For Algorithms and Systems Researchers

Scala comes with a strong support for various types types of concurrent and parallel programming in its standard library.

Imperative

Event-Driven

Functional

Data-Parallel

Concurrent Data Strutures

- They allow safe read and write access by concurrent threads
- Ctrie[1] - a concurrent hash-trie-based lock-free map with highly-scalable write operations and an efficient snapshot operation

```
val phonebook = concurrent.TrieMap[String, String]()
```

T1

```
phonebook.snapshot.count {
  case (name, num) =>
    name.endsWith(“Doe”)
}
```

T2

```
phonebook.remove(“Jane Doe”)
phonebook(“John Doe”) = “123417”
```

Scala Actors[2]

- Actor model programming framework - computation proceeds through many lightweight threads called `actors`
- Different actors communicate by sending each other messages

```
val a = new Actor {
  def receive = {
    case i: Int =>
      println(“Got: “ + i)
  } }
a ! 17
```

- Suitable both shared-memory concurrent programming on a single node and for distributed programming in a cluster of nodes

Futures and Promises[3]

- A framework for asynchronous programming
- The `future` construct spawns an asynchronous computation and returns a `Future` object that eventually holds the result
- Blocking and non-blocking programming styles on one axis, and event-driven and functional programming on the other

```
val f = future {
  val s = fromFile(“a.txt”)
  s.toSeq.indexOfSlice(“keyword”)
}
f onSuccess { case idx =>
  println(“Found at: “ + idx)
}
f onFailure { case t =>
  println(“Error occured.”)
}
```

```
val f = future {
  val s = fromFile(“a.txt”)
  s.getLines.toSeq
}
val g = f map {
  _.maxBy(_.length)
}
val line = Await.result(g)
println(“Longest: “ + line)
```

- Encapsulates the error-state in a uniform manner
- `Promises` serve as single-assignment variables

Parallel Collections[4]

- Data-parallel operations framework for various data structures (`Arrays`, `Vectors`, `HashMaps`, `Ranges`, ...)
- Simple programming model in the style of the Scala collections library - ideal for expressing programs that are inherently data-parallel (linear algebra, dynamic programming, Monte Carlo simulations, raytracing, numerical computations, ...)

```
val v1 = new Array[Float](50000000)
val v2 = new Array[Float](50000000)
val range = (0 until array.length).par
val dotProduct = range.aggregate(0.0f)(_ + _) {
  (sum, i) => sum + v1(i) * v2(i)
}
```

- They handle irregular parallelism seamlessly - suitable both for problems with uniform and irregular workloads
- Extensible - users can extend their custom data structures with parallel operations easily

Flavours of Parallel Programming

Framework	Imperative	Event-Driven	Task-Parallel	Functional	Data-Parallel
Concurrent Data Structures	Yes	No	No	No	No
Scala Actors	Yes	Yes	Yes	No	No
Futures and Promises	No	Yes	Yes	Yes	No
Parallel Collections	No	No	No	Yes	Yes

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

[1] A. Prokopec et al., Concurrent Tries with Efficient Non-Blocking Snapshots
[2] P. Haller et al., Actors that Unify Threads and Events
[3] P. Haller, A. Prokopec, H. Miller et al., SIP-14 Futures and Promises
[4] A. Prokopec et al., A Generic Parallel Collection Framework
[5] ScalaMeter website: http://axel22.github.io/scalameter/
[6] ScalaSTM website: http://nbronson.github.io/scala-stm/intro.html

For Parallelism and Concurrency Researchers

Scala is a high-level general purpose language suitable for the development of concurrent algorithms, data structures and runtimes.

Scala and the JVM

Why Scala

- Functional and object-oriented with access to low-level primitives
- Suitable for the development of domain specific languages
- Strong macro support aids the generation of efficient code

Why JVM

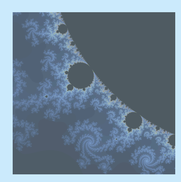
- JIT compiler - efficient execution of parallel programs
- Managed memory environment - suitable for non-blocking algorithms
- Java Memory Model - a well defined memory consistency model
- Cross-platform access to low-level threads
- Monitor-based concurrency model with low-level atomic instructions

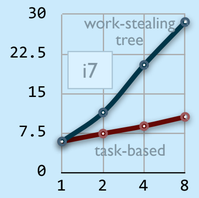
Task-Parallel and Data-Parallel Runtime

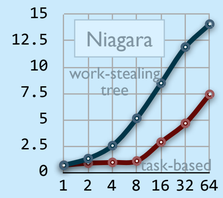
- Java Fork/Join pool in the standard Scala distribution supports task-parallel programming and is the basic building block for task-parallel runtimes
- Scala has a data-parallel runtime based on work-stealing trees designed to work well both for uniform workloads as well as irregular workloads

Example: Mandelbrot set computation

```
for (idx <- indices.par) {
  val (xc, yc) = realCoord(idx)
  image(idx) = Mandelbrot(xc, yc)
}
```

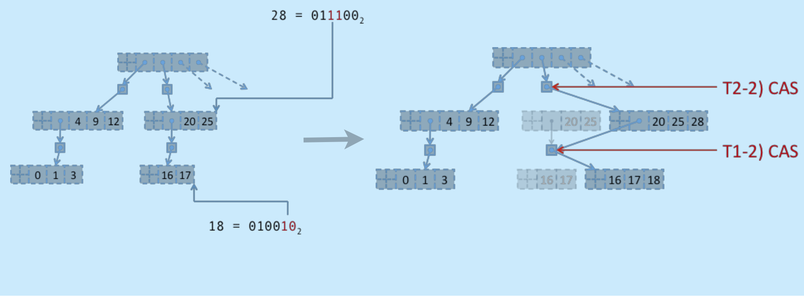






Non-Blocking Design

- Access to atomic instructions like CAS allows custom non-blocking algorithms and data structures - high-scalability and throughput



Example: Concurrent Hash Tries

Scala

Runtime Support for Concurrency

Aleksandar Prokopec, Heather Miller,
Philipp Haller, Martin Odersky

Other Tools and Frameworks

ScalaMeter[5]

A highly-configurable microbenchmarking and performance regression testing framework for the JVM platform and specifically Scala. Crucial for assesing the performance of parallel runtimes.

ScalaSTM[6]

A software transactional memory implementation for Scala designed by the STM expert group. Software transactional memory implementations expose `atomic` blocks of code that execute in isolation from the user's point of view. They generally have better scalability than alternative approaches.

ScalaSTM is available as a 3rd party library.

Tools for Distribution

Akka (Typesafe), Finagle (Twitter), Kafka (LinkedIn)

Scala in the Cloud

Heroku, Google App Engine, GridGain, CloudFoundry, ...