Practical Concurrent and Parallel Programming 4

Peter Sestoft
IT University of Copenhagen

Friday 2016-09-23
Plan for today

• **Performance measurements**
  • A class for measuring elapsed wall-clock time
    – Mark0-5: Towards reliable measurements
    – Mark6-7: Automated general measurements
  • Measuring execution time
    – of memory accesses
    – of thread creation, start, execution
    – of *volatile* fields
  • Measuring the prime counting example
  • General advice, warnings and pitfalls
Ad: Want to be a student programmer?

- **Popular Parallel Programming (P3) project**
  - Project goal: To allow spreadsheet users harness the power of multicore computers; or *end-user supercomputing*.
  - ITU: Florian and Alexander (PhDs), Peter Sestoft
  - AAU: Bent Thomsen, Lone Thomsen, Kim Larsen

- **We need a research programmer!**
- You need these skills:
  - Strong C#, OO and functional programming skills
  - Disciplined developer
  - Understanding of spreadsheets
How long does this method take?

private static double multiply(int i) {
    double x = 1.1 * (double)(i & 0xFF);
    return x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x;  
}

• Does an int operation, int-double conversion, and 20 floating-point multiplications
• So takes at least $20 \times 0.4 = 8$ ns
• Tricks used in this code:
  – Make result depend on $i$ to avoid caching
  – The $i \& 0xFF$ is in range 0—255, avoids overflow
  – Multiply $i \& 0xFF$ by 1.1 to make it floating-point
Back-of-the envelope calculations

• 2.4 GHz processor = 0.4 ns/cycle = 0.4 \times 10^{-9} s

• Throughput:
  – Addition or multiplication takes 1 cycle
  – Division maybe 15 cycles
  – Transcendental functions, \( \sin(x) \) maybe 100-200?

• Instruction-level parallelism
  – 2-3 integer operations/cycle, only sometimes

• Memory latency
  – Registers: 1 cycle
  – L1 cache: a few cycles
  – L2 cache: many cycles
  – RAM: hundreds of cycles – expensive cache misses!
Plan for today

• Performance measurements
• **A class for measuring wall-clock time**
  – **Mark0-5: Towards reliable measurements**
  – **Mark6-7: Automated general measurements**
• Measuring execution time
  – of memory accesses
  – of thread creation, start, execution
  – of volatile fields
• Measuring the prime counting example
• General advice, warnings and pitfalls
A simple Timer class for Java

- We measure elapsed wall-clock time
  - This is what matters in reality
  - Can measure uniformly on Linux, MacOS, Windows
  - Enables comparison Java/C#/C/Scala/F# etc

```java
public class Timer {
    private long start, spent = 0;
    public Timer() { play(); } 
    public double check()  
        { return (System.nanoTime()-start+spent)/1e9; } 
    public void pause() { spent += System.nanoTime()-start; } 
    public void play() { start = System.nanoTime(); }
}
```

- Alternatives: total CPU time, or user + kernel
- Never use imprecise, slow `new Date().getTime()`
- Q: Reasons to measure total CPU time?
Mark0: naïve attempt

Useless because

- Runtime start-up costs larger than execution time
- Timer resolution too coarse, likely 100 ns
- So result are unrealistic and vary a lot

```
public static void Mark0() {
    Timer t = new Timer();
    double dummy = multiply(10);
    double time = t.check() * 1e9;
    System.out.printf("%6.1f ns\n", time);
}
```

5000.0 ns
6000.0 ns
4500.0 ns
Mark1: Measure many operations

public static void Mark1() {
    Timer t = new Timer();
    Integer count = 1_000_000;
    for (int i=0; i<count; i++) {
        double dummy = multiply(i);
    }
    double time = t.check() * 1e9 / count;
    System.out.printf("%6.1f ns\n", time);
}

- Measure 1 million calls; better but fragile:
  - If count is larger, optimizer may notice that result of multiply is not used, and remove call
  - So-called “dead code elimination”
  - May give completely unrealistic results
Java compiler and virtual machine

The `javac` compiler is simple, makes no optimizations

The `java` runtime system (JIT) is clever, makes many
Mark2: Avoid dead code elimination

public static double Mark2() {
    Timer t = new Timer();
    int count = 100_000_000;
    double dummy = 0.0;
    for (int i=0; i<count; i++)
        dummy += multiply(i);
    double time = t.check() * 1e9 / count;
    System.out.printf("%6.1f ns\n", time);
    return dummy;
}

• Much more reliable
Mark3: Automate multiple samples

```java
int n = 10;
int count = 100_000_000;
double dummy = 0.0;
for (int j=0; j<n; j++) {
    Timer t = new Timer();
    for (int i=0; i<count; i++)
        dummy += multiply(i);
    double time = t.check() * 1e9 / count;
    System.out.printf("%6.1f ns\n", time);
}
```

- Multiple samples gives an impression of reproducibility
Mark4: Compute standard deviation

```java
int count = 100_000_000;
double st = 0.0, sst = 0.0;
for (int j=0; j<n; j++) {
    Timer t = new Timer();
    for (int i=0; i<count; i++)
        dummy += multiply(i);
    double time = t.check() * 1e9 / count;
    st += time;
    sst += time * time;
}
double mean = st/n,
        sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
System.out.printf("%6.1f ns +/- %6.3f \n", mean, sdev);
```

• The standard deviation $\sigma$ summarizes the variation around the mean, in a single number

30.3 ns +/- 0.137
Statistics: Central limit theorem

• The average of \( n \) independent identically distributed observations \( t_1, t_2, \ldots, t_n \) tends to follow the normal distribution \( N(\mu, \sigma^2) \) where

\[
\mu = \frac{1}{n} \sum_{j=1}^{n} t_j
\]

\[
\sigma = \sqrt{\frac{1}{n} \sum_{j=1}^{n} t_j^2} - \mu^2
\]

when \( n \) tends to infinity

• Eg with probability 68.3% the "real" result is between 30.163 ns and 30.437 ns
The normal distribution $N(\mu, \sigma^2)$

68.3% of observations in $[\mu-\sigma, \mu+\sigma]$, between blue lines

95.4% of observations in $[\mu-2\sigma, \mu+2\sigma]$, between red lines
```java
int n = 10, count = 1, totalCount = 0;
double dummy = 0.0, runningTime = 0.0;
do {
    count *= 2;
    double st = 0.0, sst = 0.0;
    for (int j=0; j<n; j++) {
        Timer t = new Timer();
        for (int i=0; i<count; i++)
            dummy += multiply(i);
        runningTime = t.check();
        double time = runningTime * 1e9 / count;
        st += time;
        sst += time * time;
        totalCount += count;
    }
    double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
} while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
return dummy / totalCount;
```
### Example results from Mark5

<table>
<thead>
<tr>
<th>mean time</th>
<th>sdev</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0 ns +/- 200.00</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>100.0 ns +/- 122.47</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>62.5 ns +/- 62.50</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>50.0 ns +/- 37.50</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>46.9 ns +/- 15.63</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>40.6 ns +/- 10.36</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>39.8 ns +/- 2.34</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>36.3 ns +/- 1.79</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>36.5 ns +/- 1.25</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>35.6 ns +/- 0.49</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>111.1 ns +/- 232.18</td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>36.1 ns +/- 1.75</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td>33.7 ns +/- 0.84</td>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>32.5 ns +/- 1.07</td>
<td>16384</td>
<td></td>
</tr>
<tr>
<td>35.6 ns +/- 4.84</td>
<td>32768</td>
<td></td>
</tr>
<tr>
<td>30.4 ns +/- 0.26</td>
<td>65536</td>
<td></td>
</tr>
<tr>
<td>33.1 ns +/- 5.06</td>
<td>131072</td>
<td></td>
</tr>
<tr>
<td>30.3 ns +/- 0.49</td>
<td>262144</td>
<td></td>
</tr>
</tbody>
</table>

Outlier, maybe due to other program activity
Advantages of Mark5

• The early rounds (2, 4, ...) serve as warm-up
  – Make sure the code is in memory and cache
• Measured code loop runs at least 0.25 sec
  – Roughly 500 million CPU cycles
  – Lessen impact of other activity on computer
  – Makes sure code has been JIT compiled
• Still, total time spent measuring at most 1 sec
  – Because last measurement runs at most 0.5 sec
  – and sum of previous times is same time as last one
    • because $2 + 4 + 8 + \ldots + 2^n < 2^{n+1}$
• Independent of problem and hardware
Development of the benchmarking method

- Mark0: Measure one call, useless
- Mark1: Measure many calls, nearly useless
- Mark2: Avoid dead code elimination
- Mark3: Automate multiple samples
- Mark4: Compute standard deviation
- Mark5: Automate choice of iteration count

- But need to measure not just multiply!
Plan for today

• Performance measurements
• A class for measuring elapsed wall-clock time
  – Mark0-5: Towards reliable measurements
  – **Mark6-7: Automated general measurements**
• Measuring execution time
  – of memory accesses
  – of thread creation, start, execution
  – of volatile fields
• Measuring the prime counting example
• General advice, warnings and pitfalls
Mark6: Generalize to any function

```java
public interface IntToDoubleFunction {
    double applyAsDouble(int i);
}

static double Mark6(String msg, IntToDoubleFunction f) {
    ...
    do {
        ...
        for (int j=0; j<n; j++) {
            ...
            for (int i=0; i<count; i++)
                dummy += f.applyAsDouble(i);
        }
    }
    ...
    System.out.printf("%-25s %15.1f ns %10.2f %10d\n", msg, ...
} while (runningTime<0.25 && count<Integer.MAX_VALUE/2);
return dummy / totalCount;
```
Example use of Mark6

Mark6("multiply", Benchmark::multiply);

| multiply | 800.0 ns | 1435.27 | 2  |
| multiply | 250.0 ns | 0.00    | 4  |
| multiply | 212.5 ns | 80.04   | 8  |
| multiply | 187.5 ns | 39.53   | 16 |
| multiply | 200.0 ns | 82.92   | 32 |
| multiply | 57.8 ns  | 24.26   | 64 |
| multiply | 46.9 ns  | 4.94    | 128|
| ...      |         |         |    |
| multiply | 30.6 ns  | 0.61    | 2097152|
| multiply | 30.0 ns  | 0.10    | 4194304|
| multiply | 30.1 ns  | 0.15    | 8388608|

Method reference to the function to be measured
public static double Mark7(String msg, IntToDoubleFunction f) {
    ...
    do {
        ...
    } while (runningTime<0.25 && count<Integer.MAX_VALUE/2);
    double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/
            System.out.printf("%-25s %15.1f ns %10.2f %10d\n", ...");
    return dummy / totalCount;
}

Mark7("pow", i -> Math.pow(10.0, 0.1 * (i & 0xFF)));
Mark7("exp", i -> Math.exp(0.1 * (i & 0xFF)));
Mark7("log", i -> Math.log(0.1 + 0.1 * (i & 0xFF)));
Mark7("sin", i -> Math.sin(0.1 * (i & 0xFF)));
Mark7("cos", i -> Math.cos(0.1 * (i & 0xFF)));
Mark7("tan", i -> Math.tan(0.1 * (i & 0xFF)));
...
Mark 7 benchmarking results for Java mathematical functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (ns)</th>
<th>Frequency (%)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pow</td>
<td>75.5</td>
<td>0.43</td>
<td>4194304</td>
</tr>
<tr>
<td>exp</td>
<td>54.9</td>
<td>0.19</td>
<td>8388608</td>
</tr>
<tr>
<td>log</td>
<td>31.4</td>
<td>0.16</td>
<td>8388608</td>
</tr>
<tr>
<td>sin</td>
<td>116.3</td>
<td>0.41</td>
<td>4194304</td>
</tr>
<tr>
<td>cos</td>
<td>116.6</td>
<td>0.33</td>
<td>4194304</td>
</tr>
<tr>
<td>tan</td>
<td>143.6</td>
<td>0.48</td>
<td>2097152</td>
</tr>
<tr>
<td>asin</td>
<td>229.7</td>
<td>2.24</td>
<td>2097152</td>
</tr>
<tr>
<td>acos</td>
<td>217.0</td>
<td>2.46</td>
<td>2097152</td>
</tr>
<tr>
<td>atan</td>
<td>54.3</td>
<td>0.84</td>
<td>8388608</td>
</tr>
</tbody>
</table>

- 2.4 GHz Intel i7; MacOS 10.9.4; 64-bit JVM 1.8.0_11

- So $\sin(x)$ takes $116.3 \text{ ns} \times 2.4 \text{ GHz} = 279 \text{ cycles}$
  - approximately
Saving measurements to a text file

• Command line in Linux, MacOS, Windows

```
java Benchmark > benchmark-20150918.txt
```

• In Linux, MacOS get both file and console

```
java Benchmark | tee benchmark-20150918.txt
```
Platform identification

```java
public static void SystemInfo() {
    System.out.printf("# OS:   %s; %s; %s%n", 
            System.getProperty("os.name"),
            System.getProperty("os.version"),
            System.getProperty("os.arch"));
    System.out.printf("# JVM:  %s; %s%n", 
            System.getProperty("java.vendor"),
            System.getProperty("java.version"));
    // The processor identifier works only on MS Windows:
    System.out.printf("# CPU:  %s; %d "cores"%n", 
            System.getenv("PROCESSOR_IDENTIFIER"),
            Runtime.getRuntime().availableProcessors());
    java.util.Date now = new java.util.Date();
    System.out.printf("# Date: %s%n", 
            new java.text.SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").format(now));
}
```

- **Output information about platform and date:**

  # OS:   Mac OS X; 10.9.5; x86_64
  # JVM:  Oracle Corporation; 1.8.0_51
  # CPU:  null; 8 "cores"
  # Date: 2015-09-15T14:36:48+0200

  15 September 2015 at 14:36 in UTC+2h
Plan for today

• Performance measurements
• A class for measuring elapsed wall-clock time
  – Mark0-5: Towards reliable measurements
  – Mark6-7: Automated general measurements
• **Measuring execution time**
  – of memory accesses
  – of thread creation, start, execution
  – of volatile fields
• Measuring the prime counting example
• General advice, warnings and pitfalls
Cost of memory access

- CPU is fast, RAM slow. Solution: caches

![Diagram showing memory hierarchy with CPU, registers, L1 cache, L2 cache, L3 cache, and RAM]

- How measure it: Array access with “jumps”

```java
int k = 0;
for (int j=0; j<33_554_432; j++)
    k = arr[k];
```

- Memory footprint equals cycle length

- Fixed number of iterations
Memory speeds ns/access as function of memory footprint (bytes)
Cost of object creation

• First: how long to create an ordinary object?

```java
class Point {
    public final int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
}
```

```java
Mark6("Point creation",
    i -> {
        Point p = new Point(i, i);
        return p.hashCode();
    });
```

• Result on i7, approximately 80 ns
• Q: Why return `p.hashCode()`?
• Computing the hash code takes 3.3 ns
  – Q: How can I know that?
Cost of thread creation

- Takes 1030 ns, or 13 x slower than a Point
  - So a Thread object must be somewhat complicated

```java
Mark6("Thread create",
    i -> {
        Thread t = new Thread(() -> {
            for (int j=0; j<1000; j++)
                ai.getAndIncrement();
        });
        return t.hashCode();
    });
```

Actual work, not run, not measured

What we measure
Cost of thread create + start

- Takes 49000 ns
- So a lot of work goes into setting up a task
  - Even after creating it
- Note: does **not** include executing the loop

```java
Mark6("Thread create start",
    i -> {
        Thread t = new Thread(() -> {
            for (int j=0; j<1000; j++)
                ai.getAndIncrement();
        });
        t.start();
        return t.hashCode();
    });
```

Actual work, mostly not run

What we measure
Cost of thread create+start+run+join

- Takes 72700 ns
- Of this, the actual work is 6580 ns, in loop
- Thus ca. 1080 ns to create; 48000 ns to start; 13000 ns run and join; 6580 ns actual work
- *Never create threads for small computations*

```java
Mark6("Thread create start join",
    i -> {
        Thread t = new Thread(() -> {
            for (int j=0; j<1000; j++)
                ai.getAndIncrement();
        });
        t.start();
        try { t.join(); }
        catch (InterruptedException exn) { } 
        return t.hashCode();
    });
```

Actual work is done because of join()
Cost of taking a free lock

- Takes 4.5 ns although sometime 20 ns instead
- Both are very fast
  - The result of much engineering on the Java VM
  - Taking a free lock was much slower in early Java
  - Today no need to use “double-checked-locking”, Goetz antipattern p. 349
- Q: Is it possible to measure time to take a lock already held by another thread?
Cost of volatile

```java
class IntArrayVolatile {
    private volatile int[] array;
    public IntArray(int length) { array = new int[length]; ... }
    public boolean isSorted() {
        for (int i=1; i<array.length; i++)
            if (array[i-1] > array[i])
                return false;
        return true;
    }
}
```

<table>
<thead>
<tr>
<th>Class</th>
<th>Time</th>
<th>Memory</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntArray</td>
<td>3.4 us</td>
<td>0.01</td>
<td>131072</td>
</tr>
<tr>
<td>IntArrayVolatile</td>
<td>17.2 us</td>
<td>0.14</td>
<td>16384</td>
</tr>
</tbody>
</table>

- Volatile read is 5 x slower in this case
  - JIT compiler performs fewer optimizations
- Q: Why not make volatile the default?
Volatile prevents JIT optimizations

• For-loop body of `isSorted`, JITted x86 code:

```assembly
0xdff0: mov 0xc(%rsi),%r8d          ; LOAD %r8d = array field
0xdfff4: mov %r10d,%r9d            ; i NOW IN %r9d
0xdfff7: dec %r9d                   ; i-1 IN %r9d
0xdfffa: mov 0xc(%r12,%r8,8),%ecx ; LOAD %ecx = array.length
0xdfff: cmp %ecx,%r9d              ; INDEX CHECK array.length <= i-1
0xe002: jae 0xe004b                ; IF SO, THROW
0xe0004: mov 0xc(%rsi),%ecx        ; LOAD %ecx = array field
0xe0007: lea (%r12,%r8,8),%r11    ; LOAD %r11 = array base address
0xe000b: mov 0xc(%r11,%r10,4),%r1ld; LOAD %r1ld = arr[i-1]
0xe0010: mov 0xc(%r12,%rcx,8),%r8d ; LOAD %r8d = array.length
0xe0015: cmp %r8d,%r10d            ; INDEX CHECK array.length <= i
0xe0018: jae 0xe006d               ; IF SO, THROW
0xe001a: lea (%r12,rcx,8),%r8     ; LOAD %r8 = array base address
0xe001e: mov 0x10(%r8,%r10,4),%r9d; LOAD %r9d = array[i]
0xe0023: cmp %r9d,%r1ld            ; IF arr[i] < array[i-1]
0xe0026: jg 0xe008d                ; RETURN FALSE
0xe0028: mov 0xc(%rsi),%r8d        ; LOAD %r8d = array field
0xe002c: inc %r10d                 ; i++
```

VolatileArray.java

array volatile

3 reads of array field

2 index checks

• Non-volatile: read `arr` once, unroll loop, ...

```assembly
0xcb9: mov 0xc(%rdi,%r11,4),%r8d   ; LOAD %rd8d = array[i-1]
0xcbe: mov 0x10(%rdi,%r11,4),%r10d; LOAD %rd10d = array[i]
0xcc3: cmp %r10d,%r8d             ; IF array[i] > array[i-1]
0xcc6: jg 0xd85                   ; RETURN FALSE
```
## Full measurements on two platforms

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Error</th>
<th>Measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashCode()</td>
<td>3.3 ns</td>
<td>0.02</td>
<td>134217728</td>
</tr>
<tr>
<td>Point creation</td>
<td>80.9 ns</td>
<td>1.06</td>
<td>4194304</td>
</tr>
<tr>
<td>Thread's work</td>
<td>6581.5 ns</td>
<td>37.64</td>
<td>65536</td>
</tr>
<tr>
<td>Thread create</td>
<td>1030.3 ns</td>
<td>20.17</td>
<td>262144</td>
</tr>
<tr>
<td>Thread create start</td>
<td>48929.6 ns</td>
<td>320.94</td>
<td>8192</td>
</tr>
<tr>
<td>Thread create start join</td>
<td>72758.9 ns</td>
<td>1204.68</td>
<td>4096</td>
</tr>
<tr>
<td>Uncontended lock</td>
<td>4.1 ns</td>
<td>0.06</td>
<td>67108864</td>
</tr>
</tbody>
</table>

**Intel i7, 2.4 GHz, 4 core**
45 W, Sep 2012, $378

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Error</th>
<th>Measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashCode()</td>
<td>15.5 ns</td>
<td>0.01</td>
<td>16777216</td>
</tr>
<tr>
<td>Point creation</td>
<td>184.1 ns</td>
<td>0.43</td>
<td>2097152</td>
</tr>
<tr>
<td>Thread's work</td>
<td>30802.5 ns</td>
<td>18.65</td>
<td>8192</td>
</tr>
<tr>
<td>Thread create</td>
<td>3690.2 ns</td>
<td>7.99</td>
<td>131072</td>
</tr>
<tr>
<td>Thread create start</td>
<td>153097.2 ns</td>
<td>11142.30</td>
<td>2048</td>
</tr>
<tr>
<td>Thread create start join</td>
<td>165992.8 ns</td>
<td>3916.62</td>
<td>2048</td>
</tr>
<tr>
<td>Uncontended lock</td>
<td>16.9 ns</td>
<td>0.01</td>
<td>16777216</td>
</tr>
</tbody>
</table>

**AMD 6386 SE, 2.8 GHz, 16 core**
140 W, Nov 2012, $1392
Plan for today

• Performance measurements
• A class for measuring elapsed wall-clock time
  – Mark0-5: Towards reliable measurements
  – Mark6-7: Automated general measurements
• Measuring execution time
  – of memory accesses
  – of thread creation, start, execution
  – of volatile fields
• Measuring the prime counting example
• General advice, warnings and pitfalls
Measuring TestCountPrimes

```java
final int range = 100_000;
Mark6("countSequential",
    i -> countSequential(range));
Mark6("countParallel",
    i -> countParallelN(range, 10));
```

- Include Mark6 and Mark7 in source file
  - Modified to show microseconds not nanoseconds
- Reduce range to 100,000
- Threads must be join()’ed to measure time
  - Else you just measure the time to create and start, not the time to actually compute
### TestCountPrimes results, 10 threads

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (us)</th>
<th>Speedup</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>countSequential</td>
<td>11117.3</td>
<td>501.25</td>
<td>2</td>
</tr>
<tr>
<td>countSequential</td>
<td>10969.3</td>
<td>82.93</td>
<td>4</td>
</tr>
<tr>
<td>countSequential</td>
<td>10935.4</td>
<td>52.34</td>
<td>8</td>
</tr>
<tr>
<td>countSequential</td>
<td>10936.0</td>
<td>32.76</td>
<td>16</td>
</tr>
<tr>
<td>countSequential</td>
<td>10970.5</td>
<td>142.69</td>
<td>32</td>
</tr>
<tr>
<td>countParallel</td>
<td>3944.9</td>
<td>764.30</td>
<td>2</td>
</tr>
<tr>
<td>countParallel</td>
<td>3397.5</td>
<td>166.58</td>
<td>4</td>
</tr>
<tr>
<td>countParallel</td>
<td>3218.1</td>
<td>59.62</td>
<td>8</td>
</tr>
<tr>
<td>countParallel</td>
<td>3224.4</td>
<td>62.28</td>
<td>16</td>
</tr>
<tr>
<td>countParallel</td>
<td>3261.4</td>
<td>65.42</td>
<td>32</td>
</tr>
<tr>
<td>countParallel</td>
<td>3379.1</td>
<td>224.53</td>
<td>64</td>
</tr>
<tr>
<td>countParallel</td>
<td>3239.2</td>
<td>111.56</td>
<td>128</td>
</tr>
</tbody>
</table>

- So 10 threads is $10970/3239 = 3.4$ x faster
- What about 1 thread, 2, ..., 32 threads?
Measuring different thread counts

Mark7("countSequential", i -> countSequential(range));

for (int c=1; c<=100; c++) {
    final int threadCount = c;
    Mark7(String.format("countParallelLocal %6d",
                          threadCount),
            i -> countParallelNLocal(range, threadCount));
}

• Q: Why the final int threadCount = c?
TestCountPrimes results

- One thread slower than sequential
- Max speedup 4.1x
- From some point, more threads are worse
- How choose best thread count?
- Tasks and executors are better than threads, week 5

<table>
<thead>
<tr>
<th>countParallel</th>
<th>1</th>
<th>11887.9 us</th>
<th>513.02</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>countParallel</td>
<td>2</td>
<td>7313.4 us</td>
<td>792.47</td>
<td>32</td>
</tr>
<tr>
<td>countParallel</td>
<td>3</td>
<td>5085.8 us</td>
<td>67.75</td>
<td>64</td>
</tr>
<tr>
<td>countParallel</td>
<td>4</td>
<td>4697.3 us</td>
<td>76.39</td>
<td>64</td>
</tr>
<tr>
<td>countParallel</td>
<td>5</td>
<td>4042.7 us</td>
<td>40.06</td>
<td>64</td>
</tr>
<tr>
<td>countParallel</td>
<td>6</td>
<td>3577.5 us</td>
<td>19.87</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>7</td>
<td>3233.1 us</td>
<td>8.28</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>8</td>
<td>3149.4 us</td>
<td>77.59</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>9</td>
<td>3196.3 us</td>
<td>11.66</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>10</td>
<td>3203.0 us</td>
<td>8.49</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>11</td>
<td>3198.5 us</td>
<td>15.70</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>12</td>
<td>3263.3 us</td>
<td>27.53</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>13</td>
<td>3128.0 us</td>
<td>16.66</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>14</td>
<td>3021.6 us</td>
<td>19.58</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>15</td>
<td>2960.8 us</td>
<td>11.23</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>16</td>
<td>3033.4 us</td>
<td>65.49</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>17</td>
<td>2926.2 us</td>
<td>5.94</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>18</td>
<td>2972.6 us</td>
<td>21.47</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>19</td>
<td>3001.7 us</td>
<td>6.40</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>20</td>
<td>3051.9 us</td>
<td>37.81</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>21</td>
<td>2992.3 us</td>
<td>8.10</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>22</td>
<td>2978.9 us</td>
<td>20.45</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>23</td>
<td>2957.3 us</td>
<td>5.70</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>24</td>
<td>2978.5 us</td>
<td>7.67</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>25</td>
<td>3006.8 us</td>
<td>38.01</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>26</td>
<td>2972.0 us</td>
<td>19.80</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>27</td>
<td>2993.0 us</td>
<td>63.53</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>28</td>
<td>3008.0 us</td>
<td>24.42</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>29</td>
<td>2997.7 us</td>
<td>5.80</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>30</td>
<td>3019.1 us</td>
<td>21.74</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>31</td>
<td>2998.5 us</td>
<td>2.80</td>
<td>128</td>
</tr>
<tr>
<td>countParallel</td>
<td>32</td>
<td>3000.7 us</td>
<td>2.38</td>
<td>128</td>
</tr>
</tbody>
</table>
Making plots of measurements

- Zillions of plotting and charting programs, including Excel, Gnuplot, R, Ploticus, ...
- Always use scatter (x-y) plots, no smoothing
General advice

• To avoid interference with measurements, shut down other programs: mail, Skype, browsers, Dropbox, iTunes, MS Office ...
• Disable logging and debugging messages
• Compile with optimizations enabled
• Never measure inside IDEs such as Eclipse
• Turn off power-savings modes
• Run on mains power, not on battery
• Lots of differences between
  – Runtime systems: Oracle, IBM Java; Mono, .NET
  – CPUs: Intel i5, i7, Xeon, AMD, ARM, ...
Mistakes and pitfalls

- Windows Upgrade etc may ruin measurements
  - Runs at unpredictable times, and is slow
- Some CPUs have a temporary “turbo mode”
  - May increase clock speed, will ruin measurements
- Some CPUs do “thermal throttling” if too hot
  - May reduce clock speed, will ruin measurements
- Measure the right thing
  - Eg when measuring binary search, do not search for the same item repeatedly (notes §11)
- Beware of irrelevant overheads
  - For instance random number generation
  - (But now you know how to measure the overhead!)
Timing threads à la Goetz & Bloch

• A countdown N-latch is a use-once gate
  – When latch.countDown() has been called N times, all threads blocked on latch.await() are unblocked

• Can use it to measure thread wall-clock time
  – excluding thread creation and start-up

• But thread start costs seems relevant too...
Timing threads à la Goetz & Bloch

- All threads start nearly at the same time
- Measure excludes thread creation overhead

```java
final CountDownLatch startGate = new CountDownLatch(1);
final CountDownLatch endGate = new CountDownLatch(threadCount);
for (int i = 0; i < threadCount; i++) {
    Thread t = new Thread(new Runnable() {
        public void run() {
            try {
                startGate.await();
                try {
                    task.run();
                } finally {
                    endGate.countDown();
                }
            } catch (InterruptedException ignored) {
            }
        }
    });
    t.start();
}

Timer timer = new Timer();
startGate.countDown();
endGate.await();
double time = timer.check();
```
Throughput versus latency

- Throughput is results per second
- Latency is time to first result

Water pipe analogy:
- Pipe diameter determines throughput, drops/sec
- Pipe length determines latency, time to first drop

We measure inverse throughput, sec/result
This week

• Reading
  – Sestoft: Microbenchmarks in Java and C#
  – (Optional) McKenney chapter 3

• Exercises week 4 = Mandatory hand-in 2
  – Conduct meaningful performance measurements and comparisons, and discuss the results

• Read before next week’s lecture
  – Goetz chapters 6 and 8
  – Bloch items 68, 69