Practical Concurrent and Parallel Programming 3

Thomas Dybdahl Ahle
IT University of Copenhagen

Thursday 2019-09-12

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
- Based on **`benchmarking.pdf`**, [Georges] and slides by Peter Sestoft

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;
}
```

- Does an int operation, int-double conversion, and 20 floating-point multiplications
- So takes at least 20 * 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 = 0.4×10^{-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

Benchmark.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

```
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);
}
```

- 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

```
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);
}
```

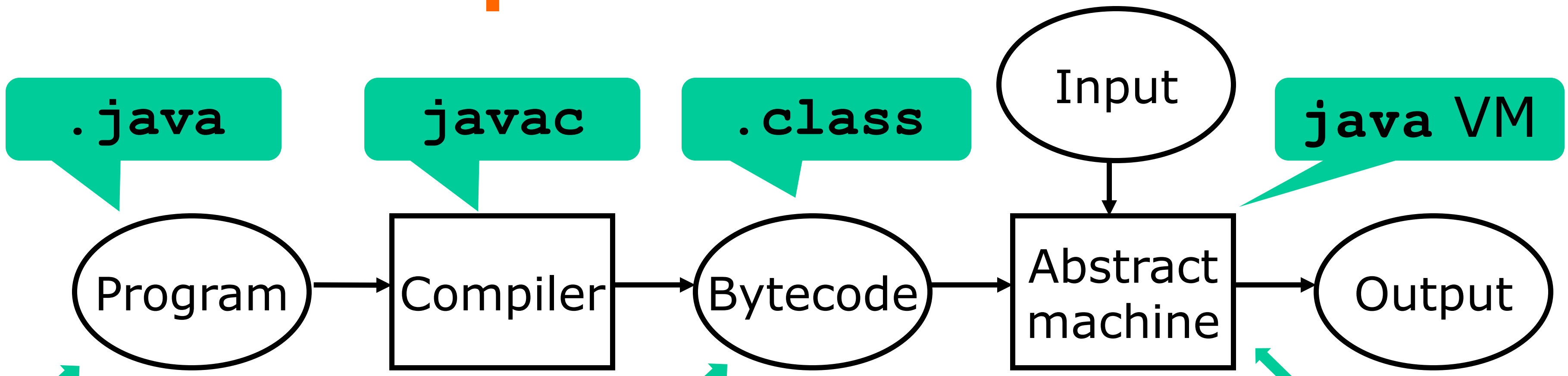
Quite useless

```
5.0 ns
5.5 ns
5.0 ns
```

- 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

```
0.1 ns
0.1 ns
0.0 ns
```

Java compiler and virtual machine




```
for (int i=0; i<n; i++)
  sum += sqrt(arr[i]);
```

JVM

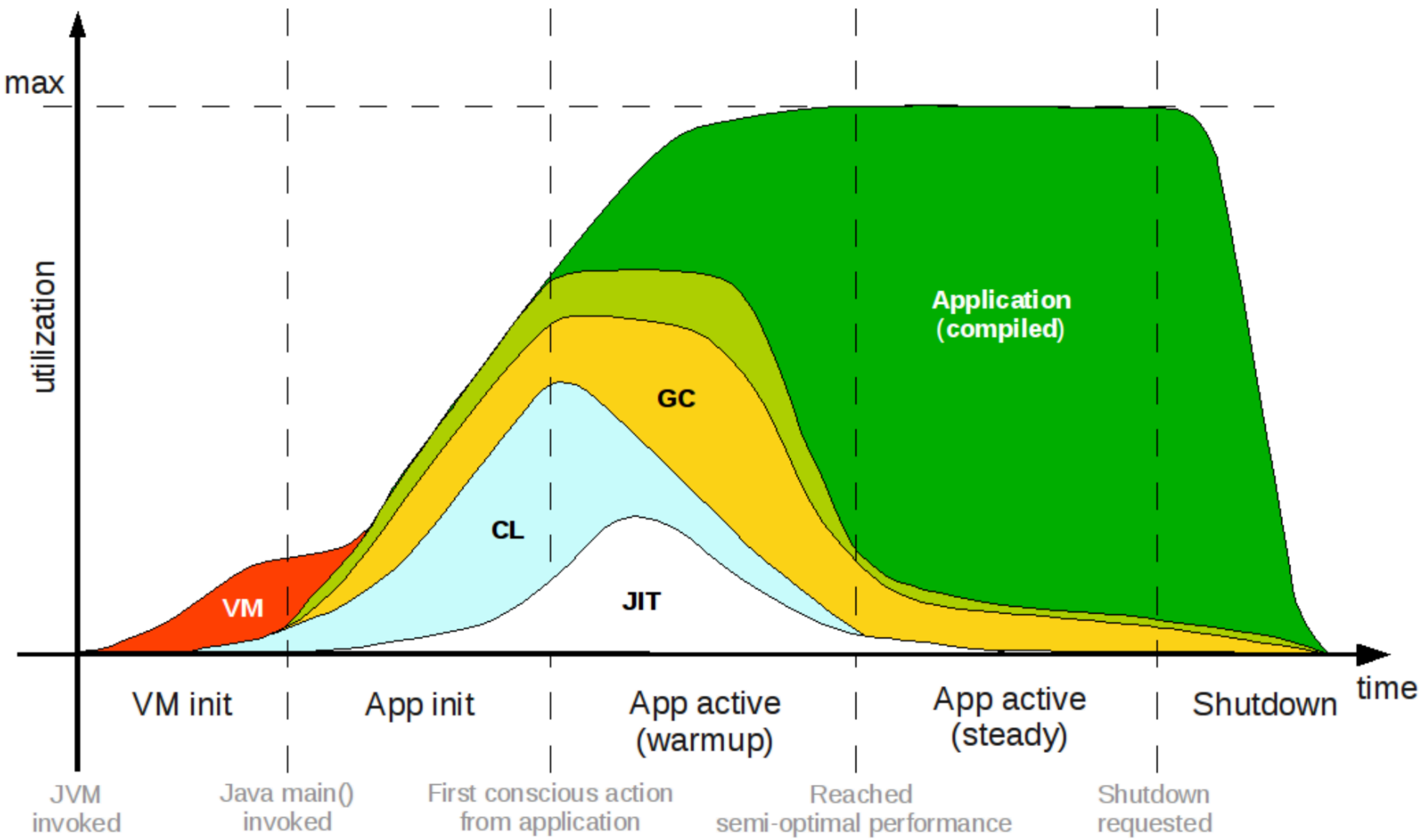
```
21 iconst_0
22 istore 5
24 iload 5
26 iload 2
27 if_icmpge     46
30 dload 3
31 aload 1
32 iload 5
34 daload
35 invokestatic Math.sqrt:(D)D
38 dadd
39 dstore 3
40 iinc 5, 1
43 goto 24
```

x86

```
19 xorl %ebx,%ebx
1b jmp 3a
1d leal 0x00(%ebp),%ebp
20 fldl 0xec(%ebp)
23 cmpl %ebx,0x0c(%edi)
26 jbe 49
2c leal 0x10(%edi,%ebx,8),%eax
30 fldl (%eax)
32 fsqrt
34 faddp %st,%st(1)
36 fstpl 0xec(%ebp)
39 incl %ebx
3a cmpl %esi,%ebx
3c jl 20
```

- The **javac** compiler is simple, makes no optimizations
- The **java** runtime system (JIT) is clever, obeys spec

Typical Java Runtime Lifecycle (Not to scale!)



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;
}
```

```
30.5 ns
30.4 ns
30.3 ns
```

- Much more reliable

Mark3: Automate multiple samples

```
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);
}
```

Number of samples

Iterations per sample

```
30.7 ns
30.3 ns
30.1 ns
30.7 ns
30.5 ns
30.4 ns
30.9 ns
30.3 ns
30.5 ns
30.8 ns
```

- Multiple samples gives an impression of reproducibility

Mark4: Compute standard deviation

```
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);
```

Is this a reasonable iteration count?

- The standard deviation σ summarizes the variation around the mean, in a single number

```
30.3 ns +/-  0.137
```

Statistics: Central limit theorem

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

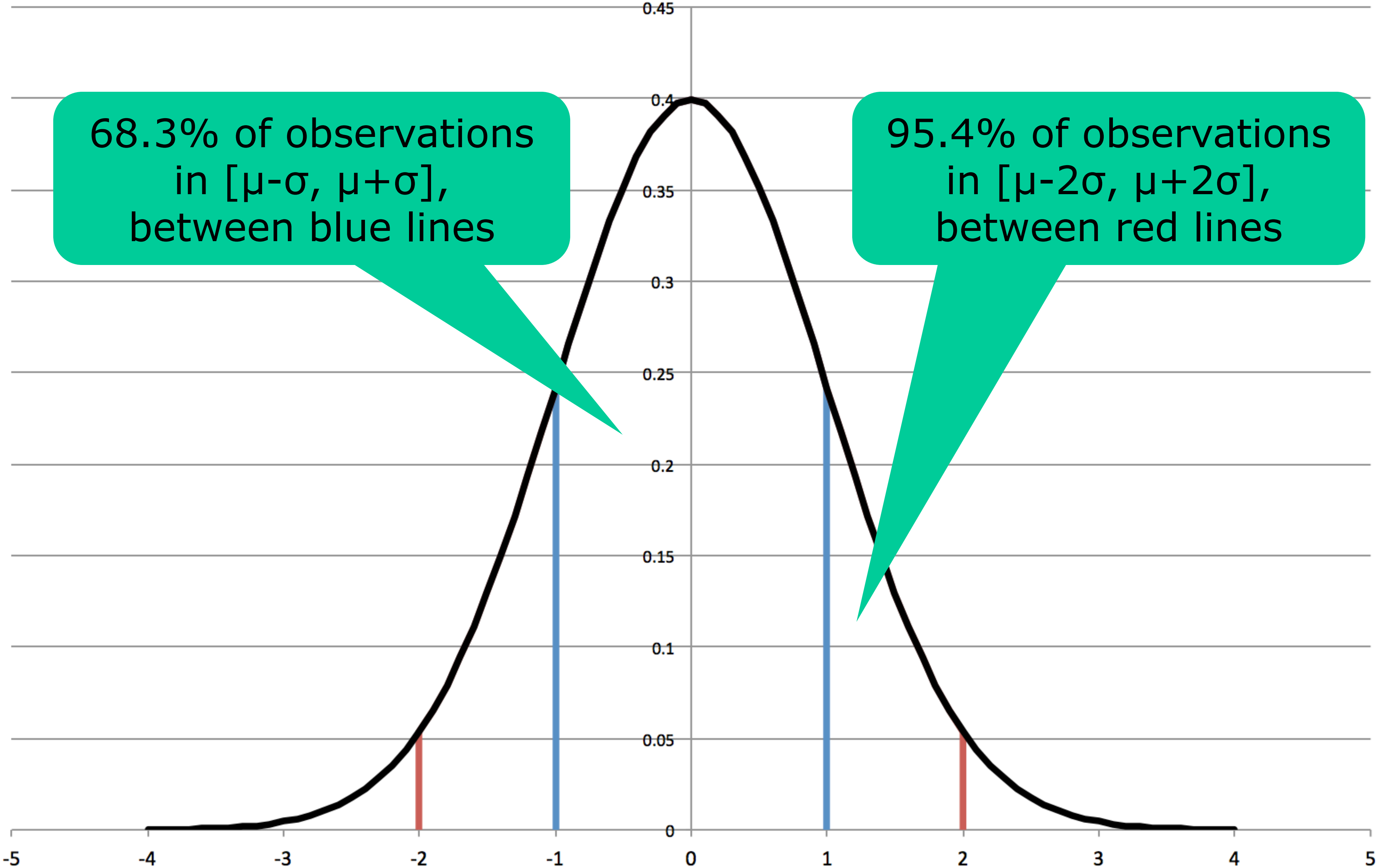
$$\mu = \frac{1}{n}\sum_{j=1}^{n} t_j$$

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

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(μ,σ²)



Warning: Direct method vs Welford's

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Requires two passes :-(

$$\sigma^2 = \frac{1}{n(n-1)}\left(n\sum_{i=1}^{n}x_i^2 - \left(\sum_{i=1}^{n}x_k\right)^2\right)$$

If x and y agree to m significant figures, up to m significant figures can be lost in computing x-y.

Digits of precision

- IEEE 754 double-precision binary floating-point format: binary64
- The 53-bit significand precision gives from 15 to 17 significant decimal digits precision
- 10101010000101101100011110101.111
- 10101010000101101100011110001.100
- 00000000000000000000000000100.011
- 00000000000000000101110101101.101

Knuth's method

Initialize $M_1 = x_1$ and $S_1 = 0$.

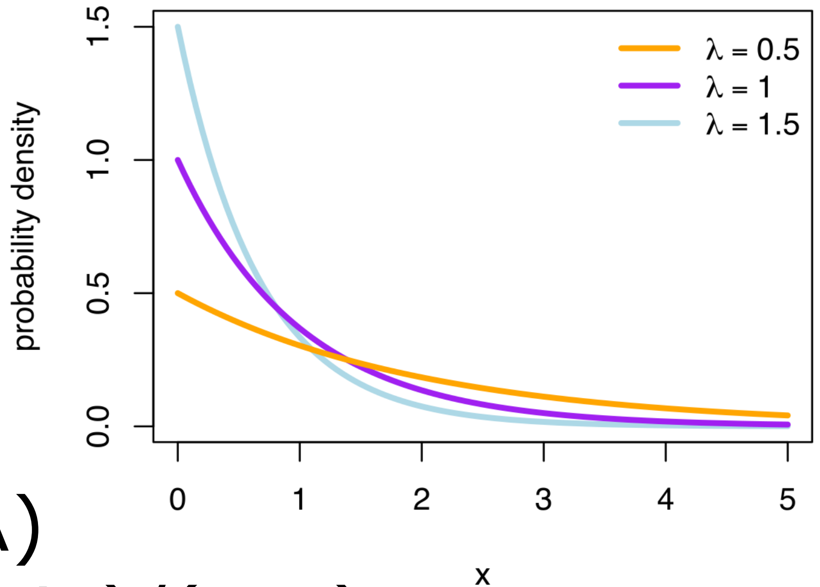
For subsequent *x*s, use the recurrence formulas:

$$M_k = M_{k-1} + (x_k - M_{k-1})/k$$
$$S_k = S_{k-1} + (x_k - M_{k-1})*(x_k - M_k)$$

Sample variance is now S_k / (k-1)

Why even take the average?

- Assume you are measuring a value **t**
- You have measurements x_1, x_2, x_3, …
- Each xi = **t** + Noise_i
- Noise is positive!!
- All xi >= **t**
- So avg(x_1, x_2, …)
  $\geq$ min(x_1, x_2, …) $\geq$ t



- If $\text{Noise}_i \sim \text{Exponential}(\lambda)$
  Then T = min – (avg – min)/(n-1)
  is an unbiased estimator of **t**.

Mark5: Auto-choose iteration count

```
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;
```

Double **count** until ...

... loop runs at least 0.25 sec

Example results from Mark5

Outlier, maybe due to other program activity

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

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

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

From java.util.function

```
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;
}
```

Call given function f

Example use of Mark6

Method reference to the function to be measured

```
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
```

Mark7: Print only last measurement

```
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;
}
```

Printing moved from here to outside loop

```
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)));
...
```

Lambda expressions for functions to be measured

Mark 7 benchmarking results for Java mathematical functions

`pow`	`75.5 ns`	`0.43`	`4194304`
`exp`	`54.9 ns`	`0.19`	`8388608`
`log`	`31.4 ns`	`0.16`	`8388608`
`sin`	`116.3 ns`	`0.41`	`4194304`
`cos`	`116.6 ns`	`0.33`	`4194304`
`tan`	`143.6 ns`	`0.48`	`2097152`
`asin`	`229.7 ns`	`2.24`	`2097152`
`acos`	`217.0 ns`	`2.46`	`2097152`
`atan`	`54.3 ns`	`0.84`	`8388608`

- 2.4 GHz Intel i7; MacOS 10.9.4; 64-bit JVM 1.8.0_11
- So **`sin(x)`** takes 116.3 ns x 2.4 GHz = 279 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

```
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: 2019-09-12T14:36:48+0200
```

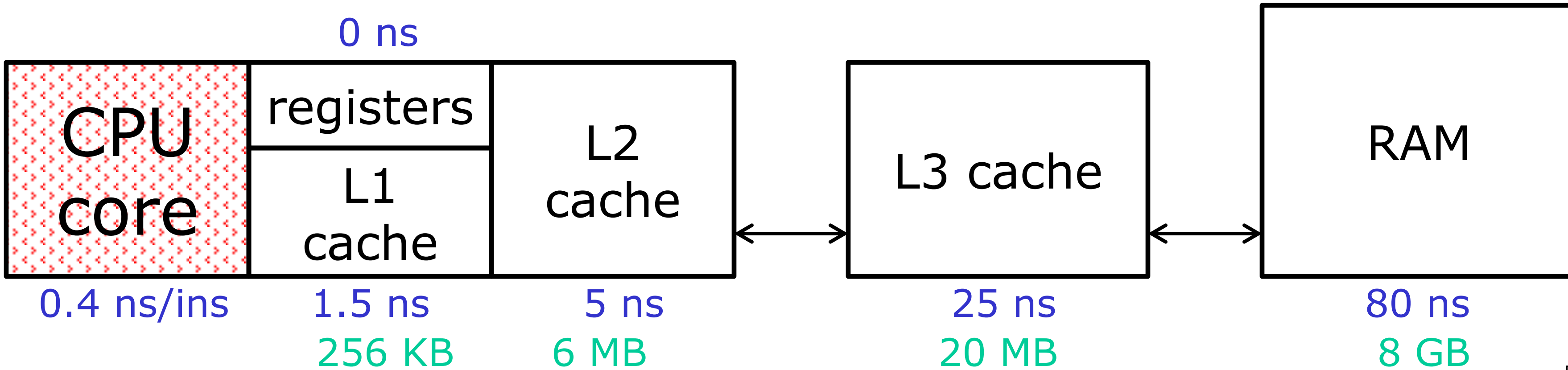
12 September 2015
at 14:36 in UTC+2h

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

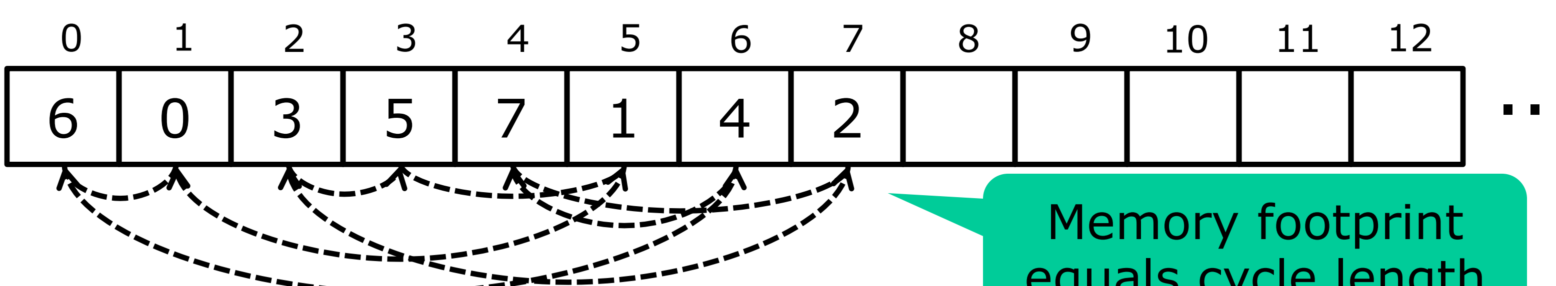


- How *measure* it: Array access with “jumps”

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

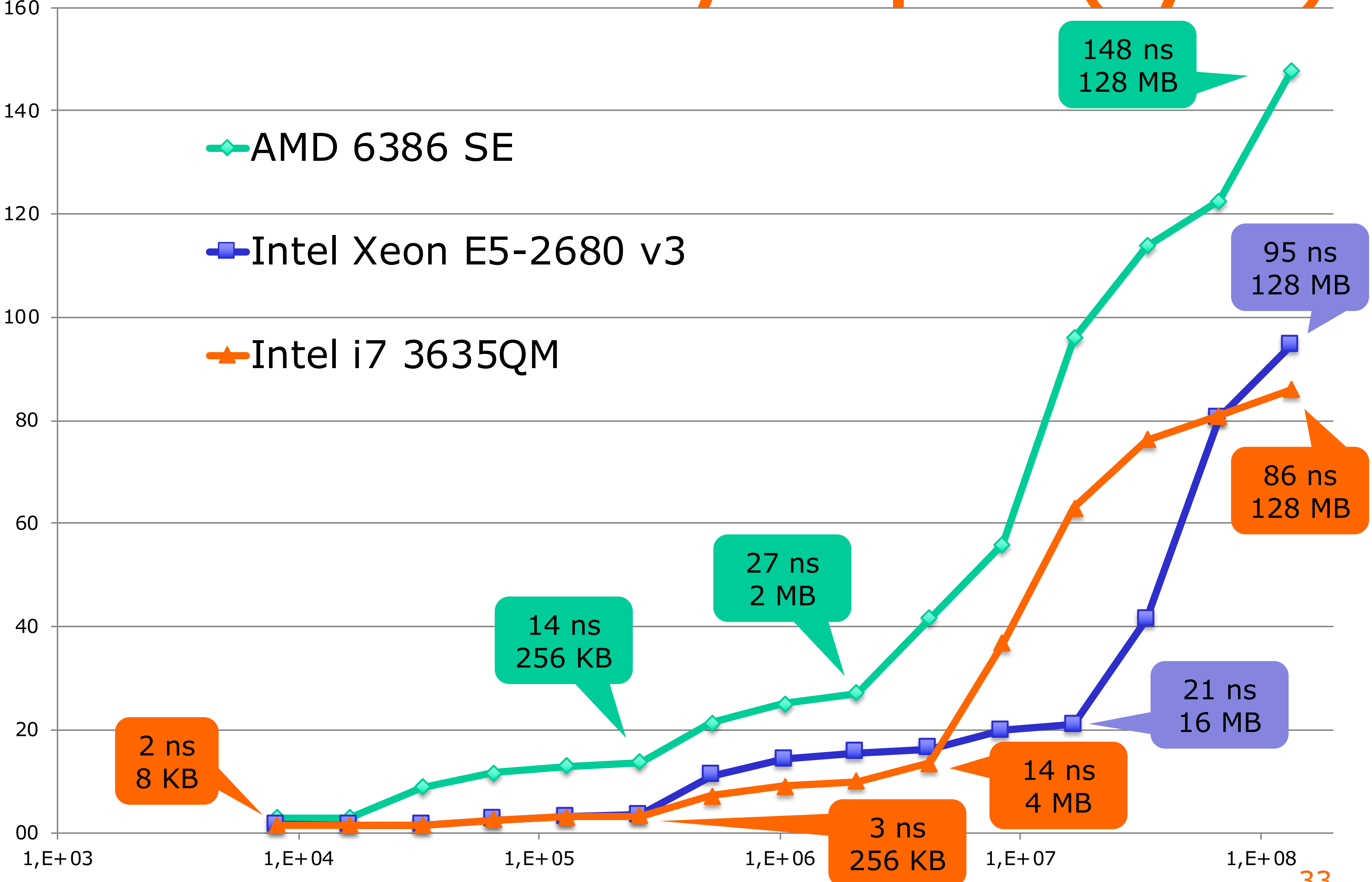
Fixed number of iterations

TestMemoryLatency.java

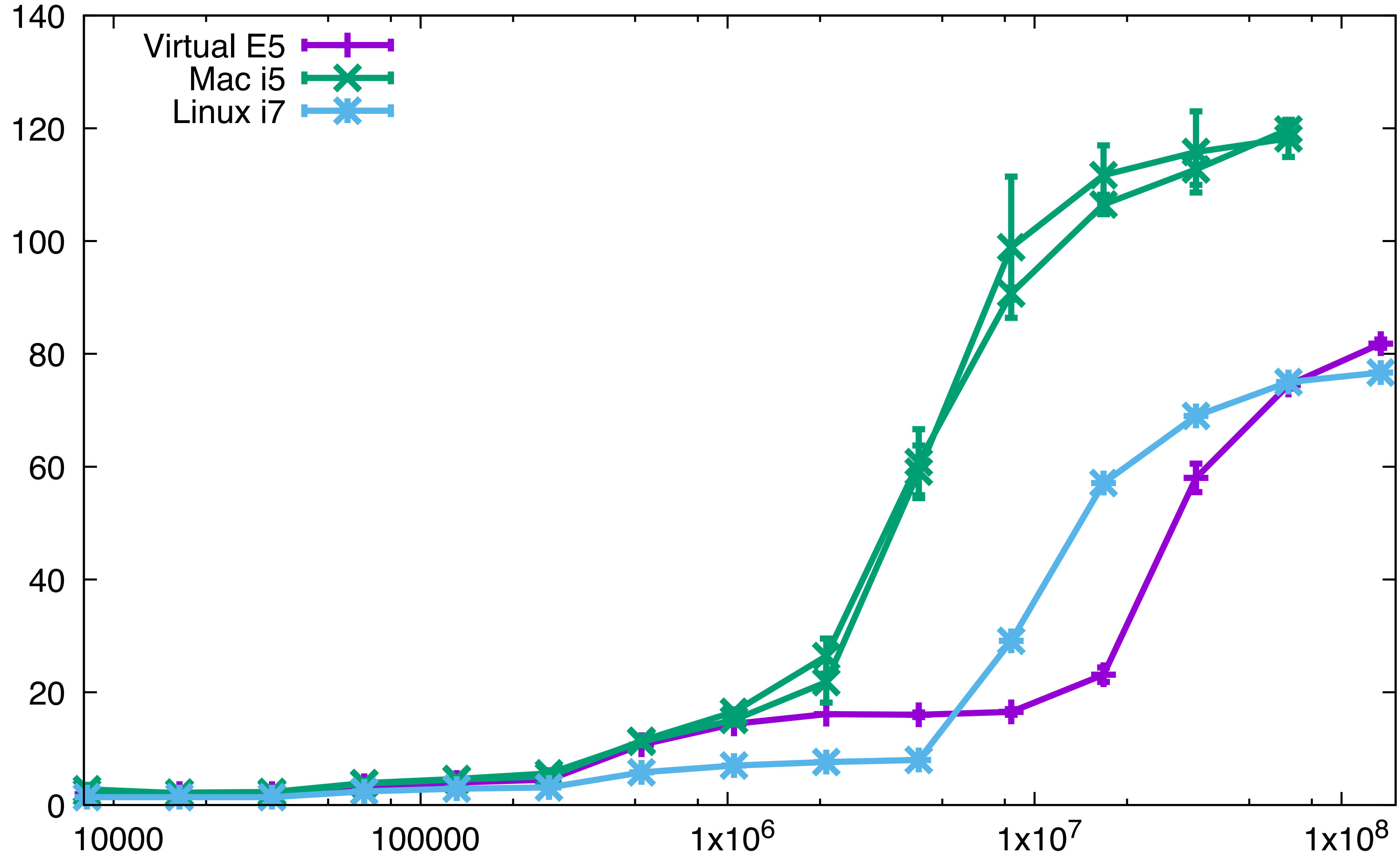


Memory footprint equals cycle length

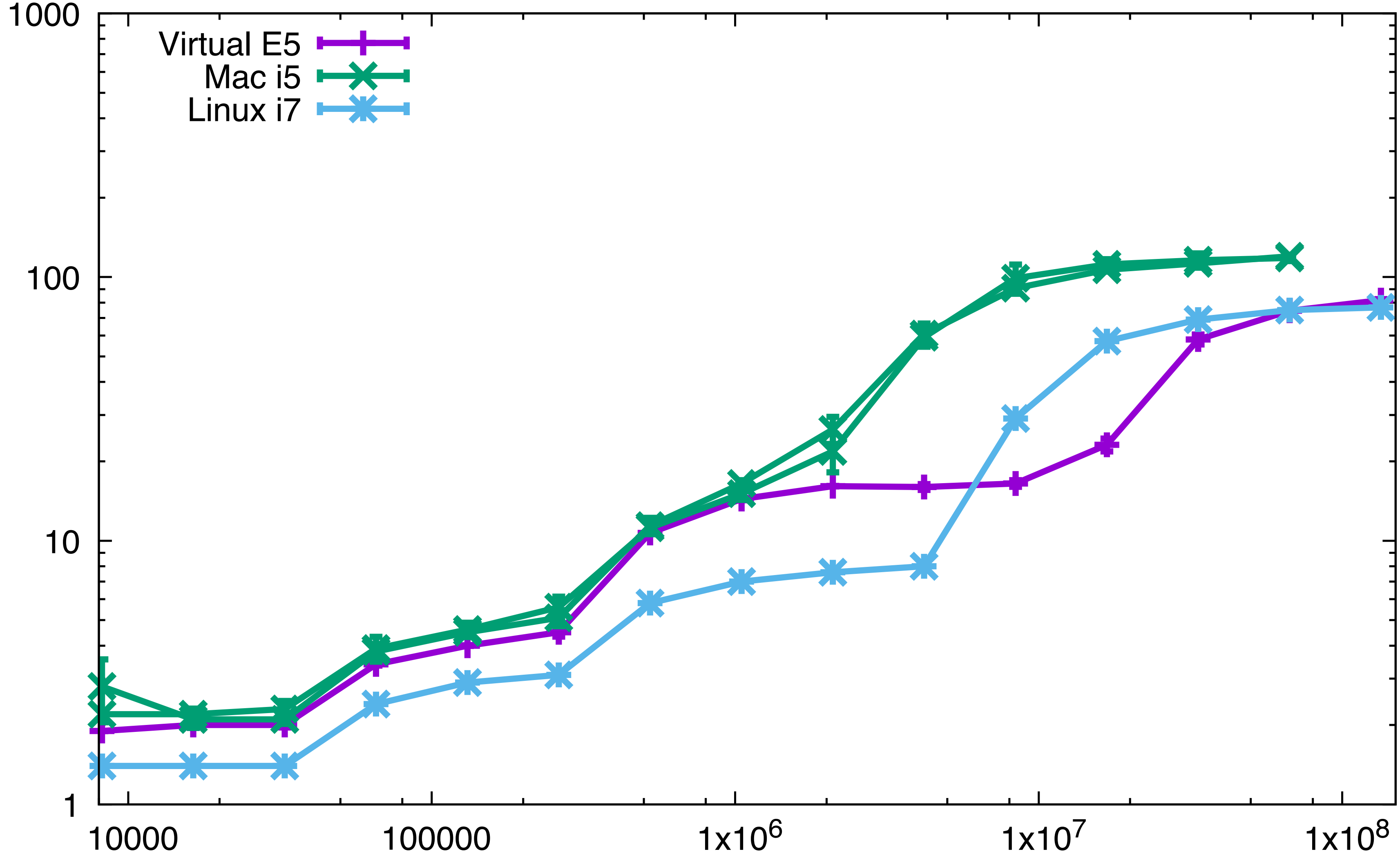
Memory speeds ns/access as function of memory footprint (bytes)



New measurements



New measurements



Cost of object creation

- First: how long to create an ordinary object?

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

TestTimeThreads.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

```
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

TestTimeThreads.java

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

Cost of thread create + start

```
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

TestTimeThreads.java

- 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

Cost of thread create+start+run+join

```
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()

TestTimeThreads.java

- 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*

Cost of taking a free lock

```
Mark6("Uncontended lock",
      i -> {
        synchronized (obj) {
          return i;
        }
      });
```

Succeeds immediately because only one thread is running

TestTimeThreads.java

- 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?

Double Checked Locking

```
@NotThreadSafe
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}
```

LISTING 16.7. Double-checked-locking antipattern. *Don't do this.*

Cost of volatile

```
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;
  }
}
```

TestVolatileCost.java

```
IntArray                     3.4 us          0.01     131072
IntArrayVolatile            17.2 us          0.14      16384
```

- 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:

```
0xdfff0: 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
0xdffff: cmp    %ecx,%r9d                ; INDEX CHECK array.length <= i-1
0xe0002: jae    0xe004b                  ; IF SO, THROW
0xe0004: mov    0xc(%rsi),%ecx           ; LOAD %ecx = array field
0xe0007: lea    (%r12,%r8,8),%r11        ; LOAD %r11 = array base addre
0xe000b: mov    0xc(%r11,%r10,4),%r11d   ; LOAD %r11d = 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,%r11d               ; IF arr[i] < array[i-1]
0xe0026: jg     0xe008d                  ; RETURN FALSE
0xe0028: mov    0xc(%rsi),%r8d           ; LOAD %r8d = array field
0xe002c: inc    %r10d                    ; i++
```

array volatile

3 reads of array field

2 index checks

VolatileArray.java

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

```
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
```

array not volatile

Full measurements on two platforms

hashCode()	3.3 ns	0.02	134217728
Point creation	80.9 ns	1.06	4194304
Thread's work	6581.5 ns	37.64	65536
Thread create	1030.3 ns	20.17	262144
Thread create start	48929.6 ns	320.94	8192
Thread create start join	72758.9 ns	1204.68	4096
Uncontended lock	4.1 ns	0.06	67108864

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

hashCode()	15.5 ns	0.01	16777216
Point creation	184.1 ns	0.43	2097152
Thread's work	30802.5 ns	18.65	8192
Thread create	3690.2 ns	7.99	131072
Thread create start	153097.2 ns	11142.30	2048
Thread create start join	165992.8 ns	3916.62	2048
Uncontended lock	16.9 ns	0.01	16777216

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

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

TestCountPrimesThreads.java

- 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

countSequential	11117.3 us	501.25	2
countSequential	10969.3 us	82.93	4
countSequential	10935.4 us	52.34	8
countSequential	10936.0 us	32.76	16
countSequential	10970.5 us	142.69	32
countParallel	3944.9 us	764.30	2
countParallel	3397.5 us	166.58	4
countParallel	3218.1 us	59.62	8
countParallel	3224.4 us	62.28	16
countParallel	3261.4 us	65.42	32
countParallel	3379.1 us	224.53	64
countParallel	3239.2 us	111.56	128

TestCountPrimesThreads.java

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

Measuring different thread counts

TestCountPrimesThreads.java

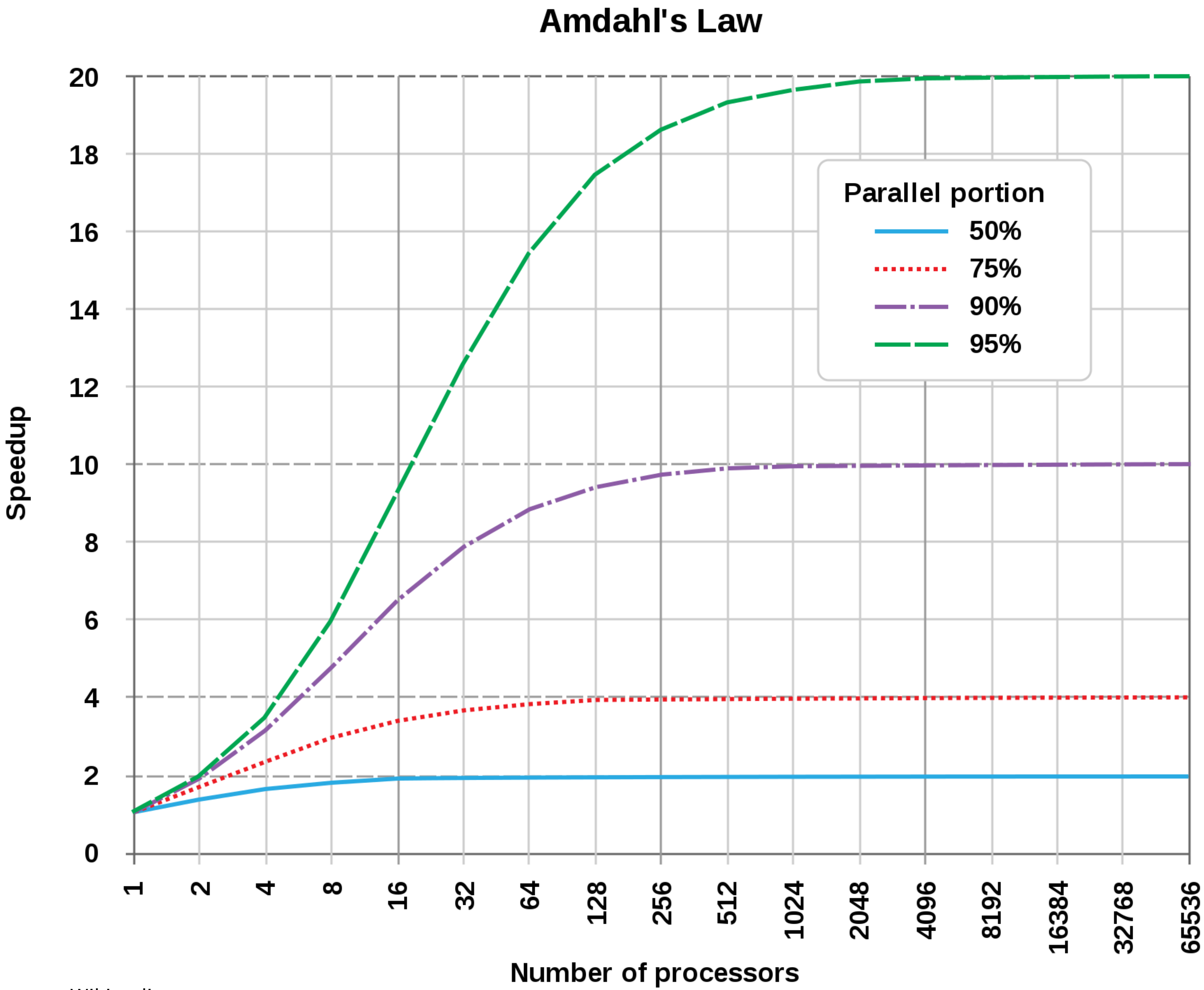
```
Mark7("countSequential", i -> countSequential(range));

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

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

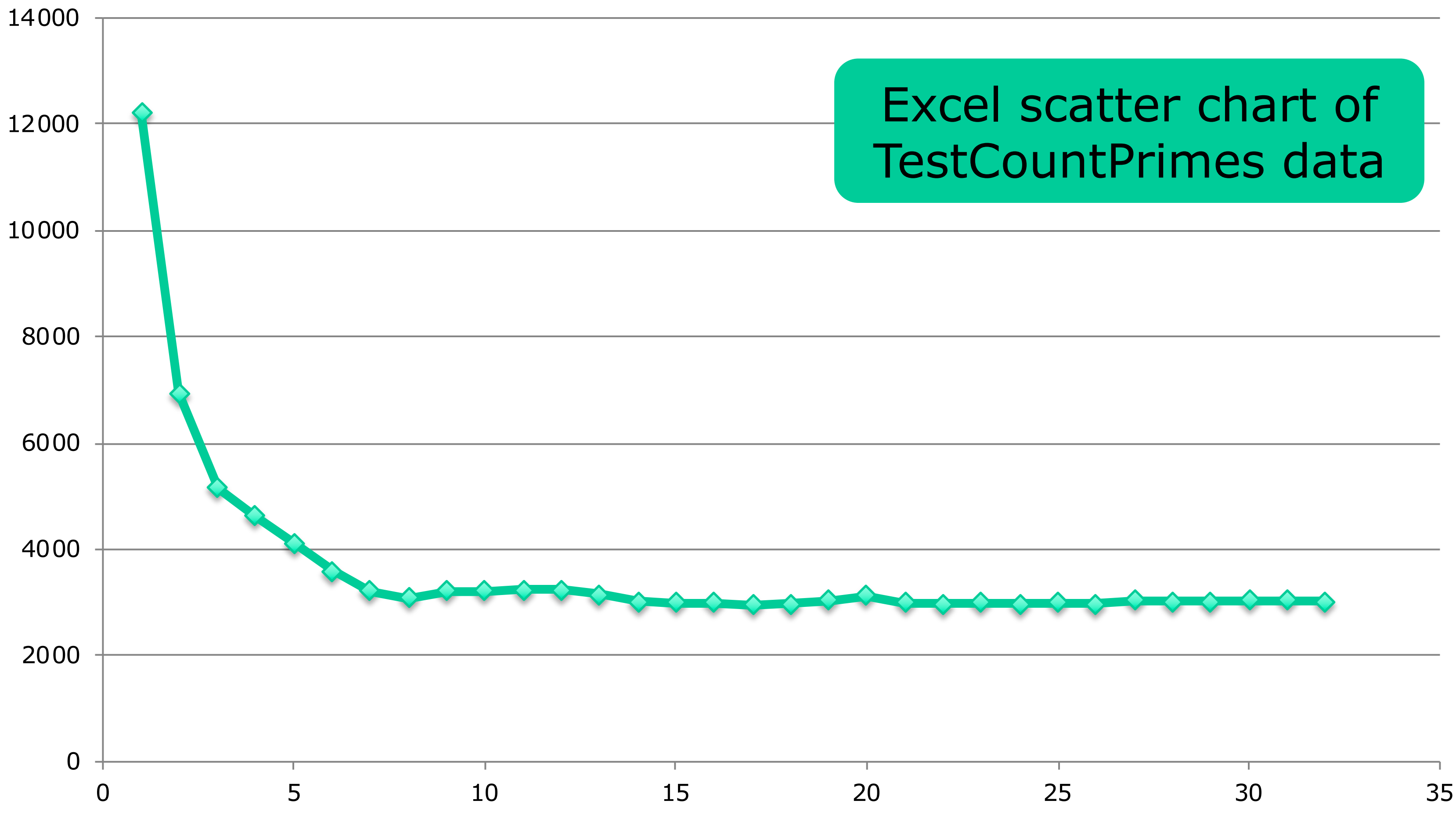
- One thread slower than sequential
- Max speedup 4.1x
- From some point, more threads are worse
- How choose best thread count?
- Tasks and executors are better than threads, week 5



Source: Wikipedia

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...

Wikipedia Latch

Timing threads à la Goetz & Bloch

```
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();
```

Await start

Do work

Signal end

Signal start

Await end

worker threads

main thd

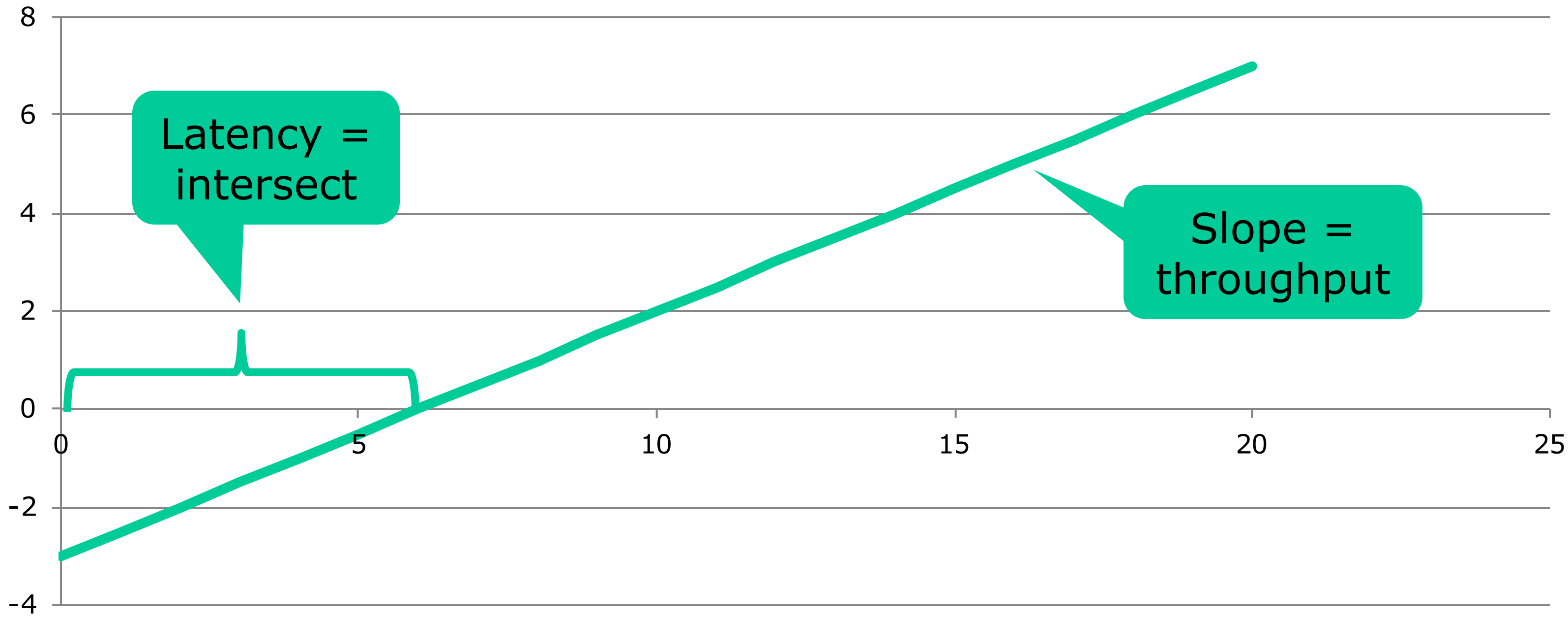
Goetz p. 96

See also Bloch p. 275

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

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

- Hand-in week 3
  - 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