Lecture 7

Reduction and its applications
Synchronization in Java
Announcements

• Quiz return
Quiz

1. When Gustafson published the paper “Reevaluating Amdahl’s Law,” what was his message?

2. What is false sharing?

3. What is guided scheduling and how does it work?
Slight return to load balancing
Load imbalance in the Mandelbrot set computation

- Some points iterate longer than others
- If we use uniform decomposition, some threads finish later than others
- We have a load imbalance

\begin{align*}
\text{do} \quad & z_{k+1} = z_k^2 + c \\
\text{until} \quad & (|z_{k+1}| \geq 2)
\end{align*}
Performance under PJ

- Minimum times of 3 runs done on ieng6-203
- Fixed, single run: 94 580 3951 **7208** 7193 3942 580 93
- Dynamic(32): 1909 1907 1906 **1914** 1906 1906 1911 1912
- `$pub/examples/MandelbrotSetSmp4.java`

<table>
<thead>
<tr>
<th>NT F</th>
<th>D</th>
<th>D(8)</th>
<th>D(16)</th>
<th>D(32)</th>
<th>D(64)</th>
<th>G(8)</th>
<th>G(16)</th>
<th>G(32)</th>
<th>G(64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>123291</td>
<td>14966</td>
<td>14899</td>
<td>14896</td>
<td>14892</td>
<td>14977</td>
<td>23108</td>
<td>23111</td>
<td>19022</td>
<td>19021</td>
</tr>
<tr>
<td>211707</td>
<td>7502</td>
<td>7534</td>
<td>7494</td>
<td>7532</td>
<td>7518</td>
<td>10570</td>
<td>9814</td>
<td>9809</td>
<td>9808</td>
</tr>
<tr>
<td>411144</td>
<td>3818</td>
<td>3827</td>
<td>3803</td>
<td>3805</td>
<td>3822</td>
<td>4466</td>
<td>4535</td>
<td>4430</td>
<td>4440</td>
</tr>
<tr>
<td>8</td>
<td>7289</td>
<td>1997</td>
<td>2044</td>
<td>1999</td>
<td>1993</td>
<td>2864</td>
<td>2662</td>
<td>2504</td>
<td>2608</td>
</tr>
<tr>
<td>16</td>
<td>2906</td>
<td>1993</td>
<td>2082</td>
<td>2068</td>
<td>2104</td>
<td>2313</td>
<td>2255</td>
<td>2394</td>
<td>2243</td>
</tr>
</tbody>
</table>
Reduction and its applications
Reduction

- Threads create local contributions to a final result
- How do we combine the results at low cost?
Application: Compute an Image’s Histogram

- Count the frequency of each possible pixel value in the Red, Green and Blue Channels
Implementation #1

• Represent histogram as a `SharedIntegerArray` and update within the innermost loop:

```java
for (int r = first; r <= last; ++r)
    for (int c = 0; c < width; ++c)
        histogram.incrementAndGet(pixel[c][r]);
```
Implementation #2

• How do we avoid contention?

    for (int r = first; r <= last; ++r)
    for (int c = 0; c < width; ++c)
        thr_histo [pixel[c][r]]++;  

• Within a critical section:
  Everyone sums their thr_histo into the global histogram

• What is the running time?
Implementation #3

• Reduction – express higher order *idiom*
• Use a logarithmic running time algorithm
• Sidebar: hypercubes
Sidebar: what is a hypercube?

- A d-dimensional graph with $2^d$ nodes
- A 0-cube is a single node, 1-cube is a line connecting two points, 2-cube is a square, etc
- Each node has $d$ neighbors
- A hypercube with $p$ nodes has $\lg(p)$ dimensions
Bookkeeping

- Label nodes with a binary reflected grey code

- Neighboring labels differ in exactly one bit position
  \[001 = 101 \oplus e_2, \quad e_2 = 100\]
Hypercube reduction algorithm with p=4

- Threads 0 & 2 sum data from their “buddy”
- Processor 0 is the root, writes final result, using value from hypercube “buddy” on processor 2 (10)
Synchronization in Java
“Happens-before”

- 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

- Memory writes by one statement are visible to another

- Different ways of doing 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);
Intrinsic locks

• Enforce exclusive access
• How is this accomplished

```java
public void addName(String name) {
    synchronized(this) {
        lastName = name; nameCount++;
    }
    nameList.add(name);
}
```
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();
    }
}
```
Other types of Atomic variables

• Atomic arrays
• Atomic references create an immutable container that holds multiple atomically accessed variables

```java
public class Q{
    private final int A, B;
    ...
    public int getA() { return A; }
    public int getB() { return B; }
}

private AtomicReference<Q> q = new AtomicReference<Q>(new Q(0, 0));
```
Deadlock, Starvation and Livelock

• Deadlock
  ▶ 2 more threads are blocked, and require something that another thread has
  ▶ Cyclic dependencies

• Starvation?

• Livelock?