Lecture 9
Announcements

• The midterm is on Tue Feb 9th in class
  ‣ Bring photo ID
  ‣ You may bring a single sheet of notebook sized paper “8x10 inches” with notes on both sides (A4 OK)
  ‣ You may not bring a magnifying glass or other reading aid unless authorized by me

• Review sessions
  ‣ In class on Thursday 2/4
  ‣ In section: Friday 2/5 (no section on Weds 2/3)

• Review questions have been posted
Today’s lecture

• OpenMP – a few loose ends
• Condition variables
Another way of annotating loops

• These are equivalent
• Why don’t we need to declare private(i)?

```c
#pragma omp parallel shared(a,b)
{
    #pragma omp for schedule(static)
    for (int i=1; i < N-1; i++)
        a[i] = (b[i+1] - b[i-1])/2h
}

#pragma omp parallel for shared(a,b) schedule(static)
for (int i=1; i < N-1; i++)
    a[i] = (b[i+1] - b[i-1])/2h
```
The No Wait clause

- Removes the barrier after an omp for loop
- Why are the results incorrect?
  - We don’t know when the threads finish
  - OpenMP doesn’t define the order that the loop iterations will be incorrect

```c
#pragma omp parallel
{
#pragma omp for nowait
  for (int i=1; i< N-1; i++)
    a[i] = (b[i+1] - b[i-1])/2h
#pragma omp for
  for (int i=N-2; i>0; i--)
    b[i] = (a[i+1] - a[i-1])/2h
}
```
Why isn’t a barrier needed between the calls to sweep()?

A. The calls to sweep occur outside parallel sections
B. 
C. OpenMP places a barrier after the \texttt{for i} loop inside Sweep
D. A & C

```cpp
for s = 1 to MaxIter do
  done = Sweep(Keys, N, 0);
  done &= Sweep(Keys, N, 1);
  if (done) break;
end do
int Sweep(int *Keys, int N, int OE) {
  bool done=true;
  #pragma omp parallel for shared(Keys) private(i) reduction(&:done)
  for i = OE; i to N−2 by 2
    if (Keys[i] > Keys[i+1]) {\textit{swap} Keys[i] ↔ Keys[i+1]; done &= false; }
  end do
return done;
```
Parallelizing a nested loop with OpenMP

• Not all implementations can parallelize inner loops

• We parallelize the outer loop index

```
#pragma omp parallel private(i) shared(n)
#pragma omp for
for(i=0; i < n; i++)
    for(j=0; j < n; j++) {
        V[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i, j+1] - h^2 f[i,j]) / 4
    }
```

• Generated code

```
mymin = 1 + ($TID * n/NT), mymax = mymin + n/NT-1
for(i=mymin; i < mymax; i++)
    for(j=0; j < n; j++)
        V[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i, j+1] - h^2 f[i,j]) / 4
Barrier();
```
An application: Matrix Vector Multiplication

\[ \begin{align*}
A & = \begin{bmatrix}
    a_1 & a_2 & a_3 & a_4 & a_5 \\
\end{bmatrix} \\
\end{align*} \]

\[ x = \begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
\end{bmatrix} \]

\[ y = \begin{bmatrix}
y \\
\end{bmatrix} \]
double **A, *x, *y;                              // GLOBAL
#pragma omp parallel shared(A,x,N)
#pragma omp for
for (i=0; i<N; i++){
    y[i] = 0.0;
    for (j=0; j<N; j++)
        y[i] += A[i][j] * x[j];
}
Support for load balancing in OpenMP

- **OpenMP** supports Block Cyclic decompositions with chunk size

```c
#pragma omp parallel for schedule(static, 2)
for ( int i = 0; i < n; i++ ) {
    for ( int j = 0; j < n; j++ ){
        do
            z = z^2 + c
        while (|z| < 2 )
    }
}
```
OpenMP supports self scheduling

• Adjust task granularity with a chunksize

```c
#pragma omp parallel for schedule(dynamic, 2)
for( int i = 0; i < n; i++ ) {
    for (int j = 0; j < n; j++) {
        do
            z = z^2 + c
        while (|z| < 2 )
    }
}
```
Iteration to thread mapping in OpenMP

```c
#pragma omp parallel shared(N, iters) private(i)
#pragma omp for
for (i = 0; i < N; i++)
    iters[i] = omp_get_thread_num();
```

N = 9, # of openMP threads = 3 (no schedule)
0 0 0 1 1 1 2 2 2

N = 16, # of openMP threads = 4, schedule(static,2)
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3

N=9: 0 0 1 1 2 2 0 0 1
Initializing Data in OpenMP

• We allocate heap storage outside a parallel region
• But we should initialize it inside a parallel region
• Important on NUMA systems, which account for most servers  http://goo.gl/ao02CO

double **A;
A =(double**) malloc(sizeof(double*)*N + sizeof(double)*N*N);
assert(A);

#pragma omp parallel private(j) shared(A,N)
for(j=0;j<N;j++)
    A[j] = (double *)(A+N) + j*N;

#pragma omp parallel private(i,j) shared(A,N)
for ( j=0; j<N; j++ )
    for ( i=0; i<N; i++ )
        A[i][j] = 1.0 / (double) (i+j−1);
OpenMP is also an API

- But we don’t use this lower level interface unless necessary
- Parallel *for* is much easier to use

```c
#ifdef _OPENMP
#include <omp.h>
#endif

int tid=0, nthrds,1;
#pragma omp parallel
{
#ifdef _OPENMP
    tid = omp_get_thread_num();
    nthrds = omp_get_num_threads();
#endif
    int i0=(n/nthrds)*tid, i1=i0+n/nthrds;
    for(i=i0; i < i1; i++)
        work(i);
}
```
Summary: what does OpenMP accomplish for us?

- Higher level interface simplifies the programmer’s model
- Spawn and join threads, “Outlining” code into a thread function
- Handles synchronization and partitioning
- If it does all this, why do you think we need to have a lower level threading interface?
Condition variables

- So far we’ve seen 3 synchronization primitives
  - Fork/Join
  - Mutex
  - Barrier
- But what if we want to communicate an event between threads?
  - Threads signal one another with a message: “the buffer is ready”
  - Wait for a certain amount of time to elapse: but what if we fell asleep during the alarm?
- We could *busy wait* on a variable protected by a critical section, but this can be wasteful
- C++ threads provides *condition variables*, a preferable way to handle the above situations
The interface for condition variables

- C++ provides condition variables
- We need a lock in order to use one

```c
#include <mutex>
#include <condition_variable>
```

- Two main entries
  - `Notify_one()` wakes up the thread waiting on the condition
  - `Wait()` will check for the desired condition within a critical section protected by the lock
    - If the condition has been met, `wait()` returns
    - Else, it unlocks the mutex and the thread enters a wait state
Thread safe queue

void Producer():
    while(more_to_produce()) {
        data_chunk const data=Produce();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }

void Consumer():
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[]{return !data_queue.empty();});
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();
        Consume(data);
        if(is_last_chunk(data)) break;
    }

#include <mutex>
#include <condition_variable>
#include <thread>
#include <queue>
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;
int main() {
    std::thread t1(Producer);
    std::thread t2(Consumer);
    t1.join();
    t2.join();
}

- Notify_one() wakes up the thread waiting on the condition
- If the condition has been met, wait() returns
- Else, it unlocks the mutex and the thread enters a wait state
The interface to `wait()`

```
template< class Predicate >
void wait( std::unique_lock<std::mutex>& lock,
Predicate pred );
```

void Consumer() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,
            []{return !data_queue.empty();});
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();
        Consume(data);
        if(is_last_chunk(data))
            break;
    }
}

- `Wait()` requires a lock
- But `lock_guard` has no way to explicitly lock and unlock
- `Unique_lock()` has this capability
- `Wait()` needs a *predicate function* to test the wait condition
  - The lambda function
    ```
    [](return data_queue.empty();)
    ```
  - Anonymous function
  - Avoids the need to define a new named function
  - Compiler infers return type based on knowledge of `empty()`
void Consumer() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();
        Consume(data);
        if(is_last_chunk(data))
            break;
    }
}
void Consumer() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,
        []{return !data_queue.empty();});
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();
        Consume(data);
        if(is_last_chunk(data))
            break;
    }
}
void Producer():
    while(more_to_produce()) {
        data_chunk const data = Produce();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }

void Consumer():
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data = data_queue.front();
        data_queue.pop();
        lk.unlock();
        Consume(data);
        if(is_last_chunk(data))
            break;
    }

Thread safe queue

Why do we unlock the lock before consuming the data?

A. The interface to mutex requires it
B. To reduce the time spent in the serial section
C. The interface to condition variables requires it
D. A & B
E. B & C
Why do we want the lock to be in the acquired state when wait() returns?

A. To protect the enqueuing operation
B. To prevent the Producer from continuing
C. Both A & B

- If the condition has been met, wait() returns
- Else, it unlocks the mutex and the thread enters a wait state
- When the waiting thread awakens
  - It reacquires the lock and checks the condition again…
  - .. If the condition has been met, wait() returns, with the lock in the acquired state
  - Else it unlocks the lock, waits again

```cpp
Consumer:
while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{
        return !data_queue.empty();
    });
    data_chunk data=data_queue.front();
    data_queue.pop();
    lk.unlock();
    Consume(data);
    if(is_last_chunk(data))
        break;
}
```
Why must wait() clear the lock before entering the wait state?

A. To prevent deadlock
B. So the Producer can continue queuing data
C. Both A & B

If the condition has been met, wait() returns
Else, it unlocks the mutex and the thread enters a wait state

When the waiting thread awakens
  It reacquires the lock and checks the condition again…
  .. If the condition has been met, wait() returns, with the lock in the acquired state
  Else it unlocks the lock, waits again

Consumer:
while(true) {
    std::unique_lock< std::mutex > lk(mut);
    data_cond.wait(lk, []{
        return !data_queue.empty();
    });
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    Consume(data);
    if (is_last_chunk(data))
        break;
}
The interface for Condition variables

- We need a lock in order to use a condition variable.
- `Wait()` will check for the desired condition within a critical section protected by the lock:
  - If the condition has been met, `wait()` returns.
  - Else, it unlocks the mutex and the thread enters a wait state.
- `Notify_one()` causes the thread waiting on the condition to awaken:
  - The thread reacquires the lock.
  - It checks the condition again…
  - .. If the condition has been met, `wait()` returns, with the lock in the acquired state.
  - Else it unlocks the lock, waits again.
Building a barrier with condition variables

- Cleaner design than with locks only
- `Wait(pred, lk)` is equivalent to `while(!pred()) wait(lk)`
- `unique_lock` objects constructed with `defer_lock` do not lock the `mutex` automatically at construction time, initially `unlocked`

```
barrier(int NT = 2) : ndone(0), _nt(NT) {};
void bsync() {
    unique_lock<mutex> lk(mtx, std::defer_lock);
    lk.lock();
    if (++ndone < _nt)
        cvar.wait(lk);
    else{
        ndone = 0;
        cvar.notify_all();
    }
}
```

```
Barrier(int NT=2): arrival(UNLOCKED),
    departure(LOCKED), count(0), _NT(NT) {};
void bsync() {
    arrival.lock();
    if (++count < NT) arrival.unlock();
    else departure.unlock();
    departure.lock();
    if (--count > 0) departure.unlock();
    else arrival.unlock();
}
```
Today’s lecture

• OpenMP – a few loose ends
• Condition variables
• Memory hierarchies
Recalling Bang’s Memory Hierarchy

- Intel “Clovertown” processor
- Intel Xeon E5355 (Introduced: 2006)
- Two “Woodcrest” dies (Core2) on a multichip module in 2 “sockets”
- * Intel 64 and IA-32 Architectures Optimization Reference Manual, Tab 2.16

Access latency, throughput (clocks)

- 3, 1
- 14*, 2
- 150/600

* Software-visible latency will vary depending on access patterns and other factors

Line Size = 64B (L1 and L2)

Associativity

- 8
- 16

Write update policy: Writeback

Sam Williams et al.

Scott B. Baden / CSE 160 / Wi '16
Sandy Bridge Memory Hierarchy

- Stampede system @ TACC (UT Austin)
- Intel E5-2680 0 @ 2.70GHz
  - 8 cores/socket x 2 sockets
  - Per core: L1 (32K) and L2 (256K)
  - Shared L3 (20MB)
- QPI connection between processors to build 16 core processor

www.qdpma.com/systemarchitecture/systemarchitecture_sandybridge.html

www.cac.cornell.edu/Stampede/CodeOptimization/multicore.aspx