Lecture 5

Revisiting the Java Memory Model
Weak scaling
Optimizing for Cache
Announcements

• A2 assigned today
• Quiz return
The Quiz
Quiz

• Why is there a limit on the number of processors that can share the same memory in an SMP?

• Explain the central theme of parallel programming design: *parallelism is not free*.

• Briefly explain the differences between *shared* variables, *per-thread* variables, and *main program local* variables, both in terms of where they appear in the source code, and any conflicts that may arise in a Java program while it is executing an *IntegerForLoop*.
Visibility of variables

From Kaminsky, “Building Parallel Programs”
Revisiting the Java Memory Model
The Java memory model

• Discussion follows Lea
• Compiler may rearrange statements to improve performance
• Processor may rearrange order of instructions
• Memory system may rearrange order that writes are committed
• Memory cells might not get updated, if ever, after a later call to check()

```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)); }
}
```
Consequences

• Not an issue in single threaded runs
• Optimizations lead to incorrect results in a multiple thread execution
• Design of the class is not thread safe either
• Could use synchronized methods, but may not be necessary

```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)); }
}
```
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
Preliminaries

- Each thread runs on its own CPU
- This may or may not be the case in a specific run of a program
- Java memory model describes an abstract relation between threads and memory
- Model makes guarantees about properties concerning interaction between instruction sequences methods and fields in memory
Rules

• Most rules concern when values must be transferred between main memory and per-thread memory

• 3 issues

• **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.
The effects of synchronization

• The three properties have a simple characterization (Atomicity, Visibility, Ordering)
• 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, and processing of synchronized methods or blocks within any given thread is in program-specified order (Think in terms of SharedInteger)
• Out of order processing cannot matter to other threads employing synchronization
• When synchronization is not used or is used inconsistently, answers are more complex
• Guarantees made by the memory model are weaker than most programmers intuitively expect, and are also weaker than those typically provided on any given JVM implementation
• Imposes additional obligations on programmers attempting to ensure object consistency relations that lie at the heart of exclusion: Objects must maintain invariants as seen by all threads that rely on them, not just by the thread performing any given state modification
Atomicity

- Accesses and updates to the memory cells corresponding to fields of any type except long or double are guaranteed to be atomic.
- Includes references to other objects.
- Extends to volatile long and double.
- When a non-long/double field is used in an expression, you will obtain either its initial value or some value that was written by some thread, but not some jumble of bits resulting from two or more threads both trying to write values at the same time.
- 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 to other threads only under certain conditions.

```java
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.

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; }
}
```

© 2010 Scott B. Baden / CSE 160 / Winter 2010
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

```java
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.

```
Lab #1

- If we produced is a non-volatile int and we use assignment: deadlock
- non-volatile int, incr/decr: 71 to 79 msec (5 runs, N=45000)
- `volatile` int, incr/decr: 12 to 13 msec [12.0 13.5 13.0 12.1 12.3]
- `SharedInteger`: 13 to 58 msec [58.3 55.4 49.4 12.8 13.1]
- `volatile` int, assignment: 12.5 to 45 msec [20.8 12.5 13.6 44.8 17.7]

### PRODUCER

```java
for (int r = 0; r < N; r++) {
    produce x[]
    FULL(produced)
    while (produced is FULL)
        Check sum
}
```

### CONSUMER

```java
for (int r = 0; r < N; r++) {
    while (produced is EMPTY)
        consume x[], compute sum
    EMPTY(produced)
}
```
Revisiting 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 $B==1 \land A == 0$

\[
\begin{array}{ccc}
\text{P1} & \text{P2} & \text{P3} \\
A=1 & \text{if (A==1)} & \text{if (B==1)} \\
B=1 & C=A
\end{array}
\]
Weak Scaling
Scaled Speedup

• Is Amdahl’s law pessimistic?
• Observation: Amdahl’s law assumes that the workload ($W$) remains fixed
• But parallel computers are used to tackle more ambitious workloads

$W$ increases with $P$

$f$ often decreases with $W$
Computing scaled speedup

• Instead of asking what the speedup is, let’s ask how long a parallel program would run on a single processor [J. Gustafson 1992]


• Let $T_P = 1$

• $f' = \text{fraction of serial time spent on the parallel program}$

• $T_1 = f' + (1-f') \times P = S'_P = \text{scaled speedup}$

• Scaled speedup is linear in $P$
Isoefficiency

- Consequence of Gustafson’s observation is that we increase $N$ with $P$.
- Kumar: We can maintain constant efficiency so long as we increase $N$ appropriately.
- The isoefficiency function specifies the growth of $N$ in terms of $P$.
- If $N$ is linear in $P$, we have a scalable computation.
Optimizing for Cache
What is performance programming?

• Most of our time is spent fixing bugs
• We assume that a bug is related to some aspect of correctness
• But we can also have **performance bugs**
Loop Fusion

for (i=0; i<n; i++)
    A[i] += alpha*B[i];

dot = 0;
for (i=0; i<n; i++)
    dot += A[i]*B[i];

\[ N = 4M \]
Matrix multiplication
Matrix Multiplication

• An important core operation in many numerical algorithms

• Given two *conforming* matrices $A$ and $B$, form the matrix product $A \times B$
  
  - $A$ is $m \times n$
  - $B$ is $n \times p$

• Operation count: $O(n^3)$ multiply-adds for an $n \times n$ square matrix
Matrix multiply algorithm

function MM(Matrix A, Matrix B, Matrix C)
    for i := 0 to n – 1
        for j := 0 to n – 1 do
            C[i, j] = 0;
            for k := 0 to n - 1
                C[i, j] += A[i, k] * B[k, j];
            end for
        end for
    end MM
Blocking for cache (tiling)

• Amortize memory accesses by increasing memory reuse

• Discussion follows from James Demmel, UC Berkeley

http://www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html
Unblocked Matrix Multiplication

\[
C += A*B \\
\text{for } i := 0 \text{ to } n-1 \\
\quad \text{for } j := 0 \text{ to } n-1 \\
\quad \text{for } k := 0 \text{ to } n-1 \\
\quad \quad \quad \quad C[i,j] += A[i,k] * B[k,j]
\]
Analysis of performance

for i = 0 to n-1
    // for each iteration i, load all of B into cache
    for j = 0 to n-1
        // for each iteration (i,j), load A[i,:] into cache
        // for each iteration (i,j), load and store C[i,j]
        for k = 0 to n-1
            C[i,j] += A[i,k] * B[k,j]
Analysis of performance

for i = 0 to n-1
  // n \times n^2 / b loads = \frac{n^3}{b}, b=cache line size  B[:,i]
  // n^2 / b loads = \frac{n^2}{b}  A[i,:]
  for j = 0 to n-1
    // n^2 / b loads + n^2 / b stores = \frac{2n^2}{b}  C[i,j]
    for k = 0 to n-1
      C[i,j] += A[i,k] \times B[k,j]  Total: (n^3 + 3n^2) / b
Flops to memory ratio

Let \( q = \# \text{flops} / \text{main memory reference} \)

\[
q = \frac{2n^3}{n^3 + 3n^2}
\]

\( \approx 2 \) as \( n \to \infty \)
Blocked Matrix Multiply

- A, B, C are $n \times n$ matrices
- Divide into $N \times N$ sub blocks
- Each sub block is $B \times B$
  - $B = n/N$ is called the block size
  - how do we establish $B$?
  - assume we have a good quality library to perform matrix multiplication on subblocks

$$C[i,j] = C[i,j] + A[i,k] \ast B[k,j]$$
Blocked Matrix Multiplication

for i = 0 to N-1
    for j = 0 to N-1
        // load each block C[i,j] into cache, once : \( n^2 \)

        // B= n/N = cache line size
        for k = 0 to N-1
            // load each block A[i,k] and B[k,j] \( N^3 \) times
            // = \( 2N^3 \times (n/N)^2 \) = \( 2N^2n^2 \)

        // write each block C[i,j] once : \( n^2 \)
        
Total: \( (2N+2)\times n^2 \)
Flops to memory ratio

Let $q = \#\text{flops} / \text{main memory reference}$

\[
q = \frac{2n^3}{(2N + 2)n^2} = \frac{n}{N + 1}
\]

$\approx \frac{n}{N} = b$

as $n \to \infty$
The results

<table>
<thead>
<tr>
<th>N, B</th>
<th>Unblocked Time</th>
<th>Blocked Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>256, 64</td>
<td>0.6</td>
<td>0.002</td>
</tr>
<tr>
<td>512, 128</td>
<td>15</td>
<td>0.24</td>
</tr>
</tbody>
</table>
Optimizing the block size

N=512