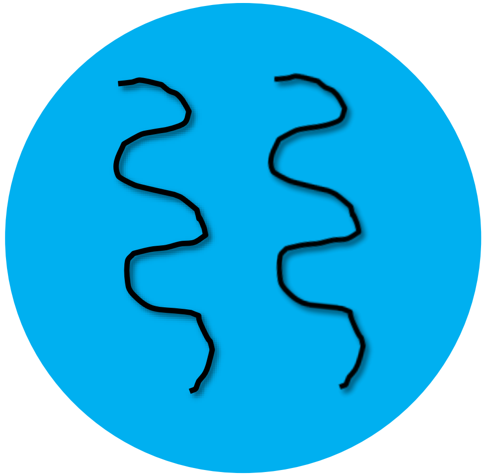
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

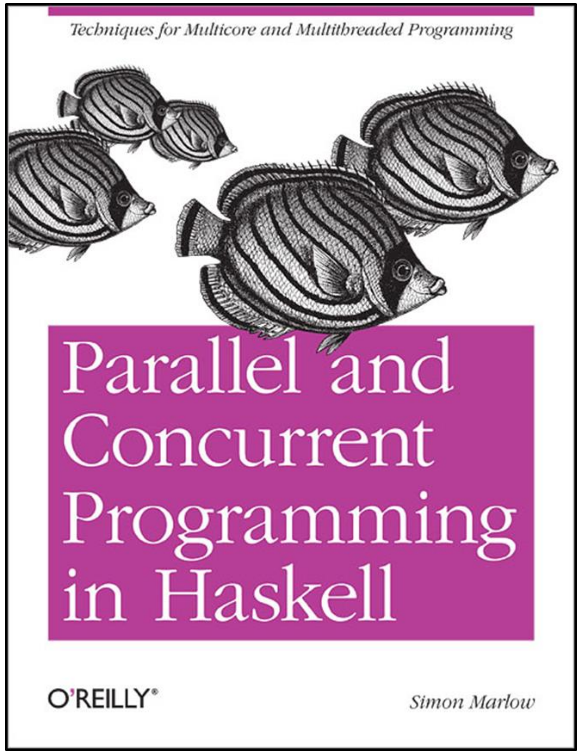
Concurenta

Threaduri

Memorie Partajata

Ioana Leustean

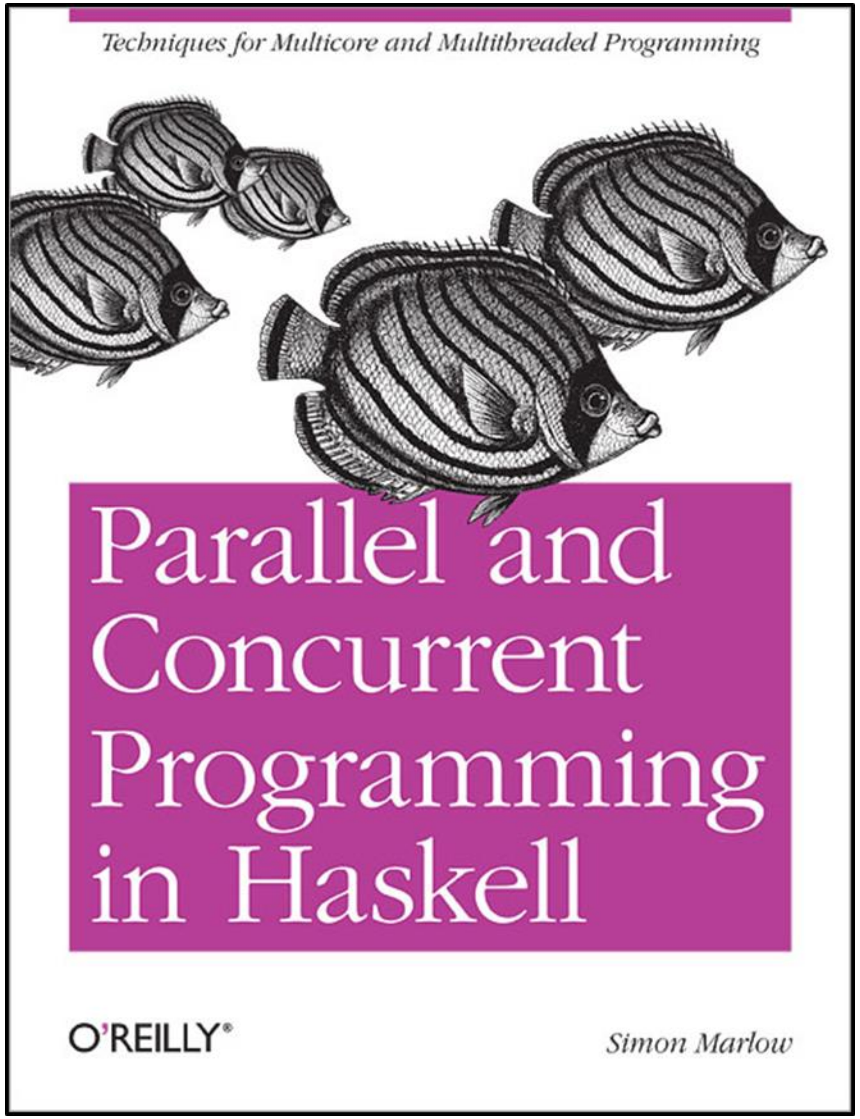




[Part II. Concurrent Haskell](#)
Cap.7 & 8

"Haskell does not take a stance on which concurrent programming model is best: actors, shared memory, and transactions are all supported, for example."

"Haskell provides all of these concurrent programming models and more - but this flexibility is a double-edged sword. The advantage is that you can choose from a wide range of tools and pick the one best suited to the task at hand, but the disadvantage is that it can be hard to decide which tool is best for the job."
S. Marlow



➢ Thread-urile in Haskell:

Thread-urile au efecte si interactioneaza cu lumea exterioara.

Programarea concurenta in Haskell are loc in monada IO.

La rulare, efectele thread-urilor sunt intercalate nedeterminist.

Thread-urile in Haskell sunt create si gestionate intern,
fara a folosi facilitati specifice sistemului de operare.

Implementarea threadurilor asigura verificarea anumitor conditii de corectitudine (fairness)

➢ Crearea thread-urilor

forkIO :: IO () -> IO ThreadId

```
Prelude> :m + Control.Concurrent
Prelude Control.Concurrent> :t forkIO
forkIO :: IO () -> IO ThreadId
```

```
import Control.Concurrent
import Control.Monad

main = do
        forkIO (replicateM_ 100 (putChar 'A'))  -- child thread
        replicateM_ 100 (putChar 'B')  -- main thread
        putStrLn " "
```

[replicateM](https://www.haskell.org/hoogle/)

```
Prelude> :m + Control.Monad
Prelude Control.Monad> :t replicateM_
replicateM_ :: Monad m => Int -> m a -> m ()
Prelude Control.Monad> replicateM_ 5 (putStrLn "A")
A
A
A
A
A
```

forkIO :: IO () -> IO ThreadId

```
import Control.Concurrent
import Control.Monad

main = do
        forkIO (replicateM_  100 (putChar 'A'))   -- child thread
        replicateM_ 100 (putChar 'B')  -- main thread
        putStrLn " "
```

```
*Main Control.Monad> main
ABABABABABABABABABABABABABABABABABABABABABABABAB.
BABABABABABABABABABABABABABABABAB
```

La rulari diferite se pot obtine rezultate diferite!

forkIO :: IO () -> IO ThreadId

```
import Control.Concurrent
import Control.Monad

main = do
        forkIO (replicateM_  100 (putChar 'A'))   -- child thread
        replicateM_ 100 (putChar 'B')  -- main thread
        putStrLn " "
```

Daca fisierul se numeste *thread.hs* atunci el poate fi compilat si executat astfel

>ghc thread.hs -threaded
> ./thread

```
PS C:\Users\igleu\Documents\DIR\ICLP22\Curs-2022\Haskell22\pgh\haskell2022> ./thread
BBBBABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABAA
AA
PS C:\Users\igleu\Documents\DIR\ICLP22\Curs-2022\Haskell22\pgh\haskell2022> ./thread
BBBBABABABABABABABABABABABABABABABABABABABABABABABABABABAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBB
```

La rulari diferite se pot obtine rezultate diferite!

forkIO :: IO () -> IO ThreadId

myThreadId :: IO ThreadId

print x = putStrLn (show x)

```
import Control.Concurrent
import Control.Monad

main = do
        forkIO (replicateM_  100 (myThreadID >>= print))   -- child thread
        replicateM_ 100 (myThreadID >>= print)  -- main thread
        putStrLn " "
```

```
PS C:\Users\igleu\Documents\DIR\ICLP22\Curs-2022\Haskell22\pgh\haskell2022> ./threadID
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
ThreadId 3
ThreadId 4
```

"The computation passed to forkIO is executed in a new thread that runs concurrently with the other threads in the system. If the thread has effects, those effects will be interleaved in an indeterminate fashion with the effects from other threads."
S. Marlow, PCPH

"forkIO is assymetrical: when a process executes a forkIO it spawns a child process that executes concurrently with the continued execution of the parent"
SL Peyton Jones, A Gordon, S Finne, Concurrent Haskell

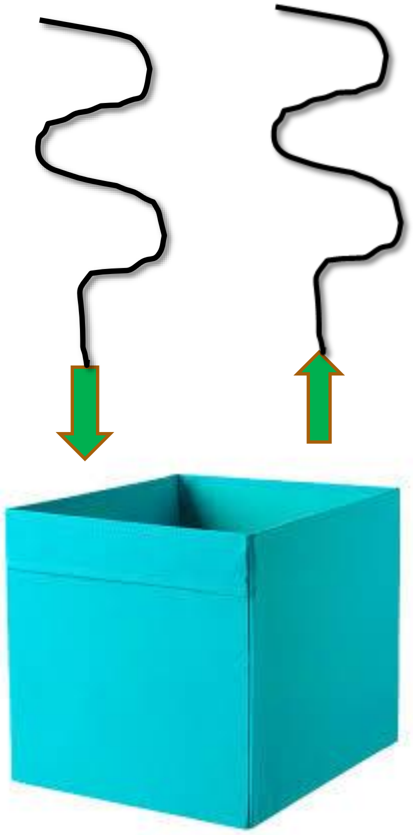
"**GHC's runtime system treats the program's original thread of control differently from other threads**.
When this thread finishes executing, the runtime system considers the program as a whole to have completed.
If any other threads are executing at the time, they are terminated."
B. O'Sullivan, D. Stewart, J. Goerzen, Real World Haskell

➢ Interleaving

```
import Control.Concurrent
import Control.Monad

myread1 = do
        putStrLn "thread1"
        s<- getLine
        putStrLn $ "citit 1: " ++ s

myread2 = do
        putStrLn "thread2"
        s<- getLine
        putStrLn $ "citit 2:" ++ s


main = do
          forkIO (replicateM_  10 myread1)
          replicateM_ 10 myread2
```

```
*Main> main
thread1
thread2
e
citit 1: e
thread1
s
citit 2:s
thread2
r
citit 1: r
thread1
e
citit 2:e
thread2
f
citit 1: f
thread1
f
```

➢ Comunicarea thread-urilor



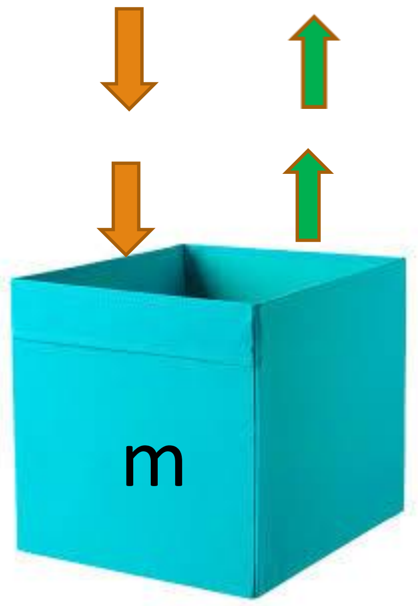
MVar

mutable variable

➢ Comunicarea folosind MVar se face in monada IO

data MVar a

- o data de tipul MVar a reprezinta o locatie mutabila care poate fi goala sau
- poate contine o singura valoare de tip a
- thread-urile pot comunica prin intermediul datelor de tip MVar



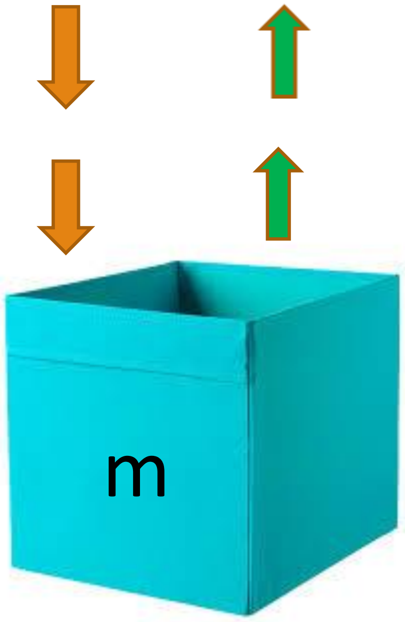
m :: MVar a

poate fi vazuta ca:

- un semafor binar
- un monitor cu o variabila

➢ Comunicarea folosind MVar se face in monada IO

```
data MVar a


newEmptyMVar :: IO (MVar a)  -- m <- newEmptyMVar
                             -- m este o locatie goala

newMVar :: a -> IO (MVar a)    -- m <- newMVar v
                              -- m este o locatie care contine valoarea v

takeMVar :: MVar a  -> IO a    -- v <- takeMVar m
                              -- intoarce in v valoarea din m si goleste m
                              -- asteapta (blocheaza thread-ul)  daca m este goala

putMVar :: Mvar a -> a -> IO()  -- putMVar m v
                               -- pune in m valoarea v
                               -- asteapta (blocheaza thread-ul) daca m este plina
```



➢ takeMVar

- takeMVar este o operatie care blocheaza thread-urile

- takeMVar este *single-wakeup*:

 daca variabila MVar este goala, toate thread-urile care vor sa execute takeMVar sunt blocate; cand variabila devine plina, un singur thread este trezit si acesta va executa takeMVar

- daca mai multe thread-uri sunt blocate pe acelasi MVar, ele vor fi trezite in ordinea FIFO

https://www.haskell.org/hoogle/?hoogle=MVar

```haskell
import Control.Concurrent

main = do
        m <- newEmptyMVar
        forkIO $ do
                putMVar m 'x'
                putMVar m 'y'
        x <- takeMVar m
        print x
        x <- takeMVar m
        print x
```

```
*Main> main
'x'
'y'
```

```haskell
newEmptyMVar :: IO (MVar a)
```

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

```haskell
takeMVar :: MVar a  -> IO a
```

```
import Control.Concurrent

main = do
        m <- newEmptyMVar
        takeMVar m
```

```
*Main> main
*** Exception: thread blocked indefinitely in an MVar operation
```

➢ takeMVar vs readMVar

readMVar

Citeste atomic continutul unui MVar.

Daca variabile MVar este goala, thread-ul care apeleaza readMVar va astepta pana cand MVar primeste o valoare si va citi valoarea pusa de urmatoarea operatie putMVar.

readMVar este *multiple-wakeup*, deci toate threa-urile care asteapta sa citeasca din MVar vor fi trezite in acelasi timp.

Implementarea veche

```
readMVar :: MVar a -> IO a
readMVar m =  do
             a <- takeMVar m
             putMVar m a
             return a
```

Implementarea actuala garanteaza ca readMVar este o operatie atomica.

https://www.haskell.org/hoogle/?hoogle=MVar

- MVar ca semafor binar

```
newLock = newMVar ()      -- MVar care contine ()
aquireLock  m = takeMVar m
releaseLock  m = putMVar m ()
```

```
act1 m = do
        aquireLock m
        print "I have the lock"
        releaseLock m
```

```
act2 m = do
        aquireLock m
        print "Now I am have the  lock"
        releaseLock m
```

```
main = do
      m <- newLock
      forkIO $  act1 m
      forkIO $  act2 m
      getLine
```

- MVar ca semafor binar

```
newLock = newMVar ()      -- MVar care contine ()
aquireLock  m = takeMVar m
releaseLock  m = putMVar m ()
```

```
main = do
          m <- newLock
          forkIO $  forever (act1 m)
          forkIO $  forever (act2 m)
          getLine
```

forever repeta o actiune monadica de un numar infinit de ori

- MVar ca semafor binar

```
newLock = newMVar ()      -- MVar care contine ()
aquireLock  m = takeMVar m
releaseLock  m = putMVar m ()
```

```
main = do
          m <- newLock
          forkIO $  forever (act1 m)
          forkIO $  forever (act2 m)
          getLine
```

```
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
"I have the lock"
"Now I have the lock"
```

➢ Sincronizare : doua thread-uri incrementeaza acelasi contor
vrem sa citim valoarea contorului dupa ce ambele thread-uri au terminat

```
add m  =  replicateM_ 1000 $ do
                                 x <- takeMVar m
                                 putMVar m (x +1)

main = do
         m <- newMVar 0
         forkIO (add m )
          forkIO (add m )
          x  <- takeMVar m
          print x
```

- **Sincronizare** : doua thread-uri incrementeaza acelasi contor
  vrem sa citim valoarea contorului dupa ce ambele thread-uri au terminat

```
add m  =  replicateM_ 1000 $ do
                                x <- takeMVar m
                                putMVar m (x +1)

main = do
          m <- newMVar 0
          forkIO (add m )
          forkIO (add m )
          x  <- takeMVar m
          print x
```

```
*Main> :l cont.hs
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> main
0
*Main> main
0
```

- **Sincronizare** : doua thread-uri incrementeaza acelasi contor
  vrem sa citim valoarea contorului dupa ce ambele thread-uri au terminat

```
add m  =  replicateM_ 1000 $ do
                      x <- takeMVar m
                      putMVar m (x +1)

main = do
         m <- newMVar 0
         forkIO (add m )
         forkIO (add m )
         x  <- takeMVar m
         print x
```

```
*Main> :l cont.hs
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> main
0
*Main> main
0
```

trebuie sa ne asiguram ca ambele thred-uri au terminat

➢ Sincronizare

```
add m  ms1 = do
                    replicateM_ 1000 $ do
                                        x <- takeMVar m
                                        putMVar mv (x +1)
                     putMVar ms1 "ok"
```

```
main = do
        m <- newMVar 0
        ms1 <- newEmptyMVar
        ms2 <- newEmptyMVar
        forkIO (add m ms1)
        forkIO (add m ms2)
        takeMVar ms1
        takeMVar ms2
        x  <- takeMVar m
        print x
```

variabilele ms1 si ms2 actioneaza ca niste semafoare ; astfel ne asiguram ca ambele thread-uri au terminat

➢ Sincronizare

```
add m  ms1 = do
              replicateM_ 1000 $ do
                                  x <- takeMVar m
                                  putMVar mv (x +1)
              putMVar ms1 "ok"
```

```
main = do
          m <- newMVar 0
          ms1 <- newEmptyMVar
          ms2 <- newEmptyMVar
          forkIO (add m ms1)
          forkIO (add m ms2)
          takeMVar ms1
          takeMVar ms2
          x  <- takeMVar m
          print x
```

```
*Main> main
2000
```