



SOFTWARE · PRODUCTIVITY · GROUP



Threads e Variáveis Compartilhadas em Haskell

Fernando Castor

Professor Adjunto

Centro de Informática

Universidade Federal de Pernambuco



UNIVERSIDADE FEDERAL
DE PERNAMBUCO

cin.ufpe.br

Pode não parecer, mas...

- Linguagens funcionais são **ótimas** para programação **paralela!**
- Ausência de **efeitos colaterais...**
 - Diminui a necessidade de exclusão mútua
 - Em muitos casos, evita não-determinismo
 - Em geral, facilita a programação
- Não é mágica, porém
 - **Granularidade** faz muita diferença
 - Operações de E/S complicam as coisas

Como Haskell pode ajudar você

(a tornar paralelas suas aplicações)

• **Seis** abordagens para programação paralela (*and counting...*)

- Paralelismo semi-explícito
- Estratégias para execução paralela
- Threads com variáveis compartilhadas
 - (mais ou menos)
- *Threads* com memória transacional
- Paralelismo de dados
- Atores e passagem de mensagens



Como Haskell pode ajudar você

(a tornar paralelas suas aplicações)

• **Seis** abordagens para programação paralela (*and counting...*)

- Threads com variáveis compartilhadas
 - (mais ou menos)
- *Threads* com memória transacional
- Atores e passagem de mensagens



Threads em Haskell

- Por que usar *threads* explicitamente?
 - **Paralelismo semi-explicito** tem suas peculiaridades
 - **Maior controle** sobre criação e uso de *threads*
 - **Operações de E/S**
 - **Concorrência vs. Paralelismo**
- Threads => **variáveis compartilhadas**
 - E o “puramente funcional”?
 - Variante **um pouco mais segura** dessa abordagem

Criação de *threads*

- Pacote `Control.Concurrent`
- Duas formas:
 - `forkIO :: IO () -> IO ThreadId`
 - `forkOS :: IO () -> IO ThreadId`
- **Importante:**
 - O programa principal **não espera** pelo término das *threads* filhas
 - O programador deve cuidar disso

Comunicação entre *threads*

- Threads comunicam-se usando **variáveis compartilhadas**
 - Diferentes das de Java

Comunicação entre *threads*

- Threads comunicam-se usando **variáveis compartilhadas**
 - Diferentes das de Java
- Comunicação entre *threads* é **atômica**
 - Usando valores do tipo **MVar**
 - Pacote **Control.Concurrent**
 - Uma **MVar** é análoga a uma **fila bloqueante** com espaço para apenas um item
 - Alternativa: **Control.Concurrent.Chan**

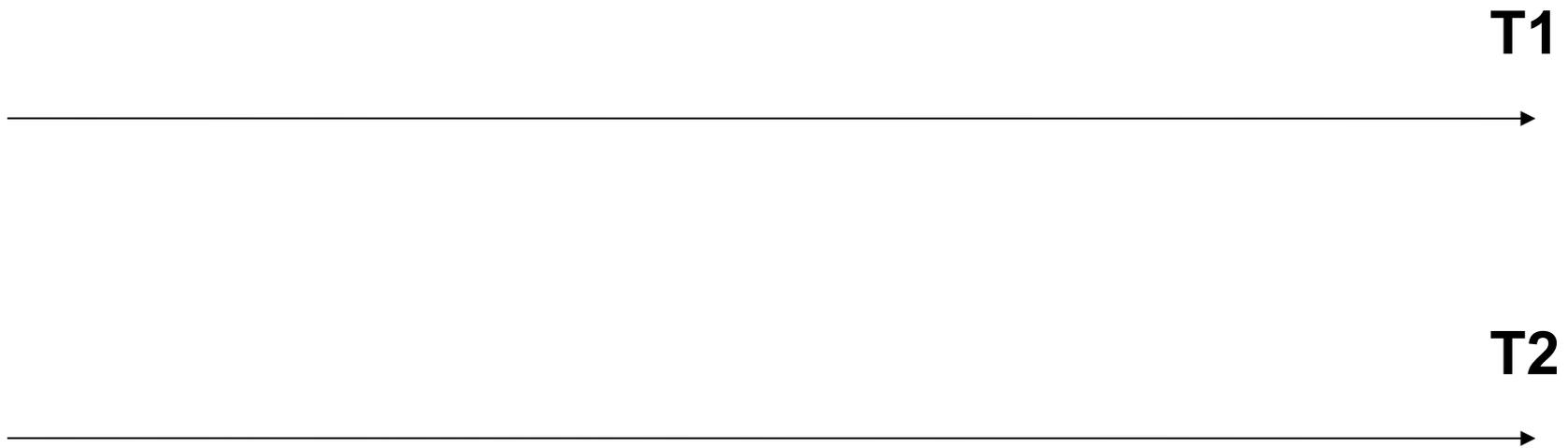
Funções sobre **MVar**'s

- **takeMVar** :: **MVar a -> IO a**
 - Se a MVar estiver vazia, **bloqueia a thread** até que um item seja colocado na MVar
 - Caso contrário,
 - **devolve seu conteúdo**, a esvazia e
 - **avisa** alguma *thread* que esteja **bloqueada** pela função **putMVar**
- **putMVar** :: **MVar a -> a -> IO ()**
 - **Coloca** um item em uma MVar (ver caso anterior)

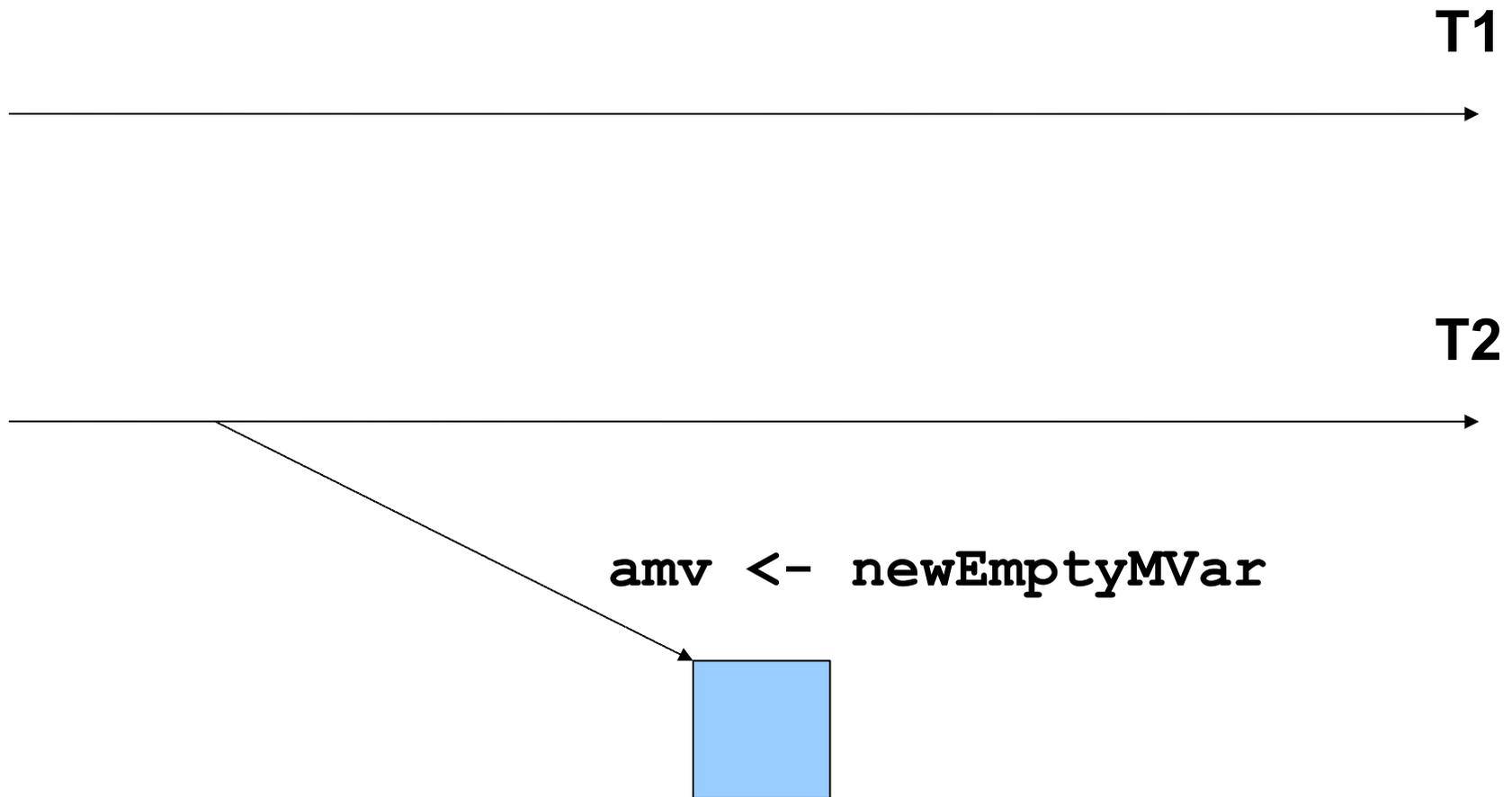
Mais funções sobre **MVar**'s

- `newEmptyMVar :: IO (MVar a)`
- `newMVar :: a -> IO (MVar a)`
- `readMVar :: MVar a -> IO a`
 - Devolve o elemento armazenado na **MVar**
 - **Bloqueia** caso esteja vazia
- `isEmptyMVar :: MVar a -> IO Bool`
- `tryTakeMVar :: MVar a -> IO (Maybe a)`
 - Versão não-bloqueante de **takeMVar**
- `tryPutMVar :: MVar a -> a -> IO Bool`

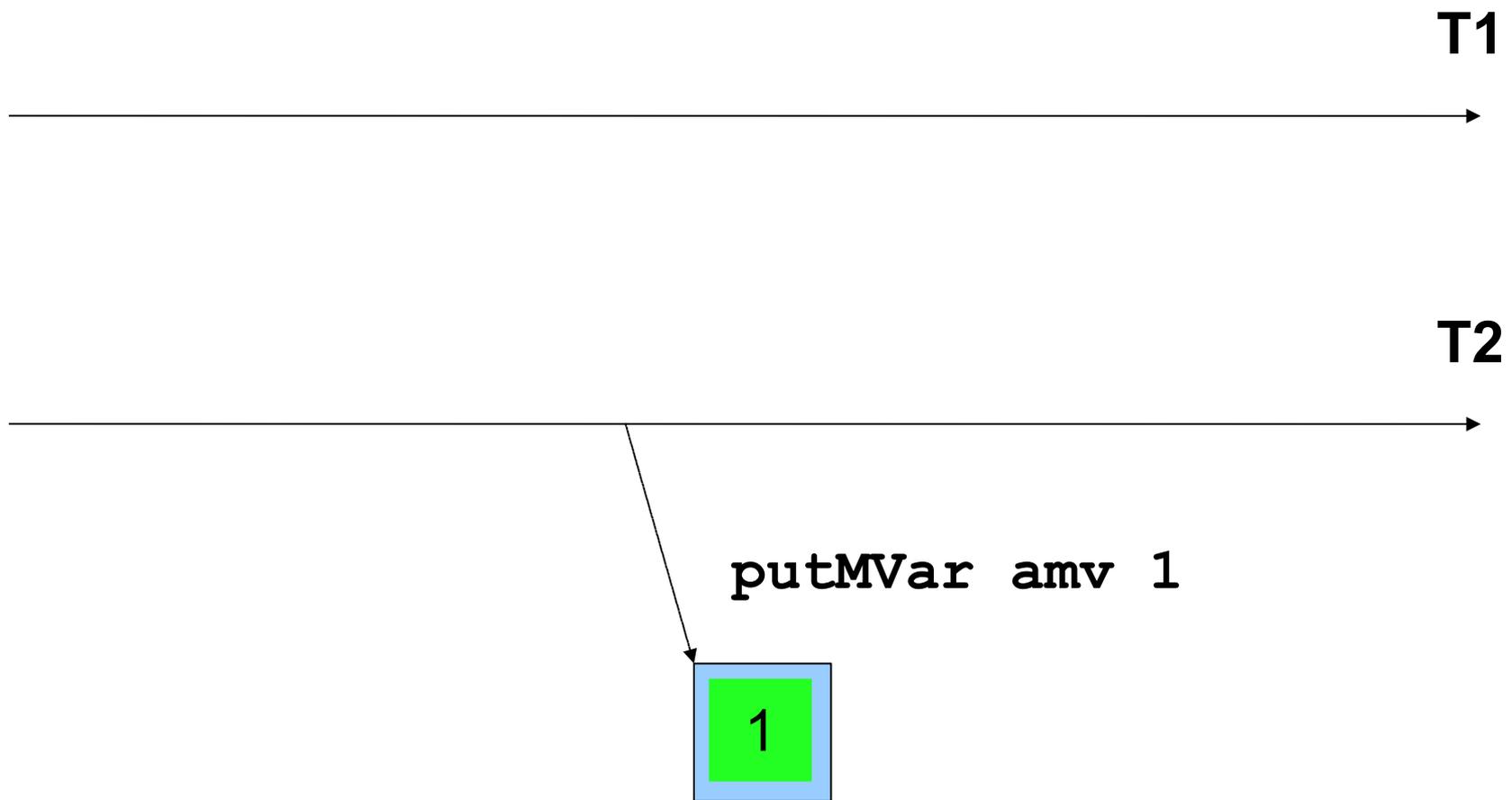
Uma conversa entre *threads*



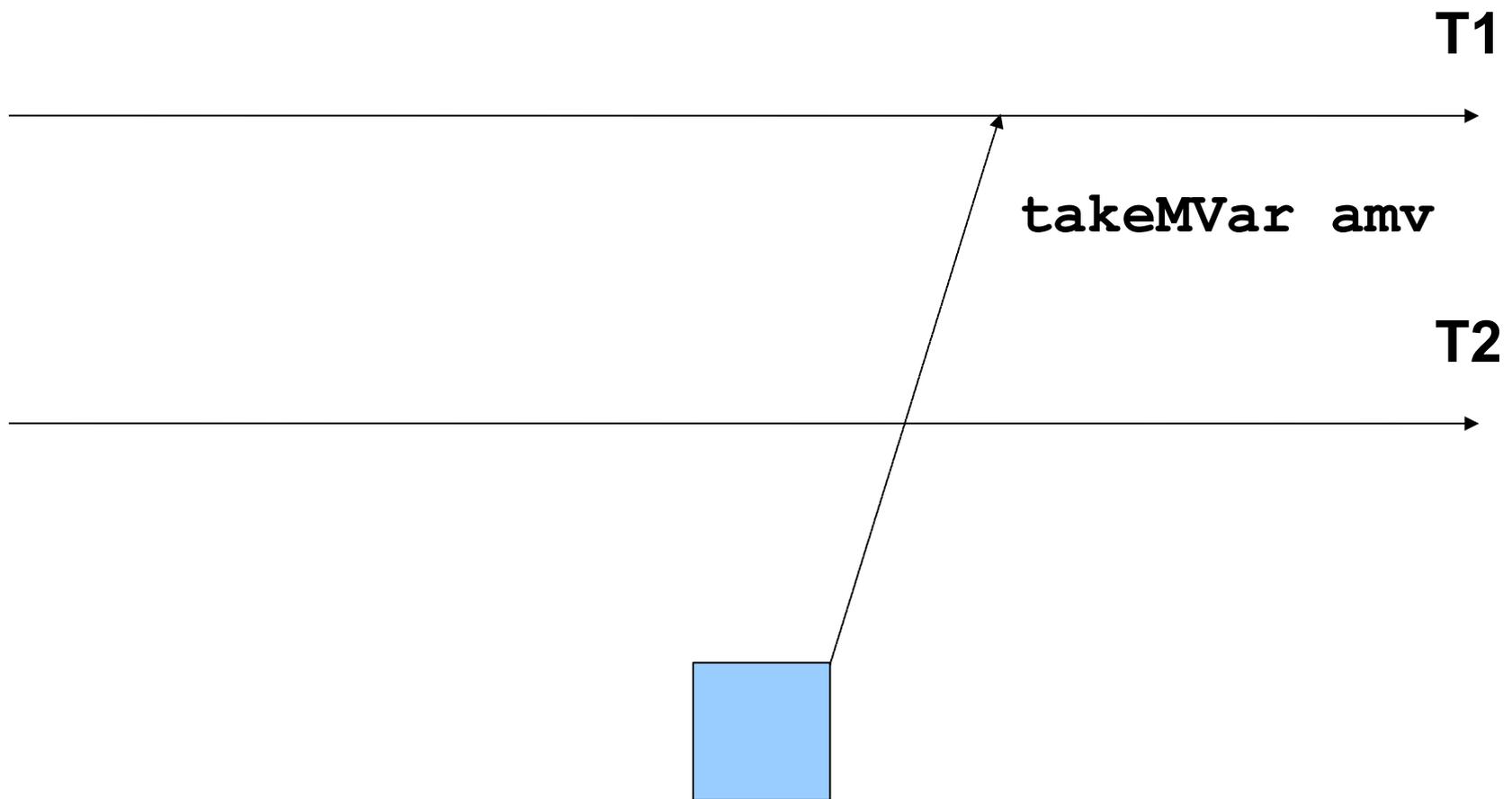
Uma conversa entre *threads*



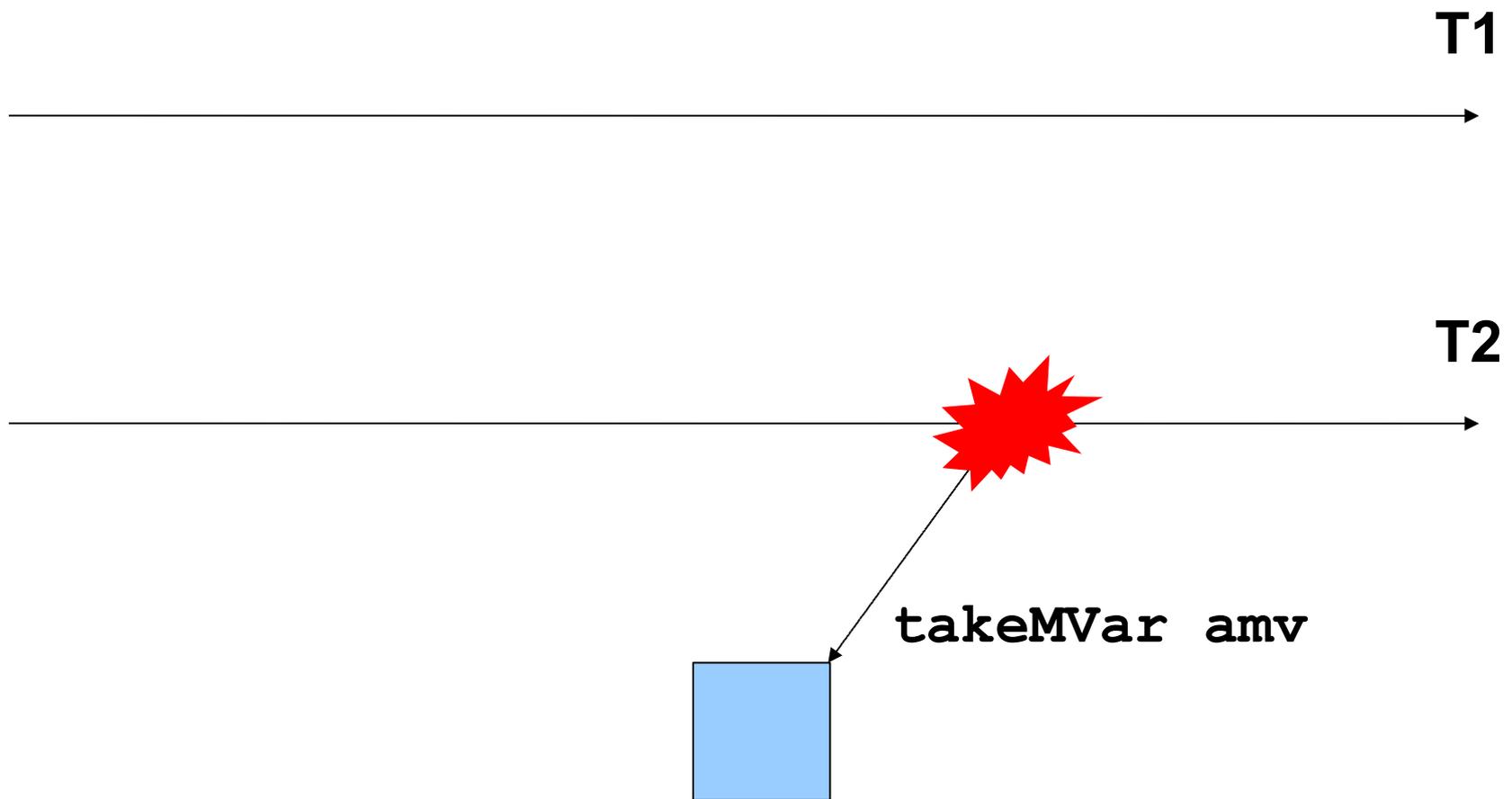
Uma conversa entre *threads*



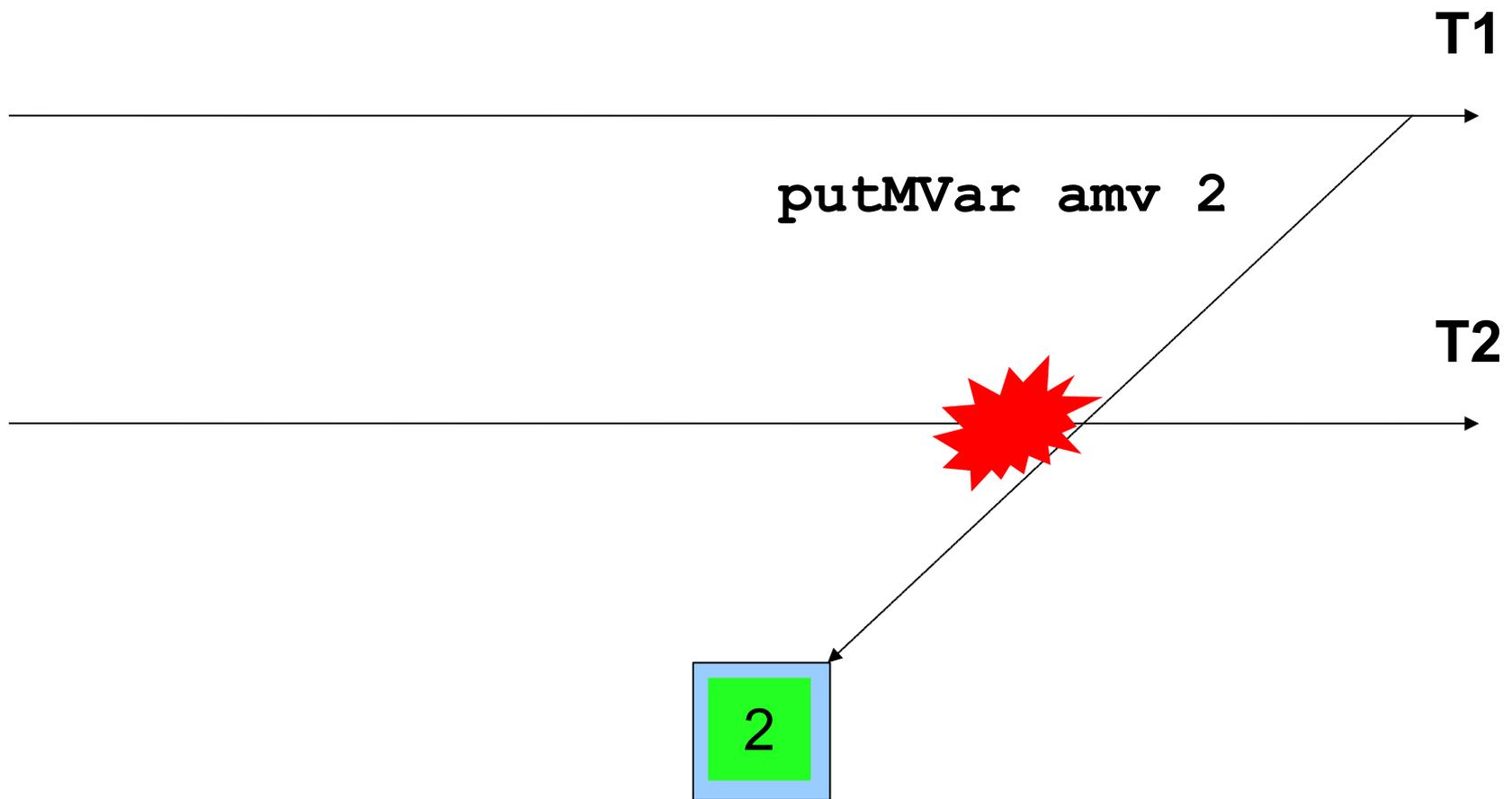
Uma conversa entre *threads*



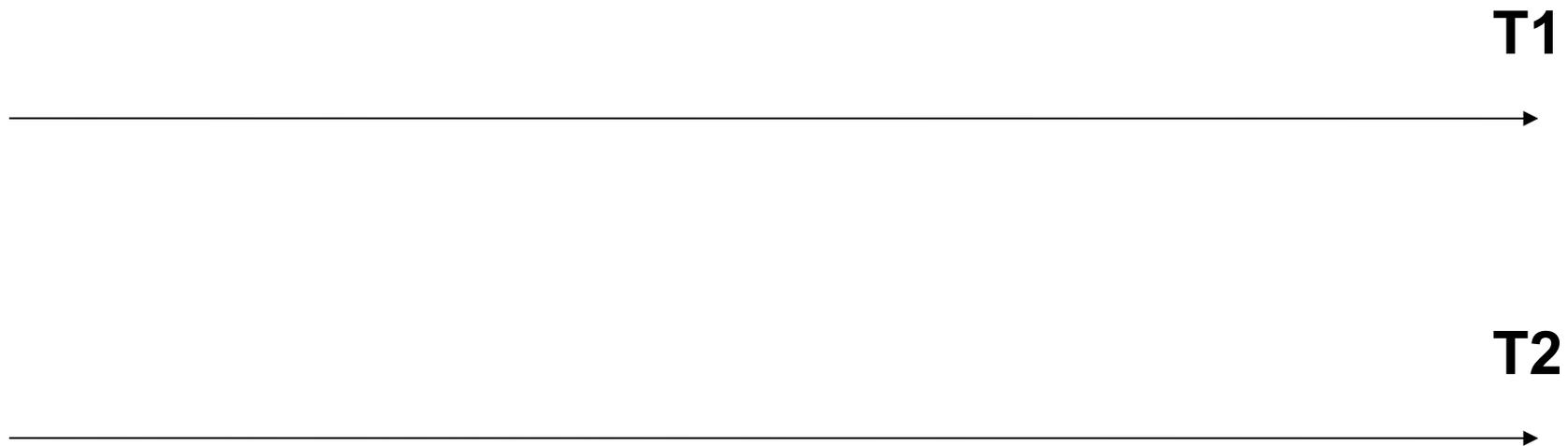
Uma conversa entre *threads*



Uma conversa entre *threads*

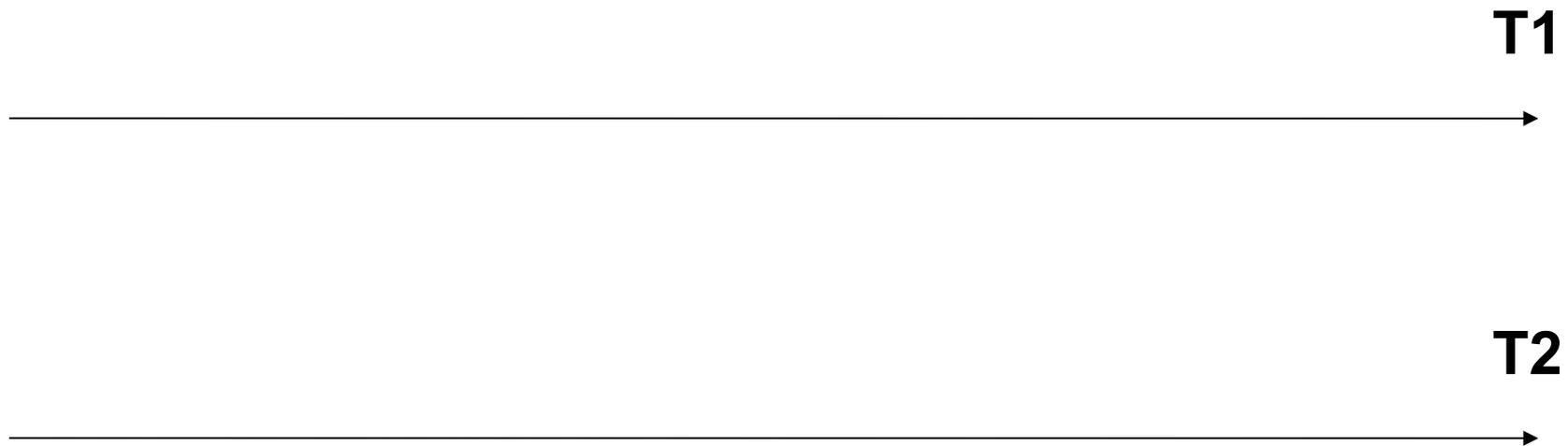


Uma conversa entre *threads*



T2 é desbloqueada

Uma conversa entre *threads*



T2 obtém o valor guardado em `amv`
(execução de `takeMVar` procede)

Exemplo 1 com *threads*

```
module Main where  
import Control.Concurrent  
import Control.Concurrent.MVar  
  
threadA :: MVar Float -> MVar Float -> IO ()  
threadA toSend toReceive  
  = do putMVar toSend 72  
      v <- takeMVar toReceive  
      putStrLn (show v)
```

Exemplo 1 com *threads* (cont.)

```
threadB::MVar Float->MVar Float->IO()
threadB toReceive toSend
  = do z <- takeMVar toReceive
      putMVar toSend (1.2 * z)

main::IO()
main = do aMVar <- newEmptyMVar
        bMVar <- newEmptyMVar
        forkIO (threadA aMVar bMVar)
        forkIO (threadB aMVar bMVar)
        threadDelay 1000
        -- espera um tempo (feio!)
```

Compilando programas paralelos e concorrentes

- GHC 6.12.*+
- Incluindo o pacote **Parallel** ou **Concurrent**
 - Depende do que você for fazer
 - Parte da Haskell Platform
 - Fácil de baixar com o Cabal Install

```
$ sudo cabal install parallel
```

- Compilando:
 - `ghc --make -rtsopts -threaded programa.hs`
- Executando:
 - `programa +RTS -N2 [-s]`

Um exemplo mais interessante: Fib+Euler

```
module Main where
import Control.Parallel
import Control.Concurrent
import Control.Concurrent.MVar

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

fibThread :: Int -> MVar Int -> IO ()
fibThread n resMVar = putMVar resMVar (fib n)
```

Um exemplo mais interessante (cont.)

```
mkList :: Int -> [Int]
```

```
mkList n = [1..n-1]
```

```
mkList :: Int -> [Int]
```

```
mkList n = [1..n-1]
```

```
relprime :: Int -> Int -> Bool
```

```
relprime x y = (gcd x y == 1)
```

```
euler :: Int -> Int
```

```
euler n = length (filter (relprime n) (mkList n))
```

```
sumEuler :: [Int] -> Int
```

```
sumEuler l = sum $ map euler l
```

Revisitando Fib+Euler (cont.)

```
s1 :: Int
s1 = sumEuler (mkList 4750)

main :: IO ()
main
  = do putStrLn "explicit SumFibEuler"
      fibResult <- newEmptyMVar
      forkIO (fibThread 37 fibResult)
      pseq s1 (return ())
      f <- takeMVar fibResult
      putStrLn ("sum: " ++ show (s1+f))
```

Como é o **desempenho** neste caso?

Melhorando Fib+Euler

Problema: `fib n` não está sendo avaliada na outra *thread*

- Uma solução **parcial**:

```
fibThread :: Int -> MVar Int -> IO ()
```

```
fibThread n resMVar =  
    pseq f (putMVar resMVar f)  
    where f = fib n
```

Desempenho pode **variar**, dependendo do hardware, dos dados e das configurações usadas!

A alteração acima ainda **NÃO É** suficiente...

O Bom e Velho Contador

```
waitThreads :: MVar Int -> IO ()
waitThreads fim =
  do f <- takeMVar fim
     if (f > 0) then
       do putMVar fim f
          waitThreads fim
     else
       return ()

main :: IO ()
main = do contador <- newMVar 0
        fim <- newMVar 2
        forkIO (oper (+) contador fim 100000)
        forkIO (oper (-) contador fim 100000)
        waitThreads fim
        return ()
```

O Bom e Velho Contador ERRADO

```
module Main where
import Control.Concurrent
import Control.Concurrent.MVar

oper :: (Int->Int->Int)->MVar Int->MVar Int->Int->IO()
oper op cont fim 0
    = do v <- takeMVar cont
         putStrLn (show v)
         f <- takeMVar fim
         putMVar fim (f-1)
oper op cont fim num
    = do v <- takeMVar cont
         putMVar cont (op v 1)
         oper op cont fim (num-1)
```

Código rodando...

O Bom e Velho Contador ERRADO

```
module Main where
import Control.Concurrent
import Control.Concurrent.MVar

oper :: (Int->Int->Int)->MVar Int->MVar Int->Int->IO()
oper op cont fim 0
    = do v <- takeMVar cont -- remove mas não recoloca!
        putStrLn (show v)
        f <- takeMVar fim
        putMVar fim (f-1)
oper op cont fim num
    = do v <- takeMVar cont
        putMVar cont (op v 1)
        oper op cont fim (num-1)
```

Corrigido o problema, funciona com *inúmeras threads!*

Configurando alguns parâmetros do *runtime*

```
$ programa +RTS -N2 -K50M -H300M
```

—**-K50M** : 50 MB alocados para a pilha

—**-H300M** : 300 MB alocados para o *heap*

- *Heap*: região da memória usada para variáveis com **tempo de vida arbitrário**

—Essas opções evitam que ocorra um excesso de coleta de lixo

- Caro computacionalmente