Souběžnost a paralelismus

IB016 Seminář z funkcionálního programování

Fakulta informatiky, Masarykova univerzita

Souběžnost a paralelismus

Souběžnost (concurrency)

- spuštění více výpočtů, které se překrývají v čase
- **nemusí být spuštěné zároveň** (můžou se střídat na jednom procesoru)
- výpočty nemusí být nezávislé, můžou komunikovat
- výpočty nemají určené pořadí
- účel = systém může dělat víc věcí (např. kreslit na obrazovku, přehrávat hudbu a komunikovat s uživatelem)

Paralelismus (parallelism)

- spuštění více výpočtů nebo podvýpočtů **zároveň** (na více procesorech, grafických kartách, výpočetních uzlech, ...)
- výpočty nemusí být nezávislé, můžou komunikovat
- výpočty nemají určené pořadí
- účel = zvýšení výkonu

Paralelismus v čistém kódu

Paralelismus v čistém kódu

Pozorování: Čisté funkce je jednoduché paralelizovat

- nemění a nečtou sdílený stav
- nemají vedlejší efekty
- jsou referenčně transparentní
- $\Rightarrow$ libovolné čisté funkce jde pustit paralelně a nezmění se výsledek
- $\Rightarrow$ libovolné čisté výrazy jde pustit paralelně a nezmění se výsledek

Paralelismus v čistém kódu

Pozorování: Čisté funkce je jednoduché paralelizovat

- nemění a nečtou sdílený stav
- nemají vedlejší efekty
- jsou referenčně transparentní
- $\Rightarrow$ libovolné čisté funkce jde pustit paralelně a nezmění se výsledek
- $\Rightarrow$ libovolné čisté výrazy jde pustit paralelně a nezmění se výsledek

Použití:

- stačí překladači říct, co **chceme** pustit paralelně
- nemusíme řešit, jestli to bude korektní, protože zaručeně bude!

Připomenutí lenost a vynucení vyhodnocení

Každý výraz v Haskellu je nejprve nevyhodnocený *thunk*

Připomenutí lenost a vynucení vyhodnocení

Každý výraz v Haskellu je nejprve nevyhodnocený *thunk*

Thunky se vyhodnocují, když jsou potřeba v nějakém vzoru, vestavěné funkci, nebo

Připomenutí lenost a vynucení vyhodnocení

Každý výraz v Haskellu je nejprve nevyhodnocený *thunk*

Thunky se vyhodnocují, když jsou potřeba v nějakém vzoru, vestavěné funkci, nebo pomocí funkce `seq`.

`seq`

- volání `seq a b` vyhodnotí `a` a vrátí výsledek `b`
- `a` se vyhodnotí do *weak head normal form (WHNF)*, tj. po první hodnotový konstruktor, lambda abstrakci nebo vestavěnou funkci s nedostatkem argumentů
- `a` **nemusí být** vyhodnoceno před `b`, pokud překladač usoudí, že by to zhoršilo výkon

Pro úplné vyhodnocení je potřeba použít funkce `deepseq` nebo `force` z modulu **`Control.DeepSeq`**

Paralelní kombinátory

Na paralelismus v čistém kódu stačí kombinátory z modulu `Control.Parallel`

`par`

- volání `par a b` řekne překladači, že `a` může vyhodnotit paralelně s počítáním výsledku `b`
- vrátí výsledek `b`

`pseq`

- volání `pseq a b` vyhodnotí `a` a poté vrátí výsledek `b`
- `a` **zaručeně bude** vyhodnoceno před `b`, na rozdíl od `seq a b`

Paralelní kombinátory

Na paralelismus v čistém kódu stačí kombinátory z modulu **`Control.Parallel`**

`par`

- volání `par a b` řekne překladači, že `a` může vyhodnotit paralelně s počítáním výsledku `b`
- vrátí výsledek `b`

`pseq`

- volání `pseq a b` vyhodnotí `a` a poté vrátí výsledek `b`
- `a` **zaručeně bude** vyhodnoceno před `b`, na rozdíl od `seq a b`

Tradiční použití

- komplikovaný výraz `a`, komplikovaný výraz `b`, jejich kombinace `foo a b`
- `a `par` (b `pseq` foo a b)`
- k zamyšlení: proč je potřeba `pseq` a nestačí `a `par` foo a b`?

Příklad

```haskell
import Control.Parallel
import System.Environment

isPrime :: Integer -> Bool
isPrime n = all (\d -> n `mod` d /= 0)  [2..(n-1)]

isTwinPrime :: Integer -> Bool
isTwinPrime n = prime1 `par` (prime2 `pseq` prime1 && prime2)
  where
    prime1 = isPrime n
    prime2 = isPrime (n+2)

main :: IO ()
main = do
  [n] <- getArgs
  let result = isTwinPrime (read n)
  print result
```

Kompilace a spuštění

Kompilace

- ve výchozím nastavení kompilátor používá běhové prostředí (runtime), které celé běží v jednom vlákně OS → nemůže použít více procesorů
- je potřeba kompilovat s přepínačem `-threaded`

Spuštění

- počet procesorů, které může aplikace použít, se zadává při spuštění programu pomocí *runtime options* (RTS)
- RTS se předávají programu pomocí „`+RTS nastavení -RTS`“ a nebudou součástí `getArgs`
- konkrétně pro počet jader procesoru „`+RTS -N4 -RTS`“ pro 4 jádra

Kompilace a spuštění

Program zkompilujeme pomocí

```
ghc --make -threaded par.h
```

A spustíme

```
> time ./par 1048571 +RTS -N1 -RTS
True
-----------------------------------------------------
Executed in  134.72 milli

> time ./par 1048571 +RTS -N2 -RTS
True
-----------------------------------------------------
Executed in   74.85 millis
```

Vyhodnocení složitějších struktur

```haskell
getPrimes :: Integer -> [Integer]
getPrimes n = firstHalf `par` (secondHalf `pseq` firstHalf ++ secondHalf)
  where middle = n `div` 2
        firstHalf = filter isPrime [1..middle]
        secondHalf = filter isPrime [middle..n]
```

Spuštění se 2 jádry moc nepomůže, proč?

Vyhodnocení složitějších struktur

```haskell
getPrimes :: Integer -> [Integer]
getPrimes n = firstHalf `par` (secondHalf `pseq` firstHalf ++ secondHalf)
  where middle = n `div` 2
        firstHalf = filter isPrime [1..middle]
        secondHalf = filter isPrime [middle..n]
```

Spuštění se 2 jádry moc nepomůže, proč?

Funkce `par` vyhodnocuje jen do WHNF!

Vyhodnocení složitějších struktur

```haskell
getPrimes :: Integer -> [Integer]
getPrimes n = firstHalf `par` (secondHalf `pseq` firstHalf ++ secondHalf)
  where middle = n `div` 2
        firstHalf = filter isPrime [1..middle]
        secondHalf = filter isPrime [middle..n]
```

Spuštění se 2 jádry moc nepomůže, proč?

Funkce `par` vyhodnocuje jen do WHNF!

Je potřeba použít `force`:

```haskell
getPrimes n =
    force firstHalf `par` (force secondHalf `pseq` firstHalf ++ secondHalf)
```

Souběžnost `IO` akcí

Spouštění souběžných IO akcí

- paralelismus v čistém kódu ne vždy stačí
- je potřeba spouštět více vstupně/výstupních akcí, které existují současně a mohou komunikovat a měnit stav světa
- podpora v Haskellu v modulu **`Control.Concurrent`**

```
forkIO :: IO () -> IO ThreadId
```

- spustí zadaný výpočet v separátním vlákně
- používá vlastní odlehčenou implementaci vláken běhového prostředí Haskellu (*green threads*), ne vlákna OS
- běží souběžně, ale **nemusí běžet paralelně**, záleží na počtu procesorů

Spouštění souběžných IO akcí

- paralelismus v čistém kódu ne vždy stačí
- je potřeba spouštět více vstupně/výstupních akcí, které existují současně a mohou komunikovat a měnit stav světa
- podpora v Haskellu v modulu `Control.Concurrent`

```
forkIO :: IO () -> IO ThreadId
```

- spustí zadaný výpočet v separátním vlákně
- používá vlastní odlehčenou implementaci vláken běhového prostředí Haskellu (*green threads*), ne vlákna OS
- běží souběžně, ale **nemusí běžet paralelně**, záleží na počtu procesorů

```
forkFinally :: IO a -> (Either SomeException a -> IO ()) -> IO ThreadId
```

- spustí zadaný výpočet hodnoty typu `a` v separátním vlákně
- až výpočet skončí, spustí zadanou funkci s výslednou hodnotou (nebo výjimkou, pokud nastala)

Spouštění souběžných IO akcí: příklad

```haskell
import Control.Concurrent

main = do
  forkIO (putStrLn "hello")
  forkIO (putStrLn "concurrent")
  putStrLn "world"
```

Komunikace pomocí MVar

Různá vlákna spolu můžou komunikovat přes synchronizované proměnné.

MVar a

- sdílená proměnná, kterou je možné číst a zapisovat z více vláken zároveň
- je buď prázdná, nebo obsahuje hodnotu typu a
- jde číst a měnit jen uvnitř **IO**

Komunikace pomocí MVar

`newEmptyMVar :: IO (MVar a)`

- vytvoří prázdnou **MVar**

`newMVar :: a -> IO (MVar a)`

- vytvoří **MVar** s danou hodnotou

`takeMVar :: MVar a -> IO a`

- vyprázdní **MVar** a vrátí její hodnotu
- pokud je **MVar** prázdná, zablokuje vlákno, dokud do ní někdo nezapíše

`putMVar :: MVar a -> a -> IO ()`

- nastaví **MVar** na zadanou hodnotu
- pokud **MVar** není prázdná, zablokuje vlákno, dokud ji někdo nevyprázdní

MVar: Příklad

```haskell
import Control.Concurrent

expensiveIOComputation :: MVar Int -> IO ()
expensiveIOComputation mv = do
  putStrLn "Starting expensive computation"
  threadDelay 5000000
  putMVar mv 42

main :: IO ()
main = do
  mv <- newEmptyMVar
  forkIO (expensiveIOComputation mv)
  putStrLn "Waiting for result"
  result <- takeMVar mv
  putStrLn ("Result: " ++ show result)
```

Hlavní a vedlejší vlákna

Vlákno, ve kterém byla spuštěna funkce `main`, je hlavní:

- pokud skončí, skončí celý program
- na ostatní vlákna se nečeká

Vlákna spuštěná pomocí `forkIO` jsou vedlejší:

- pokud skončí hlavní vlákno, vedlejší vlákna se zabijí

Hlavní a vedlejší vlákna

Vlákno, ve kterém byla spuštěna funkce `main`, je hlavní:

- pokud skončí, skončí celý program
- na ostatní vlákna se nečeká

Vlákna spuštěná pomocí `forkIO` jsou vedlejší:

- pokud skončí hlavní vlákno, vedlejší vlákna se zabijí

Příklad:

```
main = do
  forkIO (putStrLn "hello")
  forkIO (putStrLn "concurrent")
  putStrLn "world"
```

Jak zaručit vypsání všech tří řetězců?

Neblokující operace s MVar

```
tryTakeMVar :: MVar a -> IO (Maybe a)
```

- pokud je **MVar** prázdná, vrátí **Nothing**
- pokud není, **MVar** vyprázdní a vrátí **Just** hodnota

```
tryPutMVar :: MVar a -> a -> IO Bool
```

- pokud je **MVar** prázdná, zapíše do ní a vrátí **True**
- pokud není prázdná, vrátí **False**

```
isEmptyMVar :: MVar a -> IO Bool
```

- vrátí **True**, pokud je **MVar** prázdná; jinak vrátí **False**

Neblokující operace: příklad

```haskell
expensiveIOComputation :: MVar Int -> IO ()
expensiveIOComputation mv = do
  putStrLn "Starting expensive computation"
  threadDelay 5000000
  putMVar mv 42

spinAndPrint :: MVar Int -> IO ()
spinAndPrint mv = do
  putStrLn "Waiting for result"
  isEmpty <- isEmptyMVar mv
  if isEmpty
    then threadDelay 1000000 >> spinAndPrint mv
    else takeMVar mv >>= \res -> putStrLn ("Result: " ++ show res)

main :: IO ()
main = do
  mv <- newEmptyMVar
  forkIO (expensiveIOComputation mv)
  spinAndPrint mv
```

Neblokující operace: příklad

```haskell
expensiveIOComputation :: MVar Int -> IO ()
expensiveIOComputation mv = do
  putStrLn "Starting expensive computation"
  threadDelay 5000000
  putMVar mv 42

spinAndPrint :: MVar Int -> IO ()
spinAndPrint mv = do
  putStrLn "Waiting for result"
  isEmpty <- isEmptyMVar mv
  if isEmpty
    then threadDelay 1000000 >> spinAndPrint mv
    else takeMVar mv >>= \res -> putStrLn ("Result: " ++ show res)

main :: IO ()
main = do
  mv <- newEmptyMVar
  forkIO (expensiveIOComputation mv)
  spinAndPrint mv
```

Je taková implementace `spinAndPrint` vždy bezpečná?

Nástrahy souběžných programů: uváznutí, souběh

Programování souběžných programů je těžké.

Uváznutí (deadlock)

- několik vláken na sebe čeká navzájem; nikdy se neprobudí
- např.
  - vlákno A čeká na `MVar`, do které zapisuje jen vlákno B
  - vlákno B čeká na `MVar`, do které zapisuje jen vlákno A

Souběh (race condition)

- chování/výsledek programu závisí na tom, v jakém pořádí se vykonaly souběžné výpočty
- typicky když víc vláken zapisuje do jedné proměnné

Nástrahy souběžných programů: příklad

```haskell
increaseBoth :: MVar Int -> MVar Int -> IO ()
increaseBoth v1 v2 = do
  val1 <- takeMVar v1
  val2 <- takeMVar v2
  putMVar v1 (val1 + 1)
  putMVar v2 (val2 + 1)

main :: IO ()
main = do
  v1 <- newMVar 1
  v2 <- newMVar 2
  forkIO (increaseBoth v1 v2)
  increaseBoth v2 v1
  readMVar v1 >>= print
  readMVar v2 >>= print
```

Nástrahy souběžných programů: příklad

```haskell
updateMVar :: MVar a -> IO ()
updateMVar mv = do
  val <- takeMVar mv
  let newVal = computeSomething val
  putMVar mv newVal
```

Vidíte nějaký problém?

Nástrahy souběžných programů: příklad

```haskell
updateMVar :: MVar a -> IO ()
updateMVar mv = do
  val <- takeMVar mv
  let newVal = computeSomething val
  putMVar mv newVal
```

Vidíte nějaký problém?

```haskell
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
```

- upraví hodnotu `MVar` zadanou funkcí
- pokud funkce vyhodí výjimku, **hodnotu nezmění**

Komunikace pomocí kanálů

```
newChan :: IO (Chan a)
```

- vytvoří prázdný kanál

```
writeChan :: Chan a -> a -> IO ()
```

- zapíše na konec kanálu zadanou hodnotu

```
readChan :: Chan a -> IO a
```

- vyjme z kanálu první hodnotu a vrátí ji
- pokud je kanál prázdný, zablokuje vlákno, dokud do kanálu někdo nezapíše

Kanály: příklad

```haskell
import Control.Concurrent

main :: IO ()
main = do
  ch <- newChan
  forkIO (writeChan ch 1 >> writeChan ch 2)
  forkIO (writeChan ch 3)
  readChan ch >>= print
  readChan ch >>= print
  readChan ch >>= print
```

Kanály: příklad

```haskell
import Control.Concurrent
import Control.Monad

expensiveComputation = threadDelay 100000
lessExpensiveComputation = threadDelay 10000

producer :: Chan Int -> IO ()
producer ch = mapM_ (\v -> lessExpensiveComputation >> writeChan ch v) [1..100]

consumer :: String -> Chan Int -> IO ()
consumer n ch = forever $ do
  val <- readChan ch
  expensiveComputation
  putStrLn (n ++ ": " ++ show val)

main :: IO ()
main = do
  chan <- newChan
  forkIO (producer chan)
  --forkIO (consumer "helper" chan)
  consumer "main" chan
```

Pokročilejší témata

Programování souběžných programů je těžké. Opravdu.

Existuje knihovna Software Transactional Memory (`stm`), která umožňuje:

- definovat paralelní výpočty na vyšší úrovni abstrakce,
- jednoduše je skládat pomocí monád a kombinátorů (např. `orElse`)
- vyrábět atomické výpočty pomocí transakcí: buď se výpočet úspěšně dokončí celý, nebo nezmění stav (kombinátor `atomically`)

A to je vše

Kam dál?

- Zajímavé monády k zamyšlení
  - `Cont`, `Tardis`, `Logic`, LGBT
  - *Freer monads* (monády s rozšiřitelnými, jemnějšími efekty)
  - dláždivý správce oken `xmonad`
- IA014 Advanced Functional Programming
  - λ-kalkul do hloubky
  - vystavění Haskellu z λ-kalkulu
  - GADT (generalisované algebraické datové typy)
  - závislé typy

A to je vše

Kam dál?

- Zajímavé monády k zamyšlení
  - `Cont`, `Tardis`, `Logic`, LGBT
  - *Freer monads* (monády s rozšiřitelnými, jemnějšími efekty)
  - dláždivý správce oken `xmonad`
- IA014 Advanced Functional Programming
  - λ-kalkul do hloubky
  - vystavění Haskellu z λ-kalkulu
  - GADT (generalisované algebraické datové typy)
  - závislé typy
- Chcete bakalářskou práci na téma související s funkcionálním programováním?
  - Ozvěte se. :-)

Samostatné programování

Paralelizace prvočísel

Upravte funkci `getPrimes`, aby seznam rozdělila na 4 části, a zkuste spustit výpočet s 1 až 4 jádry procesoru.

Pro odvážné: Upravte funkci `getPrimes`, aby jako argument brala i počet seznamů, na které má vstup rozdělit a má je počítat paralelně. Zkuste chování s různými počty jader.

Paralelní `map`

Napište paralelní verzi funkce `map`, která funkci aplikuje paralelně. Zkuste vyhodnotit zrychlení pomocí funkce `isPrime` a seznamu velkých čísel.

Kanály

Upravte příklad s kanály a výpočty `producer` a `consumer`, aby

- producent signalizoval konec vstupu
- všichni konzumenti při dosažení konce vstupu skončili výpočet (můžou o tom i informovat uživatele zprávou na standardní výstup)

Nápověda: Změňte typ kanálu na **`Chan (Maybe Int)`**.

Souběžné kopírování souboru

Pomocí funkce `forkIO` a typu **`MVar`** napište program, který

- v jednom vlákně čte řádky z jednoho souboru,
- ve druhém vlákně tyto řádky souběžně zapisuje do jiného souboru.