

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

Lecture - 29
Accessing variables in tasks

(Refer Slide Time: 00:01)

```
/* 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) 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 ) ;
```

And what about the data that I am accessing inside the region hash pragma omp task which data am I referring to?

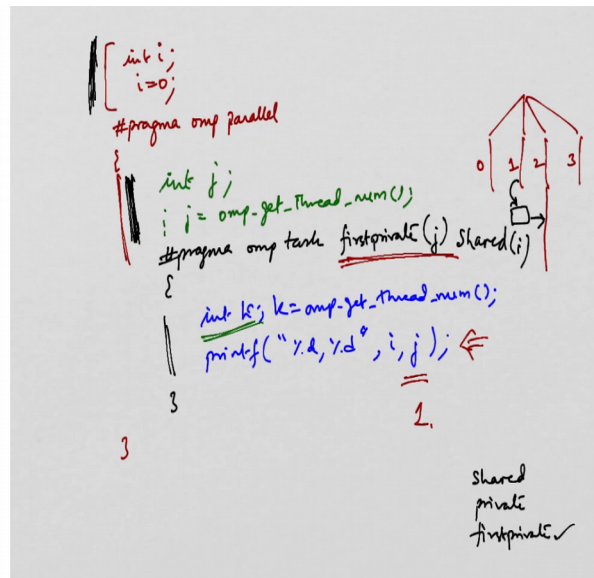
Student: (Refer Time: 00:10).

That is a very important question.

Student: The thread which created it.

The thread which created it.

(Refer Slide Time: 00:20)



So, here is some data that I have defined right let us say int i, and then I say hash pragma omp parallel and inside this now, I say int j and again I write some code and then I say hash pragma omp task and inside that I say int k. So, I initialise i to be equal to 0 and I initialise j to be equal to omp get thread num and finally, I print i comma j.

So, what is going to get printed over here? So, let us worry about I later, let us first talk about j. So, j is what. So, j is omp get thread num. So, it is the thread number. So, at this point of time 4 threads are executing right three threads got launched, and one of them has thread id 0 one of them has thread id 1, thread id 2, thread id 3. Now let us say that thread one created a task which is now finally, getting executed on thread number two right is it clear.

So, I created a task on thread 1 and it finally, got executed by thread number 2 what will get printed over here?

Student: (Refer Time: 02:04).

One or two for j I am not asking for I right now.

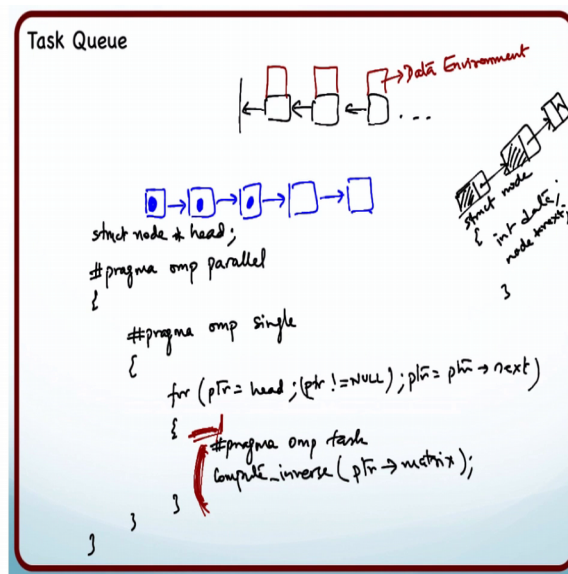
Student: One.

One even though thread 2 is executing it, what you will see over here is one. Why because when I am executing this code and I am creating this task, as a programmer right

what are the variable that I am referring to what am I trying to refer to. I am trying to refer to the variables that are just in the context over here, how can I refer to the variables throughout the thread where is going to get scheduled I do not even know which thread is going to get scheduled on. So, that is never mind tension as a programmer to do that right.

As a programmer my intention is always to access those variables, which were just being accessed before this hash pragma omp task.

(Refer Slide Time: 02:56)



So, what openmp does is that when it queues up a task it also associates some data environment with it. By default what it does is it creates a new copy of all these variables that are in the context of the thread.

(Refer Slide Time: 03:17)

```
/* 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) 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 ) ;
```

All right. So, at this point of time when this task is encountered it creates a new task.

It also creates a new variable `j` and a new variable `start` and a new variable `end`. It allocates memory for these variable and copies the content from whatever the value was just for this task was initiated all right that is why in this what you will see is that the value of `j` `start` and `end` will be just what you intended it to be `start` will be `i` `end` will be `I plus step size minus one` that is what will be passed to the task.

Student: Can we say that (Refer Time: 03:57) one created the task and the task is depend upon the (Refer Time: 04:00) some memory of (Refer Time: 04:03) or thread number 2 that the same incident it created some other memories means for thread one it was `j`, but for thread 2 (Refer Time: 04:17) is not actually created some other memory of this and this (Refer Time: 04:21).

Yeah yeah. So, additional memory is being created right a copy is being created it is not the same variable, it is not the same memory location a fresh copy is being created. So, that is the default behaviour.

Student: So, this (Refer Time: 04:36) copying and (Refer Time: 04:39).

There is some overhead, but that is right that is slight same overhead has you invoke a function.

Student: There would not be some cache for thread initially thread (Refer Time: 04:55).

So, you are getting into something deep over here, you are saying that this code was executing on thread 0 which was executing on some core right and the data decided in so on cache or registers and now you spawned of the task, which is getting executed on some other core which has its own cache and registers and therefore, when it starts executing the code it is not going to find these variables in the cache or the registers and therefore, it is going to have to load yeah that is true.

Student: So, if I have a (Refer Time: 05:22) if we pass arrays.

Which have to be never pass arrays, you pass pointers.

Student: We pass chunks of arrays (Refer Time: 05:31).

No no you do not even pass chunks of arrays, you pass the pointer to the starting of the array or the.

Student: (Refer Time: 05:36).

Particular location of the array, but yeah that that is a fair point that if you have already let us say it taken an iteration over the data and it is in the cache right let us say the entire arrays in the cache, and now you are off loading the processing to another task. If the cropping of this variable is not the issue because what you will pass to the other thread is just a pointer to where you wanted to start working on in the array, but pointer copy is just a.

Student: (Refer Time: 05:59).

Four byte or 8 byte copy right that pointer is going to get loaded by the other core, that is not an issue that is fine that is one miss right one cache miss, but after that now if it has to traverse the array, yes it is going to load all the elements of that array to process it right that something you have to be aware of if you are trying to shoot for that ultimate performance right.

If you have already loaded that data and one core and then you have creating a task which is going to work on half that data and it is going to have to reload it in the cache

yeah you have to be careful about those things. These are minor things I do not think you should be worrying about this data being copied in and out right.

Student: So, in the example that we saw (Refer Time: 06:45) `omp_get_thread_num` we would get actual thread number (Refer Time: 06:53).

Yeah here if I said `k = omp_get_thread_num` I would get the thread which is executing this task all right. We saw this here, from inside also we printed which is the task computing the sum right and we saw that though the different task which is actually computing the sum all right. So, all the variables that are defined after `hash pragma omp parallel`, here the parallel region starts all variables defined after that.

If I access them inside the task actually a new copy is created for them right this is same as scoping the variable as `first private j`. So, just like you have `shared` and `private` right similarly you can specify variable to be `first private` what that says is. So, that create a copy of this variable and initialise it to the value just before this point. So, you can actually use `first private` event in `hash pragma omp for` and other places right where ever you are starting a new region, you can specify `first private` just like you specified `shared` and `private` all right.

So, this is the default behaviour in tasks variables which are declared after `hash pragma omp parallel`, where is the parallel region starts all of them are treated by default as `first private` all right. What about these variables which are outside the `hash pragma omp parallel`.

Student: They are global.

They are global even the threads were viewing them as global. So, why should the tasks see you them as `private` right why should the tasks have a private copy? I mean the threads also wanted to be the same copy global copies, as a programmer I probably wanted to access the global variable right when I was trying to access that inside the task. So, that is why these are treated as `shared` variables. So, this is equivalent to saying here `shared I`.

So, let us go back we actually already did this, without explicitly mentioning it, but if you see this code right `sum plus equal P sum`. So, we accessed `sum` was actually a global

variable declared outside the parallel region right. So, I was actually accessing the same copy of sum every time and what about k this is private.

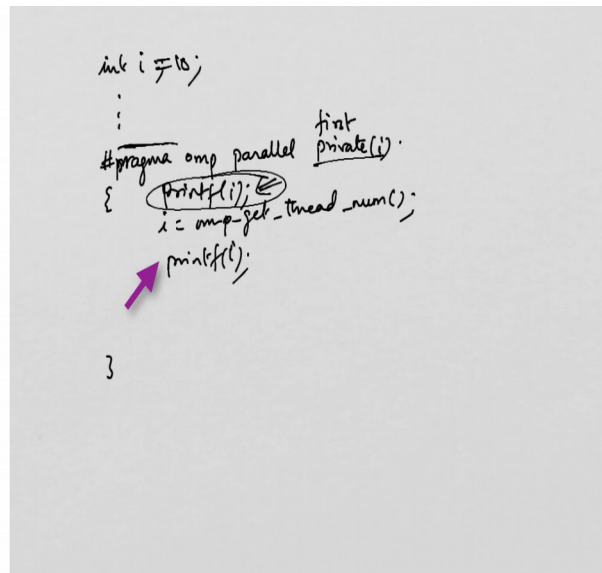
Student: (Refer Time: 09:32).

This is inside the task right. So, I will create memory for it which is just associated with the lifetime of the task and once the task is over that memory will be gone let see this is simple, this is nothing to do with outside world.

Student: Repeat it what do you mean by first private (Refer Time: 09:50).

First private. So, you remember private right.

(Refer Slide Time: 09:52)



```
int i = 10;
...
#pragma omp parallel first private(i)
{
    printf(i);
    i = omp_get_thread_num();
    printf(i);
}
}
```

The image shows a handwritten code snippet on a light gray background. The code is as follows: `int i = 10;`, followed by three vertical dots. Then `#pragma omp parallel first private(i)`. Below this is an opening curly brace `{`. Inside the brace, there are three lines: `printf(i);`, `i = omp_get_thread_num();`, and `printf(i);`. A purple arrow points from the second `printf(i);` line to the `printf(i);` line above it. A purple circle is drawn around the first `printf(i);` line. To the right of the `first private(i)` text, there is a handwritten note `first private(i)` with a purple arrow pointing to the `printf(i);` line above it. The code ends with a closing curly brace `}`.

So, I have a variable `int i` and at some point of time I decided to create some threads and I said `i` is equal to `omp_get_thread_num()`. I wanted to get the thread number, but `i` is declared outside. So, I had to explicitly say `private i` right remember this right that is what `private i` that a separate memory location it is create space for `i`.

The only difference is `first private` does exactly the same thing just set it initializes this `i` with whatever was the value just before the parallel region is encountered that is all nothing else just initializes that value. So, if I said `int i` is equal to 10 over here, it will copy that ten over here if I if I print `i` here, I will get the value 10 right of course, if I print it over here I will get the thread number, but if I print it over here I will get the

value of 10. If I said `private i` and I printed the value over here I would get some garbage
it clear is it is nothing, but `private` with an initialisation that is all it is.