Lecture 20

Final Exam Review
Announcements

• The Final Exam will be held in this room on Tuesday, March 16, 2010 3pm to 6pm
• Closed book, closed notes
• Office hours
  Monday Afternoon
  Or by appointment on Friday
• Cape!

CSE 160  28  6  21.43%  3/10/2010 1:44:48 PM
Terms and concepts

• Know the definition and significance of ….
• Granularity, surface-to-volume effect
• Data dependencies, loop carried dependence
• Load balancing, loop distribution
• SPMD, MIMD, SIMD
• Multiprocessors and multicomputers
• SIMT
Shared Memory Terms and Concepts

- Pthreads, Java threads and OpenMP
- Race conditions, non-determinacy, critical sections, atomicity
- Cache coherence and memory consistency
- Volatile storage
- The Java Memory Model
- The happens-before relationship
- Synchronized methods
- Semaphores, locks (mutexes) and barriers
- Atomic updates, CAS
- SMPs, snooping protocol, bus based coherence
- False sharing
Message passing

- Message startup, half power point $n^{1/2}$, peak bandwidth
- Point to point and collective communication
- Collectives: broadcast, all-to-all, irregular all-to-all ($v$), Barrier
- Message filtering, message ordering
- Deadlocks
- Non-blocking communication and overlap
- Eager limit, rendezvous
- $\alpha$-$\beta$ communication performance model
- Interconnection networks
- Hypercube, ring, mesh, k-ary d-cube
- Diameter and bisection bandwidth
Performance and performance programming

- Parallel speedup and efficiency, super-linear speedup, strong scaling, weak scaling
- Amdahl’s law, serial bottlenecks
- Load balancing:
  - blocked data decompositions, cyclic decomposition, self-scheduling, client/server computation
  - Irregular decomposition using recursive coordinate bisection
- Cache locality
Scalability

- We want performance to scale linearly with the number of processors
- Difficulties
  - Serial sections: code that runs on only one processor
  - “Non-productive” work associated with parallel execution, e.g. communication
  - Load imbalance: uneven work assignments over the processors
- Some algorithms present intrinsic barriers to scalability leading to alternatives
  
  \[
  \text{for } i=0:n-1 \quad \text{sum} = \text{sum} + x[i]
  \]
Serial section

- Limits scalability
- Let $f = \text{the fraction of } T_1 \text{ that runs serially}$
- $T_1 = f \times T_1 + (1-f) \times T_1$
- $T_P = f \times T_1 + (1-f) \times T_1 / P$
  Thus $S_P = 1/[f + (1 - f)/p]$
- As $P \to \infty$, $S_P \to 1/f$
- This is known as *Amdahl’s Law* (1967)
Technological Disruption
Technological disruption

• New capabilities
• Changes the common wisdom for solving a problem including the implementation
• “Today’s laptop would have been yesterday’s supercomputer”

Cray-1, 1976, 240 Megaflops

ASCI Red, 1997, 1Tflop

Intel Teraflop on a chip 2007

Nvidia Tesla, 4.14 Tflops
GPU processing

- nVidia: hierarchically organized clusters of streaming multiprocessors
- Small is mighty
- GTX-280 has 240 cores @ 1.296 GHz
- Delivers up to 933 Gflops/s
- SIMT parallelism, explicit data motion in the memory hierarchy
Performance programming issues

- Hide latency with many threads in flight
- Manage locality on chip: shared memory is faster than uncached global memory
- Tradeoff occupancy and locality
- Coalesce writes, avoid bank conflicts
- Minimize thread divergence under the SIMT
Memory consistency

• Recall the 3 conditions for consistency
  – Program order
  – Definition of a coherent view of memory
  – Serialization of writes

• Under a non-causal consistency model, it is possible for processor P2 to observe $A=1 \land B==1$ while processor P3 sees $B==1 \land A == 0$!
The Java Memory Model
The Java Memory model

- Minimal guarantees about semantics of memory access
- Bounds the potential effects of optimizations on execution semantics and discusses techniques for programmers to control some aspects of semantics
- Describes an abstract relation between threads and memory
- Guarantees concerning interaction between instruction sequences methods and fields in memory
Why do we need a memory model?

- Compiler, processors, or Memory system may rearrange order that writes are committed
- Memory cells might not get updated, if ever, after a later call to check()
- We would like to avoid synchronized methods

```java
final class SetCheck {
    private int a = 0; private long b = 0;
    void set() { a = 1; b = -1; }
    boolean check() { return ((b == 0) || (b == -1 && a == 1)); } 
}
```
Rules

- Most rules concern when values must be transferred between main memory and per-thread memory
- **Atomicity.** Which instructions must have indivisible effects. Only concerned with fields - instance and static variables, including array elements, but **not** local variables inside methods
- **Visibility.** Under what conditions the effects of one thread are visible to another. The effects of interest are: writes to fields, as seen via reads of those fields
- **Ordering.** Under what conditions the effects of operations can appear out of order to any given thread. In particular, reads and writes associated with sequences of assignment statements.
- Guarantees made are weaker than encountered in practice!
- Objects must maintain invariants as seen by all threads that rely on them, not just by the thread performing any given state modification
The effects of synchronization

• All changes made in one synchronized method or block are atomic and visible with respect to other synchronized methods and blocks employing the same lock,
• Blocks or synchronized methods within a given thread are processed in program-specified order
• Out of order processing cannot matter to other threads employing synchronization
• When synchronization is not used or is used inconsistently, answers are more complex
Atomicity

- Accesses and updates to the memory cells corresponding to fields of any type except long or double are guaranteed to be atomic.
- Extends to volatile long and double.
- Atomicity alone does not guarantee that you will get the value most recently written by any thread, thus atomicity guarantees per se normally have little impact on concurrent program design.
Visibility

• Changes to fields made by one thread are guaranteed to be visible, \textit{eventually}, to other threads only under certain conditions
  – releasing a lock
  – writing a volatile variable

voltile boolean ready = false;
Int answer = 0

All the memory contents seen by T1, before it wrote to ready, must be visible to T2, after it reads the value true for ready.
Visibility

- Changes to fields made by one thread are guaranteed to be visible to other threads only under the following conditions.
- A writing thread releases a synchronization lock and a reading thread subsequently acquires that same lock (applies to access to `SharedInteger`).
  - Releasing a lock flushes all writes from the thread’s working memory, acquiring a lock forces a (re)load of the values of accessible fields.
  - While lock actions provide exclusion only for the operations performed within a synchronized method or block, these memory effects are defined to cover all fields used by the thread performing the action.
- If a field is declared as `volatile`
  - Any value written to it is flushed and made visible by the writer thread before the writer thread performs any further memory operation.
  - Readers must reload the values of volatile fields upon each access.
- As a thread terminates, all written variables are flushed to main memory. Thus, if one thread synchronizes on the termination of another thread using `Thread.join`, then it is guaranteed to see the effects made by that thread.
“Eventually can be a long time”

- The memory model guarantees that, given the eventual occurrence of the above operations, a particular update to a particular field made by one thread will **eventually** be visible to another. But **eventually can be an arbitrarily long time**. Long stretches of code in threads that use no synchronization can be hopelessly out of synch with other threads with respect to values of fields. It is always wrong to write loops waiting for values written by other threads unless the fields are volatile or accessed via synchronization.
- Rules do not require visibility failures across threads, they merely allow these failures to occur.
- Not using synchronization in multithreaded code doesn't guarantee safety violations, it just allows them.
- Detectable visibility failures rarely occur on most current JVM implementations and platforms, even those employing multiple processors.
- Testing for freedom from visibility-based errors impractical, since such errors might occur extremely rarely, or only on platforms you do not have access to, or only on those that have not even been built yet.
Volatile

- In terms of atomicity, visibility, and ordering, declaring a field as volatile is nearly identical in effect to using a fully synchronized class protecting only that field via get/set method.
- Declaring a field as volatile differs only in that no locking is involved:
  - Operations such as "++" on volatile variables are not performed atomically.
  - Ordering and visibility effects surround only the single access or update to the volatile field itself.
  - Declaring an array field as volatile does not ensure visibility of its elements. Array elements themselves cannot be declared as volatile.
- Declaring fields as volatile is likely to be cheaper than using synchronization, except if accessed frequently inside methods.

```java
final class VFloat {
    private float value;
    final synchronized void set(float f) { value = f; }
    final synchronized float get() { return value; }
}
```
Communication

• Special mechanisms are needed to guarantee that communication happens between threads
• Memory writes made by one thread can become visible, but no guarantee.
• *Without explicit communication, you can't guarantee which writes get seen by other threads, or even the order in which they get seen*
• The Java `volatile` modifier constitutes a special mechanism to guarantee that communication happens between threads
• When one thread writes to a volatile variable, and another thread sees that write, the first thread is telling the second about all of the contents of memory up until it performed the write to that volatile variable

volatile boolean ready = false;
Int answer = 0

All the memory contents seen by T1, before it wrote to ready, must be visible to T2, after it reads the value true for ready.

Happens-before
“Happens-before relationship”

- Run on two separate threads, with counter = 0
  A: counter++;
  B: prints out counter
- Even if B occurs after A, no guarantee that B will see 0 …
- Unless we establish happens-before relationship between these two statements
- … a guarantee that memory writes by one specific statement are visible to another specific statement
- Different ways of accomplishing this: synchronization, volatile variables, thread creation and completion

http://java.sun.com/javase/7/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility
Synchronized methods

- When a synchronized method exits, it automatically establishes a happens-before relationship with what?

```java
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }
    public synchronized int value() { return c; }
}
```
Complications

private int foo;

public synchronized int getFoo() { return foo; }

public synchronized void setFoo(int f) { foo = f; }

• Is this thread-safe?
  
  setFoo(getFoo() + 1);
Atomic access

- Reads and writes are atomic for reference variables and for most primitive variables (all types except `long` and `double`)
- So are Reads and writes for all variables declared `volatile` (including `long` and `double` variables)
- Any write to a `volatile` variable establishes a *happens-before* relationship with subsequent reads of that same variable
- Simple atomic variable access is more efficient than accessing variables through synchronized code, but requires care to avoid memory consistency errors
- Also Atomic type from `java.util.concurrent.atomic`
java.util.concurrent.atomic

- Establishes happens-before relationship

```java
class Counter {
    private AtomicLong c = new AtomicLong(0);
    public long next() {
        return c.getAndIncrement();
    }
    public long getCount() { return count.get(); }
}
```
Compare and set

- Atomically set value to updated value if current value = expected value
- Returns false if actual value ≠ expected value, else true
- Let value = 0
  - value.compareAndSet(0, 100): value ← 100
  - value.compareAndSet(10, 100): no change

boolean compareAndSet(int expect, int update)
Building atomic counters

```java
public class AtomicCounter{
    private value;
    public int getValue() { return value; }
    public int increment() {
        int oldVal = value.getValue();
        while (!value.CAS(oldVal, oldVal + 1))
            oldVal = value.getValue();
        return oldVal + 1;
    }
}
```

value ← updated if current = expected
Application Programming
Algorithms

- Know the purpose of the following algorithms, and the significant implementation issues affecting performance.
- Be familiar with performance models for each and be prepared to analyze performance and scalability

**Sorting**
- Bucket sort,

**Stencil methods**
- Multidimensional cases: 2D and 3D
- “Curse of dimensionality:” surface to volume ratios
- Ghost cells, partitioning, performance models
- Dealing with Loop carried dependences

**Image processing**
- Image morphing
- Image segmentation
- Image blurring
Image smoothing algorithm

- Repeat as many times as needed

\[
\text{for } (i,j) \text{ in } 0:N-1 \times 0:N-1 \\
u'[i,j] = \frac{(u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i, j+1])}{4} \\
u = u'
\]
Parallel Implementation

- Partition data into parts, assigning each to a unique thread
- Dependences on values found on neighboring processes
- Threads share boundary values
False sharing in higher dimension arrays

• Compare with distributed memory solution

In global memory

Distributed memory

Parallel Computer Architecture, Culler, Singh, & Gupta
Barrier Synchronization

The following code performs barrier synchronization, where
$nt =$ number of threads

Explain how the code works, demonstrating correct operation
on 3 threads
Barrier Synchronization

Semaphore arrival(..), departure(..);
int Count;
Void barrier(...){
    arrival.aquire();
    Count++;
    If (Count < nt) arrival.release();
    Else departure.release()
    departure.aquire();
    Count--;
    If (count > 0) departure.release()
    Else arrival.release()
    Return;
}
OpenMP

List all possible outputs that result when the following OpenMP annotated C code is executed:

```c
#pragma omp parallel for shared(j,k)
for ( int i=0, j=0, k=0; i< 2; i++ )
    j = j + 10;
    k = j + 10;
}
cout << “k = “ << k << endl;
```
OpenMP

Now with critical sections (see `$pub/Midterm/openmp_ex.C`)

g++ -fopenmp openmp_ex.C

#pragma omp parallel for shared(j,k)
for ( int i=0, j=0, k=0; i< 2; i++ )
    #pragma omp critical
    j = j + 10;
    #pragma omp critical
    k = j + 10;
}
cout << “k = “ << k << endl;
Performance

- You observe the following running times for a parallel program running a fixed workload N
- What fraction of the total running time that runs on a single processor?
- What will the running time be on 4 processors?
- What is the speedup and efficiency on 8 processors?
- What is the maximum possible speedup on an infinite number of processors?

<table>
<thead>
<tr>
<th>NT</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
</tr>
<tr>
<td>2</td>
<td>6000</td>
</tr>
<tr>
<td>8</td>
<td>3000</td>
</tr>
</tbody>
</table>
Threads

In the following code a thread increments a shared variable until reaching a given maximum value.

class Q2 extends Thread {
    ...
    public void run() {
        synchronized(sharedVar) {
            while(sharedVar < MAX_VAL) {
                sharedVar++;
            }
        }
    }
}

• If we create two threads running object Q2, both try to update the shared object. What will happen to the shared variable?
• If we create and run 100 Q2 objects, what amount thread parallelism would be achieved? Explain.
• Will all of the Q2 objects get to update the shared variable? If one of them does not, we call that effect "starvation." If starvation is a possibility, suggest a way to avoid such a situation. Explain in detail why our approach would ensure fairness.
Consistency

Assume that memory is sequentially consistent
How many times can procedure foo() be run?

Assume that both X and Y are shared \texttt{int}, and have been initialized to zero

<table>
<thead>
<tr>
<th></th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 1;</td>
<td>Y = 1;</td>
<td></td>
</tr>
<tr>
<td>If (y==0)</td>
<td>If (x == 0)</td>
<td></td>
</tr>
<tr>
<td>foo();</td>
<td>foo();</td>
<td></td>
</tr>
</tbody>
</table>
Data parallelism

• Can we parallelize the inner loops as shown?

**LOOP #1**

for \( j = 0 \) to \( n-1 \)

for \( i = 0 \) to \( n-1 \)

\[
A[i, j+1] = A[i, j];
\]

for \( i = 0 \) to \( n-1 \)

\[
A[i, 1:n] = A[i,0:n-1];
\]

**LOOP #2**

for \( j = 0 \) to \( n-1 \)

for \( i = 0 \) to \( n-1 \)

\[
A[i, j+1] = A[i, j];
\]

for \( j = 0 \) to \( n-1 \)

\[
A[0:n-1, j+1] = A[0:n-1, j];
\]
Interconnection Networks

- Diameter, bisection bandwidth
- Map a 1D ring onto a hypercube, mesh, k-ary d-cube
- How many parallel/unique paths between any two nodes in a hypercube?
- Sort on a 2D network
- Sort on a hypercube