https://gitter.im/ScalaTaiwan/ScalaTaiwan

Scala & Spark(1.6) in Performance Aspect

Jimin Hsieh
https://tw.linkedin.com/in/jiminhsieh

2016/06/14 @ Scala Taiwan Meetup

Who am I?

- Server-side engineer who develops anything except UI and loves performance tuning, FP, OOP, and Linux.
- Recently, I am doing data processing with Spark Scala.
- Experience:
  - QA (Computer Network)
  - Embedded Application Engineer (VoIP)
  - Server-side Engineer

Agenda

- How to Performance tuning?
- Scala Performance
- Why Scala?(*)
- Spark(1.6) Scala Performance

How to performance tuning?

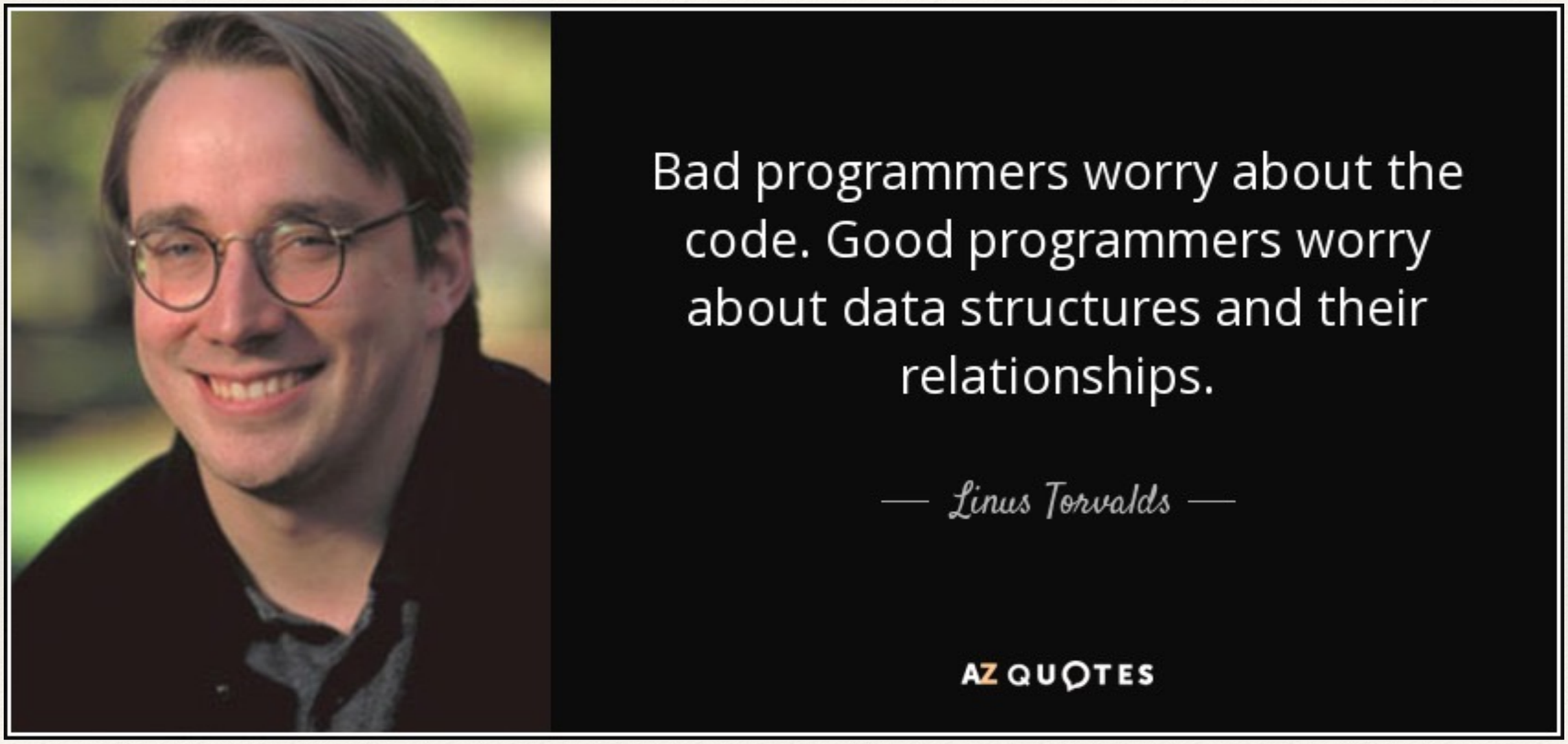
- The only way to become a better programmer is by not programming. Code is important, but it's a small part of the overall process. To truly become a better programmer, you have to to cultivate passion for everything else that goes on around the programming. From How To Become a Better Programmer by Not Programming by Jeff Atwood, co-founder of Stack Overflow

- Experiment + Know it through and through + Patience

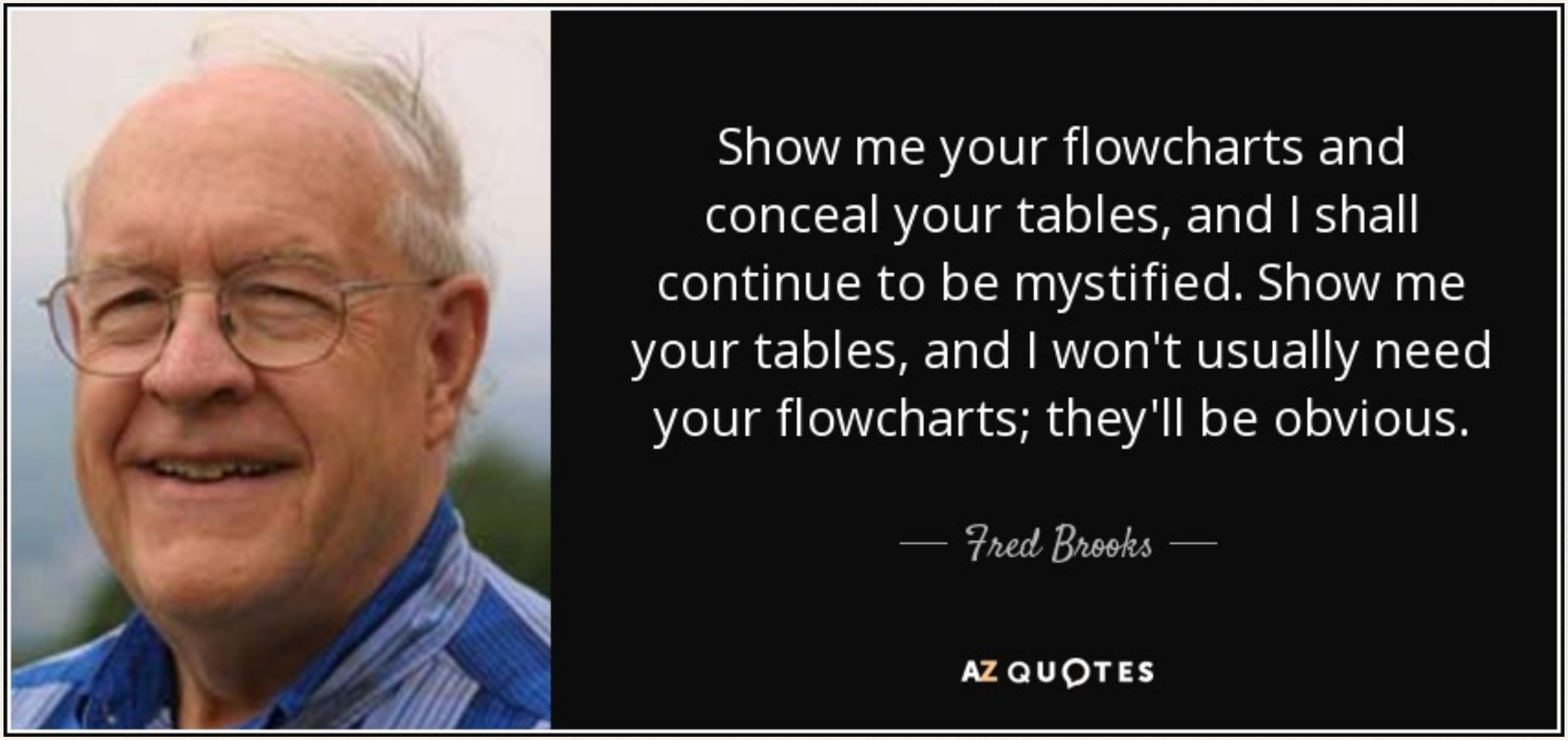
Scala Performance

- Collection Performance
  - immutable(Persistent data structure) vs mutable
- Loop vs Recursion vs Combinators
- Primitive type vs Boxed primitive
- String size

Collection Performance



Collection Performance



Collection Performance - immutable sequence

	head	tail	apply	update	prepend	append	insert
List	**C**	**C**	L	L	C	**L**	
Stream	**C**	**C**	L	L	C	**L**	
Vector	eC	eC	eC	eC	eC	eC	
Stack	C	C	L	L	C	C	L
Queue	aC	aC	L	L	L	C	
Range	C	C	C				
String	C	L	C	L	L	L	

Collection Performance - mutable sequence

	head	tail	apply	update	prepend	append	insert
ArrayBuffer	C	L	C	C	**L**	aC	L
ListBuffer	C	L	L	L	**C**	**C**	L
StringBuilder	C	L	C	C	**L**	aC	L
MutableList	C	L	L	L	C	C	L
Queue	C	L	L	L	C	C	L
ArraySeq	C	L	C	C			
Stack	C	L	L	L	C	L	L
ArrayStack	C	L	C	C	aC	L	L
Array	C	L	C	C			

Collection Performance - set & map

	lookup	add	remove	min
immutable				
HashSet/HashMap	eC	eC	eC	L
TreeSet/TreeMap	Log	Log	Log	Log
BitSet	C	L	L	eC*
ListMap	L	L	L	L
mutable				
HashSet/HashMap	eC	eC	eC	L
WeakHashMap	eC	eC	eC	L
BitSet	C	aC	C	eC
TreeSet	Log	Log	Log	Log

How to analyze Scala?

- Scala will be compiled to bytecode.

Bytecode

```
private Lscala/Option; checkpointData

// access flags 0x82
private transient Z org$apache$spark$rdd$RDD$$doCheckpointCalled

// access flags 0x82
private transient Lorg/slf4j/Logger; org$apache$spark$Logging$$log_

// access flags 0x9
// signature <K:Ljava/lang/Object;V:Ljava/lang/Object;>(Lorg/apache/spark/rdd/RDD<Lscala/Tuple2<TK;TV;>;>;)Lscala/runtime/Null$;
// declaration: scala.runtime.Null$ rddToPairRDDFunctions$default$4<K, V>(org.apache.spark.rdd.RDD<scala.Tuple2<K, V>>)
public static rddToPairRDDFunctions$default$4(Lorg/apache/spark/rdd/RDD;)Lscala/runtime/Null$;
  GETSTATIC org/apache/spark/rdd/RDD$.MODULE$ : Lorg/apache/spark/rdd/RDD$;
  ALOAD 0
  INVOKEVIRTUAL org/apache/spark/rdd/RDD$.rddToPairRDDFunctions$default$4 (Lorg/apache/spark/rdd/RDD;)Lscala/runtime/Null$;
  ARETURN
  MAXSTACK = 2
  MAXLOCALS = 1

// access flags 0x9
// signature <T:Ljava/lang/Object;>(Lorg/apache/spark/rdd/RDD<TT;>;Lscala/math/Numeric<TT;>;)Lorg/apache/spark/rdd/DoubleRDDFunctions;
// declaration: org.apache.spark.rdd.DoubleRDDFunctions numericRDDToDoubleRDDFunctions<T>(org.apache.spark.rdd.RDD<T>, scala.math.Numeric<T>)
public static numericRDDToDoubleRDDFunctions(Lorg/apache/spark/rdd/RDD;Lscala/math/Numeric;)Lorg/apache/spark/rdd/DoubleRDDFunctions;
  GETSTATIC org/apache/spark/rdd/RDD$.MODULE$ : Lorg/apache/spark/rdd/RDD$;
  ALOAD 0
  ALOAD 1
  INVOKEVIRTUAL org/apache/spark/rdd/RDD$.numericRDDToDoubleRDDFunctions (Lorg/apache/spark/rdd/RDD;Lscala/math/Numeric;)Lorg/apache/spark/rdd/DoubleRDDFunctions;
  ARETURN
  MAXSTACK = 3
  MAXLOCALS = 2

// access flags 0x9
// signature (Lorg/apache/spark/rdd/RDD<Ljava/lang/Object;>;)Lorg/apache/spark/rdd/DoubleRDDFunctions;
// declaration: org.apache.spark.rdd.DoubleRDDFunctions doubleRDDToDoubleRDDFunctions(org.apache.spark.rdd.RDD<java.lang.Object>)
public static doubleRDDToDoubleRDDFunctions(Lorg/apache/spark/rdd/RDD;)Lorg/apache/spark/rdd/DoubleRDDFunctions;
  GETSTATIC org/apache/spark/rdd/RDD$.MODULE$ : Lorg/apache/spark/rdd/RDD$;
  ALOAD 0
  INVOKEVIRTUAL org/apache/spark/rdd/RDD$.doubleRDDToDoubleRDDFunctions (Lorg/apache/spark/rdd/RDD;)Lorg/apache/spark/rdd/DoubleRDDFunctions;
  ARETURN
  MAXSTACK = 2
  MAXLOCALS = 1
```

Bytecode

```
private Lscala/Option; checkpointData

// access flags 0x82
private transient Z org$apache$spark$rdd$RDD$$doC

// access flags 0x82
private transient Lorg/slf4j/Logger; org$apache$s

// access flags 0x9
// signature <K:Ljava/lang/Object;V:Ljava/lang/Ob
// declaration: scala.runtime.Null$ rddToPairRDDF
public static rddToPairRDDFunctions$default$4(Lor
  GETSTATIC org/apache/spark/rdd/RDD$.MODULE$ : L
  ALOAD 0
  INVOKEVIRTUAL org/apache/spark/rdd/RDD$.rddToPa
  ARETURN
  MAXSTACK = 2
  MAXLOCALS = 1

// access flags 0x9
// signature <T:Ljava/lang/Object;>(Lorg/apache/s
// declaration: org.apache.spark.rdd.DoubleRDDFun
public static numericRDDToDoubleRDDFunctions(Lorg
  GETSTATIC org/apache/spark/rdd/RDD$.MODULE$ : L
  ALOAD 0
  ALOAD 1
  INVOKEVIRTUAL org/apache/spark/rdd/RDD$.numeric
  ARETURN
  MAXSTACK = 3
  MAXLOCALS = 2

// access flags 0x9
// signature (Lorg/apache/spark/rdd/RDD<Ljava/lan
// declaration: org.apache.spark.rdd.DoubleRDDFun
public static doubleRDDToDoubleRDDFunctions(Lorg/
  GETSTATIC org/apache/spark/rdd/RDD$.MODULE$ : L
  ALOAD 0
  INVOKEVIRTUAL org/apache/spark/rdd/RDD$.doubleR
  ARETURN
  MAXSTACK = 2
  MAXLOCALS = 1
```

a/runtime/Null$;
K, V>>)

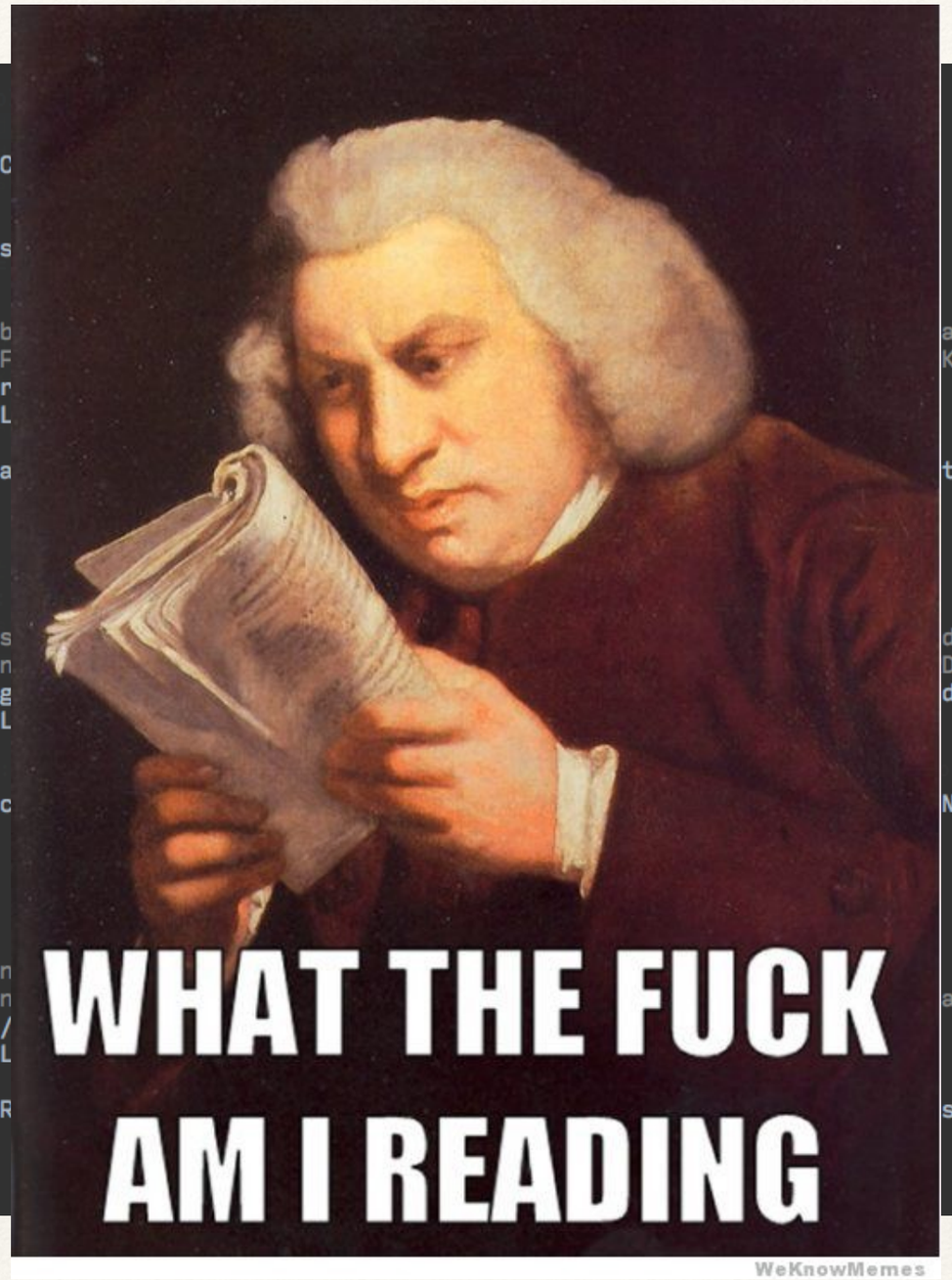
time/Null$;

dd/DoubleRDDFunctions;
DD<T>, scala.math.Numeric<T>)
d/DoubleRDDFunctions;

Numeric;)Lorg/apache/spark/rdd/DoubleRDDFunctions;

ava.lang.Object>)

spark/rdd/DoubleRDDFunctions;



Reverse Engineering

- scalac -Xprint:all {source code}
- Java Decompiler
- Luyten
- But Java bytecode is still the best way to know under the hood.

combinator vs recursion vs iteration

- There is no free lunch!

How to benchmark?

- sbt-jmh
- Benchmark Environment
  - Debian 8
  - OpenJDK 7
  - Scala 2.10.5

Loop2

```
[info] Benchmark                Mode  Cnt     Score    Error   Units
[info] Loop2.combinator        thrpt  400    58.372 ±  0.173  ops/ms
[info] Loop2.forLoop           thrpt  400    77.289 ±  0.992  ops/ms
[info] Loop2.tailCall          thrpt  400   168.048 ±  5.360  ops/ms
[info] Loop2.whileLoop         thrpt  400   202.866 ±  4.149  ops/ms
[info] Loop2.combinator          avgt  400     0.017 ±  0.001   ms/op
[info] Loop2.forLoop             avgt  400     0.013 ±  0.001   ms/op
[info] Loop2.tailCall            avgt  400     0.006 ±  0.001   ms/op
[info] Loop2.whileLoop           avgt  400     0.006 ±  0.001   ms/op
[info] Loop2.combinator            ss  400     0.046 ±  0.009   ms/op
[info] Loop2.forLoop               ss  400     0.038 ±  0.004   ms/op
[info] Loop2.tailCall              ss  400     0.067 ±  0.007   ms/op
[info] Loop2.whileLoop             ss  400     0.067 ±  0.007   ms/op
```

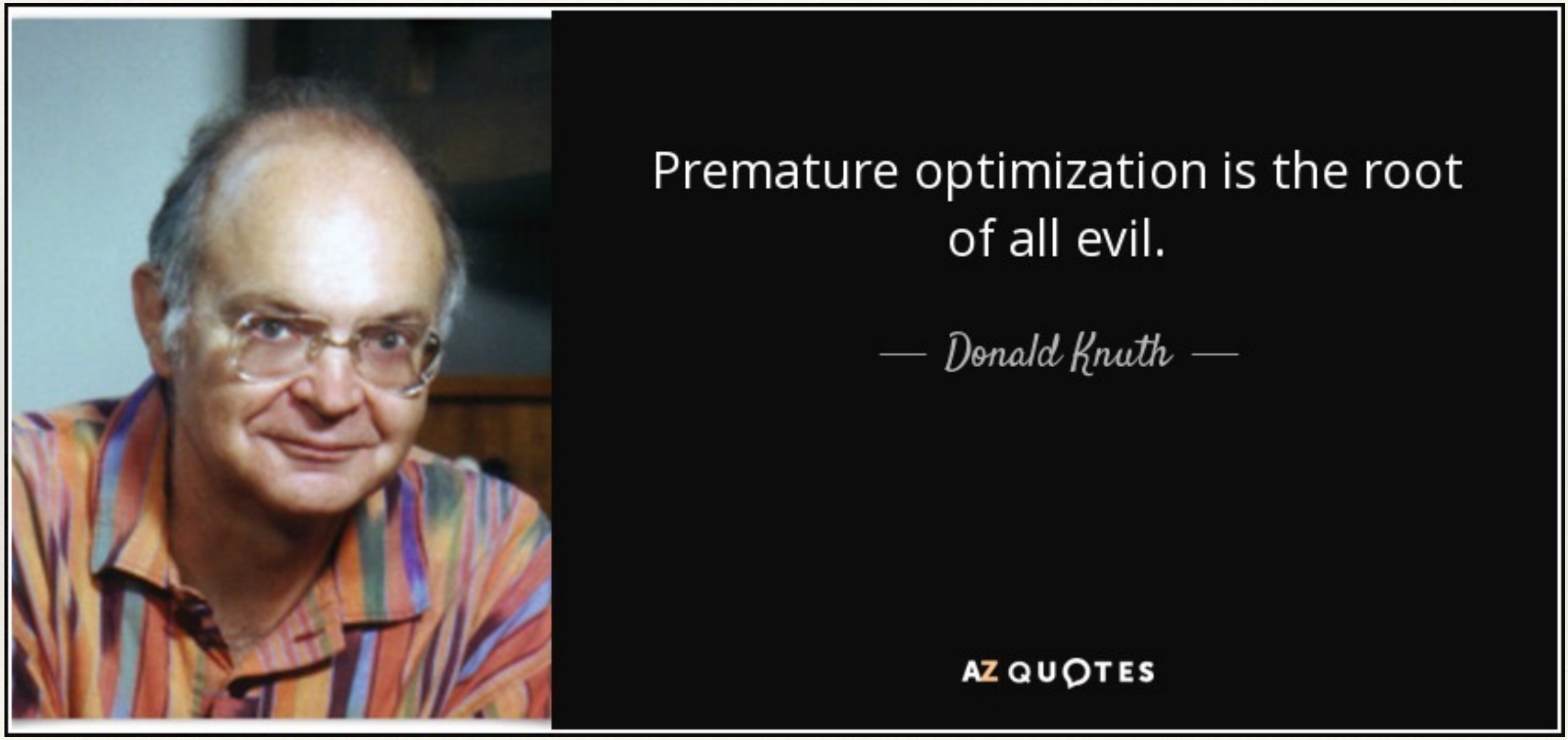
Loop3

```
[info] Benchmark                          Mode  Cnt   Score    Error   Units
[info] Loop3.combinator                  thrpt  400  57.734 ±  0.216  ops/ms
[info] Loop3.forLoopListBuffer           thrpt  400  70.846 ±  0.363  ops/ms
[info] Loop3.tailCallIfElse              thrpt  400  73.723 ±  0.190  ops/ms
[info] Loop3.tailCallPatternMatch        thrpt  400  79.851 ±  0.393  ops/ms
[info] Loop3.combinator                   avgt  400   0.017 ±  0.001   ms/op
[info] Loop3.forLoopListBuffer            avgt  400   0.014 ±  0.001   ms/op
[info] Loop3.tailCallIfElse               avgt  400   0.014 ±  0.001   ms/op
[info] Loop3.tailCallPatternMatch         avgt  400   0.013 ±  0.001   ms/op
[info] Loop3.combinator                     ss  400   0.039 ±  0.001   ms/op
[info] Loop3.forLoopListBuffer              ss  400   0.030 ±  0.001   ms/op
[info] Loop3.tailCallIfElse                 ss  400   0.033 ±  0.001   ms/op
[info] Loop3.tailCallPatternMatch           ss  400   0.029 ±  0.001   ms/op
```

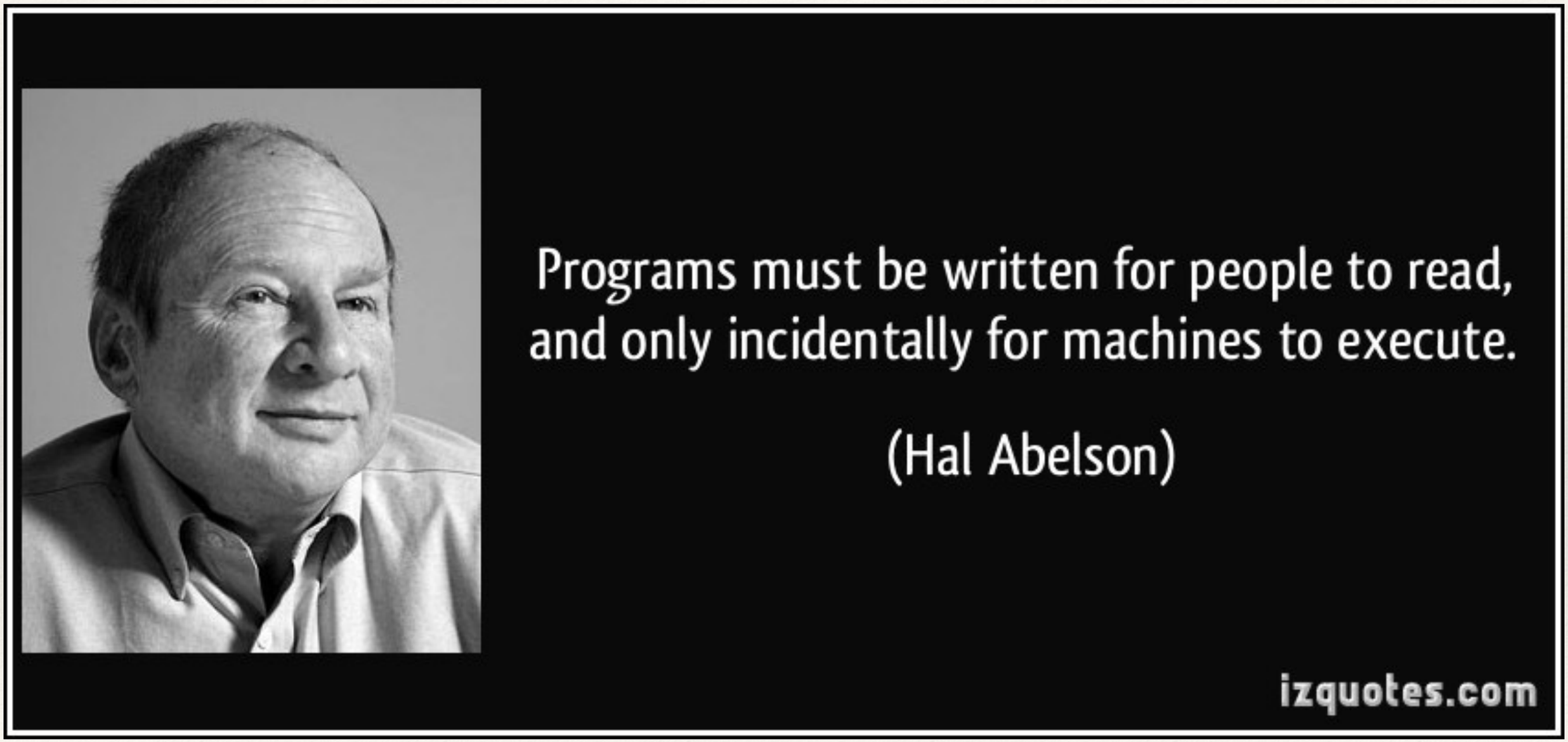
Loop4

```
[info] Benchmark                Mode  Cnt    Score     Error   Units
[info] Loop4.combinator        thrpt  400    5.026 ±   0.019  ops/ms
[info] Loop4.tailCall          thrpt  400   86.751 ±   0.400  ops/ms
[info] Loop4.whileLoop         thrpt  400   77.319 ±   2.406  ops/ms
[info] Loop4.combinator         avgt  400    0.199 ±   0.001   ms/op
[info] Loop4.tailCall           avgt  400    0.012 ±   0.001   ms/op
[info] Loop4.whileLoop          avgt  400    0.014 ±   0.001   ms/op
[info] Loop4.combinator           ss  400    1.154 ±   0.106   ms/op
[info] Loop4.tailCall             ss  400    0.037 ±   0.007   ms/op
[info] Loop4.whileLoop            ss  400    0.071 ±   0.014   ms/op
```

combinator vs recursion vs loop



combinator vs recursion vs loop



combinator vs recursion vs loop

- Consider using combinators first.
- If this becomes too tedious, or efficiency is a big concern, fall back on tail-recursive functions.
- Loop can be used in simple case, or when the computation inherently modifies state.
- from Martin Odersky: Scala with Style

loop vs recursion vs combinators

- Summary:
  - Choice the way has better readability and it's easier to reason.
- Based on my summary, it seems my talk is useless.
  - No.
    - Choice hot spot to optimise!
  - Yes.
    - The most important thing is Scala compiler/JVM will evolve.

Loop5

```
[info] Benchmark                  Mode  Cnt       Score      Error   Units
[info] Loop5.forLoopDouble       thrpt  400      ≈ 10⁻⁴                ops/us
[info] Loop5.forLoopFloat        thrpt  400      ≈ 10⁻⁴                ops/us
[info] Loop5.forLoopInt          thrpt  400      ≈ 10⁻⁴                ops/us
[info] Loop5.forLoopLong         thrpt  400      ≈ 10⁻⁴                ops/us
[info] Loop5.whileLoopDouble     thrpt  400      ≈ 10⁻⁴                ops/us
[info] Loop5.whileLoopFloat      thrpt  400      ≈ 10⁻⁴                ops/us
[info] Loop5.whileLoopInt        thrpt  400      ≈ 10⁻⁴                ops/us
[info] Loop5.whileLoopLong       thrpt  400      ≈ 10⁻⁴                ops/us
[info] Loop5.forLoopDouble        avgt  400  20889.321 ±  54.857   us/op
[info] Loop5.forLoopFloat         avgt  400  12479.768 ± 295.988   us/op
[info] Loop5.forLoopInt           avgt  400   9532.830 ±  17.743   us/op
[info] Loop5.forLoopLong          avgt  400  20062.058 ±  13.270   us/op
[info] Loop5.whileLoopDouble      avgt  400  20222.500 ± 102.163   us/op
[info] Loop5.whileLoopFloat       avgt  400  11553.178 ± 274.842   us/op
[info] Loop5.whileLoopInt         avgt  400   9488.849 ±   7.718   us/op
[info] Loop5.whileLoopLong        avgt  400  19047.508 ±   4.433   us/op
[info] Loop5.forLoopDouble          ss  400  20959.765 ±  46.540   us/op
[info] Loop5.forLoopFloat           ss  400  11780.323 ± 106.223   us/op
[info] Loop5.forLoopInt             ss  400   9586.689 ±  31.832   us/op
[info] Loop5.forLoopLong            ss  400  19077.921 ±  28.002   us/op
[info] Loop5.whileLoopDouble        ss  400  20046.600 ±   5.705   us/op
[info] Loop5.whileLoopFloat         ss  400  10731.143 ±   9.356   us/op
[info] Loop5.whileLoopInt           ss  400   9513.217 ±   8.281   us/op
[info] Loop5.whileLoopLong          ss  400  19078.069 ±  12.329   us/op
```

Loop5 - Scala vs Java

	Scala(us/op)	Java(us/op)
for-loop-int	9532.83	9489.191
while-loop-int	9488.849	9491.391
for-loop-long	20062.058	18991.304
while-loop-long	19047.508	18988.561
for-loop-float	12479.768	11365.767
while-loop-float	11553.178	11198.945
for-loop-double	20889.321	19798.314
while-loop-double	20222.5	19772.461

Loop5 - Scala vs Java

```scala
def forLoopInt(state:
BenchmarkState) = {
  var s = 0
  for (i <- 1 to state.n;
       j <- 1 to state.n;
       k <- 1 to state.n)
    s += state.random.nextInt(4)
  s
}
```

```java
public int forLoopInt(BenchmarkStateLoop5Java
state) {
    int s = 0;
    for (int i = 1; i <= state.n; i++)
        for (int j = 1; j <= state.n; j++)
            for (int k = 1; k <= state.n; k++)
                s += state.ranom.nextInt(4);

    return s;
}
```

Loop5 - Scala vs Java

- Scala nested for-loop is just slightly slower than Scala while-loop.
- Scala while-loop is pretty close to Java for-loop and Java while-loop.

primitive type vs boxed primitive

```
[info] Specialization.tuple2LoopDouble          thrpt  400     0.006 ±  0.001  ops/ms
[info] Specialization.tuple2LoopInt             thrpt  400     0.006 ±  0.001  ops/ms
[info] Specialization.tuple2SpeicalLoopDouble   thrpt  400     0.007 ±  0.001  ops/ms
[info] Specialization.tuple2SpeicalLoopInt      thrpt  400     0.008 ±  0.001  ops/ms
[info] Specialization.tuple2LoopDouble           avgt  400   152.308 ±  1.626   ms/op
[info] Specialization.tuple2LoopInt              avgt  400   154.144 ±  1.676   ms/op
[info] Specialization.tuple2SpeicalLoopDouble    avgt  400   155.701 ±  2.723   ms/op
[info] Specialization.tuple2SpeicalLoopInt       avgt  400   133.234 ±  1.106   ms/op
[info] Specialization.tuple2LoopDouble             ss  400   155.689 ±  2.899   ms/op
[info] Specialization.tuple2LoopInt                ss  400   155.466 ±  1.683   ms/op
[info] Specialization.tuple2SpeicalLoopDouble      ss  400   156.365 ±  3.034   ms/op
[info] Specialization.tuple2SpeicalLoopInt         ss  400   132.948 ±  1.291   ms/op
```

primitive type vs boxed primitive

```
[info] Specialization.tuple2LoopDouble          thrpt  400      0.006 ±  0.001  ops/ms
[info] Specialization.tuple2LoopInt             thrpt  400      0.006 ±  0.001  ops/ms
[info] Specialization.tuple2SpeicalLoopDouble   thrpt  400      0.007 ±  0.001  ops/ms
[info] Specialization.tuple2SpeicalLoopInt      thrpt  400      0.008 ±  0.001  ops/ms
[info] Specialization.tuple2LoopDouble           avgt  400    152.308 ±  1.626   ms/op
[info] Specialization.tuple2LoopInt              avgt  400    154.144 ±  1.676   ms/op
[info] Specialization.tuple2SpeicalLoopDouble    avgt  400    155.701 ±  2.723   ms/op
[info] Specialization.tuple2SpeicalLoopInt       avgt  400    133.234 ±  1.106   ms/op
[info] Specialization.tuple2LoopDouble             ss  400    155.689 ±  2.899   ms/op
[info] Specialization.tuple2LoopInt                ss  400    155.466 ±  1.683   ms/op
[info] Specialization.tuple2SpeicalLoopDouble      ss  400    156.365 ±  3.034   ms/op
[info] Specialization.tuple2SpeicalLoopInt         ss  400    132.948 ±  1.291   ms/op
```

- I didn't add annotation for Double, so it still does lots of box and unbox.

primitive type vs boxed primitive

- Primitive type always fast than boxed primitive.
- Multiplying two matrices of 200x100 is 2.5 times faster with specialization.
  - from Specialization in Scala 2.8

String Size

- Object header is around 16 bytes in 64-bit and 12 bytes in 32-bit.
- String has 40 bytes (32-bit) or 56 bytes (64-bit) overhead.
  - Apache spark suggests that uses numeric IDs or enumeration other than String.
- Reduce memory overhead.

Why Scala?

- Why FP?
- Benchmark
  - Speed
  - Memory footprint
  - Line of code
  - Size of Binary and JAR
  - Compilation time
  - Go vs Scala in parallel and concurrent computing
- Subjective opinion

Why FP(Scala)?

- Fewer errors
- Better modularity
- High-level abstractions
- Shorter code
- Increased developer productivity(*)
- Parallel (or Concurrent) computing(*)
- Distributed computing(*)

Why Scala? (Speed)

Benchmark	Time [sec]	Factor
C++ Opt	23	1.0x
C++ Dbg	197	8.6x
Java 64-bit	134	5.8x
Java 32-bit	290	12.6x
Java 32-bit GC*	106	4.6x
Java 32-bit SPEC GC	89	3.7x
Scala	82	3.6x
Scala low-level*	67	2.9x
Scala low-level GC*	58	2.5x
Go 6g	161	7.0x
Go Pro*	126	5.5x

Why Scala? (Speed)

- We find that in regards to performance, C++ wins out by a large margin. However, it also required the most extensive tuning efforts, many of which were done at a level of sophistication that would not be available to the average programmer.
- from Loop Recognition in C++/Java/Go/Scala by Google

Why Scala? (Memory)

Benchmark	Virt	Real	Factor Virt	Factor Real
C++ Opt	184m	163m	1.0	1.0
C++ Dbg	474m	452m	2.6-3.0	2.8
Java	1109m	617m	6.0	3.7
Scala	1111m	293m	6.0	1.8
Go	16.2g	501m	90	3.1

Why Scala? (Lines of code)

Benchmark	wc -l	Factor
C++ Dbg/Opt	850	1.3x
Java	1068	1.6x
Java Pro	1240	1.9x
Scala	658	1.0x
Scala Pro	297	0.5x
Go	902	1.4x
Go Pro	786	1.2x

Why Scala?

- Scala concise notation and powerful language features allowed for the best optimization of code complexity.
- from Loop Recognition in C++/Java/Go/Scala by Google

Why Scala? (Parallel)

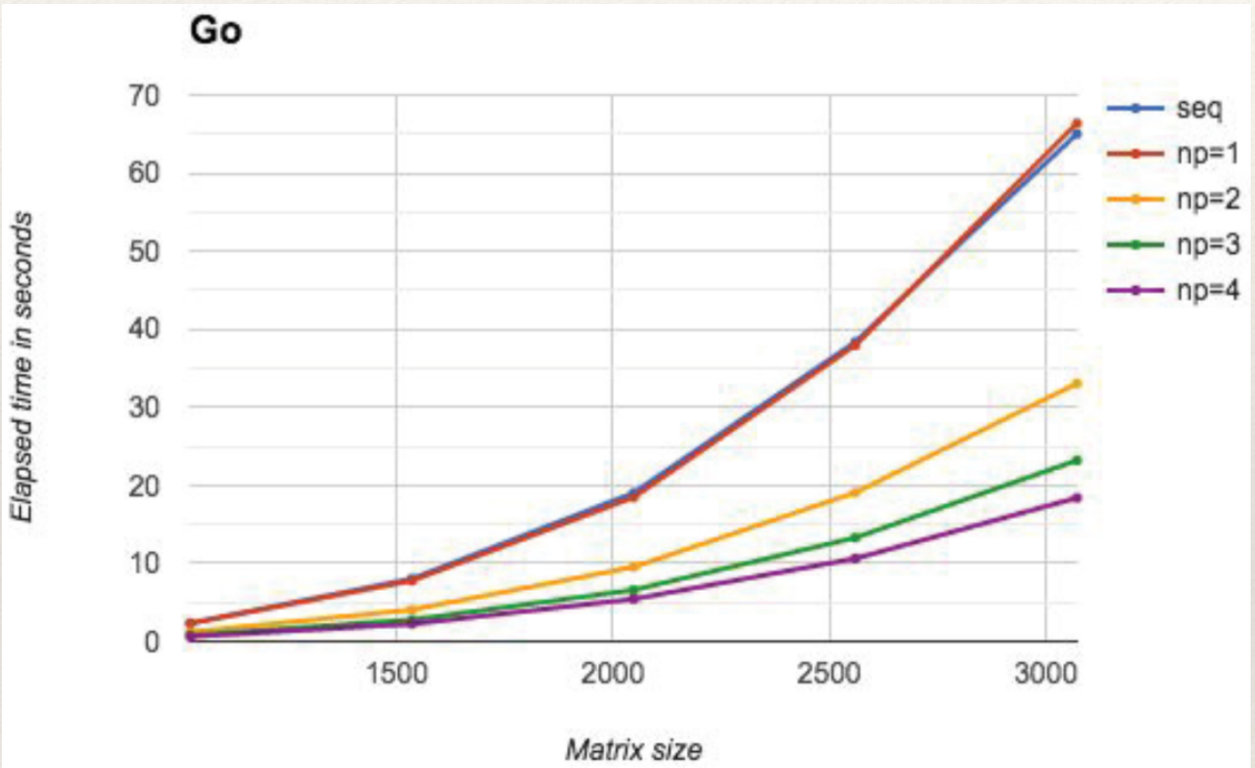


Figure 5-1: Matrix multiplication in Go

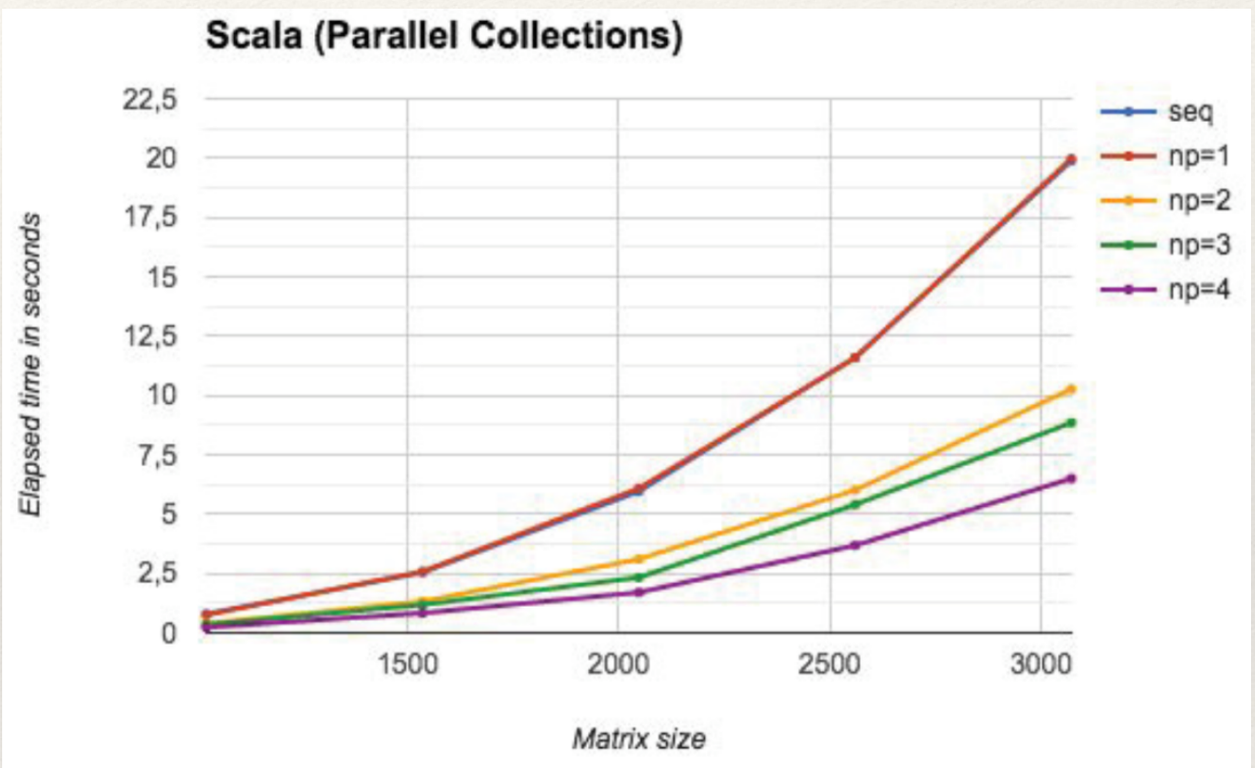


Figure 5-2: Matrix multiplication in Scala with parallel collections

Why Scala? (Concurrent)

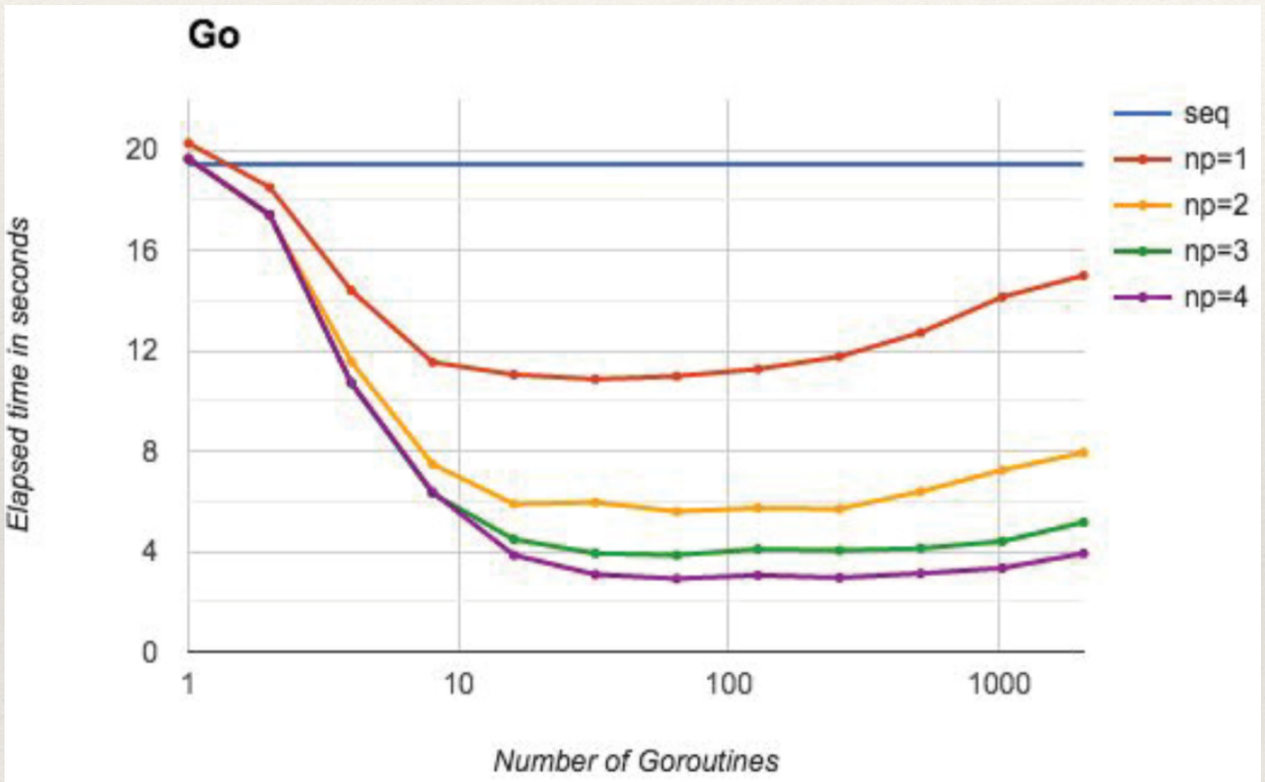


Figure 5-7: Matrix chain multiplication in Go

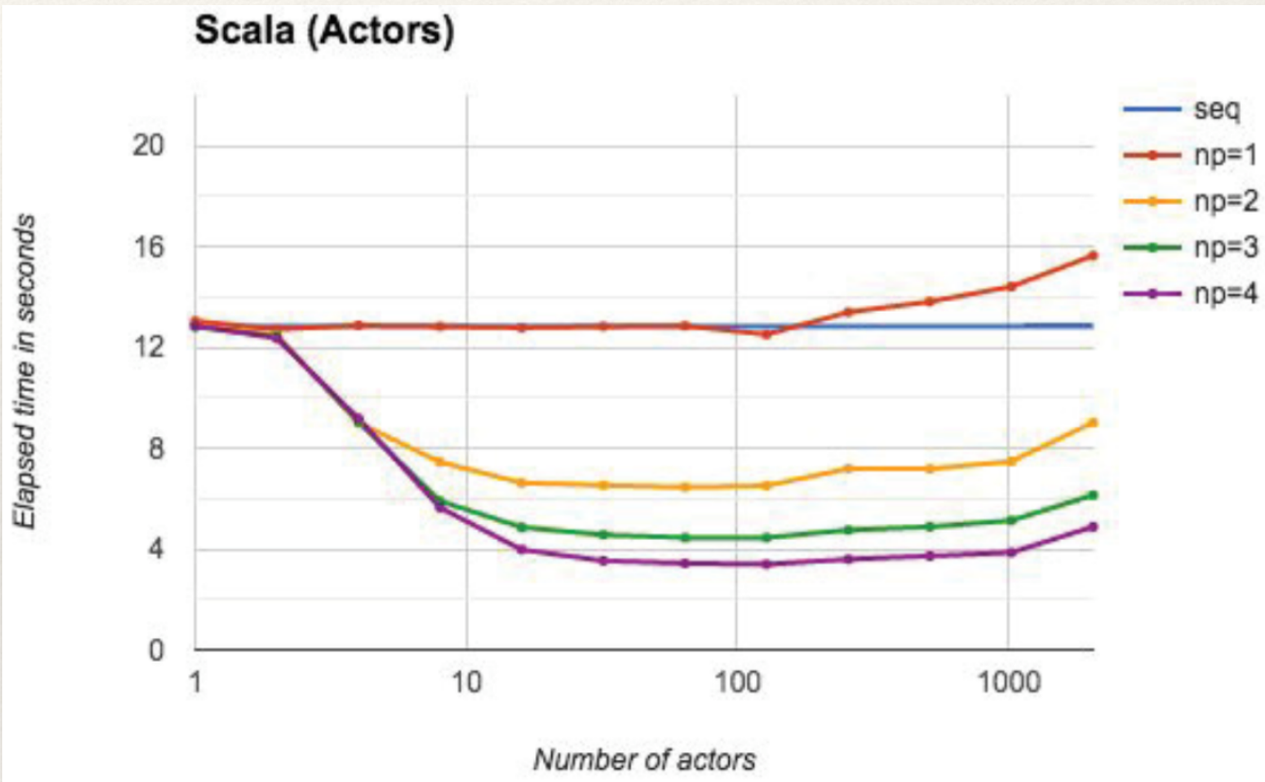


Figure 5-8: Matrix chain multiplication with actors in Scala

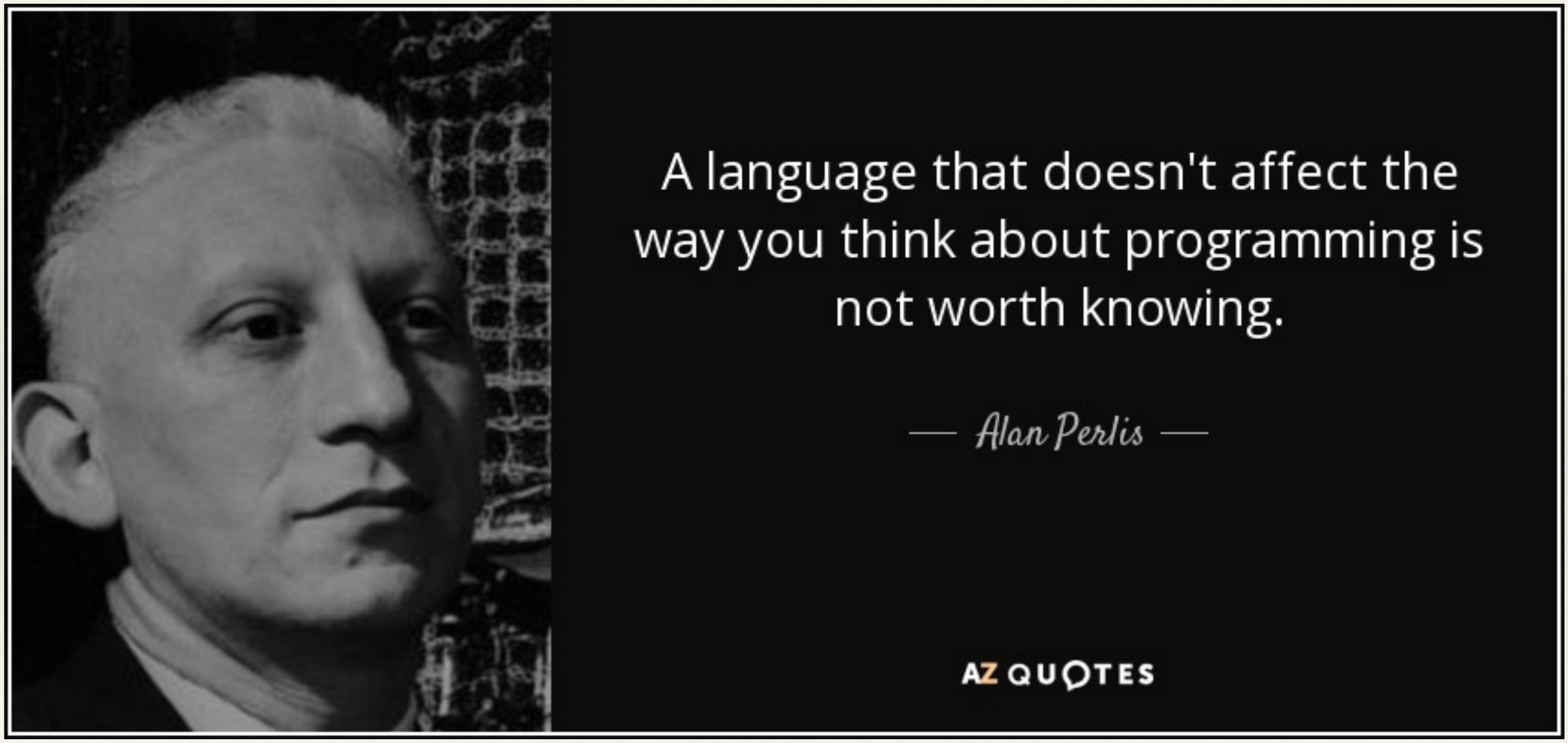
Downside of Scala

Benchmark	Binary or Jar [Byte]	Factor
C++ Dbg	592892	45x
C++ Opt	41507	3.1x
Java	13215	1.0x
Java Pro	21047	1.6x
Scala	48183	3.6x
Scala Pro	36863	2.8x
Go	1249101	94x
Go Pro	1212100	92x

Downside of Scala

Benchmark	Compile Time	Factor
C++ Dbg	3.9	6.5x
C++ Opt	3.0	5.0x
Java	3.1	5.2x
Java Pro	3.0	5.0x
Scala scalac	13.9	23.1x
Scala fsc	3.8	6.3x
Scala Pro scalac	11.3	18.8x
Scala Pro fsc	3.5	5.8x
Go	1.2	2.0x
Go Pro	0.6	1.0x

Why Scala? (Subjective)



Downside of Scala (Subjective)

- I enjoy writing Scala code but I hate reading other people's Scala code. from Reddit
  - Code review could solve this issue.
- OOP or FP?
  - double-edged sword

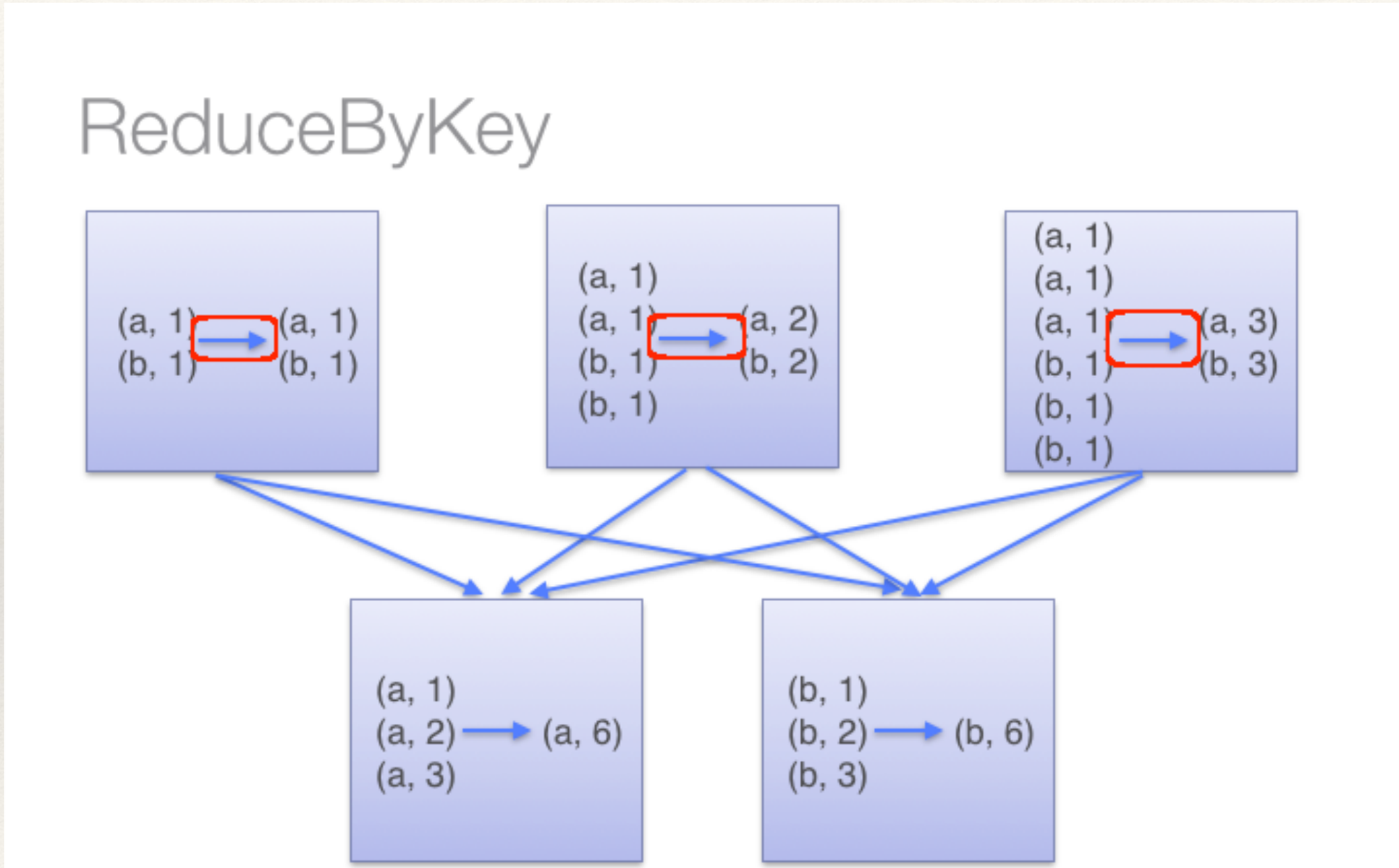
Why Scala? (Summary)

- Scala isn't the best at any of these things though. It's great because it's an all-rounder language.
- Scala doesn't - quite - have the **safety** and **conciseness** of *Haskell*. But it's close.
- Scala doesn't - quite - have the **enterprise support** and **tooling infrastructure** (monitoring/instrumentation, profiling, debugging, IDEs, library ecosystem) that *Java* does. But it's close.
- Scala doesn't - quite - have the **clarity** of *Python*. But it's close.
- Scala doesn't - quite - have the **performance** of *C++*. But it's close.
- From https://news.ycombinator.com/item?id=9398911 by Michael Donaghy(m50d)

Spark Performance

- 1. Programming API
  - reduceByKey
  - aggregateByKey
  - coalesce
- 2. Spark configuration and JVM tuning.
  - Spark: Java/Kyro, Memory Management, Persist
  - VM Options: JIT compiler, Memory, GC, JMX, Misc

reduceByKey



aggregateByKey

- Spark Source Code.
- Serialise the zero value to a byte array so that we can get a new clone of it on each key.
- It's a shallow copy so it's fast than new object each time.

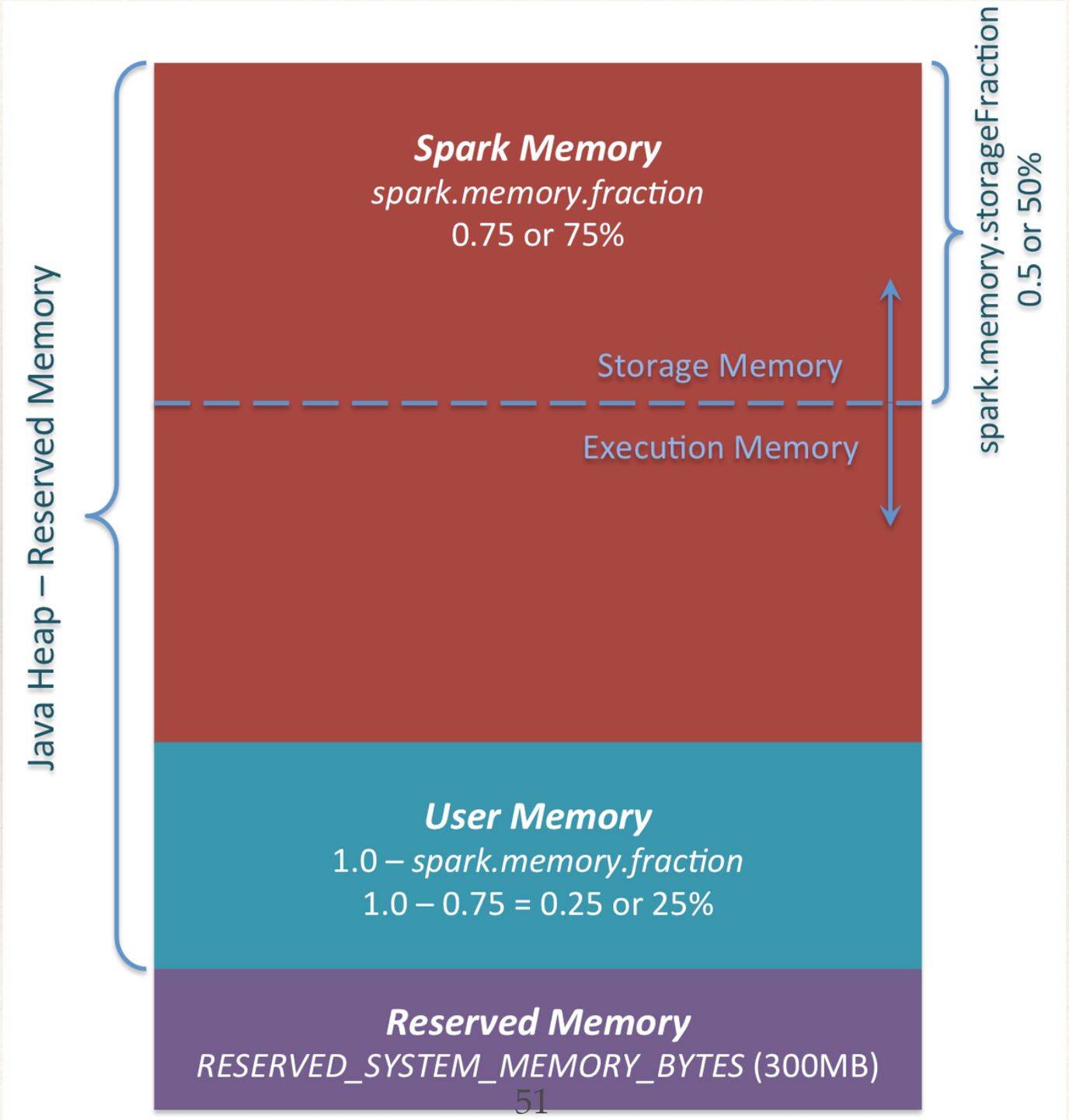
coalesce

- coalesce() that allows avoiding data movement, but only if you are decreasing the number of RDD partitions.
  from Learning Spark - Lightning-Fast Big Data Analysis
- coalesce with shuffle?
  - If you have a few partitions being abnormally large.

Spark Serialization

- Java serialization
- Kyro serialization
  - Pro: Faster than Java serialization .
  - Con: Not support all Serializable type.

Spark Memory Management



Spark Persist

- MEMORY_ONLY = Cache
- MEMORY_SER: Cache with serialization.
- MEMORY_AND_DISK: Cache, but it will flush to disk when meet the limit.
- MEMORY_AND_DISK_SER
- DISK_ONLY
- OFF_HEAP: Work with Alluxio(Tachyon)
- *_2: With replication

Spark Persist

- MEMORY_ONLY: If you have sufficient memory.
- MEMORY_ONLY_SER: It can help to reduce memory usage. But it also consumes cpu. (Non-serialization will 2~3 time bigger than serialization)
- *_DISK: Don't use it unless your computing are expensive. (Official). I think It will depend on which type of disk you use?
- Remember unpersist cache when you don't need.
  - blocking or not? Blocking for preventing memory pressure.
  - non-blocking only for you have sufficient memory

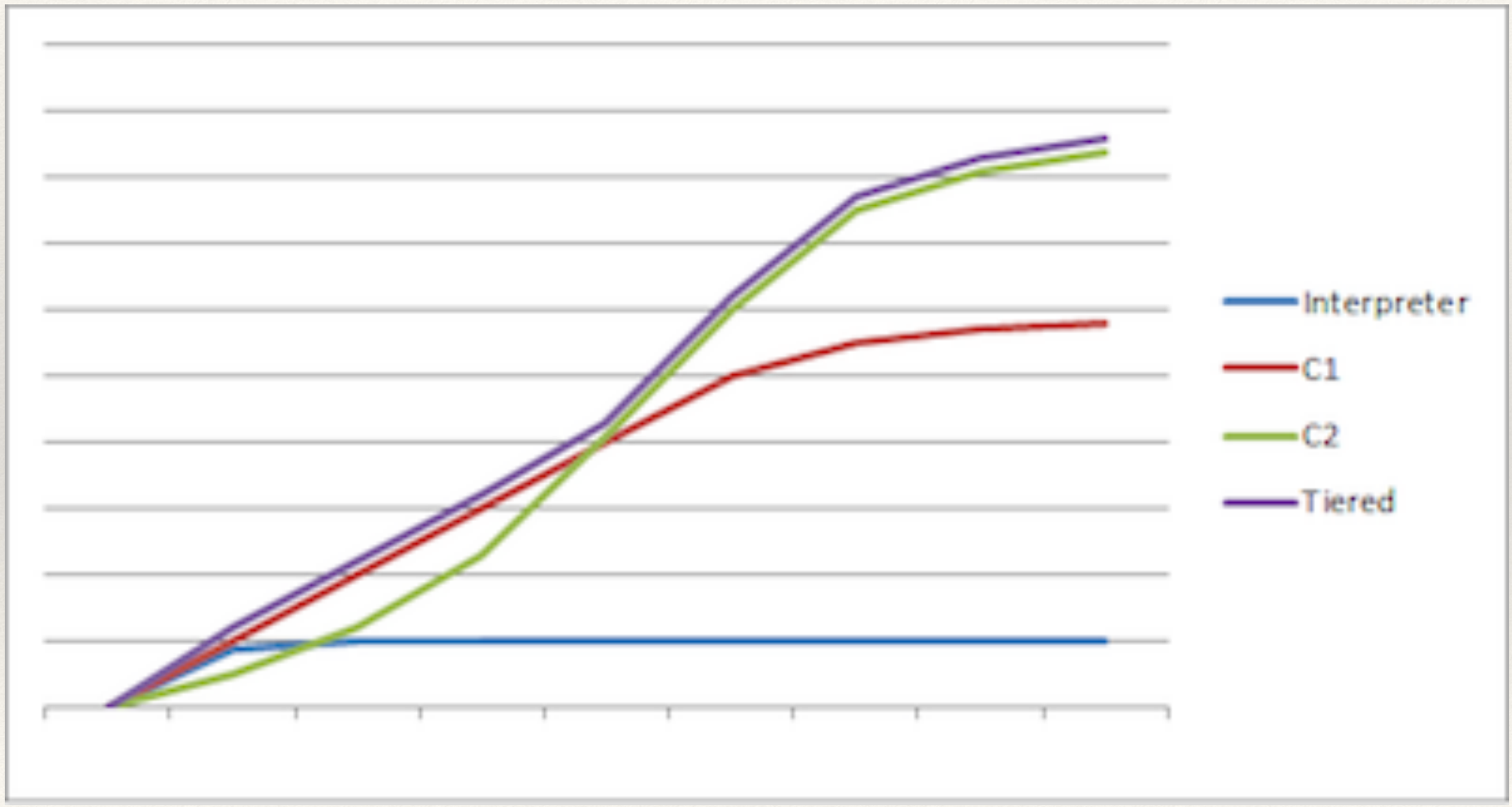
Spark Memory Management

- How much memory you should allocate?
- When JVM uses more than 200G, you should consider to use 2 workers per node then each work has 100G.
- Total Memory * 3/4 for Spark. The rest is used for OS and buffer cache.

VM Options - JIT

- JIT Compiler
  - -client (1500)
  - -server (10,000)
  - -XX:+TieredCompilation
    - -client (2000) + -server (15,000) from Azul Systems
    - default setup in JAVA 8.

VM Options - JIT



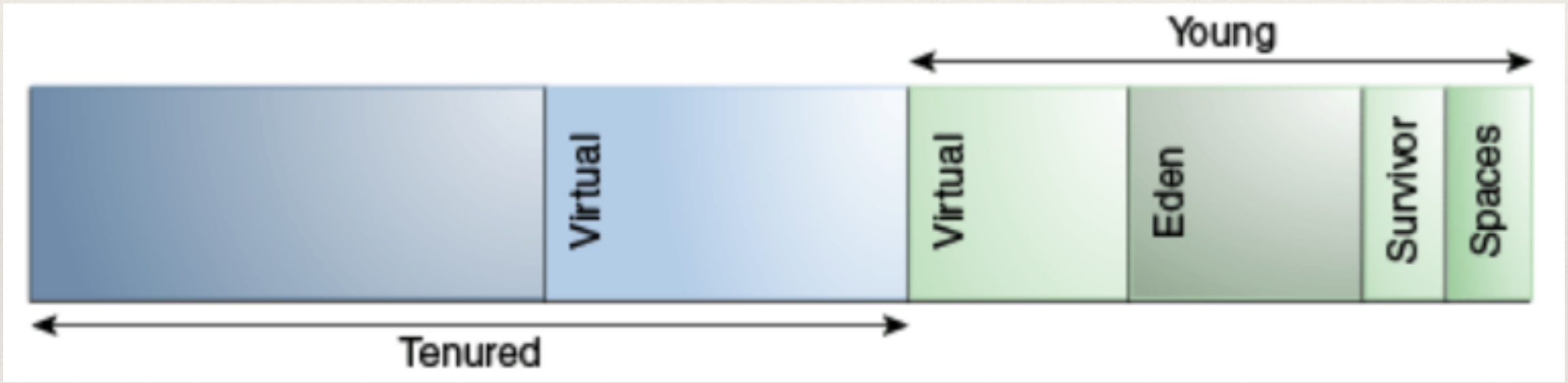
VM Options - Memory

- -XX:+UseCompressedOops
  - oop = ordinary object pointer
  - Emulates 35-bit pointers for object references.
  - When you use memory lower than 32 G.
  - Reduce memory footprint slightly.

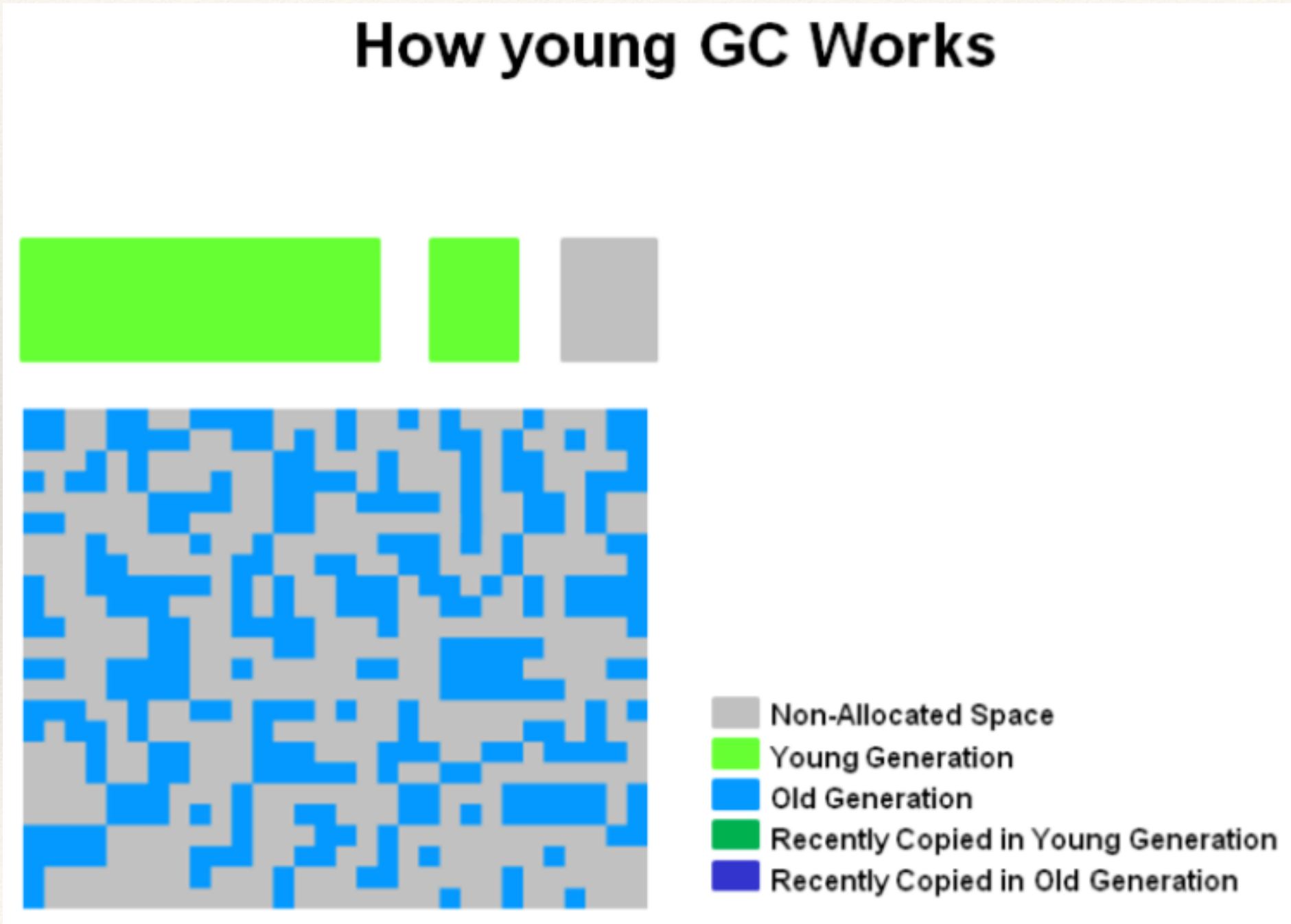
VM Options - GC

- Parallel GC
  - Throughput
  - Batch, Scientific computing
- CMS GC
  - Latency
  - Streaming, Web Service
- G1 GC
  - Throughput & Latency
  - Batch & Streaming
  - The long term replacement for CMS.

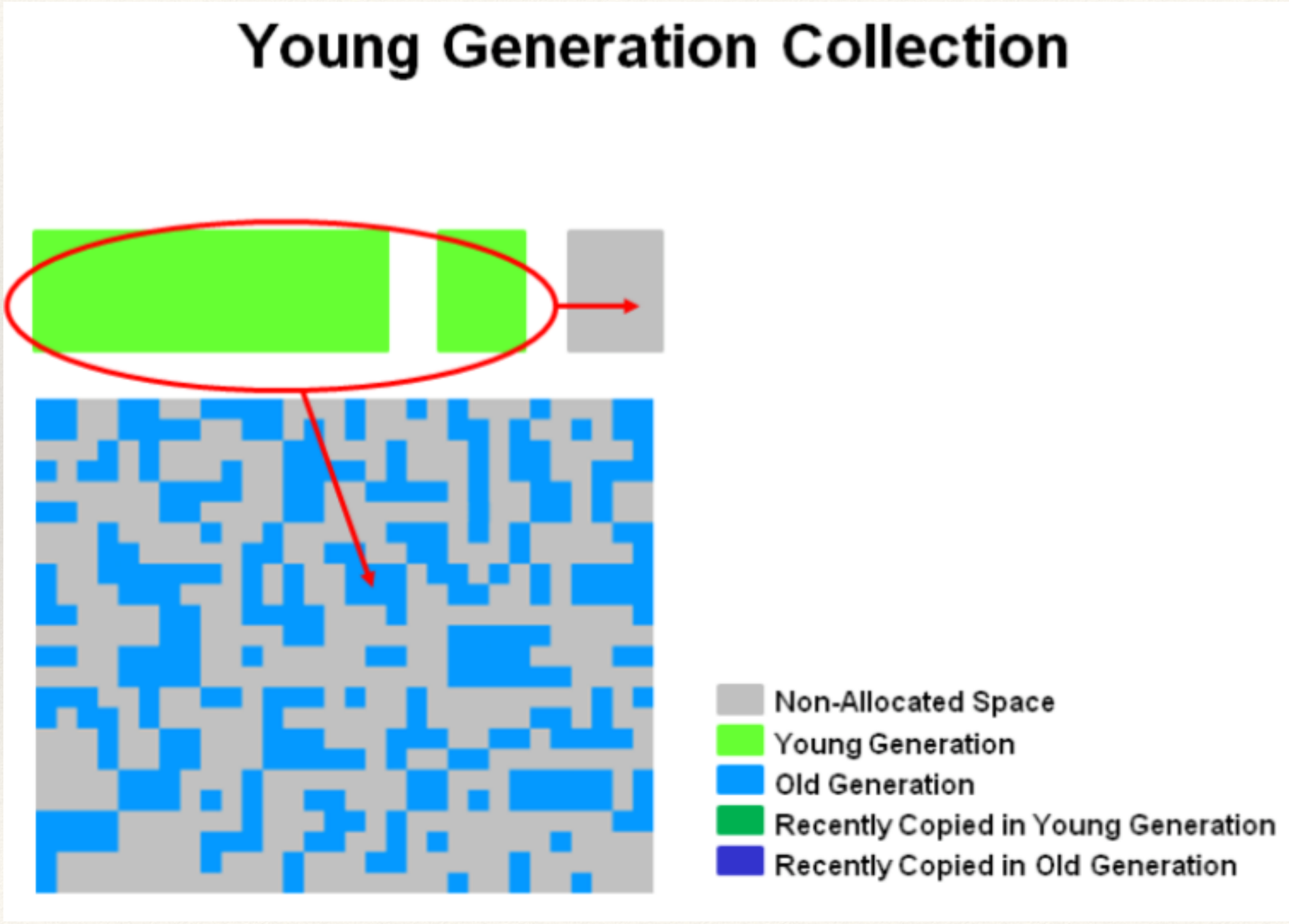
VM Options - Parallel Collector



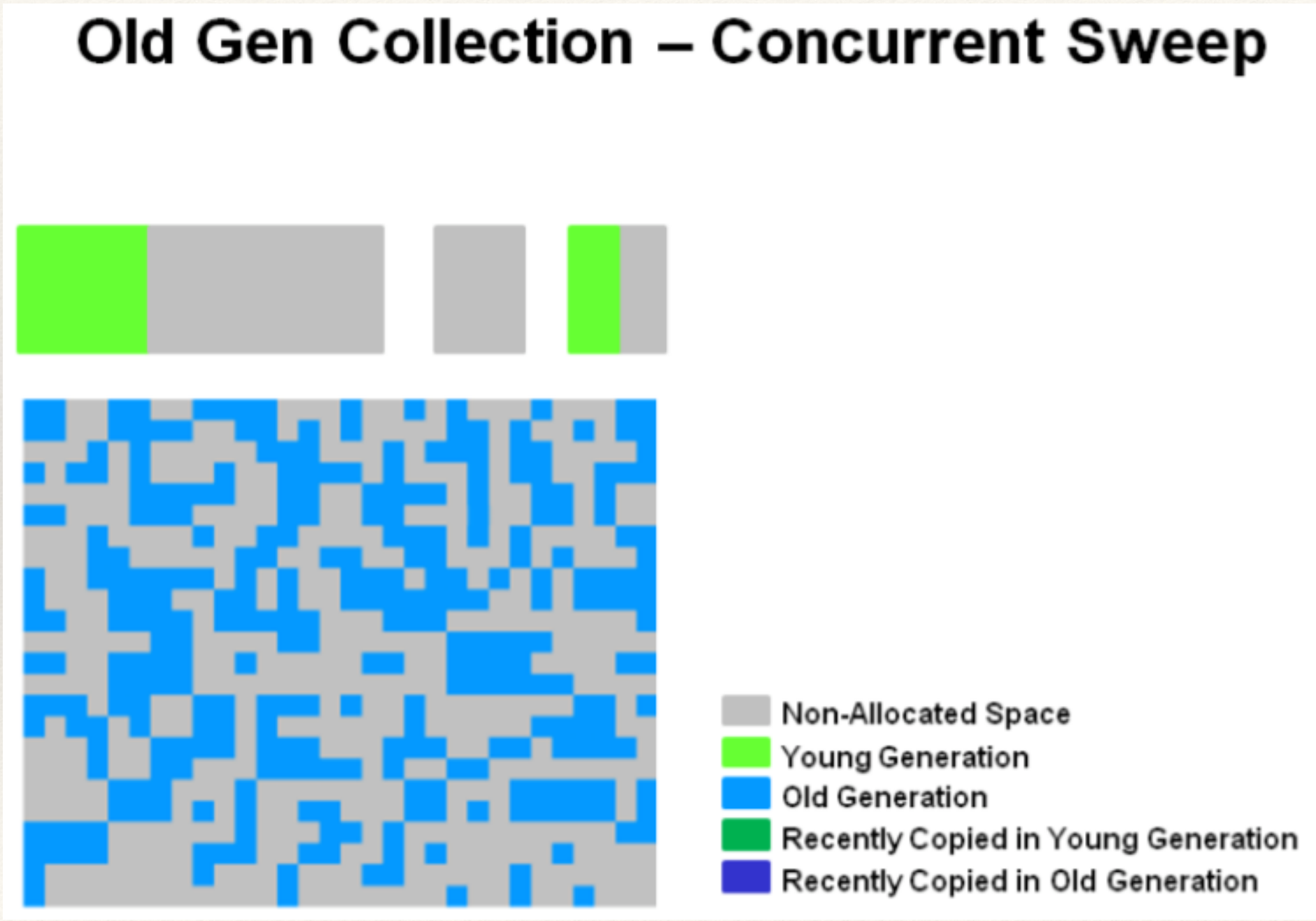
VM Options - CMS



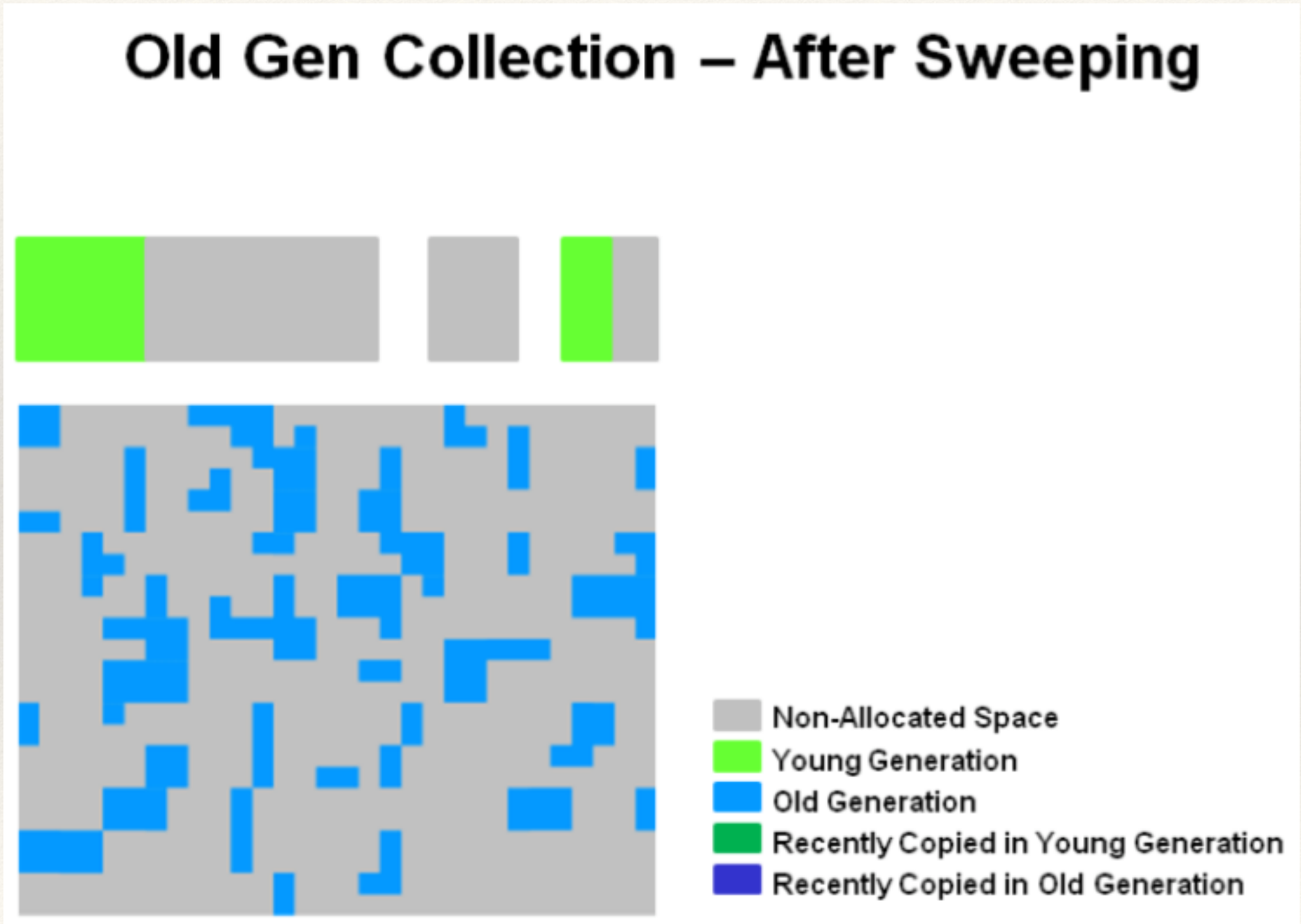
VM Options - CMS



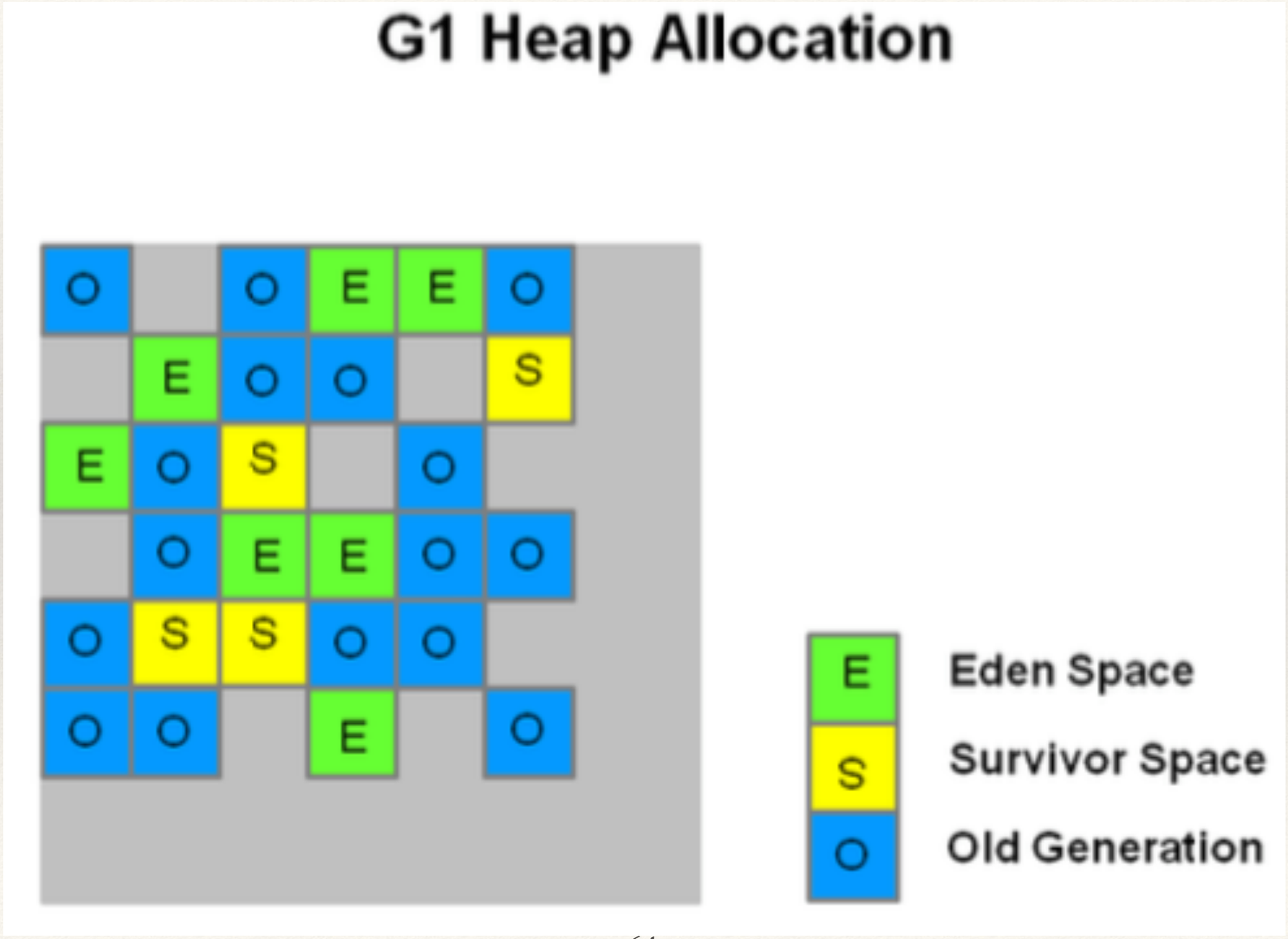
VM Options - CMS



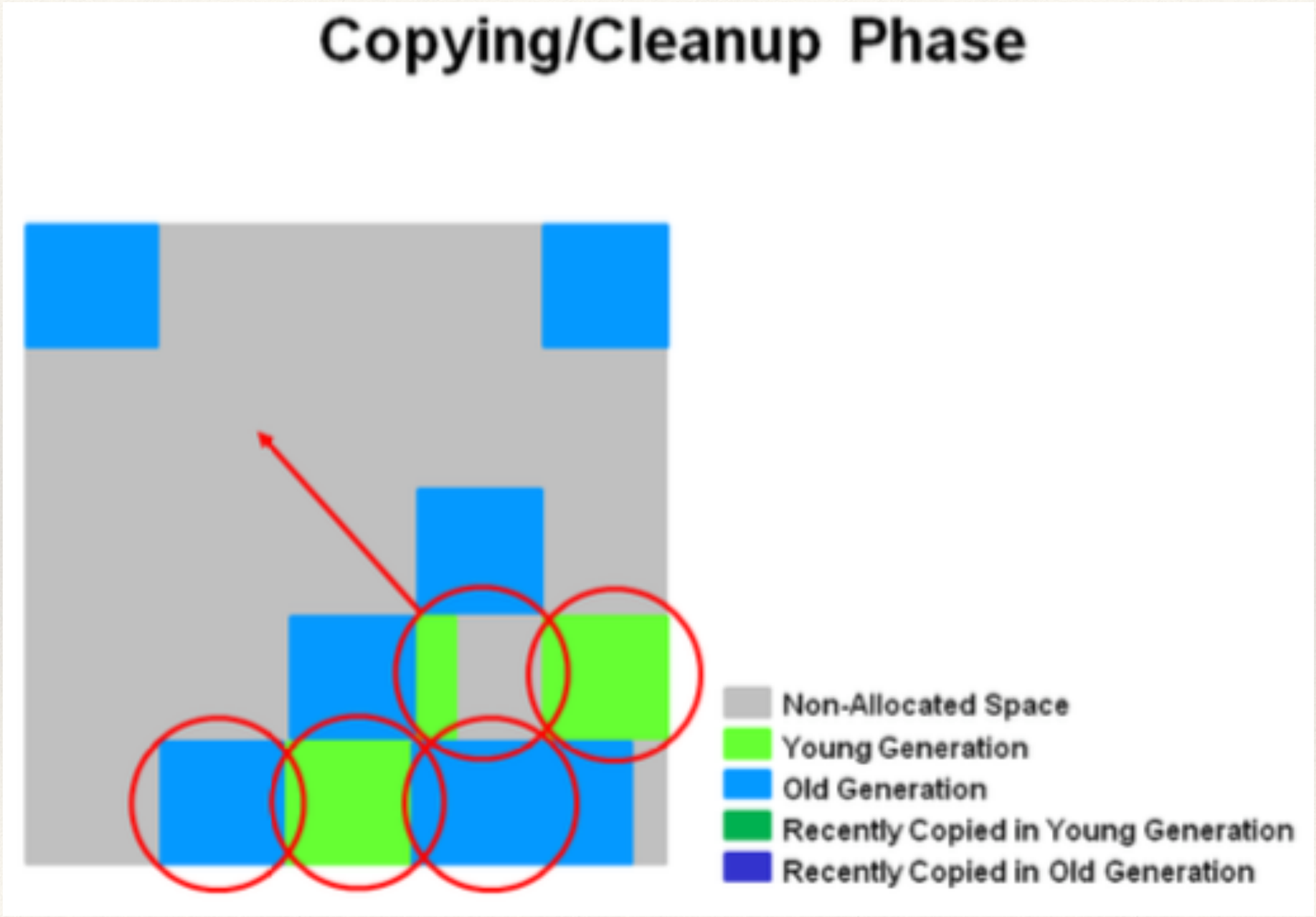
VM Options - CMS

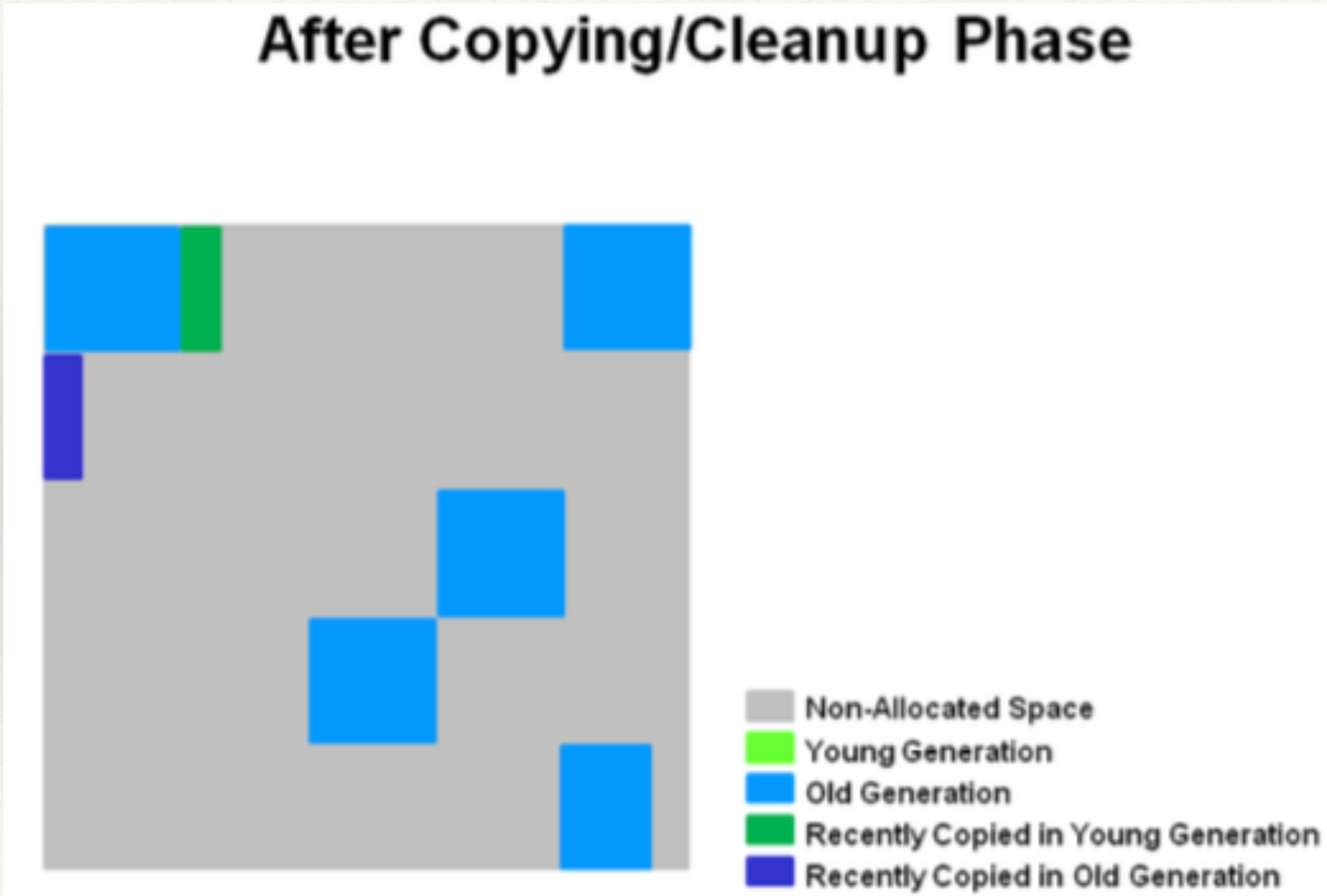


VM Options - G1 GC



VM Options - G1 GC




After Copying/Cleanup Phase
Non-Allocated Space
Young Generation
Old Generation
Recently Copied in Young Generation
Recently Copied in Old Generation

VM Options - G1 GC

- Based on the usage of the entire heap.
- -XX:InitiatingHeapOccupancyPercent=n
  - n = 35 # original value = 45
- -XX:ParallelGCThreads=n
  - if (cores > 8) cores * 5/8 else cores # From Oracle
- -XX:ConcGCThreads=n,
  - n = ParallelGCThreads * 1/4 # From Oracle

G1 vs CMS

	G1	CMS
Heap compaction	Minor, Major, Full GC	Full GC
Heap Layout	Partition to regions	Young(eden, S0, S1) and Old
Heap Requirement	Normal Memory	Higher Memory

VM Options - JMX

- JMX = Java Management Extensions
- -Dcom.sun.management.jmxremote -
Dcom.sun.management.jmxremote.port={port} -
Dcom.sun.management.jmxremote.rmi.port={port} -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl=false -
Djava.rmi.server.hostname={ip or hostname}

VM Options - JMX

- VisualVM(jvisualvm)
- jmc

VM Options - Misc

- (A) -XX:+UseCompressedStrings(~~JDK 7~~, ~~JDK 8~~)
- (B) -XX:+AlwaysPreTouch
- (C) -XX:+UseStringCache(~~JDK 8~~)
- (D) -XX:+OptimizeStringConcat
- Add B, C, and D will reduce around 10s from 6m 48s to 6m 37s in 48 cores and 256 GB RAM.

Summery

- The Java version was probably the simplest to implement, but the hardest to analyze for performance. Specifically the effects around garbage collection were complicated and very hard to tune. Since Scala runs on the JVM, it has the same issues. from Loop Recognition in C++/Java/Go/Scala by Google.
- Knowledge of JVM(Java) still works for Scala.

Performance in production

- Performance in production = System Performance
  - System performance is the study of the entire system, including all physical components and the full software stack.
  - Performance engineering should ideally begin before hardware is chosen or software is written.
  - Performance is often subjective.
  - Performance can be a challenging discipline due to the complexity of systems and the lack of a clear staring point for analysis.
    - From Systems Performance: Enterprise and the Cloud by Brendan Gregg (Senior performance architect @ Netflix)

References

- http://izquotes.com/
- http://www.azquotes.com/
- https://blog.codinghorror.com/how-to-become-a-better-programmer-by-not-programming/
- docs.scala-lang.org/overviews/collections/performance-characteristics.html
- https://www.gitbook.com/book/databricks/databricks-spark-knowledge-base/details
- https://0x0fff.com/spark-memory-management/
- www.javaworld.com/article/2078635/enterprise-middleware/jvm-performance-optimization-part-2-compilers.html
- https://github.com/dougqh/jvm-mechanics/blob/master/JVM%20Mechanics.pdf
- Java Performance by Charile Hunt and Binu John
- Loop Recognition in C++/Java/Go/Scala by Google
- Systems Performance: Enterprise and the Cloud by Brendan Gregg, Senior Performance Architect of Netflix
- https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html
- Understanding Java Garbage Collection and what you can do about it by Gil Tene, CTO of Azul Systems
- Java Performance: The Definitive Guide by Scott Oaks, Architect of Oracle
- Martin Odersky: Scala with Style
- Scala Performance Considerations by Nermin Serifovic
- Scala for the Impatient by Cay Horstmann
- Parallel programming in Go and Scala A performance comparison by Carl Johnell