



Shared Memory Programming with 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)

- 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!!

Example:

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



- Barrier required to force synchronisation on **a**

- 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).

- 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).

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
```

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

Fortran: **!\$OMP ATOMIC**

statement

where *statement* must have one of these forms:

$x = x \text{ op } \text{expr}$, $x = \text{expr op } x$, $x = \text{intr} (x, \text{expr})$ or

$x = \text{intr} (\text{expr}, x)$

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

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

C/C++: `#pragma omp atomic`
statement

where *statement* must have one of the forms:

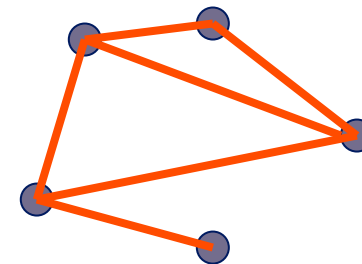
$x \text{ binop} = \text{expr}$, $x++$, $++x$, $x--$, or $--x$

and *binop* is one of $+$, $*$, $-$, $/$, $\&$, \wedge , \ll , or \gg

- Note that the evaluation of *expr* is not atomic.
- May be more efficient than using CRITICAL directives, e.g. if different array elements can be protected separately.
- No interaction with CRITICAL directives

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]++;
    }
```



- 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

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);
```

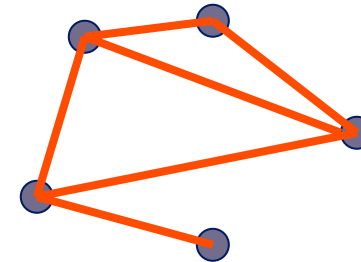
```
void omp_destroy_lock(omp_lock_t *lock);
```

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.

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]);  
    }
```



- 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 atomics, or by using a reduction array (C programmers will need to implement this “by hand”).