Concurrency : OpenMP

04/08/2021

Professor Amanda Bienz

How Does OpenMP Work?

- A number of threads execute a section of code concurrently
- User tells compiler how to parallelize code through pragma statements
- Race conditions, deadlocks, etc. still exist

Threads with OMP Parallel

- **#pragma omp parallel**
  {
  …
  }
  initializes threads and executes all statements between brackets concurrently

- **#pragma omp parallel for**
  for (…)
  {
  …
  }
  initializes threads and executes for loop concurrently, each thread executing a portion of the loops

Pragmas

- A pragma is a comment-like structure that tells the compiler additional information about the program
    - Processed when the program is compiled
- OMP pragmas tell the compiler:
    - How to parallelize the program
    - Important information about program variables (shared, private)

Helpful Hints

- **omp_get_num_threads()** returns the number of threads executing the block
- **omp_get_thread_num()** returns the id of the calling thread
- **OMP_NUM_THREADS** environment variable sets the number of threads for all #pragma omp parallel statements that do not specify thread count
- Code must be compiled with -fopenmp

Race Conditions

- A thread’s local variables can be either **shared** or **private**
- **Race conditions happen when multiple threads access a shared variable at the same time**

Using locks

- Around each race condition:
  omp_set_lock(&lock);
  omp_unset_lock(&lock);

- As with pthread locks, relatively expensive to lock/unlock and can serialize code (only one thread can execute at a time)

Critical Section

- **#pragma omp critical :** only one thread can execute following block of code at a time
- Still serializes code, but does not have the cost of locking and unlocking like using omp_set_lock

Atomic Operations

- Can read, write, and perform updates with only a single thread at a time
- As with pthreads, atomic operations can be performed at the hardware level. No locking necessary if hardware is utilized.

```
for(i = 0; i < 10; i++)
  {
    #pragma omp atomic read
    temp[i] = x[f(i)];

    #pragma omp atomic write
    x[i] = temp[i]*2;

    #pragma omp atomic update
    x[i] *= 2;
  }
```

Barrier

- **#pragma omp barrier** : no thread can move past the barrier until all threads reach the barrier

- Helpful for synchronization of programs, if one operation must complete across all threads before the next operation can begin

Implicit Barriers

- **#pragma omp for** : there is an implicit barrier at the end of the for loop… no thread can move on until all threads finish the for loop
- **#pragma omp for nowait** : says not to implement the implicit barrier

Nested Locks

- A lock cannot be locked if it is already locked, even if my thread locked it
- A nested lock can be locked multiple times by the same thread before it is unlocked

For Next Class

- We will start discussing I/O
- Read pages 489-500