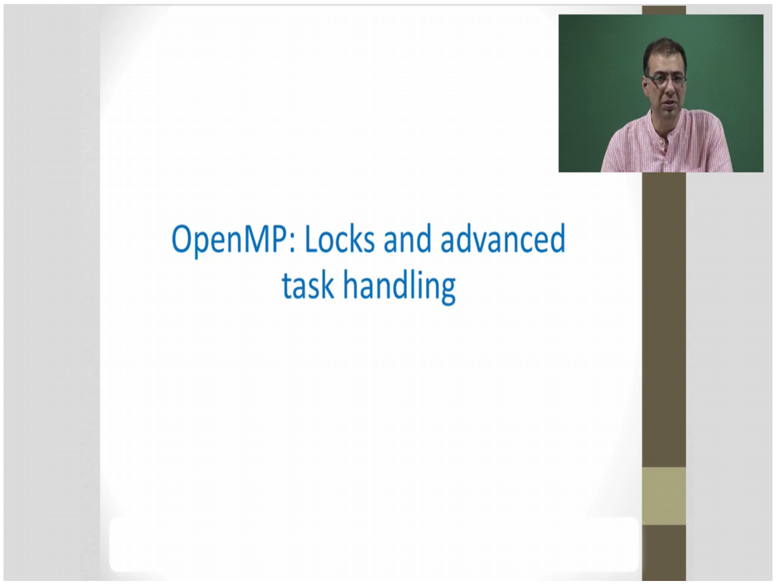
Introduction to Parallel Programming in OpenMP

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

Lecture – 10
OpenMP Locks and advanced task handling

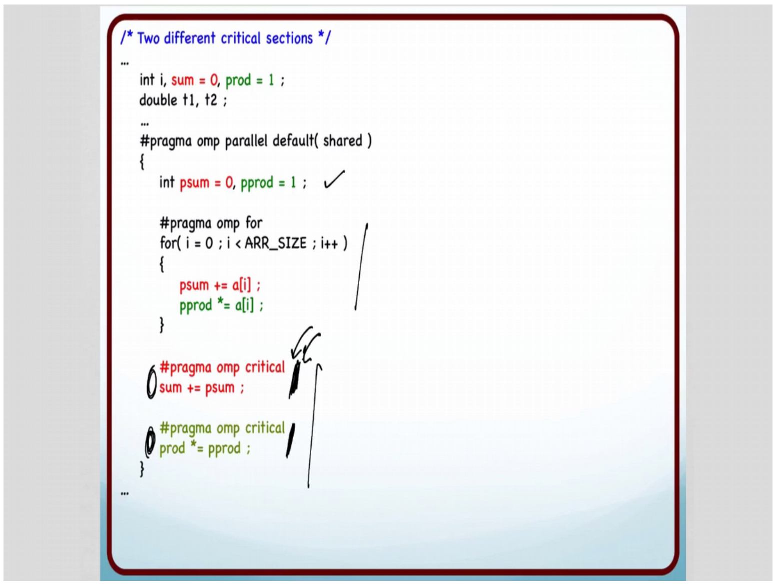
(Refer Slide Time: 00:26)



OpenMP: Locks and advanced task handling

So, today, we will look at locks in openMP and also talk about some advanced tasks handling.

(Refer Slide Time: 00:32)



```
/* Two different critical sections */
...
   int i, sum = 0, prod = 1 ;
   double t1, t2 ;
   ...
   #pragma omp parallel default( shared )
   {
      int psum = 0, pprod = 1 ;   ✓

      #pragma omp for
      for( i = 0 ; i < ARR_SIZE ; i++ )
      {
         psum += a[i] ;
         pprod *= a[i] ;
      }

      #pragma omp critical
      sum += psum ;

      #pragma omp critical
      prod *= pprod ;
   }
...
```

So, let us start with critical sections. So, we already discussed critical sections; right. So, this is an example that we had looked at some time back. So, what is this doing essentially its running a loop over an array and in this loop; it is trying to compute the sum of the elements as well as the product of the elements, right. So, sum and product are initialized to 1. So, this is partial sum and partial product, right.

So, we discussed that if we have multiple thread and I want to divide the work amongst these threads then instead of updating the same shared variable what I should do is I should keep a partial sum and a partial product, right a variable which is local which is private and once I have computed my part of the computation of the elements whatever operation needs to be done at the end I go and update the shared variable, right. So, I hope everybody remembers this.

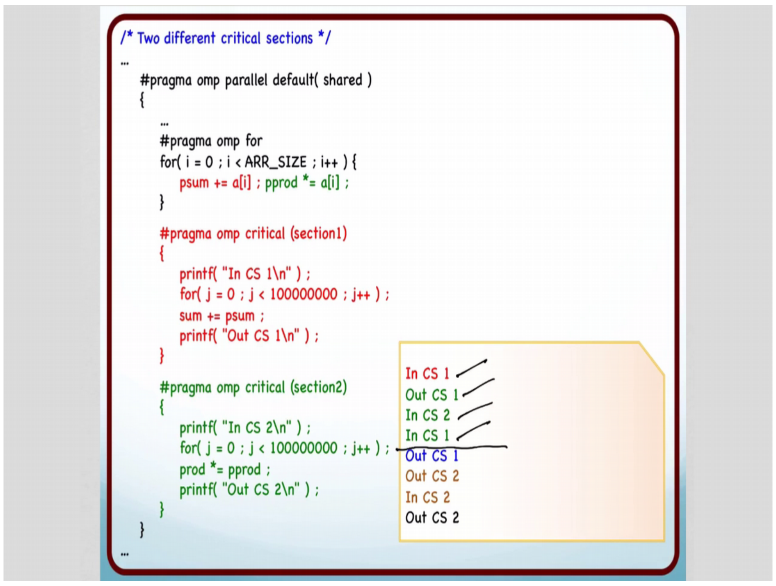
So, now in this code it does not matter. So, much, but there is one issue over here, right what is that at the end each thread is going to enter this critical section and update sum, right with the partial sum that it has and then it is going to go down and enter this critical section and update the product with the partial product that it has, right.

Now suppose that thread number 0 finishes first it comes here it updates sum and then it goes ahead and it starts updating the product it enters this critical section and at this point in time another thread; let us say thread one comes here and wants to update the sum, but it will not be able to enter the critical section what does critical section say critical section say is that only one thread is allowed to enter a critical section at any point in time.

But these 2 operations are completely unrelated right. So, if another thread was updating sum it would not matter to me right because I am updating product and another thread is updating sum. So, I do not care, it is not going to cause any race conditions, right what I want is that 2 threads should not simultaneously be updating sum or 2 threads should not be simultaneously updating prod, but its fine if one thread is updating sum and one thread is updating prod I do not want to you know serialize that.

So, critical section does not do that for you, right. So, there is an extension to critical sections in openMP.

(Refer Slide Time: 02:57)



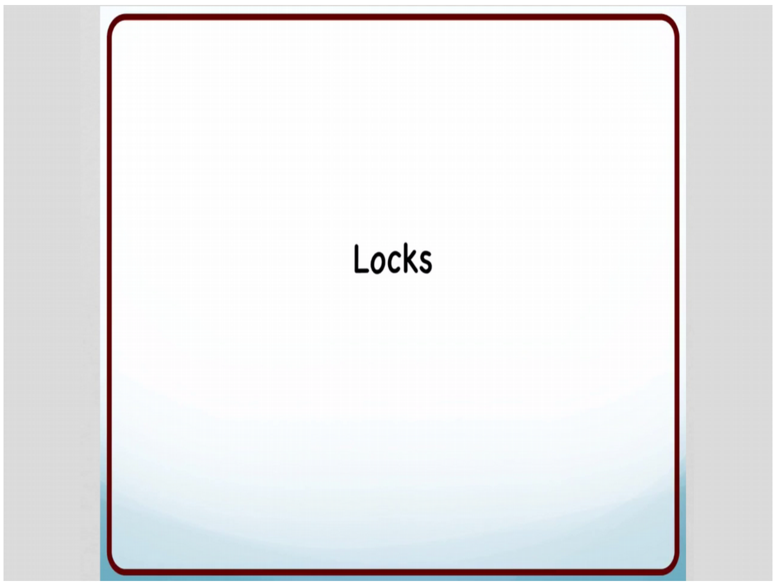
```
/* Two different critical sections */
...
   #pragma omp parallel default( shared )
   {
      ...
      #pragma omp for
      for( i = 0 ; i < ARR_SIZE ; i++ ) {
         psum += a[i] ; pprod *= a[i] ;
      }

      #pragma omp critical (section1)
      {
         printf( "In CS 1\n" ) ;
         for( j = 0 ; j < 100000000 ; j++ ) ;
         sum += psum ;
         printf( "Out CS 1\n" ) ;
      }

      #pragma omp critical (section2)
      {
         printf( "In CS 2\n" ) ;
         for( j = 0 ; j < 100000000 ; j++ ) ;
         prod *= pprod ;
         printf( "Out CS 2\n" ) ;
      }
   }
...
```

```
In CS 1
Out CS 1
In CS 2
In CS 1
Out CS 1
Out CS 2
In CS 2
Out CS 2
```

What you can do is you can give names to critical sections. So, you can say hash pragma omp critical and you can specify a name section one right and over here I say section two. So, then it knows that these are 2 different critical sections. So, what it will ensure is that no 2 threads enter the critical section one together and no 2 threads enter critical section 2 together, but at the same time you can have one thread in critical section one and one thread in critical section 2 and this is a small example to illustrate that whenever we enter a critical section and whenever we exit the critical section we have just printed it out, right and over here you can see some thread has entered critical section one over here and then it got out of critical section one, right.

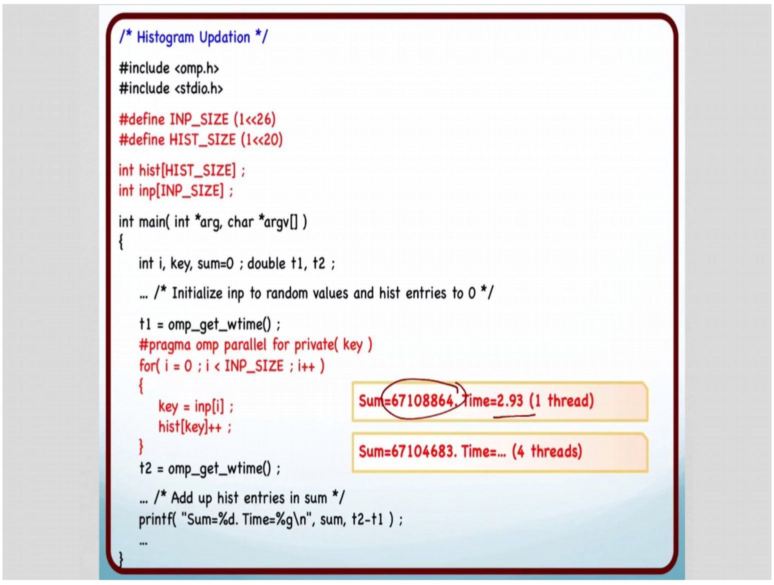
And then some thread entered critical section 2 and over here is some other thread entered critical section 1. So, at this point in time, there are 2 threads one in critical section 2 one in critical section 1. So, that is now possible.

(Refer Slide Time: 04:01)


Locks

So, if I have 2 critical sections, I can give them names, right if I have ten critical sections I can give them names what if I have critical sections which are dependent on the number of elements or the size of the data that I have right then I cannot give them static names I cannot say you know critical section one critical section 2 and so on; I cannot give them static names I do not know how many elements there will be at run time, right, maybe I am taking some input from the user what do I do in that case. So, in that case I need to resort to locks there are other advantages of locks also, but this is one differentiating thing from critical sections, right.

(Refer Slide Time: 04:36)


```
/* Histogram Updation */

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

#define INP_SIZE (1<<26)
#define HIST_SIZE (1<<20)

int hist[HIST_SIZE] ;
int inp[INP_SIZE] ;

int main( int *arg, char *argv[] )
{
    int i, key, sum=0 ; double t1, t2 ;

    ... /* Initialize inp to random values and hist entries to 0 */

    t1 = omp_get_wtime() ;
    #pragma omp parallel for private( key )
    for( i = 0 ; i < INP_SIZE ; i++ )
    {
        key = inp[i] ;
        hist[key]++ ;
    }
    t2 = omp_get_wtime() ;

    ... /* Add up hist entries in sum */
    printf( "Sum=%d. Time=%g\n", sum, t2-t1 ) ;
    ...
}
```

```
Sum=67108864. Time=2.93 (1 thread)
Sum=67104683. Time=... (4 threads)
```

So, what are locks? So, we are going to look at a simple example or histogram updation what is happening over here. So, there is an input and the size of this input is 2 to the power 26 and there is a histogram which has 2 to the power twenty elements I mean just think of it as that you have a huge document and you are trying to count the occurrence of words, right. So, you are trying to count how many a s appear how many does appear how many times the word study appears and so on, right.

So, think of this histogram as the words and input size as the document, right. So, there is a huge document and you want to count the occurrence of every word right how many times it appears in the document. So, in this case we are going to simplify it instead of having an actual document and words we are going to assume that these are all integers, right.

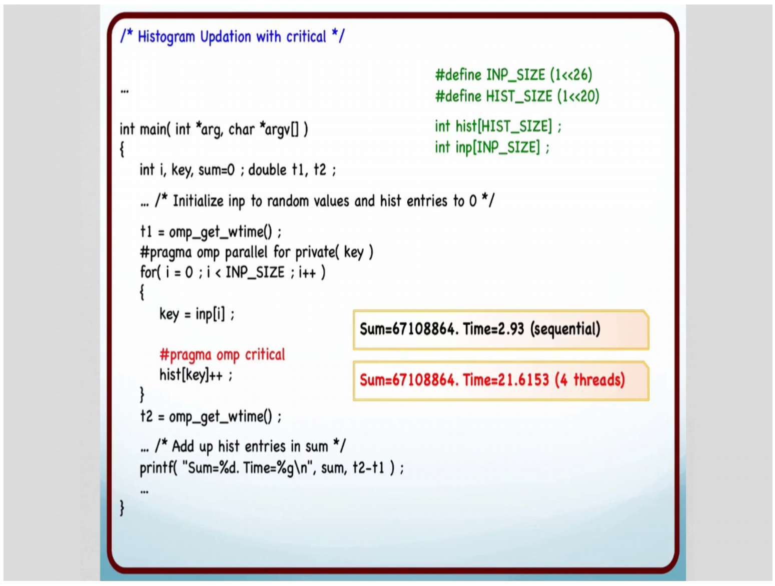
So, the input consists of 2 to the power 26 integers and the histogram that I want to build is of size 2 to the power 20, right. So, everybody understands what is in histogram, right, just counting the number of occurrences how do we do this. So, this is a simple program to do this. So, I start by initializing the inp the histogram to all 0s right. So, I have not written that piece of code and finally, I time this section where I actually go and do the updates and what are the updates that I am doing I say hash pragma omp parallel. So, this is the parallel region and so, I am paralyzing this for loop and key is a private variable right everything else by default is shared and if you remember if I have a hash pragma omp for right the variable of the for loop is automatically made private by openMP, right, openmp takes care of that; obviously, it cannot be shared.

So, key is private, I is private and everything else is shared. So, now, what do I do I just run through this entire input and I look at every element I extract the key and I go and update or increment that particular histogram entry. So, that is the code for histogram updation. So, what is the issue here do you see any issues; if 2 different threads read the same integer in the input they may be at different locations, but if that value is same then they will have the same value of key, right and then they will go and try to update the same histogram entry together and that will cause that race condition, right.

So, if I run this program. So, when I run it with a single thread then this is the output I get. So, I am also printing at the end of the day I go and add up all the entries the sum should be 2 to the power 20; in this case the sum is correct and the time it takes is 2 point nine three seconds with one thread, right and when I run with 4 threads it takes some time I do not care what the time is because my sum is incorrect and why is it incorrect we have already discussed that it does the race condition because these threads are trying to update the same entry.

Even though there are only 4 threads, right, but you can see that how many times the race condition has occurred right quite a few times.

(Refer Slide Time: 08:03)



```
/* Histogram Updation with critical */

                                          #define INP_SIZE (1<<26)
...                                       #define HIST_SIZE (1<<20)

int main( int *arg, char *argv[] )        int hist[HIST_SIZE] ;
{                                         int inp[INP_SIZE] ;
   int i, key, sum=0 ; double t1, t2 ;

   ... /* Initialize inp to random values and hist entries to 0 */

   t1 = omp_get_wtime() ;
   #pragma omp parallel for private( key )
   for( i = 0 ; i < INP_SIZE ; i++ )
   {
      key = inp[i] ;

      #pragma omp critical
      hist[key]++ ;
   }
   t2 = omp_get_wtime() ;

   ... /* Add up hist entries in sum */
   printf( "Sum=%d. Time=%g\n", sum, t2-t1 ) ;
   ...
}
```

```
Sum=67108864. Time=2.93 (sequential)
```

```
Sum=67108864. Time=21.6153 (4 threads)
```

So, what is the fix for that this is one fix that we have already seen, right put the update in the critical section can you do tricks like; what we did earlier with partial product partial sum can you do stuff like that here.

Student: (Refer Time: 08:23).

So, you can do that, but you have to replicate the entire histogram, right. So, in some applications that may be feasible in some applications that may not be feasible, but its not scalable right because as you increase the number of threads you are picking up more and more memory, right. So, it depends on how many threads you are using in the application. So, in order to do that you will have to maintain a separate copy of the histogram each thread will have to maintain a separate copy of the histogram and in the end you will have to merge all of them, right. So, that is going to have its own overheads. So, you have to be careful that you know whether you are going to get benefit out of it or not you have to take a lot of things into account. So, let us ignore that solution for the time being and let us look at this, right. So, this is one solution, I put critical here can I make use of the naming of critical sections that we just looked at yeah; how do I use it if I want to use it.

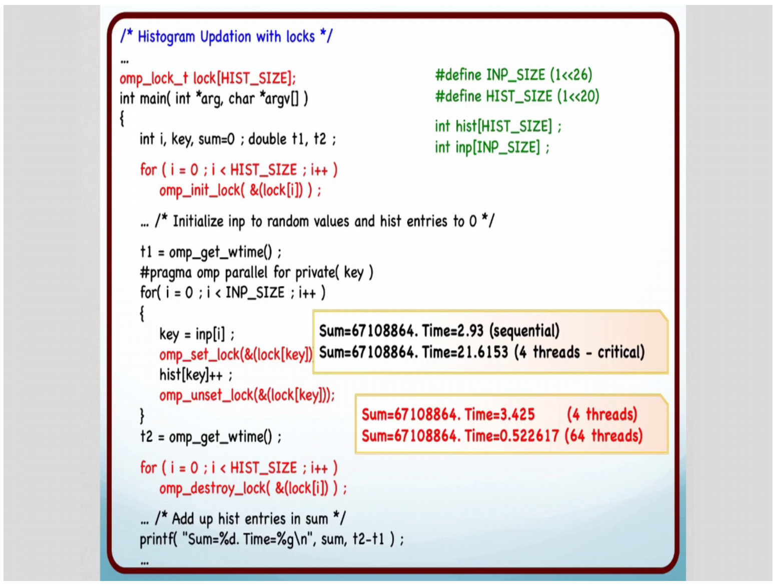
Student: (Refer Time: 09:24).

Now, you cannot use variables you have to give a name it is something that the compiler recognizes, right that this is critical section one critical section 2. So, key is a variable it is determined at run time. So, you cannot say critical bracket key it can only be a static name, right. So, it is not easy to use named critical sections over here. So, let us see what happens with critical. So, when I run this program with critical the sequential code took 2 point nine three seconds and with 4 threads with critical it takes twenty one point six one five three seconds, right.

Critical has overheads using critical sections is expensive here we are just doing a very very minor operation inside the critical region, right we are just doing one addition which is fine; if I had a substantial code and the critical region was small that would have been, but here my entire code is so small; right all I am doing is I am assigning something to key which hardly takes any time and then I am getting into a critical region to update the histogram.

So, most of the time in every loop is going to be taken entering the critical region exiting the critical region by openMP to ensure that only one thread is executing this part to implement that critical region code. So, this does not work we have seen the problems with critical before also, right that is where we had shifted to that partial sum partial product and all that. So, we know that there is a problem with critical. So, what do I do?

(Refer Slide Time: 10:57)



```
/* Histogram Updation with locks */
...
omp_lock_t lock[HIST_SIZE];
int main( int *arg, char *argv[] )
{
    int i, key, sum=0 ; double t1, t2 ;

    for ( i = 0 ; i < HIST_SIZE ; i++ )
        omp_init_lock( &(lock[i]) ) ;

    ... /* Initialize inp to random values and hist entries to 0 */

    t1 = omp_get_wtime() ;
    #pragma omp parallel for private( key )
    for( i = 0 ; i < INP_SIZE ; i++ )
    {
        key = inp[i] ;
        omp_set_lock(&(lock[key]));
        hist[key]++ ;
        omp_unset_lock(&(lock[key]));
    }
    t2 = omp_get_wtime() ;

    for ( i = 0 ; i < HIST_SIZE ; i++ )
        omp_destroy_lock( &(lock[i]) ) ;

    ... /* Add up hist entries in sum */
    printf( "Sum=%d. Time=%g\n", sum, t2-t1 ) ;
    ...
```

```
#define INP_SIZE (1<<26)
#define HIST_SIZE (1<<20)

int hist[HIST_SIZE] ;
int inp[INP_SIZE] ;
```

```
Sum=67108864. Time=2.93 (sequential)
Sum=67108864. Time=21.6153 (4 threads - critical)
```

```
Sum=67108864. Time=3.425      (4 threads)
Sum=67108864. Time=0.522617 (64 threads)
```

So, the solution to that is locks; how do we use locks; locks are variables. So, they are implemented using variables. So, you can declare an omp lock t structure like here what are we doing here we are allocating 2 to the power 20 locks as many as the number of histogram elements as the size of the histogram, right; how do I use locks. So, first I have to initialize the locks, I have to initialize every lock before using it. So, here I am initializing the locks. So, for that we basically use the omp init lock function and you pass the address of the lock, right.

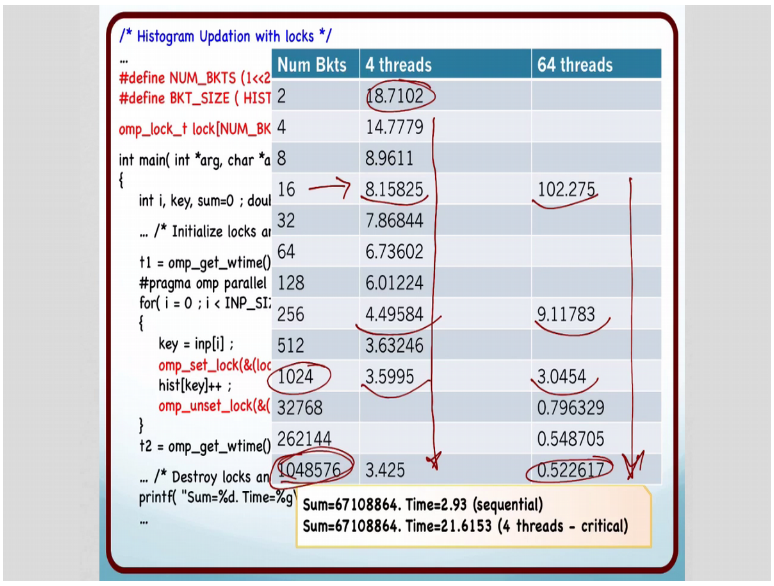
So, I have to use a for loop and initialize all the locks and how do I use the lock when I want the lock I called omp set lock function and I pass to it the address of the lock it must be initialized right and how do I release a lock I call omp unset lock same arguments. So, what does a lock ensure? So, what will happen if one thread gets the lock and then another thread tries to get that lock it is going to wait until that lock is not released by the first thread. So, it is a blocking call; what is a blocking call it means that it will not returned until; it does not get the lock this performs pretty much the same function as what we saw with that critical section with the name right critical section one critical section 2, right.

So, except that we are passing a particular lock. So, for every histogram entry I have a different lock this lock is indexed by the key. So, now, I have one lock for every histogram entry, right that is what I wanted which I could not do using critical sections once I am done with the lock I have to call omp destroy lock to release the lock. So, again for any particular lock it can only be acquired by 1 thread at a time. So, if we run this code; this is what I see. So, definitely I have got rid of the overheads of critical. So, with critical, it was taking about 21 seconds, but at least I have it down to like 3.4 seconds with 4 threads and if I increase the number of threads to 64; I start seeing some gains, right, with 4 threads; I am still not seeing gains; it is still worse than running a single thread. So, why is that happening?

Student: (Refer Time: 13:43) lock and locks (Refer Time: 1344).

Lock and lock has its overheads right the point here is that different threads are able to get their respective locks simultaneously critical section was not allowing different threads to enter that region of the code, right here different thread if they are working on different elements at least they can enter the code, right. So, you get some advantage out of that that was the whole point of using locks here instead of a critical section, but the degradation is because locks also have overheads right we have got gains eventually with 64 threads I have got better than what I had with a single thread, but it is not satisfactory I have got like factor 6 speed up, but for that I had to use 64 threads what we would like is linear speed up right that is the ideal scenario.

(Refer Slide Time: 14:32)



```
/* Histogram Updation with locks */
...
#define NUM_BKTS (1<<2
#define BKT_SIZE ( HIST

omp_lock_t lock[NUM_BK

int main( int *arg, char *a
{
    int i, key, sum=0 ; doul
    ... /* Initialize locks ar
    t1 = omp_get_wtime()
    #pragma omp parallel
    for( i = 0 ; i < INP_SI:
    {
        key = inp[i] ;
        omp_set_lock(&(loc
        hist[key]++ ;
        omp_unset_lock(&(
    }
    t2 = omp_get_wtime()
    ... /* Destroy locks an
    printf( "Sum=%d. Time=%g
    ...
```

Num Bkts	4 threads	64 threads
2	18.7102	
4	14.7779	
8	8.9611	
16	8.15825	102.275
32	7.86844	
64	6.73602	
128	6.01224	
256	4.49584	9.11783
512	3.63246	
1024	3.5995	3.0454
32768		0.796329
262144		0.548705
1048576	3.425	0.522617

```
Sum=67108864. Time=2.93 (sequential)
Sum=67108864. Time=21.6153 (4 threads - critical)
```

Another issue with these locks is that they take a lot of memory how many locks are we having we are having as many locks as the number of histogram entries there is another side effect of that I am accessing the histogram the histogram is very large I do not expect it to be in the cache, right, it cannot fit in the cache. So, that is not something I can take care of I mean I am accessing entries at random, right. So, how can I ensure that they will be in the cache, but the locks is something that I maybe could have kept in the cache, but in this case I cannot because the number of locks is also 2 to the power 20 which is huge.

So, because the number of locks is huge they are also going to get swapped out of the cache regularly. So, I cannot hope to benefit by the cache with the locks currently; what have I done. So, this is my histogram these are the entries of the histogram I associated a lock with the first entry a lock with the second entry a lock with the third entry and. So, on right I have a lock with each and every entry, but an alternative that I could have done is I could have just taken a block of entries and associated a lock with that what is the trade off the number of locks is reduced drastically, right depends on the number of entries I have in the bucket.

So, I am calling this a bucket right depending on the number of entries in the bucket the number of locks reduces right. So, I have lesser memory and if it is small enough then it might even fit in the cache look the histogram does not fit in the cache, but the locks may fit in the cache. So, what is the advantage the advantage is that my memory footprint has gone down maybe I can make use of the cache out of that and what is the disadvantage. So, if 2 threads are trying to update histogram entries which are in the same bucket then they are going to again contend for the same lock right and that is going to slow down the code right reduce parallelism. So, we run this code with 4 threads and 64 threads and this is the kind of pattern we see, right.

So, let us just observe a few things over here. So, when the number of buckets is very small right they are just 2 buckets then with 4 threads its taking 18.7 seconds. Why is that because there is a lot of contention, right there only 2 buckets most of the time these threads are wanting the same lock and as I increase the number of bucket this is coming down and that is expected as I increase the number of buckets there will be less and less contention, the same behavior is observed with 64 threads, it starts with the large number and then it comes down, right, this is also as expected, but let us have a look at when the number of buckets is 16, right, then what is the time taken by 4 threads; about 8 seconds and what is the time taken by 64 threads; 102 seconds, why because there are more threads which are contending for the same buckets, right, 64 threads with just 16 locks they are going to contend.

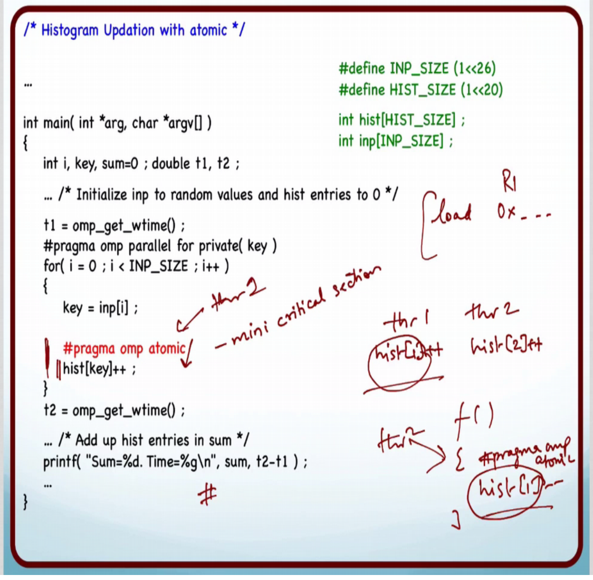
Student: (Refer Time: 17:44).

Right, whereas, when you have 4 threads and 16 locks that contention is less when you see for 256 buckets. So, this gap is becoming closer and when you look at 1024 buckets, then 64 threads has already started to do better.

Now because the number of buckets is so large that even 64 threads are able to work in parallel. So, the amount of parallelism that we are getting with 64 threads is substantial and eventually when you reach this large number here you know 64 threads is doing considerably better than 4 threads. So, this is just to give you an idea about how contention scales with increasing number of threads.

So, we see that with 64 threads with large enough number of buckets we do have good performance 0.5 seconds, but again as I say it, right, it is not linear speed up with 64 threads if the sequential code is taking like 3 seconds I was expecting much better, right, with 64 threads definitely sub second and maybe something like 0.2 seconds or something, right.

(Refer Slide Time: 18:51)



```
/* Histogram Updation with atomic */
                                                #define INP_SIZE (1<<26)
...                                             #define HIST_SIZE (1<<20)

int main( int *arg, char *argv[] )              int hist[HIST_SIZE] ;
{                                               int inp[INP_SIZE] ;
    int i, key, sum=0 ; double t1, t2 ;

    ... /* Initialize inp to random values and hist entries to 0 */

    t1 = omp_get_wtime() ;
    #pragma omp parallel for private( key )
    for( i = 0 ; i < INP_SIZE ; i++ )
    {
        key = inp[i] ;

        #pragma omp atomic
        hist[key]++ ;
    }
    t2 = omp_get_wtime() ;

    ... /* Add up hist entries in sum */
    printf( "Sum=%d. Time=%g\n", sum, t2-t1 ) ;
    ...
}
```

So, there is actually something called atomic in openMP, it is also called a mini critical section and this is not like a critical section for the following piece of code instead it is a critical section on the memory location.

And there is only a fixed set of operations that you can do, right, it is like memory update operations right increment addition and so on some limited set of operations that you can do. So, it says that the following memory update in the next instruction will be performed atomically for this memory location what; that means, is if there is a thread; thread 1 and a thread; thread 2, right and thread 1 comes to this instruction trying to update HIST 1 plus plus and thread 2 comes here and tries to update HIST 2 plus plus.

These are 2 different memory locations. So, even though the code is the same; so, that is where it differs from critical section it is not a critical section on the code. It is on the memory location, right. So, it says that it will allow both of these to happen in parallel, but if let us say thread 2 was somewhere else maybe not in this piece of code, but somewhere else maybe I called a function somewhere and inside that function thread; thread 2 was trying to do HIST 1 minus minus, right.

And this was inside hash pragma omp atomic. So, if thread 2 is executing this code and thread one is executing this code and both of them are trying to update HIST 1; it will not allow them to happen together this is ensuring that that particular memory location is not updated by 2 threads at the same time; that is what it does. So, you can read up the openMP manual on what all is supported under hash pragma omp atomic.

Student: During the function, this would be some other variables that addresses it.

That does not matter; the memory location is what matters; the address, it just figures out that whether the address being updated is the same or not; that is what matters, this is runtime support.

Student: If we have one variable in main memory and another in the function.

Right.

Student: So, with the same thing that treated as a different (Refer Time: 21:22). So, like there is a function that have a and in the main we have a variable a. So, this address of this a will be different that a;

Yeah.

Student: So, in that cases, if we are doing atomic operation. So, both the; both case are having different addresses. So, in that that case, it will not (Refer Time: 21:48) matter (Refer Time: 21:49)?

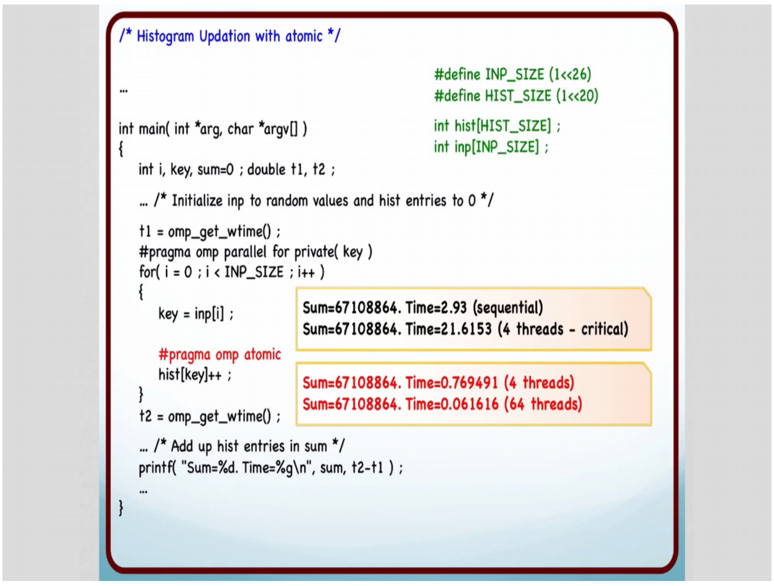
Yeah both of them can happen in parallel, it is the address of the variable being updated what is the variable HIST of one this is HIST of one first entry of the histogram right the address of the HIST one is the same. So, look at the end of the day, it does not matter who is doing what operation; what matters is that at the time this load is issued its going to be a set of instructions load store or some kind of an atomic instruction being executed there is going to be an address that is going to be passed to it, right.

0 x something which is stored in some register right all that matters is what is that address. If there is another hash pragma omp atomic already happening with that address, it is not going to allow this to happen until that is not completely, right. So, this requires support in the hardware to implement, this is runtime, this is not static.

Student: (Refer Time: 22:39) even if you use different variable name point to the same location it will (Refer Time: 22:42)

Yeah absolutely; so, we run this code and you see some very good results.

(Refer Slide Time: 22:50)



```
/* Histogram Updation with atomic */
                                              #define INP_SIZE (1<<26)
...                                           #define HIST_SIZE (1<<20)

int main( int *arg, char *argv[] )            int hist[HIST_SIZE] ;
{                                             int inp[INP_SIZE] ;
    int i, key, sum=0 ; double t1, t2 ;

    ... /* Initialize inp to random values and hist entries to 0 */

    t1 = omp_get_wtime() ;
    #pragma omp parallel for private( key )
    for( i = 0 ; i < INP_SIZE ; i++ )
    {
        key = inp[i] ;

        #pragma omp atomic
        hist[key]++ ;
    }
    t2 = omp_get_wtime() ;

    ... /* Add up hist entries in sum */
    printf( "Sum=%d. Time=%g\n", sum, t2-t1 ) ;
    ...
}
```

```
Sum=67108864. Time=2.93 (sequential)
Sum=67108864. Time=21.6153 (4 threads - critical)
```

```
Sum=67108864. Time=0.769491 (4 threads)
Sum=67108864. Time=0.061616 (64 threads)
```

With 4 threads, I am almost seeing a factor for speed up, it just a little bit less than factor 4 and with 64 threads it is not linear, but close to it, right, substantially good. Critical sections and locks have considerable overheads that is atomic is very very light, but the drawback of atomic is obviously, that it only applies to the following memory update instruction that is all it applies on, right, but most of the time that does the job for us typically we are doing some very simple operation, right. So, this is something you should keep in mind.

Student: Can atomic be multiline (Refer Time: 23:23)?

No, atomic is only for a memory location and that memory location is identified by the update in the next instruction address of the update in the next instruction. So, it is only a single instruction. So, there is a limited set of operations that you can do with atomic. So, you can find that in the openmp programmer's guide.