Programação Paralela em Haskell

Fabrício Olivetti de França

02 de Fevereiro de 2019

Quem sou eu?

Quem sou eu?

Professor do curso de Ciência da Computação da Universidade Federal do ABC.

Linha de Pesquisa: Co-agrupamento de Dados, Regressão Simbólica.

Quem sou eu?

Coordenador do Heuristics, Analysis and Learning Laboratory (HAL)

http://pesquisa.ufabc.edu.br/hal/

Quem sou eu?

Anteriormente consultor de uma Startup onde ajudei a adaptar algoritmos para o paradigma MapReduce.

Como cheguei aqui?

Desde 2013 fiquei responsável pela disciplina de Big Data na pós-graduação.

Foco: algoritmos escaláveis em relação a quantidade de entrada de dados e versão distribuída de algoritmos tradicionais de Aprendizado de Máquina.

Desafios

Desafios: não tenho um cluster disponível, não tenho verba para Amazon AWS, como o aluno pode perceber se o algoritmo está distribuindo corretamente as tarefas?

Soluções

Python MRJob https://pythonhosted.org/mrjob/: possibilidade de executar dentro de uma rede interna em diversas máquinas heterogêneas.

Soluções

Porém, a versão utilizada era instável, alto overhead no envio de pacotes pela rede e limitado quanto ao que podia ser paralelizado.

Soluções

Utilizar uma instalação local de **Apache Spark** para que os alunos tivessem contato com uma ferramenta muito utilizada atualmente.

Soluções

Porém, era difícil perceber o ganho em uma máquina local e muitas das soluções feitas pelos alunos era *colocar a base de dados na memória* e processar.

Dificuldades

O paradigma MapReduce segue um paradigma funcional, poucos estão acostumados com esse paradigma!

Dificuldades

Migrar de Python para Scala? Os alunos continuaram com um pensamento orientado a objetos ou procedural...

Solução final (por enquanto)

Linguagem oficial do curso é Haskell!

(tente implementar utilizando outro paradigma agora!)

Vantagens:

- linguagem funcional (e nada mais que funcional)
- o compilador possui suporte nativo a *threads*
- sintaxe limpa (maioria dos algoritmos cabe em um slide)
- avaliação preguiçosa (assim como no MapReduce)

Ferramenta que permite verificar se a computação está realmente distribuída:

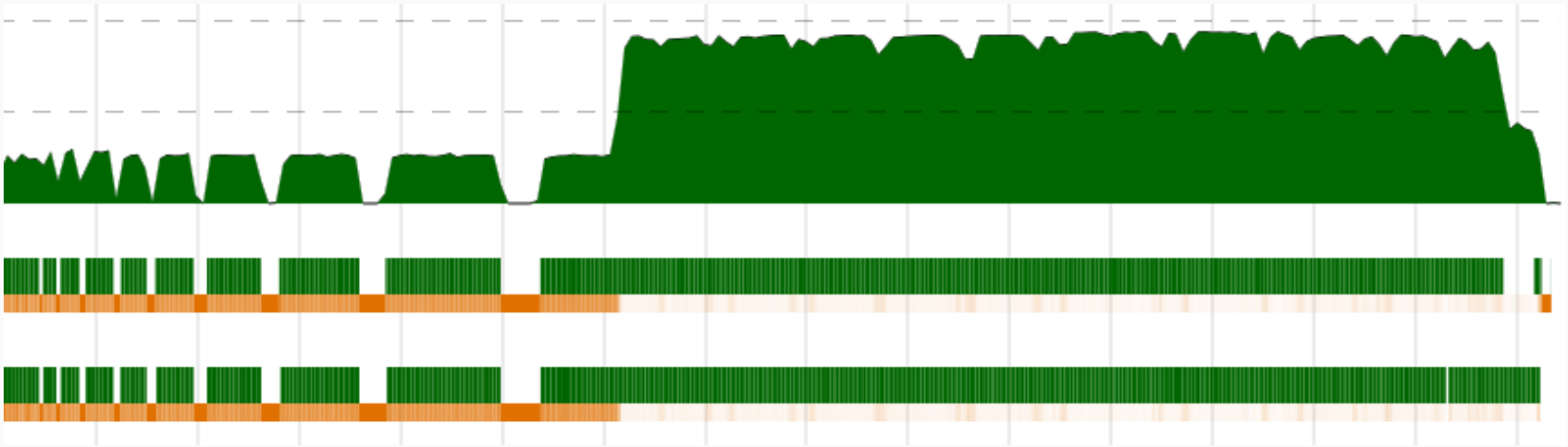


Figura 1: Threadscope

Desvantagens:

- Duas semanas de curso expresso de programação funcional com Haskell (sofrido)
- Quando descobrem que Monads não são burritos...e o curso não oferece coffee-break

Resultados

Embora alguns alunos apresentaram dificuldades para programar em Haskell, boa parte concluiu o curso com códigos de algoritmos complexos e que apresentaram ganho considerável ao serem paralelizados.

Amor Funcional

Desde então, junto com o prof. Emílio, oferecemos cursos de extensão em programação funcional para a comunidade:

http://pesquisa.ufabc.edu.br/haskell/

Ainda esse ano, cursos de:

- Introdução à Haskell
- Teoria das Categorias para Programadores
- Estrutura de Dados Funcionais

Novo desafio

Consigo ensinar Haskell **E** Programação Paralela em cerca de 3 horas?

Novo desafio

Vamos tentar fazer isso com **categoria**.

Haskell

Haskell

- Surgiu em 1990 com o objetivo de ser a primeira linguagem puramente funcional.
- Por muito tempo considerada uma linguagem acadêmica.
- Atualmente é utilizada em diversas empresas (totalmente ou em parte de projetos).

Haskell: Principais Características

- Códigos concisos e declarativos
- Sistema de tipagem forte
- Imutabilidade
- Funções de alta ordem
- Tipos polimórficos
- Avaliação preguiçosa

Por que Haskell?

Programação em Haskell segue uma linha de raciocínio comum na computação: **Dividir para conquistar**.

- Quebramos nosso problema em pequenos pedaços
- Resolvemos cada pedaço
- Juntamos os pedaços

Por que Haskell?

Isso é feito através da composição de funções!

Exemplo

Considere uma função em C++ que soma o quadrado dos elementos positivos :

```
double soma_quad_pos(std::vector<double> xs) {
    double soma = 0.0;
    for (auto x : xs)
    {
        if (x > 0) soma += x*x;
    }

    return soma;
}
```

Exemplo

Em Haskell poderíamos fazer:

```haskell
soma_quad_pos :: [Double] -> Double
soma_quad_pos xs = (sum . fmap (^2) . filter (>0)) xs
```

ou

```haskell
soma_quad_pos :: [Double] -> Double
soma_quad_pos = sum . fmap (^2) . filter (>0)
```

Abstração

Outra vantagem do Haskell é o poder de abstração. Uma função no Haskell é sempre acompanhada de uma **assinatura** que indica os tipos de entrada e saída:

```
dobra :: Int -> Int
dobra x = 2*x
```

Notação próxima da matemática

Abstração

Podemos generalizar uma função utilizando parâmetros de tipos (similar aos *templates*, porém mais poderosos):

```haskell
-- funciona para qualquer tipo numérico
dobra :: Num a => a -> a
dobra x = 2*x
```

Categoria dos Tipos

Uma Categoria é definida por um conjunto de objetos e morfismos (relações), um morfismo identidade, e um operador de composição que possui transitividade:

$$f \circ (g \circ h) = (f \circ g) \circ h$$

Categoria dos Tipos

Os tipos no Haskell formam uma categoria, pois temos a função identidade:

```
id :: a -> a
id x = x
```

E o operador transitivo de composição: `.`

Funções e Tipos

Funções e Tipos

No Haskell tudo gira em torno dos **tipos de dados algébricos** (*Type-level programming*) e das funções que podemos definir com eles.

Vamos começar por alguns tipos básicos!

Tipos Básicos

Um **tipo** é um **conjunto de valores** associados com alguma propriedade! Booleano possui dois valores, Int possui 2^{32}, etc.

Caracterizamos os tipos pela quantidade de valores que ele possui!

Tipos Básicos

O tipo mais simples possível é aquele sem elemento algum:

```
data Void
```

Que funções podemos definir com esse tipo?

Tipos Básicos

```haskell
absurd :: Void -> a
```

ex falso quodlibet

Tipos Básicos

Vamos definir agora o tipo com apenas um elemento, chamado de *unit*:

```
data () = ()
```

Quais funções podemos definir?

Tipos Básicos

```haskell
unit :: a -> ()
unit x = ()

choose :: () -> Int
choose () = 10
-- == x = 10
```

Tipos Básicos

E um tipo com dois valores?

```
data Bool = True | False
```

Tipos Básicos

Além disso temos como padrão os tipos:

- **Bool:** contém os valores `True` e `False`.
- **Int:** inteiros com precisão fixa em 64 bits.
- **Integer:** inteiros de precisão arbitrária.
- **Float:** valores em ponto-flutuante de precisão simples.
- **Double:** valores em ponto-flutuante de precisão dupla.
- **Char:** contém todos os caracteres no sistema **Unicode**.
- **String:** lista de caracteres.

Tipo Produto

Um **tipo produto** é a combinação de dois ou mais tipos (tuplas!) e é definido como:

```
data Par = Par Int String
```

Nesse caso temos um tipo equivalente a tupla `(Int, String)`.

Tipo Produto

Esses tipos podem ser parametrizados:

```haskell
data Par a b = Par a b
-- = (a, b)
```

Tipo Soma

O **tipo soma** é descrito por:

```haskell
data Either a b = Left a | Right b
```

e indica que um valor desse tipo ou pertence ao tipo **a** ou ao tipo **b**.

Tipos Recursivos

Um tipo pode ter uma representação recursiva:

```
data [a] = [] | a : [a]
```

Tipos Recursivos

Com isso podemos definir listas como:

```haskell
xs :: [Int]
xs = 1 : 2 : 3 : 4 : []
```

ou

```haskell
xs :: [Int]
xs = [1, 2, 3, 4]
```

Functors

Containers

Os tipos paramétricos podem ser entendidos como um container!

Temos uma lista de inteiros, um mapa entre caracter e número real, tupla de inteiro com booleano, etc.

Mapa de morfismos

Digamos que temos uma **lista** `[Int]` e uma função que transforma um `Int` em uma `String`, ou seja, `f :: Int -> String`. Podemos criar uma função `g :: [Int] -> [String]`?

Mapa de morfismos

```haskell
g :: [Int] -> [String]
g []     = []
g (x:xs) = f x : g xs
```

Mapa de morfismos

Essa função pode ser generalizada para quaisquer tipos `a` e `b`?

Mapa de morfismos

```haskell
g :: (a -> b) -> ([a] -> [b])
g f []     = []
g f (x:xs) = f x : g xs
```

Mapa de morfismos

Essa função é um mapa entre uma função `a -> b` para uma função `F a -> F b`, sendo `F` um container de dados. Ela é chamada de `fmap`.

Functor

Um **Functor** é um tipo paramétrico *equipado* com uma função `fmap` que transforma funções entre dois tipos para funções desses mesmos tipos, só que dentro dos containers:

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

Monads

Traço de execução

Imagine que temos o seguinte trecho de código:

```
not :: Bool -> Bool
not True  = False
not False = True

even :: Int -> Bool
even x = x `mod` 2 == 0
```

Traço de execução

Para definir a função **odd**, basta fazer:

```
odd :: Int -> Bool
odd = not . even
```

Traço de execução

Seu gerente de projetos pediu para que você criasse um *log* contendo a sequência de execução das funções! Ou seja, se o seu programa executa:

```
odd (bool2int (not (even x)))
```

O log deve conter “even not bool2int odd”.

Traço de execução

Como você resolveria em linguagens tradicionais?

Traço de execução

Uma tentação é o uso de uma variável global, e alterar cada função para concatenar a string no log. Isso é razoável?

Traço de execução

Outra alternativa é alterar cada uma das funções para receber o log atual e retornar um novo log:

```
not :: (Bool, String) -> (Bool, String)
not (True, s) = (False, s ++ " not ")
not (False, s) = (True, s ++ " not ")

even :: (Int, String) -> (Bool, String)
even (x, s) = (x `mod` 2 == 0, s ++ " even ")
```

mas isso demanda muito trabalho...

Traço de execução

Uma solução mais elegante é criarmos funções auxiliares que nos permitam:

- Criar uma função que retorne um *log* padrão, a partir de uma função qualquer.
- Criar um sequenciamento de funções que retornem *logs*, que gerencie a concatenação deles.

Traço de execução

```haskell
not :: Bool -> Bool
not b = not b

notW = createLog . not
```

Traço de execução

```haskell
evenW :: Int -> (Bool, String)
evenW x = (x `mod` 2 == 0, " even ")

(evenW x) `bindTo` notW
-- ~ notW (evenW x)
```

Traço de execução

A função `bindTo` funciona de forma similar ao *pipe* do unix, com alguns extras:

1. Aplica a função inicial no argumento fornecido, retornando o resultado com o log
2. Repassa apenas o resultado para a função seguinte (*bind*), retornando o novo resultado e os logs concatenados

Traço de execução

Com isso podemos criar uma sequência lógica de aplicação de funções *embelezadas* de tal forma que o tratamento do *log* seja feito externamente pela função `bindTo`.

Função `bindTo`

A função `bindTo` deve ter a seguinte assinatura:

```
bindTo :: (a, String)
       -> (a -> (b, String)) -> (b, String)
```

recebe um tipo `a` embelezado, uma função de `a` para um tipo embelezado e retorna essa saída cuidando de manter o *log*.

Função bindTo

```
bindTo :: (a, String)
       -> (a -> (b, String)) -> (b, String)
bindTo (a, s) fmb = (b, s ++ s')
  where (b, s') = fmb a
```

Função `createLog`

Para a função `createLog` temos a seguinte assinatura:

```
createLog :: a -> (a, String)
```

Função `createLog`

A definição é simples:

```haskell
createLog :: a -> (b, String)
createLog a = (a, "")
```

Função `createLog`

Como essa função está simplesmente embelezando um valor qualquer com uma string neutra, vamos renomeá-la para `return`:

```
return :: a -> (b, String)
return a = (a, "")
```

Ela retorna nosso valor embelezado!

Embelezando a função `not`

Mas queremos também fazer com que nossa função `not` seja embelezada com a string ”not ”, para isso podemos usar nossa função `bindTo`:

```
notW x = ((),” not ”) `bindTo` (\() -> return (not x))
```

Embelezando a função **not**

Essa função **diz** para a função embelezada **not** incorporar a string ”not ”. Podemos deixar mais claro com:

```
tell :: String -> ((), String)
tell s = ((), s)

passThru :: ((), String) -> (a, String) -> (a, String)
passThru toldS as = toldS `bindTo` (\() -> as)
```

Embelezando a função `not`

Com isso nossa função **`notW`** se torna:

```
notW x = tell " not " `passThru` return (not x)
```

Traço de execução

Em resumo criamos funções que:

1. Transforma um tipo `a` para `(a, String)`, armazenando um log padrão (`return`).

Traço de execução

Em resumo criamos funções que:

2. Dados um tipo `(a, String)` e uma função `a -> (b, String)`, gere um `(b, String)` com os dois logs concatenados (`bindTo`).

Traço de execução

Em resumo criamos funções que:

3. Pegue uma `String` e retorne um `((), String)`, representando um *transportador* de logs (`tell`).

Traço de execução

Em resumo criamos funções que:

4. Receba um `((), String)` e um `(a, String)` e concatene as `String` sem alterar o valor de `a` (`passThru`).

Tipo Writer

No Haskell, nosso embelezador é generalizado pelo tipo `Writer`:

```haskell
data Writer s a = Writer (a, s)
```

Ele permite que o *log* tome outras formas além de uma `String`.

Tipo Writer

Nossa função `bindTo` é implementada pelo operador `>>=`.

A função `passThru` é o operador `>>`.

Tipo Writer

Com esses operadores, podemos transformar as funções `not` e `even` originais em funções `Writers` com:

```
tell s = Writer ((), s)

notW b = tell " not " >> return (not b)
```

Syntactic Sugar

Da forma que está, a sintaxe pode ficar monstruosamente confusa em pouco tempo! Para resolver isso foi introduzida a notação **do** que transforma a aplicação desses operadores em algo mais simples e natural:

```
notW b = do tell " not "
            return (not b)
```

Syntactic Sugar

E a função `oddW` se tornaria:

```
oddW x = do tell " odd "
            tell " even "
            ev <- return (even x)
            tell " not "
            return (not ev)
```

Syntactic Sugar

Cada linha contendo uma seta para esquerda é concatenada com `>>=`, sem seta concatena com `>>`.

Syntactic Sugar

A seta para esquerda `<-` significa: *o nome a esquerda contém o valor dentro do embelezamento da direita*. No nosso caso o primeiro valor da tupla do `Writer`.

Monad

Um **Monad** é todo container que também é um Functor e que possui uma função `return` e um operador `>>=`.

Ou seja, são tipos embelezados com um contexto que possuem uma forma de terem suas funções compostas.

Programação Paralela e Concorrente em Haskell

Paralelismo vs Concorrência

Um programa **paralelo** é aquele que usa diversos recursos computacionais para terminar a tarefa mais rápido. Distribuir os cálculos entre diferentes processadores.

Paralelismo vs Concorrência

Um programa **concorrente** é uma técnica de estruturação em que existem múltiplos caminhos de controle. Conceitualmente, esses caminhos executam em paralelo, o usuário recebe o resultado de forma intercalada. Se realmente os resultados são processados em paralelo é um detalhe da implementação.

Paralelismo vs Concorrência

Imagine uma lanchonete servindo café. Nós podemos ter:

- Um caixa único e uma fila única: processamento sequencial
- Um caixa único e múltiplas filas: processamento concorrente
- Múltiplos caixas e uma fila: processamento paralelo
- Múltiplos caixas e múltiplas filas: processamento concorrente e paralelo

Paralelismo vs Concorrência

Uma outra distinção é que o processamento paralelo está relacionado com um **modelo determinístico** de computação enquanto o processamento concorrente é um **modelo não-determinístico**.

Os programas concorrentes sempre são não-determinísticos pois dependem de agentes externos (banco de dados, conexão http, etc.) para retornar um resultado.

Haskell Paralelo

No Haskell o paralelismo é feito de forma declarativa e em alto nível.

Não é preciso se preocupar com *sincronização* e *comunicação*.

Haskell Paralelo - Vantagens

- Programador não precisa se preocupar com detalhes específicos de implementação
- Funciona em uma diversidade de hardwares paralelos
- Melhorias futuras na biblioteca de paralelismo tem efeito imediato (ao recompilar) nos programas paralelos atuais

Haskell Paralelo - Desvantagens

- Como os detalhes técnicos estão escondidos, problemas de performance são difíceis de detectar
- Uma vez detectados, os problemas de perfomance são difíceis de resolver

Haskell Paralelo

A única tarefa do programador é a de dividir as tarefas a serem executadas em pequenas partes que podem ser processadas em paralelo para depois serem combinadas em uma solução final.

O resto é trabalho do compilador...

Avaliação Preguiçosa

Quero evitar a fadiga

Vamos verificar como a avaliação preguiçosa funciona no Haskell.

Para isso utilizaremos a função *sprint* no ghci que mostra o estado atual da variável.

Quero evitar a fadiga

```
Prelude> :set -XMonomorphismRestriction
Prelude> x = 5 + 10
Prelude> :sprint x
x = _
```

Quero evitar a fadiga

```
Prelude> x = 5 + 10
Prelude> :sprint x
x = _
Prelude> x
3
Prelude> :sprint x
x = 3
```

O valor de x é computado apenas quando requisitamos seu valor!

Quero evitar a fadiga

```
Prelude> x = 1 + 1
Prelude> y = x * 3
Prelude> :sprint x
x = _
Prelude> :sprint y
y = _
```

Quero evitar a fadiga

```
Prelude> x = 1 + 1
Prelude> y = x * 3
Prelude> :sprint x
x = _
Prelude> :sprint y
y = _
Prelude> y
6
Prelude> :sprint x
x = 2
```

Eu quero agora!

A função seq recebe dois parâmetros, avalia o primeiro e retorna o segundo.

```
Prelude> x = 1 + 1
Prelude> y = 2 * 3
Prelude> :sprint x
x = _
Prelude> :sprint y
y = _
Prelude> seq x y
6
Prelude> :sprint x
x = 2
```

Quero evitar a fadiga

```
Prelude> let l = map (+1) [1..10] :: [Int]
Prelude> :sprint l
l = _
Prelude> seq l ()
Prelude> :sprint l
l = _ : _
Prelude> length l
Prelude> :sprint l
l = [_,_,_,_,_,_,_,_,_,_]
Prelude> sum l
Prelude> :sprint l
l = [2,3,4,5,6,7,8,9,10,11]
```

Weak Head Normal Form (WHNF)

Ao fazer:

```
> z = (2, 3)
> z `seq` ()
()
> :sprint z
z = (_,_)
```

A função *seq* apenas forçou a avaliação da estrutura de tupla. Essa forma é conhecida como Weak Head Normal Form.

Normal Form

Para avaliar uma expressão em sua forma normal, podemos usar a função **force** da biblioteca **Control.DeepSeq**:

```
> z = (2,3)
> force z
> :sprint z
z = (2,3)
```

Eval Monad

Eval Monad

A biblioteca **Control.Parallel.Strategies** fornece os seguintes tipos e funções para criar paralelismo:

```
data Eval a (instance Monad Eval)

runEval :: Eval a -> a

rpar :: a -> Eval a
rseq :: a -> Eval a
```

Eval Monad

A função `rpar` indica que *meu argumento pode ser executado em paralelo*, já a função `rseq` diz *meu argumento deve ser avaliado e o programa deve esperar pelo resultado*.

Em ambos os casos a avaliação é para WHNF. Além disso, o argumento de `rpar` deve ser uma expressão ainda não avaliada, ou nada útil será feito.

Eval Monad

Finalmente, a função `runEval` executa uma expressão (em paralelo ou não) e retorna o resultado dessa computação.

Eval Monad

Considere a implementação ingênua de fibonacci:

```haskell
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib 2 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Eval Monad

Digamos que queremos obter o resultado de `fib 40` e `fib 41`:

```
f = (fib 41, fib 40)
```

Podemos executar as duas chamadas de `fib` em paralelo!

Eval Monad

```haskell
fparpar :: Eval (Integer, Integer)
fparpar = do a <- rpar (f 41)
             b <- rpar (f 40)
             return (a, b)
```

Eval Monad

Medindo o tempo computacional utilizando uma única thread temos:

```
Tempo para executar fparpar: 0.000s
Resultado: (165580141,102334155)
Tempo apos imprimir resultado: 15.691738s
```

Eval Monad

E para duas threads:

```
Tempo para executar fparpar: 0.000s
Resultado: (165580141,102334155)
Tempo apos imprimir resultado: 9.96815ss
```

Com duas threads o tempo é reduzido pois cada thread calculou um valor de fibonacci em paralelo. Note que o tempo não se reduziu pela metade pois as tarefas são desproporcionais.

rpar-rpar

A estratégia `rpar-rpar` não aguarda o final da computação para liberar a execução de outras tarefas:

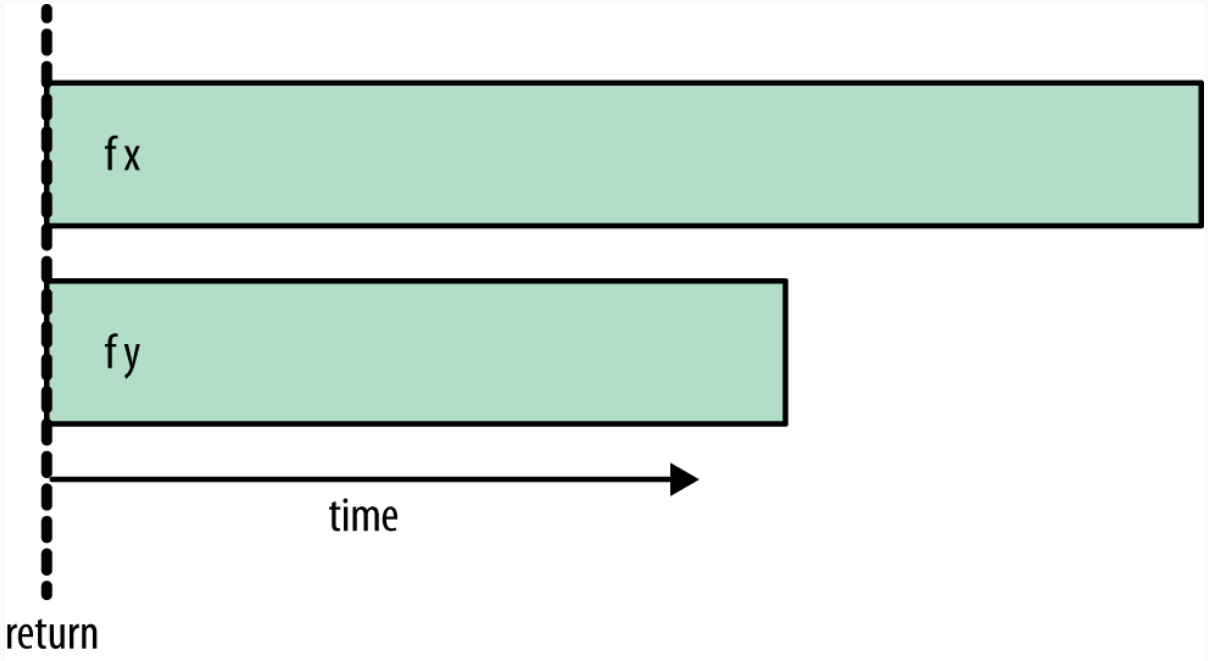


Figura 2: rpar-rpar

rpar-rseq

Definindo a expressão **fparseq** e alterando a função **main** para utilizá-la:

```
fparseq :: Eval (Integer, Integer)
fparseq = do a <- rpar (fib 41)
             b <- rseq (fib 40)
             return (a,b)
```

Ao medir o tempo que leva para terminar a tarefa `(fib 40)`

```
Tempo para executar fparpar: 5.979055s
Resultado: (165580141,102334155)
Tempo apos imprimir resultado: 9.834702s
```

Agora `runEval` aguarda a finalização do processamento de *b* antes de liberar para outros processos.

A estratégia `rpar-rseq` aguarda a finalização do processamento `seq`:

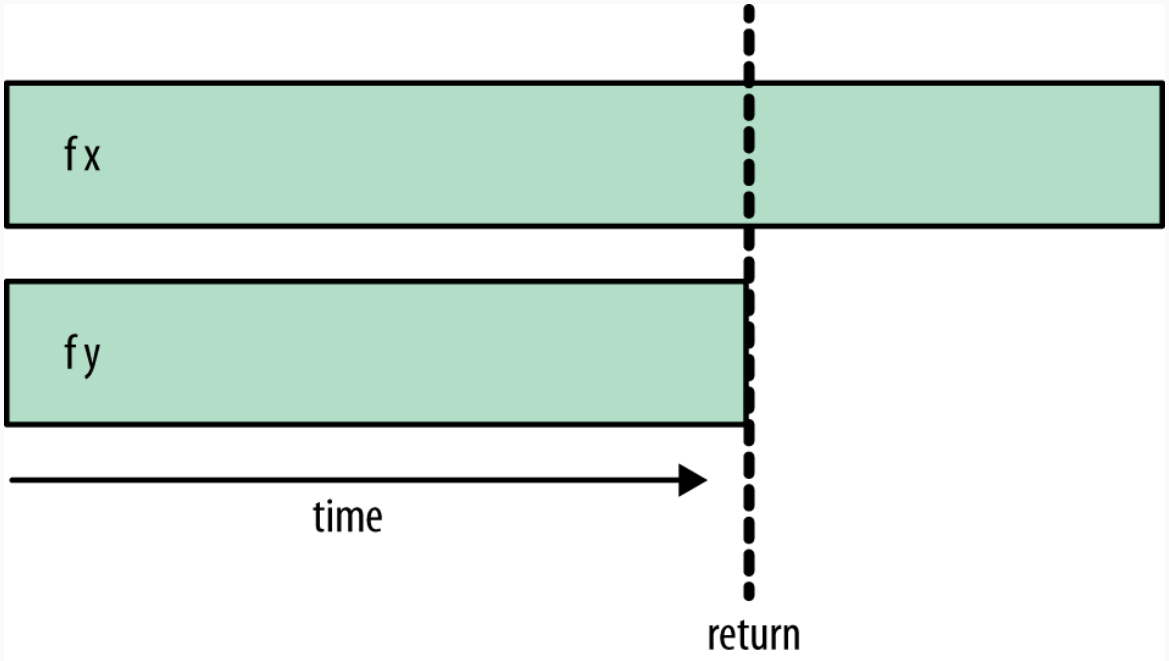


Figura 3: rpar-rseq

rpar-rpar-rseq-rseq

Finalmente podemos fazer:

```
fparparseq :: Eval (Integer, Integer)
fparparseq = do a <- rpar (fib 41)
                b <- rpar (fib 40)
                rseq a
                rseq b
                return (a,b)
```

rpar-rpar-rseq-rseq

E o resultado da execução com *N2* é:

```
Tempo para executar fparpar: 10.094192s
Resultado: (165580141,102334155)
Tempo apos imprimir resultado: 10.094287s
```

Agora `runEval` aguarda o resultado de todos os threads antes de retornar:

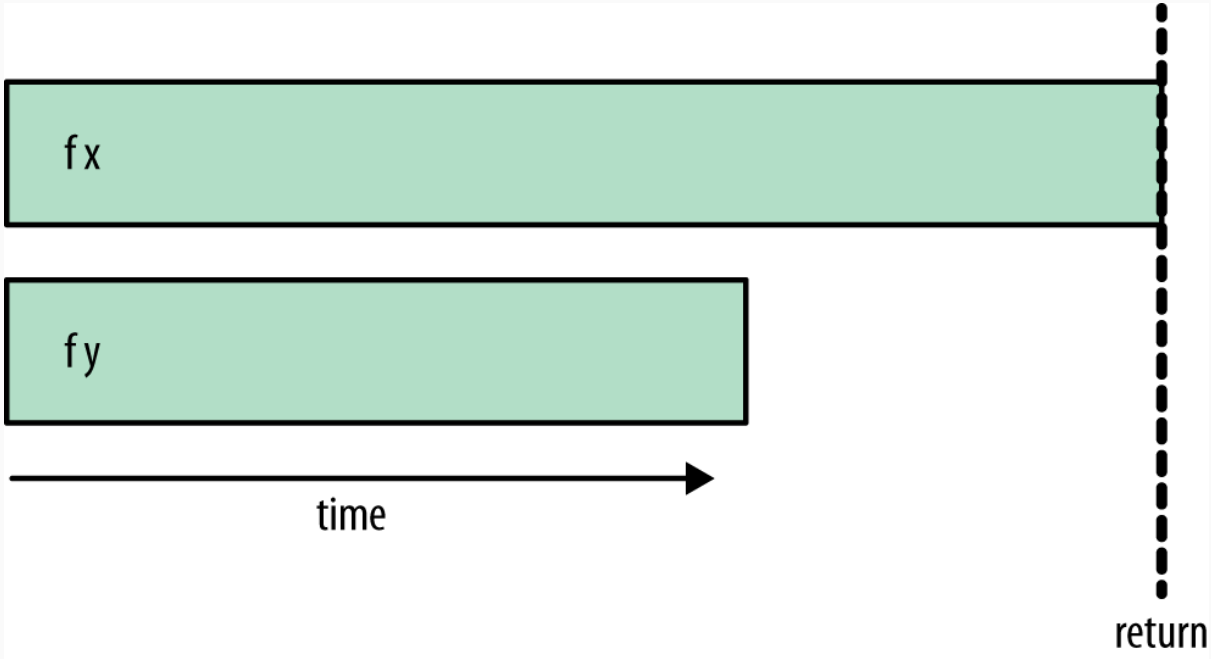


Figura 4: rpar-rpar-rseq-rseq

Escolhendo a estratégia

A escolha da combinação de estratégias depende muito do algoritmo que está sendo implementado.

Se pretendemos gerar mais paralelismo e não dependemos dos resultados anteriores, `rpar-rpar` faz sentido como estratégia.

Porém, se já geramos todo o paralelismo desejado e precisamos aguardar o resultado `rpar-rpar-rseq-rseq` pode ser a melhor estratégia.

Estratégias de Avaliação

Estratégias de Avaliação

A biblioteca `Control.Parallel.Strategies` define também o tipo:

```
type Strategies a = a -> Eval a
```

Uma função que recebe um tipo `a` e retorna uma estratégia para paralelizar esse tipo.

Estratégias de Avaliação

A ideia desse tipo é permitir a abstração de estratégias de paralelismo para tipos de dados, seguindo o exemplo anterior, poderíamos definir:

```
-- :: (a,b) -> Eval (a,b)
parPair :: Strategy (a,b)
parPair (a,b) = do a' <- rpar a
                   b' <- rpar b
                   return (a',b')
```

Estratégias de Avaliação

Dessa forma podemos escrever:

```
runEval (parPair (fib 41, fib 40))
```

Mas seria interessante separar a sintaxe da parte sequencial `(fib 41, fib 40)` da parte paralela. Isso facilitaria a manutenção do código.

Estratégias de Avaliação

Podemos então definir:

```
using :: a -> Strategy a -> a
x `using` s = runEval (s x)
```

Estratégias de Avaliação

Com isso nosso código se torna:

```
(fib 41, fib 40) `using` parPair
```

Dessa forma, uma vez que meu programa sequencial está feito, posso adicionar paralelismo sem me preocupar em quebrar o programa.

Estratégias parametrizadas

A nossa função **parPair** ainda é restritiva em relação a estratégia adotada, devemos criar outras funções similares para adotar outras estratégias. Uma generalização pode ser escrita como:

```
evalPair :: Strategy a -> Strategy b -> Strategy (a,b)
evalPair sa sb (a,b) = do a' <- sa a
                          b' <- sb b
                          return (a',b')
```

Estratégias parametrizadas

Nossa função `parPair` pode ser reescrita como:

```
parPair :: Strategy (a,b)
parPair = evalPair rpar rpar
```

Estratégias parametrizadas

Ainda temos uma restrição, pois ou utilizamos **`rpar`** ou **`rseq`**. Além disso ambas avaliam a expressão para a WHNF. Para resolver esses problemas podemos utilizar as funções:

```haskell
rdeepseq :: NFData a => Strategy a
rdeepseq x = rseq (force x)

-- funcao de Parallel.Strategies
rparWith :: Strategy a -> Strategy a
```

Estratégias parametrizadas

Dessa forma podemos fazer:

```
parPair :: Strategy a -> Strategy b -> Strategy (a,b)
parPair sa sb = evalPair (rparWith sa) (rparWith sb)
```

Estratégias parametrizadas

E podemos garantir uma estratégia paralela que avalia a estrutura por completo:

```
(fib 41, fib 40) `using` parPair rdeepseq rdeepseq
```

Estratégia para listas

Como as listas representam uma estrutura importante no Haskell, a biblioteca já vem com a estratégia `parList` de tal forma que podemos fazer:

```
map f xs `using` parList rseq
```

Estratégia para listas

Essa é justamente a definição de `parMap`:

```
parMap :: (a -> b) -> [a] -> [b]
parMap f xs = map f xs `using` parList rseq
```

Exemplo: média

Exemplo: média

Vamos definir a seguinte função que calcula a média dos valores de cada linha de uma matriz:

```
mean :: [[Double]] -> [Double]
mean xss = map mean' xss  `using` parList rseq
  where
    mean' xs = (sum xs) / (fromIntegral $ length xs)
```

Cada elemento de *xss* vai ser potencialmente avaliado em paralelo.

Exemplo: média

Compilando e executando esse código enquanto medimos o tempo de execução, temos:

```
Total   time    1.381s  (  1.255s elapsed)
```

O primeiro valor é a soma do tempo de máquina de cada thread, o segundo valor é o tempo total real de execução do programa.

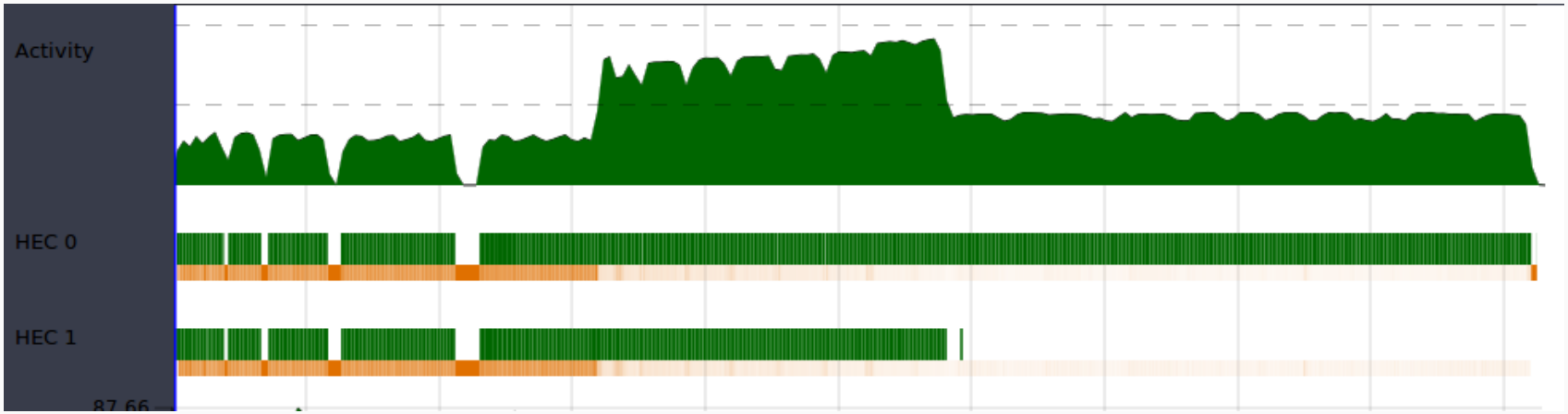
Exemplo: média

O que houve?

```
Total   time    1.381s  (  1.255s elapsed)
```

Exemplo: média

Os gráficos em verde mostram o trabalho feito por cada *core* do computador:



Por que um core fez o dobro do trabalho?

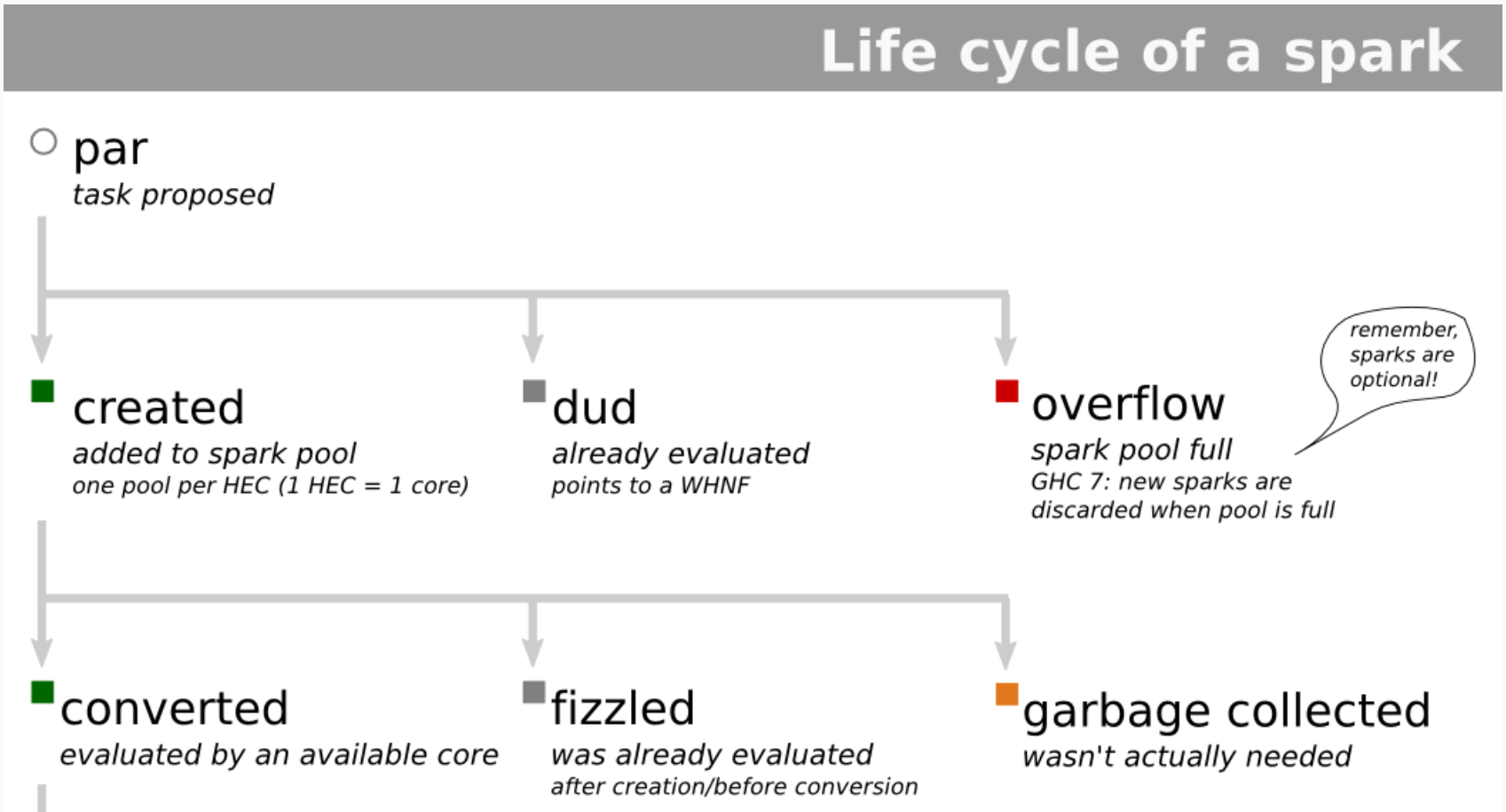
Vida de um spark

No Haskell o paralelismo é feito através da criação de **sparks**, um spark é uma promessa de algo a ser computado e que pode ser computado em paralelo.

Cada elemento da lista gera um spark, esses sparks são inseridos em um *pool* que alimenta os processos paralelos.

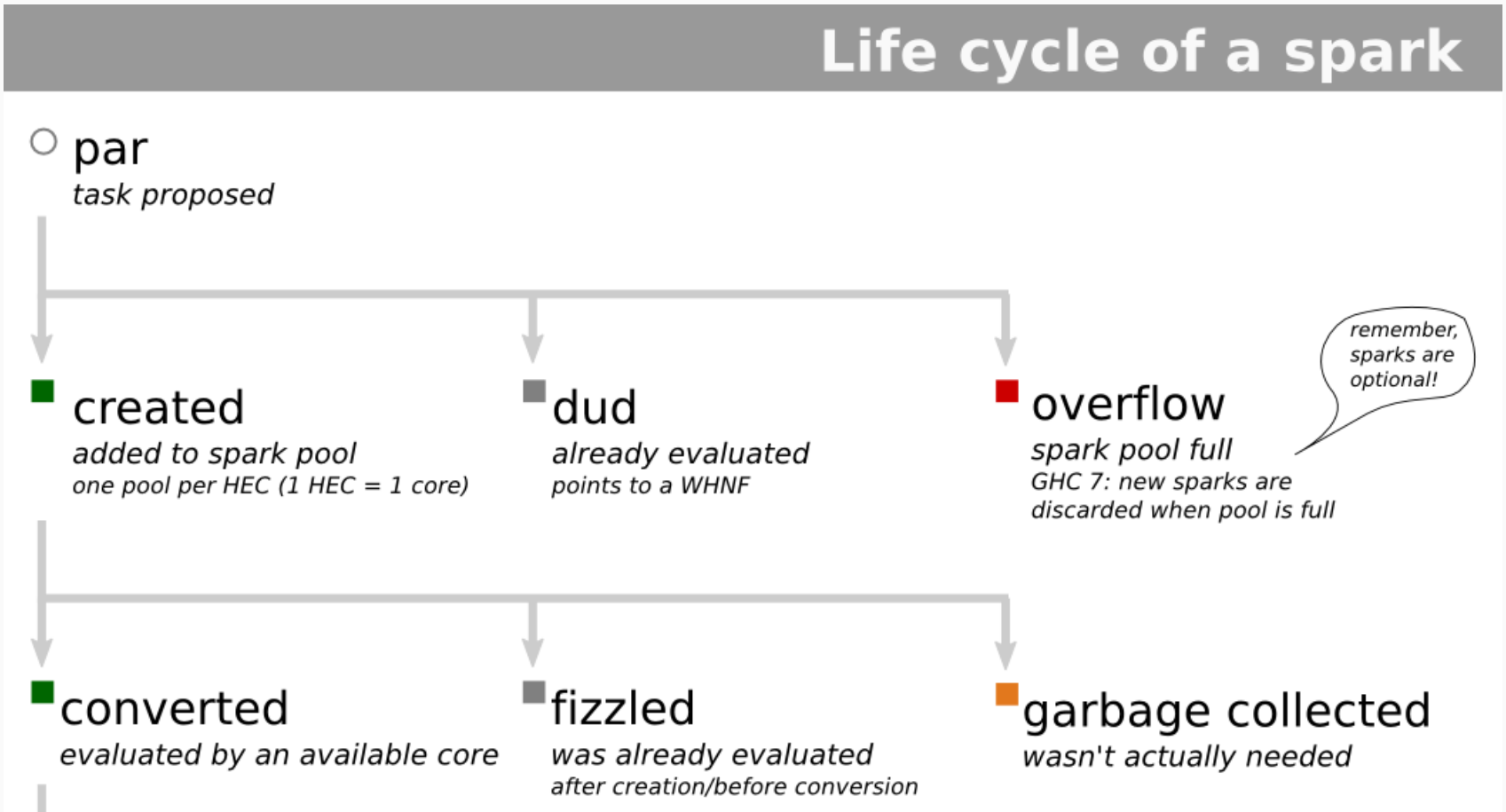
Vida de um spark

Cada elemento que é passado para a função `rpar` cria um spark e é inserido no *pool*. Quando um processo pega esse spark do pool, ele é convertido em um processo e então é executado:



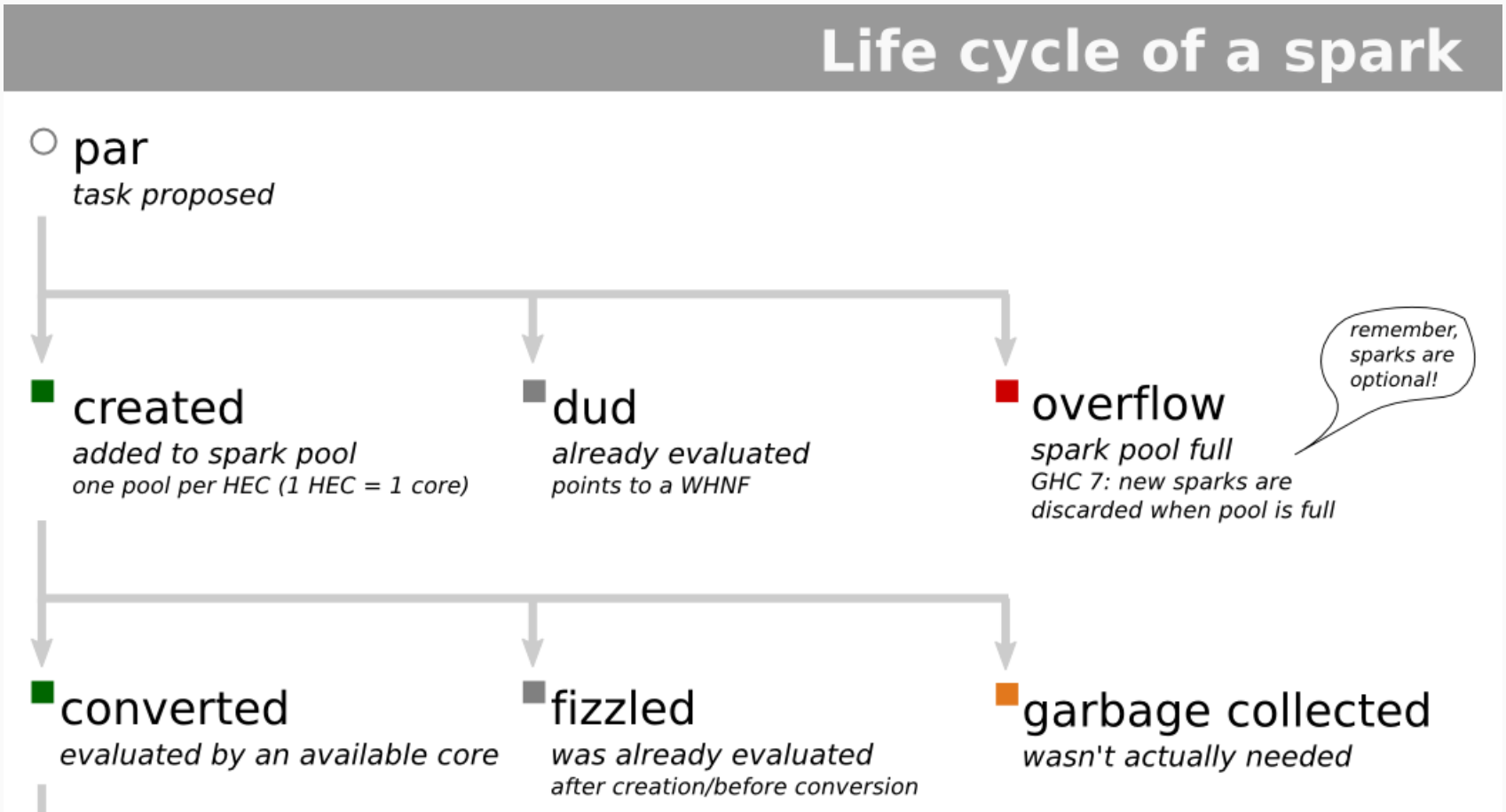
Vida de um spark

No momento da criação, antes de criar o spark, antes é verificado se a expressão não foi avaliada anteriormente. Caso tenha sido, ela vira um *dud* e aponta para essa avaliação prévia.



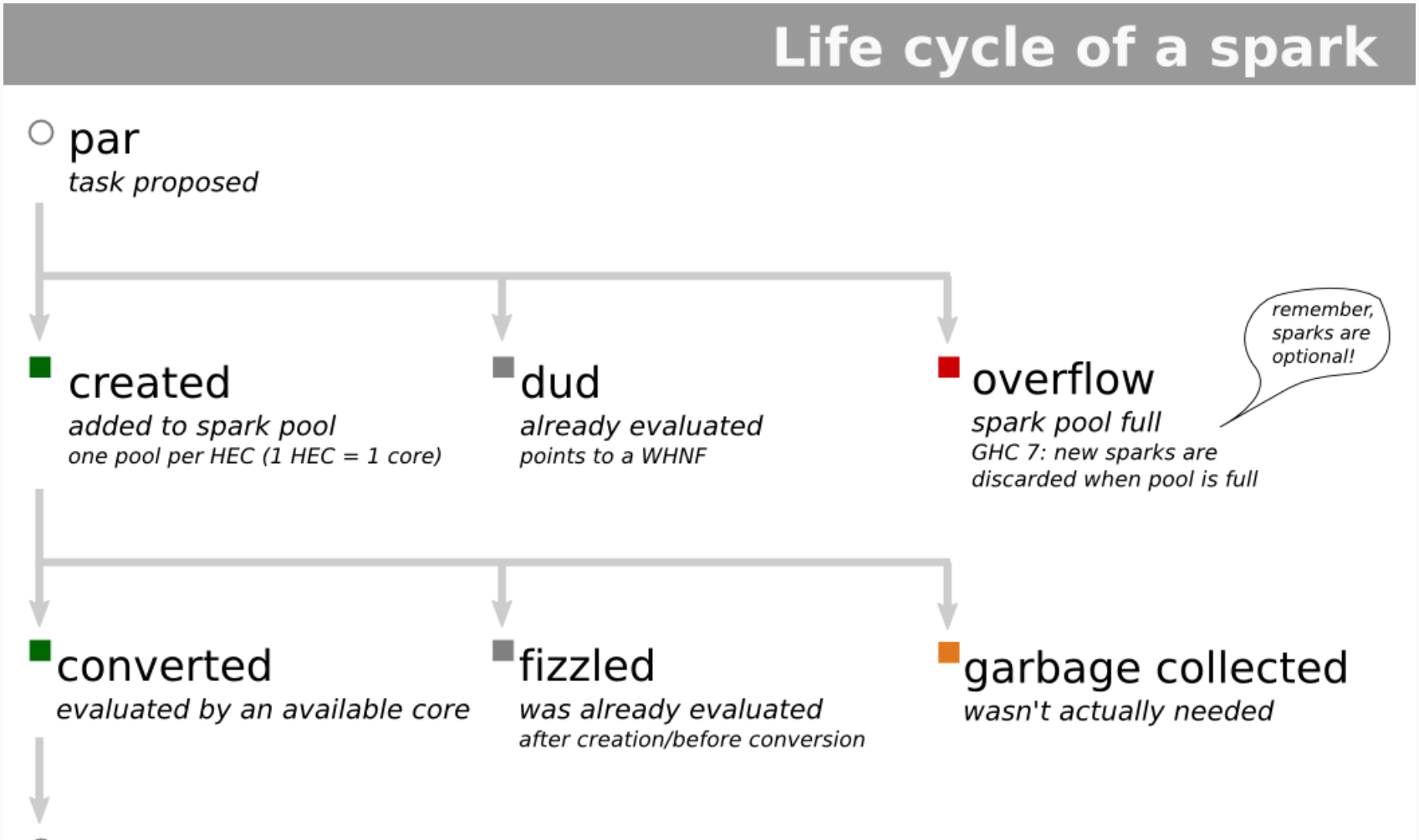
Vida de um spark

Se o pool estiver cheio no momento, ela retorna o status *overflow* e não cria o spark, simplesmente avalia a expressão no processo principal.



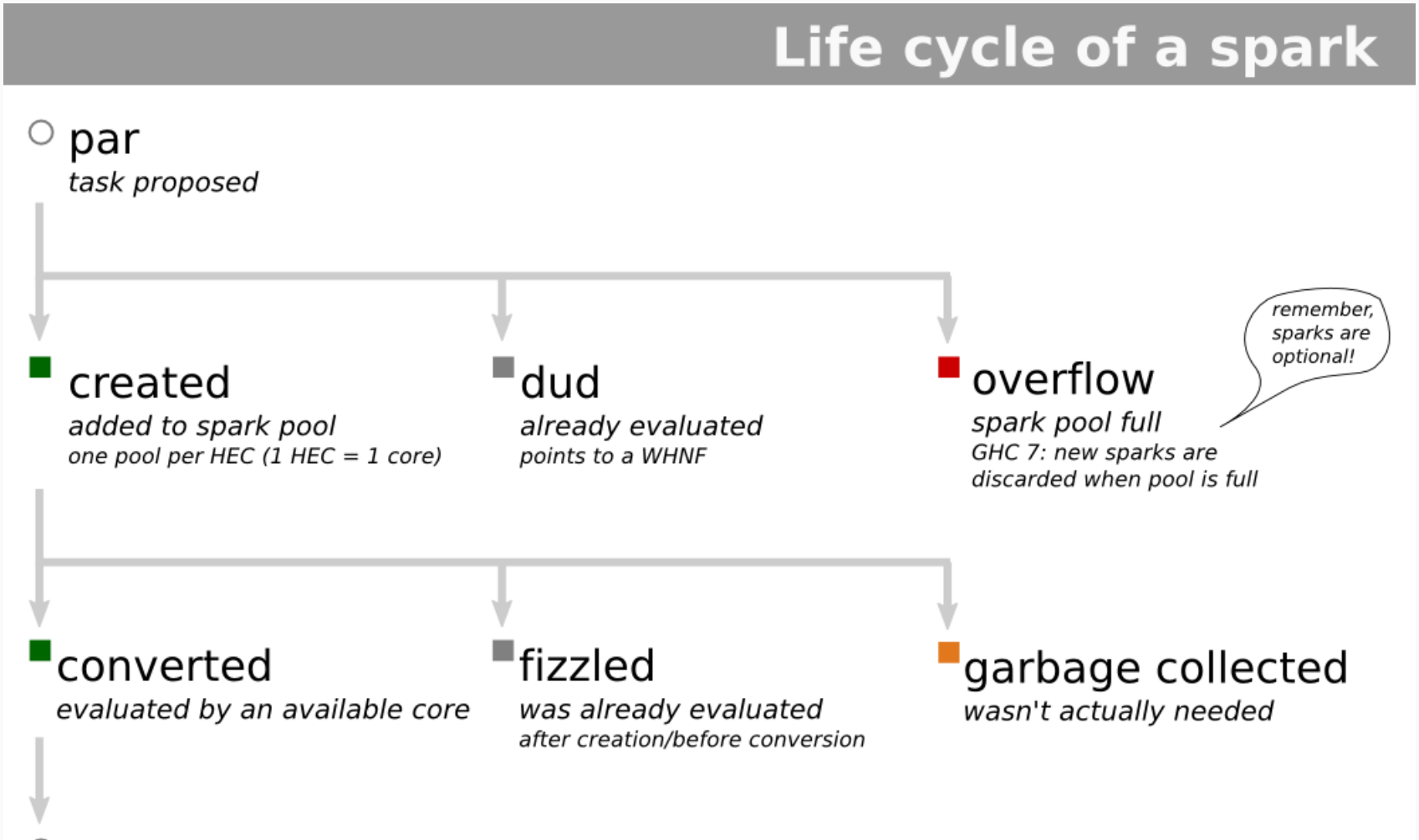
Vida de um spark

Se no momento de ser retirado do pool ele já tiver sido avaliado em outro momento, o spark retorna status *fizzled*, similar ao *dud*.



Vida de um spark

Finalmente, se essa expressão nunca for requisitada, então ela é desalocada da memória pelo *garbage collector*.



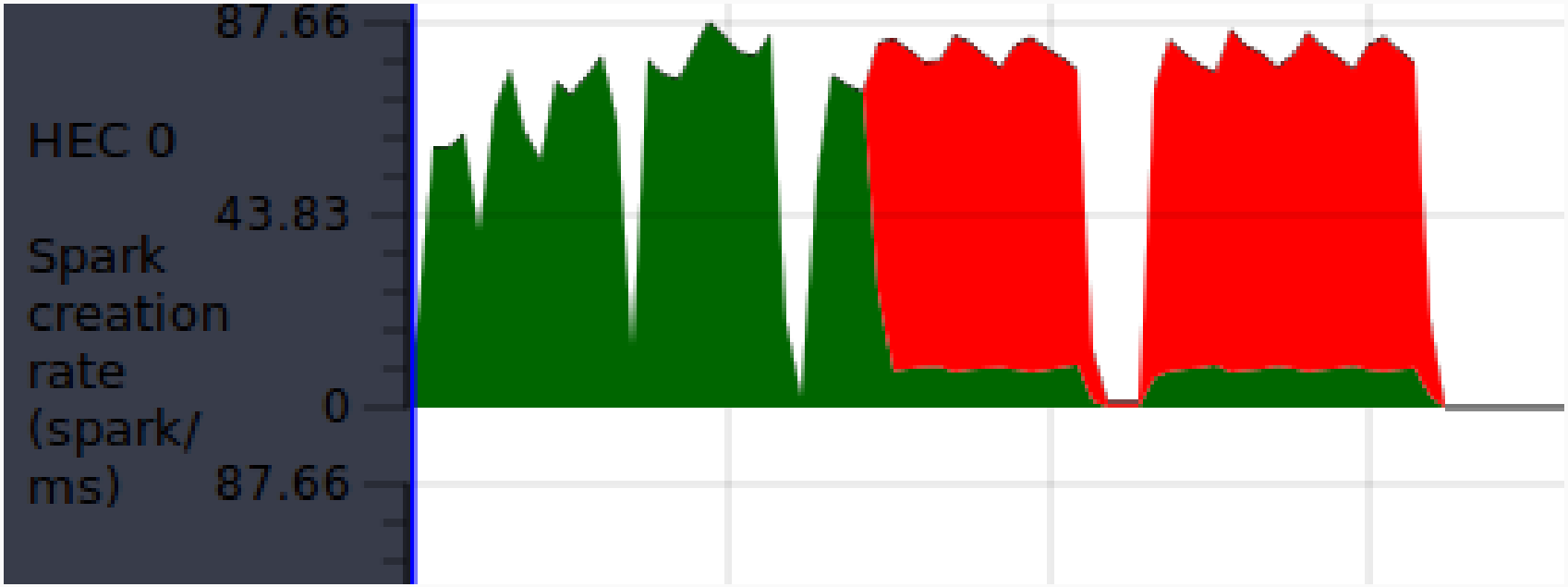
Vida de um spark

Sinais de problemas:

- Poucos sparks, pode ser paralelizado ainda mais
- Muitos sparks, paralelizando demais
- Muitos duds e fizzles, estratégia não otimizada.

Exemplo: média

Voltando ao nosso exemplo, se olharmos para a criação de sparks, percebemos que ocorreu *overflow* (parte vermelha), ou seja, criamos muitos sparks em um tempo muito curto:



Exemplo: média

Vamos tirar a estratégia...

```haskell
mean :: [[Double]] -> [Double]
mean xss = map mean' xss
  where
    mean' xs = (sum xs) / (fromIntegral $ length xs)
```

Exemplo: média

E criar uma nova função que aplica a função `mean` sequencial em pedaços de nossa matriz:

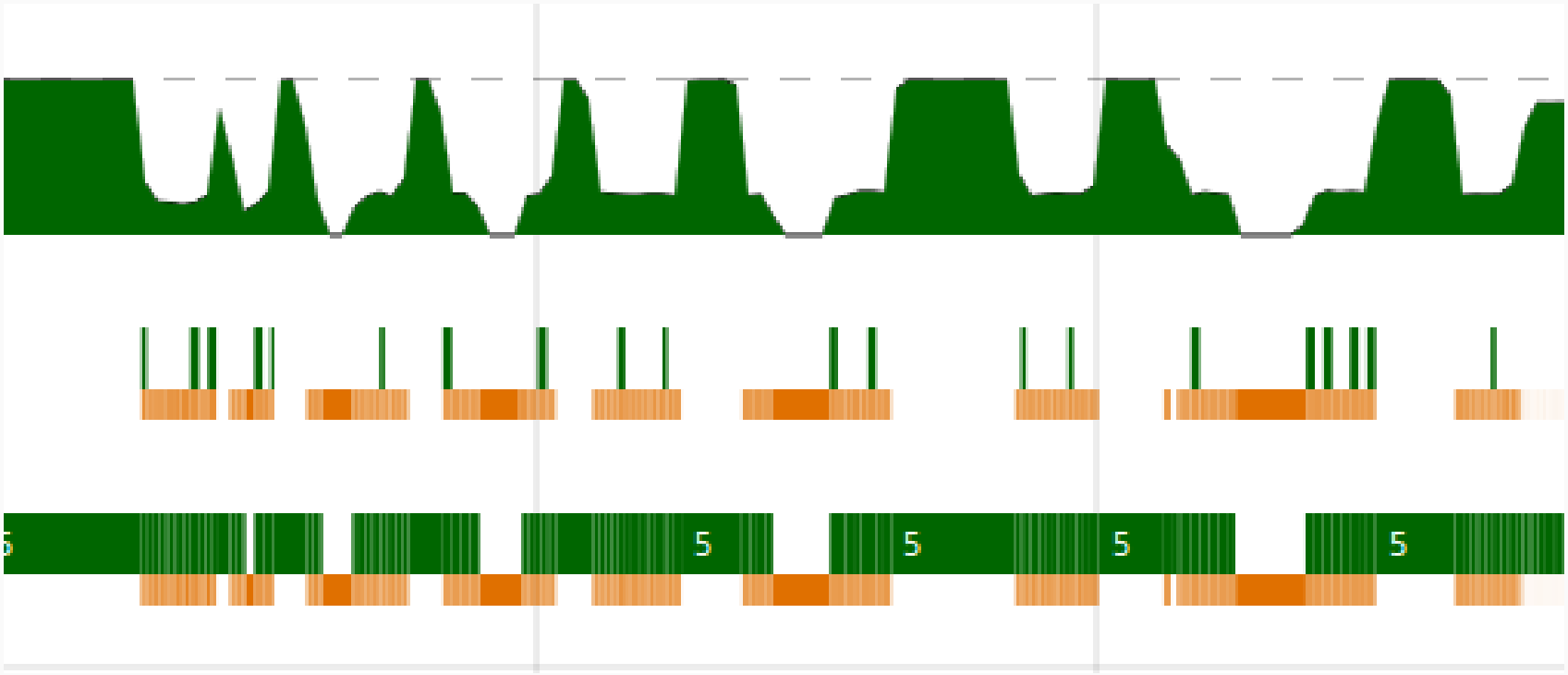
```
meanPar :: [[Double]] -> [Double]
meanPar xss = concat medias
  where
    medias = map mean chunks `using` parList rseq
    chunks = chunksOf 1000 xss
```

Agora criaremos menos sparks, pois cada spark vai cuidar de 1000 elementos de *xss*.

Exemplo: média

O resultado:

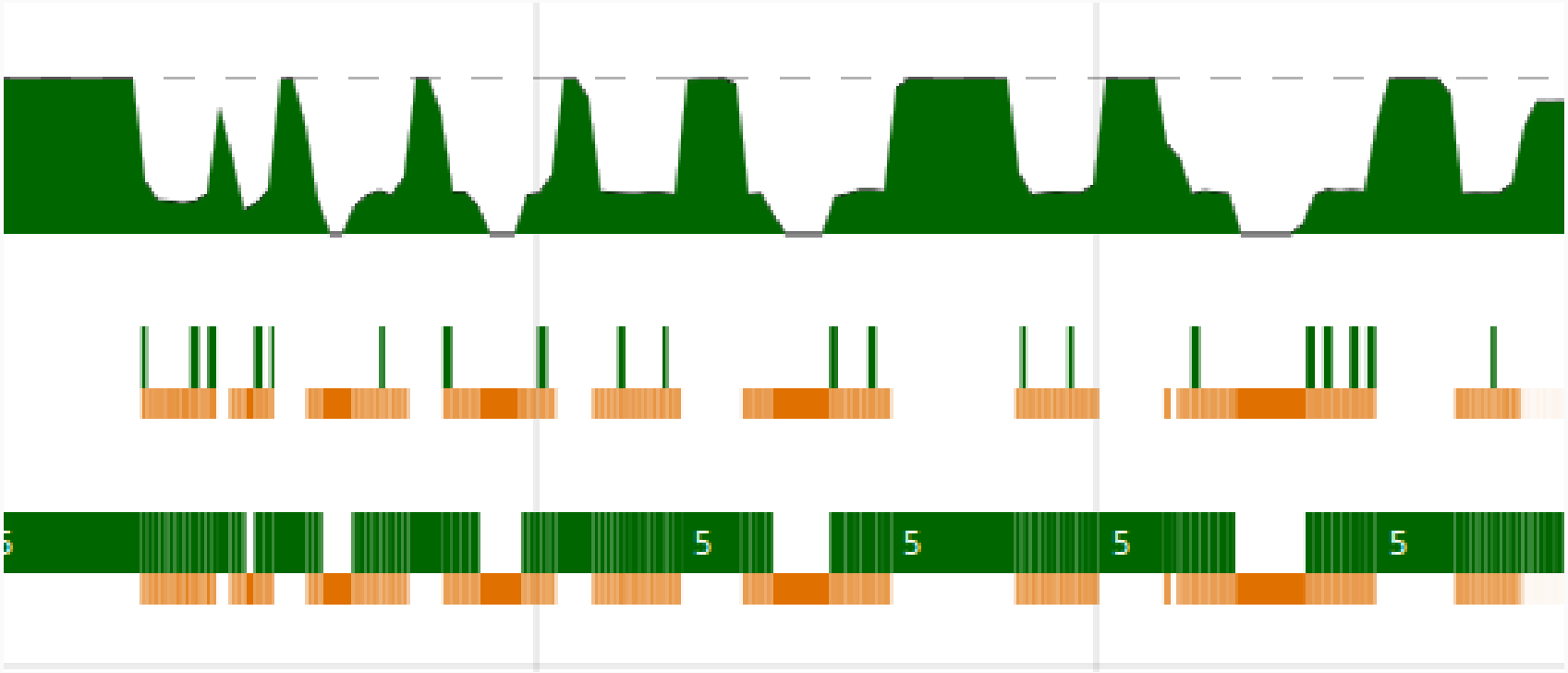
```
Total   time    1.289s  (  1.215s elapsed)
```



Não tem mais overflow! Mas...

Exemplo: média

A função `mean` é aplicada em paralelo até encontrar a WHNF, ou seja, apenas a promessa de calcular a média de cada linha!



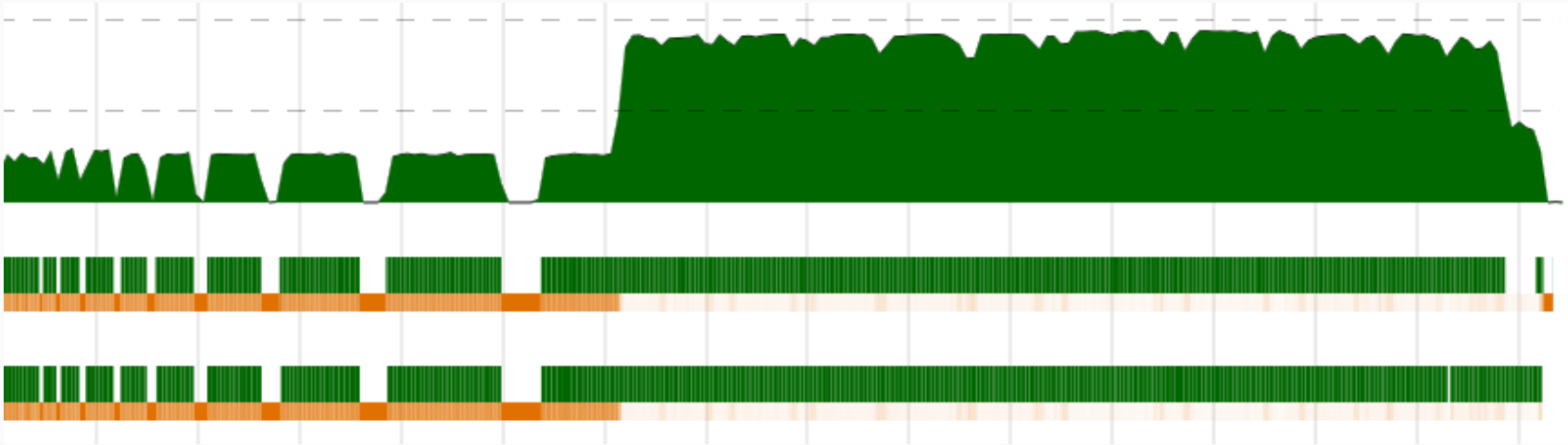
Exemplo: média

Vamos usar a estratégia *rdeepseq*.

```
meanPar :: [[Double]] -> [Double]
meanPar xss = concat medias
  where
    medias = map mean chunks `using` parList rdeepseq
    chunks = chunksOf 1000 xss
```

Exemplo: média

Total time 1.303s (0.749s elapsed)



:)

MapReduce com Eval Monad

Vamos construir uma simulação do processo de MapReduce utilizando o Eval Monad como base de paralelismo.

RDD

No Apache Spark, a base de dados distribuída é abstraída através do tipo **RDD**. Um RDD é um container que armazena dados de qualquer tipo distribuído entre os nós disponíveis.

RDD

A lista do Haskell é um bom candidato para essa abstração, sendo que cada elemento contém os dados de um nó:

```
type RDD a = [a]
```

Um exemplo do funcionamento do MapReduce:

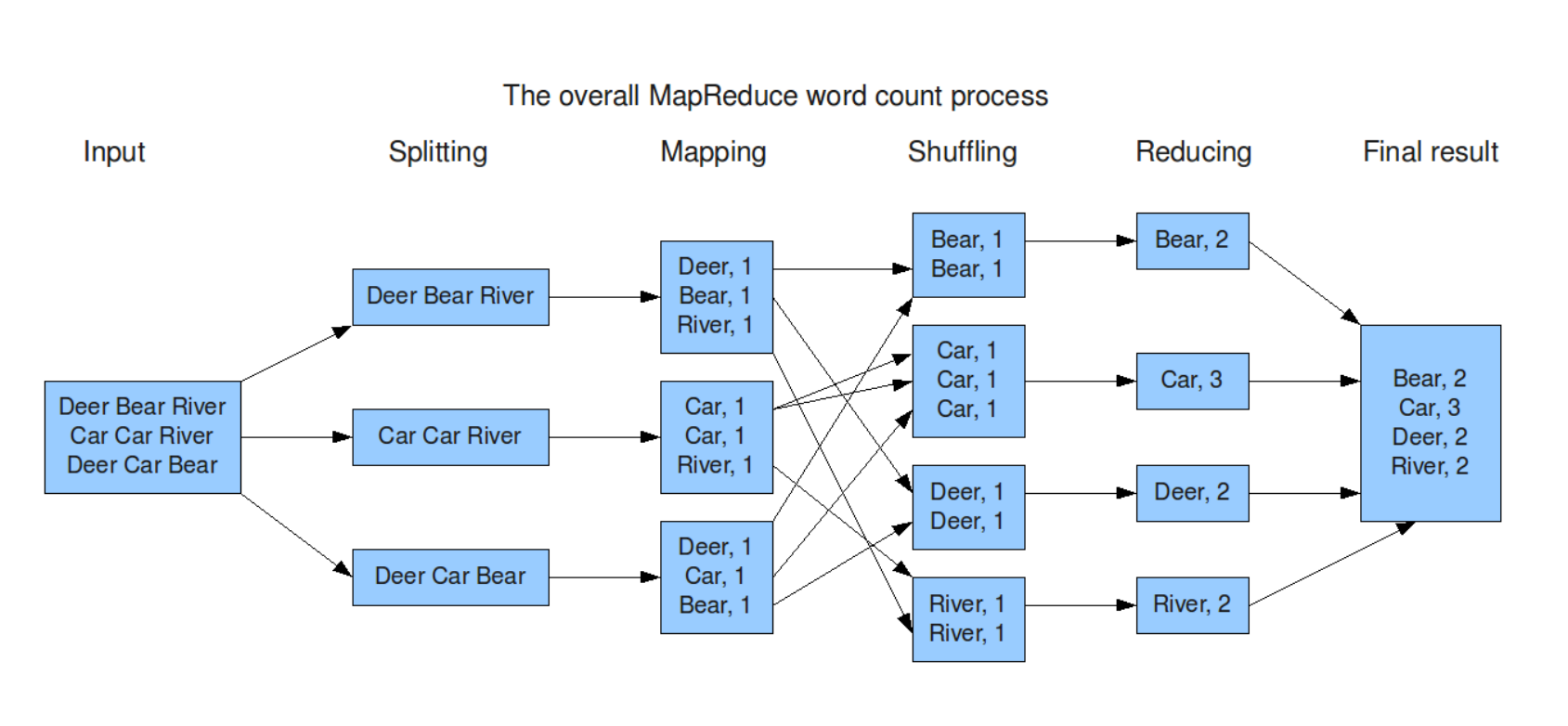


Figura 10: Fonte: UT Dallas

Mapper

Sendo cada pedaço do RDD contendo uma coleção identificada de registros, a assinatura de um `mapper` deve ser:

```
mapper :: (k1, v1) -> [(k2, v2)]
```

No nosso exemplo, a função `mapper` recebe uma linha de texto junto de um *id* e retorna uma lista de tuplas tendo uma palavra como chave e o valor 1 como valor.

Shuffling

Em seguida, o processo de shuffling organiza os dados de tal forma que cada pedaço da base contenha uma chave específica e uma lista de valores dessa chave.

Shuffling

A função **shuffler** recebe uma RDD e retorna uma outra RDD. O tipo da RDD é uma lista de chave-valor, gerada pelo **mapper**:

```
shuffler :: (Eq a, Ord a) => RDD [(a,t)] -> RDD (a,[t])
shuffler = fmap shuffle . sortAndgroup . concat
  where shuffle ((k,v):kvs) = (k, v : fmap snd kvs)
```

Shuffling

A função **sortAndgroup** se encarrega de ordenar a RDD pela chave e agrupar o conteúdo:

```
sortAndgroup :: (Eq a, Ord a) => [(a,t)] -> [[(a,t)]]
sortAndgroup = (groupByKey . sortByKey)

sortByKey :: (Ord a) => [(a,t)] -> [(a,t)]
sortByKey = sortBy (comparing fst)
```

Shuffling

A função **sortAndgroup** se encarrega de ordenar a RDD pela chave e agrupar o conteúdo:

```
groupByKey :: (Ord a) => [(a, t)] -> [[(a, t)]]
groupByKey = groupBy agrupaTupla

agrupaTupla :: (Eq a) => (a, t0) -> (a, t0) -> Bool
agrupaTupla (a1, b1) (a2, b2) = a1==a2
```

Reducer

A função **`reducer`** deve receber uma tupla chave-lista de valores `(k1, [v1])` e retornar uma nova tupla `(k2, v2)`:

```
reducer :: (k1, [v1]) -> (k2, v2)
```

Executando a sequência Mapper e Reducer

Observando a assinatura do `mapper`, temos uma função que recebe uma tupla e transforma em uma lista de tuplas. Queremos aplicar essa função em uma RDD de lista de tuplas. A execucão de um `mapper` se torna:

```
runMapper :: ((k1, v1) -> [(k2, v2)]) -> RDD [(k1,v1)]
          -> RDD [(k2, v2)]
runMapper mapper rdd = join (parmap (fmap mapper) rdd)
```

Notem que a assinatura sugere que temos um Monad!

Executando a sequência Mapper e Reducer

Criando uma instância de Monad para nosso tipo RDD, poderíamos reescrever como:

```
runMapper :: ((k1, v1) -> [(k2, v2)]) -> RDD [(k1,v1)]
          -> RDD [(k2, v2)]
runMapper mapper rdd = rdd >>= fmap mapper
```

em que o operador `>>=` aplica a função `parmap`.

Executando a sequência Mapper e Reducer

De forma similar, nosso `runReducer` simplesmente aplicar o `reducer` em paralelo:

```
runReducer :: ((k1, [v1]) -> (k2, v2)) -> RDD (k1, [v1])
runReducer reducer = parmap reducer
```

Executando a sequência Mapper e Reducer

Finalmente, um *job* de *Mapper* e *Reducer* pode ser implementado pela função:

```
mrJob :: Ord k2 => Mapper -> Reducer -> RDD [(k1, v1)]
      -> RDD (k2, v2)
mrJob mapper reducer rdd = runMapper mapper rdd
                         |> shuffler
                         |> runReducer reducer
```

Exemplo canônico

```
main = do
  let text  = [["ola mundo", "mundo cruel"],
               ["haskell ola"]]
      rdd   = fmap (zip [1..]) text

  print $ mrJob mapperTok reducerTok rdd
```

Para saber mais

- Livro gratuito: Parallel and Concurrent Programming in Haskell
- Curso de Big Data: https://folivetti.github.io/courses/BigData/
- Curso de Paradigmas de Programação: https://folivetti.github.io/courses/ParadigmasProgramacao/

Slides e Código do Curso

https://github.com/folivetti/ERAD-SP-2019