Paradigmas de Programação

Fabrício Olivetti de França

02 de Agosto de 2018

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
```

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

Eu quero agora!

```
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]
```

Exercício

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

```
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)

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.

Note que o Monad Eval é **puro** e pode ser utilizado fora de funções com IO.

Exemplo

(Slides apenas para replicação em lab)

Crie um projeto chamado `paralelo`:

```
stack new paralelo simple
stack setup
```

Exemplo

(Slides apenas para replicação em lab)

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**

Exemplo

(Slides apenas para replicação em lab)

No arquivo *Main.hs* acrescente:

```
import Control.Parallel.Strategies
import Control.Exception
import Data.Time.Clock
```

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 20` e `fib 21`:

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

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

Eval Monad

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

Exemplo

(Slides apenas para replicação em lab)

Altere a função `main` para:

```
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:

Explicando os parâmetros

(Slides apenas para replicação em lab)

- -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

Eval Monad

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

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

Eval Monad

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.

rpar-rpar

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

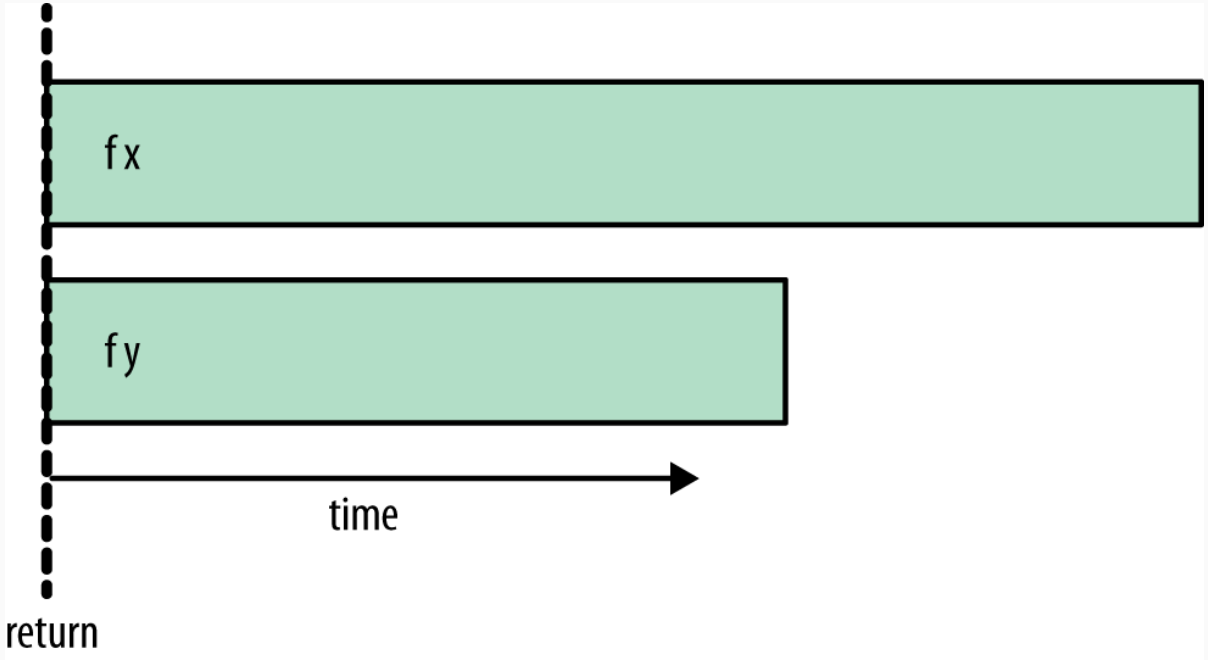


Figure 1: rpar-rpar

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)
```

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`:

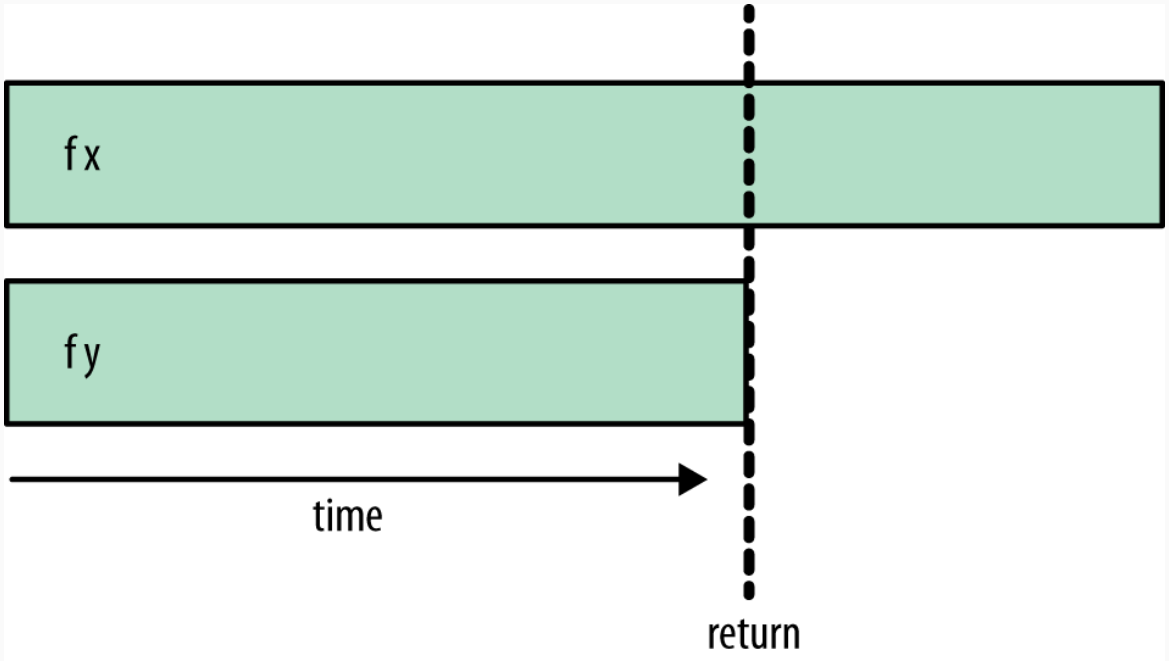


Figure 2: rpar-rseq

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:

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

rpar-rpar-rseq-rseq

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

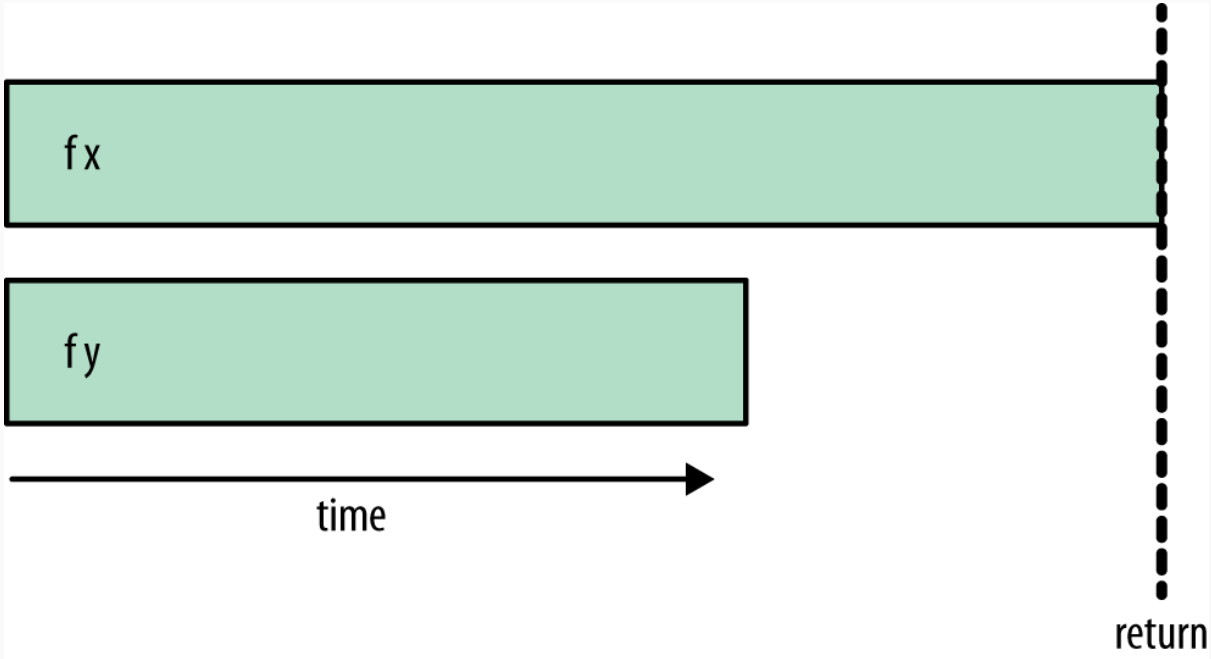


Figure 3: rpar-rpar-rseq-rseq

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.Paralle.Strategies` define também o tipo:

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

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 bom separar a parte sequencial da parte paralela para uma melhor 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:

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

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 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

O que houve?

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

threadscope

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

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

threadscope

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

```
$ threadscope media.eventlog
```

Exemplo: média

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

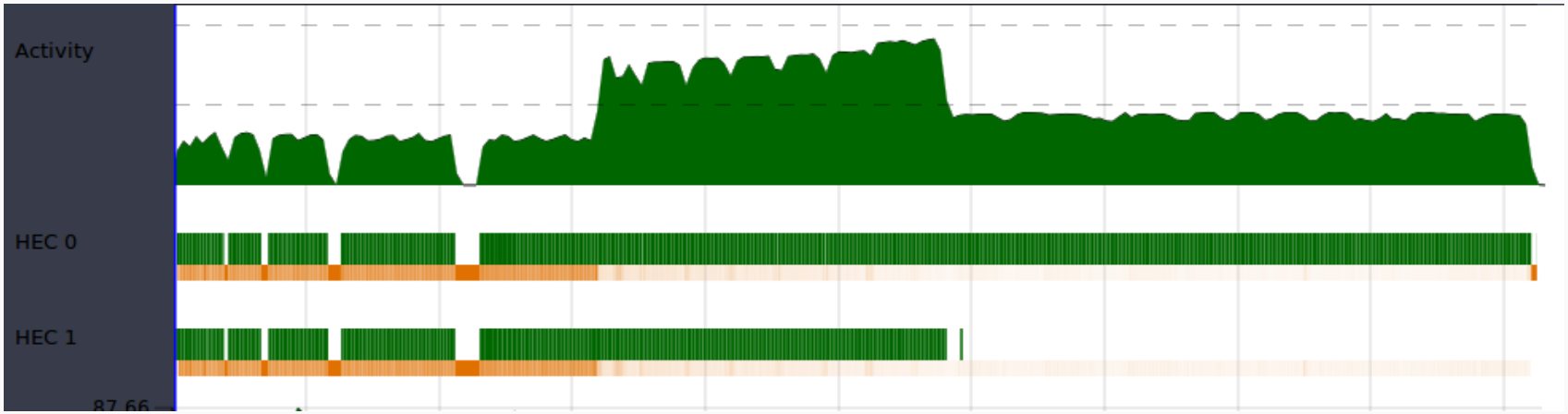


Figure 4:

Por que um core fez o dobro do trabalho?

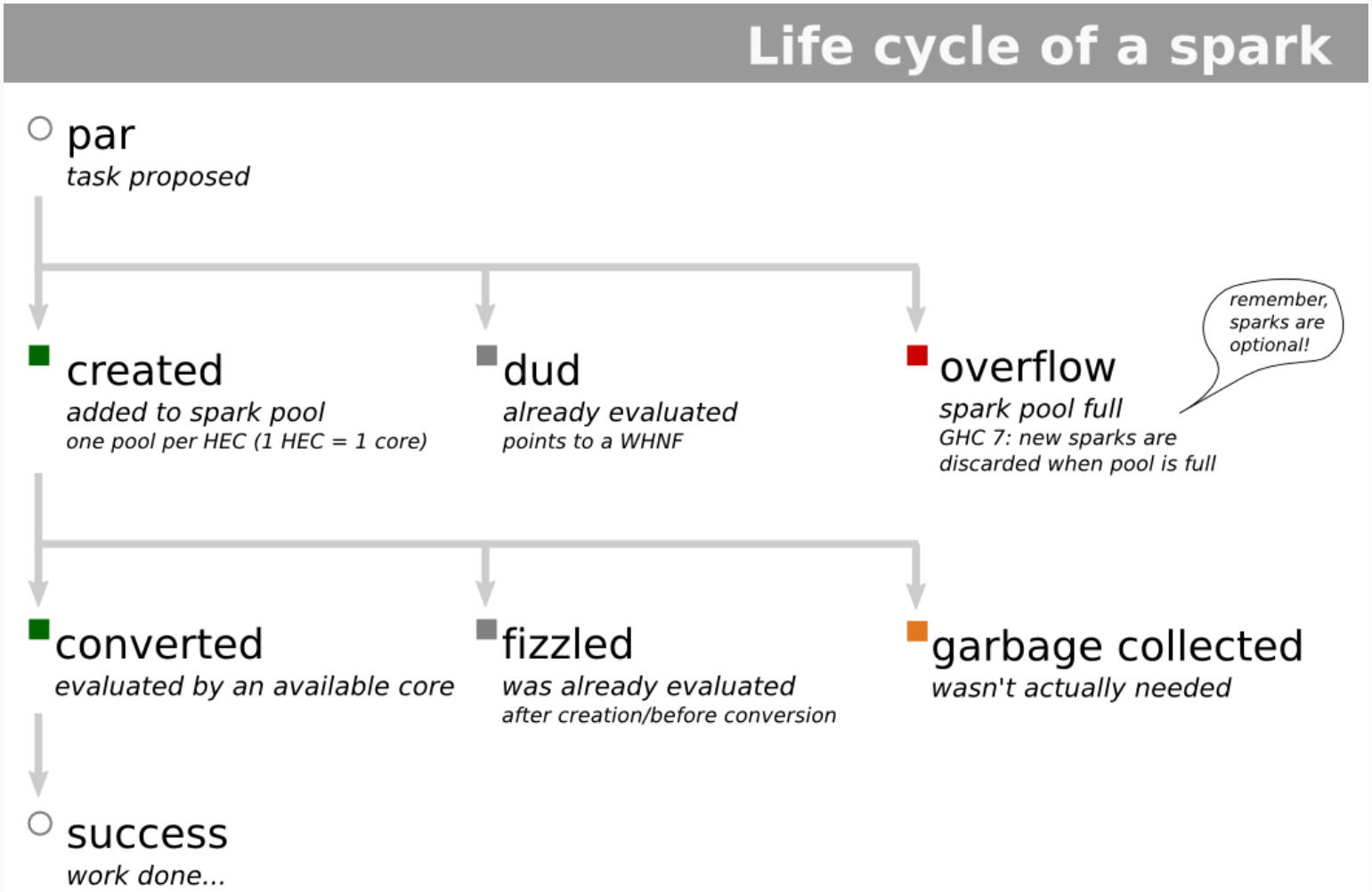
Vida de um spark

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.

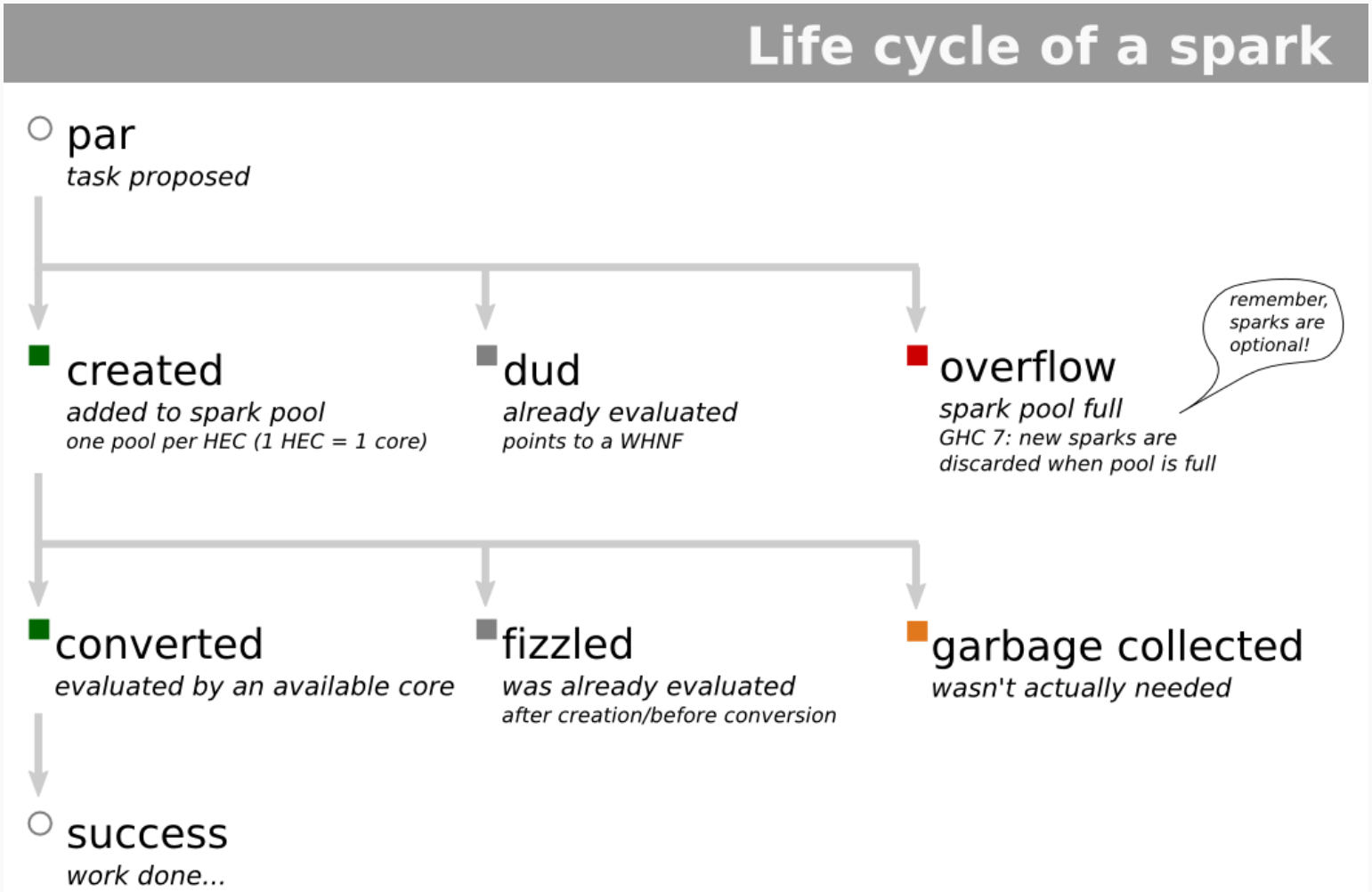
Vida de um spark

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:



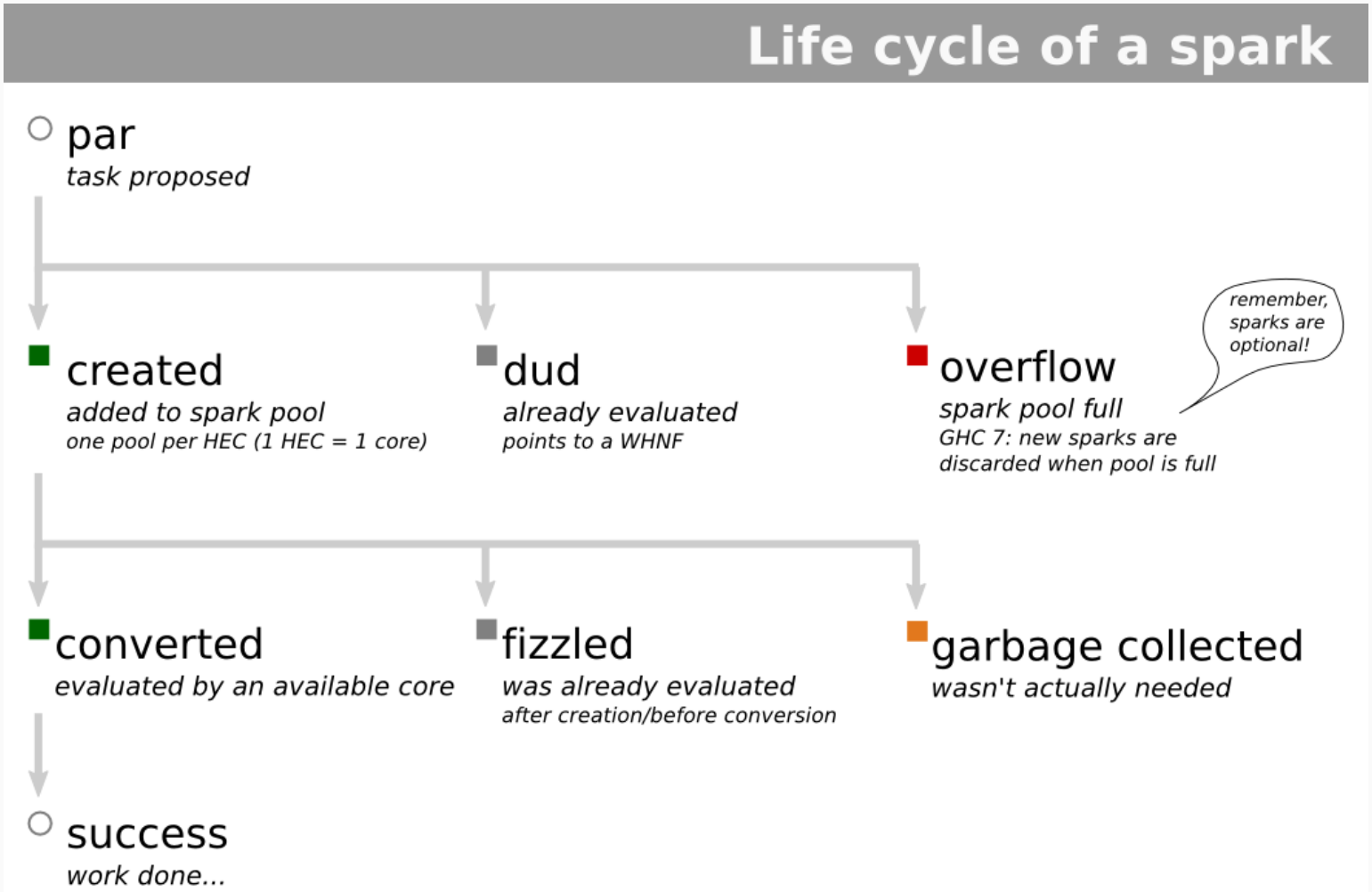
Vida de um spark

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.



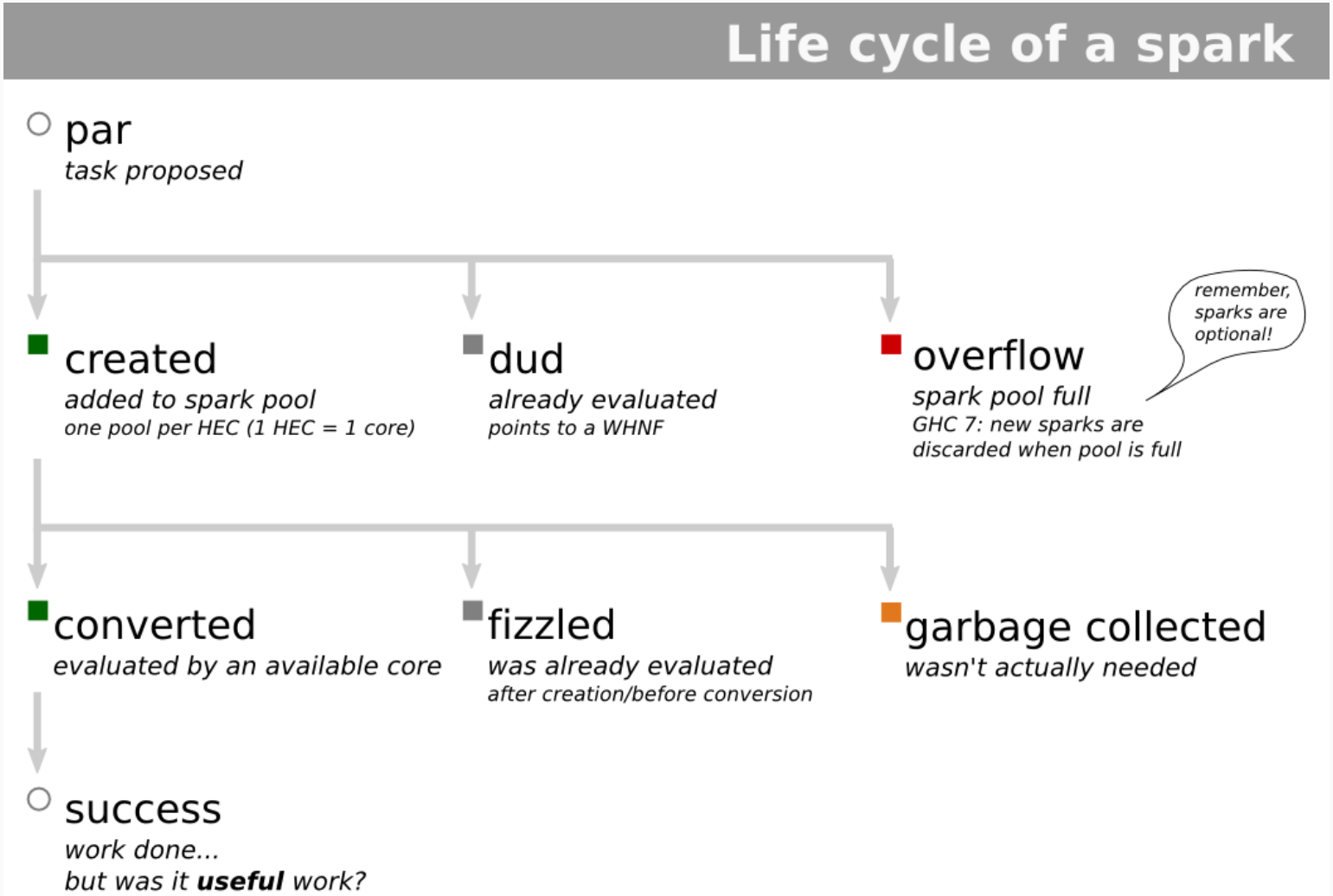
Vida de um spark

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



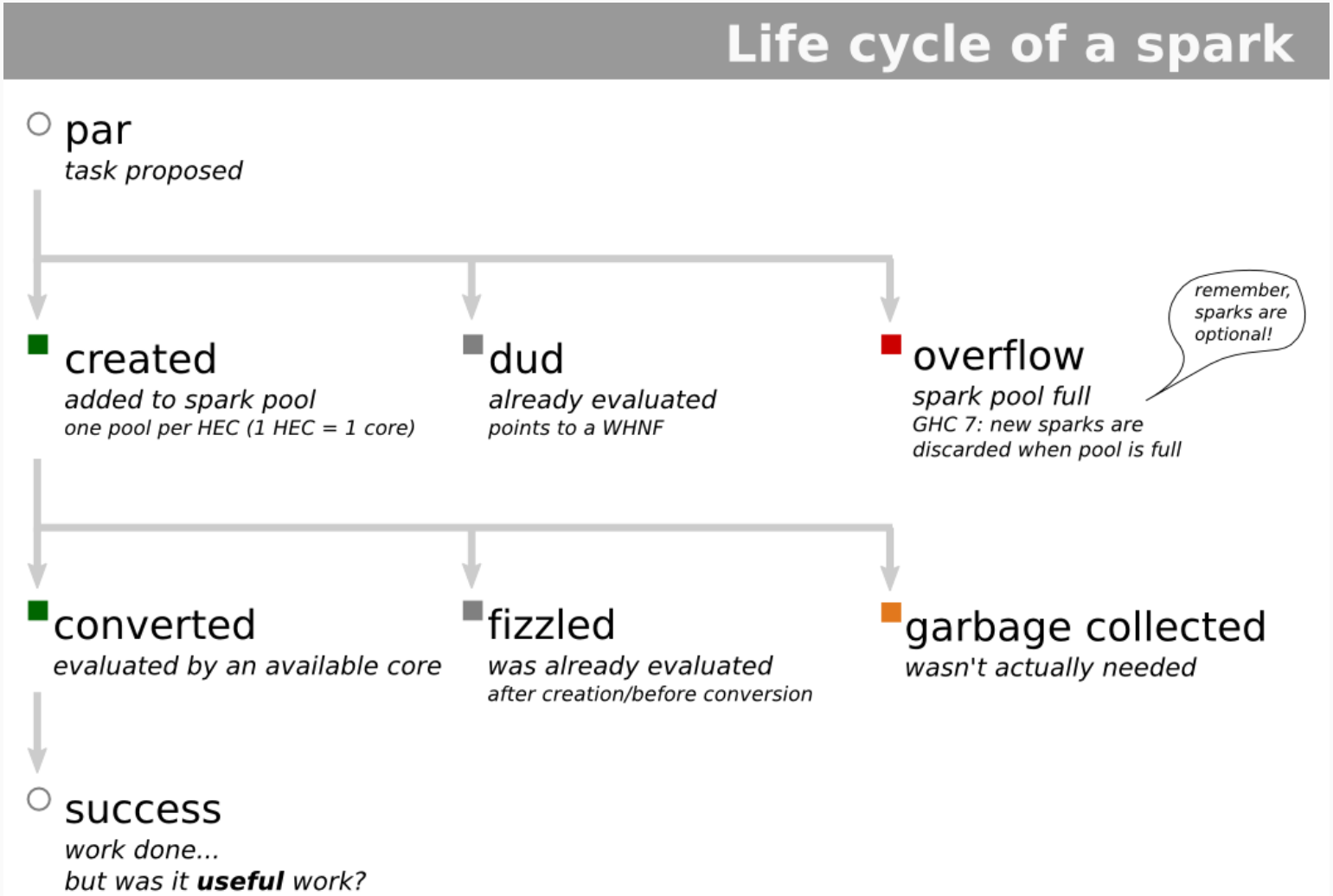
Vida de um spark

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:

Figure 10:

Exemplo: média

Vamos tirar a estratégia...

```
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)
```

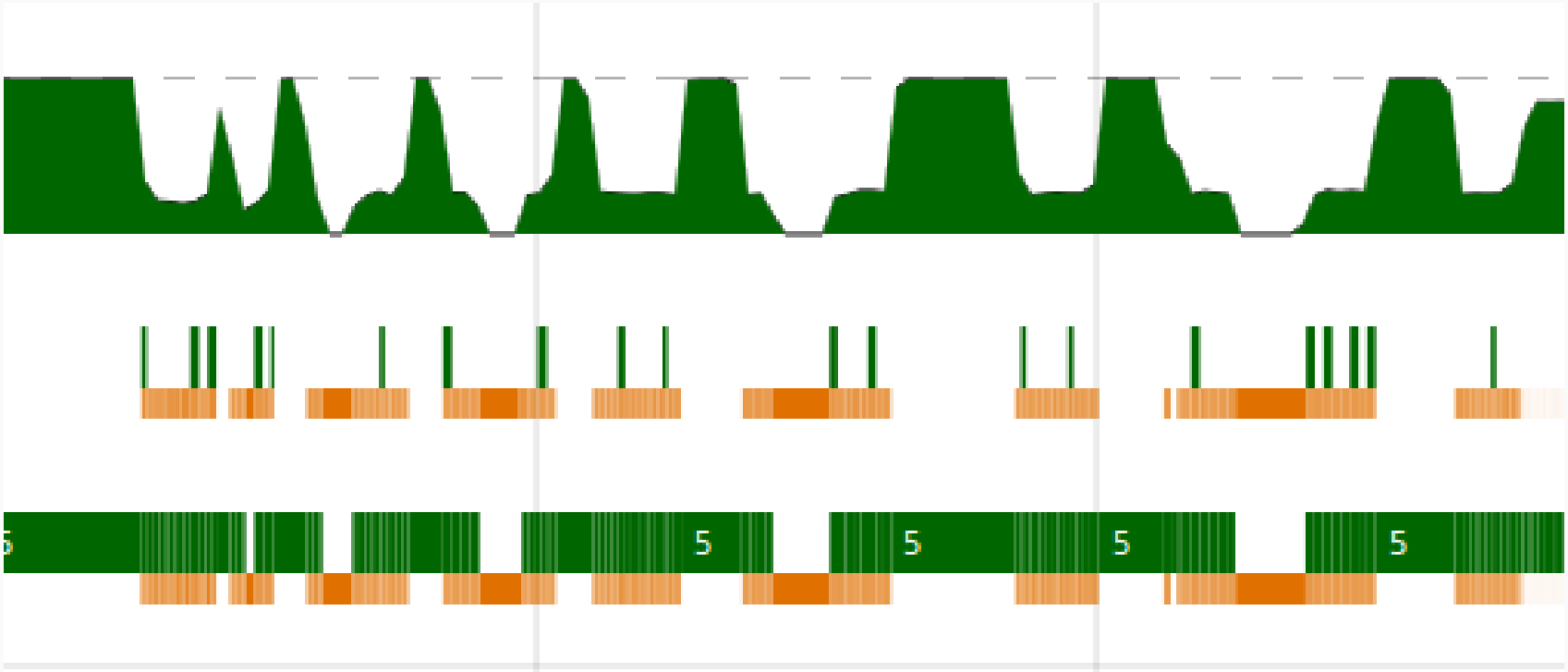


Figure 11:

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!

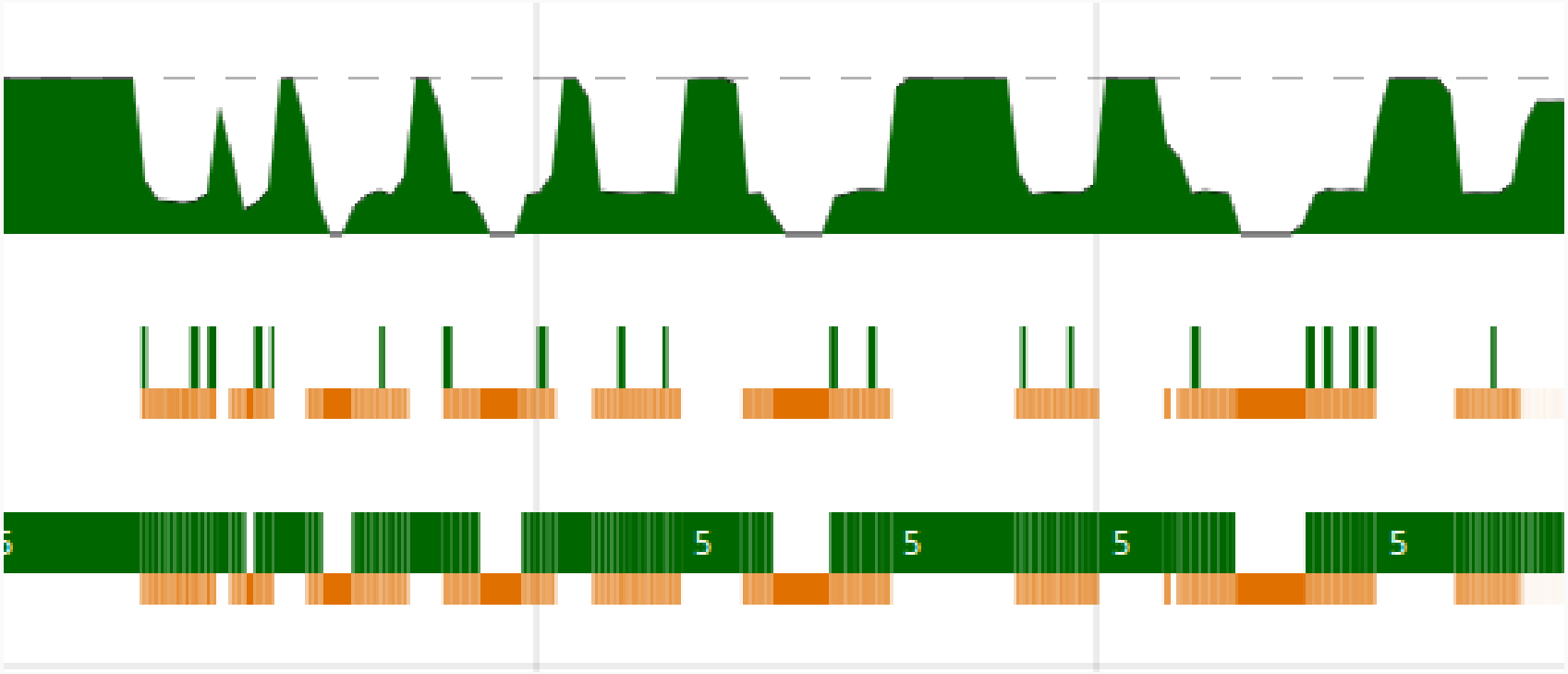


Figure 12:

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)

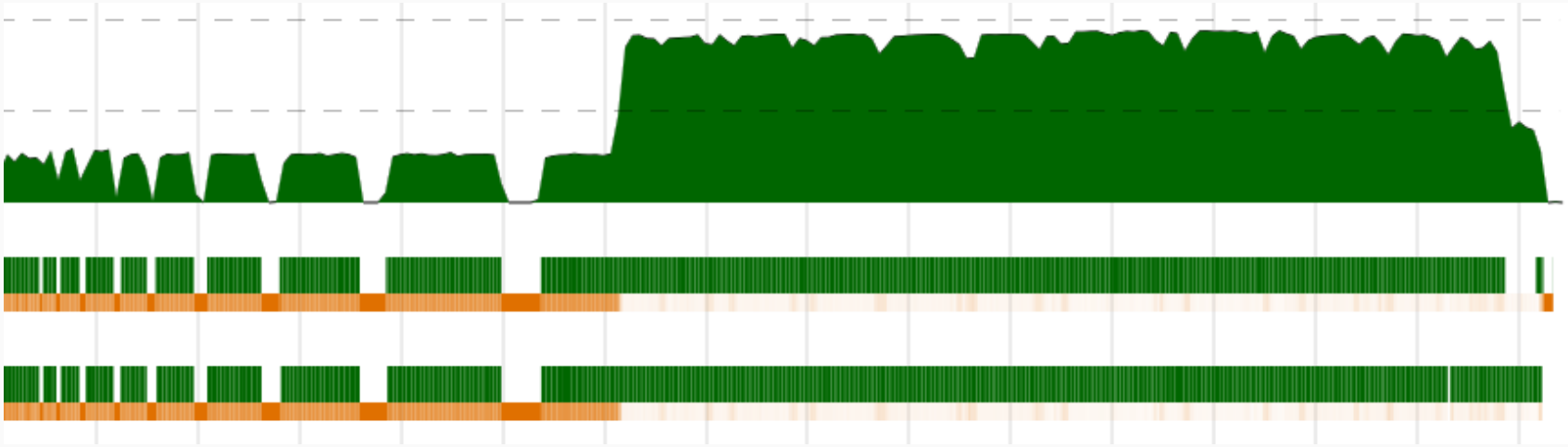


Figure 13:

:)

Próximo Lab

No próximo Lab vamos implementar o algoritmo k-Means em paralelo. Para isso se inscrevam para utilizar o DevCloud da Intel através do formulário:

https://docs.google.com/forms/d/e/1FAIpQLSdUGmUNlgEJ3V10C1Kvgl21J04cY

Sigam o tutorial que será enviado para o e-mail de vocês para acessar o espaço de vocês via terminal. Faça o donwload do stack em https://docs.haskellstack.org/en/stable/install_and_upgrade/ (manual download) e descompacte na área da nuvem.

Caso não consigam acesso, não tem problema, só que utilizaremos 2 cores ao invés de 64...