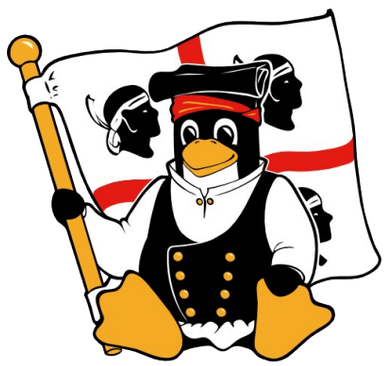
Scala e Akka
Programmazione Funzionale per architetture scalabili multicore, distribuite e su cloud

Massimiliano Dessì



Speaker

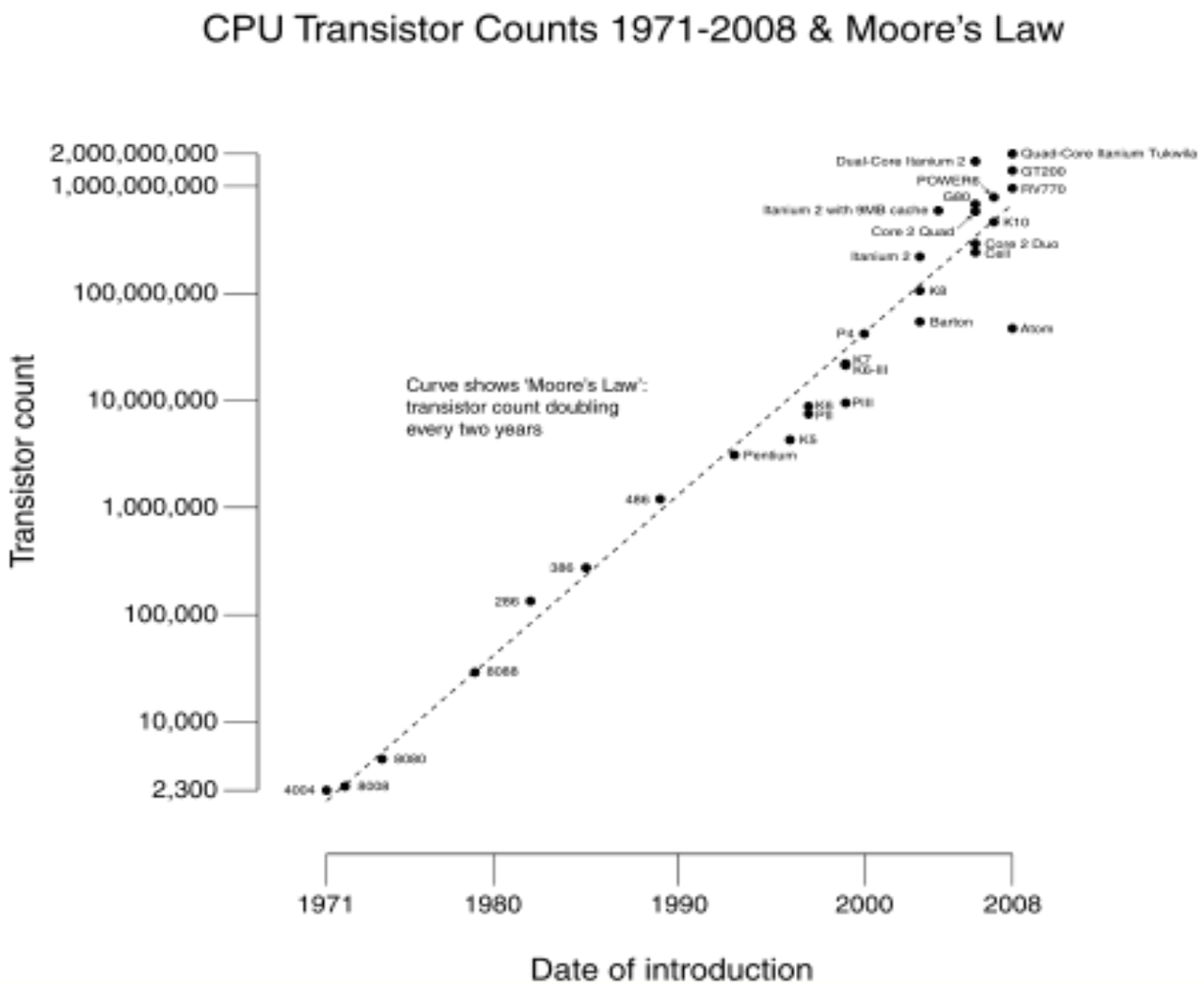
- Dad of three
- Co-fondatore e Presidente Java UG Sardegna (http://www.jugsardegna.org)
- Fondatore Google Technology UG Sardegna ((http://sardegna.gtugs.org)
- Coordinatore SpringFramework UG Italia
- Software Architect/Senior Engineer Energeya ((http://www.energeya.com)
- Autore di “Spring 2.5 Aspect Oriented Programming”

(http://www.packtpub.com/aspect-oriented-programming-with-spring-2-5/book)

Legge di Moore

Anni fa i processori eseguivano le istruzioni sequenzialmente .

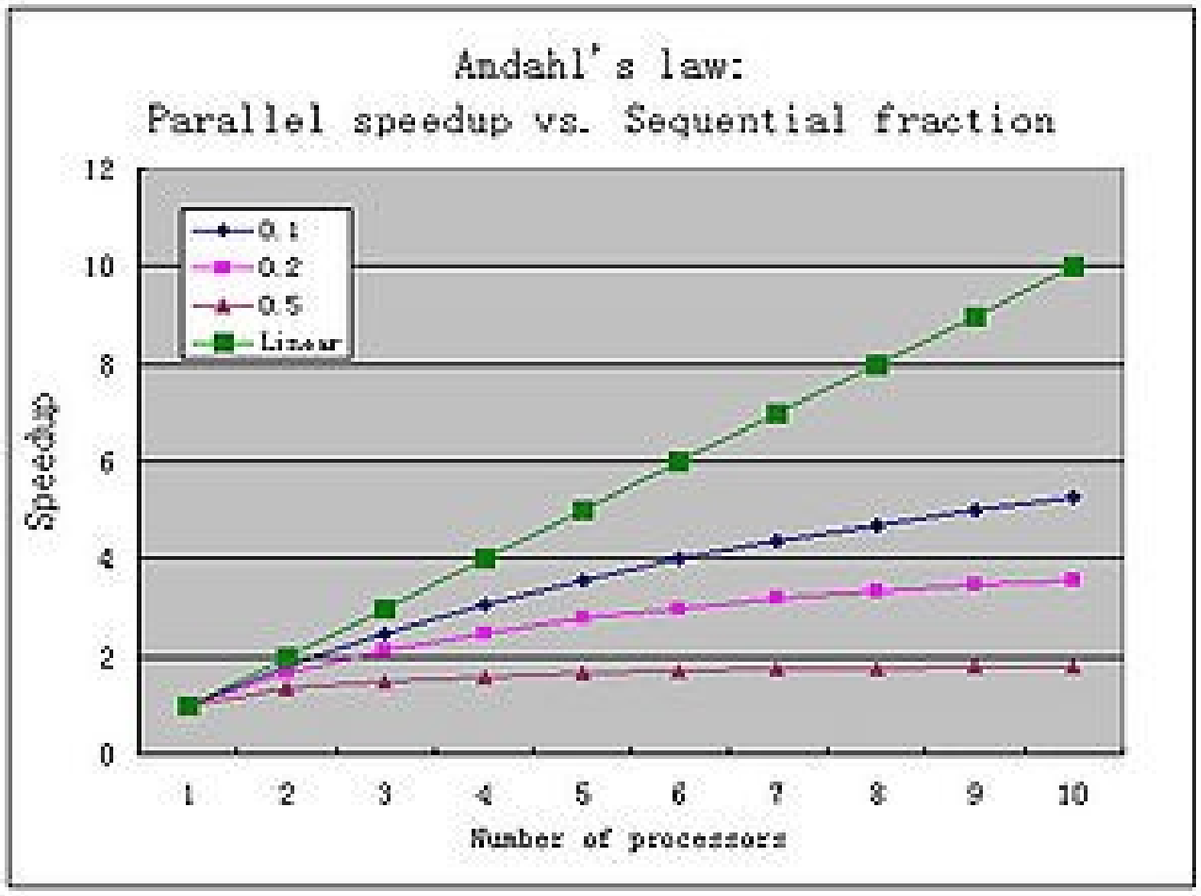
I miglioramenti della tecnologia hanno aumentato il numero di operazioni per unita di tempo aumentando i transistor.



Legge di Amdahl

Da qualche anno aumenta il numero di core

"Il miglioramento che si può ottenere su una certa parte del sistema è limitato dalla frazione di tempo in cui tale attività ha luogo", "Make the common case fast"



Multithread

La CPU simula delle esecuzioni contemporanee del software con delle pause e dei cambiamenti di contesto nei propri registri.

In alcuni sistemi come la Java Virtual Machine, vengono eseguiti algoritmi di ottimizzazione del codice compilato

eseguendo anche il riordinamento delle operazioni.

“Houston abbiamo un problema”

Per decenni sono state scritte applicazioni pensate per essere eseguite in maniera strettamente sequenziale (sequential consistency).

Oggi le applicazioni sono eseguite su processori multi core in maniera parallela e concorrente per avere migliori performance.

“Skywalker usa la forza che è in te”

Dobbiamo pensare e scrivere

il codice immaginando che

possa essere eseguito

senza vincoli di ordine,

contemporaneamente su

n-thread di esecuzione

e su core diversi

Mutable il lato oscuro della forza

Il problema nasce dalla natura mutabile dei costrutti utilizzati.

Dati contenuti in una struttura mutabile devono essere sincronizzati in lettura e scrittura affinchè due o più thread possano operare nella maniera corretta affinchè i dati siano sempre in uno stato congruente per ogni thread che deve operare su di essi,

ma sopratutto perchè i dati scritti da un thread siano visibili agli altri.

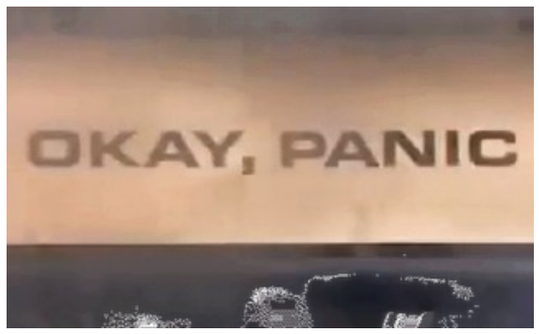
Passa la cera, togli la cera

Ogni volta che la cpu assegna uno slot di esecuzione ad un thread deve ripristinare il contesto per eseguirlo.

Se in questo slot il thread deve mettersi a sincronizzare gli accessi sui dati contenuti nei registri stiamo semplicemente sprecando cicli di clock.

No Panic.....

D'altra parte, se le parti di codice con accesso concorrente non sono ben scritte possiamo avere dei comportamenti imprevedibili oppure possiamo avere l'illusione che il codice funzioni, ma questo solo per pura fortuna.



Legge di Murphy

Semplicemente non si deve scrivere il codice pensando che alcune situazioni non avverranno.

Scommettere sulle combinazioni delle chiamate con miliardi di operazioni al secondo non si può definire meno che folle.

Legge di Murphy II

Sfortunatamente non si può
testare il codice per riprodurre
il modo in cui verrà chiamato
da più thread,
tantomeno con un ordine preciso

Java Memory Model

Le uniche certezze che abbiamo sono su

cosa viene eseguito prima di qualcos'altro,

in base alle regole happens before

del Java memory model (JSR-133)

Immutable

Oggetti immutabili possono invece essere condivisi tra più thread perchè nessuno potrà operare modifiche su di essi.

Strutture dati immutabili, cioè create nuove (in maniera opportuna e non dispendiosa) per ciascuna modifica, sono ugualmente condivisibili senza necessità di lock su di esse.

Linguaggi imperativi

Il software eseguito dalle CPU, scritto con linguaggi imperativi come C, C++, Java, C# fornisce delle astrazioni per descrive cosa deve essere eseguito.

In alcune situazioni tramite un linguaggio imperativo dobbiamo condividere delle strutture dati con altri thread di esecuzione del nostro codice.

Linguaggi imperativi

Dobbiamo quindi preoccuparci di gestire correttamente l'accesso concorrente dei diversi thread.

Dobbiamo stare attenti a dettagli di basso livello.

Possiamo fare errori molto facilmente.

Dobbiamo gestire dei data races.

Lock, Latch, Barrier, semaphores

Linguaggi imperativi

Lo shared mutable design è quello che si adotta nei linguaggi ad oggetti mainstream, Java e C#.

Un altro punti di vista

Sulla JVM possiamo scrivere ed eseguire
codice in maniera differente
dalla programmazione ad oggetti “classica”,
tutto sta nel pensare le soluzioni in maniera differente
dal design shared mutable
(condiviso e mutabile)

shared mutable design

30 alunni devono scrivere sulla lavagna ciascuno la propria età per fare la somma, usando un solo gessetto.

1 scrive col gessetto, 29 sono in coda, inattivi, in attesa del loro turno, sempre che non litighino per il gessetto.

shared mutable design

Su una risorsa contesa (il gessetto) poniamo un lock, che ci assicura che una persona alla volta possa scrivere alla lavagna.

Se i gessetti aumentano la situazione si complica perchè la lavagna e solo una, però prima capiterà che qualcuno toglierà il gesso a qualcun altro mentre stava scrivendo..

isolated mutable design

Sulla lavagna scrivo il mio numero di cellulare

30 alunni mandano un messaggio

Il gestore mi recapiterà sul telefono una coda di messaggi in ordine di arrivo e io faro la somma.

isolated mutable design

Per ricevere, possiamo usare un costrutto che
ha una mailbox, mandando
in maniera concorrenziale i messaggi
che verranno processati in maniera sequenziale dal ricevente

purely immutable design

I 30 alunni si siedono in modo da
formare una catena,
ciascuno ascolta da chi ha a sx un numero,
chi riceve aggiunge a quel numero la propria età
e lo dice a chi si trova a dx.

purely immutable design

I 30 alunni si siedono in modo da
formare una catena,
ciascuno ascolta da chi ha a sx un numero,
chi riceve aggiunge a quel numero la propria età
e lo dice a chi si trova a dx.

purely immutable design

Possiamo comporre funzioni che dato un input restituisce un output basato solamente sull' input senza nessun side effect.

Un side effect è una modifica che avviene su un altro oggetto,

a seguito di una modifica locale.

Programmazione Funzionale

Una funzione si comporta esattamente come una funzione matematica

$$y = f(x)$$

Il risultato dipende esclusivamente dai parametri di input

Functional

In un linguaggio funzionale possiamo comporre le funzioni e passarle anche come argomenti a metodi e ad altre funzioni.

Functional

Una funzione non ha side effects, non modifica cioè altro codice all'infuori della funzione stessa.

Non abbiamo stati mutabili

perciò non abbiamo nulla da controllare per l'accesso concorrente

Functional

Definendo una funzione e passandola come argomento possiamo ad esempio definire operazioni su collezioni di dati

```
val func = ceil_          //funzione
Array(3.14, 1.42, 2.0).map(func)
// Array(4.0, 2.0, 2.0)
```

Functional

Una funzione che triplica

```
def triple =(x : Double) => 3 * x
```

Functional

Somma da 1 a 10

```
val sum = fold(1 to 10, _ + _)
```

Functional

Ricorsione anziché Loop

```
def listLength1(list: List[_]): Int = {
        if (list == Nil) 0
        else 1 + listLength1(list.tail)
}
```

Functional

```
val sum = fold(1 to 10, _ + _)
```

Functional

Una funzione viene trattata come una classe perchè lo è

ad esempio la funzione A => B non è altro che

```
package scala
trait Function1[A, B] {
    def apply(x: A): B
}
```

Functional

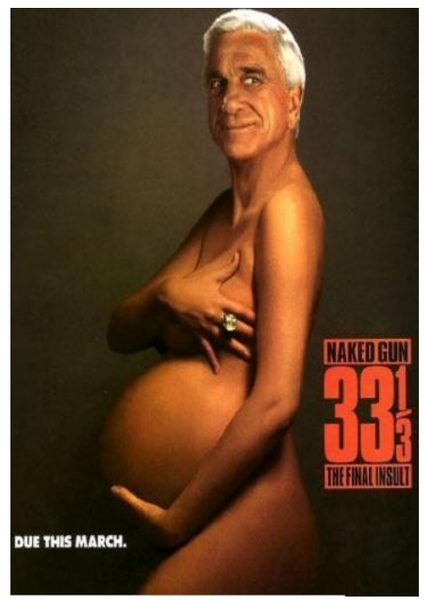
Una funzione può ricevere anche un altra funzione e diventa una High Order Function

```
def apply(f: Int => String, v: Int) = f(v)

class Decorator(left: String, right: String) {
  def layout[A](x: A) = left + x.toString() + right
}

…

val decorator = new Decorator("[", "]")
println(apply(decorator.layout, 7))
[7]
```



Scala

Scala è un linguaggio ibrido, ad oggetti e funzionale

progettato per essere scalabile utilizzando la programmazione funzionale e costrutti immutabili.

Gira sulla Java Virtual Machine

ed è pienamente interoperabile con Java

Scalable

Scala utilizza l'approccio message passing utilizzando gli actors.

Questo approccio è stato mutato da Erlang nel 2003 quando Scala è nato.

Scalable

Scala anziché lasciare che i thread collidano sui dati utilizza dei task leggeri che si scambiano in maniera asincrona messaggi immutabili tra loro.

Actor

The Actor Model, which was first proposed by Carl Hewitt in 1973 and was improved, among others, by Gul Agha .

This model takes a different approach to concurrency, which should avoid the problems caused by threading and locking

Actor

Anzichè invocare metodi su oggetti, gli
Actor si scambiano messaggi immutabili,
Ogni Actor esegue il suo task in maniera single thread
sui messaggi che ha ricevuto,
che vengono prelevati in maniera
sequenziale da un mailbox

Actor

Ogni actor è quindi sia un sender non bloccante,

sia un ricevente che può ricevere messaggi da sender concorrenti.

Gli actor sono disaccoppiati dai thread che sono una risorsa limitata.

Quando un actor ha un task da eseguire viene associato ad un thread per essere eseguito.

Akka

Akka è un toolkit che fornisce delle API Scala e Java con una propria implementazione degli Actors

per fornire scalabilità in maniera distribuita,

fault-tolerant su diversi nodi distribuiti,

su piattaforme elastic cloud.

Il modello di programmazione è ad eventi

con message passing asincrono

Simple Concurrency & Distribution

Asynchronous and Distributed by design.

High-level abstractions like Actors, Futures and STM

Resilient by Design

Write systems that self-heal.

Remote and/or local supervisor hierarchies.

Elastic & Decentralized

Adaptive load balancing, routing,

partitioning and configuration-driven remoting.

High Performance

50 million msg/sec on a single machine.

Small memory footprint;

~2.7 million actors per GB of heap.

Akka Fault tolerance

Se un actor va in crash
viene rimpiazzato da uno nuovo
che sarà rintracciabile allo stesso indirizzo,
dove i messaggi sono stati conservati in
una mailbox.

Invece in un sistema shared mutable questa funzionalità non è possibile ottenerla.

Akka Scale Out

Se aggiungiamo altri server al sistema non è un problema,
gli actors mandano messaggi verso indirizzi,
non è un problema se l' indirizzo è locale o remoto.

```
akka://mysystem@casamia.org:4040/user/mysystem1/node2
```

Qualsiasi Actor è raggiungibile tramite il path

Akka hierachy

L' `ActorSystem`, agisce come una factory per gli actor, configura i Message Dispatcher e le Mailbox, e crea un guardian.

L' Actor Guardian (`/user/`) è il parent degli actor top level.

Tutti gli actor figli del guardian possono creare degli altri actor che diventano loro figli.

In caso di crash il parent lo sostituisce con uno nuovo.

Shared state Scale Up

In un sistema shared state anche aumentando il numero dei core per aumentare il numero dei thread, abbiamo sempre il collo di bottiglia dei locks che costringono altri thread a rimanere in attesa.

In un sistema X64 con uno stack di 256kb

possono stare 4096 thread in 1Gb di ram

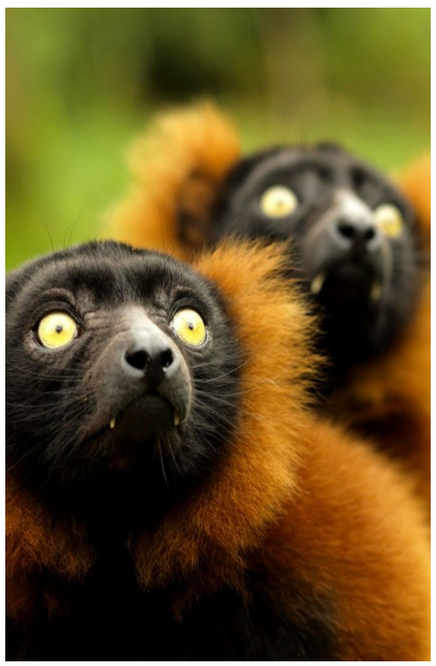
Akka Scale Up

Akka invece utilizzando un sistema message passing ha bisogno di meno thread, gli actors usano dei dispatchers configurabili che si preoccupano del tipo di thread model utilizzato e processano le mailbox.

Akka Scale Up

L'implementazione Akka degli actors è più leggera di quella standard Scala.

In 1Gb di ram possono starci 2.7 Milioni di akka actors



Actor anathomy

```
class McQueen extends Actor {
    ...
    def receive = {
        case Getaway(name, tickets)
             //TODO
        case TheGreatEscape(bike, uniform)
             //TODO write some code to sell a ticket
        case Bullit(car, gun)
             //TODO
        case_  => log.info("received unknown message")
    }
}
```

Immutable message

Una qualsiasi classe immutabile può essere un messaggio

```
case class Bike(hp: Int, cc:Int)
  val motorbike = Bike(190, 1000)
  val props = Props(new RaceTrack(100, motorbike))
  val raceTrackRef = system.actorOf(props, "raceTrack")
```

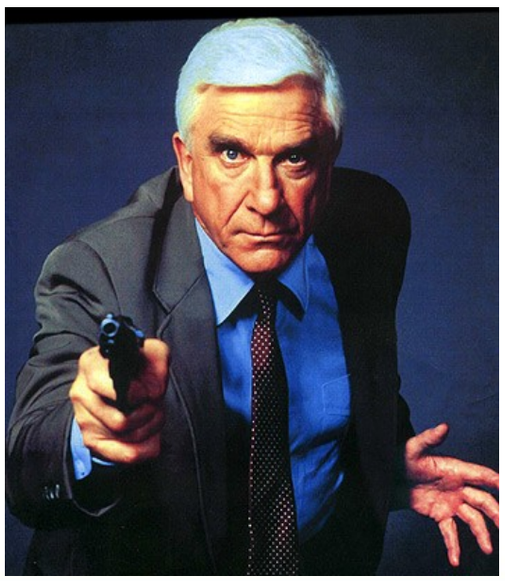
Bang !

Gli actors si inviano messaggi asincroni in questo modo:

```
actorRef ! msg
```

```
actorRef tell msg
```

Gli actors non si inviano direttamente i messaggi, ma lo fanno verso un indirizzo



Hello world actor !

```
import akka.actor.{ ActorLogging, Actor, Props, ActorSystem }

 object HelloWorld extends App {

   val system = ActorSystem("helloWorld")
   val greeter = system.actorOf(Props[Saluto],"saluto")
   greeter ! Salutando("World")
}
```

```
  case class Salutando(chi: String)

  class Saluto extends Actor with ActorLogging {
    def receive = {
       case Salutando(who) => log.info("Hello " + chi + " Actor !")
    }
  }
```

```
[helloWorld-akka.actor.default-dispatcher-3]
[akka://helloWorld/user/saluto] Hello World Actor !
```

GRAZIE PER L'ATTENZIONE

https://twitter.com/desmax74

www.scala-lang.org

http://akka.io/

http://www.javaconcurrencyinpractice.com/