# Programação Paralela e Concorrente com Linguagens Funcionais

MCZA020-13 - Programação Paralela

Emilio Francesquini
e.francesquini@ufabc.edu.br

Junho de 2020

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

# Disclaimer

UFABC

- Estes slides foram preparados para o curso de Programação Paralela da UFABC.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.

# Programação Concorrente com Estado Compartilhado

# Estimando $\pi$

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \ldots + (-1)^n \frac{1}{2n+1} + \ldots\right)$$

```
1  double factor = 1.0;
2  double sum = 0.0;
3  for (i = 0; i < n; i ++, factor = -factor) {
4      sum += factor/(2*i +1);
5  }
6  pi = 4.0*sum;
```

# Uma função para $\pi$ utilizando threads

UFABC

```
1  void* Thread_sum(void* rank) {
2    long my_rank = (long) rank;
3    double factor;
4    long long i;
5    long long my_n = n/thread_count;
6    long long my_first_i = my_n*my_rank;
7    long long my_last_i = my_first_i + my_n;
8
9    if (my_first_i % 2 == 0)
10       factor = 1.0;
11   else
12       factor = -1.0;
13   for (i = my_first_i; i<my_last_i; i++,factor=-factor)
14       sum += factor/(2*i+1);
15
16   return NULL;
17 }
```

- Não é a fórmula que converge mais rápido, ela precisa de muitos termos para chegar a uma boa precisão.

| | $10^5$ | $10^6$ | $10^7$ | $10^8$ |
|---|---|---|---|---|
| $\pi$ | 3.14159 | 3.141593 | 3.1415927 | 3.14159265 |
| 1 Thread | 3.14158 | 3.141592 | 3.1415926 | 3.14159264 |
| 2 Threads | 3.14158 | 3.141480 | 3.1413692 | 3.14164686 |

  - Note que conforme n cresce, a estimativa com 1 thread melhora

UFABC

Há uma região crítica (ou seção crítica, *critical section*) do código que precisa ser protegida.

| Tempo | Thread 0 | Thread 1 |
|---|---|---|
| 1 | Iniciado pela thread principal | |
| 2 | Chama `Compute()` | Iniciado pela thread principal |
| 3 | Atribui `y = 1` | Chama `Compute()` |
| 4 | Põe `x=0` e `y=1` em registradores | Atribui `y = 2` |
| 5 | Soma 0 e 1 | Põe `x=0` e `y=2` em registradores |
| 6 | Armazena 1 em `x` | Soma 0 e 2 |
| 7 | | Armazena 2 em `x` |

# Versão da soma usando mutexes

UFABC

```
1  void* Thread_sum(void* rank) {
2    long my_rank = (long) rank;
3    double factor; long long i;
4    long long my_n = n/thread_count;
5    long long my_first_i = my_n*my_rank;
6    long long my_last_i = my_first_i + my_n;
7    double my_sum = 0.0;
8    if (my_first_i % 2 == 0)
9        factor = 1.0;
10   else
11       factor = -1.0;
12   for (i = my_first_i; i<my_last_i; i++,factor=-factor)
13       my_sum += factor/(2*i+1);
14   pthread_mutex_lock(&mutex);
15   sum += my_sum;
16   pthread_mutex_unlock(&mutex);
17   return NULL;
18 }
```

UFABC

- Há um compartilhamento de recursos (variáveis/memória, dispositivos, ...)
- Estabelece-se um mecanismo de controle e sincronização de acesso a esses recursos
  - Travas (locks, mutexes, futexes, ...)
  - Semáforos
  - Monitores
  - ...

UFABC

- Processos e Threads são a maneira do sistema operacional representar execuções concorrentes
  - Logo nada mais natural que as linguagens de programação mapeiem estes conceitos

**Threads são utilizados pela maior parte das linguagens tradicionais**

- C/C++
- Java
- Python
- C#
- Ruby
- ...

UFABC

- Threads, por definição, compartilham os recursos do processo ao qual pertencem com as demais threads do mesmo processo.
  - Exigem mecanismos de sincronização para acesso aos recursos compartilhados para garantir
    - Corretude
    - Consistência
  - Funcionam através de modificações de estado compartilhado.

*"Programas multi-threaded não triviais são incompreensíveis para um ser humano..."*

Edward A. Lee, The Problem with Threads

# Entram em cena: linguagens funcionais

UFABC

- Como escrever o mesmo código em uma linguagem funcional?
  - Em uma linguagem funcional programas são avaliações de funções matemáticas que não alteram estados.
  - Dados são imutáveis!

```
1 foo(X, Y) ->
2    Z = 0,
3    Z = X + Y, %% Ilegal em Erlang (Detecção em RT)
4    Z.
```

- Pense em declarações matemáticas. Faria sentido dizer que $x = 3$ para logo em seguida dizer $x = x + 1$? Esse é o espírito de linguagens funcionais.

UFABC

- Como dados são imutáveis, o valor da avaliação de uma função depende exclusivamente do valor de seus parâmetros.
  - Transparência referencial.
  - Isso significa que qualquer valor pode ser substituído por uma expressão equivalente (e vice-versa) sem que o comportamento do programa se altere.
  - Isso também significa que implementar uma função que incrementa (ou altera de qualquer forma o valor de) uma variável é impossível.
  - Uma outra consideração é que variáveis não deveriam ser chamadas de variáveis já que seu valor nunca varia.
- Sendo assim, como implementar um laço? (Não pode variar o `i`)

Versão sequencial (agora em Haskell):

```haskell
pi :: Int -> Double
pi n = 4 * pi' n
  where
    pi' 0 = 1
    pi' i =
        parcela + pi' (i - 1)
      where
        sinal = if even i then 1 else -1
        parcela = sinal / fromIntegral (2 * i + 1)
```

# Versão mais idiomática

UFABC

```haskell
pi :: Int -> Double
pi n =
  4 * (pares - impares)
  where
    soma ls = sum [1 / fromIntegral(2 * i + 1) | i <- ls]
    pares   = soma [0,2..n]
    impares = soma [1,3..n]
```

```erlang
pi(N) ->
  Soma = fun (Ls) -> lists:sum([1 / (2 * I + 1) || I <- Ls]) end,
  Pares = Soma(lists:seq(0,N,2)),
  Impares = Soma(lists:seq(1,N,2)),
  4 * (Pares - Impares).
```

# Outra alternativa

```haskell
pi :: Int -> Double
pi n =
  4 * foldl (+) 0 (map f [0..n])
  where
    f i = (if even i then 1 else -1)
      / fromIntegral (2 * i + 1)
```

Equivalente em Erlang:

```erlang
sinal(I) ->
  case even(I) of
    true  -> 1;
    false -> -1
  end.

pi(N) ->
  4 * lists:foldl(
    fun(X, Soma) ->
      sinal(X) / (2 * X + 1) + Soma
    end,
    0,
    lists:seq(0, N)).
```

# Mas isto não é tudo...

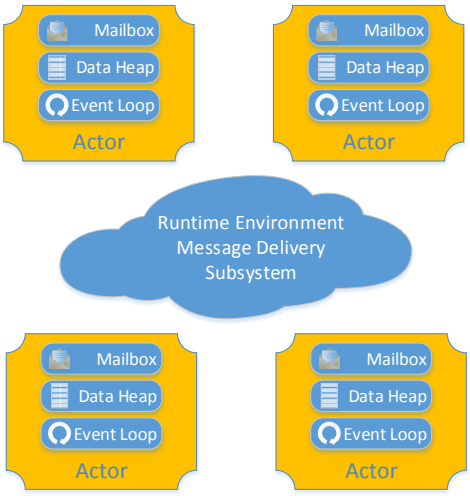
- Mesmo nas linguagens imperativas, diante da dificuldade de se utilizar threads corretamente, passou-se a procurar alternativas
  - Trocas de mensagens
    - MPI
    - Modelo de Atores (vindo do mundo funcional)
  - Memória transacional
  - Dataflow

# O Modelo de Atores

# O Modelo de Atores

Proposto originalmente por Hewitt *et al.* em 1973 e, mais tarde generalizado para concorrência em 1986 por Agha

- Baseado em princípios muito simples
  - Troca de mensagens assíncronas
  - Recepção seletiva de mensagens
  - Área de memória (*heap*) e laço de eventos privados
- Leve
  - Criado em quantidades excedendo o número de núcleos de processamento
  - Desacopla o número de atores do hardware



# Aplicações baseadas em atores

- Têm a sua disposição ambientes de execução leves com distribuição automática e transparente de carga
- São altamente otimizados para máquinas com memória compartilhada
- São muito utilizados
  - Linguagens: Erlang, Elixir, Scala, Akka, Kilim, Salsa, ...
  - Applications: WhatsApp, Facebook Chat, Chef Server, Twitter, CouchDB, ...
  - Atendem milhares de clientes simultaneamente em serviços dedicados executando em máquinas poderosas

# Principais pontos sobre o Modelo de Atores

UFABC

- Atores em vez de objetos
- Não há estado compartilhado entre atores
- Toda comunicação se dá por trocas de mensagens assíncronas
  - As mensagens são imutáveis
- Caixas postais podem fazer um buffer das mensagens recebidas
  - É o único canal de comunicação com um ator e age como uma fila com múltiplos produtores e um único consumidor

# Atores podem...

UFABC

- Reagir a alguma mensagem recebida executando um comportamento
  - Eles podem apenas alterar o próprio estado
  - ...ou mandar mensagens para outros atores.
- Modelo muito mais natural do que o modelo orientado a objetos!
  - Você é capaz de mudar o estado de algo na cabeça do seu colega? Isto é feito por compartilhamento de estado ou por troca de mensagens?
- Como os atores nunca compartilham o estado, nunca precisam competir por travas para acessar recursos compartilhados

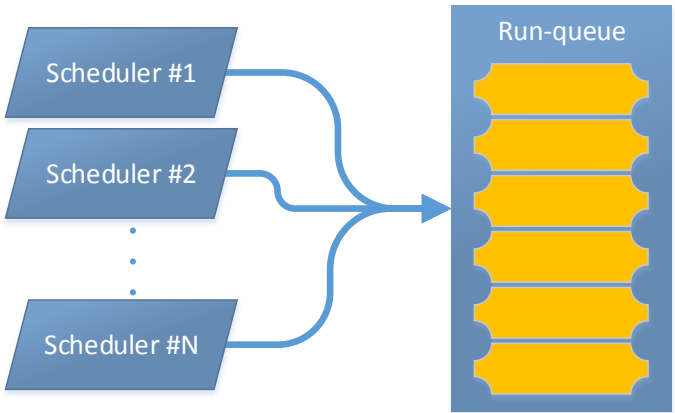
Os ambientes de execução do modelo de atores podem ser divididos em 2 principais categorias:

- Baseados em threads
  - Tem limitações sobre o número de atores e deixa a cargo do sistema operacional o escalonamento dos atores
- Baseados em eventos
  - Compostos de filas simples ou múltiplas de onde um pool de threads obtém as tarefas a serem executadas
  - Sistemas operacionais ainda não estão tão otimizados quanto poderiam para dar suporte a este tipo de aplicações

UFABC

## Single Run-Queue

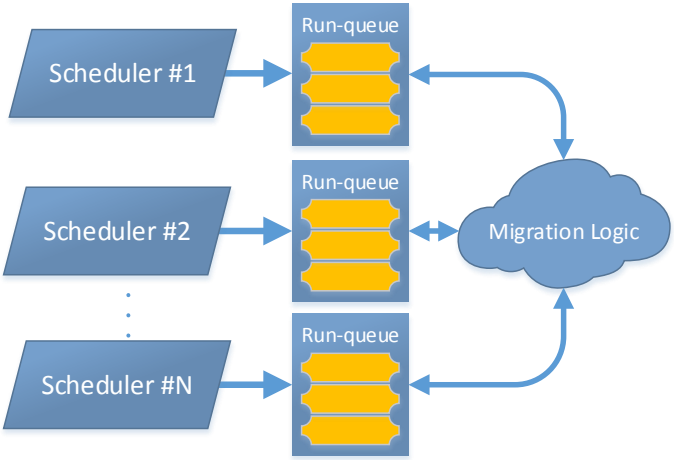
- Utilizado por Akka, Kilim, e VMs antigas de Erlang
- Pode se tornar um gargalo no desempenho
- Não garante soft-affinity



UFABC

## Mulitple Run-Queues

- Utilizado por Erlang, Kilim
- Colocação inicial/soft affinity
- Balanceamento de carga (baseado no tamanho das filas)
- *Work stealing*



# Erlang no mundo real

UFABC

- Ericson AXD 301 switch
  - Milhões de chamadas por minuto, 99,9999999% uptime
- Aplicação de chat do Facebook
- Servidores do Whatsapp
- RabbitMQ
  - AMQP de alto desempenho, 400.000 mensagens por segundo
- CouchDB
- Ejabberd XMPP server - jabber.org

UFABC

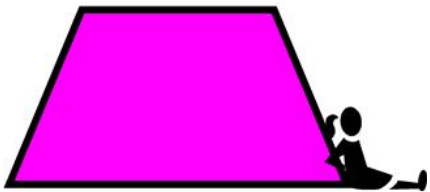
## Benefícios do modelo de atores

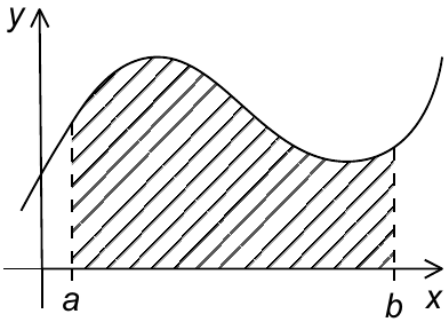
- Mais fácil de entender sistemas não triviais
- Nível de abstração alto
- Evita (ou facilita evitar)
  - Condições de corrida
  - Impasses
  - Starvation
  - Live locks
- Computação distribuída

# Aproximação por trapézios

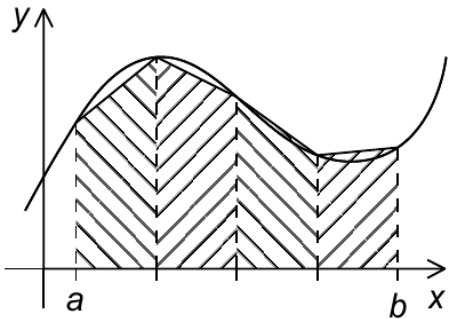
# Aproximação trapezoidal

- A ideia é aproximar o valor da integral de uma função $f(x)$ através da soma das áreas de um número crescente de trapézios
- Quanto maior o número de trapézios
  - Melhor a aproximação
  - Maior custo computacional





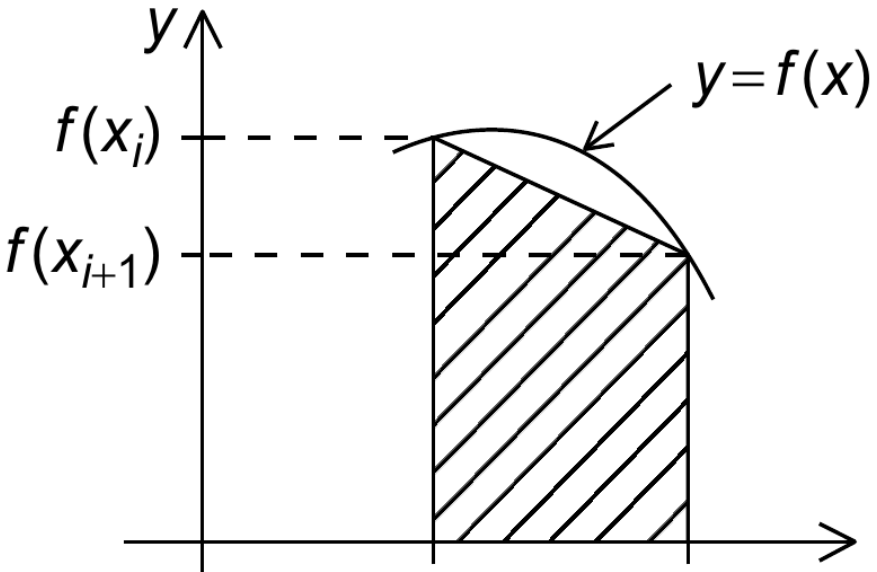

(a)




(b)

# A regra trapezoidal

- Área de um trapézio $= \frac{h}{2}[f(x_i) + f(x_{i+1})]$
- $h = \frac{b-a}{n}$
- $x_0 = a$, $x_1 = a + h$, $x_2 = a + 2h$, ..., $x_{n-1} = a + (n-1)h$, $x_n = b$
- Soma de todas as áreas dos trapézios
  $= h[f(x_0)/2 + f(x_1) + f(x_2) + \ldots + f(x_{n-1}) + f(x_n)/2]$


y
y=f(x)
f(x_i)
f(x_{i+1})

```
/* Input: a, b, n */
h = (b - a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n - 1; i++) {
    x_i = a + i * h;
    approx += f(x_i);
}
approx = h * approx;
```

# Código em Erlang

UFABC

```
1  -module(trap).
2  -export([main/1, go/6, trap/4, trapFold/4]).
3
4  -mode(native).
5
6  f(X) ->
7      X * X.
```

# Código em Erlang - Solução artificial

UFABC

```
1  trap(Left, Right, TrapCount, BaseLen) ->
2      BaseLen * trap2(
3                  Left, Right, TrapCount, BaseLen,
4                  (f(Left) + f(Right)) / 2.0, 1).
5
6  trap2 (Left, Right, TrapCount, BaseLen, Estimate, I)
7    when I < TrapCount ->
8      Estimate2 = Estimate + f(Left + I * BaseLen),
9      trap2 (Left, Right, TrapCount, BaseLen, Estimate2, I + 1);
10 trap2 (_, _, _, _, Estimate, _) ->
11     Estimate.
```

# Código em Erlang - Com cara de funcional

UFABC

```
1  trapFold (Left, Right, TrapCount, BaseLen) ->
2      F = fun (I, Acc) -> Acc + f(Left + I * BaseLen) end,
3      L = lists:seq(1, TrapCount - 1),
4      PartialInt = lists:foldl(F, (f(Left) + f(Right)) / 2.0, L),
5      BaseLen * PartialInt.
```

# Código em Erlang - Juntando tudo

UFABC

```
1  main([A, B, N, P]) ->
2      A2 = list_to_float(atom_to_list(A)),
3      B2 = list_to_float(atom_to_list(B)),
4      N2 = list_to_integer(atom_to_list(N)),
5      P2 = list_to_integer(atom_to_list(P)),
6      H = (B2 - A2) / N2,
7      Ranks = lists:seq(0, P2 - 1),
8
9      [spawn(trap, go, [self(), Rank, A2, N2, H, P2]) || Rank <- Ranks],
10     Int = lists:sum([getResults(Rank) || Rank <- Ranks]),
11
12     io:fwrite(io_lib:format("~.16f\n", [Int])),
13     init:stop().
```

# Código em Erlang - Juntando tudo

UFABC

```
1  go (Src, Rank, A, N, H, P) ->
2      LocalN = N div P,
3      LocalA = A + Rank * LocalN * H,
4      LocalB = LocalA + LocalN * H,
5      %% Descomente a linha abaixo para usar a outra implementação
6      %% LocalInt = trap (LocalA, LocalB, LocalN, H),
7      LocalInt = trapFold (LocalA, LocalB, LocalN, H),
8      Src ! [Rank, LocalInt].
9
10 getResults (_From) ->
11     %% Pode-se esperar na ordem para garantir resultados sempre iguais
12     %% receive [From, Val] ->
13     receive [_, Val] -> %% Ou em qualquer ordem
14             Val
```

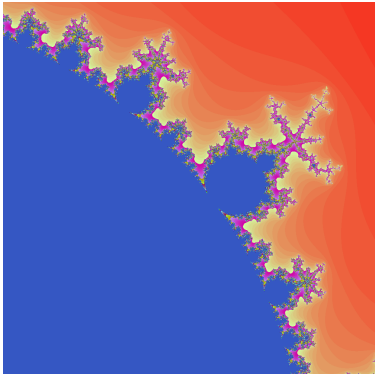
UFABC

Para compilar e rodar faça:

```
1 $ erlc trap.erl
2 $ erl -noshell -s trap main A B N P
```

- O código completo está disponível em: http://professor.ufabc.edu.br/~e.francesquini/2019.q1.pp/files/codigo/trap_atores.html

# Fractal



- Por questão de tempo, não vamos entrar nos detalhes. O código fonte completo está disponível na minha página.
- Considere apenas que há uma função `calculate_pixel` que dados X e Y devolve a cor do pixel naquele ponto.

# Código sequencial

```
1  calculate_image (Mandel) ->
2      Size = Mandel#mandel.size,
3      [calculate_pixel(Mandel, X, Y) ||
4        X <- lists:seq(1, Size), Y <- lists:seq(1, Size)].
```

- Queremos paralelizar este código!
- Como no exemplo do trapézio, é preciso decidir como quebrar as tarefas entre os trabalhadores.

# Código paralelo - Divisão por colunas!

UFABC

```
1  calculate_image (Mandel) ->
2    Size = Mandel#mandel.size,
3    Orig = self(),
4    [spawn_link(fun () -> calculate_column (X, Mandel, Orig) end) ||
5      X <- lists:seq(1, Size)],
6    lists:flatten(array:to_list(rcv_columns(array:new(Size), Size))).
7  rcv_columns(Columns, 0) ->
8    Columns;
9  rcv_columns(Columns, Remaining) ->
10   receive
11     {Column, Pixels} ->
12       rcv_columns (array:set(Column - 1, Pixels, Columns), Remaining -1)
13   end.
14 calculate_column (Column, Mandel, Orig) ->
15   Pixels = [calculate_pixel(Mandel, Column, Y) ||
16     Y <- lists:seq(1, Mandel#mandel.size)],
17   Orig ! {Column, Pixels}.
```

# Paralelismo Monádico

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`:

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

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

# Eval Monad

UFABC

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

# Exemplo

```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)
```

Com 1 linha de execução:

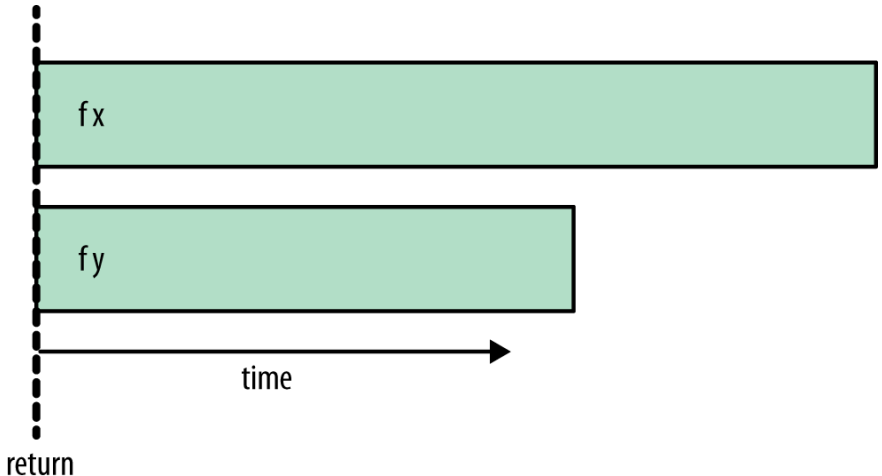
```
0.000002s
(165580141,102334155)
15.691738s
```

Com 2 linhas de execução:

```
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.

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



UFABC

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

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

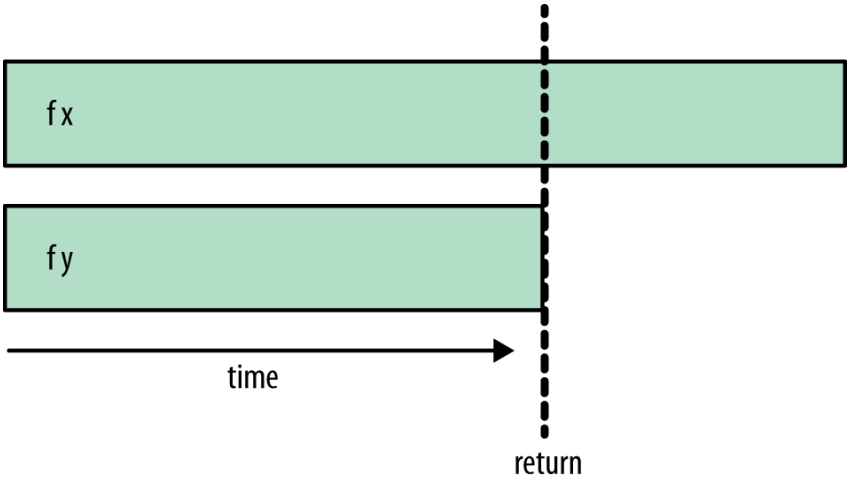
UFABC

- 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`:



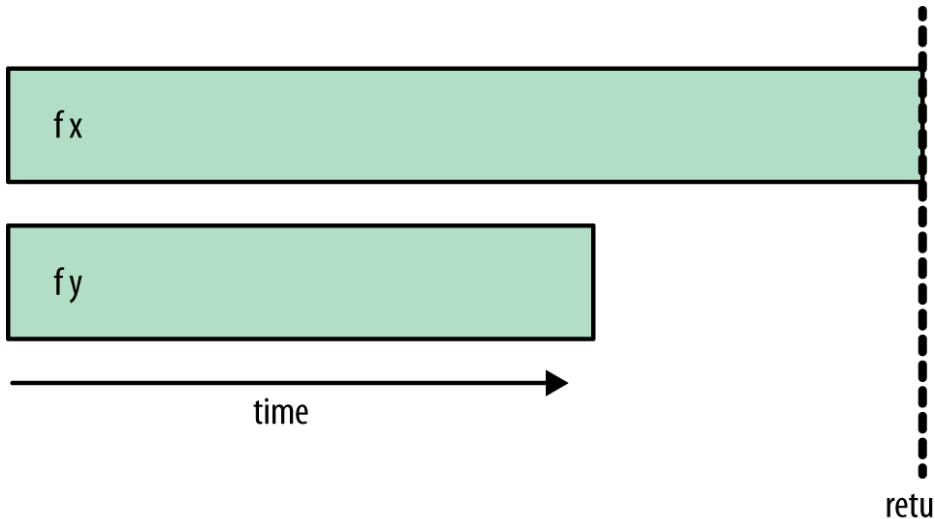
Finalmente podemos fazer:

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

E o resultado da execução com *N2* é:

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

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



# 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:

```
1 -- :: (a,b) -> Eval (a,b)
2 parPair :: Strategy (a,b)
3 parPair (a,b) = do a' <- rpar a
4                    b' <- rpar b
5                    return (a',b')
```

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.

Como as listas representam uma estrutura importante em linguagens de programação funcional, a biblioteca já vem com a estratégia `parList` de tal forma que podemos fazer:

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

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

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.

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.

O que houve?

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

Vamos 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 rdeepseq
    chunks = chunksOf 1000 xss
```

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

# Exemplo: média

UFABC

*Total time 1.303s ( 0.749s elapsed)*

😊