ORACLE®

Java SE: A Youthful Maturity

Danny Coward
Principal Engineer and Java Evangelist

The secrets of longevity ?

The secrets of longevity ?

What are the secrets for the longevity of the Java Platform?

What are the secrets for the longevity of the Java Platform?

What are the secrets for the longevity of the Java Platform?

Parallel programming

Scaling from low to high

.NET competitiveness

API breadth

Productivity

Web Services

Multiple JVM languages

Realtime characteristics

Performance

Ease of learning

What are the secrets for the longevity of the Java Platform?

Parallel programming

Scaling from low to high

.NET competitiveness

API breadth

Productivity

Web Services

Multiple JVM languages

Performance

Realtime characteristics

Ease of learning

Parallel programming →

Multiple JVM languages →

1.0 1.1 1.2 1.31.4 5.0 6 7 8 ...

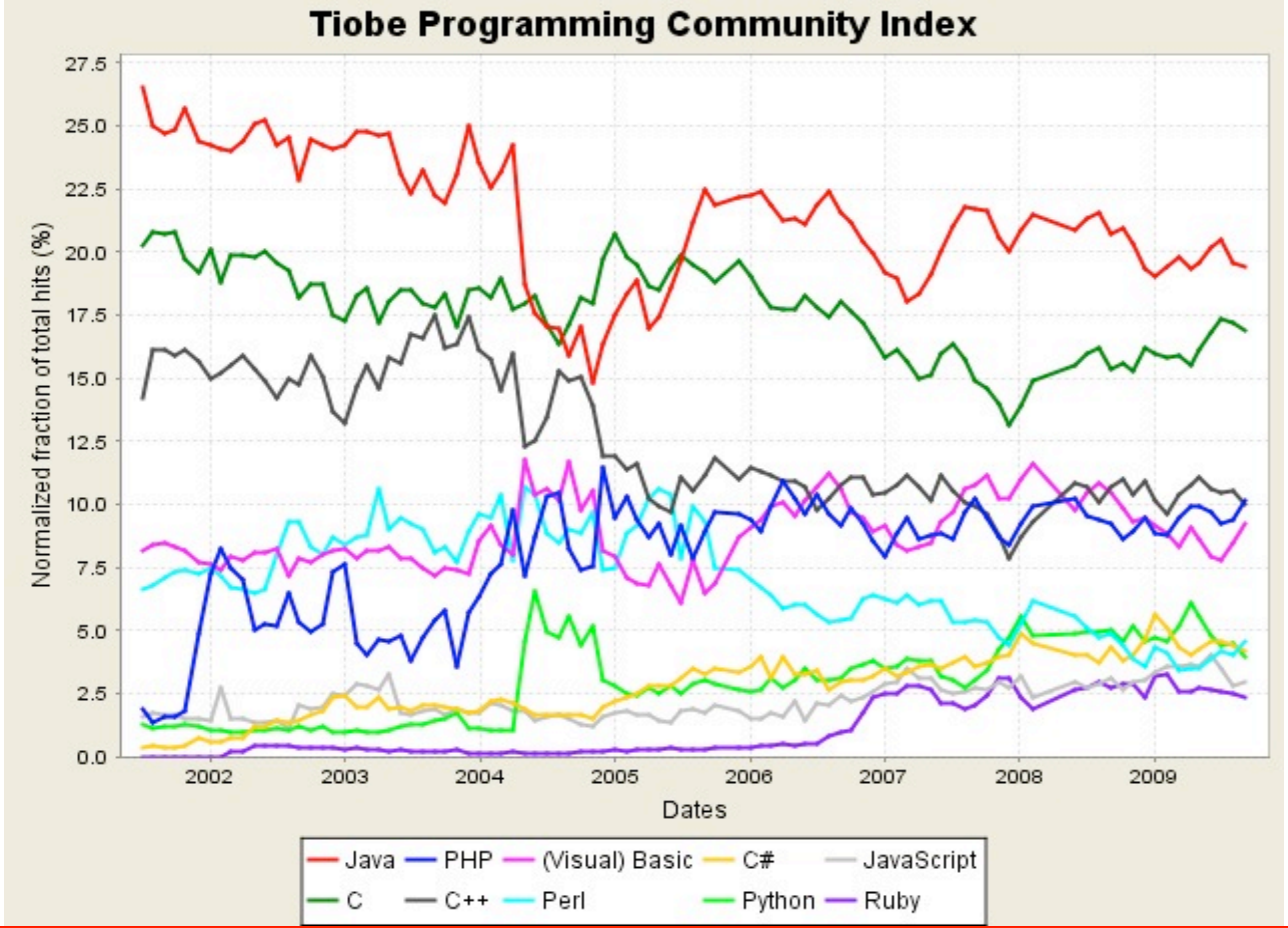
1996 1997 1998 2000 2002 2004 2006 2010 2011 2012 ... 2015

Java VM Specification, 1997

Java VM Specification, 1997

- The Java Virtual Machine knows nothing about the Java programming language, only of a particular binary format, the class file format.
- A class file contains Java Virtual Machine instructions (or bytecodes) and a symbol table, as well as other ancillary information.
- Any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine.
- Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java Virtual Machine as a delivery vehicle for their languages.
- In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages.

Trends in programming languages



Luck FScript

Bex Script foo Lisp Forth JRuby Funnel

CAL Drools Scala C# Pascal Sather Tea

Zigzag Rexx JudoScript Ada Dawn Mini

BeanShell JavaFX Script

v-language G Groovy TermWare

JavaScript Tiger iScript

Processing Icon LLP Oberon SALSA

Java™

E Smalltalk Tcl Basic PHP Hojo

JHCR Logo Jython

Yoix Yassl

Clojure WebL Prolog Piccola

Simkin Eiffel Scheme Phobos PLAN

JESS Correlate Pnuts Nice

Anvil Jickle Modula-2 ObjectScript Sleep Present

Scripting on Java SE 6 Platform

```
import javax.script.*

ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine jsEngine = m.getEngineByName("js");
```

Scripting on Java SE 6 Platform

```
import javax.script.*

ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine jsEngine = m.getEngineByName("js");


jsEngine.eval("print('Hello, world!')");
```

Scripting on Java SE 6 Platform

```
import javax.script.*

ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine jsEngine = m.getEngineByName("js");


InputStream is = this.getClass().getResourceAsStream("/scripts/hello.js");
Reader reader = new InputStreamReader(is);
jsEngine.eval(reader);
```

Scripting on Java SE 6 Platform

```
import javax.script.*

ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine jsEngine = m.getEngineByName("js");


jsEngine.eval("function sayHello() {" +
            "  println('Hello, world!');" +
            "}");

Invocable invocableEngine = (Invocable) jsEngine;
invocableEngine.invokeFunction("sayHello");
```

ORACLE

Languages ♥ Virtual Machines

- Programming languages need runtime support
    - Memory management / Garbage collection
    - Concurrency control
    - Security
    - Reflection
    - Debugging / Profiling
    - Standard libraries (collections, database, XML, etc)
- Traditionally, language implementers coded these themselves
- Many implementers now choose to target a VM to reuse infrastructure

“Java is fast because it runs on a VM”

- Major breakthrough was the advent of “Just In Time” compilers [fast]
  - Compile from bytecode to machine code at runtime
  - Optimize using information available at runtime only
- Simplifies static compilers
  - javac and ecj generate “dumb” bytecode and trust the JVM to optimize
  - Optimization is real, but invisible

Optimizations apply (more or less) to all languages

- Optimizations work on bytecode in .class files
- A compiler for any language – not just Java – can emit a .class file
- *All* languages can benefit from dynamic compilation and optimizations like inlining

HotSpot optimizations

HotSpot optimizations

compiler tactics

delayed compilation

Tiered compilation

on-stack replacement

delayed reoptimization

program dependence graph representation

static single assignment representation

proof-based techniques

exact type inference

memory value inference

memory value tracking

constant folding

reassociation

operator strength reduction

null check elimination

type test strength reduction

type test elimination

algebraic simplification

common subexpression elimination

integer range typing

flow-sensitive rewrites

conditional constant propagation

dominating test detection

flow-carried type narrowing

dead code elimination

HotSpot optimizations

compiler tactics
delayed compilation
Tiered compilation
on-stack replacement
delayed reoptimization
program dependence graph representation
static single assignment representation
proof-based techniques
exact type inference
memory value inference
memory value tracking
constant folding
reassociation
operator strength reduction
null check elimination
type test strength reduction
type test elimination
algebraic simplification
common subexpression elimination
integer range typing
flow-sensitive rewrites
conditional constant propagation
dominating test detection
flow-carried type narrowing
dead code elimination

language-specific techniques
class hierarchy analysis
devirtualization
symbolic constant propagation
autobox elimination
escape analysis
lock elision
lock fusion
de-reflection
speculative (profile-based) techniques
optimistic nullness assertions
optimistic type assertions
optimistic type strengthening
optimistic array length strengthening
untaken branch pruning
optimistic N-morphic inlining
branch frequency prediction
call frequency prediction
memory and placement transformation
expression hoisting
expression sinking
redundant store elimination
adjacent store fusion
card-mark elimination
merge-point splitting

HotSpot optimizations

compiler tactics
delayed compilation
Tiered compilation
on-stack replacement
delayed reoptimization
program dependence graph representation
static single assignment representation
proof-based techniques
exact type inference
memory value inference
memory value tracking
constant folding
reassociation
operator strength reduction
null check elimination
type test strength reduction
type test elimination
algebraic simplification
common subexpression elimination
integer range typing
flow-sensitive rewrites
conditional constant propagation
dominating test detection
flow-carried type narrowing
dead code elimination

language-specific techniques
class hierarchy analysis
devirtualization
symbolic constant propagation
autobox elimination
escape analysis
lock elision
lock fusion
de-reflection
speculative (profile-based) techniques
optimistic nullness assertions
optimistic type assertions
optimistic type strengthening
optimistic array length strengthening
untaken branch pruning
optimistic N-morphic inlining
branch frequency prediction
call frequency prediction
memory and placement transformation
expression hoisting
expression sinking
redundant store elimination
adjacent store fusion
card-mark elimination
merge-point splitting

loop transformations
loop unrolling
loop peeling
safepoint elimination
iteration range splitting
range check elimination
loop vectorization
global code shaping
inlining (graph integration)
global code motion
heat-based code layout
switch balancing
throw inlining
control flow graph transformation
local code scheduling
local code bundling
delay slot filling
graph-coloring register allocation
linear scan register allocation
live range splitting
copy coalescing
constant splitting
copy removal
address mode matching
instruction peepholing
DFA-based code generator

Inlining: Example

Inlining: Example

```
public interface FooHolder<T> {
    public T getFoo();
}
```

Inlining: Example

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
   private final T foo;

   public MyHolder(T foo32) {
      this.foo = foo;
   }

   public T getFoo() {
      return foo;
   }
}
```

Inlining: Example

```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
          implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

Inlining: Example

```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
          implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

```
public String getString(FooHolder<String> holder) {
   if (holder == null)
      throw new NullPointerException("You dummy.");
   else
      return holder.getFoo();
}
```

Inlining: Example

```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
           implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

```
public String getString(FooHolder<String> holder) {
   if (holder == null)
      throw new NullPointerException("You dummy.");
   else
      return holder.getFoo();
}
...
public String foo(String x) {
   FooHolder<String> myFooHolder = new MyHolder<String>(x);
   return getString(myFooHolder);
}
```

Inlining: Example

```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
          implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

```
public String getString(FooHolder<String> holder) {
   if (holder == null)
      throw new NullPointerException("You dummy.");
   else
      return holder.getFoo();
}
...
public String foo(String x) {
   FooHolder<String> myFooHolder = new MyHolder<String>(x);
   return getString(myFooHolder);
}
```

Step 1
Inline getString() call

Inlining: Example

```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
          implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

```
public String getString(FooHolder<String> holder) {
   if (holder == null)
      throw new NullPointerException("You dummy.");
   else
      return holder.getFoo();
}
...
public String foo(String x) {
   FooHolder<String> myFooHolder = new MyHolder<String>(x);
   if (myFooHolder == null)
      throw new NullPointerException("You dummy.");
   else
      return myFooHolder.getFoo();
}
```

Inlining: Example

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T>
            implements FooHolder<T> {
    private final T foo;
    public MyHolder(T foo) { this.foo = foo; }
    public T getFoo() { return foo; }
}
```

```
public String getString(FooHolder<String> holder) {
    if (holder == null)
        throw new NullPointerException("You dummy.");
    else
        return holder.getFoo();
}
...
public String foo(String x) {
    FooHolder<String> myFooHolder = new MyHolder<String>(x);
    if (myFooHolder == null)
        throw new NullPointerException("You dummy.");
    else
        return myFooHolder.getFoo();
}
```

Step 2
Dead code elimination

Inlining: Example

```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
            implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

```
public String getString(FooHolder<String> holder) {
   if (holder == null)
      throw new NullPointerException("You dummy.");
   else
      return holder.getFoo();
}
...
public String foo(String x) {
   FooHolder<String> myFooHolder = new MyHolder<String>(x);
   return myFooHolder.getFoo();
}
```

Inlining: Example

```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
          implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

```
public String getString(FooHolder<String> holder) {
   if (holder == null)
      throw new NullPointerException("You dummy.");
   else
      return holder.getFoo();
}
...
public String foo(String x) {
   FooHolder<String> myFooHolder = new MyHolder<String>(x);
   return myFooHolder.getFoo();
}
```

Step 3

Type sharp and inline

Inlining: Example

```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
            implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

```
public String getString(FooHolder<String> holder) {
   if (holder == null)
      throw new NullPointerException("You dummy.");
   else
      return holder.getFoo();
}
...
public String foo(String x) {
   FooHolder<String> myFooHolder = new MyHolder<String>(x);
   return myFooHolder.foo;
}
```

Inlining: Example

```
public String getString(FooHolder<String> holder) {
   if (holder == null)
      throw new NullPointerException("You dummy.");
   else
      return holder.getFoo();
}
...
public String foo(String x) {
   FooHolder<String> myFooHolder = new MyHolder<String>(x);
   return myFooHolder.foo;
}
```

```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
         implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

Step 4

Escape analysis and scalar replacement

Inlining: Example

```
public String getString(FooHolder<String> holder) {
   if (holder == null)
      throw new NullPointerException("You dummy.");
   else
      return holder.getFoo();
}
...
public String foo(String x) {
   return x;
}
```

```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
            implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

Inlining: Example

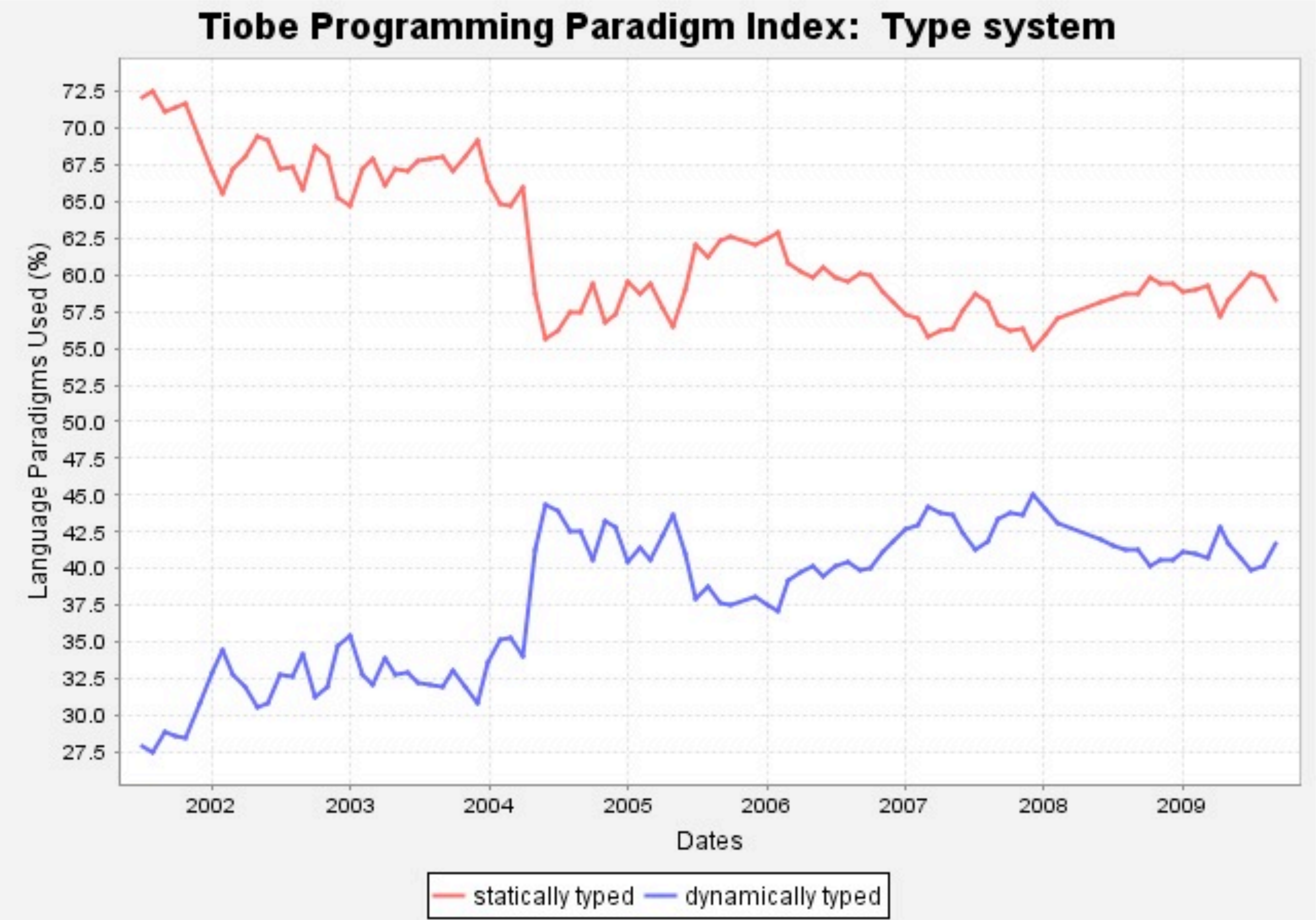
```
public interface FooHolder<T> {
   public T getFoo();
}

public class MyHolder<T>
            implements FooHolder<T> {
   private final T foo;
   public MyHolder(T foo) { this.foo = foo; }
   public T getFoo() { return foo; }
}
```

```
public String getString(FooHolder<String> holder) {
   if (holder == null)
      throw new NullPointerException("You dummy.");
   else
      return holder.getFoo();
}
...
public String foo(String x) {
   return x;
}
```

```
public String foo(String x) {
   FooHolder<String> myFooHolder = new MyHolder<String>(x);
   return getString(myFooHolder);
}
```

✗

Different kinds of languages



If we could make one change to the JVM to improve life for dynamic languages, what would it be?

If we could make one change to the JVM to improve life for dynamic languages, what would it be?

More flexible method calls

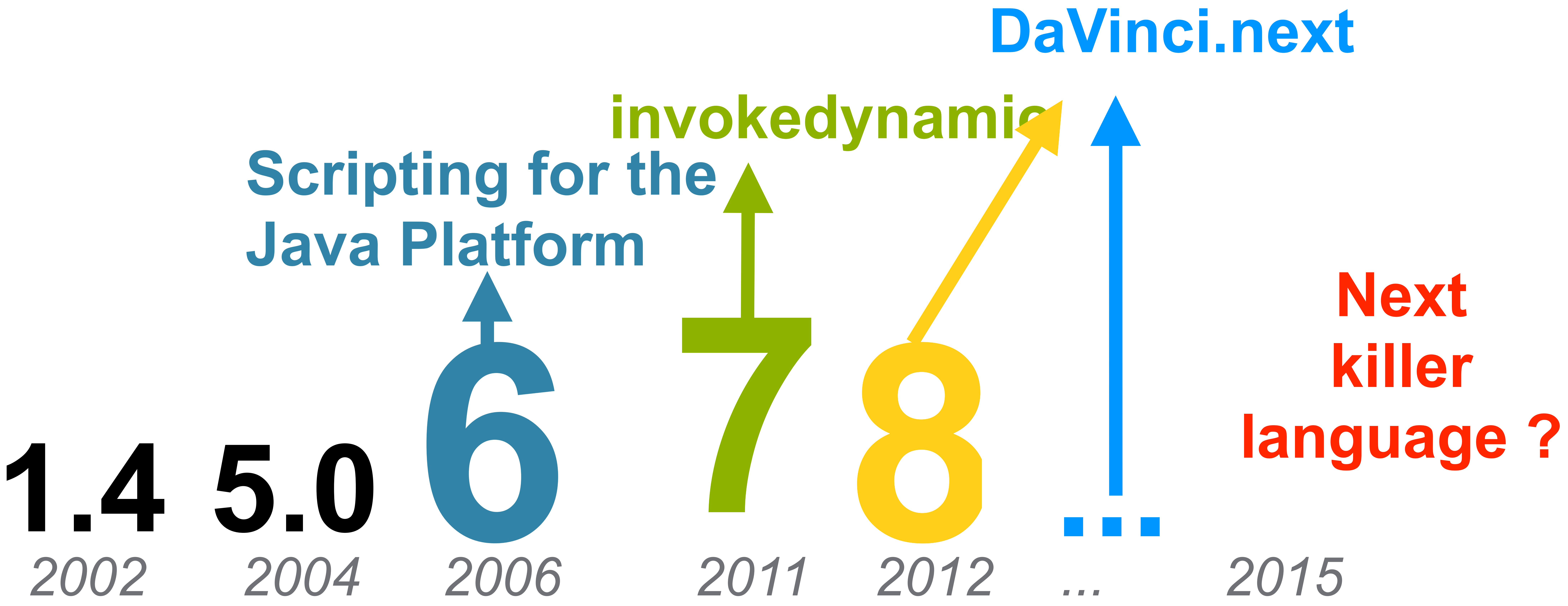
More flexible method calls

- The invokevirtual bytecode performs a method call
- Its behavior is Java-like and fixed
- Other languages need custom behavior
- Idea: let some “language logic” determine the behavior of a JVM method call
- Invention: the invokedynamic bytecode
  - VM asks some “language logic” how to call a method
  - Language logic gives an answer, and decides if it needs to stay in the loop

What's next? Da Vinci projects

- The Da Vinci Machine Project continues
- Community contributions:
  - Continuations
  - Coroutines
  - Hotswap
  - Tailcalls
  - Interface injection
- Gleams in our eyes:
  - Object "species" (for splitting classes more finely)
  - Tuples and value types (for using registers more efficiently)
  - Advanced array types (for using memory more efficiently)

Multiple JVM languages



Traffic Jams

Raise the speed limit ?

Traffic Jams

~~Raise the speed limit ?~~

Build more lanes !

Traffic Jams

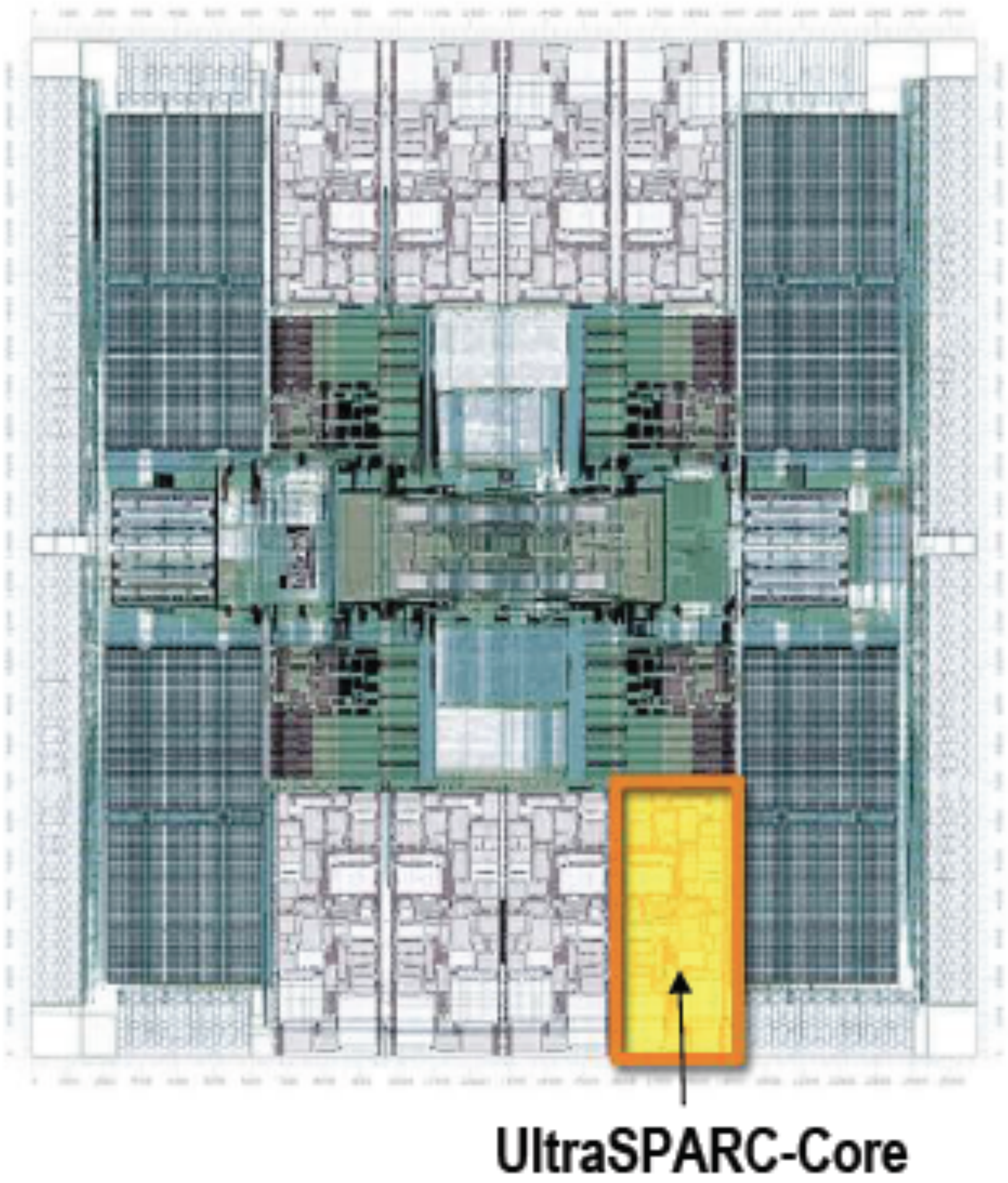
~~Raise the speed limit ?~~

Build more lanes !

The rise of multi-core/
multi-processor architectures

Multi-core, Multi Processor Servers

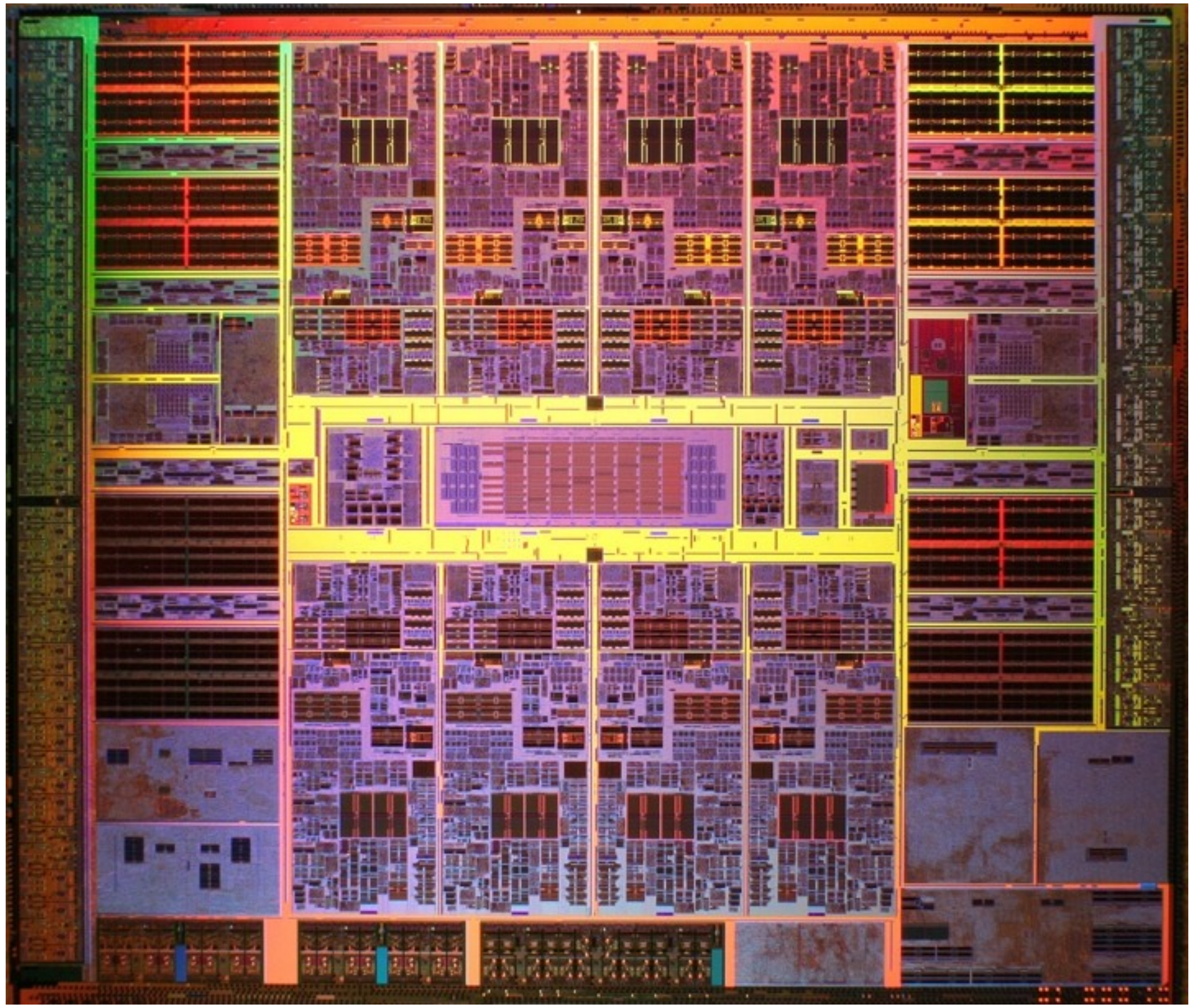
Multi-core, Multi Processor Servers



Niagara 1 (2005)

8 x 4 = **32**

Multi-core, Multi Processor Servers



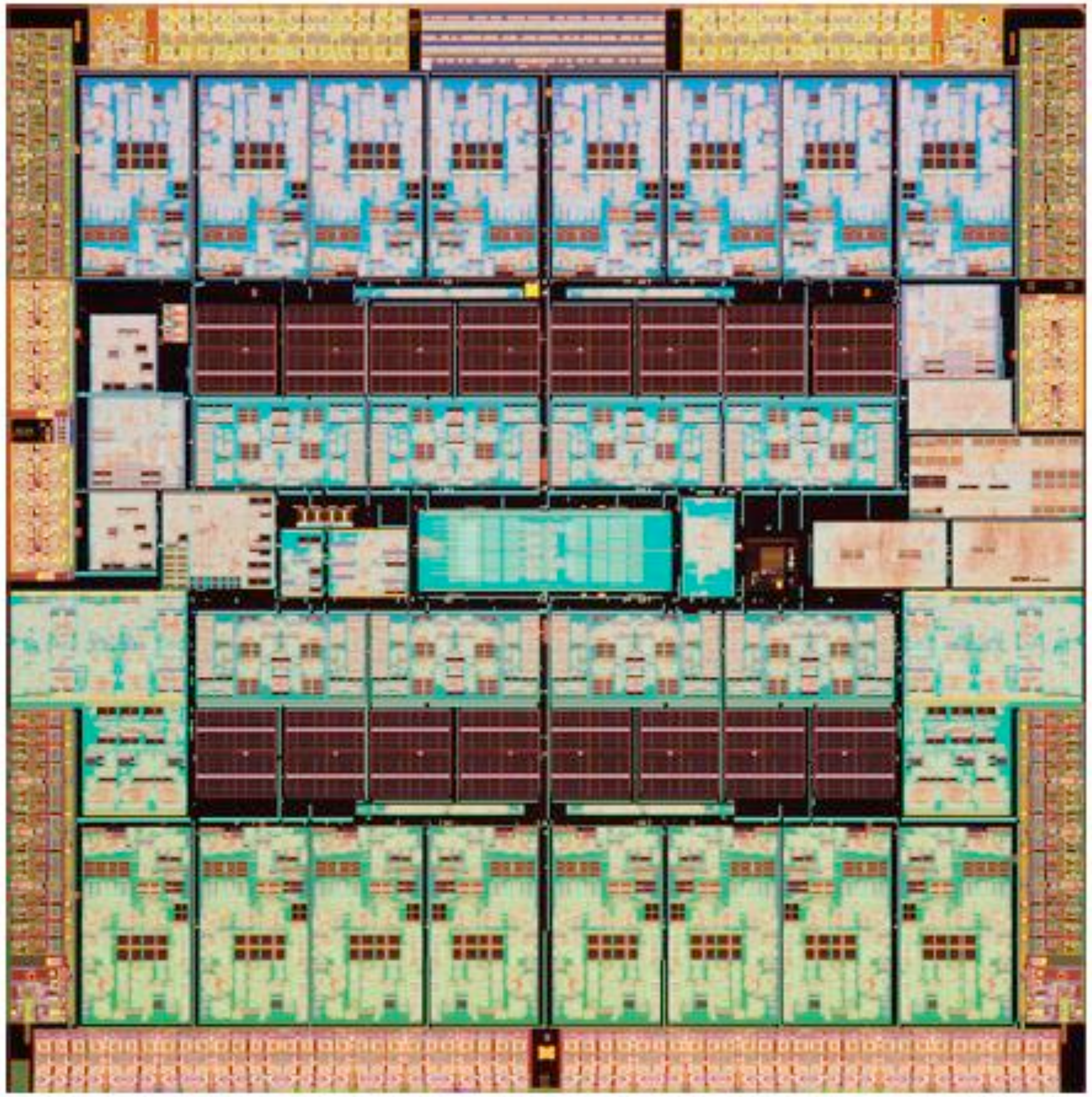
Niagara 1 (2005)

8 x 4 = **32**

Niagara 2 (2007)

8 x 8 = **64**

Multi-core, Multi Processor Servers



Niagara 1 (2005)

8 x 4 = **32**

Niagara 2 (2007)

8 x 8 = **64**

Rainbow Falls (Now)

16 x 8 = **128**

Multicore clients

Desktop ... notepad ... phone

2 ... 4 .....

8 .....

2002 2004 2006 2008 2010

ORACLE

Parallel Programming

- The runtime only has a partial view of how to optimize your application

Parallel Programming

- The runtime only has a partial view of how to optimize your application
- It needs your help so it knows how to parallelize your application

Concurrency APIs for developers: JDK 1.5

- Originally developed for JDK 1.5
- Result of work of JSR 166
- API level toolkit for concurrent programming
  - Locks
  - Threadpools
  - Blocking queues
- Found in `java.util.concurrent.*`

Concept of Fork/Join Framework

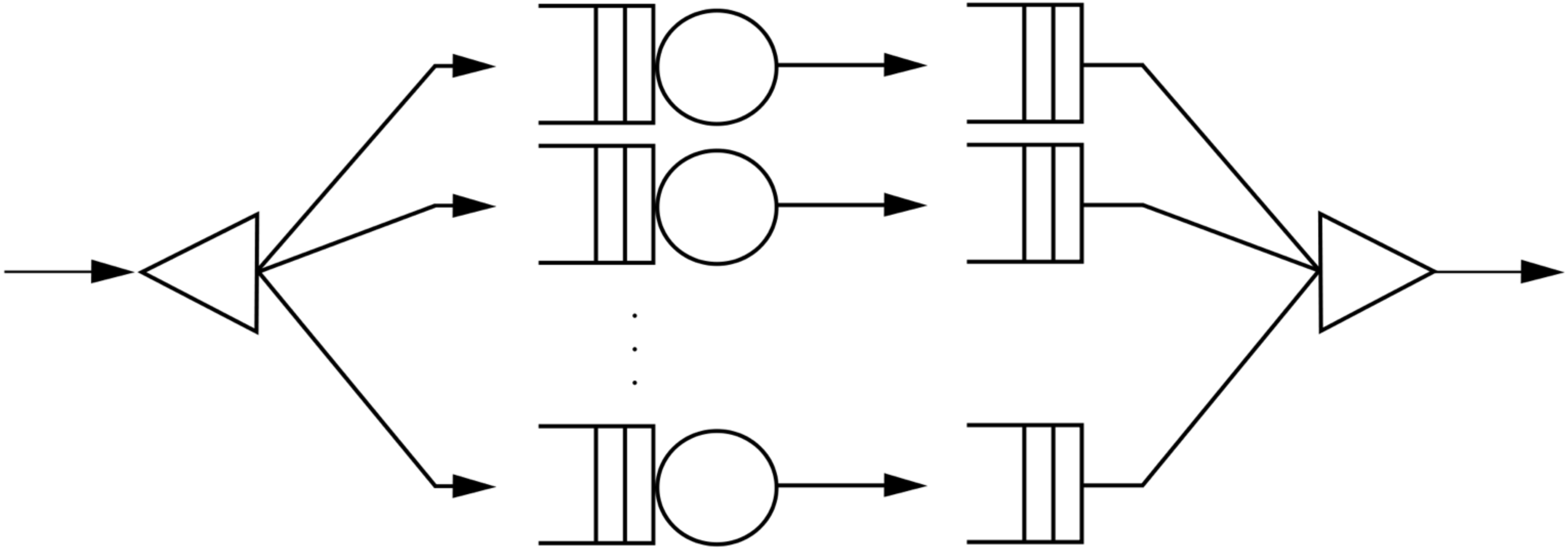


Image: Gareth Jones

Concept of Fork/Join Framework

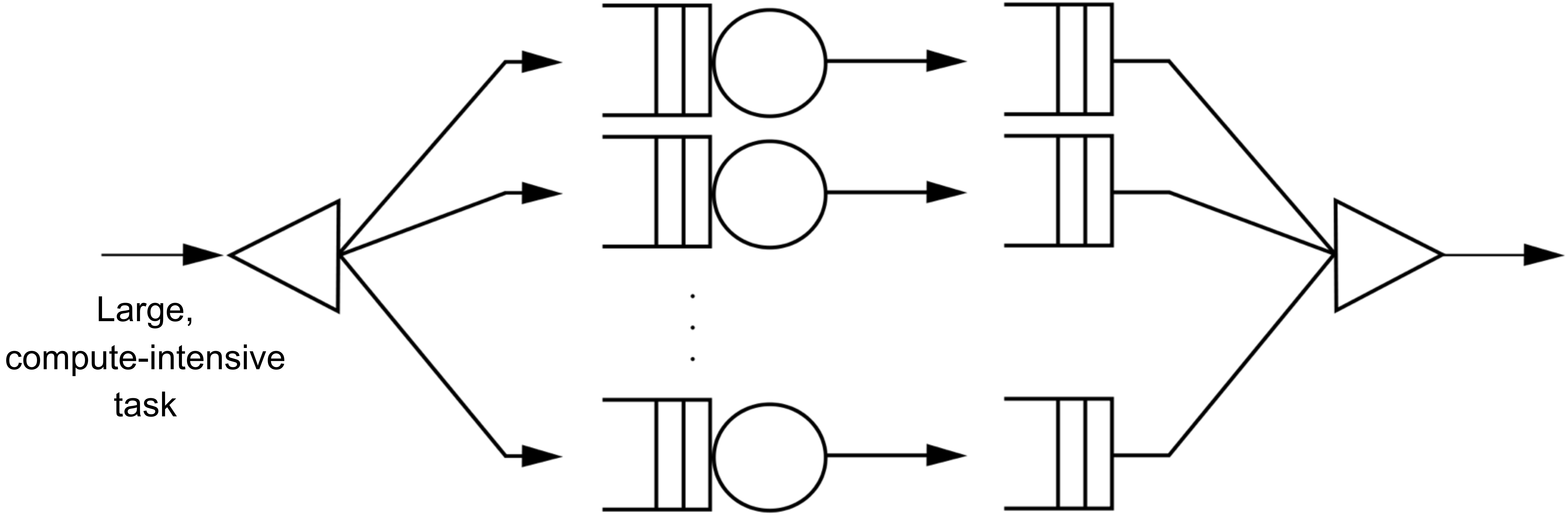
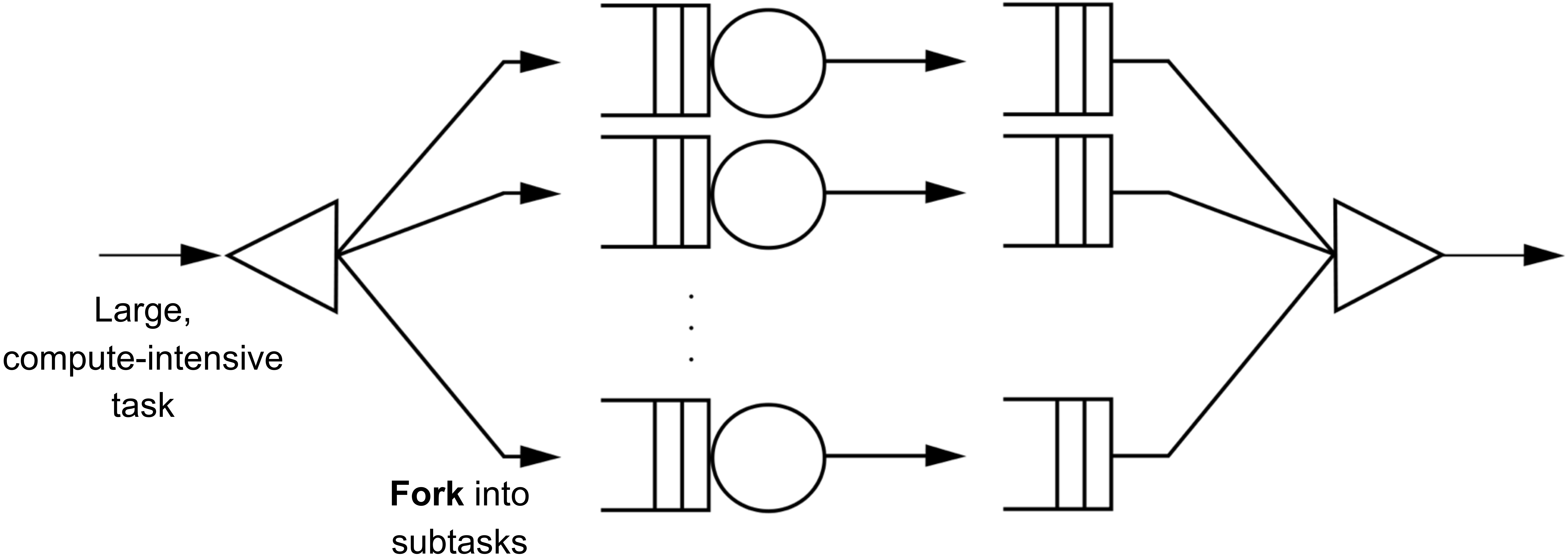


Image: Gareth Jones

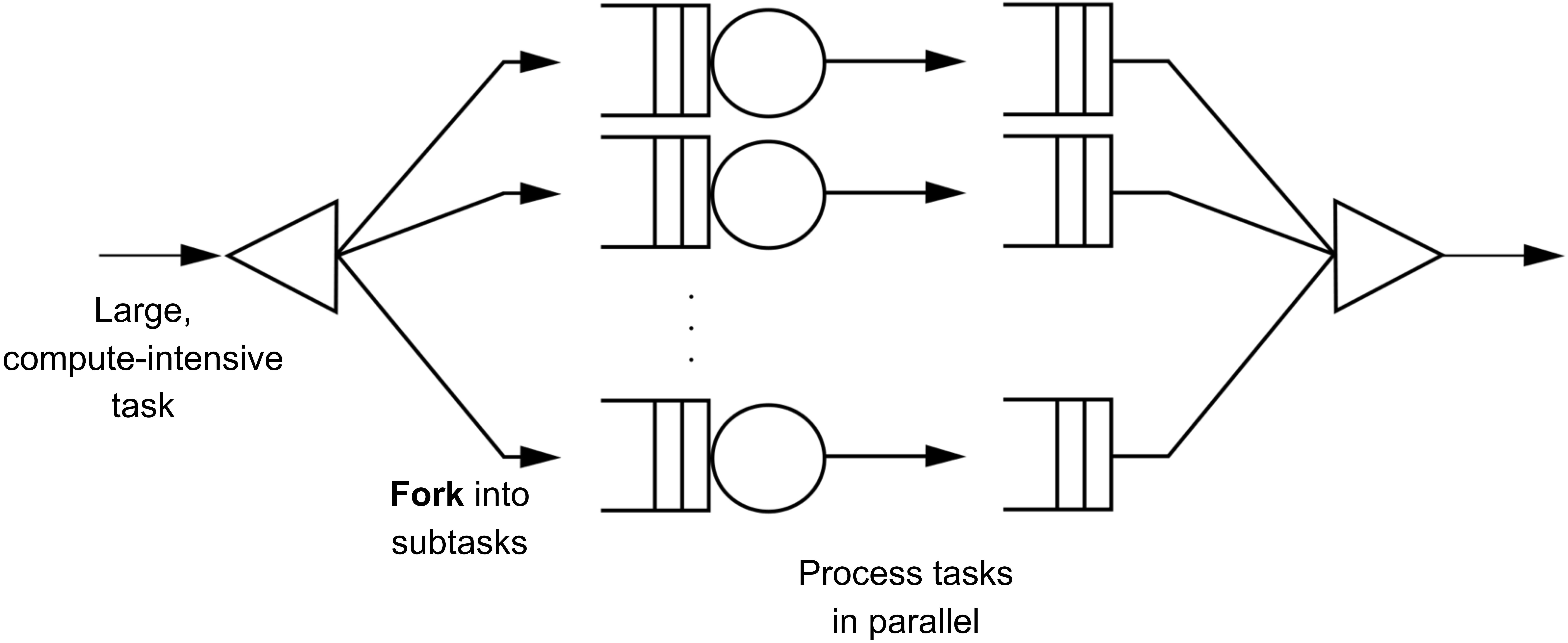
Concept of Fork/Join Framework


Large, compute-intensive task

Fork into subtasks

Image: Gareth Jones

Concept of Fork/Join Framework

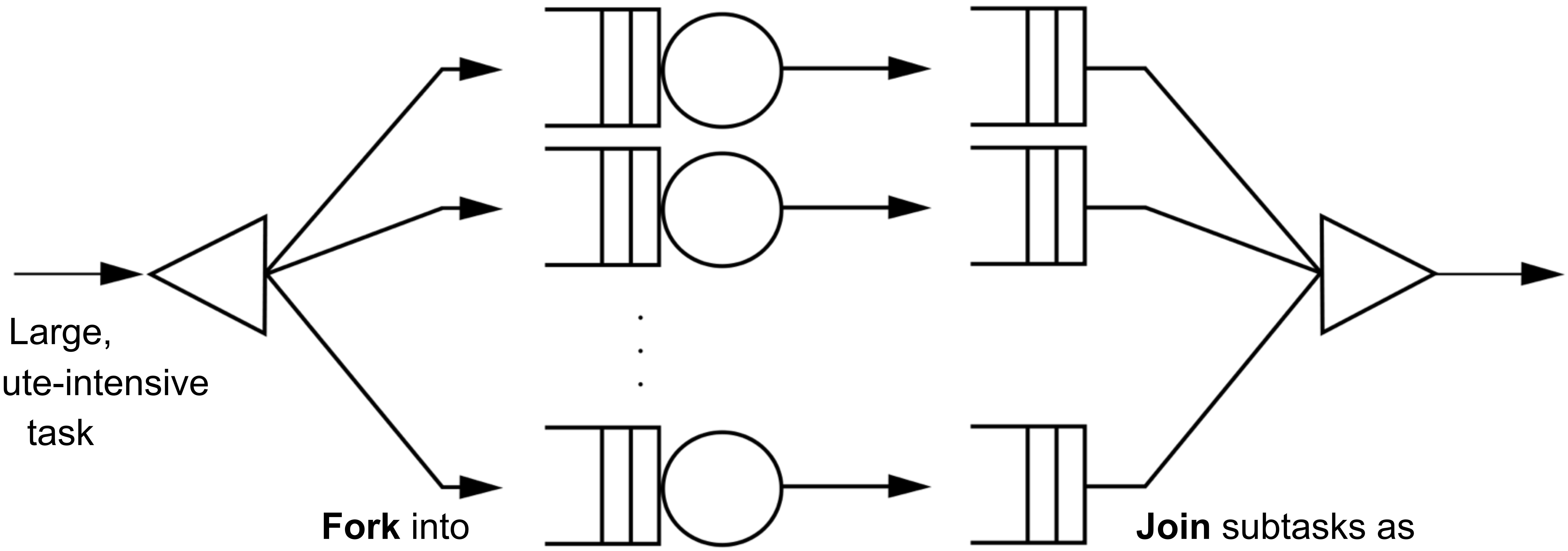

Large,
compute-intensive
task

Fork into
subtasks

Process tasks
in parallel

Image: Gareth Jones

Concept of Fork/Join Framework



Large, compute-intensive task

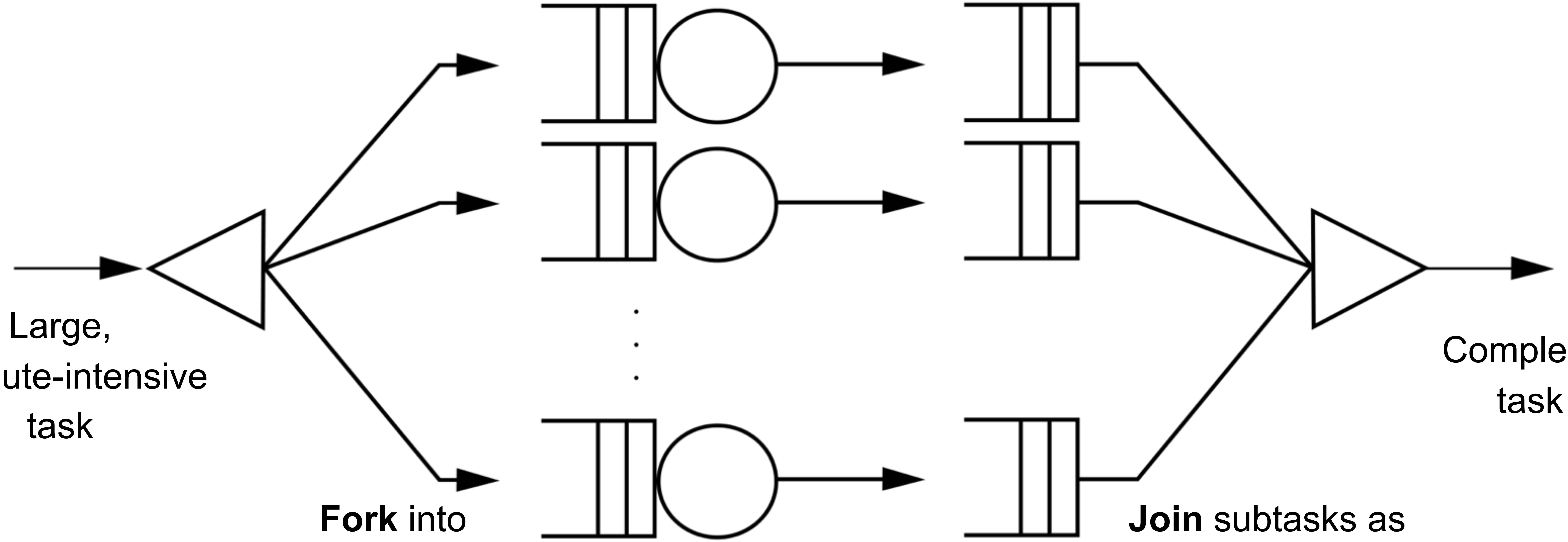
Fork into subtasks

Process tasks in parallel

Join subtasks as they complete

Image: Gareth Jones

Concept of Fork/Join Framework



Large,
compute-intensive
task

Fork into
subtasks

Process tasks
in parallel

Join subtasks as
they complete

Completed
task

Image: Gareth Jones

Basic Idea

```
if (my portion of the work is small enough) {
  do the work directly
} else {
  split my work into two pieces
  invoke the two pieces and wait for the results
}
```

Fork Join Framework API

- **`java.util.concurrent.ForkJoinPool`**
  - Special class for managing tasks that will execute in parallel
  - Submit new tasks
  - Manage lifecycle of tasks
  - Monitor task execution
- **`java.util.concurrent.ForkJoinTask`**
  - Abstract base class encapsulating task to run concurrently
  - Like a lightweight thread
  - Typically use RecursiveTask or RecursiveAction

Example: Blurring an image

```
public class ForkBlur {
    private int[] mSource;
    private int mStart;
    private int mLength;
    private int[] mDestination;

    private int mBlurWidth = 15; // Processing window size, should be odd.

    public ForkBlur(int[] src, int start, int length, int[] dst) {
       mSource = src;
       mStart = start;
       mLength = length;
       mDestination = dst;
    }

    ...
```

Example: Blurring an image

```
public class ForkBlur {

    ...
        // this is the heavy lifting
     protected void computeDirectly() {
        int sidePixels = (mBlurWidth - 1) / 2;
        for (int index = mStart; index < mStart + mLength; index++) {
           // Calculate average.
           float rt = 0, gt = 0, bt = 0;
           for (int mi = -sidePixels; mi <= sidePixels; mi++) {
              int mindex = Math.min(Math.max(mi + index, 0), mSource.length - 1);
              int pixel = mSource[mindex];
              rt += (float)((pixel & 0x00ff0000) >> 16) / mBlurWidth;
              gt += (float)((pixel & 0x0000ff00) >>  8) / mBlurWidth;
              bt += (float)((pixel & 0x000000ff) >>  0) / mBlurWidth;
        }

        // Re-assemble destination pixel.
        int dpixel = (0xff000000     ) |
                  (((int)rt) << 16) |
                  (((int)gt) <<  8) |
                  (((int)bt) <<  0);
        mDestination[index] = dpixel;
     }

     ...
 }
```

Heavy lifting in parallel using Fork/Join

```
public class ForkBlur extends RecursiveAction {

  protected void compute() {
   // use the Fork/Join pattern here
  }
    ...

     protected void computeDirectly() {
         // this is still the heavy lifting

     }

     ...

 }
```

Basic concept of Fork/Join

```
if (my portion of the work is small enough) {
  do the work directly
} else {
  split my work into two pieces
  invoke the two pieces and wait for the results
}
```

Application of Fork/Join

```
protected static int sThreshold = 100000;

protected void compute() {

   // is my portion of the work is small enough ?
   if (mLength < sThreshold) {

     // just do it
     computeDirectly();
     return;
   } else {

     // split my work into two pieces
     int split = mLength / 2;

     invokeAll(
    new ForkBlur(mSource, mStart, split, mDestination),
      new ForkBlur(mSource, mStart + split, mLength - split, mDestination)
    );

   }
}
```

Running the code

```
// source image pixels are in src
// destination image pixels are in dst
ForkBlur fb = new ForkBlur(src, 0, src.length, dst);

ForkJoinPool pool = new ForkJoinPool();

// now the work can be executed in parallel
pool.invoke(fb);
```

Parallel Programming in Java today

Ease	Supported by
Possible but very tricky	Thread API
Well-supported for specialized purposes	Concurrency APIs with Fork/Join
Commonplace	Not yet...

```
class Student {
   String name;
   int gradYear;
   double score;
}
```

```
class Student {
   String name;
   int gradYear;
   double score;
}

Collection<Student> students = ...;
```

```
Collection<Student> students = ...;
```

```
Collection<Student> students = ...;

double max = Double.MIN_VALUE;
for (Student s : students) {
    if (s.gradYear == 2010)
        max = Math.max(max, s.score);
}
```

```
Collection<Student> students = ...;

   double max = students.filter(new Predicate<Student>() {
                        public boolean op(Student s) {
                           return s.gradYear == 2010;
                                      }
                     }).map(new Extractor<Student,Double>() {
                      public Double extract(Student s) {
                                return s.score;
                                      }
                                }).max();
```

```
Collection<Student> students = ...;

  ddouble max = students.filter(new Predicate<Student>() {
                     public boolean op(Student s) {
                        return s.gradYear == 2010;
                            }
               }).map(new Extractor<Student,Double>() {
                   public Double extract(Student s) {
                            return s.score;
                            }
                       }).max();

double max =                       // Lambda expressions
   students.filter(  #{   Student  s -> s.gradYear == 2010 })
                .map(#{   Student s -> s.score })
                  .max();
```

Inner classes are imperfect closures

- Bulky syntax
- Can’t capture non-final local variables
- Transparency issues: meaning of return, break, continue, this
- No non-local control flow operators

Single Abstract Method (SAM) Types

```
public interface CallbackHandler {
       // single abstract method
    public void callback(Context c);

}
```

- Lots of examples in the Java SE APIs
  - `Runnable, Callable, EventHandler, Comparator...`

ORACLE

Single Abstract Method (SAM) Types

```
foo.doSomething(new CallbackHandler() {
                 public void callback(Context c) {
                    System.out.println("callback");
                 }
              });
```

- Noise:Work = 5:1
- Lambda grows out of the idea of making callback objects easier

Single Abstract Method (SAM) Types

```
foo.doSomething(new CallbackHandler() {
                 public void callback(Context c) {
                    System.out.println("callback");
                 }
              });
```

```
    // with Lambda
  foo.doSomething(
    #{ Context c -> System.out.println("pippo") };
   );
```

A Lambda expression with one parameter, one statement statement list block, void return type, no checked exceptions

More examples

```
#{ Context c -> System.out.println("pippo") };

#{ -> 42 }

#{ int x -> x + 1 }
```

Target Typing

Rule #1: Only in a context where it can be converted to a SAM type

```
CallBackHandler cb = #{ Context c -> System.out.println("pippo") };

Runnable r = #{ System.out.println("Running") };
Runnable r = (Runnable) #{ System.out.println("Running") };
executor.submit( #{ System.out.println("Running") } );
```

Lambda Bodies

Rule #2: A list of statements just like in a method body, except no break or continue at the top level. The return type is inferred from the unification of the returns from the set of return statements

Rule #3: ‘this’ has the same value as ‘this’ immediately outside the Lambda expression

Rule #4: Lambdas can use ‘effectively final’ variables as well as final variables.

```
Collection<Student> students = ...;
```

```
Collection<Student> students = ...;

double max =                           // Lambda expressions
    students.filter(  #{   Student  s -> s.gradYear == 2010 })
                .map(#{   Student s -> s.score })
                    .max();
```

Extending Interfaces

```
public interface Set<T> extends Collection<T> {

   public int size();

  ...


     // The rest of the existing Set methods
   public extension T reduce(Reducer<T> r)
     default Collections.<T>setReducer;
}
```

Extending Interfaces

```
public interface Set<T> extends Collection<T> {

   public int size();

  ...


     // The rest of the existing Set methods
   public extension T reduce(Reducer<T> r)
     default Collections.<T>setReducer;
}
```

tells us this method extends the interface

Extending Interfaces

```
public interface Set<T> extends Collection<T> {

   public int size();

  ...


     // The rest of the existing Set methods
   public extension T reduce(Reducer<T> r)
     default Collections.<T>setReducer;
}
```

tells us this method extends the interface

Implementation to use if none exists for the implementing class

```
Collection<Student> students = ...;

double max     =        // Lambda expressions
students.filter(        #{ s -> s.gradYear == 2010 })
              .map(     #{ s -> s.score })
                .max();
```

```
Collection<Student> students = ...;

double max     =        // Lambda expressions
students.filter(        #{ s -> s.gradYear == 2010 })
              .map(     #{ s -> s.score })
               .max();

interface Collection<T> {
   int add(T t);
   int size();
   void clear();
   ...
}
```

```
Collection<Student> students = ...;

double max      =        // Lambda expressions
students.filter(         #{ s -> s.gradYear == 2010 })
             .map(       #{ s -> s.score })
              .max();
```

```
Collection<Student> students = ...;

double max     =         // Lambda expressions
students.filter(         #{ s -> s.gradYear == 2010 })
              .map(      #{ s -> s.score })
               .max();

interface Collection<T> {     // Default methods

    extension Collection<E> filter(Predicate<T> p)
        default Collections.<T>filter;

    extension <V> Collection<V> map(Extractor<T,V> e)
        default Collections.<T>map;

    extension <V> V max()
        default Collections.<V>max;

}
```

Language futures: Collection Literals

```
final List<Integer> piDigits = Collections.unmodifiableList(
                Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9 ));
```

Language futures: Collection Literals

```
final List<Integer> piDigits = Collections.unmodifiableList(
                 Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9 ));
```

```
final List<Integer> piDigits = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9];
```

Language futures: Collection Literals

```
final List<Integer> piDigits = Collections.unmodifiableList(
                  Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9 ));
```

```
final List<Integer> piDigits = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9];
```

```
final Set<Integer> primes = { 2, 7, 31, 127, 8191, 131071, 524287 };
```

Language futures: Collection Literals

```
final List<Integer> piDigits = Collections.unmodifiableList(
                 Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9 ));
```

```
final List<Integer> piDigits = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9];
```

```
final Set<Integer> primes = { 2, 7, 31, 127, 8191, 131071, 524287 };
```

```
Set<Senator> honestPoliticians = {};
```

Language futures: Collection Literals

```
final Map<Integer, String> platonicSolids;
    static {
        Map<Integer, String> solids =
                                new LinkedHashMap<Integer, String>();
        solids.put(4,  "tetrahedron");
        solids.put(6,  "cube");
        solids.put(8,  "octahedron");
        solids.put(12, "dodecahedron");
        solids.put(20, "icosahedron");
        platonicSolids = Collections.immutableMap(solids);
    }
```

ORACLE

Language futures: Collection Literals

```
final Map<Integer, String> platonicSolids;
    static {
        Map<Integer, String> solids =
                               new LinkedHashMap<Integer, String>();
        solids.put(4,  "tetrahedron");
        solids.put(6,  "cube");
        solids.put(8,  "octahedron");
        solids.put(12, "dodecahedron");
        solids.put(20, "icosahedron");
        platonicSolids = Collections.immutableMap(solids);
    }
 }
```

Language futures: Collection Literals

```
final Map<Integer, String> platonicSolids;
    static {
        Map<Integer, String> solids =
                                new LinkedHashMap<Integer, String>();
        solids.put(4,  "tetrahedron");
        solids.put(6,  "cube");
        solids.put(8,  "octahedron");
        solids.put(12, "dodecahedron");
        solids.put(20, "icosahedron");
        platonicSolids = Collections.immutableMap(solids);
    }
 }
```

```
 final Map<Integer, String> platonicSolids = {
                                                4 : "tetrahedron",
                                                6 : "cube",
                                                8 : "octahedron",
                                                12 : "dodecahedron",
                                                20 : "icosahedron"
 };
```

Parallel Programming →

Lambdas
Concurrent collections

Fork/Join Framework

java.util.concurrent

java.lang.Thread

Automatic
parellization
?

1.4 5.0 6 7 8

2002 2004 2006 2011 2012 ... 2015

Parallel programming →
Multiple JVM languages →

1.0 **1.2** **5.0**

1.1 **1.3** **1.4** **6** **7** **8** **...**

1996 1997 1998 2000 2002 2004 2006 2010 2011 2012 ... 2015

7

Project Coin (JSR 334)
InvokeDynamic (JSR 292)
Fork/Join Framework

Mid 2011

Strict Verification
XRender Pipeline
Parallel Class Loaders
JDBC 4.1
Phasers
Transfer Queues
More New I/O (JSR 203)

Swing Nimbus
Unicode 6.0
Enhanced Locales
SDP & SCTP
TLS 1.2
ECC
Swing JLayer

7

Project Coin (JSR 334)
InvokeDynamic (JSR 292)
Fork/Join Framework

Mid 2011

Strict Verification
XRender Pipeline
Parallel Class Loaders
JDBC 4.1
Phasers
Transfer Queues
More New I/O (JSR 203)

Swing Nimbus
Unicode 6.0
Enhanced Loca
SDP & SCTP
TLS 1.2
ECC
Swing JLayer

8

Late 2012

Project Jigsaw
Project Lambda (JSR 335)

Collection Literals
Bulk-Data Operations
Type Annotations (JSR 308)
Swing JDatePicker

ORACLE

ORACLE®