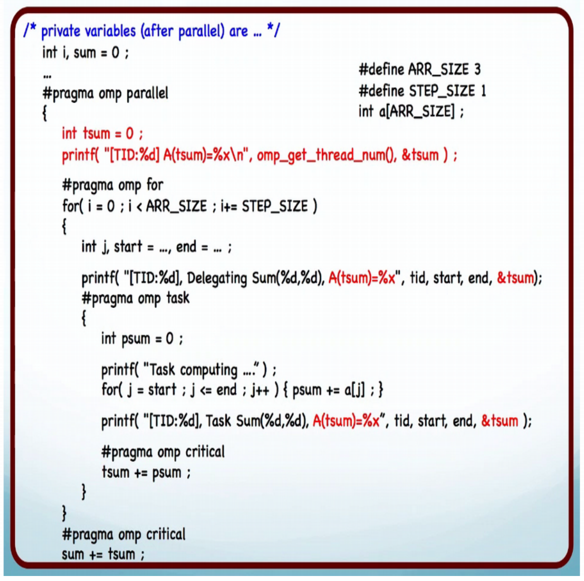
Student: (Refer Time: 01:51)

So sum was been printed here, right, it was coming out correct because there was an implicit barrier over here.

And all the tasks were completing, but in this case, now I have added a no wait. So, what is going to happen? This is what happened on an actual run; I found sum to be equal to 400 sum of the threads were still executing their tasks right one of the threads reached over here and executed this piece of code.

So, one thing I can do is I can explicitly say hash pragma omp barrier here, right. So, even if I have nowait and I have put an explicit barrier over here, in that case what is going to happen; again all the tasks have to be completed on barrier. Therefore, I will again get the correct result, all right. So, the important takeaway is that tasks complete on barrier.

(Refer Slide Time: 02:53)



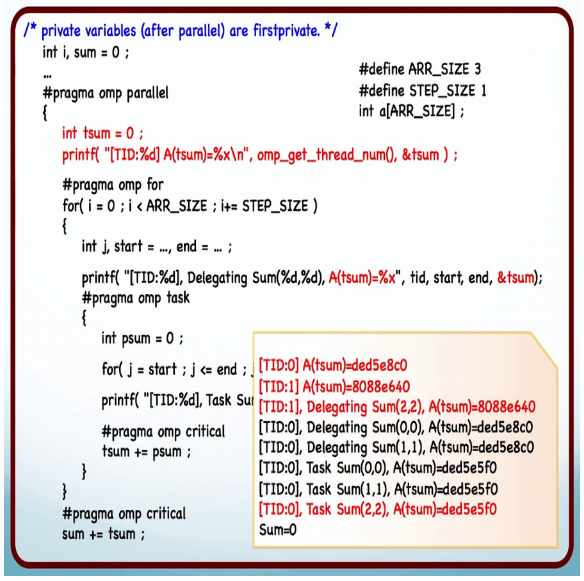
```
/* private variables (after parallel) are ... */
   int i, sum = 0 ;
   ...                                                    #define ARR_SIZE 3
   #pragma omp parallel                                   #define STEP_SIZE 1
   {                                                      int a[ARR_SIZE] ;
      int tsum = 0 ;
      printf( "[TID:%d] A(tsum)=%x\n", omp_get_thread_num(), &tsum ) ;

      #pragma omp for
      for( i = 0 ; i < ARR_SIZE ; i+= STEP_SIZE )
      {
         int j, start = ..., end = ... ;

         printf( "[TID:%d], Delegating Sum(%d,%d), A(tsum)=%x", tid, start, end, &tsum);
         #pragma omp task
         {
            int psum = 0 ;

            printf( "Task computing ...." ) ;
            for( j = start ; j <= end ; j++ ) { psum += a[j] ; }

            printf( "[TID:%d], Task Sum(%d,%d), A(tsum)=%x", tid, start, end, &tsum );

            #pragma omp critical
            tsum += psum ;
         }
      }
      #pragma omp critical
      sum += tsum ;
```

Here is a more involved piece of code that actually shows you what happens to private variables that are declared after the parallel region starts, all right.

So, we are going to look at this variable tsum, right, it is declared after the parallel region starts and what do I do here? I print the thread id, I print the address of tsum; I am printing a memory location that points to tsum, right, that is what this is doing ampersand tsum, all right, then I have this hash pragma omp for and again inside hash pragma omp for; I again print address of tsum. So, I mark here that I am delegating this summation from this start to this end and I am again printing the address of tsum and finally, inside the task again print the same thing the address of tsum; how do you differentiate what is what.

So, here I am just printing tid percent d; here I am printing delegating sum and here I am printing task sum. So, just look out for those indicators. So, when I run this code.

(Refer Slide Time: 03:59)



```
/* private variables (after parallel) are firstprivate. */
  int i, sum = 0 ;
  ...                                              #define ARR_SIZE 3
  #pragma omp parallel                             #define STEP_SIZE 1
  {                                                int a[ARR_SIZE] ;
     int tsum = 0 ;
     printf( "[TID:%d] A(tsum)=%x\n", omp_get_thread_num(), &tsum ) ;

     #pragma omp for
     for( i = 0 ; i < ARR_SIZE ; i+= STEP_SIZE )
     {
        int j, start = ..., end = ... ;

        printf( "[TID:%d], Delegating Sum(%d,%d), A(tsum)=%x", tid, start, end, &tsum);
        #pragma omp task
        {
           int psum = 0 ;

           for( j = start ; j <= end ;

           printf( "[TID:%d], Task Su

           #pragma omp critical
           tsum += psum ;
        }
     }
     #pragma omp critical
     sum += tsum ;
```

```
[TID:0] A(tsum)=ded5e8c0
[TID:1] A(tsum)=8088e640
[TID:1], Delegating Sum(2,2), A(tsum)=8088e640
[TID:0], Delegating Sum(0,0), A(tsum)=ded5e8c0
[TID:0], Delegating Sum(1,1), A(tsum)=ded5e8c0
[TID:0], Task Sum(0,0), A(tsum)=ded5e5f0
[TID:0], Task Sum(1,1), A(tsum)=ded5e5f0
[TID:0], Task Sum(2,2), A(tsum)=ded5e5f0
Sum=0
```

What do I see?

Student: (Refer Time: 04:04).

In this particular case, there are only 2 threads, I run this with 2 threads. So, initially these are the addresses of these memory locations; obviously, they both had their own copy of tsum. So, they had different addresses right now when I was delegating. So, as you can see over here this was being delegated by thread id 1 so; obviously, this address is same as this address, right and these 2 were being delegated by thread id 0. So, these were the same as this address, right; obviously, the same thread is accessing the same variable, but what happened inside the task this is a completely new address well in this particular case what happened was that all three of them ended up being executed a thread id 0, right, but if I had thread id one executing something else, I would have seen another address, right.

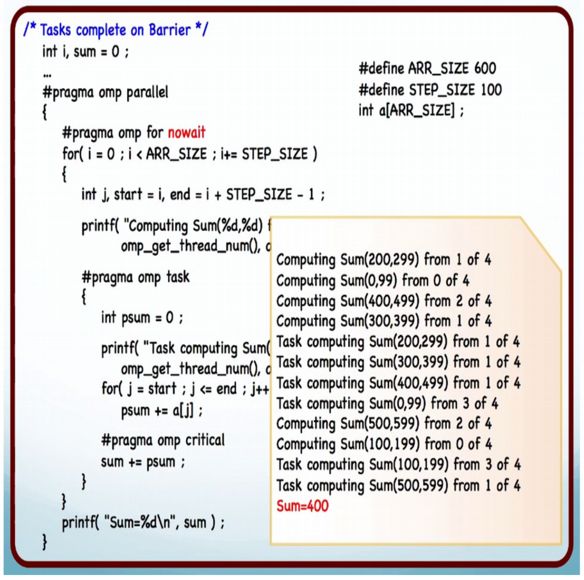
So, I see another address over here inside the task which is different from these addresses sometimes what I want to do is I want to access the variable of the delegator, right the thread that invoked this task; I want to access that variable. So, how can I do that? So, all you need to do is you say hash pragma omp task shared tsum; what that tells openmp is that you want whenever tsum is accessed inside this code inside the task it actually refers to tsum for the thread that was invoking this task thread one create a new copy, it will keep a pointer to that location of the delegator.

So, this is the run of this. So, now, what you see thread id had this address thread id one thread id 0 has this address and while delegating also you see the same addresses and when the thread is executed then also you see the same addresses. So, thread id 0 is the one which was delegating the work from summation 0 0 that was executed over here and you see that this pointer is the same as this pointer here similarly thread id 0 was a delegating sum 1 1 and you see the same pointer over here for sum 1 1 and sum 2 2 was delegated by thread id 1, right.

So, the address of tsum was this, but it was executed on thread id 0, but the address of tsum is that of the delegator just think of it as what do you do in hash pragma omp parallel or at other places right what does shared mean shared means that whenever I access this variable in this region I wanted to refer to the same variable that is just before this region starts. So, that is exactly what it saying the delegator is the one which was creating this task. So, all its saying is that this variable inside the task is going to refer to the same location that this variable was referring to just before the task started.

Student: Could it (Refer Time: 07:28) up to the (Refer Time: 07:30).

(Refer Slide Time: 07:31).



```
/* Tasks complete on Barrier */
   int i, sum = 0 ;
   ...
   #pragma omp parallel
   {
      #pragma omp for nowait
      for( i = 0 ; i < ARR_SIZE ; i+= STEP_SIZE )
      {
         int j, start = i, end = i + STEP_SIZE - 1 ;

         printf( "Computing Sum(%d,%d) f
               omp_get_thread_num(), c

         #pragma omp task
         {
            int psum = 0 ;

            printf( "Task computing Sum(
               omp_get_thread_num(), c
            for( j = start ; j <= end ; j++
               psum += a[j] ;

            #pragma omp critical
            sum += psum ;
         }
      }
      printf( "Sum=%d\n", sum ) ;
   }
```

```
#define ARR_SIZE 600
#define STEP_SIZE 100
int a[ARR_SIZE] ;
```

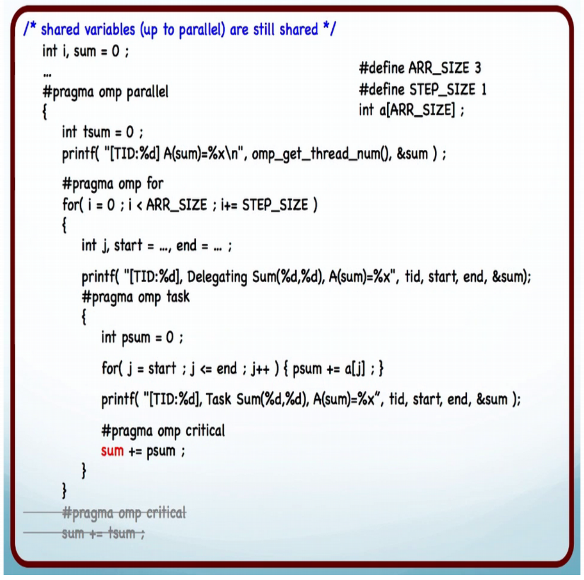
```
Computing Sum(200,299) from 1 of 4
Computing Sum(0,99) from 0 of 4
Computing Sum(400,499) from 2 of 4
Computing Sum(300,399) from 1 of 4
Task computing Sum(200,299) from 1 of 4
Task computing Sum(300,399) from 1 of 4
Task computing Sum(400,499) from 1 of 4
Task computing Sum(0,99) from 3 of 4
Computing Sum(500,599) from 2 of 4
Computing Sum(100,199) from 0 of 4
Task computing Sum(100,199) from 3 of 4
Task computing Sum(500,599) from 1 of 4
Sum=400
```

Student: If barrier was not there and all thread would have printed some sum, right?

Yeah. So, I have just showed you one output all threads would print sum some of them may end up printing correctly one of the threads would definitely end up printing it correctly, the one which executes the last task, right.

But all I wanted to show was that you can get incorrect results right in some cases one thread may give you one digit.

(Refer Slide Time: 08:10)



```
/* shared variables (up to parallel) are still shared */
    int i, sum = 0 ;
    ...                                                     #define ARR_SIZE 3
    #pragma omp parallel                                    #define STEP_SIZE 1
    {                                                       int a[ARR_SIZE] ;
        int tsum = 0 ;
        printf( "[TID:%d] A(sum)=%x\n", omp_get_thread_num(), &sum ) ;

        #pragma omp for
        for( i = 0 ; i < ARR_SIZE ; i+= STEP_SIZE )
        {
            int j, start = ..., end = ... ;

            printf( "[TID:%d], Delegating Sum(%d,%d), A(sum)=%x", tid, start, end, &sum);
            #pragma omp task
            {
                int psum = 0 ;

                for( j = start ; j <= end ; j++ ) { psum += a[j] ; }

                printf( "[TID:%d], Task Sum(%d,%d), A(sum)=%x", tid, start, end, &sum );

                #pragma omp critical
                sum += psum ;
            }
        }
        ~~#pragma omp critical~~
        ~~sum += tsum ;~~
```

This sum over here; right this is actually defined out say it omp parallel. So, it is actually shared again its printing these values I see that all of them, whether I am accessing it from the parallel region or inside the for or inside the task all of them are actually pointing to exactly the same location its treated as shared. So, all addresses of the same and this is just demonstrating that you know you can create tasks from single thread using single right we already spoke about this in the context of a link list.