Practical Concurrent and Parallel Programming VIII

Performance Measurements

Jørgen Staunstrup

Performance measurements

Make time consuming computations run faster e.g.:
- searching in large amounts of data
- simulation (weather forecasts)
- sorting large volumes of data
- …

Thread creation is "expensive" (time consuming) !!

```
Thread t= new Thread( ... );
```

Today: we will address how to measure computing times

How long does it take to create a thread?

Thread creation is "expensive" (time consuming) !!

```
Thread t= new Thread( ... );
```

But how expensive?

Assuming that a single floating-point multiplication takes **one** time unit on your PC

1: (Approximately) How many time units will it take to create and start a thread?

This week we focus on how to find out.

Agenda

- **Performance measurements: motivation and introduction**
- Pitfalls (and avoiding them)
- Measurements of thread overhead
- Algorithms for parallel computing
- Bonus: Calculating means and variance (efficiently)

Motivations for Concurrency

From Week01

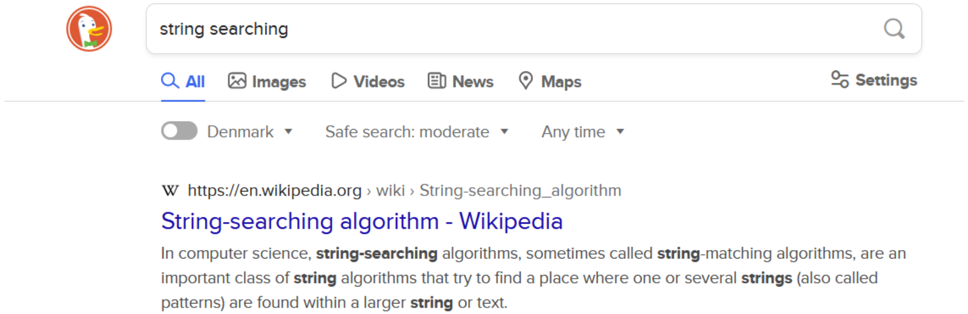
Inherent: User interfaces and other kinds of input/output

Exploitation: Hardware capable of simultaneously executing multiple streams of statements

Hidden: Enabling several programs to share some resources in a manner where each can act as if they had sole ownership

Motivation 1: Time consuming computations

Searching in a (large) text



https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/

Computing prime numbers

```
2, 3, 5, 7, 11, 13, 17, 19, 23,
29, 31, 37, 41, 43, 47, 53, 59,
61, 67, 71, 73, 79, 83, 89, 97,
...
```

Cornerstone of all computer security

https://science.howstuffworks.com/math-concepts/prime-numbers.htm

Motivation 2: Analyzing code

Thread creation is expensive ?

The Java tutorials say that creating a Thread is expensive. But why exactly is it expensive? What exactly is happening when a Java Thread is created that makes its creation expensive? I'm taking the statement as true, but I'm just interested in mechanics of Thread creation in JVM.

> *Thread lifecycle overhead. Thread creation and teardown are not free. The actual overhead*

But how expensive ?
~ 600 ns to create (on this laptop)
~ 20 times more time than creating a simple object

65000 ns to create and start a thread !!! (on this laptop)

Today: How to get such numbers !

(Performance) Measurements

Key in many sciences (experiments, observations, predictions, ...)

Computer science:
A bit of statistics
A bit of numerical analysis
A bit of computer architecture (cores, caches, number representation,)
Code for measuring execution time

Based on *Microbenchmarks in Java and C# by Peter Sestoft* (see benchmarkingNotes.pdf in material for this week)

All numbers in these slides were measured in August 2021 on a:

Intel Core i5-1035G4 CPU @ 1.10GHz, 4 Core(s), 8 Logical Processor(s)

Agenda

- Performance measurements: motivation and introduction
- **Pitfalls (and avoiding them)**
- Measurements of thread overhead
- Algorithms for parallel computing
- Bonus: Calculating means and variance (efficiently)

Example: measuring a (simple) function


```
private static int multiply(int i) {
  return i * i;
}
```

```
start= System.nanoTime();
multiply(126465);
end= System.nanoTime();

System.out.println(end-start+" ns");
```

Example: measuring a (simple) function

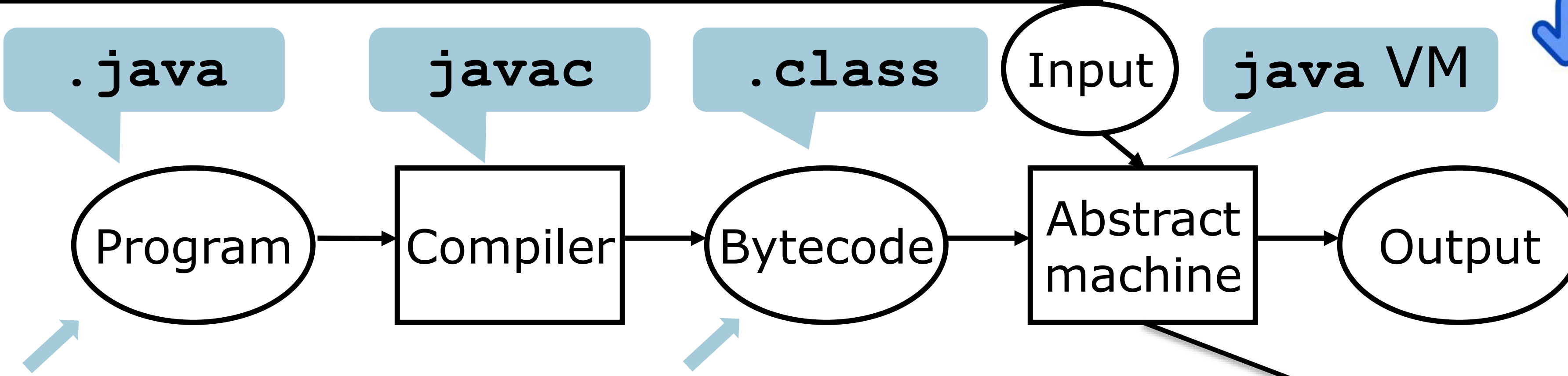

```
start= System.nanoTime();
multiply(126465);
end= System.nanoTime();

System.out.println(end-start+" ns");
```

My results: `3600 ns 1400 ns 1500 ns, …`

Should be: `~ 1-2 ns`

What is going on?

Java compiler and virtual machine




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

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

JVM

Compilation also happens at runtime

JIT (Just In Time)

```
19 xorl %ebx,%ebx
1b jmp 3a
       0x00(%ebp),%ebp
       0xec(%ebp)
       %ebx,0x0c(%edi)
26 jbe 49
2c leal
0x10(%edi,%ebx,8),%eax
…
```

x86

Benchmarking note

Microbenchmarks in Java and C#

Peter Sestoft (sestoft@itu.dk)

IT University of Copenhagen, Denmark

Version 0.8.0 of 2015-09-16

Abstract: Sometimes one wants to measure the speed of software, for instance, to measure whether a

A goldmine of good advice

Accompanying code: [Benchmark.java]

On PCPP GitHub (week08)

```
class Benchmark {
  public static void main(String[] args) { new Benchmark(); }

  public Benchmark() {
    // SystemInfo();A
    // Mark0();
    // Mark1();
    ...
    Mark6("multiply", i -> multiply(i));
    ...
    // SortingBenchmarks();
    ...
```

Our focus:

Mark functions

The Timer class (in Benchmark.java)

A simple Timer class for Java

Works on all platforms (Linux, MacOS, Windows)

```
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(); }
}
```

Example: measuring a simple function


```
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;
}
```

```
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() / count) * 1e9 ;
  System.out.printf("%6.1f ns%n", time);
  return dummy;
}
```

Example: measuring a simple function


```
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;
}
```

```
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() / count) * 1e9 ;
  System.out.printf("%6.1f ns%n", time);
  return dummy;
}
```

```
11.9 ns
```

Automating multiple runs (Mark3)

Results will usually vary

```
public static double Mark3() {
  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);
  }
  return dummy;
}
```

```
24.6 ns
24.6 ns
24.5 ns
24.6 ns
24.4 ns
24.3 ns
24.5 ns
24.4 ns
24.7 ns
24.6 ns
```

What is the running time?

What should you report as the result, when the observations are:

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 ?

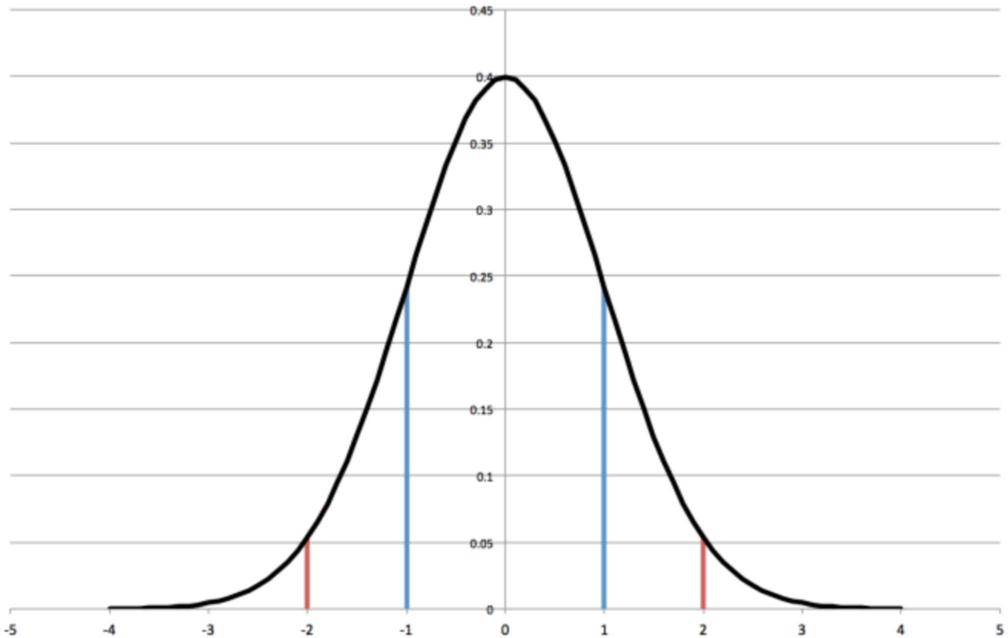
Mean: 30.4 ns

What if they are:

30.7 ns 100.2 ns 30.1 ns 30.7 ns 20.2 ns 30.4 ns 2.0 ns 30.3 ns 30.5 ns 5.4 ns ??

Mean: 31.0 ns ??

Normal distribution



Measuring physical properties

Your exam grades

Course evaluations

Fabrication faults

Running time of Java code

...

Standard deviation/variance

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

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

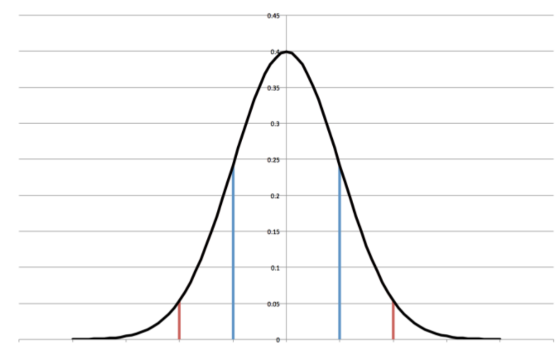
Mean

Standard deviation

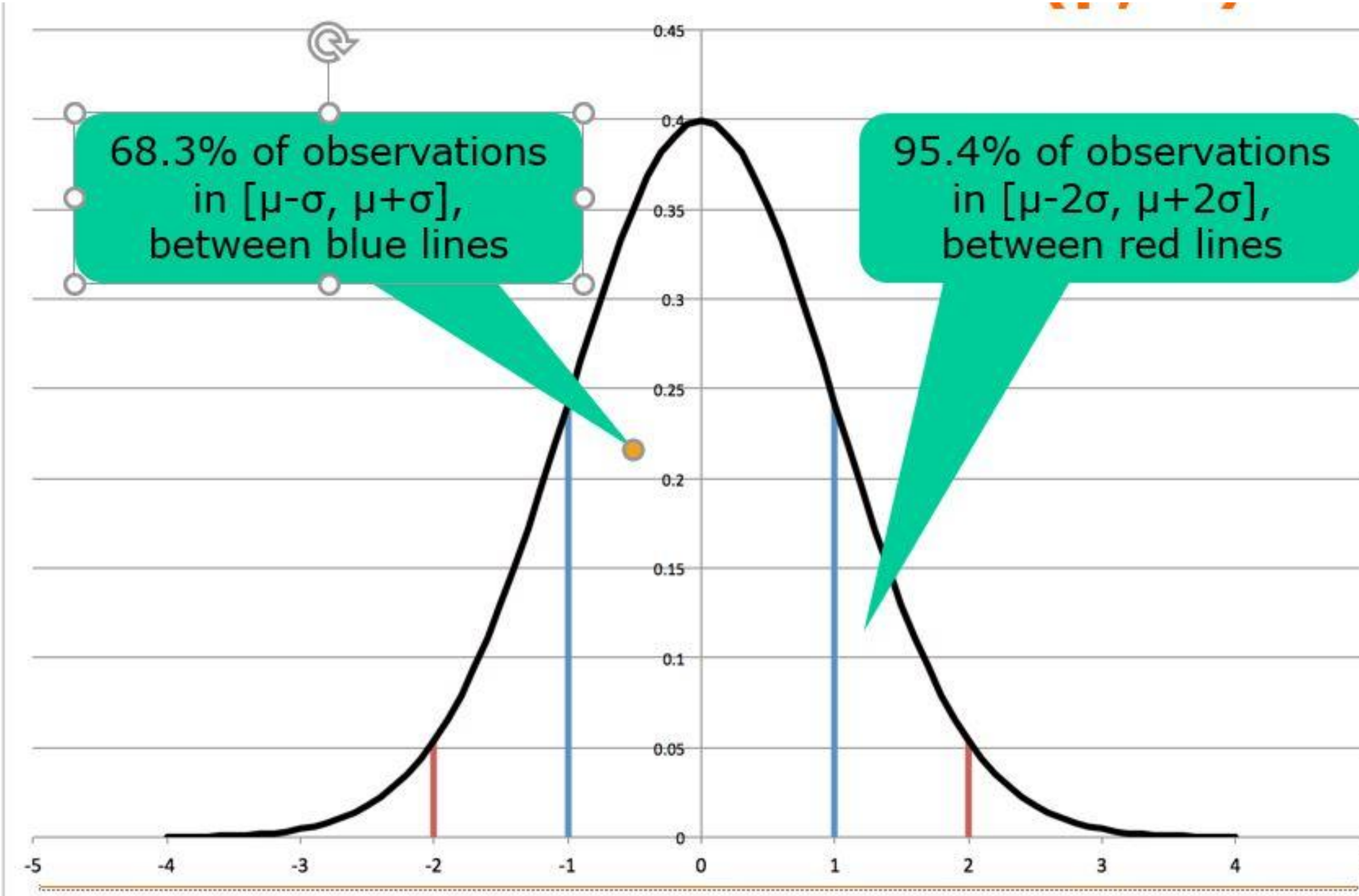
Benchmark note p6

30.7 ns 30.3 ns 30.1 ns 30.7 ns 50.2 ns 30.4 ns 30.9 ns 30.3 ns 30.5 ns 30.8 ns ??

Mean: 32.5 ns Standard deviation: 6.2



Normal distribution



Mark5 - computes mean and variance


```
public static double Mark5() {
  int n = 10, count = 1, totalCount = 0;
  double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
  do {
    count *= 2;
    st = 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));
    System.out.printf("%6.1f ns +/- %8.2f %10d%n", mean, sdev, count);
  } while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
  return dummy / totalCount;
}
```

Note:
two loops

Mark5 - computes mean and variance


```
public static double Mark5() {
  int n = 10, count = 1, totalCount = 0;
  double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
  do {
    count *= 2;
    st = 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));
    System.out.printf("%6.1f ns +/- %8.2f %10d%n", mean, sdev, count);
  } while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
  return dummy / totalCount;
}
```

Mark5

```
public static double Mark5() {
  int n = 10, count = 1, totalCount = 0;
  double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
  do {
    count *= 2;
    st = 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));
    System.out.printf("%6.1f ns +/- %8.2f %10d%n", mean, sdev, count);
  } while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
  return dummy / totalCount;
}
```

Parameterizing function to be measured


```
private static double multiply(int i) {
. . .
}
```

Java: `multiply(i)` **is a number**

Java: $\underbrace{\texttt{i -> multiply(i)}}$ **is a function**

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

```
Mark6( . . . , i -> multiply(i));
```

Mark6 - introduce a functional argument


```
public static double Mark6(String msg, IntToDoubleFunction f) {
  int n = 10, count = 1, totalCount = 0;
  double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
  do {
    count *= 2;
    st = sst = 0.0;
    for (int j=0; j<n; j++) {
      Timer t = new Timer();
      for (int i=0; i<count; i++) dummy += f.applyAsDouble(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));
    System.out.printf("%-25s %15.1f ns %10.2f %10d%n", msg, mean, sdev, count);
  } while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
  return dummy / totalCount;
}
public interface IntToDoubleFunction { double applyAsDouble(int i); }

Mark6("multiply", i -> multiply(i));
```

The function f is benchmarked

lambda

Example use of Mark6


```
Mark6("multiply", i -> multiply(i));
```

\# iterations

```
multiply                          595.0 ns    1407.81          2
multiply                          147.5 ns      90.10          4
multiply                          212.5 ns     152.53          8
multiply                          170.6 ns      59.44         16
multiply                          201.9 ns     157.69         32
multiply                           60.8 ns      34.55         64
multiply                           65.1 ns      59.83        128
multiply                           54.3 ns      14.85        256
...
multiply                           24.6 ns       0.75     524288
multiply                           24.6 ns       0.88    1048576
multiply                           24.9 ns       2.71    2097152
multiply                           24.3 ns       0.85    4194304
multiply                           24.2 ns       0.72    8388608
multiply                           25.0 ns       1.38   16777216
```

Mark7 - printing only final values


```
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)/(n-1));
  System.out.printf("%-25s %15.1f %10.2f %10d%n", msg, mean, sdev, count);
  return dummy / totalCount;
}
```

Agenda

- Performance measurements: motivation and introduction
- Pitfalls (and avoiding them)
- **Measurements of thread overhead**
- Algorithms for parallel computing
- Bonus: Calculating means and variance (efficiently)

Thread creation


```
Mark7("Thread create",
  i -> {
        Thread t= new Thread(() -> {
          for (int j= 0; j<1000; j++)
            ai.getAndIncrement();
        });
  return t.hashCode();  // to confuse compiler to not optimize
});
```

Takes 620 ns

2: What are we really measuring?

Thread creation


```
Mark7("Thread create",
  i -> {
        Thread t= new Thread(() -> {
          for (int j= 0; j<1000; j++)   // not executed
            ai.getAndIncrement();       // thread t created, but not started
        });
  return t.hashCode();    // to confuse compiler to not optimize
});
```

Takes 620 ns

Creating a thread/an object
Return statement

Measuring thread creation

```
Mark7("Thread create",
  i -> {
        Thread t= new Thread(() -> {
          for (int j= 0; j<1000; j++)  // not executed
            ai.getAndIncrement();      // thread t created, but not started
        });
    return t.hashCode();   // to confuse compiler to not optimize
});
```

Find out running times for:

1. creating an object
2. calculating hash-code
3. starting a thread

Creating an object

A thread is an object, so let us start finding the cost of creating a simple object.

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

Mark7("hashCode()", i -> myPoint.hashCode());

Mark7("Point creation",
      i -> {
        Point p= new Point(i, i);
        return p.hashCode();
      });
```

hashCode()   3 ns
Point creation 33 ns

So, object creation is: ~ 30 ns

Thread create + start


```
Mark7(" ... ",
      i -> {
        Thread t= new Thread(() -> {
          for (int j= 0; j<1000; j++)
            ai.getAndIncrement();
        });
        t.start();
        return t.hashCode();
      });
```

3: What are we really measuring?

Thread create + start


```
Mark7(" ... ",
      i -> {
        Thread t= new Thread(() -> {
          for (int j= 0; j<1000; j++)
            ai.getAndIncrement();
        });
        t.start();
        return t.hashCode();
      });
```

For loop not included!!!

Thread create + start


```
Mark7(" ... ",
      i -> {
        Thread t= new Thread(() -> {
          for (int j= 0; j<1000; j++)
            ai.getAndIncrement();
        });
        t.start();
        return t.hashCode();
      });
```

lambda returns right after start

Thread create + start

```
Mark7(" ... ",
      i -> {
        Thread t= new Thread(() -> {
          for (int j= 0; j<1000; j++)   //most iterations not done
            ai.getAndIncrement();      // Why?
        });
        t.start();
        return t.hashCode();
      });
```

Takes ~ 65000 ns

- So, a lot of work goes into starting a thread
- Even after creating it (takes ~ 600 ns)
- Note: does not include executing the for loop

Never create threads for small computations !!!

Agenda

- Performance measurements: motivation and introduction
- Pitfalls (and avoiding them)
- Measurements of thread overhead
- **Algorithms for parallel computing**
- Bonus: Calculating means and variance (efficiently)

Quicksort: https://www.chrislaux.com/quicksort.html

```
private static void qsort(int[] arr, int a, int b) {
  if (a < b) {
    int i = a, j = b;
    int x = arr[(i+j) / 2];
    do {
      while (arr[i] < x) i++;
      while (arr[j] > x) j--;
      if (i <= j) { swap(arr, i, j); i++; j--; }
    } while (i <= j);
    qsort(arr, a, j); qsort(arr, i, b);
  }
}
```

see SearchAndSort.java in week08 material

Prime counting: https://www.dcode.fr/prime-number-pi-count

```
long count = 0;
final int from = 0, to = range;
for (int i=from; i<to; i++) if (isPrime(i))  count++;
```

Multithreaded version of CountPrimes

2, 3, 4, 5, ……… range

thread0 thread1 threadN

Code for exercises week08: [TestCountPrimesThreads.java]

Java code for TestCountTimesThreads


```
private static long countParallelN(int range, int threadCount) {
  final int perThread= range / threadCount;
  final LongCounter lc= new PrimeCounter();
  Thread[] threads= new Thread[threadCount];
  for (int t=0; t<threadCount; t++) {
    final int from= perThread * t,
      to= (t+1==threadCount) ? range : perThread * (t+1);
      threads[t]= new Thread(()->
        {for (int i=from; i<to; i++)
            if (isPrime(i)) lc.increment();
        });
  }
  for (int t=0; t<threadCount; t++) threads[t].start();
  try { for (int t=0; t<threadCount; t++)  threads[t].join();
  } catch (InterruptedException exn) { }
  return lc.get();
}
```

Running times for prime counting


```
countParallel  1         7107236.6 ns  448417.55
countParallel  2         6069944.7 ns  802224.61
countParallel  3         3621185.5 ns  152693.03
countParallel  4         3124067.0 ns  640480.51
...
```

Good or bad ?

Number of primes are very different in these ranges !!!

2, 3, 4, 5, ... range

thread0 thread1 threadN

Breaking the task into smaller pieces/tasks

2, 3, 4, 5, ... range

When a thread is done with one task, it gets a new task

until all tasks are done

Tasks

[low..high]

```
when high-low > threshold
int mid= low+(high-low)/2;
```

```
else sequential count
```

[a..mid]
[mid+1..high]

Prime counter task (skeleton)

```
public class countPrimesTask implements Runnable {
  private final int low;
  private final int high;
  private final ExecutorService pool;

  @Override public void run() {

    int mid= low+(high-low
    pool.submit( new c        sTask(low, mid, pool) );
    pool.submit( ne        imesTask(mid+1, high, pool) );

  }
}
```

igh]

More on executors next week

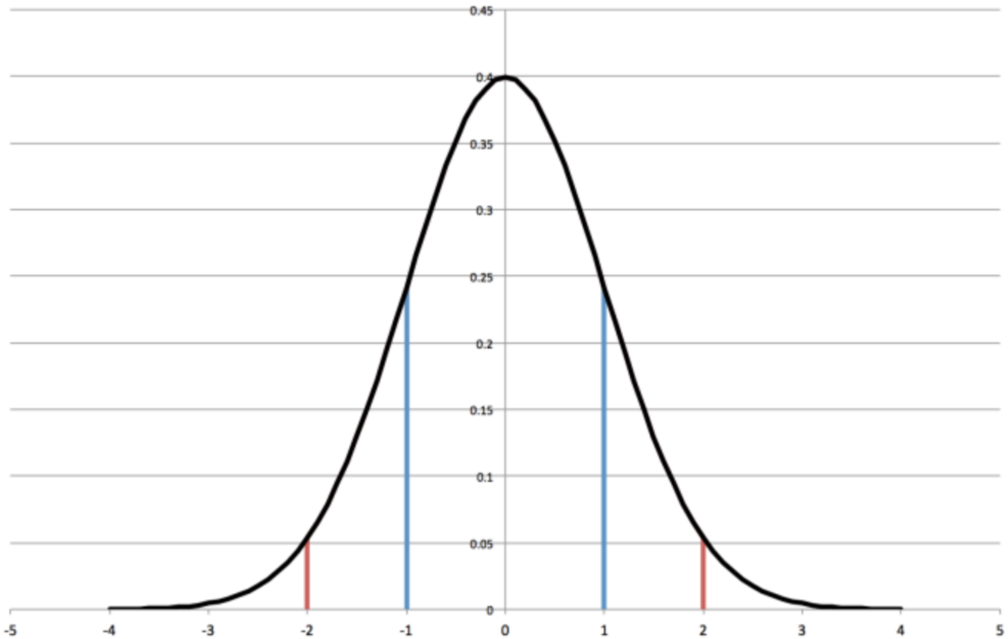
Shortcomings:

1. How to stop?
2. Will create too many "small" tasks
3. Returning result (# primes)

Agenda

- Performance measurements: motivation and introduction
- Pitfalls (and avoiding them)
- Measurements of thread overhead
- Algorithms for parallel computing
- **Bonus: Calculating means and variance (efficiently)**

Normal distribution



Measuring physical properties

Your exam grades

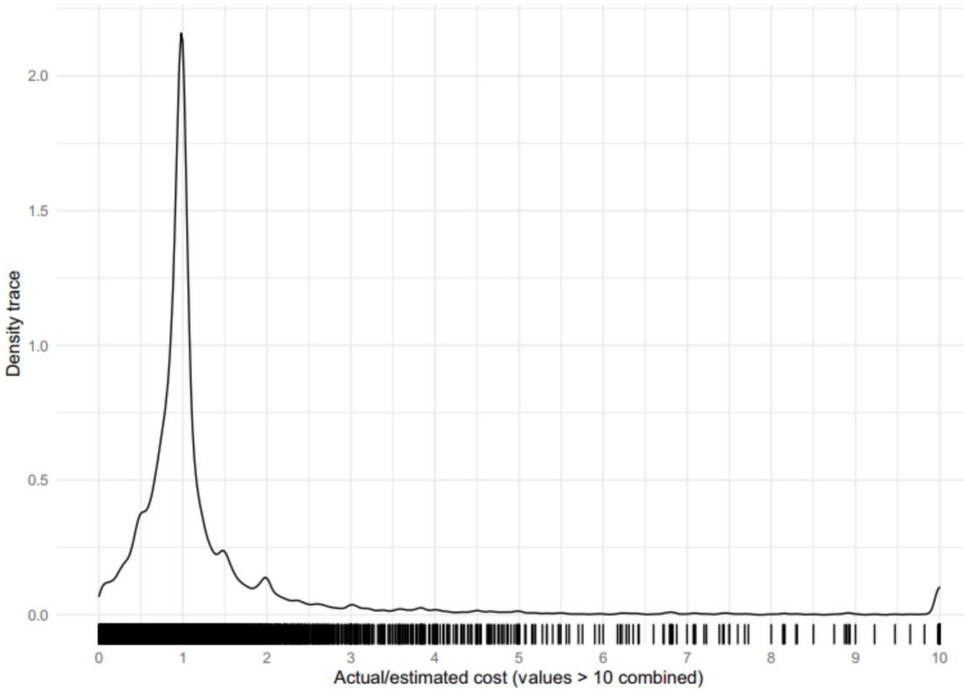
Course evaluations

Fabrication faults

Running time of Java code

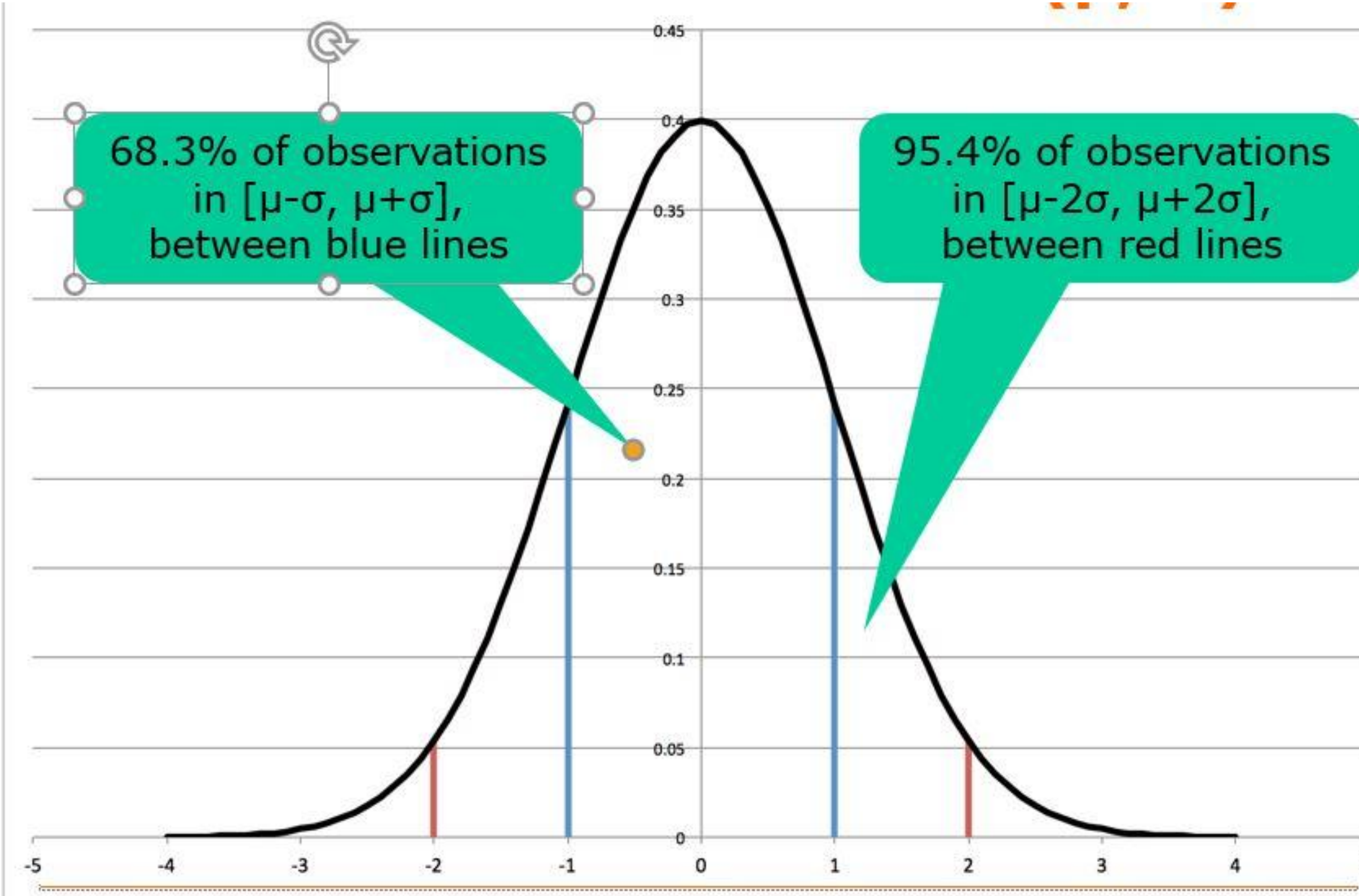
...

But there are exceptions



Source: Bent Flyvbjerg, Alexander Budzier, Jong Seok Lee, Mark Keil, Daniel Lunn & Dirk W. Bester (2022) The Empirical Reality of IT Project Cost Overruns: Discovering A Power-Law Distribution, Journal of Management Information Systems, 39:3, 607-639, DOI: 10.1080/07421222.2022.2096544

Normal distribution



Outliers

What should you report as the result, when the observations are:

30.7 ns 30.3 ns 30.1 ns 30.7 ns 50.2 ns 30.4 ns 30.9 ns 30.3 ns 30.5 ns 30.8 ns ??

Mean: 32.5 ns Standard deviation: 6.2

50.2 is an outlier

because there is a probability of less than 4.6 % that 50.2 is a valid observation

Computing the variance

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

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

Requires two passes through the data

$$\sigma^2 = \frac{1}{n(n-1)}\left(n\sum_{j=1}^{n} t_j^2 - \left(\frac{1}{n}\sum_{j=1}^{n} t_j\right)^2\right)$$

Can be done in one pass (on-line alg.)

```
for (int j=0; j<n; j++) {
   Timer t = new Timer();
   for (int i=0; i<count; 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 two formulas give the same result

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

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

Formula in Benchmark note

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

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

$$\sigma^2 = \frac{1}{n-1}\left(\sum_{j=1}^{n} t_j^2+\sum_{j=1}^{n}(\mu^2-2t_j\mu)\right)$$

$$\sigma^2 = \frac{1}{n-1}\left(\sum_{j=1}^{n} t_j^2+n\mu^2-2\mu\sum_{j=1}^{n} t_j\right)$$

See exercises08.pdf

$$\sigma^2 = \frac{1}{n-1}\left(\sum_{j=1}^{n} t_j^2+n\mu^2-2\mu n\mu\right)$$

$$\sigma^2 = \frac{1}{n-1}\left(\sum_{j=1}^{n} t_j^2-n\mu^2\right)$$

$$\sigma^2 = \frac{1}{n(n-1)}\left(n\sum_{j=1}^{n} t_j^2-\mu^2\right)$$

also https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

$$\sigma^2 = \frac{1}{n(n-1)}\left(n\sum_{j=1}^{n} t_j^2-\left(\frac{1}{n}\sum_{j=1}^{n} t_j\right)^2\right)$$

Formula used in code (one pass algorithm)

Warning

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

```
int n = 10;
...
for (int j=0; j<n; j++) {
  Timer t = new Timer();
  for (int i=0; i<count; 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);
```

Beware: sst - mean * mean * n   can be a very small number

Digit loss

Beware of cancellation when subtracting numbers that are close to each other:

28 significant digits

```
 1010101000010110110001110101.111
-1010101000010110110001110001.100
 ------------------------------
 0000000000000000000000000100.011
```

3 significant digits

https://blog.demofox.org/2017/11/21/floating-point-precision