OpenMP sync

OMP Synchronization

- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

critical

A critical section marks a mutual exclusion of a stmt/block

- inside a parallel region
- only 1 thread can execute the block
- any other arriving thread waits
  - no order here, either entering or waiting

warnings on critical

- there are warnings online that a critical section has "considerable overhead"
  - in any event, mutexs slow things down.
- if you don't say otherwise, every critical section shares the same underlying mutex
  - one blocked critical blocks every critical

critical naming

#pragma omp critical(my_name)

You can name your critical sections and they will then have a separate mutex.

Atomic

The phrase "atomic" refers to the original meaning of the word, which is "indivisible".

An atomic operation is guaranteed to not be "divisible" among threads.

- if one thread starts the operation, no other thread can interfere until the atomic operation finishes.

limited form of syn

As such, an atomic is usually a single operation on a variable, such as incrementing it or something similar.

Typically atomic operations are fast, much faster than a critical though limited in what they can do.

OMP atomic ops

- An atomic construct is limited to a single statement using:
  - +, *, -, /, &, ^, |, <<, >>

```
int ic, i, n;
ic = 0;
#pragma omp parallel shared(n,ic) private(i)
   for (i=0; i++, i<n)
    {
      #pragma omp atomic
         ic = ic + 1;
    }
```

“ic” is a counter. The atomic construct ensures that no updates are lost when multiple threads are updating a counter value.

```
int ic, i, n;
ic = 0;
#pragma omp parallel shared(n,ic) private(i)
   for (i=0; i++, i<n)
    {
      #pragma omp atomic
        ic = ic + bigfunc();
    }
```

- The atomic construct does not prevent multiple threads from executing the function bigfunc() at the same time.

barriers

Barriers are a little strange since implicit barriers are everywhere already, but you can place one in code and all threads have to reach the barrier before ***any*** thread can move on

nowait

- stop the implicit barrier at the end of a loop so that the work can move forward.

collapse

- Rules are that the loops have no work in them except in the bottom loop level (not intervening code).
- runs all the loop code as one loop

ordered

There is no guarantee in a for construct that the work is done "in order" of the for loop.

However, we can "force" a part of the code to come out in "iteration order"

single/master or if(id==0)

- single means only 1 thread will run the block ever
  - implicit barrier
- master means only the master will run the block ever
  - no barrier
- you can also check the thread number and have one thread do the work.

locks

Locks differ from critical sections in that you can lock ***data*** with a lock.

Locks are expensive to set up and use!

lock functions

- omp_init_lock(omp_lock_t *) – initialize a lock
- omp_set_lock(omp_lock_t*) – wait until the lock is available, then set it. No other thread can set the lock until it is released
- omp_unset_lock(omp_lock_t*) – unset (release) the lock
- omp_destroy_lock(omp_lock_t*) – The reverse of omp_init_lock