PRuby : Una libreria per la programmazione parallela

Guy Tremblay

Département d'informatique
Université du Québec à Montréal
www.labunix.uqam.ca/~tremblay

Seminario presentato al Dipartimento di Informatica
Università degli Studi di Torino
4 luglio 2017

UQÀM

2017-07-04

PRuby : Una libreria per la programmazione parallela

- Oggi, voglio parlare di un lavoro che ho iniziato, due anni fa, quando ero a Torino per un anno sabbatico
- Sarà il primo seminario che farò, in parte, in italiano, quindi, dovrete essere indulgenti, e sopratutto dovrete perdonare il mio accento strano e cattivo

- Però, non avere paura : non farò tutto il seminario in italiano

2017-07-04

PRuby : Una libreria per la programmazione parallela

- Vengo dell'UQAM — Université du Québec à Montréal
- É un università «francophone» — francofona — quindi, tutto si fa in francese

UQAM è parte del'Université du Québec (UQ)

- UQAM è parte dell "*Réseau de l'Université du Québec*", una rete di università pubbliche in la provincia del Québec.
- Il Québec e una provincia di Canada, la sola dove la lingua ufficiale è il francese
- La mia lingua madre è il francese. Quindi, ho un accento strano in italiano, ma ho anche un accento strano in inglese
- E devo dire che quando io sono in Francia e parlò francese, i francesi dicono che ho un accento strano. Ma per i Québécois — Quebecese — anche i francisi hanno un accento strano

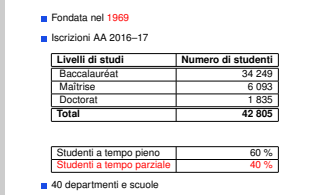
UQAM è una giovane università

- Fondata nel 1969
- Iscrizioni AA 2016–17

Livelli di studi	Numero di studenti
Baccalauréat	34 249
Maîtrise	6 093
Doctorat	1 835
Total	**42 805**

Studenti a tempo pieno	60 %
Studenti a tempo parziale	40 %

- 40 departmenti e scuole



• UQAM è in Montréal, la più grande città del Québec.
• UQAM ha 40 dipartimenti e scuole, però non ci sono medicina o ingegneria.
• Una caratteristica importante di UQAM è che molti studenti sono adulti che lavorano a tempo pieno e studiano a tempo parziale — quasi 40 percento. Quindi, molti corsi vengono insegnati in serata.

Le département d'informatique

40 Professori
Génie logiciel
Bases de données
Informatique cognitive
Bio-informatique
Réseaux et téléinformatique

Studenti	
Bacc. & Certificats	$\approx$ 700
Maîtrise & DESS	$\approx$ 100
Ph.D.	$\approx$ 50

40 Professori
Génie logiciel
Bases de données
Informatique cognitive
Bio-informatique
Réseaux et téléinformatique

Studenti	
Bacc. & Certificats	$\approx$ 700
Maîtrise & DESS	$\approx$ 100
Ph.D.	$\approx$ 50

- Il dipartmento di informatica ha 40 professori, che lavorano in diverse aree di ricerca.
- Abbiamo diversi corsi di studio : 2 baccalauréats, 4 "certificats", 3 masters, 2 DESS, 2 PhD — "informatique cognitive" e un altro, più recente (10 anni), in informatica.

2017-07-04

PRuby : Una libreria per la programmazione parallela

Durante l’inverno, fa molto freddo in Montréal e a volte ma non sempre c’è un sacco di neve, come potete vedere in queste foto.

2017-07-04

PRuby : Una libreria per la programmazione parallela

2017-07-04

PRuby : Una libreria per la programmazione parallela

• Montréal ha una carattistica strana : benché ci sia un sacco di neve, in vari luoghi, le scale sono al di fuori. Quindi, c’è molto lavoro da fare quando nevica ☹

2017-07-04

PRuby : Una libreria per la programmazione parallela

Ma non è sempre l’inverno. Ci sono anche l’estate e l’autunno. Quello che è strano : il primavera non c’è l’ha : un giorno è l’inverno e il giorno dopo, boom !, è l’estate !

2017-07-04

PRuby : Una libreria per la programmazione parallela

PRuby : Una libreria per la programmazione parallela

Guy Tremblay

Département d'informatique
Université du Québec à Montréal
www.labunix.uqam.ca/~tremblay

Seminario presentato al Dipartimento di Informatica
Università degli Studi di Torino
4 luglio 2017

UQÀM

2017-07-04

PRuby : Una libreria per la programmazione parallela

- Dunque, in questo seminario, parlero di PRuby, une libreria per la programmazione parallela . . .

Contenuto della presentazione

1 Che cosa è PRuby ?

2 Perché Ruby ?

3 Modelli di programmazione con PRuby

-

Che cosa è PRuby ?

Che cosa è PRuby ?

PRuby

- È un «*gem*» Ruby — una libreria — per la programmazione parallela

- È per introdurre gli studenti alla programmazione parallela

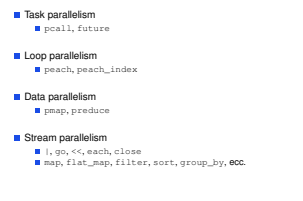
└Che cosa è PRuby ?

PRuby
- È un «*gem*» Ruby — una libreria — per la programmazione parallela
- È per introdurre gli studenti alla programmazione parallela

-

PRuby consente vari modelli di programmazione parallela

- Task parallelism
  - `pcall`, `future`
- Loop parallelism
  - `peach`, `peach_index`
- Data parallelism
  - `pmap`, `preduce`
- Stream parallelism
  - `|`, `go`, `<<`, `each`, `close`
  - `map`, `flat_map`, `filter`, `sort`, `group_by`, ecc.



- Con PRuby, si può utilizzarre quattro modeli di programmazione parallela
- Stream parallelism :
  - À la Unix, quindi, con *pipelines and implicit channels*
  - À la Go, quindi, con *explicit channels and processes*

PRuby è usato in due corsi (opzionali)

INF5171 Programmation concurrente et parallèle

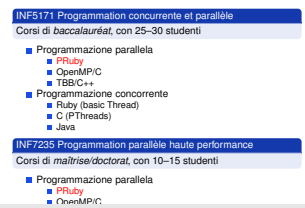
Corsi di *baccalauréat*, con 25–30 studenti

- Programmazione parallela
  - PRuby
  - OpenMP/C
  - TBB/C++
- Programmazione concorrente
  - Ruby (basic Thread)
  - C (PThreads)
  - Java

INF7235 Programmation parallèle haute performance

Corsi di *maîtrise/doctorat*, con 10–15 studenti

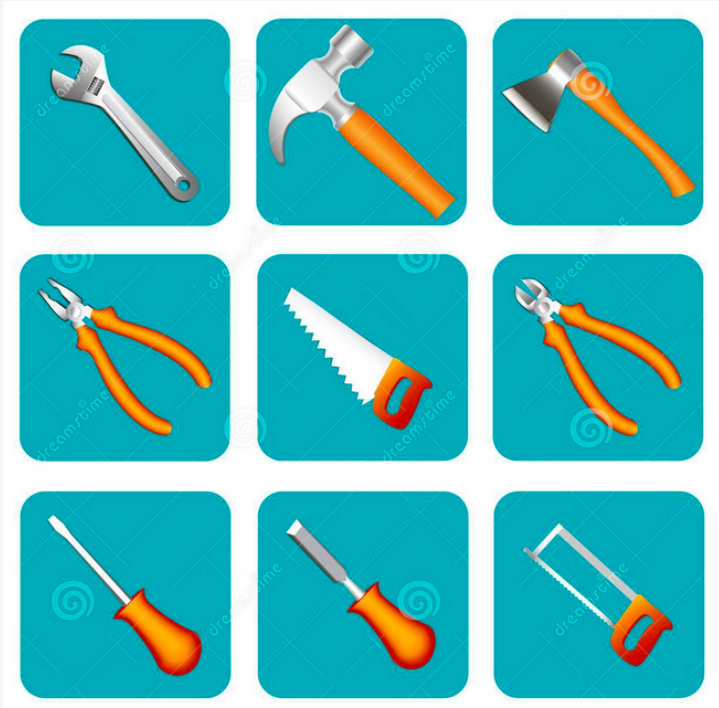
- Programmazione parallela
  - PRuby
  - OpenMP/C



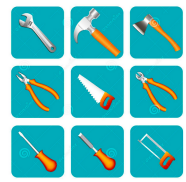
• Il primo è un corso di baccalauréat, che tratta di programmazione parallela, usando parecchi linguaggi, e anche di programmazione concorrente, cioè, gestione delle risorse con locks, monitors, condition variables, ecc.

• Il secondo è un corso di master's e PhD, che tratta solamente di programmazione parallela, anche con parecchi linguaggi e con MPI su un cluster, cioè, usando distributed memory programming with message passing

Un obiettivo fondamentale di PRuby è quello di mostrare diversi paradigmi, diversi «tools»



https://thumbs.dreamstime.com/z/
icon-set-tools-saw-hammer-screwdriver-illustration-53699981.jpg



- Dunque, voglio che gli studienti imparino nuovi tools, «nuovi modi di vedere le cose», di scrivere dei programmi

Un obiettivo fondamentale di PRuby è quello di mostrare diversi paradigmi, diversi «tools»

“Quando il martello è il solo attrezzo che si conosce, si vede chiodi ovunque” (A. Maslow)

http://www.foundationaz.com/images/blogs/if-all-you-had-was-a-hammer11.jpg

- Perché gli studenti sono abituati ad utilizzare Java e C, quindi, abituati alla programmazione sequenziale e imperativa — é quasi il solo modo di fare che conoscono

Un obiettivo fondamentale di PRuby è quello di mostrare diversi paradigmi, diversi «tools»

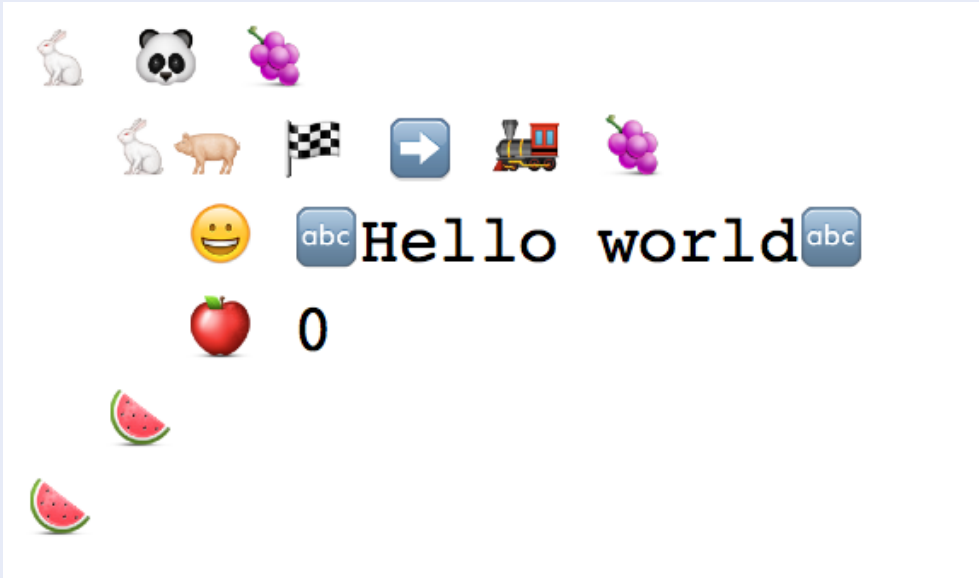
- Dunque, voglio che gli studienti imparano nuovi tools, «nuovi modi di vedere le cose», di scrivere dei programmi
- Voglio che capiscono che iln martello va bene per i chiodi però non va bene per tutti

Perché Ruby ?

In teoria, tutti i linguaggi di programmazione sono Turing-completi

Varie versioni di un programma per «Hello world !»

Emoji code



Source: `https://gist.github.com/EranAvidor/0b9a491bbe365bbf2cbe`

- Qui vi presento vari esempi di programmi per «Hello World», in vari linguaggi
- Ecco un nuovo linguaggio, con emoji

In teoria, tutti i linguaggi di programmazione sono Turing-completi

Varie versioni di un programma per «Hello world !»

Brainfuck

```
++++++++++[>+++++++>++++++++++>+++>\
+<<<<-]>++.>+.+++++++..+++.>++.<<\
+++++++++++++++.>.+++.------.--------.>+.>.
```

Source: `http://esolangs.org/wiki/Hello_world_program_in_esoteric_languages`

Brainfuck
++++++++++[>+++++++>++++++++++>+++>\
+<<<<-]>++.>+.+++++++..+++.>++.<<\
+++++++++++++++.>.+++.------.--------.>+.>.
Source: http://esolangs.org/wiki/Hello_world_program_in_esoteric_languages

- Il nome di questo linguaggio di progammazione. . . dice tutto !
- Utilizza solo 8 caratteri

In teoria, tutti i linguaggi di programmazione sono Turing-completi

Varie versioni di un programma per «Hello world !»

Anguish

Here's an Anguish program that prints `Hello World`:

Source: `http://blogs.perl.org/users/zoffix_znet/2016/05/anguish-invisible-programming-language-and-invisible-data-theft.html`

- E questo utilizza solo caratteri bianchi — invisibili !
- Anche i due prossimi programmi sono strani, con caratteri ordinari ma con molte «strane» parole. . . che danno programmi molto «verbosi»

In teoria, tutti i linguaggi di programmazione sono Turing-completi

Varie versioni di un programma per «Hello world !»

Verbose

```
PUT THE NUMBER LXXII ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER CI ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER CVIII ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER CXI ONTO THE TOP OF THE PROGRAM STACK
...
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER CVIII ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER C ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER XXXIII ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
```

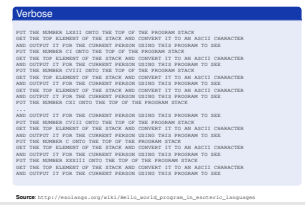
Source: `http://esolangs.org/wiki/Hello_world_program_in_esoteric_languages`

2017-07-04

PRuby : Una libreria per la programmazione parallela

└Perché Ruby ?

└In teoria, tutti i linguaggi di programmazione sono Turing-completi



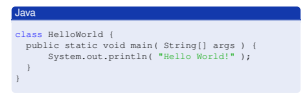
- E il prossimo anche è molto verboso, però un po' meno. Si chiama. . .

In teoria, tutti i linguaggi di programmazione sono Turing-completi

Varie versioni di un programma per «Hello world !»

Java

```
class HelloWorld {
  public static void main( String[] args ) {
      System.out.println( "Hello World!" );
  }
}
```

- Il prossimo programma è quello che mi piace di più, perché è «*short and sweet*» — breve e conciso

In teoria, tutti i linguaggi di programmazione sono Turing-completi

Varie versioni di un programma per «Hello world !»

Ruby

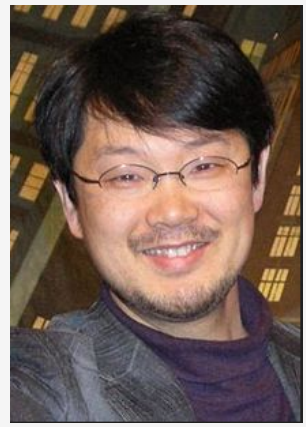
```
puts "Hello World!"
```


-

Scrivere programmi in Ruby è come scrivere pseudocode

Sometimes people jot down pseudo-code on paper. If that pseudo-code runs directly on their computers, it's best, isn't it ? Ruby tries to be like that, **like pseudo-code that runs**. *Python people say that too.*

Yukihiro Matsumoto



Source: `http://www.artima.com/intv/ruby.html`

Sometimes people jot down pseudo-code on paper. If that pseudo-code runs directly on their computers, it's best, isn't it ? Ruby tries to be like that, **like pseudo-code that runs**. *Python people say that too.*
Yukihiro Matsumoto

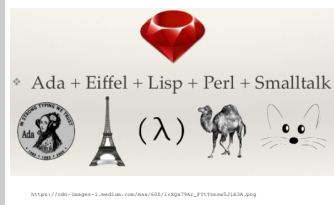
Source: `http://www.artima.com/intv/ruby.html`

- Ruby è come pseudo-code, ma che può essere eseguito
- Questo è vero anche per altri linguaggi, spesso — ma non solo — «dynamic languages»

Con Ruby, si possono usare vari paradigmi di programmazione

Gli antenati di Ruby

https://cdn-images-1.medium.com/max/600/1*XQx79Ar_FItTnnaw5JiE3A.png

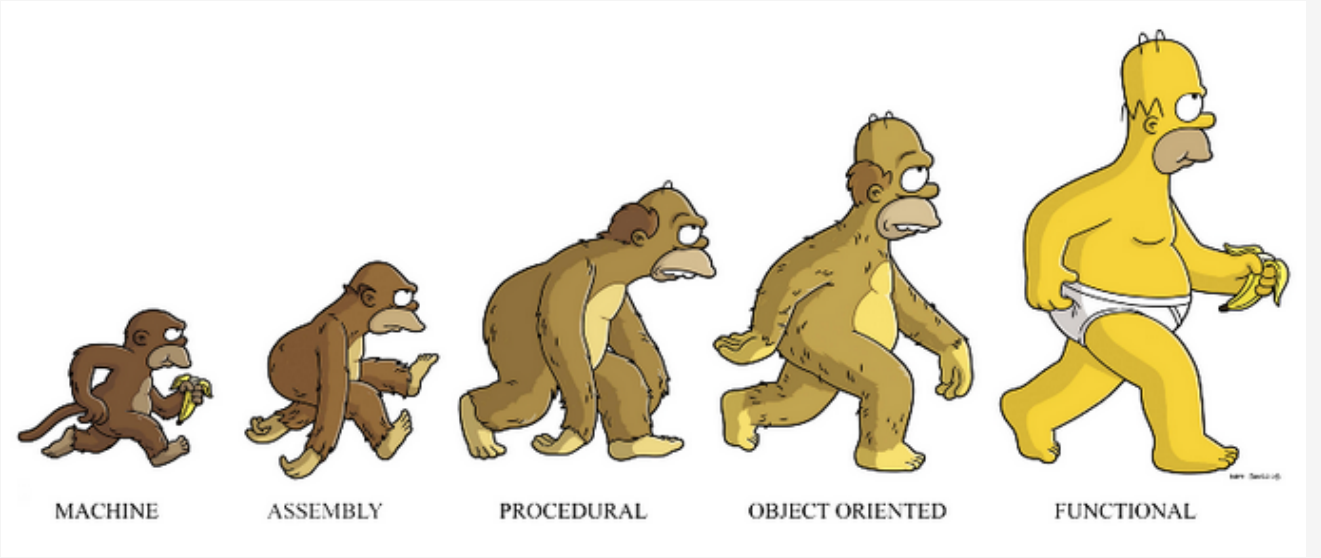


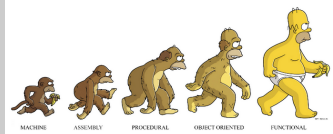
-

Linguaggio	Anno	Caratteristiche
Lisp	1958	Programmmazione funzionale Metaprogrammazione
CLU	1974	Iterators
Smalltalk	1980	Oggetti dovunque
Eiffel	1986	*Uniform Access Principle*
Perl	1987	*Regular expressions* & *pattern matching*
Ruby	1993	

• È così perché Ruby ha incorporato molte caratteristiche di parecchi antenati

Con Ruby, si possono usare vari paradigmi di programmazione





- Ruby è «*a pure object-object oriented language*» — *objects are everywhere*, tutto è un oggetto
- Però con Ruby si può anche utilizarre la programmazione funzionale

Perché Ruby ?

- INF5171 e INF7235 non sono preliminari ad altri corsi

⇒ alcun linguaggio di programmazione può essere usato

└Perché Ruby ?
- INF5171 e INF7235 non sono preliminari ad altri corsi
⇒ alcun linguaggio di programmazione può essere usato
I Ruby

• Perché Ruby mi piace e posso scegliere io i languaggi utilizzati nei corsi di programmazione parallela — quelli corsi non sono requisiti indispensabile ad alcuni altri corsi

• E perché non mi piace molto Python ☹ Ma questa è un'altra storia

Alcune caratteristiche di Ruby

-

General characteristics of Ruby

General characteristics of Ruby

-

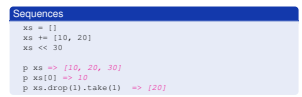
Alcune caratteristiche di Ruby

-

Ruby ha strutture dati ad alto livello

Sequences

```
xs = []
xs += [10, 20]
xs << 30

p xs => [10, 20, 30]
p xs[0] => 10
p xs.drop(1).take(1)  => [20]
```

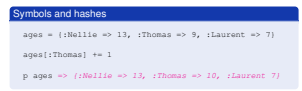



- Vi presento brevemente alcuni elementi di Ruby che saranno utili per capire PRuby
- Hashes : che vengono chiamati “dictionaries” o “maps” in altri linguaggi.
- I «Symbols», che vengono da Lisp. Sono degli «unique and non-mutable identifiers». Sono spesso utilizzati per le chiavi dei Hashes
- Il metodo `p` è per `print`

Ruby ha strutture dati ad alto livello

Symbols and hashes

```
ages = {:Nellie => 13, :Thomas => 9, :Laurent => 7}

ages[:Thomas] += 1

p ages => {:Nellie => 13, :Thomas => 10, :Laurent 7}
```

- Vi presento brevemente alcuni elementi di Ruby che saranno utili per capire PRuby
- Hashes : che vengono chiamati “dictionaries” o “maps” in altri linguaggi.
- I «Symbols», che vengono da Lisp. Sono degli «unique and non-mutable identifiers». Sono spesso utilizzati per le chiavi dei Hashes
- Il metodo `p` è per `print`

Ruby permette lo stile funzionale e lo stile imperativo

Stile funzionale

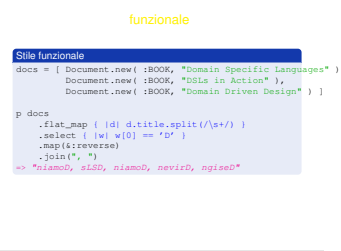
```
docs = [ Document.new( :BOOK, "Domain Specific Languages" ),
         Document.new( :BOOK, "DSLs in Action" ),
         Document.new( :BOOK, "Domain Driven Design" ) ]

p docs
    .flat_map { |d| d.title.split(/\s+/) }
    .select { |w| w[0] == 'D' }
    .map(&:reverse)
    .join(", ")
=> "niamoD, sLSD, niamoD, nevirD, ngiseD"
```


- In questo esempio, `flat_map`, `select`, `map` e `join` sono tutte funzioni pure, simile alle funzioni in Haskell, però senza “lazy evaluation”.
- Questo esempio seleziona le parole che iniziano con la lettera "D", poi le inverte e infine fa il join di queste in una singola stringa.

Ruby permette lo stile funzionale e lo stile imperativo

Stile funzionale

```
docs = [ Document.new( :BOOK, "Domain Specific Languages" ),
         Document.new( :BOOK, "DSLs in Action" ),
         Document.new( :BOOK, "Domain Driven Design" ) ]

p docs
    .flat_map { |d| d.title.split(/\s+/) }
    .select { |w| w[0] == 'D' }
    .map(&:reverse)
    .join(", ")
=> "niamoD, sLSD, niamoD, nevirD, ngiseD"
```

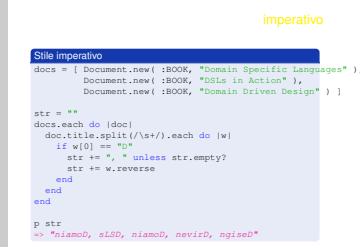



- Questo esempio dimostra anche l'utilizzo di blocks
- Il block è la «*killer feature*» di Ruby. I blocks sono simili alle lambda-expressions, e sono utilizzati quasi ovunque, per le strutture di controllo. Vedremo un esempio presto.

Ruby permette lo stile funzionale e lo stile imperativo

Stile imperativo

```ruby
docs = [ Document.new( :BOOK, "Domain Specific Languages" ),
         Document.new( :BOOK, "DSLs in Action" ),
         Document.new( :BOOK, "Domain Driven Design" ) ]

str = ""
docs.each do |doc|
  doc.title.split(/\s+/).each do |w|
    if w[0] == "D"
      str += ", " unless str.empty?
      str += w.reverse
    end
  end
end

p str
=> "niamoD, sLSD, niamoD, nevirD, ngiseD"
```

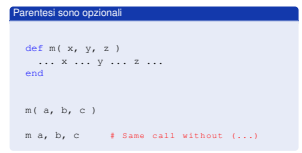



- Questo è lo stesso esempio, però con lo stile imperativo, con le strutture di controllo convenzionali.

Ruby ha una sintassi flessibile

Parentesi sono opzionali

```
def m( x, y, z )
  ... x ... y ... z ...
end


m( a, b, c )

m a, b, c      # Same call without (...)
```

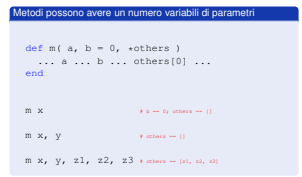



- Sintassi flessibile perché il parser è indulgente e tollerante. Per esempio, i punto e virgola o le parentesi sono opzionali,

Ruby ha una sintassi flessibile

Metodi possono avere un numero variabili di parametri

```ruby
def m( a, b = 0, *others )
  ... a ... b ... others[0] ...
end


m x                          # b == 0; others == []

m x, y                       # others == []

m x, y, z1, z2, z3           # others == [z1, z2, z3]
```

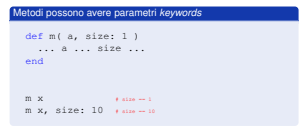



-

Ruby ha una sintassi flessibile

Metodi possono avere parametri *keywords*

```ruby
def m( a, size: 1 )
  ... a ... size ...
end


m x                 # size == 1
m x, size: 10       # size == 10
```

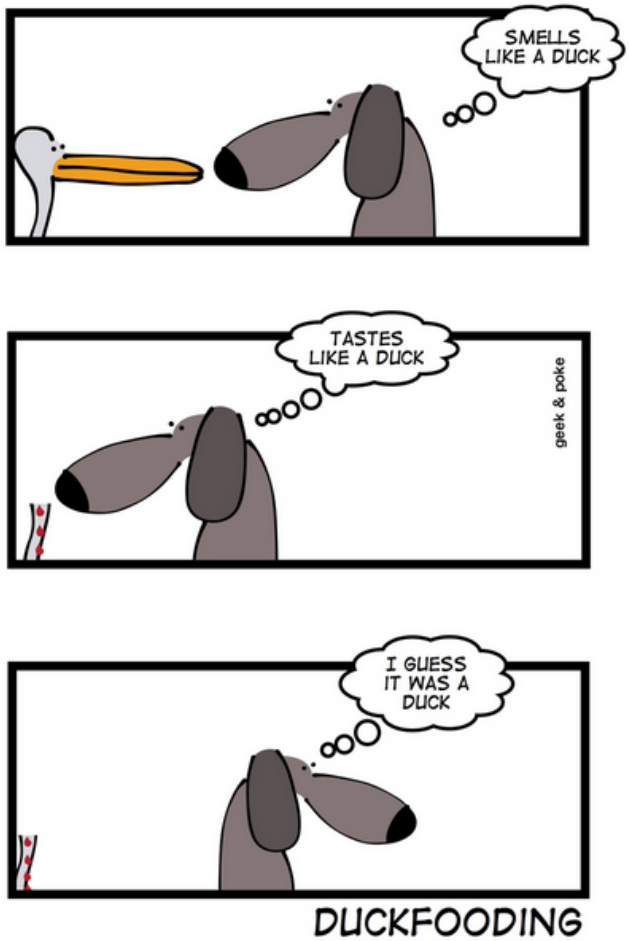
- Questo è simile ai «keyword parameters» di Smalltalk.

Ruby utilizza polimorfismo “*à la duck typing*”

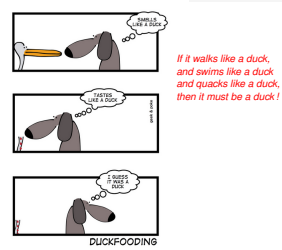
“Duck typing”

Ciò che conta è i messaggi a cui un oggetto può rispondere

- Duck typing è une specie di dynamic typing



If it walks like a duck, and swims like a duck and quacks like a duck, then it must be a duck !



-

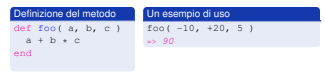
Ruby utilizza polimorfismo “*à la duck typing*”

Definizione del metodo

```
def foo( a, b, c )
  a + b * c
end
```

Un esempio di uso

```
foo( -10, +20, 5 )
=> 90
```

- Questo è un esempio semplice, con un metodo `foo` che calcola «a più (b volte c)»
- Quando gli argomenti sono dei numeri, il risultato è un numero

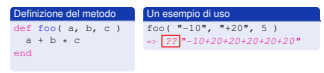
Ruby utilizza polimorfismo “*à la duck typing*”

Definizione del metodo

```
def foo( a, b, c )
  a + b * c
end
```

Un esempio di uso

```
foo( "-10", "+20", 5 )
=> ??"-10+20+20+20+20+20"
```

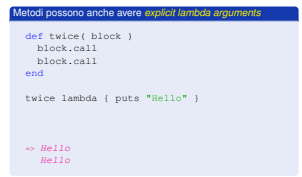



- Però, quando gli argomenti sono due strings e un numero, il risultato è un string, perché il senso di più, e anche di volte, è diverso
- Reformuler ! !
- Ruby è un «object-oriented language», dunque il senso di un metodo dipende dal **destinatario** del messaggio

Ruby facilita la programmazione di nuove strutture di controllo

Metodi possono anche avere *explicit lambda arguments*

```
def twice( block )
  block.call
  block.call
end

twice lambda { puts "Hello" }



=> Hello
   Hello
```

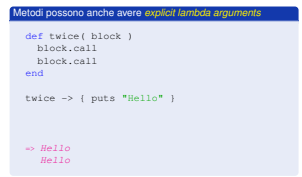



- Un block e una lambda expression sono simili
- Un block è valutato con `yield`
- Una lambda expression è valutata con `call`

Ruby facilita la programmazione di nuove strutture di controllo

Metodi possono anche avere *explicit lambda arguments*

```
def twice( block )
  block.call
  block.call
end

twice -> { puts "Hello" }



=> Hello
   Hello
```

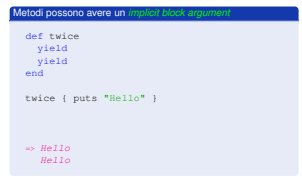



- Si può usare anche il simbolo “->” per una lambda expression, come si fa in molti altri linguaggi.

Ruby facilita la programmazione di nuove strutture di controllo

Metodi possono avere un *implicit block argument*

```
def twice
  yield
  yield
end

twice { puts "Hello" }



=> Hello
   Hello
```

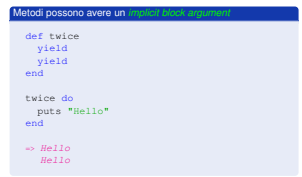



- Con i blocks e la istruzione `yield`, è possibile definire delle nuove strutture di controllo.
- In realtà, in Ruby, ci sono poche vere strutture di controllo. Però, ci sono molti metodi nella libreria, nel modulo `Enumerable`, metodi che sono disponibili quando un metodo `each` è definito, usando i mixin.
- Reformuler

Ruby facilita la programmazione di nuove strutture di controllo

Metodi possono avere un *implicit block argument*

```
def twice
  yield
  yield
end

twice do
  puts "Hello"
end

=> Hello
   Hello
```

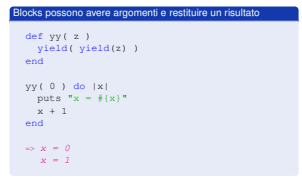



- L'instruzione `yield` viene nel linguaggio CLU, nel 1974, quaranta anni fa, ma con una sintassi poco differente.
- In Ruby, si può indicare un block in due modi : con brackets o con do/end.
- Si utilizzano i brackets quando il block è su una sola linea. Si utilizza do/end quando il block è su parecchie linee

Ruby facilita la programmazione di nuove strutture di controllo

Blocks possono avere argomenti e restituire un risultato

```ruby
def yy( z )
  yield( yield(z) )
end

yy( 0 ) do |x|
  puts "x = #{x}"
  x + 1
end

=> x = 0
   x = 1
```

- In quest'esempio, il risultato del primo yield è utilizzato come un argomento del secondo yield. Questo è possibile perché il block restituisce un risultato, qui è l'espressione x più uno

Quale implementazione di Ruby si deve utilizzare per PRuby ?

Quale implementazione di Ruby si deve utilizzare per PRuby ?

-

Esistono parecchie *implementations* di Ruby

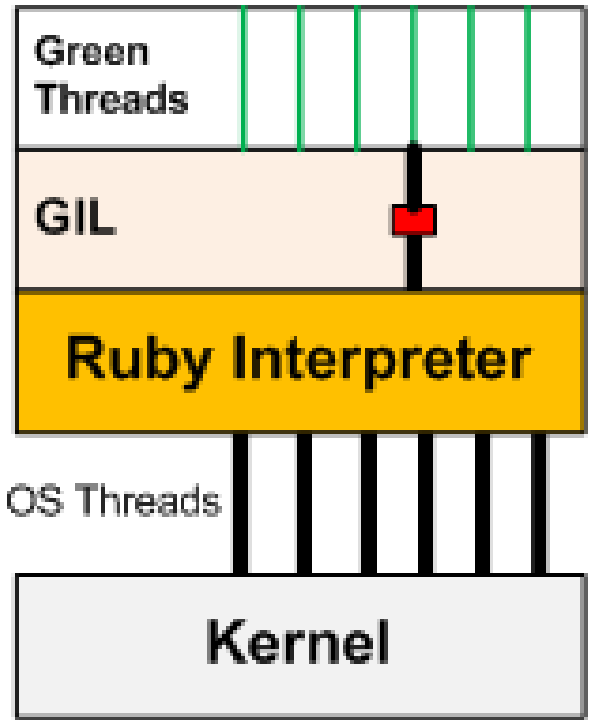
```
$ rvm list known
# MRI Rubies
# JRuby
# Rubinius
# Opal
# Minimalistic ruby implementation - ISO 30170:2012
# Ruby Enterprise Edition
# GoRuby
# Topaz
# MagLev
# Mac OS X Snow Leopard Or Newer
# IronRuby
```

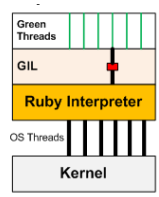

$ rvm list known
# MRI Rubies
# JRuby
# Rubinius
# Opal
# Minimalistic ruby implementation - ISO 30170:2012
# Ruby Enterprise Edition
# GoRuby
# Topaz
# MagLev
# Mac OS X Snow Leopard Or Newer
# IronRuby

- I due principali sono MRI Ruby e JRuby

MRI Ruby — C Implementation

https://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby/

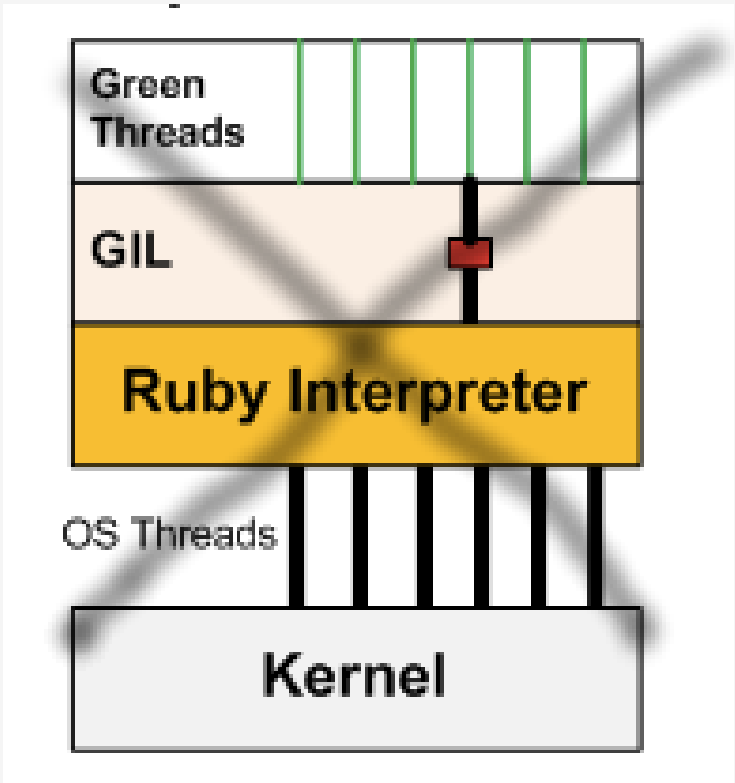


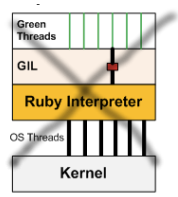


- MRI Ruby ha un GIL — *global interpreter lock*. Quindi, non ci sono *«real» lightweight threads*

MRI Ruby — C Implementation

https://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby/

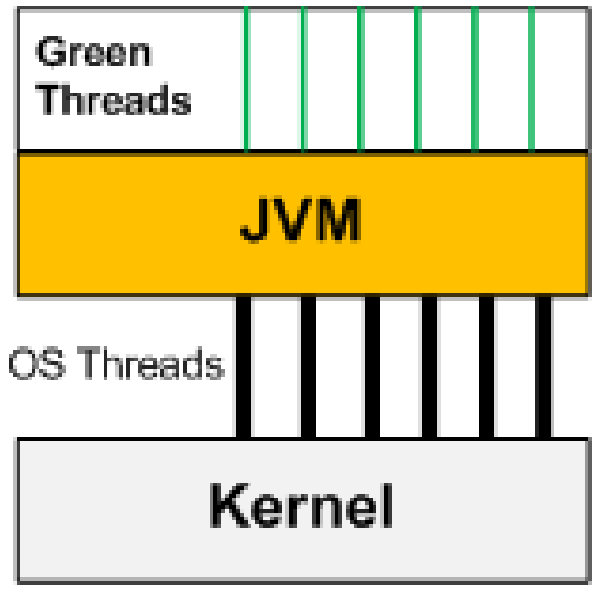


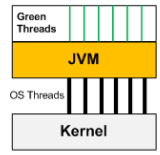


-

JRuby — Java/JVM Implementation

https://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby/



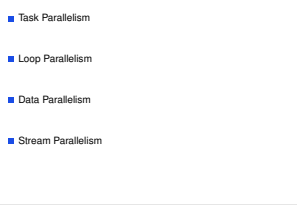


- Invece, JRuby genera *bytecode* per il JVM, quindi si possono utilizzare i *lightweight threads*

Modelli di programmazione con PRuby

I quattro modelli di programmazione parallela che si possono sfruttare con PRuby

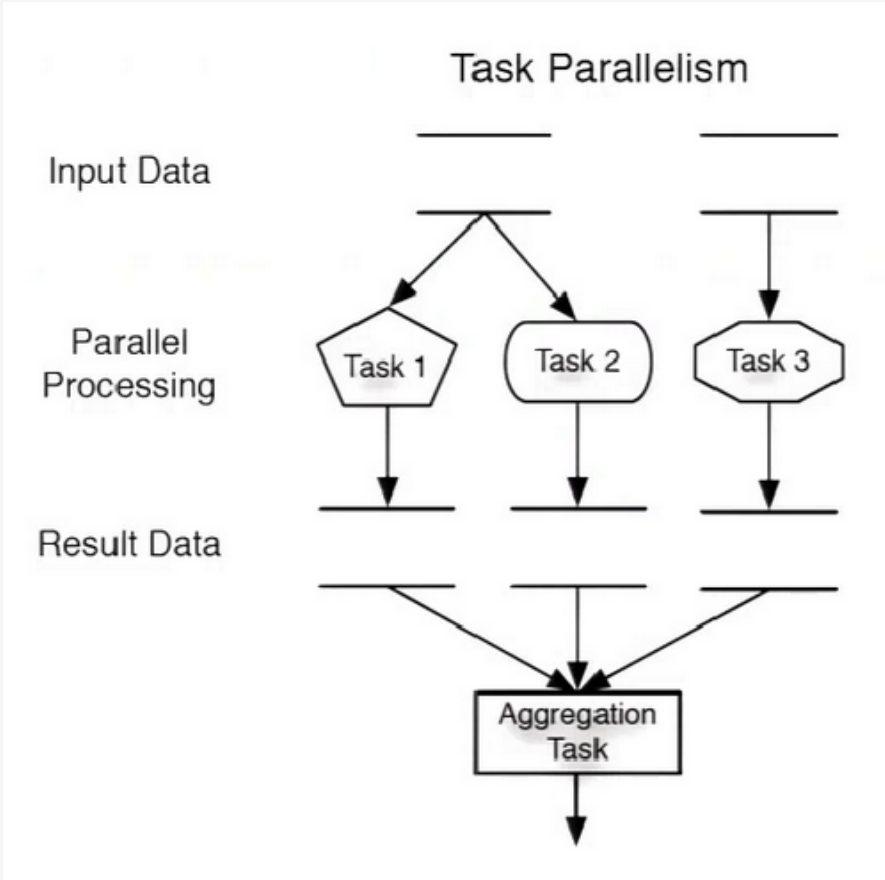
- Task Parallelism
- Loop Parallelism
- Data Parallelism
- Stream Parallelism



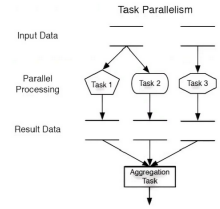
• Pruby permette quattro modelli principali di programmazione parallela, con vari metodi

Task Parallelism

-



https://www.tele-task.de/media/hpi/slides/7441/pass1/3075.jpg



-

Task Parallelism

Task parallelism ([aka.] function parallelism and control parallelism) [. . .] focuses on distributing tasks—concurrently performed by processes or threads—across different processors.

`https://en.wikipedia.org/wiki/Task_parallelism`

- Un utilizzo tipico = *Recursive divide-and-conquer parallelism*

Task parallelism ([aka.] function parallelism and control parallelism) [. . .] focuses on distributing tasks—concurrently performed by processes or threads—across different processors.
`https://en.wikipedia.org/wiki/Task_parallelism`
- Un utilizzo tipico = *Recursive divide-and-conquer parallelism*

• Dunque, *task parallelism* — parallelismo su attività — si utilizza quando ci sono diverse attività — *function instances* — che si possono fare in parallelo

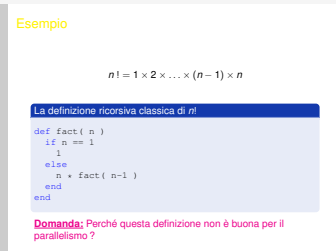
Esempio: Calcolare $n!$

$$n! = 1 \times 2 \times \ldots \times (n-1) \times n$$

La definizione ricorsiva classica di $n!$

```
def fact( n )
  if n == 1
    1
  else
    n * fact( n-1 )
  end
end
```

Domanda: Perché questa definizione non è buona per il parallelismo ?

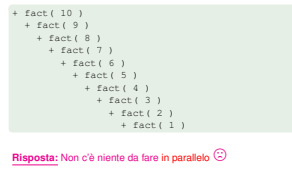


- Guardiamo un piccolo esempio di parallelismo ricorsivo, il calcolo di n fattoriale, per n (greater or equal than?) maggiore o uguale a uno
- Domanda. . .

L'albero di ricorsione per calcolare `fact(10)`... è lineare ☹

```
+ fact( 10 )
  + fact( 9 )
    + fact( 8 )
      + fact( 7 )
        + fact( 6 )
          + fact( 5 )
            + fact( 4 )
              + fact( 3 )
                + fact( 2 )
                  + fact( 1 )
```

Risposta: Non c'è niente da fare in parallelo ☹

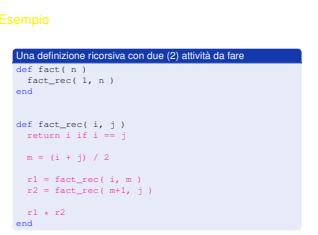


- **Risposta:** Perché ogni *function call* fa una sola altra *function call*
- Perché c'è una sola altra attività da fare, l'albero di ricorsione è lineare, quindi, il parallelismo non c'è

Esempio: Calcolare *n*!

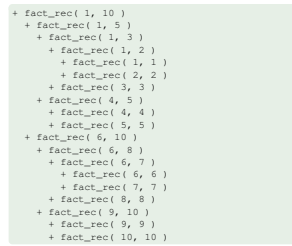
Una definizione ricorsiva con due (2) attività da fare

```
def fact( n )
  fact_rec( 1, n )
end


def fact_rec( i, j )
  return i if i == j

  m = (i + j) / 2

  r1 = fact_rec( i, m )
  r2 = fact_rec( m+1, j )

  r1 * r2
end
```

- Per il parallelismo, abbiamo bisogno di almeno due attività da fare
- Ecco una definizione reursiva con due chiamate ricorsivi
- Dobbiamo definire una funzione ausiliare, con due argomenti che definiscono il problema da trattare, cioè, l'intervallo per il quale dobbiamo calcolare *i* volte *i* più uno volte *i* più due . . . volte *j*
- Quando il problema è triviale, con i uguale j, ritorniamo subito i
- Quando il problema è complesso, dividiamo il problema in due problemi più semplici, trattiamo questi due problemi e poi combiniamo i due risultati

```
+ fact_rec( 1, 10 )
  + fact_rec( 1, 5 )
    + fact_rec( 1, 3 )
      + fact_rec( 1, 2 )
        + fact_rec( 1, 1 )
        + fact_rec( 2, 2 )
      + fact_rec( 3, 3 )
    + fact_rec( 4, 5 )
      + fact_rec( 4, 4 )
      + fact_rec( 5, 5 )
  + fact_rec( 6, 10 )
    + fact_rec( 6, 8 )
      + fact_rec( 6, 7 )
        + fact_rec( 6, 6 )
        + fact_rec( 7, 7 )
      + fact_rec( 8, 8 )
    + fact_rec( 9, 10 )
      + fact_rec( 9, 9 )
      + fact_rec( 10, 10 )
```

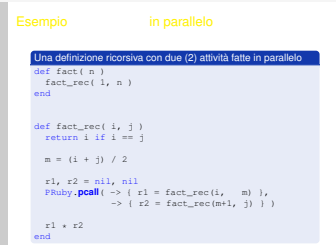



- Ecco l'albero di ricorsione di `fact(10)`, in cui sono indicate i *function calls* di `fact_rec`
- Si vedono i vari sotto-problemi che vengono creati
- Adesso, con quella definizione, è facile vedere come si possono creare attività in parallelo : le *function calls* sugli stessi livelli possono essere valutate in parallelo

Esempio: Calcolare *n*! in parallelo

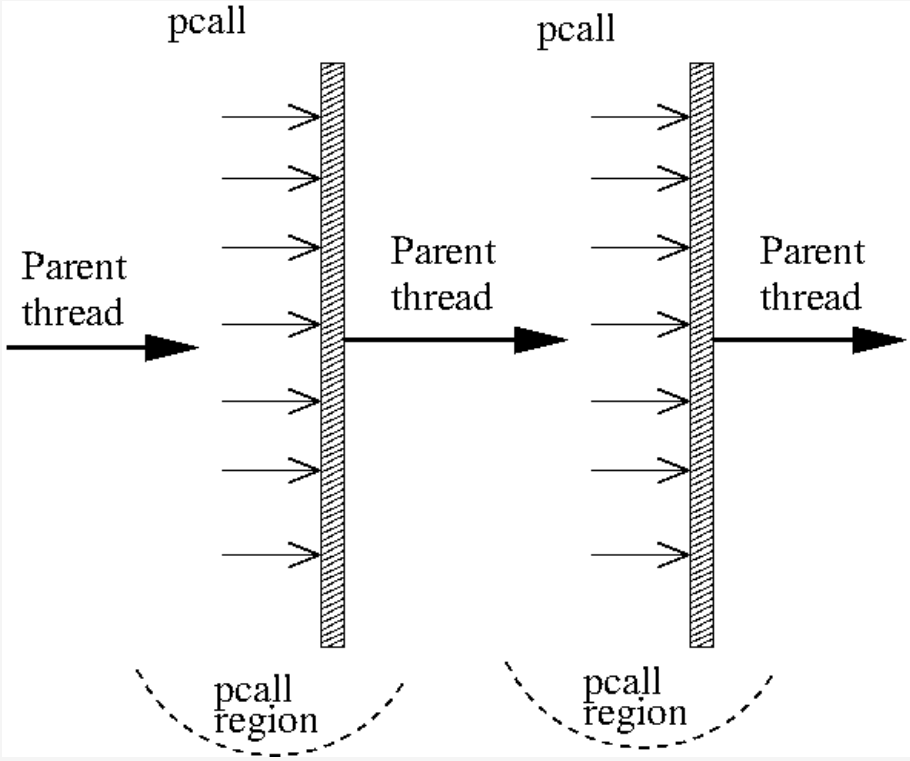
Una definizione ricorsiva con due (2) attività fatte in parallelo

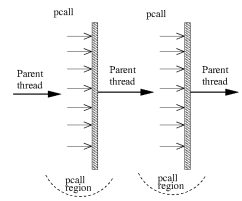
```
def fact( n )
  fact_rec( 1, n )
end


def fact_rec( i, j )
  return i if i == j

  m = (i + j) / 2

  r1, r2 = nil, nil
  PRuby.pcall( -> { r1 = fact_rec(i,   m) },
               -> { r2 = fact_rec(m+1, j) } )

  r1 * r2
end
```

- In PRuby, si fa con il metodo `pcall` — *parallel call*.
- Note to me/Claudia : Contrary to what I first wrote, i decided not to use the term 'istruzione' because this is not the term used in Ruby. Instead, I now use the term 'metodo' (method), which is maschile, so to refer to PRuby methods I will use the maschile.
- Great !
- Gli argomenti di `pcall` devono essere delle *lambda-espressions*. Qui, ci sono due lambda
- Un *block* è una *closure*, quindi, un *block* cattura i *non-local variables*. Qui, il primo *block* cattura r1 e il secondo cattura r2. Queste variabili sono usate per ritornare il risultato delle *recursive calls*, utilizzando *side-effects*.

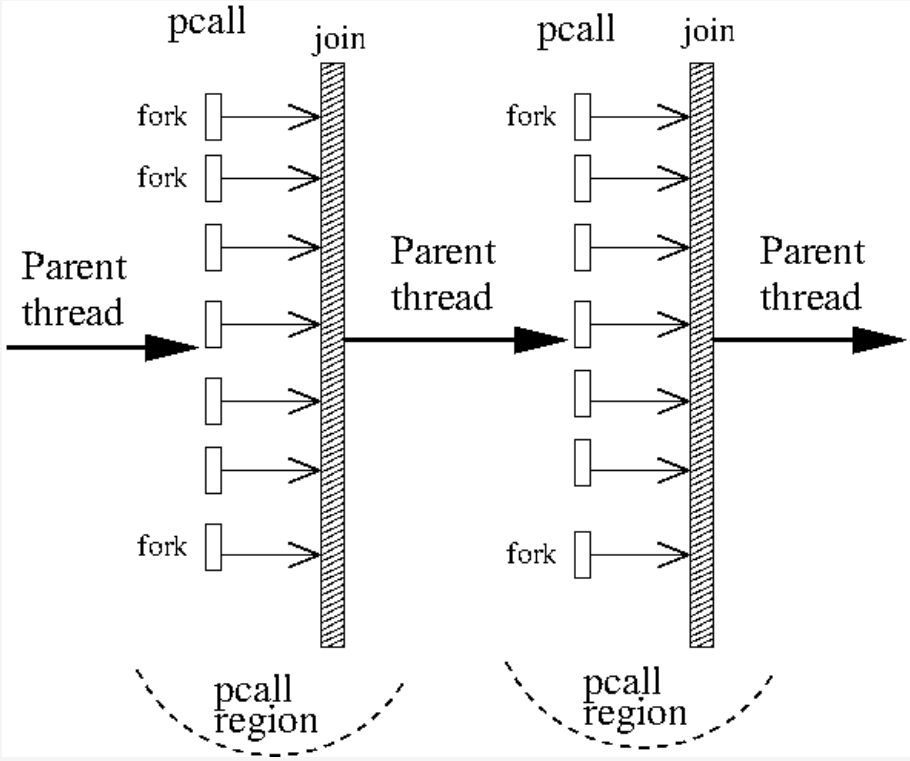
Il metodo `pcall` è una specie di `cobegin/coend`

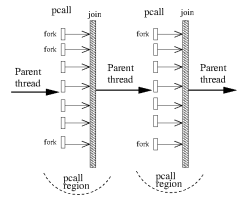




- All' inizio di `pcall`, un *thread* nuovo e indipendente è creato da ogni `lambda`
- Il `pcall` si conclude quando—e solo quando—tutti i *threads* sono terminati

Il metodo `pcall` è una specie di `cobegin/coend` con *implicit fork/join*



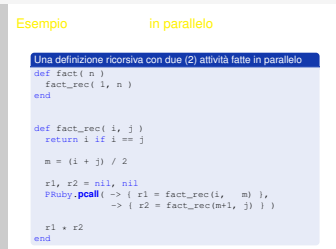


- Quindi, il `pcall` è una specie di *fork/join*, dove i join dei *threads* sono impliciti

Esempio: Calcolare *n*! in parallelo

Una definizione ricorsiva con due (2) attività fatte in parallelo

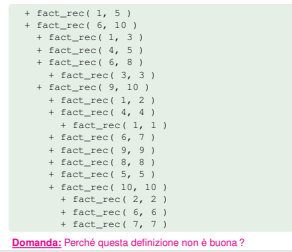
```
def fact( n )
  fact_rec( 1, n )
end


def fact_rec( i, j )
  return i if i == j

  m = (i + j) / 2

  r1, r2 = nil, nil
  PRuby.pcall( -> { r1 = fact_rec(i,   m) },
               -> { r2 = fact_rec(m+1, j) } )

  r1 * r2
end
```

-

```
+ fact_rec( 1, 5 )
+ fact_rec( 6, 10 )
  + fact_rec( 1, 3 )
  + fact_rec( 4, 5 )
  + fact_rec( 6, 8 )
    + fact_rec( 3, 3 )
  + fact_rec( 9, 10 )
    + fact_rec( 1, 2 )
    + fact_rec( 4, 4 )
      + fact_rec( 1, 1 )
    + fact_rec( 6, 7 )
    + fact_rec( 9, 9 )
    + fact_rec( 8, 8 )
    + fact_rec( 5, 5 )
    + fact_rec( 10, 10 )
      + fact_rec( 2, 2 )
      + fact_rec( 6, 6 )
      + fact_rec( 7, 7 )
```

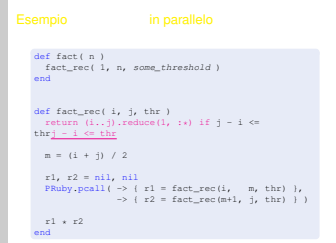
Domanda: Perché questa definizione non è buona ?



- Con questa definizione, l'albero di ricorsione è simile a quello della definizione sequenziale, forse con un ordine differente delle `print` a causa del parallelismo
- a causa del is correct !
- Risposta : Perché ogni attività è troppo piccola — non c'è abbastanza lavoro da fare, quindi l'*overhead* per creare i *threads* è troppo grande, perché creare un *thread* è più costoso di chiamare un metodo

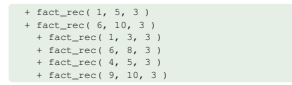
Esempio: Calcolare *n*! in parallelo con meno *threads*

```ruby
def fact( n )
  fact_rec( 1, n, some_threshold )
end


def fact_rec( i, j, thr )
  return (i..j).reduce(1, :*) if j - i <=
thrj - i <= thr

  m = (i + j) / 2

  r1, r2 = nil, nil
  PRuby.pcall( -> { r1 = fact_rec(i,   m, thr) },
               -> { r2 = fact_rec(m+1, j, thr) } )

  r1 * r2
end
```

- Dunque, si deve creare meno *threads* e ogni *thread* deve fare più lavoro.
- Ecco un modo di farlo, utilizzando una *threshold*, una soglia, che indica quando la ricorsione deve terminare
- In questa definizione, non c'è ricorsione quando il problema è «semplice» ma non triviale. Cioè, quando il numero degli elementi da trattare è più piccolo o uguale a la *threshold*

```
+ fact_rec( 1, 5, 3 )
+ fact_rec( 6, 10, 3 )
  + fact_rec( 1, 3, 3 )
  + fact_rec( 6, 8, 3 )
  + fact_rec( 4, 5, 3 )
  + fact_rec( 9, 10, 3 )
```

- Con la soglia di ricorsione, le attività diventano più grosse e ci sono meno threads
- Ecco l'albero di ricorsione di `fact(10)` con il *threshold* uguale a 3

Esempio: Calcolare *n*! in parallelo con meno *threads*

```
def fact( n )
  fact_rec( 1, n, some_threshold )
end


def fact_rec( i, j, thr )
  return (i..j).reduce(1, :*) if j - i <= thr

  m = (i + j) / 2

  r1, r2 = nil, nil
  PRuby.pcall( -> { r1 = fact_rec(i,   m, thr) },
               -> { r2 = fact_rec(m+1, j, thr) } )

  r1 * r2
end
```

Domanda: Si può fare meglio, con soluzione più semplice ?

Esempio in parallelo

```
def fact( n )
  fact_rec( 1, n, some_threshold )
end


def fact_rec( i, j, thr )
  return (i..j).reduce(1, :*) if j - i <= thr

  m = (i + j) / 2

  r1, r2 = nil, nil
  PRuby.pcall( -> { r1 = fact_rec(i,   m, thr) },
               -> { r2 = fact_rec(m+1, j, thr) } )

  r1 * r2
end
```

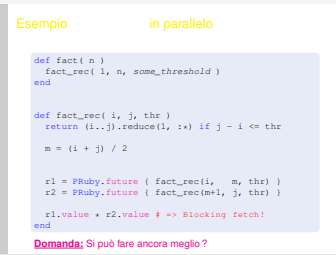
Domanda: Si può fare meglio, con soluzione più semplice ?

- C'è una soluzione più semplice ? Più elegante ?

Esempio: Calcolare *n*! in parallelo con meno *threads*

```ruby
def fact( n )
  fact_rec( 1, n, some_threshold )
end


def fact_rec( i, j, thr )
  return (i..j).reduce(1, :*) if j - i <= thr

  m = (i + j) / 2


  r1 = PRuby.future { fact_rec(i,   m, thr) }
  r2 = PRuby.future { fact_rec(m+1, j, thr) }

  r1.value * r2.value # => Blocking fetch!
end
```

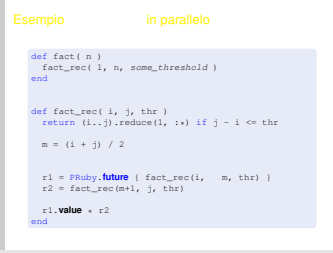
Domanda: Si può fare ancora meglio ?



- **Risposta:** Si, ma non con il `pcall`. Qui, `fact_ij` è una funzione che ritorna un valore, Quindi, si può utilizzare un altro metodo, il `future`
- Il `future` ritorna un risultato subito, però, non è un numero. È una «promessa» di dare il numero, con una call al metodo `value`, ma solamente se questo risoltato è pronto, se il suo calcolo è finito. Se non è pronto, il callo da `value` bloccarà finché il risoltato sia pronto
- Here you can say both "una call al metodo" and "una chiamata al metodo", because we can interleave english and italian in such cases

Esempio: Calcolare *n*! in parallelo con meno *threads*

```ruby
def fact( n )
  fact_rec( 1, n, some_threshold )
end


def fact_rec( i, j, thr )
  return (i..j).reduce(1, :*) if j - i <= thr

  m = (i + j) / 2


  r1 = PRuby.future { fact_rec(i,   m, thr) }
  r2 = fact_rec(m+1, j, thr)

  r1.value * r2
end
```

- **Risposta:** Si, perché il *thread* genitore non ha niente da fare, quindi, può generare un unico future e fare esso stesso una parte del lavoro

```
+ fact_rec( 1, 5, 3 )
  + fact_rec( 6, 8, 3 )
  + fact_rec( 1, 3, 3 )
```

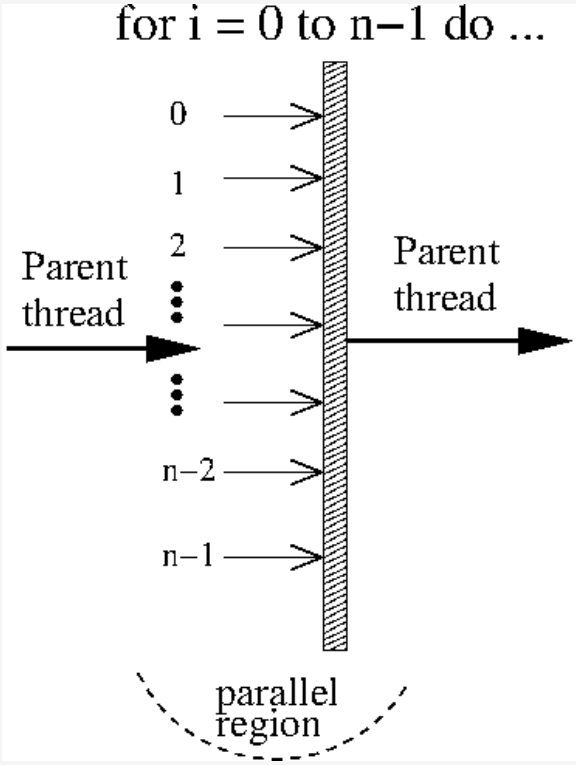

+ fact_rec(1, 5, 3)
  + fact_rec(6, 8, 3)
  + fact_rec(1, 3, 3)

- Ecco l'albero con i *threads* che sono creati
- Solo tre *threads* sono creati, quindi, l'*overhead* è molto più piccolo

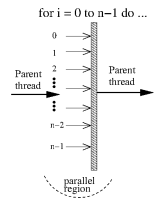
Loop Parallelism

-

Loop Parallelism



Quando gli iterazioni di un *for loop* sono indipendenti, e quindi si possono fare in parallelo



Quando gli iterazioni di un *for loop* sono indipendenti, e quindi si possono fare in parallelo

-

Esempio: Somma di due vettori

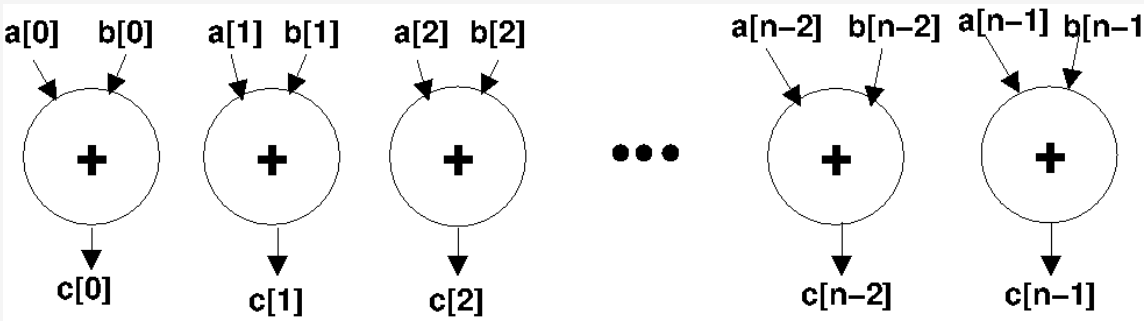
```ruby
def sum( a, b )
  c = Array.new(a.size)

  (0...c.size).each do |k|
    c[k] = a[k] + b[k]
  end

  c
end
```

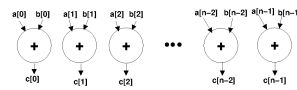

Esempio

```ruby
def sum( a, b )
  c = Array.new(a.size)

  (0...c.size).each do |k|
    c[k] = a[k] + b[k]
  end

  c
end
```

- In Ruby, si utilizza il metodo `each` per le iterazioni
- Prima, si crea l'`Array` per il risultato
- Poi, si calcola ogni elemento da `c`, utilizzando i vari `k`

Data dependencies da `a[0..n] + b[0..n]`





- Ecco un *data dependencies graph*, che indica i dati che sono utilizzati per calcolare le varie parti del risultato
- Si può vedere che tutti gli elementi sono indipendenti, quindi, si possono calcolare tutti in parallelo

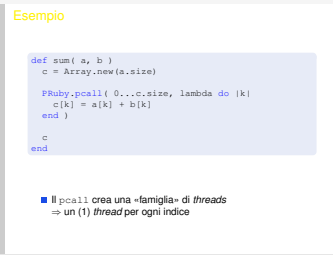
Esempio: Somma di due vettori con «*fine-grain parallelism*» : `pcall`

```
def sum( a, b )
  c = Array.new(a.size)

  PRuby.pcall( 0...c.size, lambda do |k|
    c[k] = a[k] + b[k]
  end )

  c
end
```

- Il `pcall` crea una «famiglia» di *threads*
  $\Rightarrow$ un (1) *thread* per ogni indice

2017-07-04

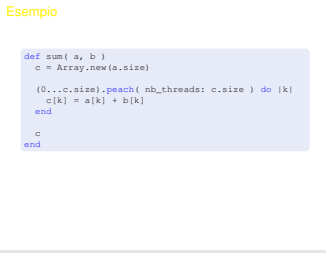
PRuby : Una libreria per la programmazione parallela
└Modelli di programmazione con PRuby
└Esempio: Somma di due vettori con



- Ecco un primo modo di fare la somma in parallelo, con il `pcall`
- Uno*thread* è creato, con la `lambda`, per ogni indice. Quindi, la `lambda` ha un argomento, che indica il proprio indice
- Questa soluzione é con *fine-grain parallelism* perché ogni attività da fare è piccolissima — *it cannot be decomposed*

Esempio: Somma di due vettori con «*fine-grain parallelism*» : `peach`

```
def sum( a, b )
  c = Array.new(a.size)

  (0...c.size).peach( nb_threads: c.size ) do |k|
    c[k] = a[k] + b[k]
  end

  c
end
```

- Ecco un altro modo, anche con *fine-grain parallelism*, di calcolare la somma in parallelo, però con il metodo `peach` — Parallel EACH
- I due motodi sono (totally equivalent?) completamente equivalenti

Esempio: Somma di due vettori con «*fine-grain parallelism*» : `pcall` vs. `peach`

```ruby
def sum( a, b )
  c = Array.new(a.size)

  PRuby.pcall( 0...c.size, lambda do |k|
    c[k] = a[k] + b[k]
  end )

  c
end
```

Domanda:
Perché non sono buone soluzioni ?

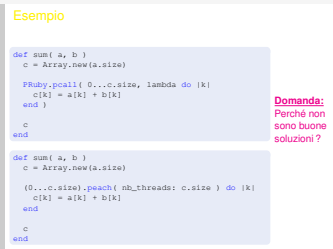
```ruby
def sum( a, b )
  c = Array.new(a.size)

  (0...c.size).peach( nb_threads: c.size ) do |k|
    c[k] = a[k] + b[k]
  end

  c
end
```

2017-07-04

PRuby : Una libreria per la programmazione parallela

└Modelli di programmazione con PRuby

└Esempio: Somma di due vettori con

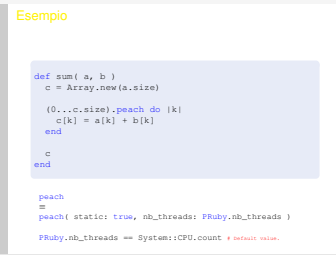


-

Esempio: Somma di due vettori con «*coarse-grain parallelism*» : `peach`

```ruby
def sum( a, b )
  c = Array.new(a.size)

  (0...c.size).peach do |k|
    c[k] = a[k] + b[k]
  end

  c
end
```

```ruby
peach
≡
peach( static: true, nb_threads: PRuby.nb_threads )

PRuby.nb_threads == System::CPU.count # Default value.
```

- Non sono buone soluzioni perché la *granularity* — il *grain-size* — è troppo piccola : il lavoro fatto per creare il *thread* è molto più che il lavoro fatto dal *thread*.
- Quindi, come abbiamo fatto con Fibonacci, dobbiamo abbassare il numero di *threads* e ogni *thread* deve fare lavoro di più
- Questo è molto facile con il `peach`, come si vede nel esempio : basta indicare nessun argomento
- Il `peach` senza argomento è come il `peach` con questi due argomenti

I parametri di `peach` sono come quelli di `OpenMP`

Esempio : 12 elementi da distribuire fra 3 *threads*, t_0, t_1, t_2

```
peach( static: true, nb_threads: 3 )
```

t_0	t_0	t_0	t_0	t_1	t_1	t_1	t_1	t_2	t_2	t_2	t_2

Block distribution

```
peach( static: 2, nb_threads: 3 )
```

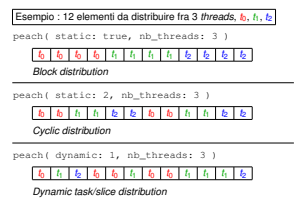
t_0	t_0	t_1	t_1	t_2	t_2	t_0	t_0	t_1	t_1	t_2	t_2

Cyclic distribution

```
peach( dynamic: 1, nb_threads: 3 )
```

t_0	t_1	t_2	t_0	t_0	t_1	t_0	t_0	t_1	t_1	t_1	t_2

Dynamic task/slice distribution



- Gli argomenti `static` (o `dynamic`) indicano come il lavoro da fare deve essere (distributed?) distribuito fra i *threads*
- `static: true` indica una *static distribution by block*, cioè, ogni *thread* va a trattare un *block* di elementi che sono adiacenti, come si vede nel grafico. E tutti i *threads* hanno, più o meno, la stessa quantità di elementi da trattare
- I parametri di `peach` sono come quelli di `OpenMP`. Si può anche indicare una *cyclic and static distribution* o una *dynamic distribution*.
- *Static* significa che gli elementi da trattare da un *thread* sono conosciuti quando il *thread* inizia.
- *Dynamci* significa che gli elementi da trattare da un *thread* sono conosciuti solo mentre il *thread* esegue, e dipende degli altri *threads*.

Data Parallelism

-

Data Parallelism

Data parallelism *is the simultaneous execution on multiple cores* *of the same function* *across the elements of a dataset.*

`http://blog.accelereyes.com/blog/2009/01/22/data-parallelism-vs-task-parallelism/`

- Permette parallelismo molto scalable :

⇒ Più dati ci sono da trattare, più parallelismo si può sfruttare

Data parallelism *is the simultaneous execution on multiple cores* *of the same function* *across the elements of a dataset.*
`http://blog.accelereyes.com/blog/2009/01/22/data-parallelism-vs-task-parallelism/`
- Permette parallelismo molto scalable :

⇒ Più dati ci sono da trattare, più parallelismo si può sfruttare

- Un altro modo di parallelismo è il *data parallelism*
- Si utilizza quando ci sono molti dati da trattare, tutti con la stessa funzione

I due motodi principali del *data parallelism*

`map`

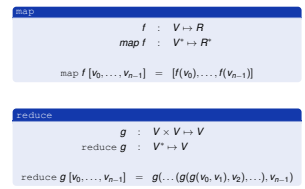
$$\begin{aligned} f &: V \mapsto R \\ map\ f &: V^* \mapsto R^* \end{aligned}$$

$$\texttt{map}\ f\ [v_0, \ldots, v_{n-1}] = [f(v_0), \ldots, f(v_{n-1})]$$

`reduce`

$$\begin{aligned} g &: V \times V \mapsto V \\ \texttt{reduce}\ g &: V^* \mapsto V \end{aligned}$$

$$\texttt{reduce}\ g\ [v_0, \ldots, v_{n-1}] = g(\ldots(g(g(v_0, v_1), v_2), \ldots), v_{n-1})$$



- Ci sono due metodi principali per il *data parallelism* : il metodo `map` e il metodo `reduce`
- A VERIFIER ! ! !
- Il metodo `map` riceve una funzione f, con un argomento, e una *collection*, per esempio v_0, v_1, ..., v_{n-1}. Il metodo `map` applica f su ogni elemento della *collection* e crea una nuova *collection*, con lo stesso numero di elementi
- Il metodo `reduce` riceve una funzione g, con due argomenti, e combina tutti gli elementi utilizzando g

I due metodi principali del *data parallelism*

map

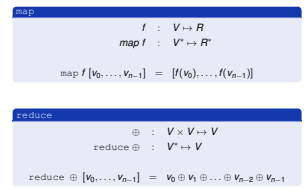
$$\begin{aligned} f &: V \mapsto R \\ \textit{map}\ f &: V^* \mapsto R^* \end{aligned}$$

$$\texttt{map}\ f\ [v_0, \ldots, v_{n-1}] = [f(v_0), \ldots, f(v_{n-1})]$$

reduce

$$\begin{aligned} \oplus &: V \times V \mapsto V \\ \texttt{reduce}\ \oplus &: V^* \mapsto V \end{aligned}$$

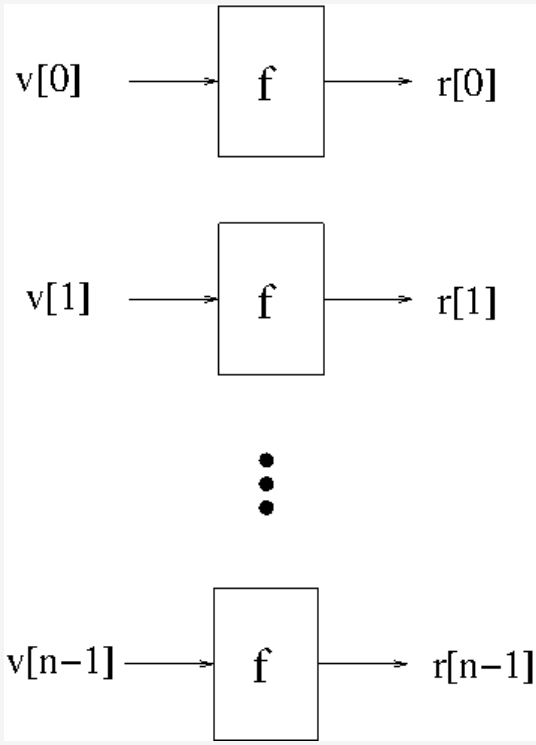
$$\texttt{reduce}\ \oplus\ [v_0, \ldots, v_{n-1}] = v_0 \oplus v_1 \oplus \ldots \oplus v_{n-2} \oplus v_{n-1}$$

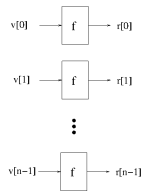


- Se si utilizza g come un operatore binario infisso $\oplus$ e $\oplus$ è associativo, il metodo `reduce` si può anche essere scritto così

La `map` è naturalmente parallela

```
[r[0], r[1], ..., r[n-1]] = map f [v[0], v[1], ..., v[n-1]]
```

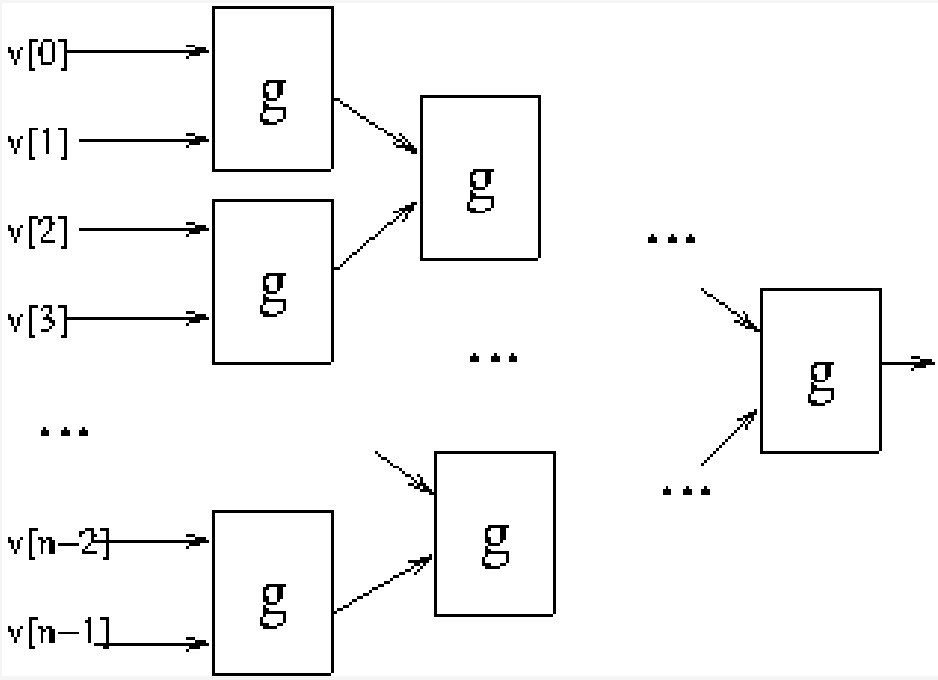


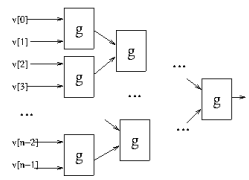


- Per il metodo `map`, ogni parte del risultato è indipendente, quindi si possono calcolare in parallelo

Il metodo `reduce` può anche essere computata in parallelo se . . . *g* è associativa

Con tempo asintotico di esecuzione con complessità $O(lg\ n)$



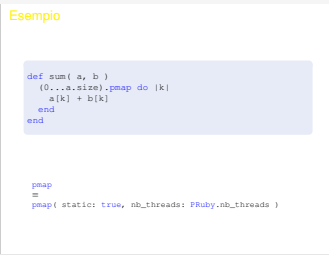


- Da solito, la funzione *g* è associativa, quindi il `reduce` può essere calcolato in parallelo così. Non è completamente parallelo come il `map` ma non è sequenziale

Esempio: Somma di due vettori con «*coarse-grain parallelism*» : `pmap`

```ruby
def sum( a, b )
  (0...a.size).pmap do |k|
    a[k] + b[k]
  end
end
```

```ruby
pmap
≡
pmap( static: true, nb_threads: PRuby.nb_threads )
```

- Ritorniamo all' esempio della somma di due vettori. Si può scriverlo così, utilizzando il metodo `pmap` — *parallel map* — con *coarse-grain parallelism*

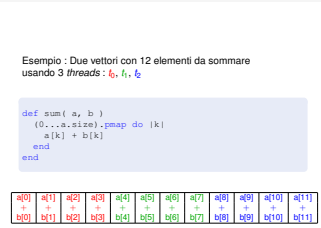
I parametri di `pmap` sono come quelli di `peach`

Default = Block distribution

Esempio : Due vettori con 12 elementi da sommare usando 3 *threads* : t_0, t_1, t_2

```
def sum( a, b )
  (0...a.size).pmap do |k|
    a[k] + b[k]
  end
end
```

a[0] + b[0]	a[1] + b[1]	a[2] + b[2]	a[3] + b[3]	a[4] + b[4]	a[5] + b[5]	a[6] + b[6]	a[7] + b[7]	a[8] + b[8]	a[9] + b[9]	a[10] + b[10]	a[11] + b[11]

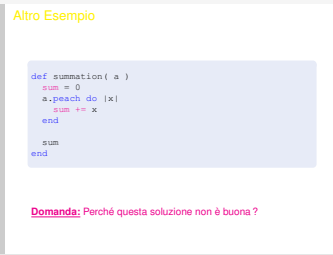


- È molto semplice perché il `pmap` ha gli stessi argomenti di `peach`, quindi utilizza una *block distribution* dei dati e un *thread* per *processor*/*core*

Altro Esempio: Somma degli elementi di un array con «*coarse-grain parallelism*» : `peach`

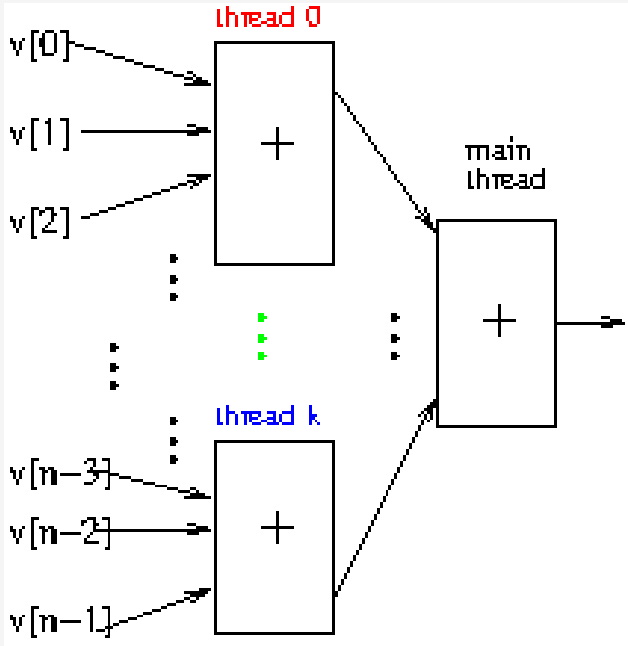
```
def summation( a )
  sum = 0
  a.peach do |x|
    sum += x
  end

  sum
end
```

Domanda: Perché questa soluzione non è buona ?

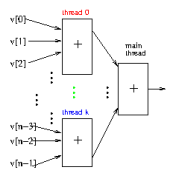


- Ecco un altro esempio, dove si deve calcolare la somma degli elementi di un array
- **Risposta:** Perché le iterazioni non sono indipendenti : c'è una *race condition* fra i *threads* per modificare la variabile `sum`

Altro Esempio: Somma degli elementi di un array con «*coarse-grain parallelism*» : `preduce`



Altro Esempio



-

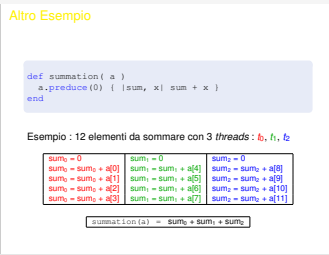
Altro Esempio: Somma degli elementi di un array con «*coarse-grain parallelism*» : `preduce`

```
def summation( a )
  a.preduce(0) { |sum, x| sum + x }
end
```

Esempio : 12 elementi da sommare con 3 *threads* : t_0, t_1, t_2

$sum_0 = 0$	$sum_1 = 0$	$sum_2 = 0$
$sum_0 = sum_0 + a[0]$	$sum_1 = sum_1 + a[4]$	$sum_2 = sum_2 + a[8]$
$sum_0 = sum_0 + a[1]$	$sum_1 = sum_1 + a[5]$	$sum_2 = sum_2 + a[9]$
$sum_0 = sum_0 + a[2]$	$sum_1 = sum_1 + a[6]$	$sum_2 = sum_2 + a[10]$
$sum_0 = sum_0 + a[3]$	$sum_1 = sum_1 + a[7]$	$sum_2 = sum_2 + a[11]$

`summation(a)` = $sum_0 + sum_1 + sum_2$



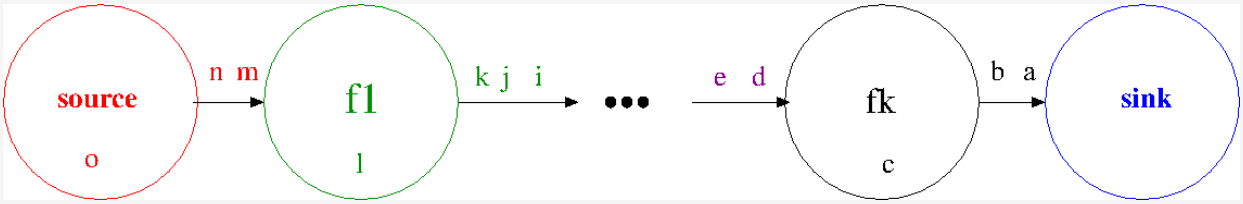
- Anziché utilizzare il metodo `peach`, qui si deve utilizzare il metodo `preduce` — *parallel reduce*, anche con *coarse-grain parallelism*

Stream Parallelism

-

Stream Parallelism

Quando ci sono diversi dati da trattare, con diversi trattementi — man mano che i dati sono prodotti



- Il quarto modello di programmazione di PRuby è lo *stream parallelism*
- Lo *stream parallelism* è spesso associato con le *pipelines*
- Il lavoro da fare è decomposto in una serie di stadi.
- Un pezzo di lavoro deve essere trattato da ogni stadio, uno dopo l'altro. Ma c'è del parallelismo perché i diversi stadi possono funzionare in parallelo
- Dunque, è come una catena di montaggio

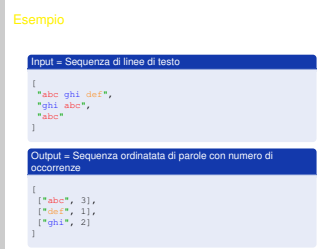
Esempio: *Word Count*

Input = Sequenza di linee di testo

```
[
 "abc ghi def",
 "ghi abc",
 "abc"
]
```

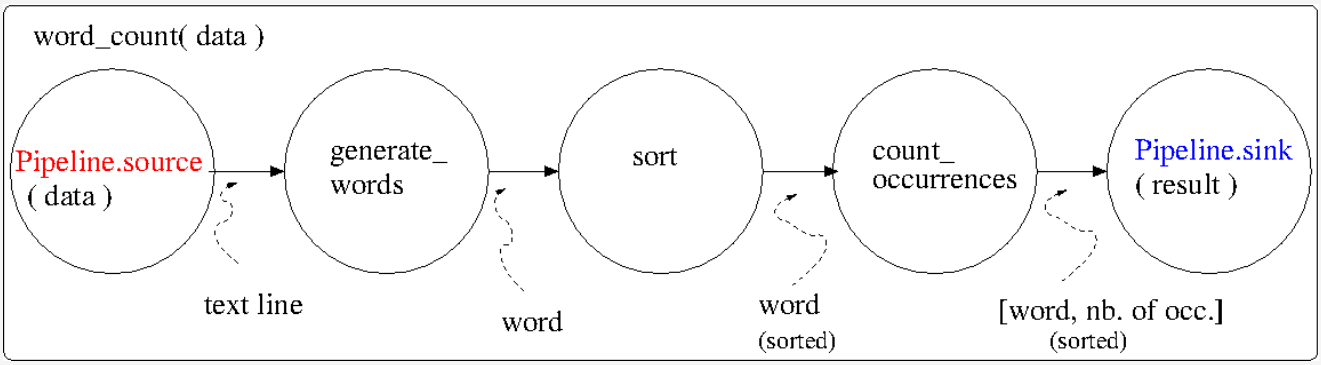
Output = Sequenza ordinatata di parole con numero di occorrenze

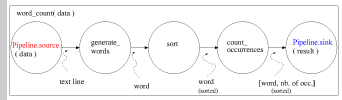
```
[
 ["abc", 3],
 ["def", 1],
 ["ghi", 2]
]
```

- Ecco un esempio, che è il *Hello world* per questo tipo di problema
- Da una sequenza di linee di testo, vogliamo produrre una sequenza ordinata delle parole con il numero di occorrenze
- Una parola eùna sequenza di lettere senze spazi
- Ecco un esempio con tre parole, con sei occorrenze

1. Con pipeline «*à la Unix*»

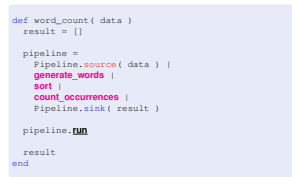




- Un primo modo di farlo in PRuby è utilizzando *a Unix-style pipeline*
- Ci sono cinque stadi, però guarderemo solamente i tre stadi principali

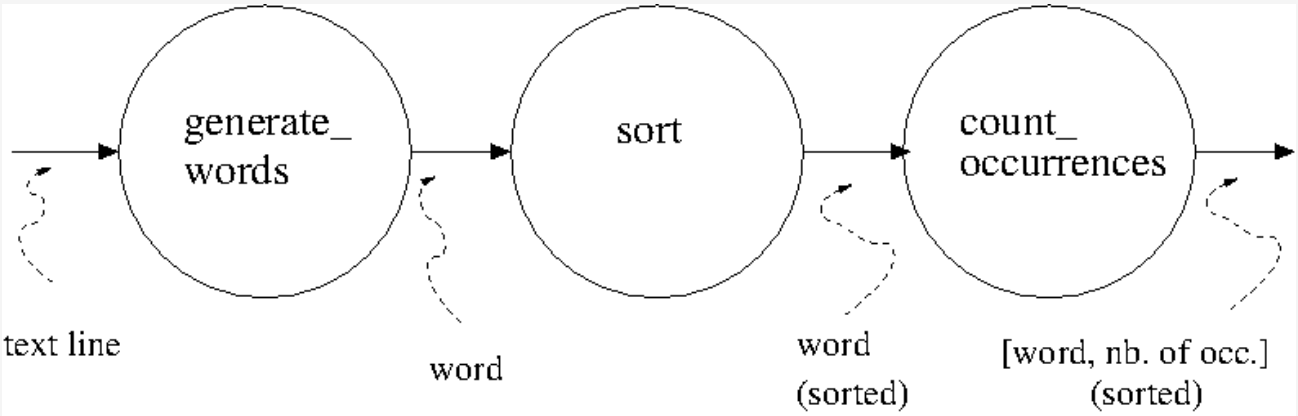
1. Con pipeline «*à la Unix*»

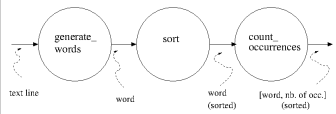
```ruby
def word_count( data )
  result = []

  pipeline =
    Pipeline.source( data ) |
    generate_words |
    sort |
    count_occurrences |
    Pipeline.sink( result )

  pipeline.run

  result
end
```

- Ecco la *top-level pipeline* scritta con PRuby
- Una *pipeline* è creata, utilizzando il simbolo per la *pipe* e poi la *pipeline è eseguita*, con il metodo `run`

1. Con pipeline «*à la Unix*»

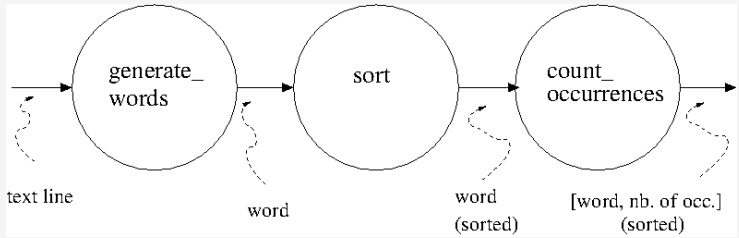


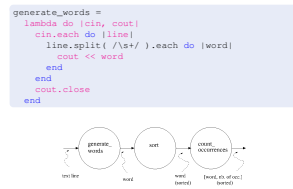


- Il primo stadio trova e genera le parole da ogni linea di testo
- Il secondo stadio fa il *sort*, quindi dopo questo stadio le parole sono ordinatate
- Il terzo stadio conta il numero di occorrenze per ogni parola : è facile da calcolare perché le varie occorrenze d'una parola sono adiacenti

1. Con pipeline «*à la Unix*»
`generate_words`

```
generate_words =
  lambda do |cin, cout|
    cin.each do |line|
      line.split( /\s+/ ).each do |word|
        cout << word
      end
    end
    cout.close
  end
```



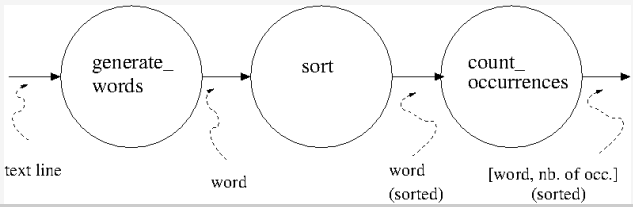


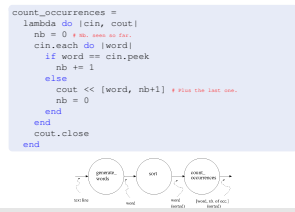
- Ecco la definizione del processo per `generate_words`
- Il processo è una *lambda-expression* con due argomenti : l'*input channel* e l'*output channel*
- Si utilizza il metodo *each* per leggere gli elementi dall' *input channel*
- Si utilizza il metodo *push* per scrivere sull' *output channel*
- E si utilizza *close* per indicare la fine dello *stream*, l'*end of stream*

1. Con pipeline «*à la Unix*»

count_occurrences

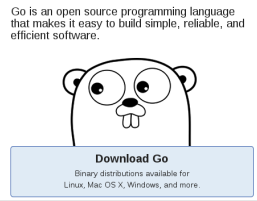
```
count_occurrences =
  lambda do |cin, cout|
    nb = 0 # Nb. seen so far.
    cin.each do |word|
      if word == cin.peek
        nb += 1
      else
        cout << [word, nb+1] # Plus the last one.
        nb = 0
      end
    end
    cout.close
  end
```





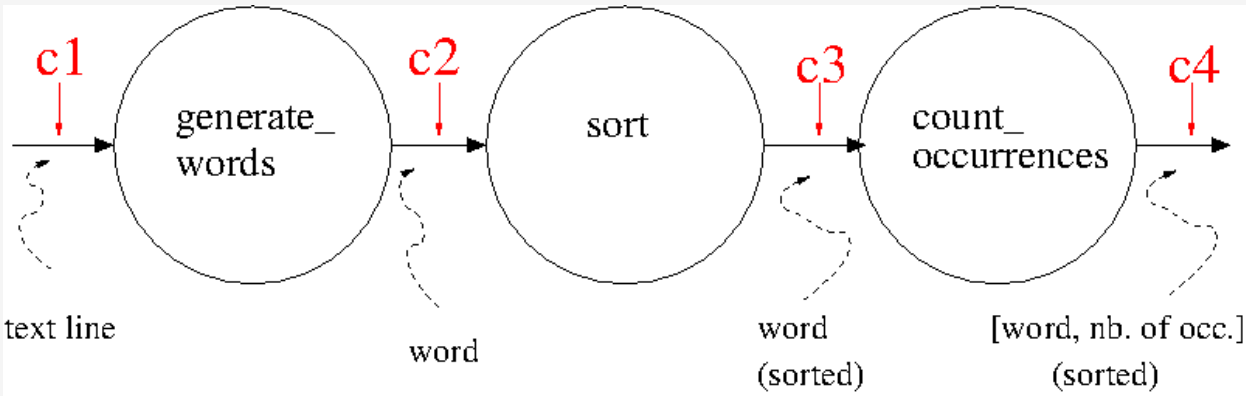
-

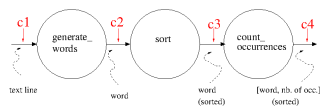
2. Con canali espliciti «*à la Go*»



- Go è un languaggio di programmazione che è concurrent. È stato svilupppato da Google, per Griesemer, Pike e Thompson — Rob Pike e Ken Thompson sono *well-known C and Unix guys*

2. Con canali espliciti «*à la Go*»

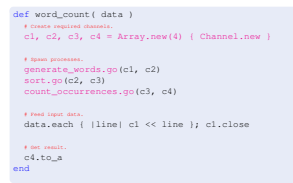




- Un altro modo per descrivere la stessa *pipeline* è con lo stile «*à la Go*», dove si utilizzano dei canali espliciti

2. Con canali espliciti «*à la Go*»

```ruby
def word_count( data )
  # Create required channels.
  c1, c2, c3, c4 = Array.new(4) { Channel.new }

  # Spawn processes.
  generate_words.go(c1, c2)
  sort.go(c2, c3)
  count_occurrences.go(c3, c4)

  # Feed input data.
  data.each { |line| c1 << line }; c1.close

  # Get result.
  c4.to_a
end
```

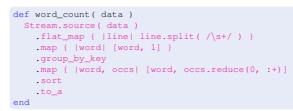



- Go si ispira da CSP di Hoare, quindi, si utilizza *explicit communication channels with implicit processes*
- Con PRuby, i *channels* sono creati con Channel.new
- E poi, i processi, che sono anche *lambda-expressions*, vengono lanciati con *go*
- In questo esempio, le definizioni dei processi sono le stesse dell'esempio precedente

3. Con streams «*à la Spark/Java 8.0*»

Usa l'operazione `group_by` su coppie `[parola, numero]`

```
def word_count( data )
  Stream.source( data )
    .flat_map { |line| line.split( /\s+/ ) }
    .map { |word| [word, 1] }
    .group_by_key
    .map { |word, occs| [word, occs.reduce(0, :+)] }
    .sort
    .to_a
end
```

- Il terzo modo per *stream parallelism* è usando gli *streams* «à la Spark» o «à la Java 8.0»
- In questo modo, un processamento parallelo è anche una serie di stadi, ma dove il risultato da ogni stadio è visto come una collezione completa, sulla quella si usa un metodo sull'intera collezione.
- I vari stadi si aggiungono con *method chaining*
- Ecco lo stesso esempio, ma con gli *streams*
- Il primo stadio genera una collezione di linee
- Il secondo stadio genera una collezione di parole
- Il terzo stadio genera una collezione di coppia, dove il primo elemento è una parola e il secondo elemento è il numero uno
- Alcuni stadi possono trattare la collezione in un modo *incremental*, cioè, trattando un elemento, e poi un altro, ecc.
- In questi casi, c'é del parallelismo tra gli processi

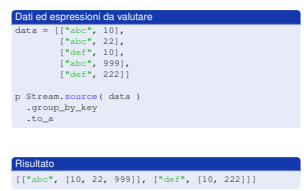
Un esempio di utilizzo di `group_by_key`

Dati ed espressioni da valutare

```
data = [["abc", 10],
        ["abc", 22],
        ["def", 10],
        ["abc", 999],
        ["def", 222]]

p Stream.source( data )
  .group_by_key
  .to_a
```

Risultato

```
[["abc", [10, 22, 999]], ["def", [10, 222]]]
```

-

3. Con streams «*à la Spark/Java 8.0*»

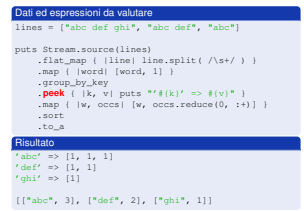
Effetto ottenuto dall'utilizzo de `group_by_key`

Dati ed espressioni da valutare

```
lines = ["abc def ghi", "abc def", "abc"]

puts Stream.source(lines)
    .flat_map { |line| line.split( /\s+/ ) }
    .map { |word| [word, 1] }
    .group_by_key
    .peek { |k, v| puts "'#{k}' => #{v}" }
    .map { |w, occs| [w, occs.reduce(0, :+)] }
    .sort
    .to_a
```

Risultato

```
'abc' => [1, 1, 1]
'def' => [1, 1]
'ghi' => [1]

[["abc", 3], ["def", 2], ["ghi", 1]]
```

- Altri stadi non si possono fare così, perché devono vedere tutta la collezione prima di produrre un risultato.
- Qui, ci sono due stadi di questo tipo : `group_by_key` e `sort`
- Perché non c'é abbastanza tempo, non spiegherò il `group_by_key`. Ma per il `sort`, è facile vedere che nessuna parte del risultato può essere prodotta prima che tutti gli elementi sono conosciuti — forse l'ultimo elemento è il minimo, quindi si deve aspettare il suo arrivo…
- Ogni tappi del pipeline è eseguito da almeno un thread indipendente. Alcuni metodi possono anche essere esiguiti da parecchi threads, per esempio, map, filter, ecc.

Conclusione

-

La mia esperienza con PRuby

È stato utilizzato quattro volte : $2 \times INF5171 + 2 \times INF7235$

- PRuby è facile da imparare e utilizzare
- PRuby rende più semplice l'apprendimento di altri linguaggi di programmazione
- PRuby piace alla maggior parte degli studenti

- PRuby è facile da imparare e utilizzare
- PRuby rende più semplice l'apprendimento di altri linguaggi di programmazione
- PRuby piace alla maggior parte degli studenti

-

Però, Ruby non piace a tutti ☹

Nessun linguaggio è perfetto

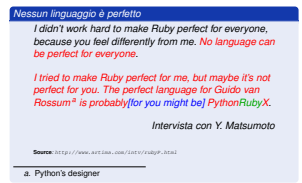
I didn't work hard to make Ruby perfect for everyone, because you feel differently from me. No language can be perfect for everyone.

I tried to make Ruby perfect for me, but maybe it's not perfect for you. The perfect language for Guido van Rossum[a] is probably[for you might be] PythonRubyX.

Intervista con Y. Matsumoto

Source: `http://www.artima.com/intv/rubyP.html`

a. Python's designer



- Ma, per alcuni studenti, «Ruby is strange» — è strano
- Matz ha detto anche che nessun linguaggio è perfetto per tutti
- Credo che sia perfetto per me, pero non è per voi

Sviluppi futuri

- Ottimizzazioni !
- Aggiungere altri modelli di programmazione
  - Actors «*à la Celluloid*» ?

- Ottimizzazioni !
- Aggiungere altri modelli di programmazione
  - Actors «*à la Celluloid*» ?

-

Sviluppi futuri
Actors à la Celluloid

```ruby
class A
  include Actor

  def foo( n ); ... end
  def bar;      ... end
end

a = A.new

# Synchronous calls.
a.foo(N)
a.bar

# Asynchronous calls.
a.async.foo(N)
f = a.future.bar
...

p f.value
```


class A
include Actor
def foo(n); ... end
def bar; ... end
end
a = A.new
# Synchronous calls.
a.foo(N)
a.bar
# Asynchronous calls.
a.async.foo(N)
f = a.future.bar
...
p f.value

- *Actors* à la Celluloid, da Ruby, sono simili agli *actors* di Akka, Java o Scala, ma un po' più semplici

Sviluppi futuri

- Ottimizzazioni !
- Aggiungere altri modelli di programmazione
  - Actors «*à la Celluloid*» ?
- Aggiungere nuovi costruzioni
  - `Hash` : `peach`, `peach_key`, `pmap` ?

- Ottimizzazioni !
- Aggiungere altri modelli di programmazione
  - Actors «*à la Celluloid*» ?
- Aggiungere nuovi costruzioni
  - `Hash` : `peach`, `peach_key`, `pmap` ?

-

Domande ?

Domande ?

- Grazie. Spero che il mio accento strano non abbia scorticato troppo le vostra orecchie
- E grazie a Claudia Misale, che ha letto le mie diapositive e le mie note e ha fatto numerose correzioni