

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

Lecture – 06
Distributing Work in OpenMP

(Refer Slide Time: 00:27)

Computing the sum of an array

So, let us try to compute the sum of the elements in an array right something very simple.

(Refer Slide Time: 00:34)

```
/* Getting time and resolution */
```

```
#include <omp.h>  
#include <stdio.h>
```

```
int main( int *argc, char *argv[] )
```

```
{
```

```
printf( "omp_get_wtime()=%g\n", omp_get_wtime() );
```

```
printf( "omp_get_wtick()=%g\n", omp_get_wtick() );
```

```
return 0 ;
```

```
}
```

```
omp_get_wtime()=6.98887e+06  
omp_get_wtick()=1e-09
```

time in seconds

1 nanosecond


All right. So, the first thing you should know if you are trying to optimize code or gauge the performance of different codes, you need to be familiar with some functions. OMP get_wtick tells you the resolution of a timer in seconds right this means that the resolution of this timer is one nanosecond. Resolution is important right because if you are timing a piece of code and let us say you using a timer which is one second resolution, and your code runs in 30 nanoseconds right, your timer is going to say one second, it took one second to execute that piece of code or 0 seconds.

Because you make a call to the timer, you execute the code you make another call to the time right and then you calculate the difference that tells you how long it took to execute that piece of code. Ideally the resolution of your timer should be considerable smaller than the time it takes to execute that code. It otherwise you are not going to get accurate results. So, what omp_get_wtime returns means the time in seconds starting from some universal time right some number of years ago right you will typically see very large values.

(Refer Slide Time: 01:46)

```
/* Compute the sum of elements in an array - Sequential code */
#include <omp.h>
#include <stdio.h>
#define ARR_SIZE 1000000000
int a[ARR_SIZE];

int main( int *argc, char *argv[] )
{
    int i, sum = 0;
    double t1, t2;
    /* Initialize the array */
    for( i = 0 ; i < ARR_SIZE ; i++ )
        a[i] = 1;
    t1 = omp_get_wtime();
    /* Sum up the array */
    for( i = 0 ; i < ARR_SIZE ; i++ )
        sum += a[i];
    t2 = omp_get_wtime();
    printf( "Sum of the array elements = %d. Time=%g\n", sum, t2-t1 );
    return 0;
}
```



Now, what we are going to do is we are going to start with the sequential code and then slowly we are going to see how we can parallelize it. So, here is what I have? I have a variable a, which has one billion integers right and this is the array for which I want to calculate the sum.

So, how do I do that? First I am going to initialize the array this is simple I just create a loop and I am going to initialize them with once, and then I simply add up the elements of the array how do I do that? I keep a variable sum initialize it to zero and within this loop I just say sum plus equal to ai right I mean those are simplest code that one can write. So, and I print the sum of the array elements all right and I have also want to figure out how long it takes to execute this for that I am going to use omp get wtime. So, I make a call to omp get wtime before I do the summation and after I do the summation I am leaving the initialization out of this right I am not timing the initialization and just timing that time it takes to do the actual addition the summing. So, that is also stored in t1 and t2 which are doubles and finally, I say the sum of the array elements is whatever percent d and this is the time it takes, and what is the time it is t2 minus t1 that is the way use timers.

(Refer Slide Time: 03:18)

```
/* Compute the sum of elements in an array - Sequential code */
#include <omp.h>
#include <stdio.h>
#define ARR_SIZE 1000000000
int a[ARR_SIZE];
int main( int *argc, char *argv[] )
{
    int i, sum = 0;
    double t1, t2;

    /* Initialize the array */
    for( i = 0; i < ARR_SIZE; i++ )
        a[i] = 1;

    t1 = omp_get_wtime();
    /* Sum up the array */
    for( i = 0; i < ARR_SIZE; i++ )
        sum += a[i];
    t2 = omp_get_wtime();

    printf( "Sum of the array elements = %d. Time=%g\n", sum, t2-t1 );
    return 0;
}
```

Sum of the array elements = 1000000000. Time=13.8598

So, I run this code and some system and it took 13.85 seconds to run all right.

(Refer Slide Time: 03:24)


```
/* Compute the sum of elements in an array - Parallel code */
...
int i, tid, numt, sum = 0 ;
double t1, t2 ;

for( i = 0 ; i < ARR_SIZE ; i++)
    a[i] = 1 ;

t1 = omp_get_wtime() ;
/* Do the sum in parallel */
#pragma omp parallel default( shared ) private( i, tid )
{
    tid = omp_get_thread_num() ;
    numt = omp_get_num_threads() ;
    for( i = 0 ; i < ARR_SIZE ; i++)
        sum += a[i] ;
}
t2 = omp_get_wtime() ;

printf( "Sum of the array elements = %d. Time=%g\n", sum, t2-t1 ) ;
...

```



Sequential Sum: 1000000000, Time: 13.8598

Sum of the array elements = 1373461697, Time=31.1103

So, now, let me write to paralyze this code. So, this is how a paralyze it, I do the sum in parallel for that I introduces this hash pragma omp parallel region, I keep I and tid is private and I say default everything else is shared right. So, there is a default key word which I can use to say what the default behavior is. I can default private and then what will happen is that everything that I do not explicitly scope that I do not explicitly say whether that private or shared will be treated as private.

So, in this case I have just said default shared and I and tid are private and then a I mean I have my usual tid numt initializations and then I execute this code for I equal to 0, I less than ARR size I plus plus sum plus equal to ai I run this code and this is what I see well I had a billion entries and this is one point three billion. So, I expected the sum to be one billion right the sum should have been one billion, but I see one point three billions value right. So, what is wrong here?

Student: There is a race on sum.

There is a race on sum. So, there are two issues even if there was a race on sum rate why I am saying a value greater than one billion some increment should have got an lost. So, I should have seen a value less than one billion there is some other fundamental problem with this code.

Student: (Refer Time: 05:15).

So, unless I specify otherwise right throughout and assuming that the number of threads is four that I have said the variable. So, that the number of threads is 4. So, what are the four threads doing, what is thread 0 doing? It is going from i equal to 0 to i one billion and trying to add of the elements to sum what is thread one doing.

Student: (Refer Time: 05:46).

It is also a going from 0 to billion billion minus 1 whatever right. So, all the threads are actually trying to add up entire array if I wanted to actually do this the value I should have seen is 4billion.

Student: (Refer Time: 06:00).

Why because there are 4 threads, but of course, there are two problems over here one is that I have not actually paralyze the code I am asking each thread to do exactly same thing, I am asking each thread to compute the sum from of the entire array from 0 to billion minus 1 right

So, that is one bug and the other bug is that I have a race condition right and we already discuss this race condition enough times that there is with this variable sum.

Student: (Refer Time: 06:25).

I am trying to say some plus equal to a i , sum is a shared variable right and all the threads are trying to update sum together doing an increment or addition if I want to do sum plus equal to a i , even though in c it looks like a single instruction it is broken down into a number of instructions that data is fetched into the register incremented and then put back into the memory right. All of them are trying to changes the value of sum together. So, there is a race condition over here and what is the time I see? 31 seconds 31.1 seconds what is the time I have seen earlier in the sequential code 13 seconds, why is this time one up? I mean I would I expected to take thirteen 0.85 seconds right because all of us doing the same work which that sequential thread was doing ok.

So, maybe there is a little bit of overhead in launching the threads the folks and joins, but not this much how can I go from 13 to 31 that is like sub second kind of overhead. This is a cache coherence comes in. These four processors they have their own cache and these variables particular the variable sum it is in the cache every time a thread updates

sum whatever the protocol be whether it be the invalidate or the update protocol for cache coherency it has overheads right. So, that cache coherency overhead is causing the increase in time.

So, look we studied the bare minimum when we talked about cache coherency or architecture we studied the bare minimum which is essential for you to know in order to write parallel programs you have to be aware of what cache coherency is we did not get in to the cache coherency protocols, but you have to be aware of the fact that cache coherency has its overheads. The moment you try to write parallel codes you are going to see some weird behavior which you are not expecting and you have to try to explain that.