Shared Memory Programming with OpenMP

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

- 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!!

BARRIER directive (cont)

Example:

```
#pragma omp parallel private(myid,neighb) shared(a,b,c)
{
   myid = omp_get_thread_num();
   neighb = myid - 1;
   if (myid.eq.0) neighb = omp_get_num_threads()-1;
   ...
   a[myid] *= 3.5;
#pragma omp barrier
   b[myid] = a[neighb] + c;
   ...
}
```

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

BARRIER directive (cont)

Example:

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

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

Critical sections

- 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.

CRITICAL directive

- Syntax:

Fortran: **`!$OMP CRITICAL`**

block

`!$OMP END CRITICAL`

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

structured block

CRITICAL directive (cont)

Example: appending to a shared list

```
#pragma omp parallel for shared(list, N) private(newitem_p)
for (int i=0; i<N; i++) {
   newitem_p = createitem(i);
#pragma omp critical
   {
     append(&list,p_newitem);
   }
}
```

CRITICAL directive (cont)

Example: appending to a shared list

```
!$OMP PARALLEL DO SHARED(list,n) PRIVATE(newitem)
do i=1,n
   newitem = createitem(i)
!$OMP CRITICAL
   call append(list,newitem)
!$OMP END CRITICAL
end do
```

CRITICAL directive (cont)

Example: pushing and popping a task stack

```
#pragma omp parallel shared(stack) private(p_next,p_new,done)
{
while (!done) {
#pragma omp critical
  {
   p_next = pop(&stack);
  }
   p_new = process(p_next);
#pragma omp critical
  {
   if (p_new != NULL) push(p_new,&stack);
   done = isempty(&stack);
  }
  }
}
```

CRITICAL directive (cont)

Example: pushing and popping a task stack

```
!$OMP PARALLEL SHARED(stack),PRIVATE(next,new,done)
   do while (.not. done)
!$OMP CRITICAL
      next = pop(stack)
!$OMP END CRITICAL
      new = process(next)
!$OMP CRITICAL
      if (valid(new)) call push(new,stack)
      done = isempty(stack)
!$OMP END CRITICAL
    end do
!$OMP END PARALLEL
```

ATOMIC directive

- Used to protect a single update to a shared scalar variable of basic type.
- Applies only to a single statement.
- Syntax:

Fortran: **`!$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 (cont)

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

statement

where *statement* must have one of the forms:

x binop **=** *expr*, *x*++, ++*x*, *x*--, or --*x*

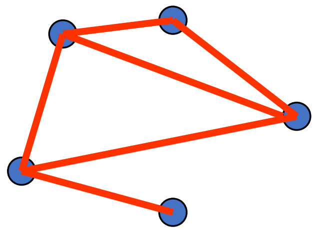
and *binop* is one of **+**, *****, **-**, **/**, **&**, **^**, **<<**, or **>>**

- 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

ATOMIC directive (cont)

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]++;
      }
```



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);
```

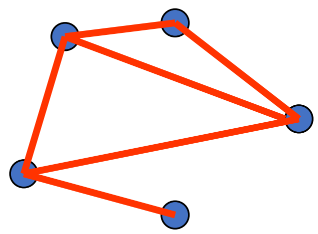
Lock example

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

```
omp_lock_t lockvar[nvertices];

for (i=0; i<nvertices; 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]);
       }
```



Exercise: 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 **`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.

Reusing this material