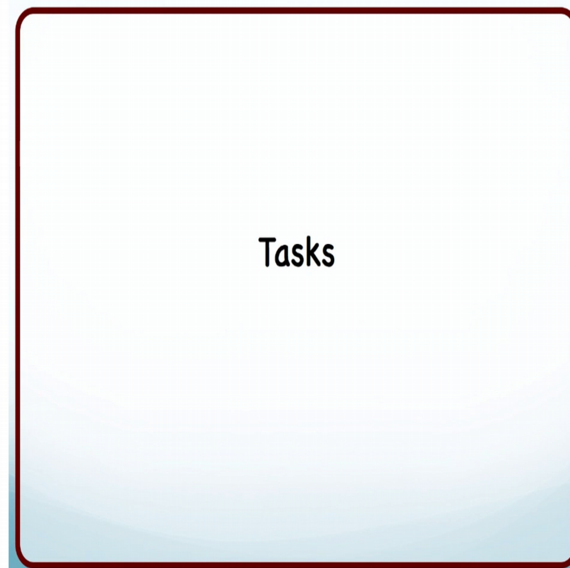
Introduction to Parallel Programming in OpenMp
Dr. Yogish Sabharwal
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture - 08
OpenMP Tasks

(Refer Slide Time: 00:26)



We will start with tasks what are openmp tasks.

(Refer Slide Time: 00:31)

```
/* Computing Array sum using tasks */
   int i, sum = 0 ;
   ...                                           #define ARR_SIZE 600
   #pragma omp parallel                          #define STEP_SIZE 100
   {                                             int a[ARR_SIZE] ;
      #pragma omp for
      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 ) ;
```

So here is a code an openmp code for computing the sum of an array right; some of the elements of an array which uses openmp tasks. So, as usual you have your hash pragma omp parallel right that is where your parallel region begins. So, at this point of time multiple threads start executing right and what we are doing here is we are dividing this array up. So, array is of size ARR size right.

And we are dividing it up into pieces each of size step size all right. So, this is what this for loop is doing for I is equal to 0, I is less than the ARR size I plus equal to step size right. So, it jumps in blocks of step size and what does it do inside the for loop. So, it sets a start and an end. So, start is set to I and end is set to I plus step size minus 1 right is going to work on this piece of chunk starting at location I, up to location I plus step size minus one right. So, it is a its a chunk of size step size and here we are assuming that we are all size is a multiple of step size right we do not want to get into boundary cases for now and then it basically prints that I am computing the sum from which thread number right.

So, the first integer here is going to be the thread number and the second integer here is going to be the number of threads right. So, it is just saying that who is doing the computing right. And now at this point which specifies something called a task what is a task? A task is a piece of code that can be executed independently at any point in time. So, it is an independent piece of code that need not be executed right, now it can be scheduled later, but it is something that can be spawned of that can be put aside that this is some piece of code that needs to be executed and executed whenever you have time something of that sort right.

So, what is being done here is that a task is being created, whatever is specified in the structured block immediately following hashed by more mp task, that is a piece of code that is going to get executed independently at some point in time right all right and what do I do inside this task? I basically initialize a partial sum variable p sum to 0 and then I report that again which task is computing this sum right.

So, I say task computing sum from which thread number and what is the total number of threads right and it actually computes the partial sum over here and finally, it adds this sum to the shared variable sum right. So, this is sum is a shared variable over here.

(Refer Slide Time: 03:29)

```
/* Computing Array sum using tasks */
int i, sum = 0 ;
...
#pragma omp parallel
{
    #pragma omp for
    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(), o

        #pragma omp task
        {
            int psum = 0 ;

            printf( "Task computing Sum(
                omp_get_thread_num(), o
            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(0,99) from 0 of 4
Computing Sum(100,199) from 0 of 4
Task computing Sum(0,99) from 3 of 4
Computing Sum(400,499) from 2 of 4
Task computing Sum(100,199) from 0 of 4
Computing Sum(200,299) from 1 of 4
Computing Sum(500,599) from 2 of 4
Task computing Sum(500,599) from 3 of 4
Task computing Sum(400,499) from 0 of 4
Task computing Sum(200,299) from 3 of 4
Computing Sum(300,399) from 1 of 4
Task computing Sum(300,399) from 2 of 4
Sum=600
```

And when I execute this code, this is what I see. So, first this print is coming from here right this print computing some 0 to 99 from thread 0 or 4. So, this hash pragma omp 4 is dividing this for loop amongst the.

Student: (Refer Time: 03:51).

Different threads right. So, different threads are going to execute different iterations the first iteration went to thread 0 right. So, it was responsible for computing some 0 to 99, because step size is 100 and the total error size is 600. So, there will be 6 iterations and then the second iteration was also done by thread number 0.

So, at this point of time what did it do it spawned off a new task, it is just a block of code which is going to a get executed at some point and time, it could be executed by some other thread it need not be executed by the same thread. It is just like creating work is just saying that this is some piece of work that needs to be done, whoever is free whenever he is free comment do this work all right. So, what happened here is that this sum 0 to 99 was created by thread 0 or 4, and if you see down here right this is printed from this print f inside the task.

So, what does this indicate? This indicates that this task was actually performed by thread number three. The task was spawned by thread number 0, but the task was performed by thread number three all right and similarly here what you see is that task 0

spawned this task which was supposed to do the summation for 100 to 199 and in this particular case thread 0 itself ended up executing that task ok.

So, why do we need tasks? Look it is not always simple to say that who is going to do what right. So, we have been looking at simple examples where you have simple for loop which you say I divide among 4 threads it is very easy to divide you just divided in 2 iterations, either you put a hash pragma omp 4 or you yourself divide the work up and tell if my thread number is this I will do this piece of work right.

But life is not always that simple, a lot of times you do not know what is the work that needs to be done. You do not know how much work needs to be done; you do not know how to divide it amongst the threads. This is a very very simple mechanism which just says that here is an independent piece of code that can be executed by anybody I am just telling you that this is an independent piece of code right execute it whenever some thread is free ok.

So, I can spawn off work as and when I wish all right. Also we have been looking at examples like arrays right, but what if you have to traverse more complex data structures like let us say trees right or hash table or something of that sort right where you do not know what is the number of elements what is the number of iterations you have to perform. So, there as in when you are traversing the data structure you are realizing that more and more work needs to be done, you cannot divide it up front right.

So, in all these cases tasks are very very handy all right. Let us complete this example first then we will get into more details of tasks. So, at different points of time, different threads are executing, different iterations the spawn of the tasks and the tasks get computed on some thread or the other right and so on and finally, I print the sum down here and I get a sum of 600.