Introduction to OpenMP

Lecture 5: Synchronisation

Why is it required?

Recall:

- Need to synchronise actions on shared variables.
- Need to ensure correct ordering of reads and writes.
- Need to protect updates to shared variables (not atomic by default)

BARRIER directive

BARRIER directive

- No thread can proceed past a barrier until all the other threads have arrived.
- Note that there is an implicit barrier at the end of DO/FOR, SECTIONS and SINGLE directives.

- Syntax:

Fortran: **`!$OMP BARRIER`**

C/C++: **`#pragma omp barrier`**

- Either all threads or none must encounter the barrier: otherwise DEADLOCK!!

A simple example

- `3 threads`
- `Everyone multiplies`

```
a[myid] = a[myid] * 3.5
```

- `Everyone sets:`

```
b[myid] = a[neighb] + c
```

BARRIER directive

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

```
!$OMP PARALLEL PRIVATE(I,MYID,NEIGHB)
   myid = omp_get_thread_num()
   neighb = myid - 1
   if (myid.eq.0) neighb = omp_get_num_threads()-1
   ...
   a(myid) = a(myid)*3.5
   b(myid) = a(neighb) + c
   ...
!$OMP END PARALLEL
```

A simple example

```
c=10
```

Original array **a**:

```
a[0]=30
a[1]=20
a[2]=10
```

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

Final array **a**:

```
a[0]=105
a[1]=70
a[2]=35
```

Final array **b**:

```
b[0]=45
b[1]=115
b[2]=80
```

BARRIER directive

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

	Enter parallel region	*get myid*	*calculate neighb*	*Get a[myid]*	*multiply by 3.5*	*assign result to a[myid]*	*get a[m*
Thread 0	✓	***myid=0***	*neighb=2*	*a[0]=30*	*105*	*a[0]=105*	*a[2]*
Thread 1	✓	***myid=1***	*neighb=0*	*a[1]=20*	*70*	*a[1]=70*	*a[0]*
Thread 2	✓	***myid=2***	*neighb=1*	*a[2]=10*	*35*	*a[2]=35*	*a[1]*

Start of parallel region

current

End of parallel region

BARRIER directive

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

	Enter parallel region	*get myid*	*calculate neighb*	*Get a[myid]*	*multiply by 3.5*	*assign result to a[myid]*	*get a[myid-1]*
Thread 0	✓	*myid=0*	*neighb=2*	*a[0]=30*	*105*	*a[0]=105*	*a[2]=35*
Thread 1 (stalling!)	✓	*myid=1*	*neighb=0*	*a[1]=20*	*70*	*a[1]=70*	*a[0]=105*
Thread 2	✓	*myid=2*	*neighb=1*	*a[2]=10*	*35*	*a[2]=35*	*a[1]=20*

Start of parallel region

current

End of parallel region

BARRIER directive

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

er allel ion	get myid	calculate neighb	Get a[myid]	multiply by 3.5	assign result to a[myid]	get a[myid-1]	add 10	assign b[myid
Thread 0	***d=0***	***neighb=2***	***a[0]=30***	***105***	***a[0]=105***	***a[2]=35***	*45*	*b[0]=4*
Thread 1	***d=1***	***neighb=0***	***a[1]=20***	***70***	*a[1]=70*	*a[0]=105*	*115*	*b[1]=11*
Thread 2	***d=2***	***neighb=1***	***a[2]=10***	***35***	***a[2]=35***	***a[1]=20***	*30*	*b[2]=3*

Start of parallel region

current

End of parallel region

BARRIER directive

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

	d	*calculate neighb*	*Get a[myid]*	*multiply by 3.5*	*assign result to a[myid]*	*get a[myid-1]*	*add 10*	*assign to b[myid]*
Thread 0 *(waiting for thread 1)*		*=2*	*a[0]=30*	*105*	*a[0]=105*	*a[2]=35*	*45*	*b[0]=45*
Thread 1		*=0*	*a[1]=20*	*70*	*a[1]=70*	*a[0]=105*	*115*	*b[1]=115*
Thread 2 *(waiting for thread 1)*		*=1*	*a[2]=10*	*35*	*a[2]=35*	*a[1]=20*	*30*	*b[2]=30*

Start of parallel region

current

End of parallel region

BARRIER directive

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

multiply by 3.5	assign result to a[myid]	get a[myid-1]	add 10	assign to b[myid]
Thread 0	**a[0]=105**	a[2]=35	45	**b[0]=45**
Thread 1	**a[1]=70**	a[0]=105	115	**b[1]=115**
Thread 2	**a[2]=35**	a[1]=20	30	**b[2]=30**

Final array `a`:

`a[0]=105`

`a[1]=70`

`a[2]=35`

Final array `b`:

`b[0]=45`

`b[1]=115`

`b[2]=`~~80~~ 30

Start of parallel region

current

End of parallel region

BARRIER directive

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

	calculate neighb	Get a[myid]	multiply by 3.5	assign result to a[myid]	get a[myid-1]	add 10	assign to b[myid]
Thread 0	*2*	*a[0]=30*	*105*	*a[0]=105*	*a[2]=35*	*45*	*b[0]=45*
Thread 1	*0*	*a[1]=20*	*70*	*a[1]=70*	*a[0]=105*	*115*	*b[1]=115*
Thread 2	*1*	*a[2]=10*	*35*	*a[2]=35*	*a[1]=20*	*30*	*b[2]=30*

Start of parallel region

current

End of parallel region

BARRIER directive

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

	get myid	calculate neighb	Get a[myid]	multiply by 3.5	assign result to a[myid]	Barrier	get a[myid-1]	add 10	assign to b[myid]
Thread 0		***neighb=2***	***a[0]=30***	***105***	***a[0]=105***		*a[2]=35*	*45*	*b[0]=45*
Thread 1		***neighb=0***	***a[1]=20***	*70*	*a[1]=70*		*a[0]=105*	*115*	*b[1]=115*
Thread 2		***neighb=1***	***a[2]=10***	***35***	***a[2]=35***		*a[1]=20*	*30*	*b[2]=30*

Start of parallel region

current

End of parallel region

BARRIER directive

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

	get myid	*calculate neighb*	*Get a[myid]*	*multiply by 3.5*	*assign result to a[myid]*	Barrier	*get a[myid-1]*	*add 10*	*assign to b[myid]*
Thread 0 *(waiting for thread 1)*		*neighb=2*	*a[0]=30*	*105*	*a[0]=105*		*a[2]=35*	*45*	*b[0]=45*
Thread 1		*neighb=0*	*a[1]=20*	*70*	*a[1]=70*		*a[0]=105*	*115*	*b[1]=115*
Thread 2 *(waiting for thread 1)*		*neighb=1*	*a[2]=10*	*35*	*a[2]=35*		*a[1]=20*	*30*	*b[2]=30*

Start of parallel region

current

End of parallel region

BARRIER directive

epcc

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

multiply by 3.5	*assign result to a[myid]*	Barrier	*get a[myid-1]*	*add 10*	*assign to b[myid]*
Thread 0	***a[0]=105***		*a[2]=35*	*45*	***b[0]=45***
Thread 1	***a[1]=70***		*a[0]=105*	*115*	***b[1]=115***
Thread 2	***a[2]=35***		*a[1]=70*	*80*	***b[2]=80***

Final array `a`:

```
a[0]=105
a[1]=70
a[2]=35
```

Final array `b`:

```
b[0]=45
b[1]=115
b[2]=80
```

Correct!

Start of parallel region

current

End of parallel region

BARRIER directive

A simple example

```
a[myid] = a[myid] * 3.5
b[myid] = a[neighb] + c
```

```
!$OMP PARALLEL PRIVATE(I,MYID,NEIGHB)
   myid = omp_get_thread_num()
   neighb = myid - 1
   if (myid.eq.0) neighb = omp_get_num_threads()-1
   ...
   a(myid) = a(myid)*3.5
!$OMP BARRIER
   b(myid) = a(neighb) + c
   ...
!$OMP END PARALLEL
```

Barrier required to force synchronisation on a

Critical sections

Critical sections

- A critical section is a block of code which can be executed by only one thread at a time.
- Can be used to protect updates to shared variables.
- The CRITICAL directive allows critical sections to be named.
- If one thread is in a critical section with a given name, no other thread may be in a critical section with the same name (though they can be in critical sections with other names).

CRITICAL directive

- Syntax:

Fortran: **`!$OMP CRITICAL`** *[(* *name* *)]*

block

`!$OMP END CRITICAL` *[(* *name* *)]*

C/C++: **`#pragma omp critical`** *[(* *name* *)]*

structured block

- In Fortran, the names on the directive pair must match.
- If the name is omitted, a null name is assumed (all unnamed critical sections effectively have the same null name).

CRITICAL directive (cont)

Example: pushing and popping a task stack

```
!$OMP PARALLEL SHARED(STACK),PRIVATE(INEXT,INEW)
      ...
!$OMP CRITICAL (STACKPROT)
      inext = getnext(stack)
!$OMP END CRITICAL (STACKPROT)
      call work(inext,inew)
!$OMP CRITICAL (STACKPROT)
      if (inew .gt. 0) call putnew(inew,stack)
!$OMP END CRITICAL (STACKPROT)
      ...
!$OMP END PARALLEL
```

Lock routines

Lock routines

- Occasionally we may require more flexibility than is provided by CRITICAL directive.
- A lock is a special variable that may be *set* by a thread. No other thread may *set* the lock until the thread which set the lock has *unset* it.
- Setting a lock can either be blocking or non-blocking.
- A lock must be initialised before it is used, and may be destroyed when it is not longer required.
- Lock variables should not be used for any other purpose.

Lock routines - syntax

Fortran:

```
USE OMP_LIB
SUBROUTINE OMP_INIT_LOCK(OMP_LOCK_KIND var)
SUBROUTINE OMP_SET_LOCK(OMP_LOCK_KIND var)
LOGICAL FUNCTION OMP_TEST_LOCK(OMP_LOCK_KIND var)
SUBROUTINE OMP_UNSET_LOCK(OMP_LOCK_KIND var)
SUBROUTINE OMP_DESTROY_LOCK(OMP_LOCK_KIND var)
```

var should be an INTEGER of the same size as addresses (e.g. INTEGER*8 on a 64-bit machine)

OMP_LIB defines OMP_LOCK_KIND

Lock routines - syntax

C/C++:

```
#include <omp.h>
  void omp_init_lock(omp_lock_t *lock);
  void omp_set_lock(omp_lock_t *lock);
  int omp_test_lock(omp_lock_t *lock);
  void omp_unset_lock(omp_lock_t *lock);
  void omp_destroy_lock(omp_lock_t *lock);
```

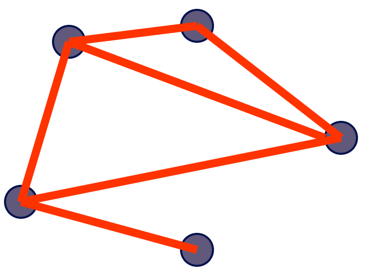
There are also nestable lock routines which allow the same thread to set a lock multiple times before unsetting it the same number of times.

Lock example

Example (compute degree of each vertex in a graph):

```
for (i=0; i<nvertexes; i++){
  omp_init_lock(lockvar[i]);
}

#pragma omp parallel for
      for (j=0; j<nedges; j++){
         omp_set_lock(lockvar[edge[j].vertex1]);
         degree[edge[j].vertex1]++;
         omp_unset_lock(lockvar[edge[j].vertex1]);
         omp_set_lock(lockvar[edge[j].vertex2]);
         degree[edge[j].vertex2]++;
         omp_unset_lock(lockvar[edge[j].vertex2]);
       }
```



Atomic directive

Atomic directive

- Used to protect an update to a single shared variable.
- Applies only to a single statement.

- May be more efficient than using CRITICAL directives (i.e. if different array elements can be protected separately.
- No interaction with CRITICAL directives.
- May be especially efficient if supported by hardware.

Atomic directive - Fortran syntax

```
!$OMP ATOMIC
        statement
```

where *statement* must have one of these forms:

x = x ***op*** ***expr***

x = ***expr op*** x

x = ***intr*** (x, ***expr***)

or

x = ***intr***(***expr***, x)

op is one of +, *, -, /, .and., .or., .eqv., or .neqv.

intr is one of MAX, MIN, IAND, IOR or IEOR

Atomic directive - C syntax

```
#pragma omp atomic
statement
```

where *`statement`* must have one of these forms:

x ***binop***= expr

or

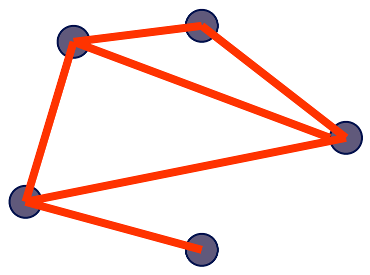
x++ | ++x | x-- | --x

binop is one of +, *, -, /, &, ^, <<, or >>

- Note that the evaluation of expr is not atomic!

Atomic directive - example

Example (compute degree of each vertex in a graph):



```
#pragma omp parallel for
      for (j=0; j<nedges; j++){
     #pragma omp atomic
        degree[edge[j].vertex1]++;

     #pragma omp atomic
        degree[edge[j].vertex2]++;

      }
```

Molecular dynamics

- The code supplied is a simple molecular dynamics simulation of the melting of solid argon.
- Computation is dominated by the calculation of force pairs in subroutine **`forces`**.
- Parallelise this routine using a DO/FOR directive and critical sections.
  - Watch out for PRIVATE and REDUCTION variables.
  - Choose a suitable loop schedule
- Extra exercise: can you improve the performance by using locks, or by using a reduction array (C programmers will need to implement this “by hand”).