Synchronization
(OpenMP Synchronization)

Synchronization

- OMP_SINGLE
- OMP MASTER
- OMP CRITICAL
- OMP ATOMIC
- OMP BARRIER
- OpenMP programs use shared variables to communicate.
- We need to make sure these variables are not accessed at the same time by different threads (will cause race conditions, WHY?).
- OpenMP provides a number of directives for synchronization.

OMP_SINGLE

- Only one of the threads will execute the following block of code
  - **The rest will wait for it to complete**
  - Good for non-thread-safe regions of code (such as I/O)
  - Must be used in a parallel region
  - **Applicable to parallel for sections**

OMP_SINGLE

- It specifies that the enclosed code is to be **executed by only one thread in the team.**
- The thread chosen could vary from one run to another.
- Threads that are not executing in the SINGLE directive wait at the END SINGLE directive unless NOWAIT is specified.

```
#include<stdio.h>
#include<omp.h>
int main()
{ int i, sal1=0,sal2=0;
#pragma omp parallel shared(sal1,sal2)
{
for(i=0;i<5;i++)
{ Sal1=sal1+I;  }
Printf("Salary of the first company=%d", sal1);
for(i=0;i<5;i++)
{
Sal2=sal2+i
}
Printf("Salary of the first company=%d", sal2);
}
Return 0;
}
```

Threads – 4
Data Races

```
Salary First Company = 10        thread 1
 Salary second Company = 10      thread 1
Salary First Company = 30        thread 3
 Salary second Company = 20      thread 3
Salary First Company = 40        thread 2
 Salary second Company = 30      thread 2
Salary First Company = 20        thread 0
 Salary second Company = 40      thread 0
admin@ubuntu ~ $
```

```
#include<stdio.h>
#include<omp.h>
Int main()
{ Int I, sal1=0,sal2=0;
#pragma omp parallel reduction(:+sal1)
For(i=0;i<5;i++)
{
Sal1=sal1+i
}
Printf("Salary of the first company=%d", sal1);
#pragma omp parallel reduction(:+sal2)
For(i=0;i<5;i++)
{
Sal2=sal2+i
}
Printf("Salary of the first company=%d", sal2);
Return 0;
}
```

```
Salary First Company = 10          thread no 0
Salary second Company = 10         thread no 0
admin@ubuntu ~ $
```

- This solution is quite nice. **It doesn't have any data races** and it uses less lines of code to express the solution.
- A possible disadvantage of this version is that we **have to wait for computations of all salaries to complete before we are able to see the results.**

```
#pragma omp parallel shared(sal1,sal2)
{
#pragma omp parallel reduction(:+sal1)
For(i=0;i<5;i++)
{Sal1=sal1+i}
#pragma omp single
{  Printf("Salary of the first company=%d", sal1);  }
#pragma omp parallel reduction(:+sal2)
For(i=0;i<5;i++)
{
Sal2=sal2+i
}}
Printf("Salary of the first company=%d", sal2);
Return 0; }
```

```
admin@ubuntu ~ $ ./a.out
Salary First Company = 10       thread no 1
Salary second Company = 10      thread no 1
Salary second Company = 20      thread no 0
Salary second Company = 30      thread no 3
Salary second Company = 40      thread no 2
```

```
#pragma omp parallel shared(sal1,sal2)
{
#pragma omp parallel reduction(:+sal1)
For(i=0;i<5;i++)
{Sal1=sal1+i}
#pragma omp single
{
 Printf("Salary of the first company=%d", sal1);
}
#pragma omp parallel reduction(:+sal2)
For(i=0;i<5;i++)
{Sal2=sal2+I }
#pragma omp single
{
Printf("Salary of the first company=%d", sal2);
} }
Return 0;
}
```

OMP_SINGLE

Only one thread initializes the shared variable a

```
#pragma omp parallel shared(a,b) private(i)
{
   #pragma omp single
   {
      a = 10;
      printf("Single construct executed by thread %d\n",
             omp_get_thread_num());
   }
   /* A barrier is automatically inserted here */

   #pragma omp for
   for (i=0; i<n; i++)
       b[i] = a;

} /*-- End of parallel region --*/

   printf("After the parallel region:\n");
   for (i=0; i<n; i++)
       printf("b[%d] = %d\n",i,b[i]);
```

OMP Master

- **Specifies a region** that is **to be executed only by the master thread of the team.**
- **All other threads on the team skip this section** of code
- It is similar to the SINGLE construct.
- This Directive ensures that only the master threads executes instructions in the block.
- **There is no implicit barrier so other threads will not wait for master to finish.**
- The following block will be executed by the master thread
- **No synchronization involved**
- Applicable only to parallel sections

OMP Master

```
#include <omp.h>
#include <stdio.h>
 int main( )
{   int a[5], i;
#pragma omp parallel
{ // Perform some computation.
 #pragma omp for
 for (i = 0; i < 5; i++)
a[i] = i * i;
 // Print intermediate results.
 #pragma omp master
for (i = 0; i < 5; i++)
printf("a[%d] = %d\n", i, a[i]);
 // Wait.
#pragma omp barrier // Continue with the computation.
 #pragma omp for
for (i = 0; i < 5; i++) a[i] += i; } }
```

```
admin@ubuntu ~ $ ./a.out
a[3]= 9  Thread 2
a[4]= 16         Thread 2
a[0]= 0  Thread 2
a[1]= 1  Thread 2
a[2]= 4  Thread 2
a[0]=0   thread 1
a[1]=1   thread 1
a[2]=4   thread 1
a[3]=9   thread 1
a[4]=16  thread 1
a[0]=0   thread 1
a[1]=2   thread 1
a[2]=6   thread 1
a[3]=12  thread 1
a[4]=20  thread 1
admin@ubuntu ~ $
```

Difference

Single

Master

1.Wait

1.No wait

2.Separate thread for that region

2.Master thread

OMP CRITICAL

- **This Directive makes sure that only one thread can execute the code in the block.**
- **If another threads reaches the critical section it will wait untill the current thread finishes this critical section.**
- It provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously.
- An optional name can be given to a critical construct. Name must be global and unique
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name.
- Every thread will execute the critical block and they will synchronize at end of critical section

```
#include <omp.h>
#include <stdio.h>
 #include <stdlib.h>
 #define SIZE 10
 int main()
 { int i; int max; int a[SIZE];
 for (i = 0; i < SIZE; i++)
{ a[i] = rand();
 printf(“ a[%d] = %d \t thread no %d\n", i,
a[i]),omp_get_num_threads(); }
max = a[0];
#pragma omp parallel for
for (i = 1; i < SIZE; i++)  {
 if (a[i] > max)
{ #pragma omp critical
{ // compare a[i] and max again because max  could have been
changed by another thread after
 // the comparison outside the critical section
if (a[i] > max) max = a[i]; }
 } }
 printf("max = %d\t thread no = %d\n", max,
omp_get_num_threads()); }
```

Output

41 18467 6334 26500 19169 15724 11478

29358 26962 24464 max = 29358

OMP CRITICAL

#pragma omp critical [name]

- Standard critical section functionality
- **Critical sections are global in the program**
  - Can be used to protect a single resource in different functions
- Critical sections are identified by the name
  - **All the unnamed critical sections are mutually exclusive throughout the program**
  - All the critical sections having the same name are mutually exclusive between themselves

```
int x = 0;
#pragma omp parallel shared(x)
{ #pragma omp critical
```

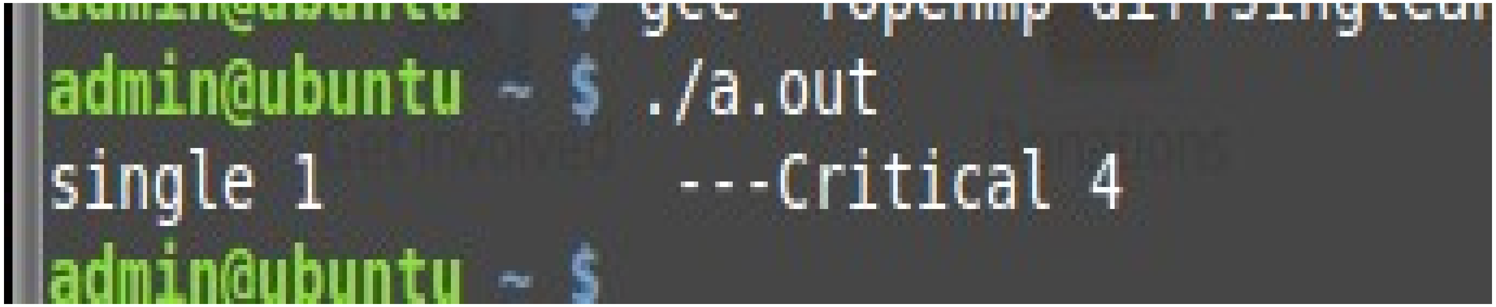
Difference between single and critical

- single specifies that a section of code should be executed **by single thread**(not necessarily the master thread)

- critical specifies that code is executed by one thread at a time

Example

```
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
#pragma omp single
a++;
#pragma omp critical
 b++;
 }
printf("single: %d -- critical: %d\n", a, b);
```


admin@ubuntu ~ $ ./a.out
single 1 ---Critical 4
admin@ubuntu ~ $

OMP ATOMIC

- #OMP ATOMIC
- This Directive is very similar to the #OMP CRITICAL
- Difference is that #OMP ATOMIC is only used for the update of a memory location.
- Specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it.
- Read,Write,Update,Capture

OMP ATOMIC

- The **omp atomic** directive allows access of a specific memory location atomically.
- It ensures that race conditions are avoided through direct control of concurrent threads that might read or write to or from the particular memory location.

```
for(i = 0; i<10; i++)
{
    #pragma omp atomic
    x[j[i]]= x[j[i]] + 1.0;
 }
```

Ordered

#pragma omp ordered

- It allows one to **execute a structured block within a parallel loop in sequential order.**
- The **code outside this block runs in parallel**.
- if threads finish out of order, there may be an additional performance penalty because some threads might have to wait.

Ordered

```
#pragma omp parallel for
for(i = 0; i < nproc; i++)
{
   do_lots_of_work(result[i]);

   #pragma omp ordered
   fprintf(fid,”%d %f\
n,”i,result[i]”);
   #pragma omp end
ordered
```

OMP BARRIER

Important restrictions:

- Each barrier must be encountered by all threads in a team, or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in the team.
- Without these rules some threads wait forever (or until somebody kills the process) for other threads to reach a barrier

OMP BARRIER

- **Synchronizes all threads in the team.**
- When a BARRIER directive is reached, **a thread will wait at that point until all other threads have reached that barrier**.
- All threads then resume executing in parallel the code that follows the barrier.

OMP BARRIER

- **barrier** synchronizes threads

```
#pragma omp parallel
private(myid,istart,iend)
myrange(myid,istart,iend);
for(i=istart; i<=iend; i++){
   a[i] = a[i] – b[i];
}
#pragma omp barrier
myval[myid] = a[istart] + a[0]
```

- Here **barrier** assures that a(1) or a[0] is available before computing **myval**

OMP BARRIER

- Performs a barrier synchronization between all the threads in a team at the given point.
- !$OMP BARRIER will **enforce every thread to wait at the barrier until all threads have reached the barrier.**
- !$OMP BARRIER is probably the most well known synchronization mechanism; **explicitly or implictly.**
- The following omp directives we discussed before include an implicit barrier:

REDUCTION

```
for ( j =0; j<N; j++) {
sum = sum+a [ j ] b [ j ] ;
}
```

- How to parallelize this code?
  - sum is not private, but accessing it atomically is too expensive
  - Have a private copy of sum in each thread, then add them up
- Use the reduction clause!

```
#pragma omp parallel for reduction(+: sum)
```

  - Any associative operator must be used: +, -, ||, |, *, etc.
  - The private value is initialized automatically (to 0, 1, ~0 . . .)

Exercise

- Develop an OpenMP C Parallel Program to perform the travelling salesman problem