SPORES: DISTRIBUTABLE FUNCTIONS in Scala

Improving Support for *Distributed Programming* in Scala

strangeloop
• sept 18-20 2013 •

Heather Miller
@heathercmiller
heather.miller@epfl.ch

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

PICKLES & SPORES

Improving Support for *Distributed Programming* in Scala

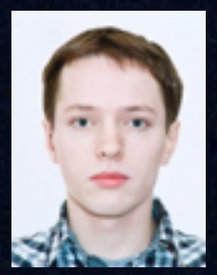
strangeloop
• sept 18-20 2013 •

Heather Miller
@heathercmiller
heather.miller@epfl.ch

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

WITH:

Philipp Haller
Typesafe



Eugene Burmako
EPFL

Martin Odersky
EPFL/Typesafe

WHAT IS THIS TALK ABOUT?

WHAT IS THIS TALK ABOUT?

MAKING DISTRIBUTED PROGRAMMING EASIER IN SCALA

THIS KIND OF DISTRIBUTED SYSTEM

Also!

THIS KIND OF DISTRIBUTED SYSTEM

BOTTOMLINE: *Machines Communicating*

BOTTOMLINE:

Machines Communicating

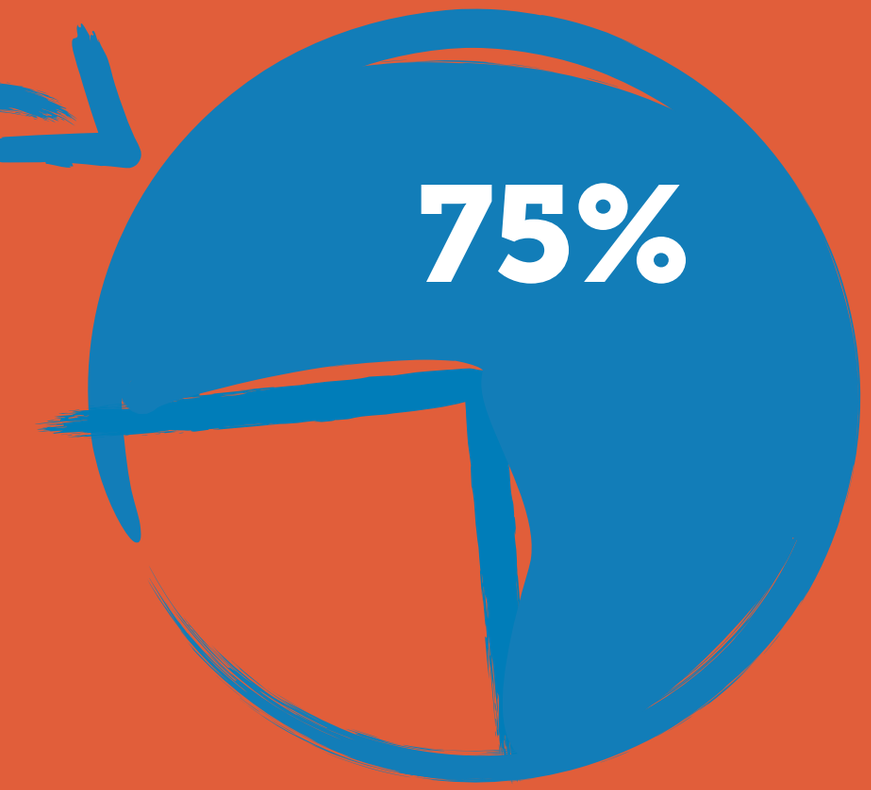
HOW CAN WE SIMPLIFY DISTRIBUTION AT THE LANGUAGE-LEVEL?

Agenda

SPORES
PICKLING

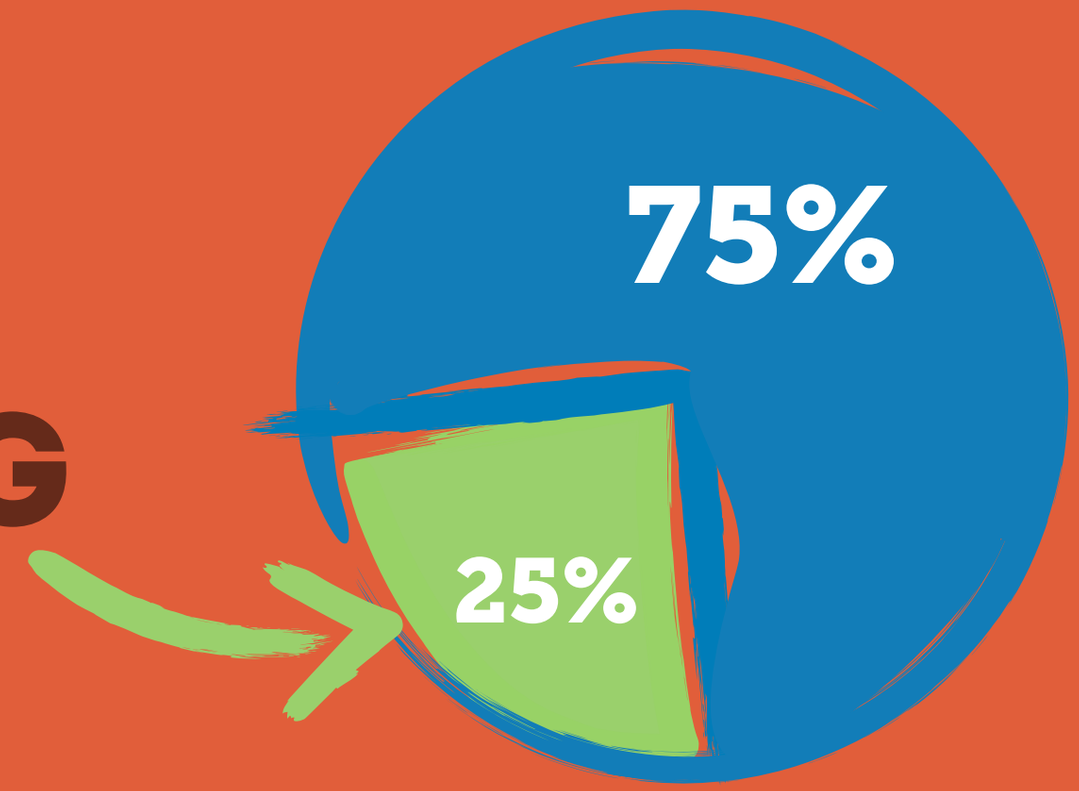
Agenda

SPORES
PICKLING



Agenda

SPORES
PICKLING



Agenda

SPORES
PICKLING

THIS STUFF IS BOTH

RESEARCH & INTENDED FOR PRODUCTION

Agenda

SPORES PICKLING

RESEARCH

Accepted for publication at OOPSLA'13

Instant Pickles: Generating Object-Oriented Pickler Combinators for Fast and Extensible Serialization,
Heather Miller, Philipp Haller, Eugene Burmako, Martin Odersky.
@OOPSLA'13, Indianapolis, IN, October 26-31, 2013.

SPLASH INDIANAPOLIS 2013 OCTOBER 26-31

PRACTICE

Used by a handful of companies, Scala Language Proposal in the works

Agenda

SPORES
PICKLING

Agenda

SPORES

PICKLING

RESEARCH

Academic paper on the foundations and practical benefits of Scala's spores in the works. Draft will be available in the coming months.

PRACTICE

Scala Improvement Proposal posted, lots of user feedback, helped reformulate the design.
Release soon upcoming of what will come in the Scala 2.11 distribution.

scala.spores

SPORES!

first, imagine the...

SINGLE MACHINE SCENARIO

CLOSURES ARE WONDERFUL

Closures ARE WONDERFUL

HOW DO YOU DO FP WITHOUT THEM?

Closures ARE WONDERFUL

HOW DO YOU DO FP WITHOUT THEM?

Ok, ok, anonymous inner classes. But still.

Closures ARE WONDERFUL

HOW DO YOU DO FP WITHOUT THEM?

Ok, ok, anonymous inner classes. But still.

FP IS ALL ABOUT TRANSFORMATIONS ON IMMUTABLE DATA.

These transformations are just closures.

Closures ARE WONDERFUL

HOW DO YOU DO FP WITHOUT THEM?

Ok, ok, anonymous inner classes. But still.

FP IS ALL ABOUT TRANSFORMATIONS ON IMMUTABLE DATA.

These transformations are just closures.

MONADS BACKED BY DATA, LIKE LISTS, OPTIONS OR FUTURES.

Typically, you pass closures to higher-order functions.

Closures ARE WONDERFUL

HOW DO YOU DO FP WITHOUT THEM?

Ok, ok, anonymous inner classes. But still.

ESSENTIALLY, YOU'RE SENDING THE CLOSURE TO THE DATA.

MONADS BACKED BY DATA, LIKE LISTS, OPTIONS OR FUTURES.

Typically, you pass closures to higher-order functions.

Closures ARE WONDERFUL

HOW DO YOU DO FP WITHOUT THEM?

Ok, ok, anonymous inner classes. But still.

ESSENTIALLY, YOU'RE SENDING THE CLOSURE TO THE DATA.

Oh yeah,

EVEN JAVA'S GOING TO GET CLOSURES

Ok, back to thinking

DISTRIBUTED

Closures are awesome.

Closures are awesome.

BUT WE CAN'T REALLY DISTRIBUTE THEM.

But, WE CAN'T REALLY DISTRIBUTE THEM

WHY?

- **OFTEN NOT SERIALIZABLE** because they capture stuff that's not serializable.

  ...enclosing `this`, anyone?

- **ACCIDENTAL CAPTURE.** Easy to reference something and unknowingly capture it.

But, WE CAN'T REALLY DISTRIBUTE THEM

WHY?

- **OFTEN NOT SERIALIZABLE** because they capture stuff that's not serializable.

  ...enclosing `this`, anyone?

- **ACCIDENTAL CAPTURE.** Easy to reference something and unknowingly capture it.

- **RUNTIME ERRORS** instead of compile-time checks.

But, WE CAN'T REALLY DISTRIBUTE THEM

WHY?

- **OFTEN NOT SERIALIZABLE** because they capture stuff that's not serializable.

  ...enclosing `this`, anyone?

- **ACCIDENTAL CAPTURE.** Easy to reference something and unknowingly capture it.

- **RUNTIME ERRORS** instead of compile-time checks.

- **WHO'S FAULT IS IT?** for a user, often unclear whether it's a user-error or the framework

But,

WE CAN'T REALLY DISTRIBUTE THEM

CONSEQUENCES THAT FOLLOW FROM THESE PROBLEMS...

But,

WE CAN'T REALLY DISTRIBUTE THEM

CONSEQUENCES THAT FOLLOW FROM THESE PROBLEMS...

FRAMEWORK BUILDERS AVOID THEM

...not just in their public APIs, but private ones too.

Users shoot themselves in the foot and blame framework.

But,

WE CAN'T REALLY DISTRIBUTE THEM

CONSEQUENCES THAT FOLLOW FROM THESE PROBLEMS...

FRAMEWORK BUILDERS AVOID THEM

...not just in their public APIs, but private ones too.

Users shoot themselves in the foot and blame framework.

When picking battles, framework designers tend to avoid issues with closures.

But,

WE CAN'T REALLY DISTRIBUTE THEM

CONSEQUENCES THAT FOLLOW FROM THESE PROBLEMS...

FRAMEWORK BUILDERS AVOID THEM

...not just in their public APIs, but private ones too.

Users shoot themselves in the foot and blame framework.

When picking battles, framework designers tend to avoid issues with closures.

RIGHTFULLY SO.

Have you heard this before?:

Have you heard this before?:

FUNCTIONAL PROGRAMMING
The Way To Go™

...for parallelism/concurrency/distribution

Have you heard this before?:

FUNCTIONAL PROGRAMMING
The Way To Go™

...for parallelism/concurrency/distribution

...And composing and passing functions around is the way to do FP.

However, managing closures in a concurrent or distributed environment, or writing APIs to be used by clients in such an environment, remains considerably precarious.

BUT WOULDN'T IT BE NICE IF WE COULD...

More reliably
develop new types of frameworks
for distributed programming based
on **passing functions**?

...foolproof dist collections, dist streams, ...

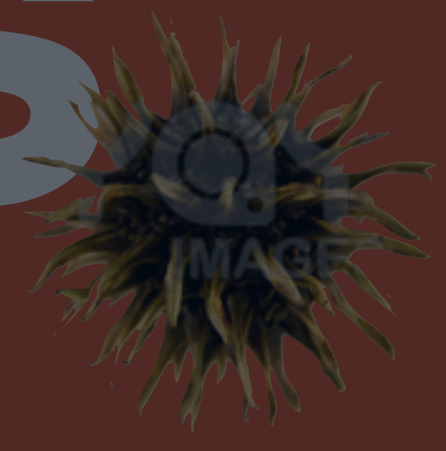
ENTER:
SPORES

ENTER:
SPORES

two types:

1. **MAINLINE SPORES**
   proposed for inclusion in Scala 2.11
2. **SPORES WITH TYPE CONSTRAINTS**
   research project @EPFL,
   research paper in the works

SPORES



two types:

1. **MAINLINE SPORES**
   proposed for inclusion in Scala 2.11

2. **SPORES WITH TYPE CONSTRAINTS**
   research project @EPFL,
   research paper in the works

mainline SPORES

WHAT ARE THEY?

SMALL UNITS OF POSSIBLY MOBILE FUNCTIONAL BEHAVIOR

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

mainline SPORES

WHAT ARE THEY?

A closure-like abstraction for use in distributed or concurrent environments.

GOAL:

Well-behaved closures with controlled environments that can avoid various hazards.

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

mainline SPORES

POTENTIAL HAZARDS WHEN USING CLOSURES INCORRECTLY:

- memory leaks
- race conditions due to capturing mutable references
- runtime serialization errors due to unintended capture of references

GOAL:

Well-behaved closures with controlled environments that can avoid various hazards.

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

motivating example: SPARK

```
class MyCoolRddApp {
 val param = 3.14
 val log = new Log(...)
 ...
 def work(rdd: RDD[Int]) {
   rdd.map(x => x + param)
      .reduce(...)
 }
}
```

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

motivating example: SPARK

```
class MyCoolRddApp {
 val param = 3.14
 val log = new Log(...)
 ...
 def work(rdd: RDD[Int]) {
   rdd.map(x => x + param)
      .reduce(...)
 }
}
```

PROBLEM:

`(x => x + param)`
not serializable
because it captures
`this` of type
`MyCoolRddApp`
which is itself not
serializable

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

motivating example: AKKA/FUTURES

```
def receive = {
 case Request(data) =>
   future {
     val result = transform(data)
     sender ! Response(result)
   }
}
```

PROBLEM: Akka actor spawns future to concurrently process incoming results

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

motivating example:
AKKA/FUTURES

AKKA ACTOR SPAWNS A FUTURE TO CONCURRENTLY PROCESS INCOMING REQS

```
def receive = {
  case Request(data) =>
    future {
      val result = transform(data)
      sender ! Response(result)
    }
}
```

PROBLEM: Akka actor spawns future to concurrently process incoming results

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

motivating example: AKKA/FUTURES

AKKA ACTOR SPAWNS A FUTURE TO CONCURRENTLY PROCESS INCOMING REQS

```
def receive = {
  case Request(data) =>
    future {
      val result = transform(data)
      sender ! Response(result)
    }
}
```

NOT A STABLE VALUE! IT'S A METHOD CALL!

PROBLEM: Akka actor spawns future to concurrently process incoming results

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

motivating example: SERIALIZATION

```
case class Helper(name: String)

class Main {
  val helper = Helper("the helper")

  val fun: Int => Unit = (x: Int) => {
    val result = x + " " + helper.toString
    println("The result is: " + result)
  }
}
```

PROBLEM:

`fun` not serializable.
Accidentally captures `this` since `helper.toString` is really `this.helper.toString`, and `Main` (the type of `this`) is not serializable.

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

motivating example: SERIALIZATION

```
case class Helper(name: String)

class Main {
  val helper = Helper("the helper")

  val fun: Int => Unit = (x: Int) => {
    val result = x + " " + helper.toString
    println("The result is: " + result)
  }
}
```

PROBLEM:

`fun` not serializable.
Accidentally captures `this` since `helper.toString` is really `this.helper.toString`, and `Main` (the type of `this`) is not serializable.

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

Ok. Got it.

WE NEED SAFER CLOSURES FOR CONCURRENT & DISTRIBUTED SCENARIOS.

SURE.

Ok. Got it.

WE NEED SAFER CLOSURES FOR CONCURRENT & DISTRIBUTED SCENARIOS. SURE.

WHAT DO THESE THINGS LOOK LIKE?

WHAT DO SPORES LOOK LIKE?

Basic usage:

```
val s = spore {
  val h = helper
  (x: Int) => {
    val result = x + " " + h.toString
    println("The result is: " + result)
  }
}
```

THE BODY OF A SPORE CONSISTS OF 2 PARTS

1. a sequence of local value (`val`) declarations only (the "*spore header*"), and
2. a closure

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

A SPORE Guarantees... (vs CLOSURES)

1. All captured variables are declared in the spore header, or using `capture`
2. The initializers of captured variables are executed once, upon creation of the spore
3. References to captured variables do not change during the spore's execution

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

SPORES & CLOSURES

EVALUATION SEMANTICS:

Remove the `spore` marker, and the code behaves as before

SPORES & CLOSURES ARE RELATED:

You can write a full function literal and pass it to something that expects a spore.
(Of course, only if the function literal satisfies the spore rules.)

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

Ok. So. HOW CAN YOU USE A SPORE?

IN APIS

If you want parameters to be spores,
then you can write it this way

```
def sendOverWire(s: Spore[Int, Int]): Unit = ...
// ...
sendOverWire((x: Int) => x * x - 2)
```

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

Ok. So. HOW CAN YOU USE A SPORE?

FOR-COMPREHENSIONS

```
def lookup(i: Int): DCollection[Int] = ...
val indices: DCollection[Int] = ...

for { i <- indices
      j <- lookup(i)
} yield j + capture(i)

trait DCollection[A] {
 def map[B](sp: Spore[A, B]): DCollection[B]
 def flatMap[B](sp: Spore[A, DCollection[B]]): DCollection[B]
}
```

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

Right,

WHAT DOES ALL OF THAT GET YOU?

WHAT DOES ALL OF THAT GET YOU?

SINCE...

→ Captured expressions are evaluated upon spore creation.

THAT MEANS...

→ Spores are like function values with an immutable environment.

→ Plus, environment is specified and checked, no accidental capturing.

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

WHAT DOES ALL OF THAT GET YOU?

OR, GRAPHICALLY...

1
Right after creation

2
During execution

SPORES

CLOSURES

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

WHAT DOES ALL OF THAT GET YOU?

OR, GRAPHICALLY...

1
Right after creation

2
During execution

SPORES

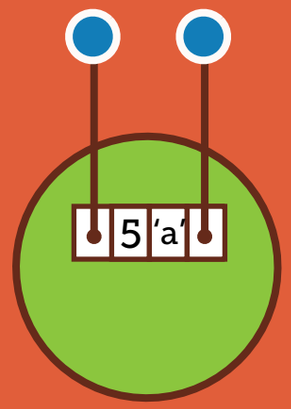
CLOSURES

Proposed for inclusion in Scala 2.11

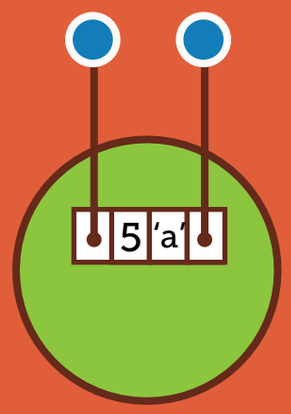
http://docs.scala-lang.org/sips/pending/spores.html

WHAT DOES ALL OF THAT GET YOU?

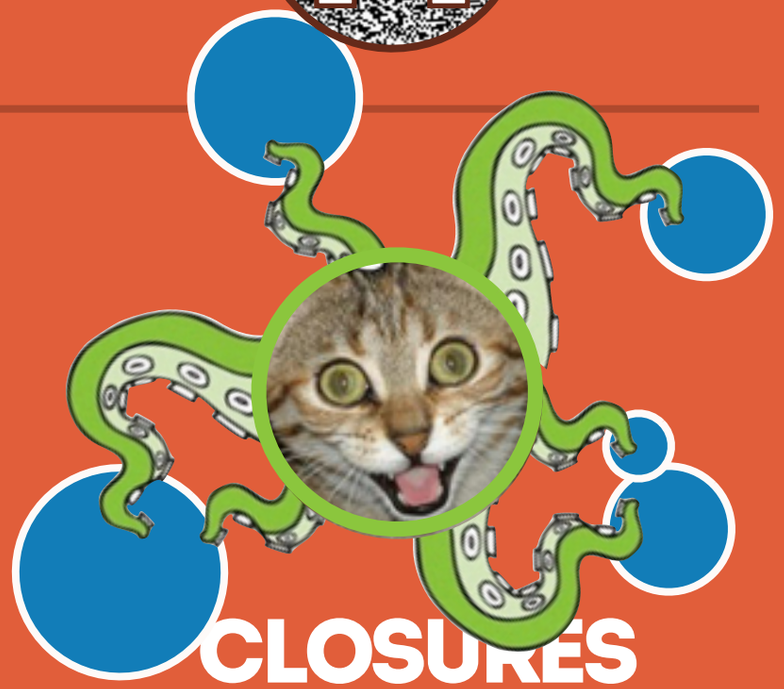
OR, GRAPHICALLY...



1
Right after creation



2
During execution



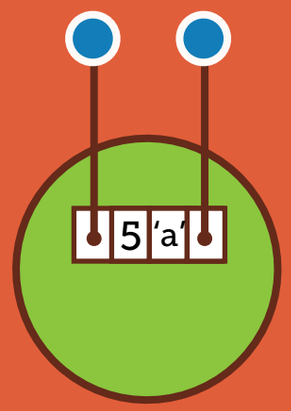
SPORES

CLOSURES

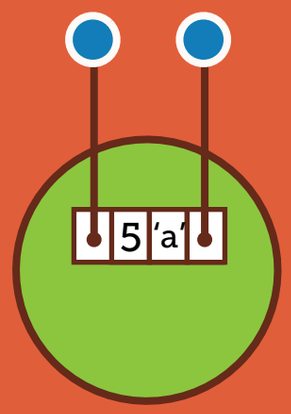
Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

WHAT DOES ALL OF THAT GET YOU?

OR, GRAPHICALLY...



1
Right after creation



SPORES

2
During execution

CLOSURES

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

Great.

SO NOW THAT WE HAVE SPORES, WHAT KIND OF Cool Patterns CAN THEY ENABLE?

Patterns ENABLED BY SPORES

STAGE AND SHIP

Build up a computation graph such that the behavior is represented as spores.

Once the computation has been built, it can be safely shipped to remote nodes.

COULD BE PIPELINE STAGES THAT ARE PLUGGED TOGETHER

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

Patterns ENABLED BY SPORES

FUNCTION-PASSING STYLE CONCURRENCY

Pass spores between concurrent entities. Compose spores with other spores to form larger, composite spores, and then send.

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

Patterns
ENABLED BY SPORES

HOT-SWAPPING ACTOR BEHAVIOR

Have a running actor and want to change its behavior.

Send it a spore and instruct the actor to use the spore from now on to process its incoming messages.

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

NEW PATTERNS
➡NEW FRAMEWORKS

SOME IDEAS FOR POTENTIAL FRAMEWORKS THAT SPORES MIGHT HELP

- Distributed Collections
- Distributed pipelines, stream processing
- Function-passing style frameworks

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

Cool.

Cool. **WHAT IF I CAPTURE A SOCKET?**

Cool.

WHAT IF I CAPTURE A SOCKET?

REALM OF RESEARCH

SPORES

two types:

1. **MAINLINE SPORES**
   proposed for inclusion in Scala 2.11
2. **SPORES WITH TYPE CONSTRAINTS**
   research project @EPFL,
   research paper in the works

Cool. **WHAT IF I CAPTURE A SOCKET?**

We don't have the means yet for frameworks to express these kinds of constraints or to enforce them when we create and compose spores.

WOULDN'T IT BE NICE IF WE COULD ADD THESE CONSTRAINTS, IN A FRIENDLY, AND COMPOSABLE WAY?

RESEARCH

Creating SPORES *w/ constraints*

Idea: KEEP TRACK OF CAPTURED TYPES
...at compile-time

```
spore { val x: Int = list.size; val a: ActorRef = this.sender
  (y: Int) => ...
} exclude[Actor]
```

The spore macro can synthesize precise types automatically for newly created spores:
(a whitebox macro)

SYNTHESIZED TYPE:

```
Spore[Int, ...] {
  type Excluded = NoCapture[Actor]
  type Facts = Captured[Int] with Captured[ActorRef]
}
```

RESEARCH

Composing SPORES *w/ constraints*

BASIC COMPOSITION OPERATORS

- andThen *(same as for regular functions)*
- compose

How do we synthesize the result type of `s1 andThen s2`?

RESULT TYPE SYNTHESIZED BY `andThen` MACRO

- type member `Facts` takes "union" of the facts of `s1` and `s2`
- type member `Excluded`: conjunction of excluded types, needs to check `Facts` to see if possible

RESEARCH

Example: Composing SPORES w/ constraints

```
val s1: Spore[Int, String] {
  type Excluded = NoCapture[Actor]
  type Facts = Captured[Int] with Captured[ActorRef]
} = ...
val s2: Spore[String, String] {
  type Excluded = NoCapture[RDD[Int]]
  type Facts = Captured[Actor]
}
s1 andThen s2  // does not compile
```

RESEARCH

Example: Composing SPORES w/ constraints

```
val s1: Spore[Int, String] {
  type Excluded = NoCapture[Actor]
  type Facts = Captured[Int] with Captured[ActorRef]
} = ...
val s2: Spore[String, String] {
  type Excluded = NoCapture[RDD[Int]]
  type Facts = Captured[Actor]
}
s1 andThen s2  // does not compile
```

RESEARCH

Example: Composing SPORES w/ constraints

```
val s1: Spore[Int, String] {
  type Excluded = NoCapture[Actor]
  type Facts = Captured[Int] with Captured[ActorRef]
} = ...
val s2: Spore[String, String] {
  type Excluded = NoCapture[RDD[Int]]
}
s1 andThen s2: Spore[Int, String] {
  type Excluded = NoCapture[Actor] with
NoCapture[RDD[Int]]
  type Facts = Captured[Int] with Captured[ActorRef]
}
```

RESEARCH

WHAT DO TYPE CONSTRAINTS BUY US?

- Stronger constraints checked at compile time (not "just" basic spore rules)
- Frameworks can make stronger assumptions about spores created by users.
- Confidence in consuming, creating, and composing spores:
  - Constraints accumulate monotonically
  - Constraints are never lost when composing spores
- Less brittleness.

AND NOW ONTO SOMETHING COMPLETELY DIFFERENT.

scala.pickling

PICKLES!

What is it?

https://github.com/scala/pickling

What is it?

PICKLING == SERIALIZATION == MARSHALLING

https://github.com/scala/pickling

What is it?

PICKLING == SERIALIZATION == MARSHALLING

VERY DIFFERENT FROM JAVA SERIALIZATION

https://github.com/scala/pickling

WAIT, WHY DO WE CARE?

https://github.com/scala/pickling

WAIT, WHY DO WE CARE?

Slow!

https://github.com/scala/pickling

WAIT, WHY DO WE CARE?

Slow!

Closed!

https://github.com/scala/pickling

WAIT, WHY DO WE CARE?

NOT SERIALIZABLE EXCEPTIONS AT RUNTIME

Closed!

https://github.com/scala/pickling

WAIT, WHY DO WE CARE?

- NOT SERIALIZABLE EXCEPTIONS AT RUNTIME
- CAN'T RETROACTIVELY MAKE CLASSES SERIALIZABLE

https://github.com/scala/pickling

ENTER: *Scala Pickling*

FAST: *Serialization code generated at compile-time and inlined at the use-site.*

FLEXIBLE: *Using typeclass pattern, retroactively make types serializable*

NO BOILERPLATE: *Typeclass instances generated at compile-time*

PLUGGABLE FORMATS: *Effortlessly change format of serialized data: binary, JSON, invent your own!*

TYPESAFE: *Picklers are type-specialized. Catch errors at compile-time!*

https://github.com/scala/pickling

What does it look like?

https://github.com/scala/pickling

What does it look like?

```
scala> import scala.pickling._
import scala.pickling._
```

https://github.com/scala/pickling

What does it look like?

```
scala> import scala.pickling._
import scala.pickling._

scala> import json._
import json._
```

https://github.com/scala/pickling

What does it look like?

```
scala> import scala.pickling._
import scala.pickling._

scala> import json._
import json._

scala> case class Person(name: String, age: Int)
defined class Person

scala> Person("John Oliver", 36)
res0: Person = Person(John Oliver,36)
```

https://github.com/scala/pickling

What does it look like?

```
scala> import scala.pickling._
import scala.pickling._

scala> import json._
import json._

scala> case class Person(name: String, age: Int)
defined class Person
```

```
scala> Person("John Oliver", 36)
res0: Person = Person(John Oliver,36)

scala> res0.pickle
res1: scala.pickling.json.JSONPickle =
JSONPickle({
  "tpe": "Person",
  "name": "John Oliver",
  "age": 36
})
```

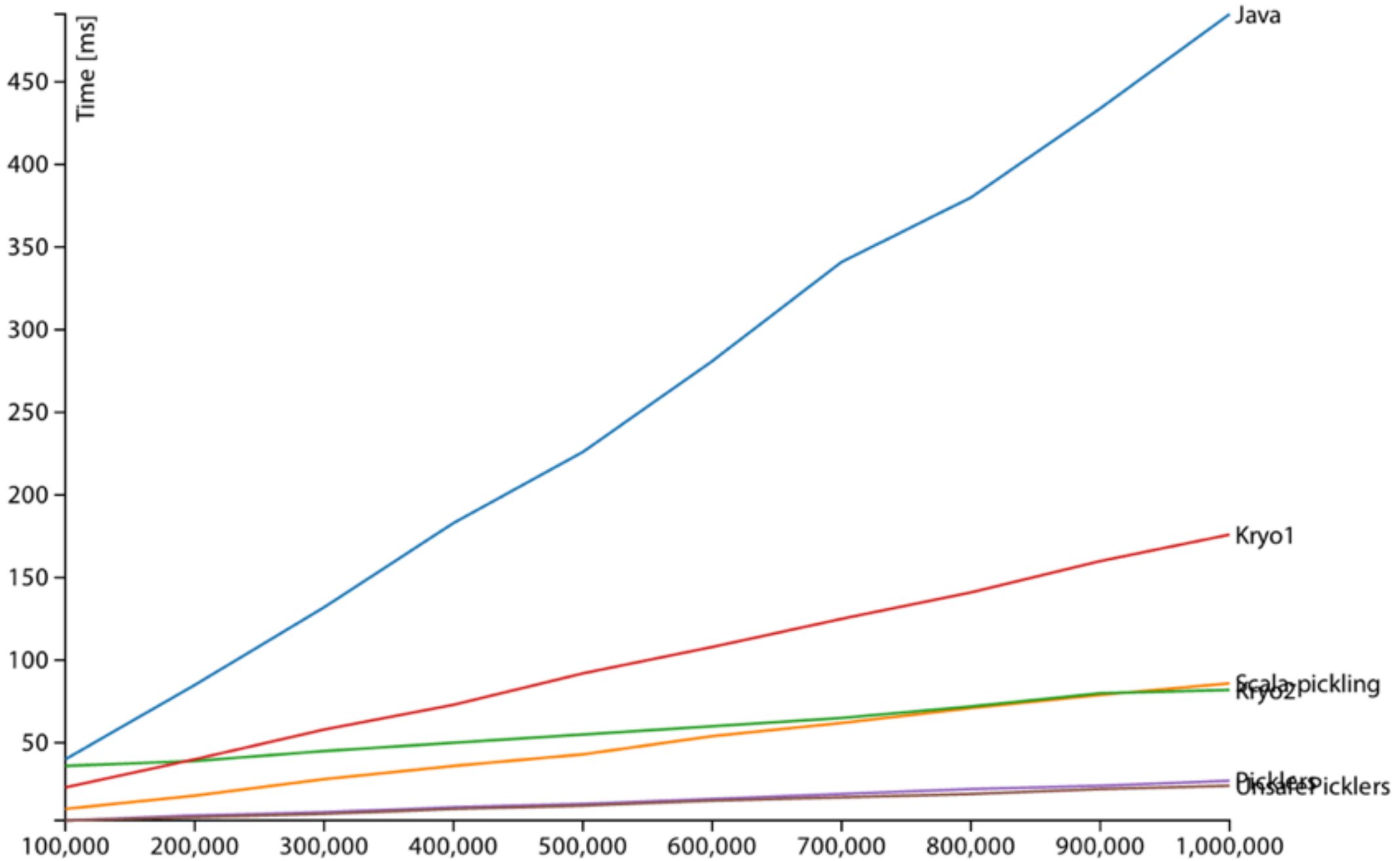
https://github.com/scala/pickling

AND...

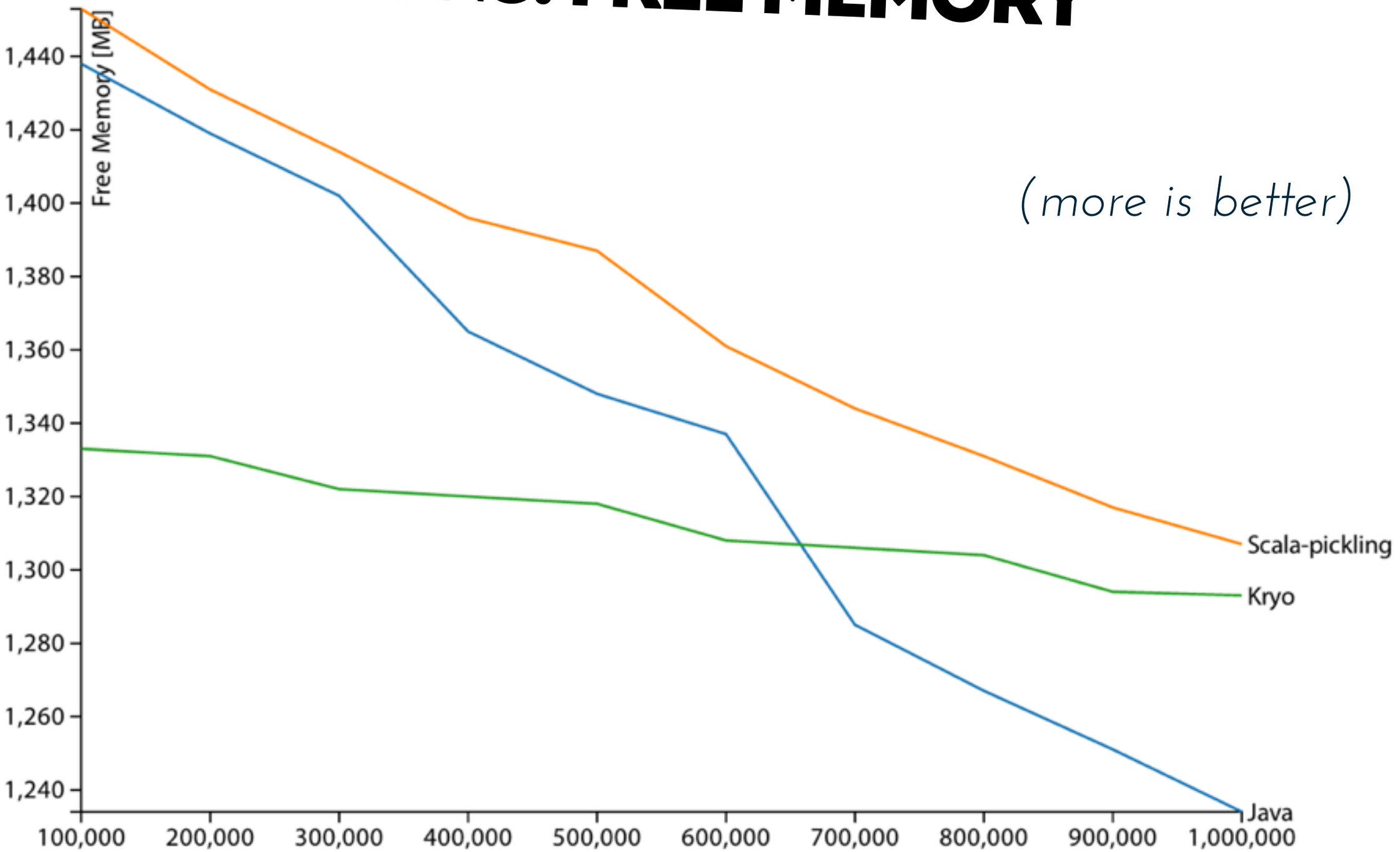
it's pretty fast

https://github.com/scala/pickling

COLLECTIONS: TIME

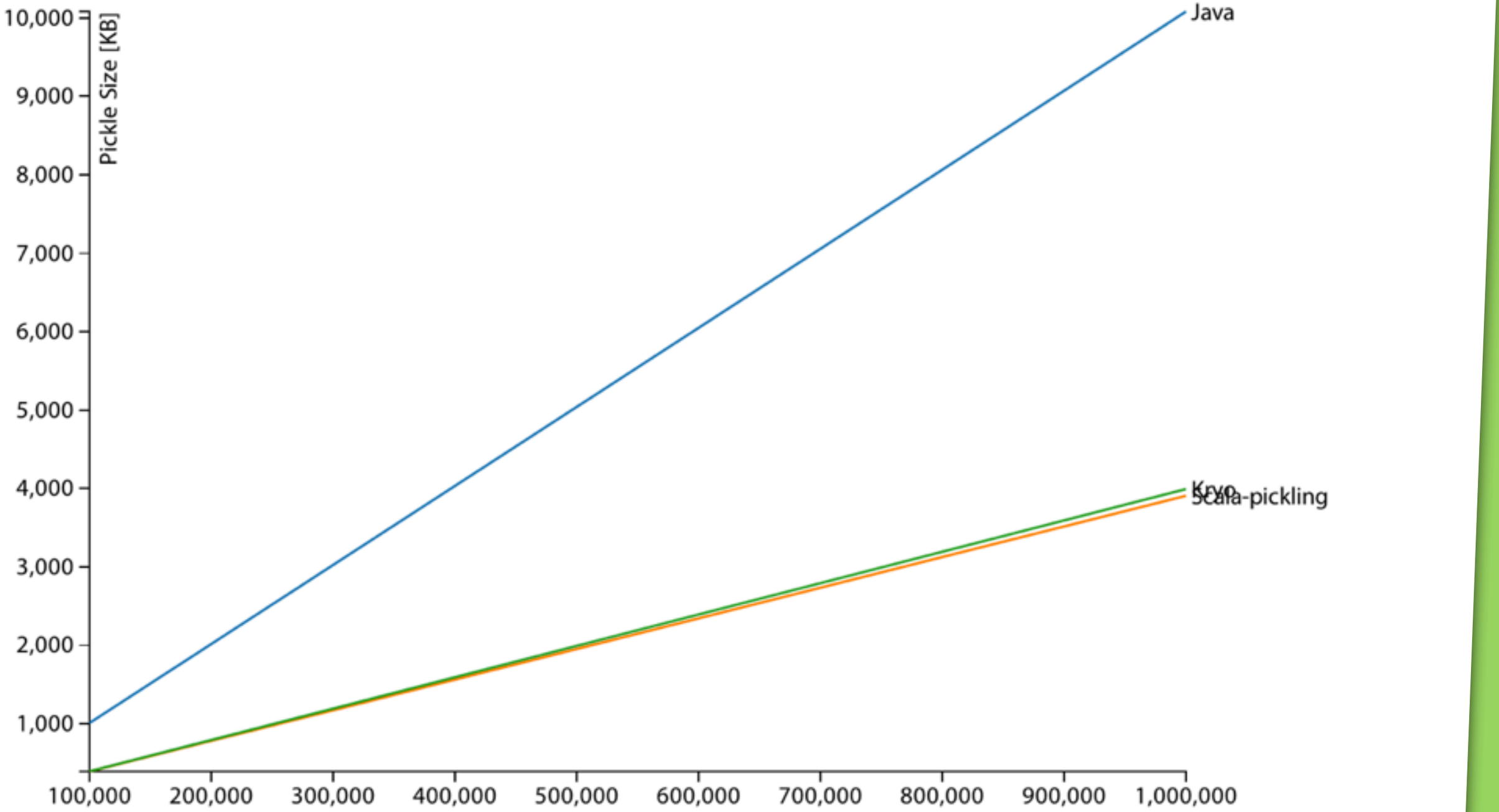


COLLECTIONS: FREE MEMORY

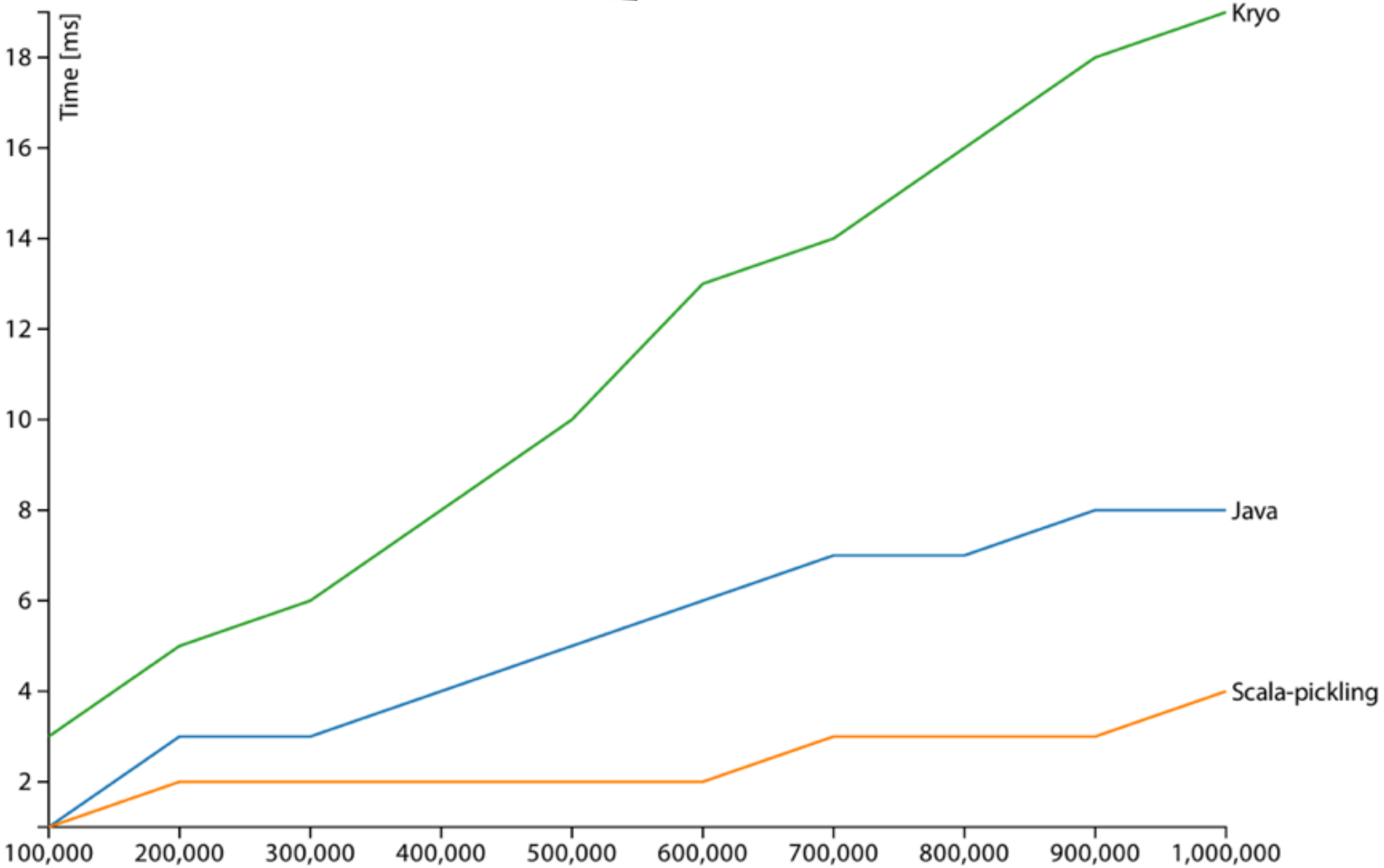


(more is better)

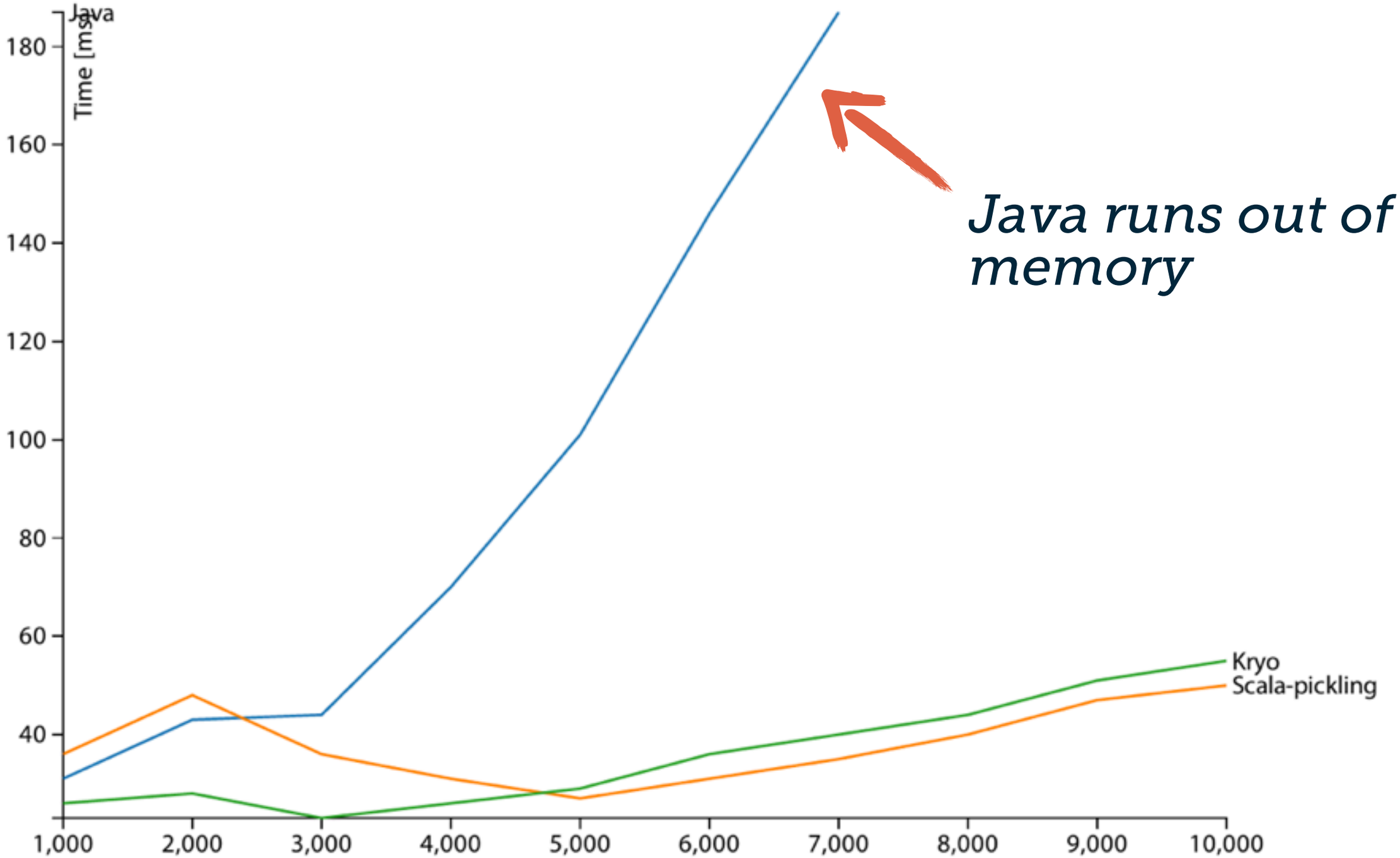
COLLECTIONS: SIZE



GEOTRELLIS: TIME

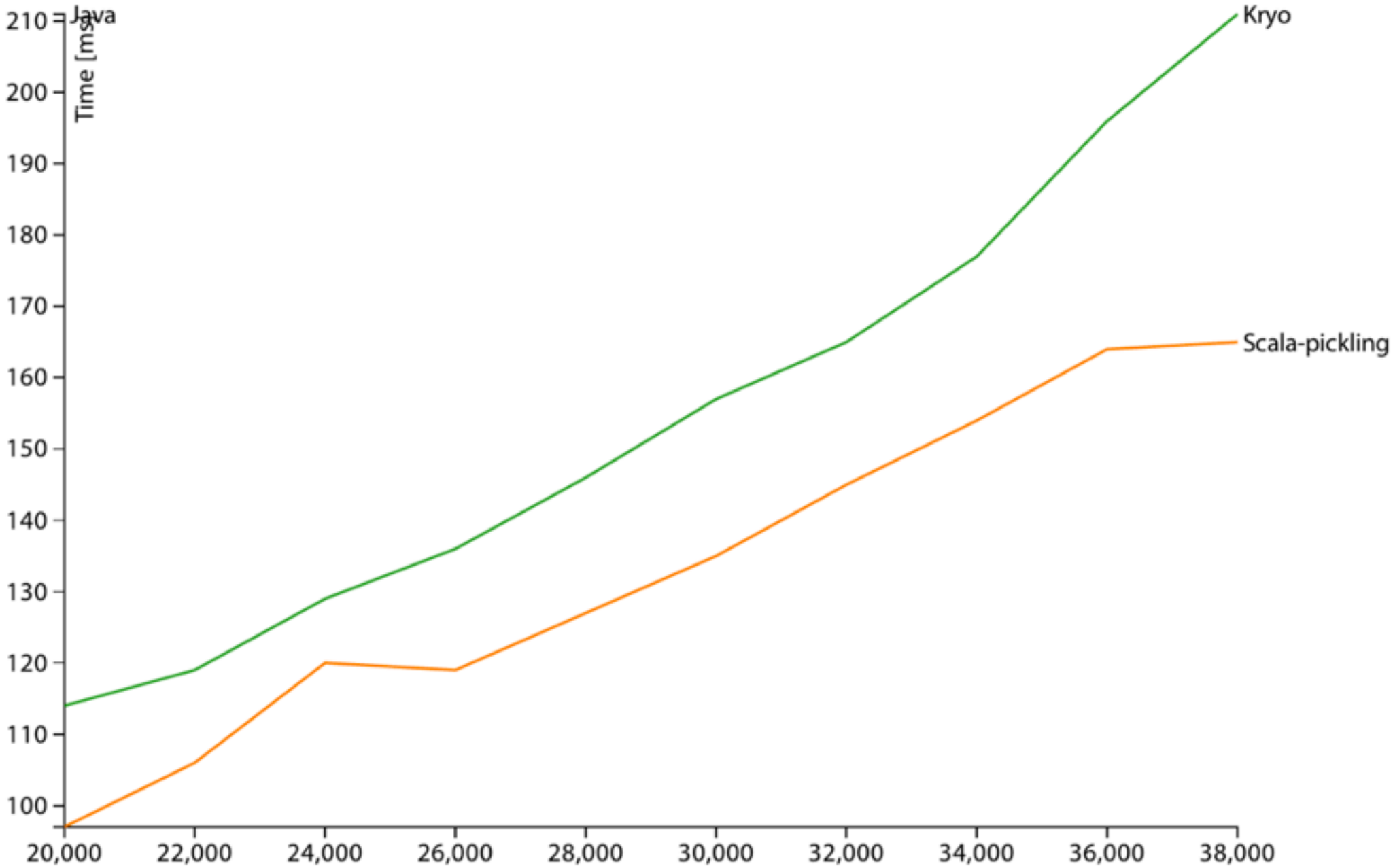


EVACTOR: TIME



Benchmarks

EVACTOR: TIME (NO JAVA, MORE EVENTS)



btw,

THAT'S JUST THE DEFAULT BEHAVIOR...

https://github.com/scala/pickling

btw,

THAT'S JUST THE DEFAULT BEHAVIOR...

YOU CAN REALLY CUSTOMIZE SCALA PICKLING TOO.

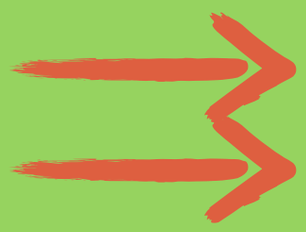
https://github.com/scala/pickling

Customizing PICKLING

PREVIOUS EXAMPLES USED DEFAULT BEHAVIOR

Generated picklers
Standard pickle format

PICKLING IS VERY CUSTOMIZABLE



Custom picklers for specific types
Custom pickle format

https://github.com/scala/pickling

Implicit PICKLERS

CUSTOMIZE WHAT YOU PICKLE!

```
case class Person(name: String, age: Int, salary: Int)

class CustomPersonPickler(implicit val format: PickleFormat) extends SPickler[Person] {
  def pickle(picklee: Person, builder: PBuilder): Unit = {
    builder.beginEntry(picklee)
    builder.putField("name", b =>
      b.hintTag(FastTypeTag.ScalaString).beginEntry(picklee.name).endEntry())
    builder.putField("age", b =>
      b.hintTag(FastTypeTag.Int).beginEntry(picklee.age).endEntry())
    builder.endEntry()
  }
}

implicit def genCustomPersonPickler(implicit format: PickleFormat) =
  new CustomPersonPickler
```

https://github.com/scala/pickling

PICKLE*Format*

OUTPUT ANY FORMAT!

```
trait PickleFormat {
  type PickleType <: Pickle
  def createBuilder(): PBuilder
  def createReader(pickle: PickleType, mirror: Mirror): PReader
}

trait PBuilder extends Hintable {
  def beginEntry(picklee: Any): PBuilder
  def putField(name: String, pickler: PBuilder => Unit): PBuilder
  def endEntry(): Unit
  def beginCollection(length: Int): PBuilder
  def putElement(pickler: PBuilder => Unit): PBuilder
  def endCollection(length: Int): Unit
  def result(): Pickle
}
```

https://github.com/scala/pickling

PICKLE Format EXAMPLE

TALK TO A CLOJURE APP

Output edn, Clojure's data transfer format.

```
scala> import scala.pickling._
import scala.pickling._

scala> import edn._
import edn._

scala> case class Person(name: String, kidsAges: Array[Int])
defined class Person

scala> val joe = Person("Joe", Array(3, 4, 13))
joe: Person = Person(Joe,[I@3d925789)

scala> joe.pickle.value
res0: String = #pickling/Person { :name "Joe" :kidsAges [3, 4, 13] }
```

toy builder implementation:
https://gist.github.com/heathermiller/5760171

STATUS

- Release 0.8.0 for Scala 2.10.2 https://github.com/scala/pickling
- ScalaCheck tests
- Support for cyclic object graphs, and most Scala types
- No support for inner classes, yet

Goal: Scala 2.11 as target

Plan:

- 1.0 release within the next few months
- Integration with sbt, Spark, and Akka, ...
- SIP for Scala 2.11

QUESTIONS?

Contact
heather.miller@epfl.ch
@heathercmiller