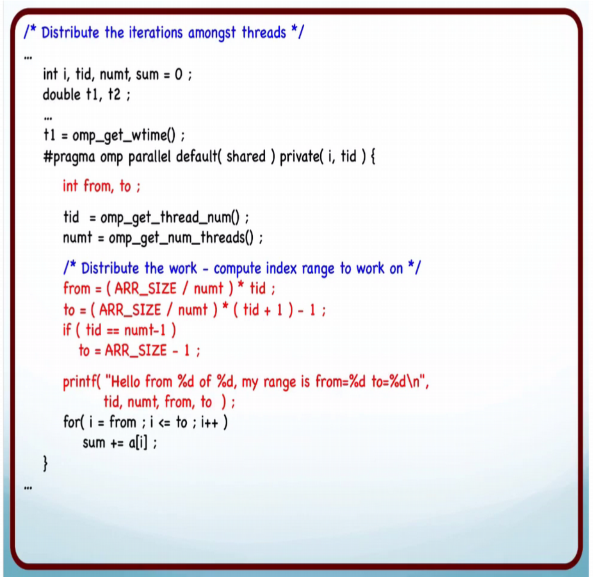
Introduction to Parallel Programming in OpenMP

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

Lecture – 22
Manual distribution of work and critical sections

So let me fix the code.

(Refer Slide Time: 00:05)



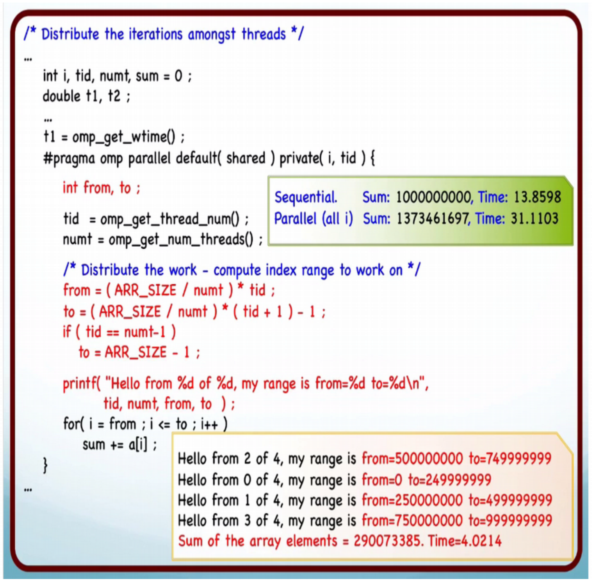
```
/* Distribute the iterations amongst threads */
...
   int i, tid, numt, sum = 0 ;
   double t1, t2 ;
   ...
   t1 = omp_get_wtime() ;
   #pragma omp parallel default( shared ) private( i, tid ) {
      int from, to ;

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

      /* Distribute the work - compute index range to work on */
      from = ( ARR_SIZE / numt ) * tid ;
      to = ( ARR_SIZE / numt ) * ( tid + 1 ) - 1 ;
      if ( tid == numt-1 )
         to = ARR_SIZE - 1 ;

      printf( "Hello from %d of %d, my range is from=%d to=%d\n",
            tid, numt, from, to  ) ;
      for( i = from ; i <= to ; i++ )
         sum += a[i] ;
   }
...
```

So, I am going to distribute the iteration amongst the threads. So, all the thread seem to be working on a entire array. So, let me have each thread work on different parts of the area. So, how do I do that? Well you can write this code on your own. So, you can have a look at it often; essentially all I am doing is that setting a from and a to a start and a end for every thread based on its thread id.

So, I divide the whole array size by the number of threads. So, that is like dividing one billion by 4 and then I allocate one region to each thread which ranges from a array size by num t into tid up to array size by num t into tid plus 1 minus 1. These are some intricate plus 1's and minus 1's; which I am sure you will be able to figure out on your own.

For the last thread; I have to be a little careful because the total space may not be equally divisible into four parts. So, I have to be careful about what I do with the last region; so, this is just a special case to handle that and then I have just printed this. This is again something you should be very very careful about; when you are trying to gauge the performance of your code; never have printf statements inside the time regions; they cause excessive overheads.

(Refer Slide Time: 01:21)



```
/* Distribute the iterations amongst threads */
...
   int i, tid, numt, sum = 0 ;
   double t1, t2 ;
   ...
   t1 = omp_get_wtime() ;
   #pragma omp parallel default( shared ) private( i, tid ) {
      int from, to ;

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

      /* Distribute the work - compute index range to work on */
      from = ( ARR_SIZE / numt ) * tid ;
      to = ( ARR_SIZE / numt ) * ( tid + 1 ) - 1 ;
      if ( tid == numt-1 )
         to = ARR_SIZE - 1 ;

      printf( "Hello from %d of %d, my range is from=%d to=%d\n",
              tid, numt, from, to  ) ;
      for( i = from ; i <= to ; i++ )
         sum += a[i] ;
   }
...
```

```
Sequential.      Sum: 1000000000, Time: 13.8598
Parallel (all i) Sum: 1373461697, Time: 31.1103
```

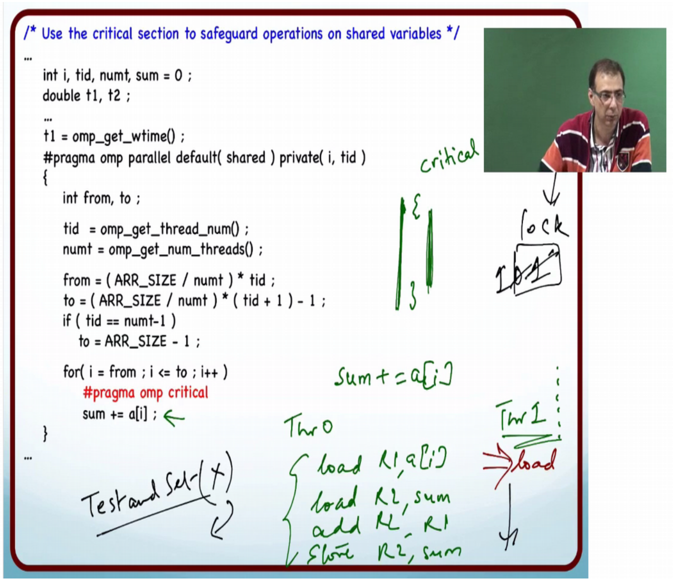
```
Hello from 2 of 4, my range is from=500000000 to=749999999
Hello from 0 of 4, my range is from=0 to=249999999
Hello from 1 of 4, my range is from=250000000 to=499999999
Hello from 3 of 4, my range is from=750000000 to=999999999
Sum of the array elements = 290073385. Time=4.0214
```

So, I run this code now and actually when I collected the time I did not execute the print statement; I did not find the print statement. So, that is just for convenience that I put this code together alright.

So, what do I see? Well now I see that thread 0 is actually working on the region from 0 to 200 and 50 million about that number and so on. So, if the array has been equally divided among the 4 threads and I get the sum of the elements as something and the total time taken is 4 seconds; it is at least I got the time down. So, one part of the code is I start to see some use of threads, but; obviously, I still have a problem; what is the problem? That the sum is not coming up correctly and we know why that is the case.

So, we need to fix that next; let us just compare this time we are getting. So, as we said right it is roughly about one fourth the time of the sequential code.

(Refer Slide Time: 02:24)



```
/* Use the critical section to safeguard operations on shared variables */
...
   int i, tid, numt, sum = 0 ;
   double t1, t2 ;
   ...
   t1 = omp_get_wtime() ;
   #pragma omp parallel default( shared ) private( i, tid )
   {
      int from, to ;

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

      from = ( ARR_SIZE / numt ) * tid ;
      to = ( ARR_SIZE / numt ) * ( tid + 1 ) - 1 ;
      if ( tid == numt-1 )
         to = ARR_SIZE - 1 ;

      for( i = from ; i <= to ; i++ )
         #pragma omp critical
         sum += a[i] ;
   }
...
```

How do I fix the other problem? So, there is a construct called hash pragma omp critical; what it does is, the region immediately after hash pragma omp critical, if we have multiple statements I can put it in curly braces like I have done before. If it is a single statement I can just write it without any curly braces and what hash pragma omp critical does is that it ensures that this particular region; the hash pragma omp critical region; it is called a critical section.

This region is the only executed by 1 thread at a time; 2 threads cannot being executing instructions inside this region at the same point in time. So, it ensures that this entire region is atomic; that no other thread can be executing these instructions while 1 thread is executing these instruction. That means, that when I am trying to do the sum plus equal to ai; if 1 thread; if thread 0 is executing the instructions load ai into R 1; does a load of sum into R 2 and then it says add R 2, R 1. So, add R 1 into R 2 and finally, store R 2 back into sum.

So, while 1 thread let us say thread 0 is executing these instructions thread 1 cannot execute any of these instructions it cannot enter this section. So, if thread 1 during its execution encounters this statement; obviously, this is the first statement due to the encounter, this particular load. It will not enter this region; it will not execute these instructions. So, there are ways of ensuring that right using locks and semaphores and other mechanisms. So, we will get into those later, but essentially the system can ensure for you that the other thread will not even reach this instruction; it will get blocked before this instruction; it will wait there.

So, I mean one simple way of doing this is that you keep a separate variable which is a lock and whenever any thread enters this region, it updates this lock to 1 saying that I have lock this region. And then if I knew the thread wants to start executing this region, it first checks this lock whether it is 1 or 0. If it finds it to be 1; it will keep waiting until it does not become 0 and thread 0; once it finishes execution it is going to reset this back to 0. So, if any thread is waiting; it will again check the value of lock, now it finds it to be 0; so, it is going to change it to 0 and start executing.

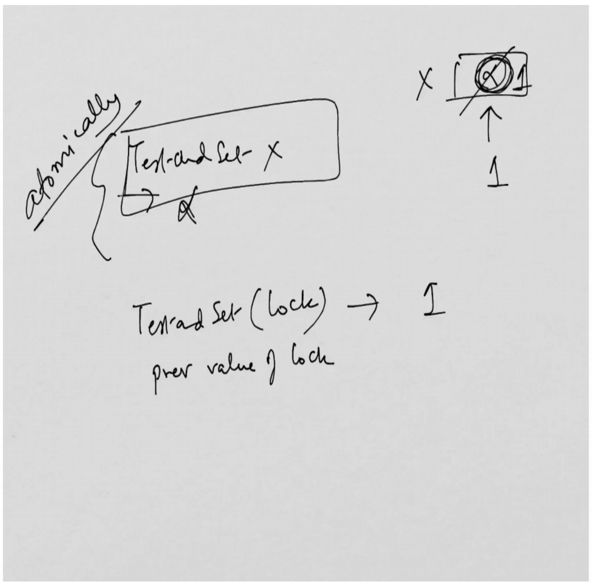
This one obvious problem you should see with this solution that I gave you for lock; what is that? But I have used lock as a mechanism to ensure that only 1 thread executes this region at a time, but what about lock itself; that is a shared variable and it is possible that 2 threads.

Student: (Refer Time: 05:40)

May updated together how can that happen? So, thread 0 tries to check whether 0 or not. So, thread 1 also tries to check whether 0 or not. So, both of them fetch into the register the value of this lock; both of them get 0, both of them generate to 1; put it back, the both thing they have the lock; both of them is proceed and execute the thread.

So, this is not the way you will observed and so this generally an instruction called test and set. So, it is a hardware instruction; so, what it does is that if you say test and set X; what it will do is it will fetch the value of X and simultaneously update it to 1; it will update it to 1 and fetch this value just before it is updated to 1 back for the programmer.

(Refer Slide Time: 06:36)



So, let us say I have a location X; so, what will test and set do. So, let us say the current value over here is some alpha; so, when I called test and set X; you mutely after test and set X its going to return me the value alpha and it is going to set this location to 1; it is going to replace alpha get 1; it is going to store the value 1 at location X. What does the programmer have? The programmer has the value of alpha; the value that was there just before I wrote 1 to this location and this is done atomically. So, how can you use this to implement a lock?

So, the way you use this is now that lock we were talking about; what we do is? You say test and set lock. What will this do this? This is set lock to one and it will return you the previous value of lock; now how do you know whether you have the lock or not?

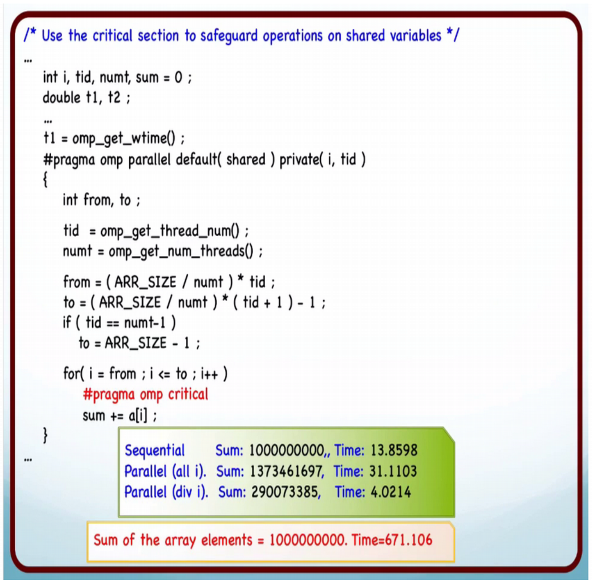
Student: If previous value (Refer Time: 07:50)

If the previous value was 0, then you have the lock because you just locked it. If the previous was 1; that means, somebody already had the lock and you have not done anything wrong by updating it to 1 because it is already 1. So, test and set is provided as an atomic instruction in most a hardware architectures, so that enables you to implement lock. So, we will not get deeper into test and set; it is just important for you to know that this instruction exist in most architecture and it can be used to implement locks.

So, using that I can implement this lock and I can implement critical sections. So, now I have ensured that the updation of sum happens in a critical sections; if 1 thread is executing the set of instructions which together are updating sum.

No other thread can at this point of time be executing and not be executing you cannot be updating sum; this is the only place where I am updating from. So, no other thread is touching sum at this point in time. So, I run this code.

(Refer Slide Time: 08:53)



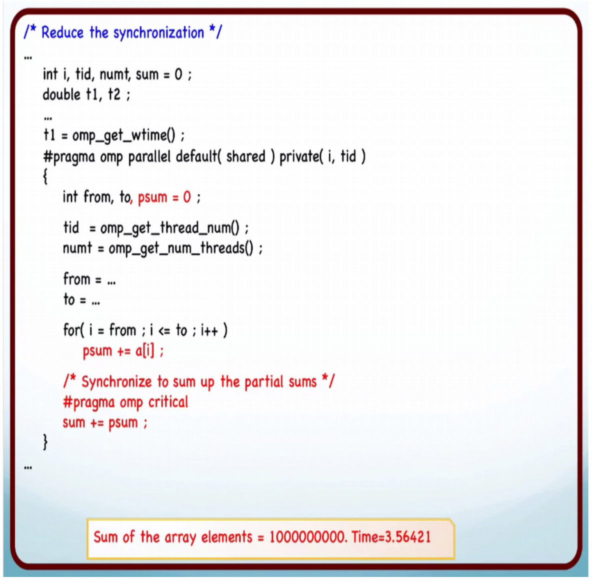
```
/* Use the critical section to safeguard operations on shared variables */
...
   int i, tid, numt, sum = 0 ;
   double t1, t2 ;
   ...
   t1 = omp_get_wtime() ;
   #pragma omp parallel default( shared ) private( i, tid )
   {
      int from, to ;

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

      from = ( ARR_SIZE / numt ) * tid ;
      to = ( ARR_SIZE / numt ) * ( tid + 1 ) - 1 ;
      if ( tid == numt-1 )
         to = ARR_SIZE - 1 ;

      for( i = from ; i <= to ; i++ )
         #pragma omp critical
         sum += a[i] ;
   }
...
```

```
Sequential      Sum: 1000000000,, Time: 13.8598
Parallel (all i).  Sum: 1373461697,  Time: 31.1103
Parallel (div i).  Sum: 290073385,   Time: 4.0214
```

```
Sum of the array elements = 1000000000. Time=671.106
```

I got the correct answer what happened to the time 671 seconds; you have to be very careful when you write parallel program. The code that the compiler introduces around a critical section where the critical section starts and where it ends is very very heavy alright. So, yeah this is just a comparison I thought I got it down to 4 seconds. So, when it is gone up to 671 seconds; when I got it correct.

(Refer Slide Time: 09:27)



```
/* Reduce the synchronization */
...
   int i, tid, numt, sum = 0 ;
   double t1, t2 ;
   ...
   t1 = omp_get_wtime() ;
   #pragma omp parallel default( shared ) private( i, tid )
   {
      int from, to, psum = 0 ;

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

      from = ...
      to = ...

      for( i = from ; i <= to ; i++ )
         psum += a[i] ;

      /* Synchronize to sum up the partial sums */
      #pragma omp critical
      sum += psum ;
   }
...
```

```
Sum of the array elements = 1000000000. Time=3.56421
```

How do I fix this? so this is how you do it why should they be updating the same. Sum is a commutative and associative operation; why cannot each thread compute the sum of its own part in a private variable and in the end I will update the shared variable and that is precise the what I am doing over here. So, there is a local variable psum; that is the private sum; I add ai to the private sum and I do not need to put that in omp critical; I do not need to put that in a critical section. There is no race condition over here, the threads are not accessing each others variables; no shared variable.

And then when I am done with this entire loop that I was supposed to work on; at the end just before exiting the parallel region, I have to synchronize to add up the partial sum. So, here I do hash pragma omp critical and I add psum to sum. How many calls was I making to critical section hash pragma omp critical? How many times those had been invoked in the earlier code?

Student: (Refer Time: 10:33)

1 billion times; how many time you had being invoked now?

Student: (Refer Time: 10:35)

4 times finally, I got the time down 3.56 second and my answer is correct. Finally, been able to write a parallel program which actually gives me benefit; when I compare it to all the other codes; I am doing quite well I have achieved what I wanted to do.