Practical Concurrent and Parallel Programming IV

Jørgen Staunstrup

IT University of Copenhagen

Friday 2020-09-18

Starts at 8:00

Plan for today

- Performance measurements: motivation and introduction
- Calculating means and variance (pitfalls)
- Measurements of thread and lock overhead
    - Questions from you (cost of volatile? and cost of changing thread?)
- Algorithms for parallel computing

About Jørgen

Datalog/Computer scientist Aarhus University: 1975
Taught first course on Concurrency: 1978 (USC, Los Angeles)
It was an on-line course !!
Joined ITU in 2001, retired in 2014, Honoary professor since 2017
Teaching MSc course Mobile App Development at ITU since 2016

Hand-out material

- Peter Sestoft: Microbenchmarks in Java and C sharp
- Slides from lecture
- Exercises week 4
- Recording of lecture (uploaded after the class on 09-18)

Suplementary material

[Slides from lecture on Computer Arithmetic:
https://www.itu.dk/people/sestoft/bachelor/computernumbers.pdf](https://www.itu.dk/people/sestoft/bachelor/computernumbers.pdf)

Take a look at the last page with references (it is a goldmine!!)

Motivation (performance measurements)

Threads are expensive

How expensive ?

~600 ns to create (on this laptop)

~20 times more time than creating a simple object

40000 ns to start a thread !!!

Today: How to get such numbers !

(Performance) Measurements

- Key in many sciences (experiments, observations, predictions, …)
- A bit of statistics
- A bit of numerical analysis
- A bit of computer architecture (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 September 2020 on a:

Intel(R) Core(TM) i5-1035G4 CPU @ 1.10GHz, 1498 Mhz, 4 Core(s), 8 Logical Processor(s)

Example

How long does this method take to run?

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

- 20 floating-point multiplications
- 1 GHz processor ~ 1 ns/cycle
- So takes around 20 ns

2-3 integer operations/cycle (sometimes)

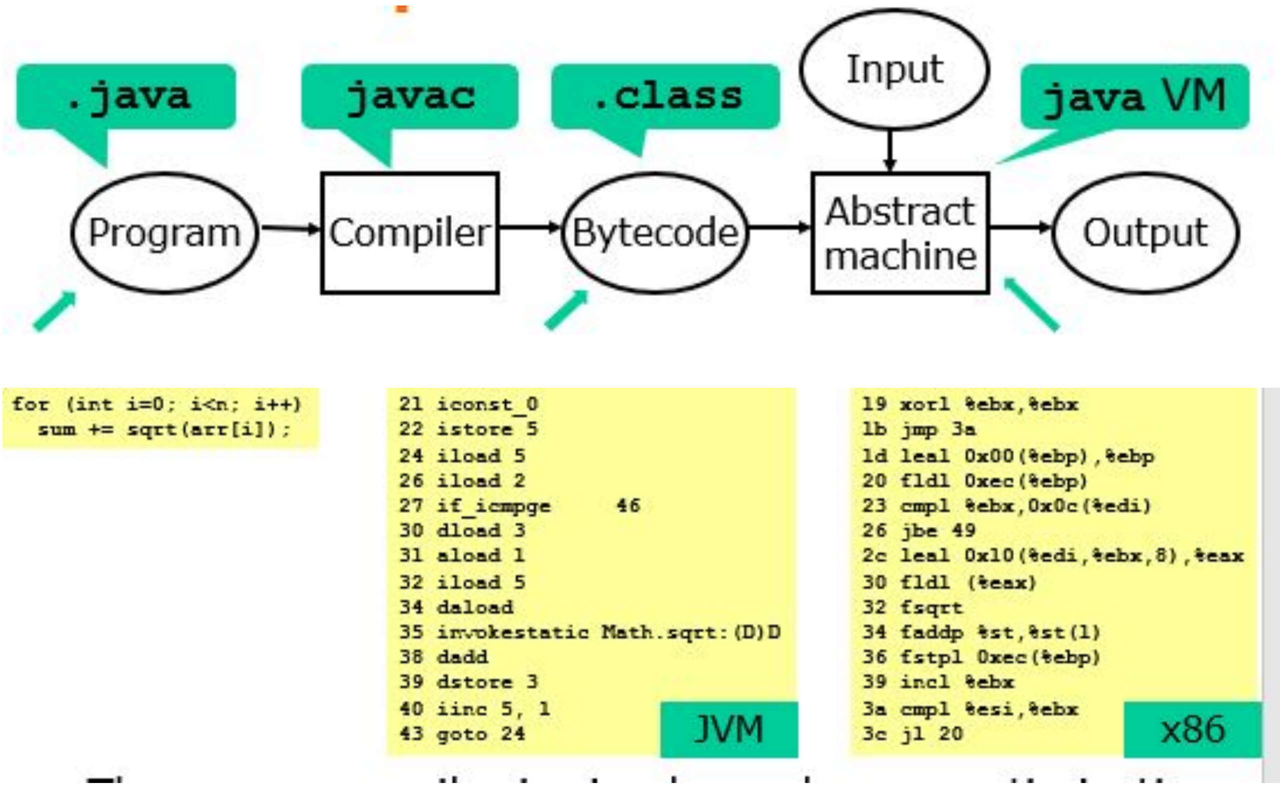
- Let us try to measure it:

```
start = System.nanoTime();
//calculation
spent= System.nanoTime()-start;
```

- 1.2 ns ???

JIT optimizes code and removes unused calculations !!!!

Java compilation and execution



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

"https://www.tutorialspoint.com/java_virtual_machine/java_virtual_machine_jit_compi
"

Alternative multiplication

```
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;
}
```

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(); }
}
```

Mark2 in BenchMark.java

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

Let us try it.

Automating multiple samples (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);
}
```

23.9 ns
23.9 ns
23.7 ns
23.9 ns
23.8 ns
23.7 ns
23.7 ns
23.7 ns
23.7 ns
23.7 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

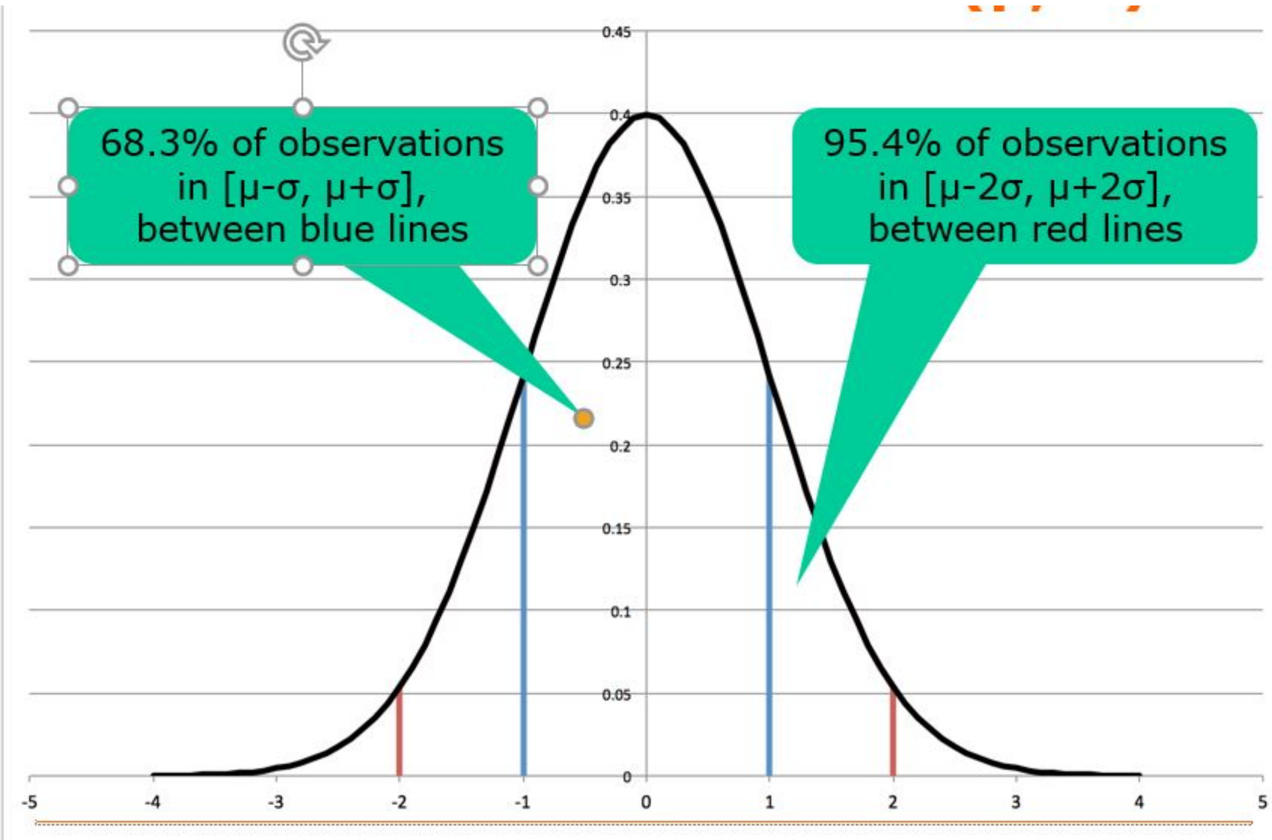
and variance

Variance

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

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

Normal Distribution



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 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 correct observation

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
```

Floating-point numbers

IEEE 754 binary32 and binary64
Which you know as `float` and `double` in Java (and C#)

IEEE 754 decimal128
Java's `java.math.BigDecimal`

The number 1.0527 can be witten as: 10527 / 10 ^ 4 (fraction)

representation is 10527 4

But not all numbers can be represesented precisely in a finite numer of bits !!!
1/10 = 0.000110011001100110011001100…

See Wikipedia page on accuracy problems

So we cannot represent 0.10 Kr. or $0.10 exactly

Nor 0.01 Kr. or $0.01 exactly

Do not use binary floating-point (float, double) for accounting!!!!

Digit loss

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

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

Therefore:

```
(sst - mean*mean)
```

can be problematic.

How to do it:
https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

Benchmark code (Mark6)

```
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;
}
```

see Benchmark.java (in code for this week)

Cost of Threads

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

```java
/** @author Brian Goetz and Tim Peierls */
class Point {
  public final int x, y;
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Benchmark code:

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

```
hashCode()        2.6 ns       0.03
Point creation   33.5 ns       0.35
```

So object creation is: ~ 31 ns

Thread creation

```
Mark6("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();
      });
```

Takes 600 ns ~ 20 times slower than Point creation

Thread create + start

```
Mark6("Thread create start",
      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 ~ 40000 ns

- So a lot of work goes into setting up a thread
- Even after creating it
- Note: does not include executing the loop (why?)

Never create threads for small computations

See more examples in BenchMark.java

Question from week 3


```java
Mark8("Thread yield", i -> { Thread t = new Thread(() -> {
              for (int j=0; j<100000; j++) Thread.yield();
              });
            t.start();
            try { t.join();  }   catch (InterruptedException exn) { }
            return t.hashCode();
          });
```

Also tried the same with and extra Thread t2 to see if `Thread.yield();` was optimized away. Apparently, it was not.

Result:

```
Thread yield                        102.2 ns       6.27          2
Thread yield                        100.6 ns       1.95          4
Thread yield                        100.8 ns       1.26          8
Thread yield                        100.6 ns       1.23         16
Thread yield                        100.8 ns       0.79         32
Thread yield                        100.4 ns       0.75         64
Thread yield                        102.9 ns       2.65        128
```

Algorithms for parallel computing

[Quicksort]

Problem: The first iteration through ALL the data (O(n)) only activates one thread, while the rest are waiting.

The second iteration only activates two threads etc.

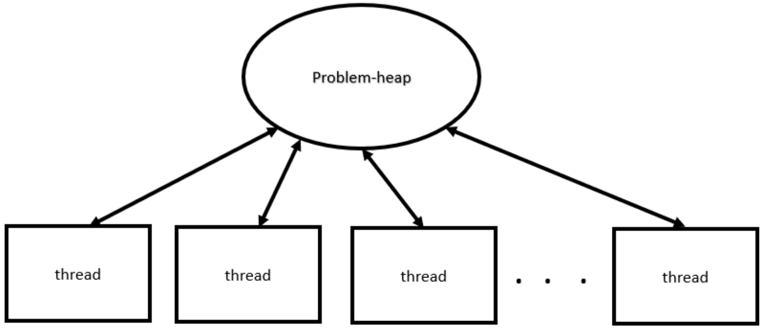
The resulting speed-up using 8 cores has been reported as 3.6.

[CountPrimes]

Problem: Computing the prime factors of small numbers much faster than for large numbers.

Work balancing is IMPORTANT.

Problem-heap




```
new Thread( () -> {
    while (newProblem= heap.getProblem())  solveProblem(nextProblem, heap);
});
```

```
public static void solveProblem(Problem p, ProblemHeap h, LongCounter lc) {
    int pc= 0; // no of primes
    if ((p.high - p.low) > threshold) { //generate new problem
    int d= (p.low+p.high)/2;
        h.add(new Problem(p.arr, p.low, d));
        h.add(new Problem(p.arr, d+1, p.high));
    } else {
      for (int i=p.low; i<=p.high; i++) if (isPrime(i)) pc= pc+1;
      lc.incr(pc); // must be atomic
    }
}
```

References

*Gelernter, David. "Generative communication in Linda". ACM Transactions on Programming Languages and Systems, volume 7, number 1, January 1985

*Møller-Nielsen, P and Staunstrup, J, Problem-heap. A paradigm for multiprocessor algorithms. Parallel Computing, 4:63-74, 1987

- Javaspaces

Problem-heap sorting performance

Problem-heap count primes performance

Reading and exercises for Week 4