Chapter 21 Synchronization

Synchronization constructs: ways of telling threads to bring a certain order to the sequence in which they do things.

critical: a section of code can only be executed by one thread at a time.

atomic: update of a single memory location. Only certain specified syntax patterns are supported.

barrier, ordered, locks, flush, nowait

1. barrier

A barrier defines a point in the code where all active threads will stop until all threads have arrived at that point.

```
#pragma omp parallel
{
 int mytid = omp_get_thread_num ();
x [mytid] = some_calculation ();
# pragma omp barrier
 y[mytid] = x[mytid] +x[mytid +1];
}
```

At the end of a parallel region the team of threads is dissolved and only the master thread continues. Therefore, there is an implicit barrier at the end of a parallel region.

The barrier behavior at the end of the loop can be cancelled by the nowait clause. Like,

```
#pragma omp for nowait
```

2. Mutual exclusion

A critical section works by acquiring a lock, which carries a substantial overhead. Furthermore, if your code has multiple critical sections, they are all mutually exclusive: if a thread is in one critical section, the other ones are all blocked.

3. Locks

A critical section is indeed about code. With a lock you can make sure that some data elements can only be touched by one process at a time.

```
void omp_init_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
```

Chapter 22 Tasks

If you specify something as being parallel, OpenMP will create a ‘block of work’: a section of code plus the data environment in which it occurred. This block is set aside for execution at some later point.

```
p = head_of_list ();

while (!end_of _list (p)){

#process (p);

p = next_element (p);

}


#pragma omp parallel

#pragma omp single

{

#pragma omp task

{ … }

}
```

First a parallel region creates a team of threads and a single thread creates the tasks, adding them to a queue that belongs to the team, and all the threads in that team can execute the tasks.

With the task construct we can parallel the ‘while loop’.

1. Task data

Shared data is shared in the task, private data becomes first-private.

2. Task synchronization

In order to have a guarantee that a task is finished, you need the taskwait directive.

```
#pragma omp parallel

#pragma omp single

{

 while (! tail (p)) {

  p = p - > next ();

#pragma omp task
```

```
    process (p);
    }
#pragma omp taskwait
}
```

3. Task dependencies

Three ways: using the ‘task wait’; the task group directive, followed by a structured block, ensures completion of all tasks created in the block; each OpenMP task can have a depend clause, indicating what data dependency of the task.

```
#pragma omp task depend (out: x)
x = f ();
#pragma omp task depend (in: x)
y = g (x);
```

Read after Write, (out: x) (in: x)

Write after Read, (in: x) (out: x)

Write after Write, Read after Read.

4. More

Questions

1. How to specify one specific thread to do a calculation and the other to do another calculation?

2. What is the difference between the two code fragments on page 267?