

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

Lecture – 19
OpenMP Scoping variables and some race conditions

(Refer Slide Time: 00:01)

```
/* What happens if we declare numt and tid outside the parallel block? */
#include <omp.h>
#include <stdio.h>

int main( int *argc, char *argv[] )
{
    /* Declare numt and tid outside parallel block */
    int numt, tid ;

    #pragma omp parallel
    {
        numt = omp_get_num_threads() ;
        tid = omp_get_thread_num() ;

        printf( "Hello World from thread %d of %d.\n", tid, numt ) ;
    }

    return 0 ;
}
```

Hello World from thread 2 of 4.
Hello World from thread 0 of 4.
Hello World from thread 1 of 4.
Hello World from thread 3 of 4.

Hello World from thread 1 of 4.
Hello World from thread 0 of 4.
Hello World from thread 1 of 4.
Hello World from thread 2 of 4.

This example that we saw last time although the program seems very simple there is a race condition that exists right we saw that last time. So, the race condition happened because of the order in which thread one and thread 3 got executed right. So, before thread 3 could print thread 1 came and updated the thread id variable and therefore, what thread 3 saw was also the value one which was not what it had actually fetched. How do we fix this? So, we are going to look at a lot of race conditions, race condition is something that you have to become familiar with you have to realize how race conditions happen right the more example you see the more aware you will become and you know you will be better equipped to fix it.

And we will see how to fix this condition, but we will walk through some example codes and see these race conditions as they happen, right. So, how do we fix this?

(Refer Slide Time: 00:51)

```
/* Explicitly declare tid to be private (and numt to be shared) */
#include <omp.h>
#include <stdio.h>

int main( int *argc, char *argv[] )
{
    int numt, tid ;
    /* specify private and shared variables */
    #pragma omp parallel private ( tid ) shared ( numt )
    {
        numt = omp_get_num_threads();
        tid = omp_get_thread_num();
        printf( "Hello World from thread %d of %d.\n", tid, numt );
    }

    return 0 ;
}
```

So there is a simple mechanism that openMP provides, what you do is you declare your variable to either be private or shared. What does it mean for a variable to be private? It means that every thread is going to have its own copy of that variable. In this particular case tid is declared outside, but its behavior would be the same as if tid was declared inside this region it is like a private copy.

So, whatever I write into tid within this parallel region is only visible to me to the same thread right and if any other thread writes to it it is tid it is only visible to it I do not get to see that right threads do not get to see each other's variable, all right and what does shared mean? Shared means that the variable is global, it is shared by all the threads. So, all the threads get to see the same memory location. In the case of private the memory location is different for each thread where that variable is stored in case of shared it is the same memory location that they are accessing any questions.

Student: If you do not write shared (Refer Time: 02:05).

Right then it comes down to what is the default behavior, for any variable for which you do not specify the scope what is the default behavior right. So, an openMP, we will come to that the default behavior is shared. If you are using a variable which is declared outside, but used inside a parallel region and you have not specified the scope in the parallel region then it is treated as shared.

Student: Is this an openMP specification or it is general implementation.

This is openMP specification. So, every compiler has to adhere to this. So, now, when I print right, I get this output.

So, this is the output that I wanted to get right it is a correct output; why because if you recall in the last example the problem was happening because tid ended up being shared right that is not something I wanted every thread should have it is own copy of what is my thread number what is my thread id.

(Refer Slide Time: 03:00)

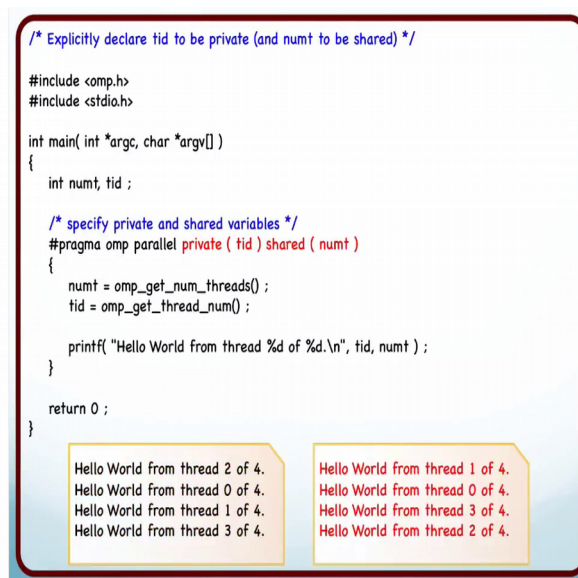
```
/* Explicitly declare tid to be private (and numt to be shared) */
#include <omp.h>
#include <stdio.h>

int main( int *argc, char *argv[] )
{
    int numt, tid ;

    /* specify private and shared variables */
    #pragma omp parallel private ( tid ) shared ( numt )
    {
        numt = omp_get_num_threads() ;
        tid = omp_get_thread_num() ;

        printf( "Hello World from thread %d of %d.\n", tid, numt ) ;
    }

    return 0 ;
}
```



Hello World from thread 2 of 4. Hello World from thread 0 of 4. Hello World from thread 1 of 4. Hello World from thread 3 of 4.	Hello World from thread 1 of 4. Hello World from thread 0 of 4. Hello World from thread 3 of 4. Hello World from thread 2 of 4.
--	--

Yeah, I run it again, I again see the correct output, but how do we know for sure that we have fixed the bug I mean when a race condition exists. Sometimes it happens sometimes it does not happen. So, you should try to figure out how the race condition is happening and then you can test out whether that race condition is for real or not, right.

(Refer Slide Time: 03:19)

```
/* Back to the code with shared tid */  
  
#include <omp.h>  
#include <stdio.h>  
  
int main( int *argc, char *argv[] )  
{  
    int numt, tid ;  
  
    #pragma omp parallel  
    {  
        numt = omp_get_num_threads() ;  
        tid = omp_get_thread_num() ;  
  
        printf( "Hello World from thread %d of %d.\n", tid, numt ) ;  
    }  
  
    return 0 ;  
}
```

How do we make this code fail (almost) every time?

So, let us go back to the code there num t and tid were shared right. So, this is the buggy code. So, how can I make this code fail every time why does this code fail? After I fetch the thread id and before I have printed the value of tid some other thread gets scheduled how can I make this program fail every time by introducing something some statements or some.

Student: (Refer Time: 03:46) we can make a remarkable (Refer Time: 03:49) having the tid value and printf (Refer Time: 03:52).

Correct. So, here is a simple way of making this program fail, right.

(Refer Slide Time: 03:55)

```
/* Make the code (with tid shared) fail (almost) everytime */
...
int numt, tid ;
#pragma omp parallel
{
  int j ;

  numt = omp_get_num_threads() ;
  tid = omp_get_thread_num() ;

  /* Introduce wait so that other threads get scheduled */
  for( j = 0 ; j < 10000000 ; j++ ) ;
  printf( "Hello World from thread %d of %d.\n", tid, numt ) ;
}
...
```

Handwritten notes:

- thr0: numt = 0, tid = 0
- thr1: numt = 1, tid = 1

Diagram:

```
graph TD
  mint --> 1
  mint --> 2
```

So, what do I do I introduce a long wait there are various ways of introducing waits I have just picked one very simple way, I have introduced a loop for j equal to 0, j less than 10 million right this is enough for the thread to get schedule out or if you are executing this on a multiprocessor, this processor is waiting for a long period of time and during that time other processors will execute their instruction right. Remember the word that I used right almost you can never insure that your program will always fail for whatever reason all the threads may end up getting stuck somewhere and not getting executed and they might end up executing in the correct order that does not cause this particular race condition to happen, right.

How do these threads get executed in the presence of this wait? What is most likely to happen? Let us take the simple case of 2 threads, right we have 4 threads, but let me just explain this using 2 thread. So, what will happen is. So, there is a thread 0 and there is a thread one right let us say thread 0 comes first, it will invoke num t equal to openMP get num threads it will get the number of threads it will execute tid equal to 20 get thread num, and what will happen when it executes this instruction it is going to store 0 in tid, right, it will get the value 0 and this is thread id 0.

And now I have introduced a long wait right. So, it is probably going to spend a long time over here, and in the meanwhile while it is waiting over here this thread is going to come it is going to initialize num t and then it will set tid equal to 1, right because of this

long wait right because thread 0 has not gone ahead and printed the value. And now thread one will continue execution and it will also get into a long wait, at some point of time both of them will come out of their long waits right and what will happen at this point in time. So, both of them will try to print and what will both of them end up printing? Whichever tid was written last right in this case in this small example if tid equal to 1.

So, both of them will end up printing one, but you get the idea right what has this long wait done? It have been soured that all the threads have executed instructions up to this point right and therefore, all the instructions after this will get executed by the threads after all the threads are executed all the instructions before this wait because all the threads are going to come in wait of. So, this wait right there are different ways that you can invoke this wait and you have to be careful while invoking this wait a lot of times you will find that the code actually does not get into a long wait. Because compilers are smart enough they will trigger out that whatever is being updated over here the (Refer Time: 06:48) j is not getting used and they might throw with this code, right.

So, if you are using some compiler options like there are various compiler options o 1,o 2, o 3 you can use advanced compiler options the more advanced compiler options you use those smarter of the compiler tries to act right it will try to remove redundant code and so on then you would not see this happening. So, if you are trying to check your code via this mechanism make sure you are not compiling with o 3, do not compile with any advanced compiler option.

(Refer Slide Time: 07:14)

```
/* Make the code (with tid shared) fail (almost) everytime */

...
int numt, tid ;

#pragma omp parallel
{
    int j ;

    numt = omp_get_num_threads() ;
    tid = omp_get_thread_num() ;

    /* Introduce wait so that other threads get scheduled */
    for( j = 0 ; j < 10000000 ; j++ ) ;

    printf( "Hello World from thread %d of %d.\n", tid, numt ) ;
}
...

Hello World from thread 3 of 4.
Hello World from thread 3 of 4.
Hello World from thread 3 of 4.
Hello World from thread 3 of 4.

Hello World from thread 2 of 4.
Hello World from thread 2 of 4.
Hello World from thread 2 of 4.
Hello World from thread 2 of 4.
```

So, now if we execute this code with 4 threads this is the output i. So, in this case what has happened thread three filled up tid last right and then I run again maybe some other thread will be of tid last right. So, the output is going to depend on whichever thread filled up tid last right, but what is important is that you know if you have a hunch that this is where the problem is in my code, you are able to reproduce that problem. So, sometimes it is useful to know that. So, now, when I think I have fixed the code I again let that wait statement stay and I can again check whether are fixed abode or not right and again I see the correct output. I again see the correct output when I; right.

So, this is with the fixed code I had declared private tid right again this does not guarantee that your code is correct, but I mean when you are debugging right this is a useful tool introducing waits to try to catch race conditions if something very useful let us try to do something else now. Let us try to optimize this code a little bit. So, what is happening is that num t is the same for all the threads, with number of threads is the same right no matter whether I am calling from thread 1, 2, 3 or 0, I mean I am making 4 calls to omp get num threads since like a wait.

So, can I optimize on that I do not want to make 4 calls why should I make 4 calls. So, maybe what I can do is that I can initialize num t outside right why do I need to do it from every thread.

(Refer Slide Time: 08:52)

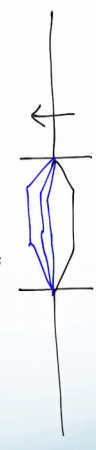
```
/* Since numt is same for all threads, we can declare and initialize it outside */
#include <omp.h>
#include <stdio.h>

int main( int *argc, char *argv[] )
{
    /* Declare and initialize numt outside parallel block */
    int numt = omp_get_num_threads();

    #pragma omp parallel default( shared )
    {
        // numt = omp_get_num_threads();
        int tid = omp_get_thread_num();

        printf( "Hello World from thread %d of %d.\n", tid, numt );
    }

    return 0 ;
}
```



Hello World from thread 2 of 1.
Hello World from thread 0 of 1.
Hello World from thread 3 of 1.
Hello World from thread 1 of 1.

So, this is the code for that. So, whatever I have done, I have just commented out that num t equal to omp get num threads inside the parallel region and I put it outside any problem. So, now, when I execute this code this is what I get. So, remember openMP follow the fork join model.

So, what happens is that till this point where it encounters the hash pragma omp parallel region, there is a single thread that is getting executed the master thread right and when it encounters this region between where this region starts and where this region ends, what is going to do is it is going to fork in this case three additional thread right and the master thread is going to continue executing one of the threads, and eventually when this region ends at that point of time it is going to wait for all the threads to come and join, and then again I will have a symbol thread that executes from there right.

So, now, what is the problem? I try to get the value of omp get num threads at this point in time. What is the number of threads at that point in time only one. So, that is why the value that is what was one right. So, how do you fix this we want to initialize the number of threads only once. Here is another way of trying to achieve that maybe I can just invoke it from one thread right maybe one thread can make a call to omp get num threads why do I need to have all the threads within the call. So, I insert this code. So, first I make a call to omp get thread num from each thread, I fetch the value into tid and now the thread which has tid equal to 0 I ask it to initialize omp get num threads right.

So, only one thread will make a call to `omp_get_num_threads` and then everybody prints hello world from thread percent d of percent d right. So, now, when I try to run this since to run fine I run it again I get some garbage. So, why did this happen? In this particular case thread number 1 and thread number 2, they basically went past this instruction and printed hello world even before thread 0 code come and execute this code right how do I make this fail every time.

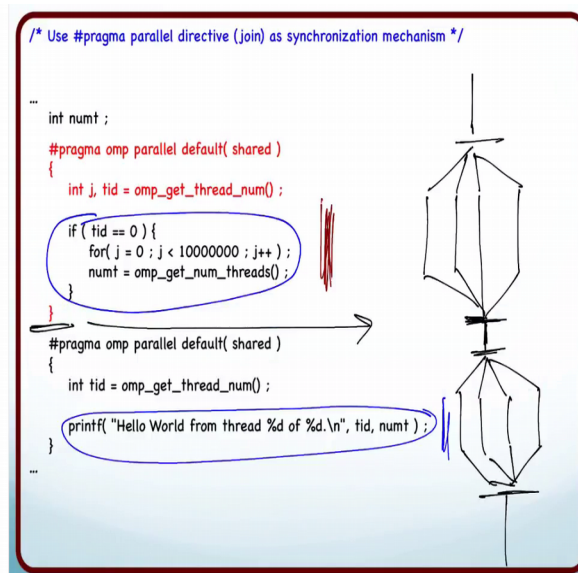
Student: Long wait (Refer Time: 11:44).

Long wait where do I put the long wait.

Student: (Refer Time: 11:49).

So, I can put a long wait inside this if condition, what will happen because of this? Thread 0 will go into a wait state and not come out of this for a long time, it will not call `omp_get_num_threads` for a long time because it is waiting, and in the meanwhile the other threads are going to go ahead and print hello world right. So, if I run this code now I will most probably see an output like this, right.

(Refer Slide Time: 12:25)



Here is another way to do this, what do I want? I want that the threads should not go and execute the print statement until thread 0 has initialized num t right. In other words I want these instructions to be executed before any thread executes this instruction right is that clear.

If I can ensure that then the output will be correct. So, how do I do that? There is one way of doing it. So, I have 2 different parallel regions in the first region what do I do? For tid 0 it makes a call to `omp_get_num_threads`, and in the second region all of them do the printing why will this gives the correct output? Because openMP follow the fork join model remember. So, what is going to happen this particular point over here where this parallel region is ending turns out to be a point of synchronization right?

So, I will have a single thread executing till this point, in this parallel region again three threads are going to get spawned and then in this point I am going to wait for all the threads to complete, and then this single thread is going to carry on only after all the other thread have joined. And then for this parallel region again it is going to spawn 4 threads right and so on.

Student: You should have a private tid (Refer Time: 14:06).

You should have a private tid well, here I do not need private because it is declared inside the parallel region. Look I only need to say whether a variable is shared or private if it has been declared outside. So, I need to tell you that you know I want this to be private otherwise openMP will treat it as shared right, but if I am declaring it inside it is obvious that it has to be private.

Student: If a variable is declared outside and it is private (Refer Time: 14:36) what is the value of the variable (Refer Time: 14:40).

You should not assume anything about that right.

So, you should be careful to write your programs in a way that you are not dependent on that right. So, what happens when I run this program?

(Refer Slide Time: 14:53)

```
/* Use #pragma parallel directive (join) as synchronization mechanism */
...
int numt ;
#pragma omp parallel default( shared )
{
    int j, tid = omp_get_thread_num() ;

    if ( tid == 0 ) {
        for( j = 0 ; j < 10000000 ; j++ ) ;
        numt = omp_get_num_threads() ;
    }
}

#pragma omp parallel default( shared )
{
    int tid = omp_get_thread_num() ;

    printf( "Hello World from thread %d of %d.\n", tid, numt ) ;
}
...
Hello World from thread 2 of 4.
Hello World from thread 0 of 4.
Hello World from thread 1 of 4.
Hello World from thread 3 of 4.
Hello World from thread 1 of 4.
Hello World from thread 0 of 4.
Hello World from thread 3 of 4.
Hello World from thread 2 of 4.
```

So, this is what I get right and I run it again seems fine to me I have a wait here. So, I am reasonably confident that you know I do not have a race condition there could be from other race condition, but this program does not have any race condition, why is this solution not elegant?

Student: (Refer Time: 15:16).

There are overheads in launching threads doing splices and joint is expensive right I am unnecessarily doing that what else? I am also making calls to integrate thread num again right which is unnecessary what was I trying to do? I have trying to save a call to omp get num threads and instead I am calling omp get thread num twice.

Now, from every thread right seems like a. So, definitely not an elegant solution it works it serves the purpose it does what I wanted it to do, but it is not really a great piece of code, but the whole purpose of doing all this is to expose you to some of these concepts right that is why we are writing this kind of code.