Programação Paralela e Concorrente em Haskell

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

Programação Paralela e Concorrente em Haskell

Paralelismo vs Concorrência

UFABC

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

Paralelismo vs Concorrência

UFABC

- 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

UFABC

Imagine uma lanchonete servindo café. Nós podemos ter:

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

Paralelismo vs Concorrência

UFABC

- 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.

- 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

UFABC

- 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

UFABC

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

- 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

UFABC

- 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

UFABC

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

Quero evitar a fadiga

UFABC

```
1  Prelude> x = 5 + 10
2  Prelude> :sprint x
3  x = _
4  Prelude> x
5  15
6  Prelude> :sprint x
7  x = 15
```

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

Quero evitar a fadiga

UFABC

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

Quero evitar a fadiga

UFABC

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

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

Eu quero agora!

UFABC

```
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

UFABC

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

Exercício 1

UFABC

O que terá sido avaliado em *lista* após a execução do seguinte código?

```haskell
f x = 2*x
g x
  | even x = x + 1
  | otherwise = f x

lista  = [ (x, g x, f x) | x <- [1..], even x ]
lista' = map snd lista
sublista = take 4 lista'

print sublista
```

Weak Head Normal Form (WHNF)

UFABC

Ao fazer:

```
1  > z = (2, 3)
2  > :sprint z
3  z = _
4  > z `seq` ()
5  ()
6  > :sprint z
7  z = (_,_)
```

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

UFABC

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

```
1  > import Control.DeepSeq
2  > z = (2,3)
3  > force z
4  > :sprint z
5  z = (2,3)
```

Eval Monad

UFABC

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

```haskell
data Eval a = ...

instance Monad Eval where ...

runEval :: Eval a -> a

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

UFABC

- A função `rpar` indica que *meu argumento pode ser executado em paralelo*.
- 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

UFABC

- Finalmente, a função `runEval` executa uma expressão (em paralelo ou não) e retorna o resultado dessa computação.
- Note que o Monad Eval é puro e pode ser utilizado fora de funções com IO.

Exemplo

Crie um projeto chamado **paralelo**:

```
1 stack new paralelo simple
2 stack setup
```

*Slides apenas para replicação em lab.

Exemplo

UFABC

- Edite o arquivo `paralelo.cabal` e na linha `build-depends` acrescente as bibliotecas `parallel, time`.
- Na linha anterior a `hs-source-dirs` acrescente a linha `ghc-options: -threaded -rtsopts -with-rtsopts=-N -eventlog`

*Slides apenas para replicação em lab.

Exemplo

UFABC

No arquivo *Main.hs* acrescente:

```
1 import Control.Parallel.Strategies
2 import Control.Exception
3 import Data.Time.Clock
```

*Slides apenas para replicação em lab.

UFABC

Considere a implementação ingênua de fibonacci:

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

UFABC

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

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

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

UFABC

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

Exemplo

UFABC

Altere a função `main` para:

```haskell
main :: IO ()
main = do
  t0 <- getCurrentTime
  -- evaluate força avaliação para WHNF
  r  <- evaluate (runEval fparpar)
  t1 <- getCurrentTime
  print (diffUTCTime t1 t0)
  print r  -- vamos esperar o resultado terminar
  t2 <- getCurrentTime
  print (diffUTCTime t2 t0)
```

Compile com `stack build --profile` e execute com:

```
$ stack exec paralelo --RTS -- +RTS -N1
```

*Slides apenas para replicação em lab.

Explicando os parâmetros

UFABC

- `-threaded`: compile com suporte a multithreading
- `-eventlog`: permite criar um log do uso de threads
- `-rtsopts`: embute opções no seu programa
- `+RTS`: flag para indicar opções embutidas
- `-Nx`: quantas threads usar
- `-s`: estatísticas de execução
- `-ls`: gera log para o threadscope

UFABC

Para o parâmetro *N1* a saída da execução retornará:

```
0.000002s
(165580141,102334155)
15.691738s
```

Para o parâmetro *N2* a saída da execução retornará:

```
0.000002s
(165580141,102334155)
9.996815s
```

- 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.

UFABC

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

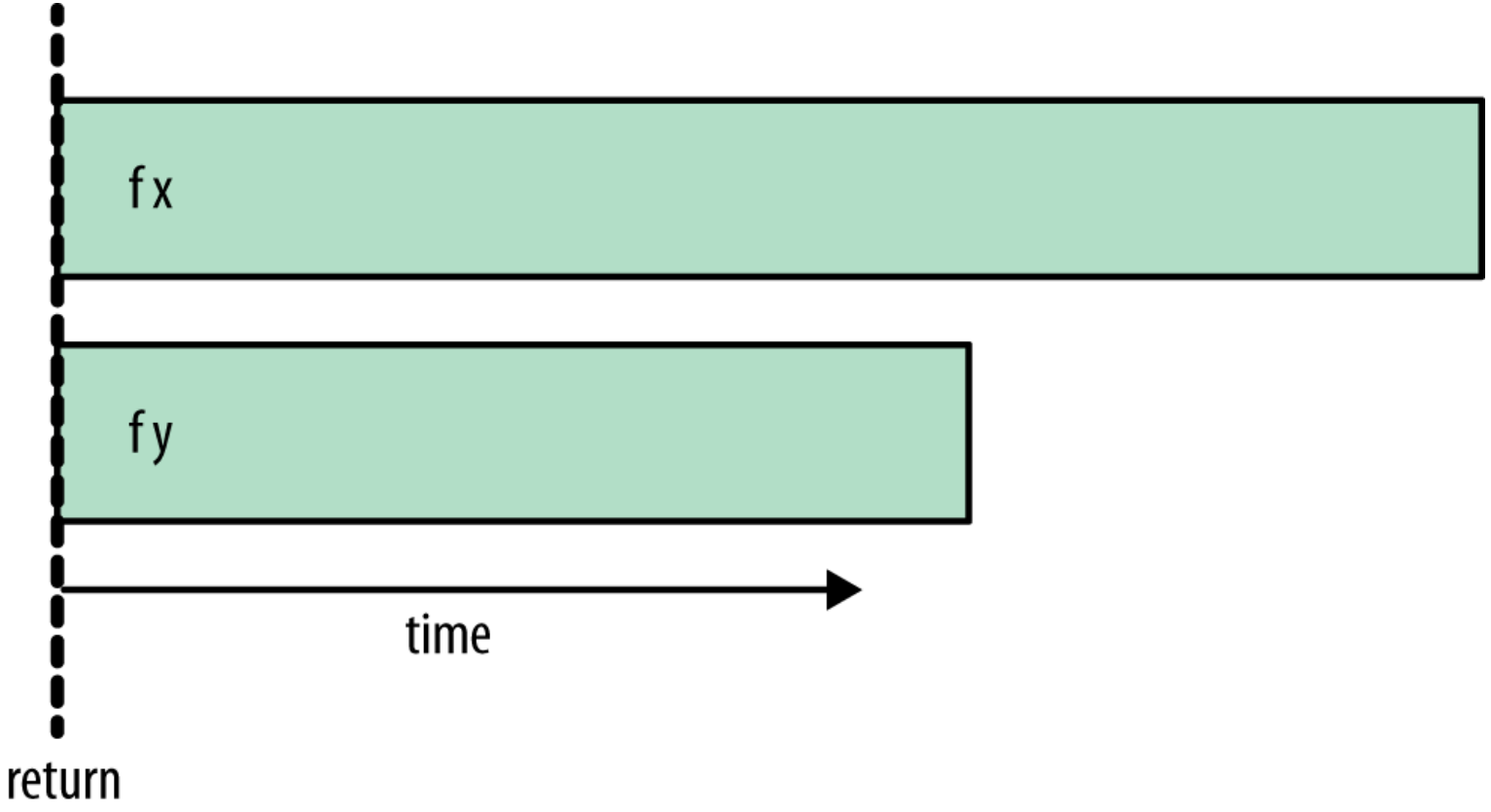


Figura 1: rpar-rpar

UFABC

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

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

UFABC

- Temos como resultado para *N2*:

```
5.979055s
(165580141,102334155)
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`:

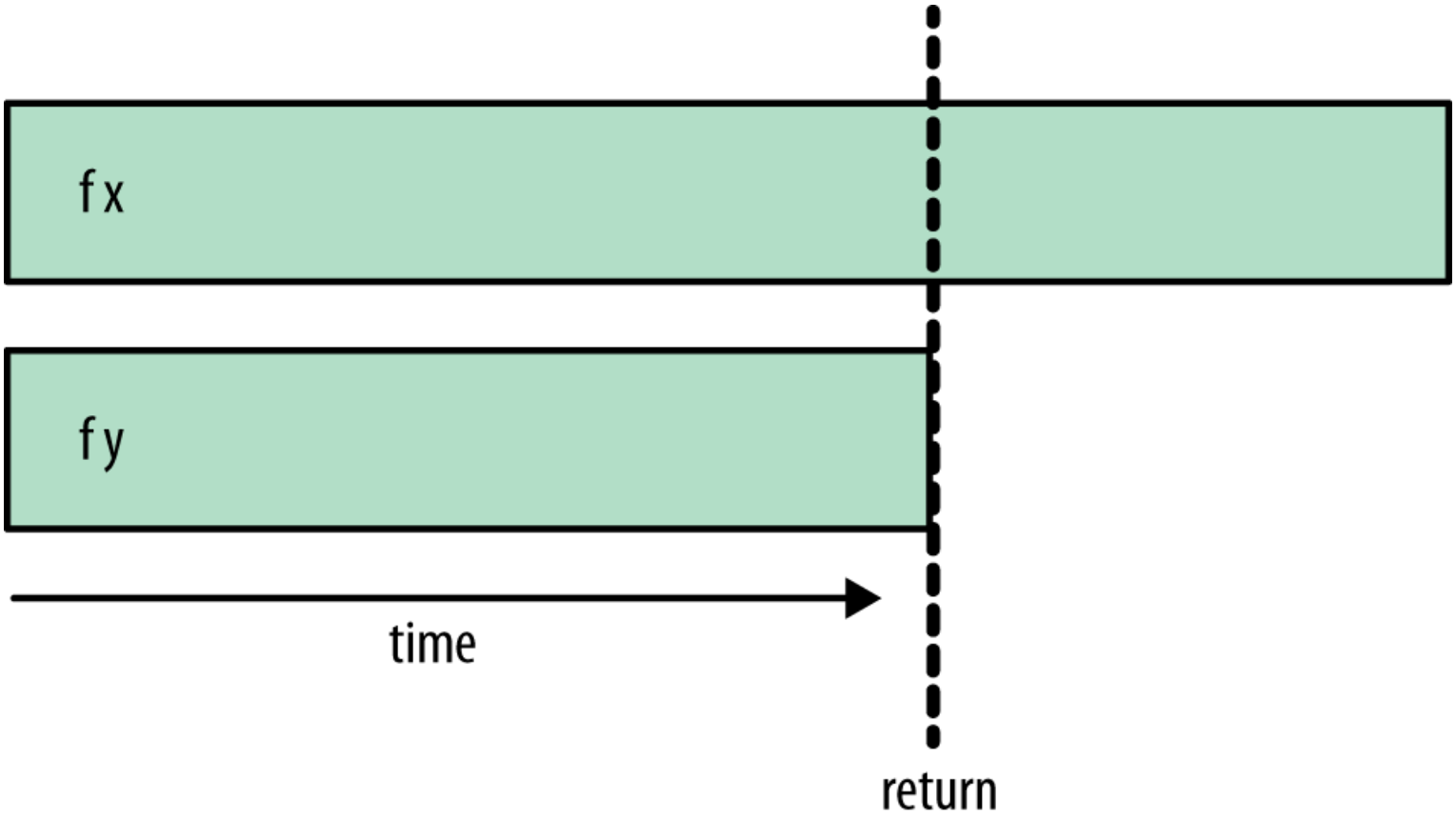


Figura 2: rpar-rseq

UFABC

Finalmente podemos fazer:

```
1  fparparseq :: Eval (Integer, Integer)
2  fparparseq = do a <- rpar (fib 41)
3                  b <- rpar (fib 40)
4                  rseq a
5                  rseq b
6                  return (a,b)
```

UFABC

E o resultado da execução com *N2* é:

```
(165580141,102334155)
10.094287s
```

UFABC

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

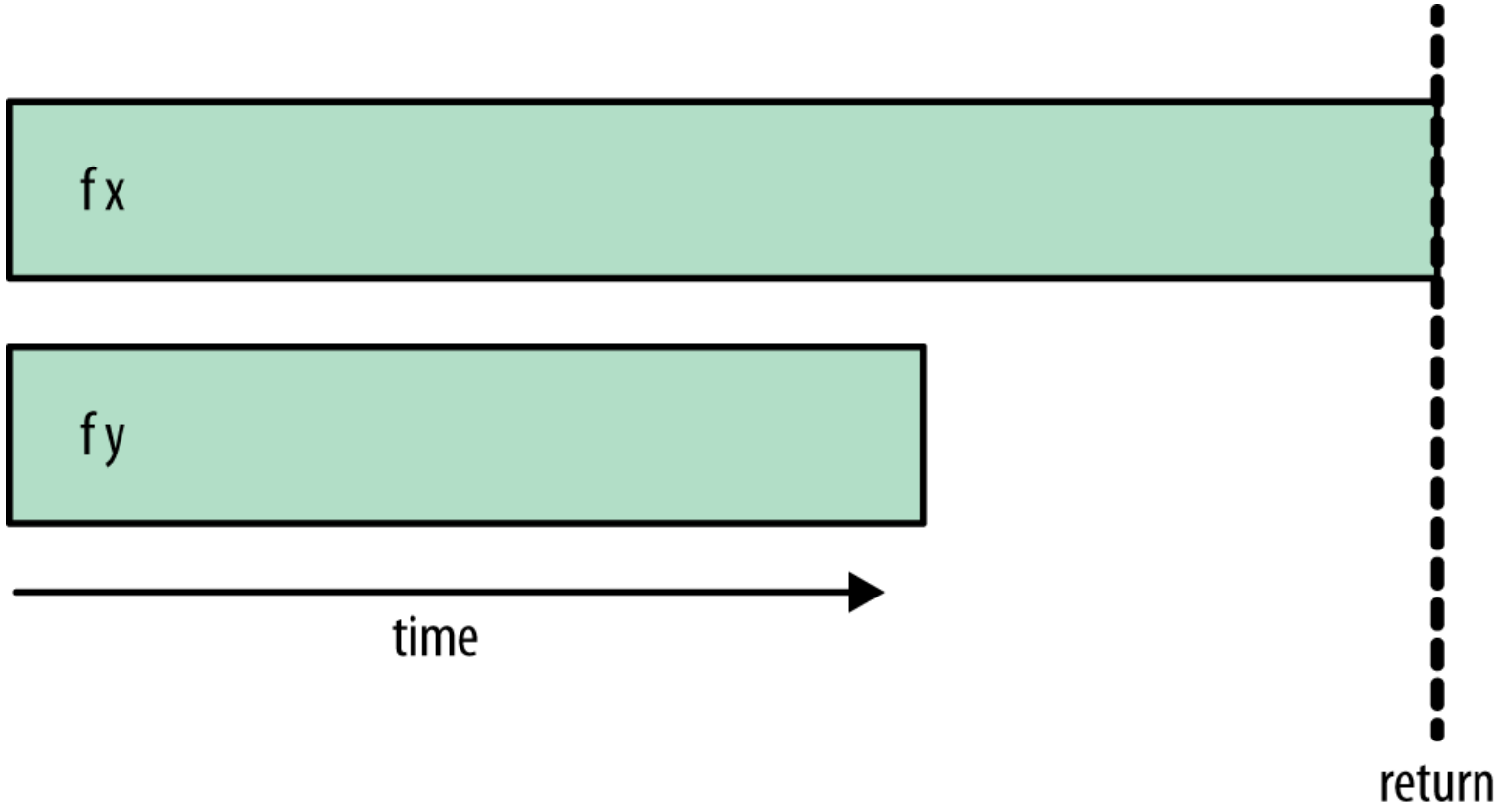


Figura 3: rpar-rpar-rseq-rseq

Escolhendo a estratégia

UFABC

- 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

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

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

UFABC

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

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

UFABC

Dessa forma podemos escrever:

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

Mas seria bom separar a parte sequencial da parte paralela para uma melhor manutenção do código.

UFABC

Podemos então definir:

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

UFABC

Com isso nosso código se torna:

```
1 (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.

UFABC

- 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:

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

UFABC

Nossa função `parPair` pode ser reescrita como:

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

UFABC

- 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)

rparWith :: Strategy a -> Strategy a
rparWith strat = parEval . strat
```

UFABC

Dessa forma podemos fazer:

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

Estratégias parametrizadas

UFABC

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

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

Estratégia para listas

UFABC

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

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

Estratégia para listas

UFABC

Essa é justamente a definição de `parMap`:

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

Exemplo: média

Exemplo: média

UFABC

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

```haskell
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 com o parâmetro `-s` obtemos:

```
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

UFABC

O que houve?

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

UFABC

Vamos instalar o programa *threadscope* para avaliar, faça o download em
http://hackage.haskell.org/package/threadscope
e:

```
1 $ tar zxvf threadscope-0.2.10.tar.gz
2 $ cd threadscope-0.2.10
3 $ stack install threadscope
```

UFABC

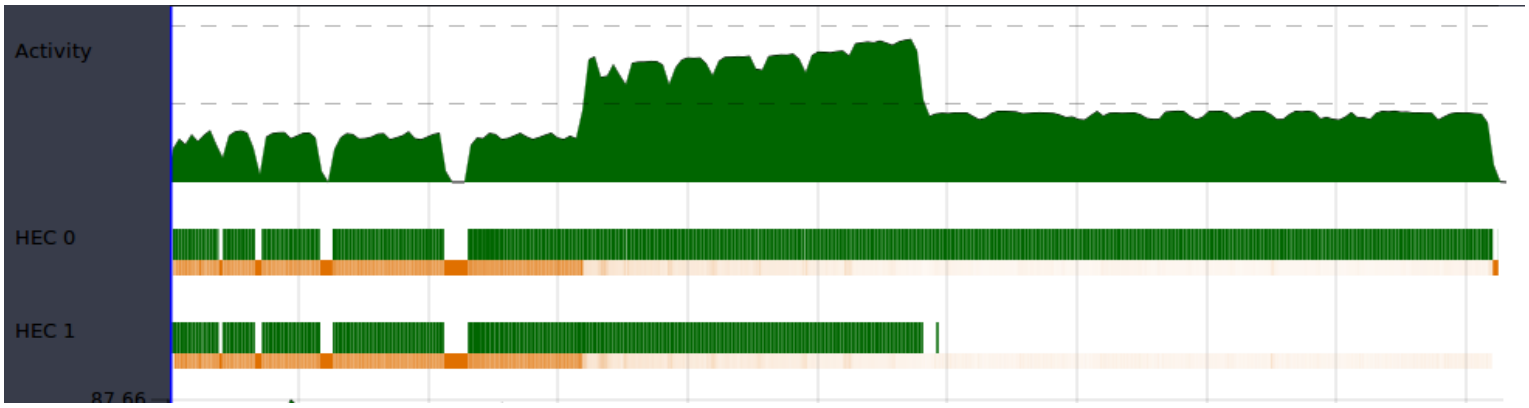
Execute o programa da média incluindo o parâmetro `-ls` e faça:

```
1 $ threadscope media.eventlog
```

Exemplo: média

UFABC

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

UFABC

- 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

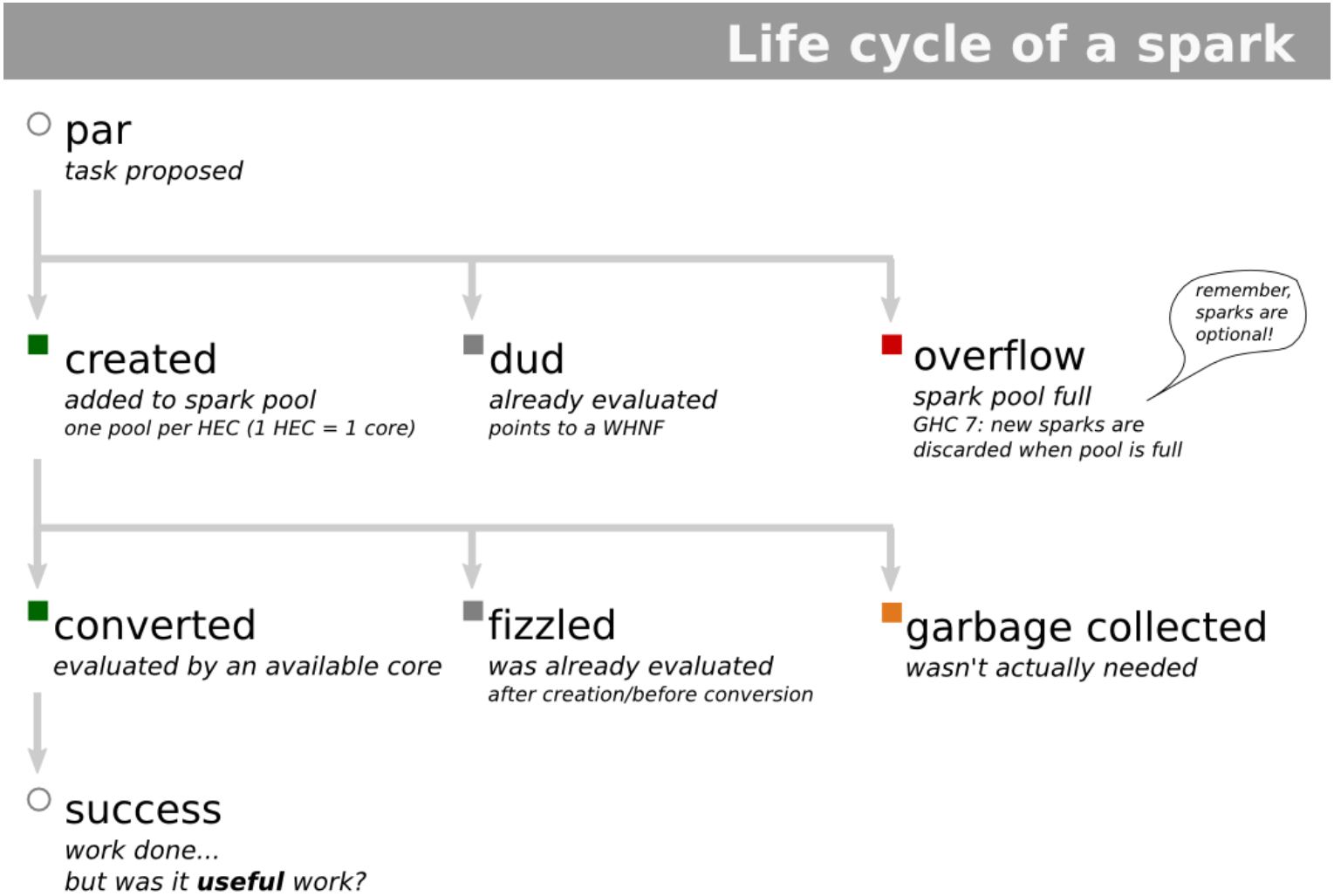
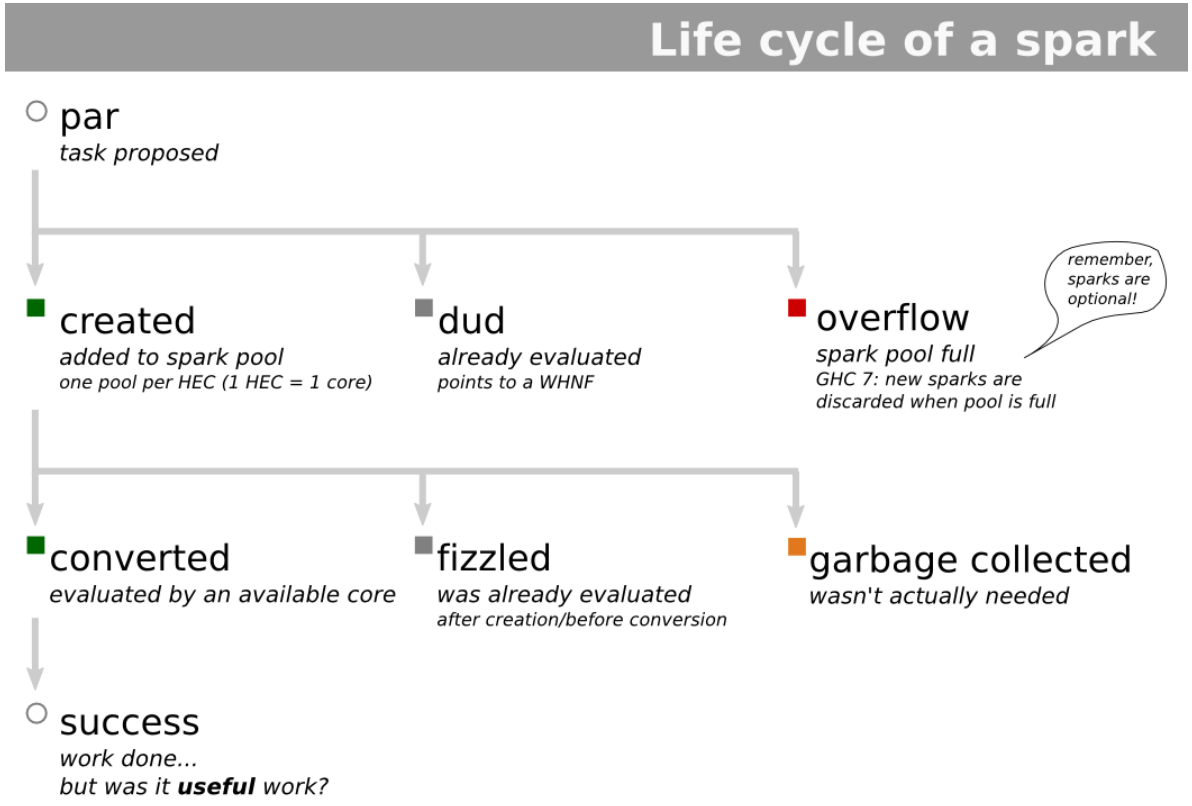


Figura 4: https://wiki.haskell.org/ThreadScope_Tour/SparkOverview

Vida de um spark

UFABC

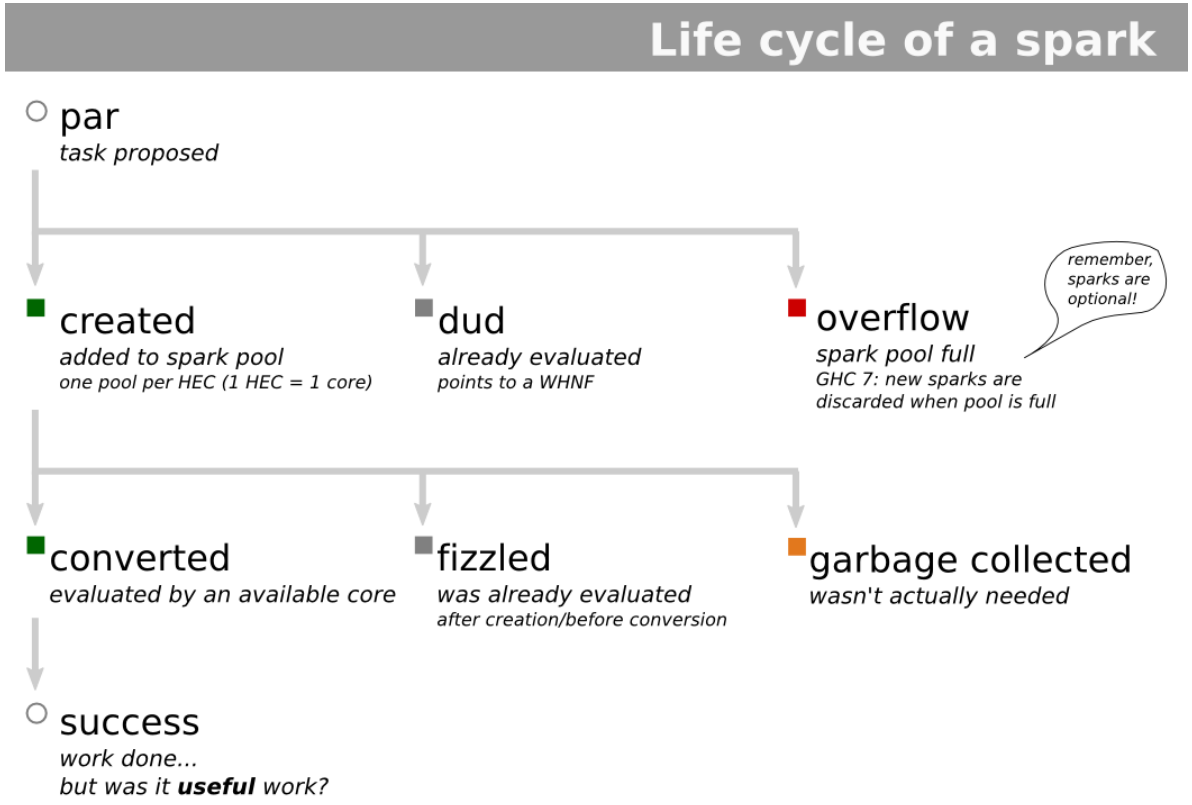
- Cada elemento que é passado para a função `rpar` cria um spark e que é inserido no *pool*.
- Quando um processo pega esse spark do pool, ele é convertido em um processo e então é executado.



Vida de um spark

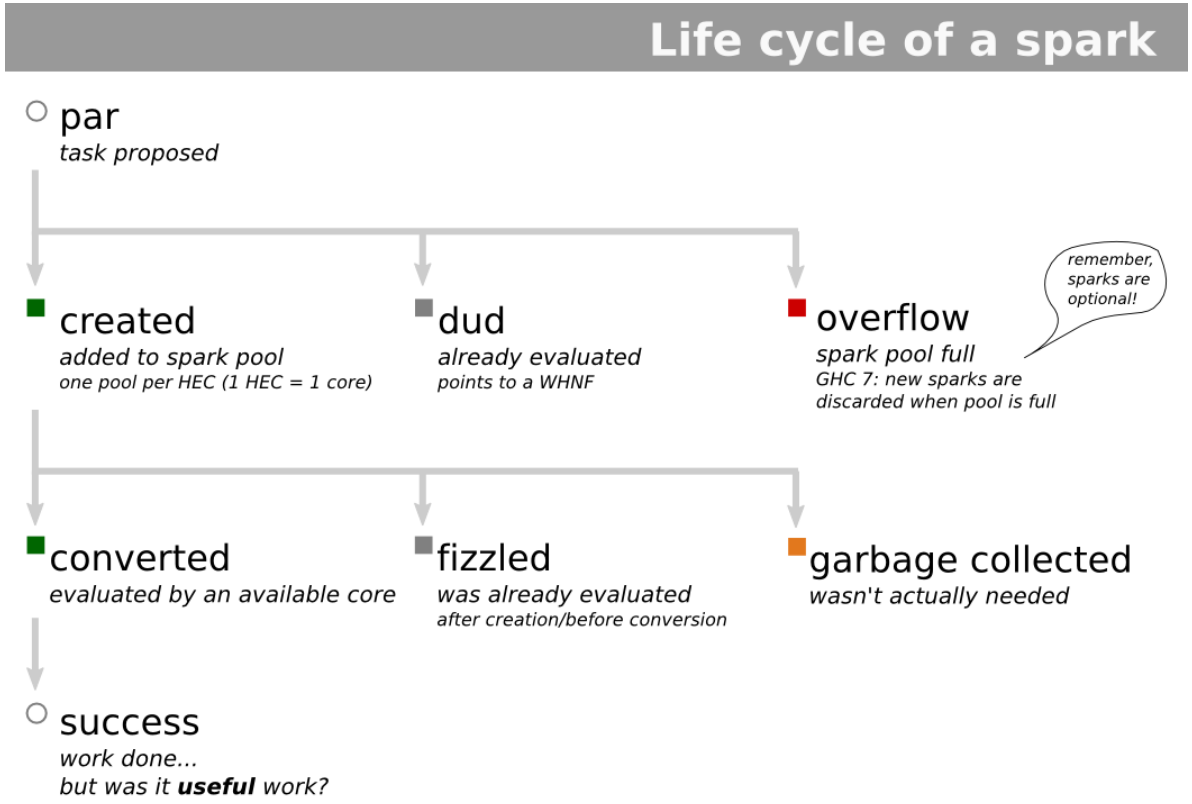
UFABC

- No momento da criação, antes de criar o spark, 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

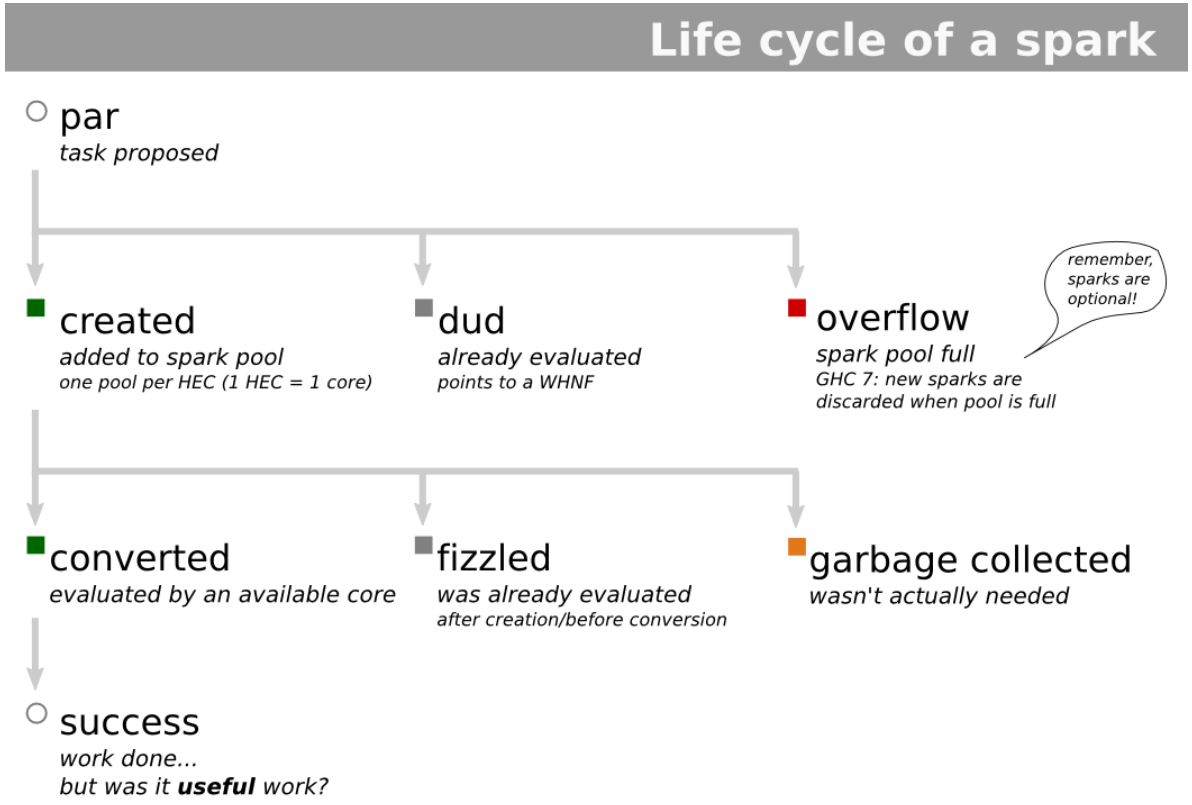
- Se o pool estiver cheio no momento, ela retorna o status *overflow* e não cria o spark, simplesmente avalia a expressão no processo principal.



Vida de um spark

UFABC

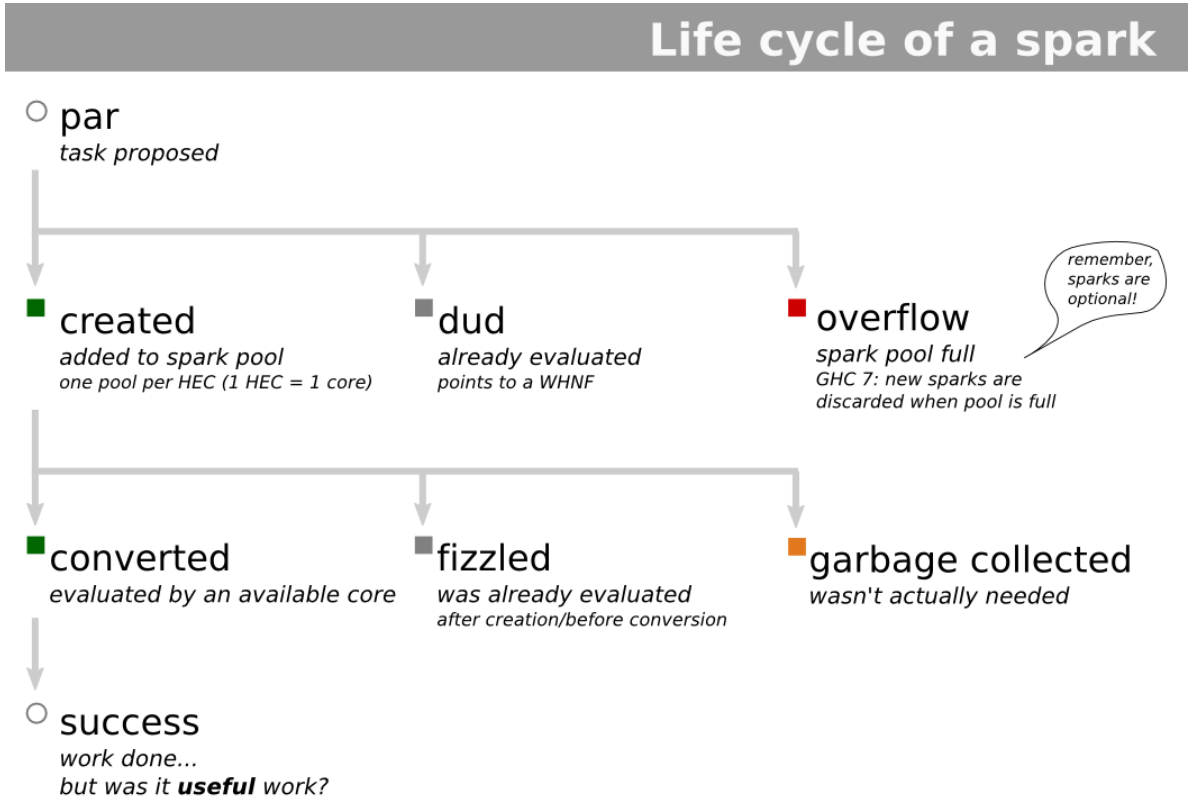
- Se no momento de ser retirado do pool ele já tiver sido avaliado em outro momento, o spark retorna status *fizzled*, similar ao *dud*.



Vida de um spark

UFABC

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



Sinais de problemas:

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

UFABC

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

UFABC

Vamos tirar a estratégia...

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

Exemplo: média

UFABC

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

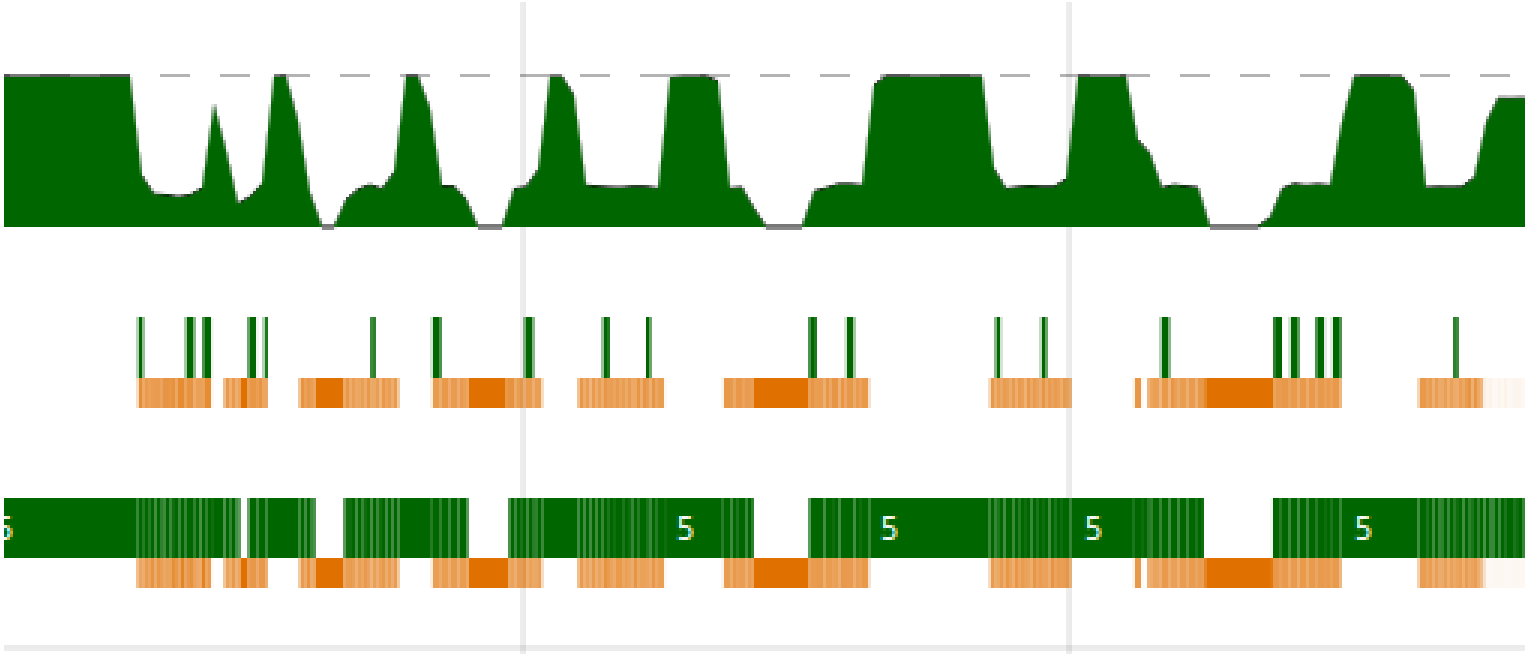
```haskell
meanPar :: [[Double]] -> [Double]
meanPar xss = concat medias
  where
    medias = map mean chunks `using` parList rseq
    chunks = chunksOf 1000 xss
```

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

UFABC

O resultado:

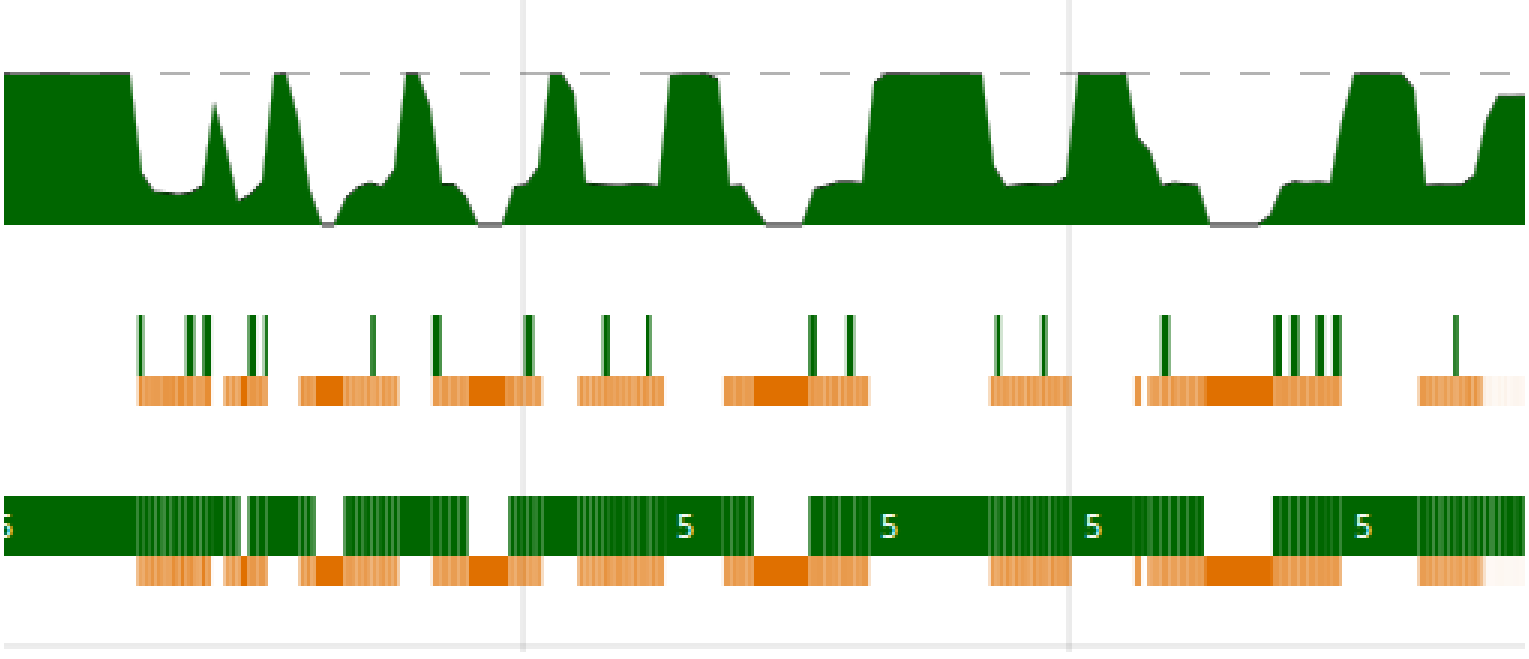
```
Total   time    1.289s  (  1.215s elapsed)
```



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

UFABC

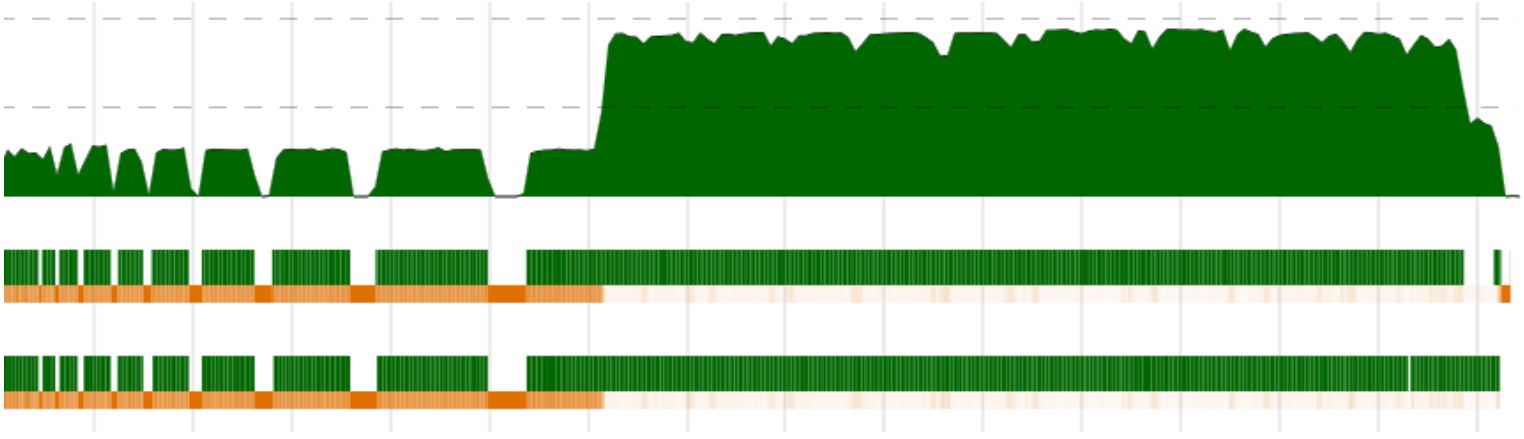
Vamos usar a estratégia *rdeepseq*.

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

Exemplo: média

UFABC

Total time 1.303s (0.749s elapsed)



☺