Párhuzamos programozás Haskellben (folytatás)

Mit tudtunk meg eddig a párhuzamos programokról?

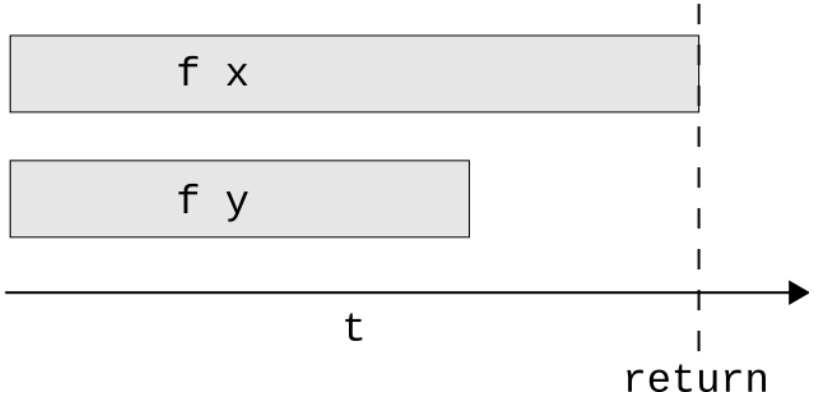
- Párhuzamos programozással gyorsíthatunk a programon, miközben megőrizzük a determinisztikusságát. Teljesen különálló lehetőség, de keverhető konkurenciával.
- Nem automatikus, de annotációkkal és egy intelligens futtató rendszerrel könnyebb, mint manuálisan. Nem a programozónak kell megoldania a szinkronizációt és a kommunikációt.
- Fontos, miként bontjuk fel a problémát. Leggyakoribb szempontok: szemcsézettség (granularity) és az adatfüggőségek (data dependencies).

Emlékeztető: Weak Head Normal Form (WHNF)

```
> let xs = map (+1) [1..10] :: [Int]
> :sprint xs
xs = _

> seq xs ()
()
> :sprint xs
xs = _ : _

> length xs
10
> :sprint xs
xs = [_,_,_,_,_,_,_,_,_,_]

> sum xs
65
> :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

Emlékeztető: az *Eval* monád

```
-- cabal install parallel
import Control.Parallel.Strategies

-- Van "run" függvény, nincs IO, kompozicionális
parParSeqSeqAction f x y = runEval $ do
  x1 ← rpar (f x) -- WHNF párhuzamosan
  y1 ← rpar (f y) -- thunk (másképp nincs hatása)
  rseq x1 -- megvárja a WHNF-et
  rseq y1
  return (x1, y1)
```



Esettanulmány: Egy Sudoku megoldó

```
-- http://www.haskell.org/haskellwiki/Sudoku
import Sudoku
import System.Environment

main :: IO ()
main = do
  [f]  ← getArgs
  file ← readFile f
  let puzzles   = lines file
  let solutions = map solve puzzles
  print $ length [ s | Just s ← solutions ]
```

Esettanulmány: Egy Sudoku megoldó (profil)

```
$ ghc -O Solver1.hs -rtsopts -main-is Solver1
...
$ ./Solver1 sudoku17.1000.txt +RTS -s
1000
   2,362,886,904 bytes allocated in the heap
      38,757,536 bytes copied during GC
         239,376 bytes maximum residency (15 sample(s))
          81,232 bytes maximum slop
               2 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      4573 colls,     0 par    1.56s    0.06s     0.0000s    0.0004s
  Gen  1        15 colls,     0 par    0.01s    0.00s     0.0002s    0.0005s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    0.01s  (  1.52s elapsed)
  GC      time    1.57s  (  0.06s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    1.58s  (  1.58s elapsed)

  %GC     time      99.2%  (4.1% elapsed)

  Alloc rate    186,730,433,380 bytes per MUT second

  Productivity   0.8% of total user, 0.8% of total elapsed
```

Sudoku megoldó: párhuzamosított verzió

```
main :: IO ()
main = do
  [f]  ← getArgs
  file ← readFile f
  let puzzles   = lines file
  let solutions = parallelMap solve puzzles
  print $ length [ s | Just s ← solutions ]
  where
    parallelMap f input = runEval $ do
      let (xs, ys) = splitAt (length input `div` 2) input
      xs1 ← rpar $ force $ map f xs
      ys1 ← rpar $ force $ map f ys
      rseq xs1
      rseq ys1
      return $ xs1 ++ ys1
```

Feltámad az erő: mi az a `force`?

```
import Control.DeepSeq

-- "Normal Form Data"
class NFData α where
  rnf :: α → () -- "reduce to normal form"
  rnf x = x `seq` ()

instance NFData α ⇒ NFData [α] where
  rnf []       = ()
  rnf (x : xs) = rnf x `seq` rnf xs

deepseq :: NFData α ⇒ α → β → β
deepseq x y = rnf x `seq` y

force :: NFData α ⇒ α → α
force x = x `deepseq` x
```

Sudoku megoldó: párhuzamosított verzió (profil)

```
$ ghc -O Solver2.hs -rtsopts -threaded -main-is Solver2
...
$ ./Solver2 sudoku17.1000.txt +RTS -N2 -s
1000
   2,370,896,032 bytes allocated in the heap
      48,743,968 bytes copied during GC
       2,644,144 bytes maximum residency (8 sample(s))
         327,552 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2990 colls,  2990 par    0.42s    0.05s     0.0000s    0.0002s
  Gen  1         8 colls,     7 par    0.01s    0.01s     0.0007s    0.0013s

  Parallel GC work balance: 49.78% (serial 0%, perfect 100%)

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

  SPARKS: 2 (1 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    1.00s  (  0.81s elapsed)
  GC      time    0.43s  (  0.05s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    1.44s  (  0.86s elapsed)

  Alloc rate    2,369,438,827 bytes per MUT second

  Productivity  69.7% of total user, 116.0% of total elapsed
```

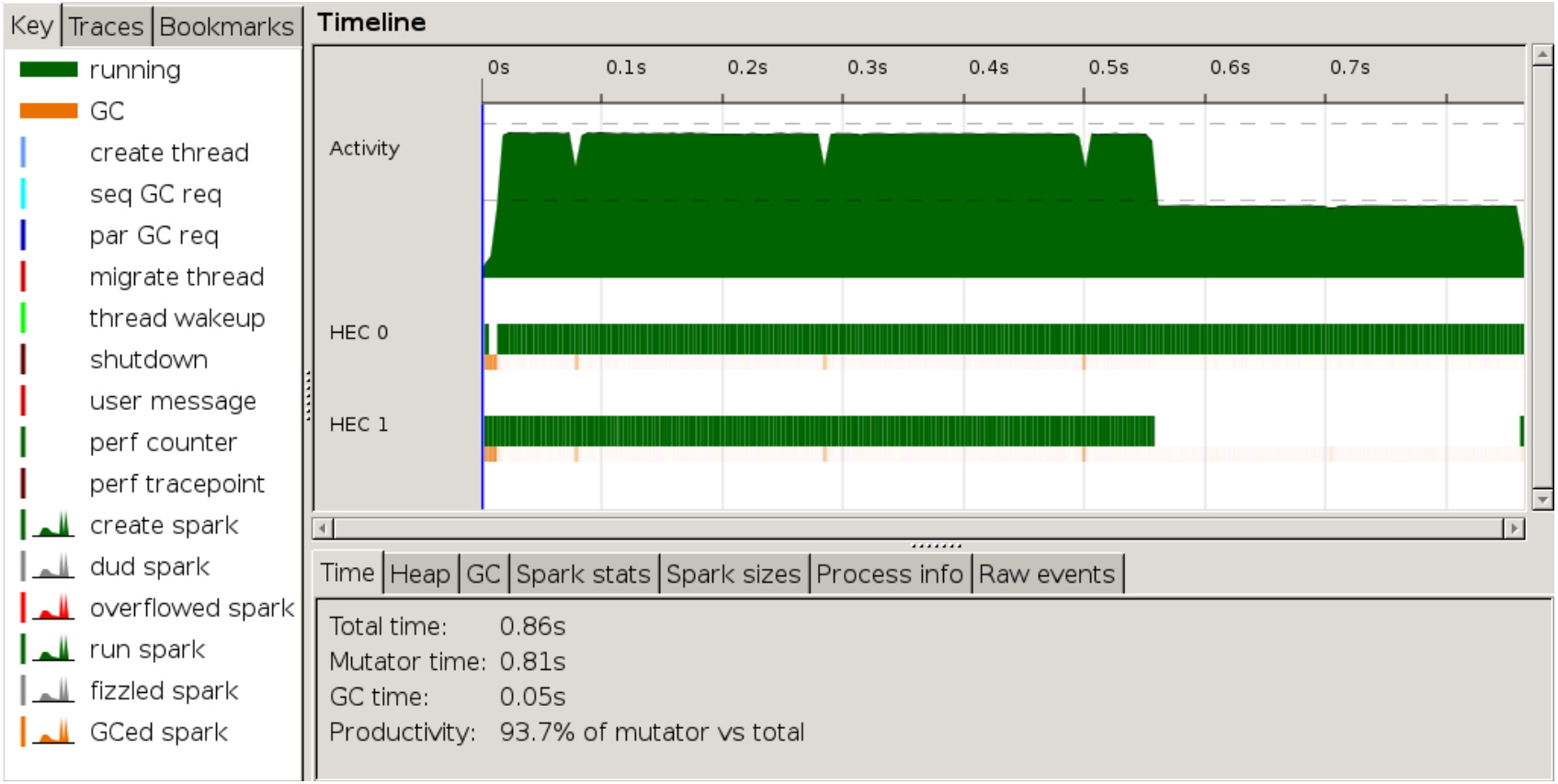
Sudoku megoldó: párhuzamosított verzió (profil)

```
$ rm Solver2
$ ghc -O Solver2.hs -rtsopts -threaded -eventlog -main-is Solver2
...
$ ./Solver2 sudoku17.1000.txt +RTS -N2 -l
1000
$ cabal install ghc-events
...
$ ghc-events show Solver2.eventlog
...
```

Vagy *ThreadScope* (Windows, Mac OS X, Linux és Unix):

```
$ cabal install threadscope
...
$ threadscope Solver2.eventlog
```

Sudoku: párhuzamosított verzió (ThreadScope)



Nincs meg a szikra?

- A fordító nem korlátozza az *rpar* hívásokat – ezeket mindig elosztja a rendelkezésre álló feldolgozó egységek közt.
- Az *rpar* paramétere egy „szikra” – *spark* –, egy kiértékelésre beütemezett kifejezés.
- A sparkokat egységenként egy-egy konténerbe – *spark pool* – tesszük, ahonnan a szálak válogathatnak, habár át is vehetnek egymástól: *"work stealing"*.
- Egy spark állapotai:
  - `converted`: kiértékelt
  - `overflowed`: nem fért bele a spark poolba
  - `dud`: duplán kiértékelt
  - `garbage-collected`: nem használt
  - `fizzled`: időközben kiértékelt
- *Haskell Execution Context (HEC)* vagy capability

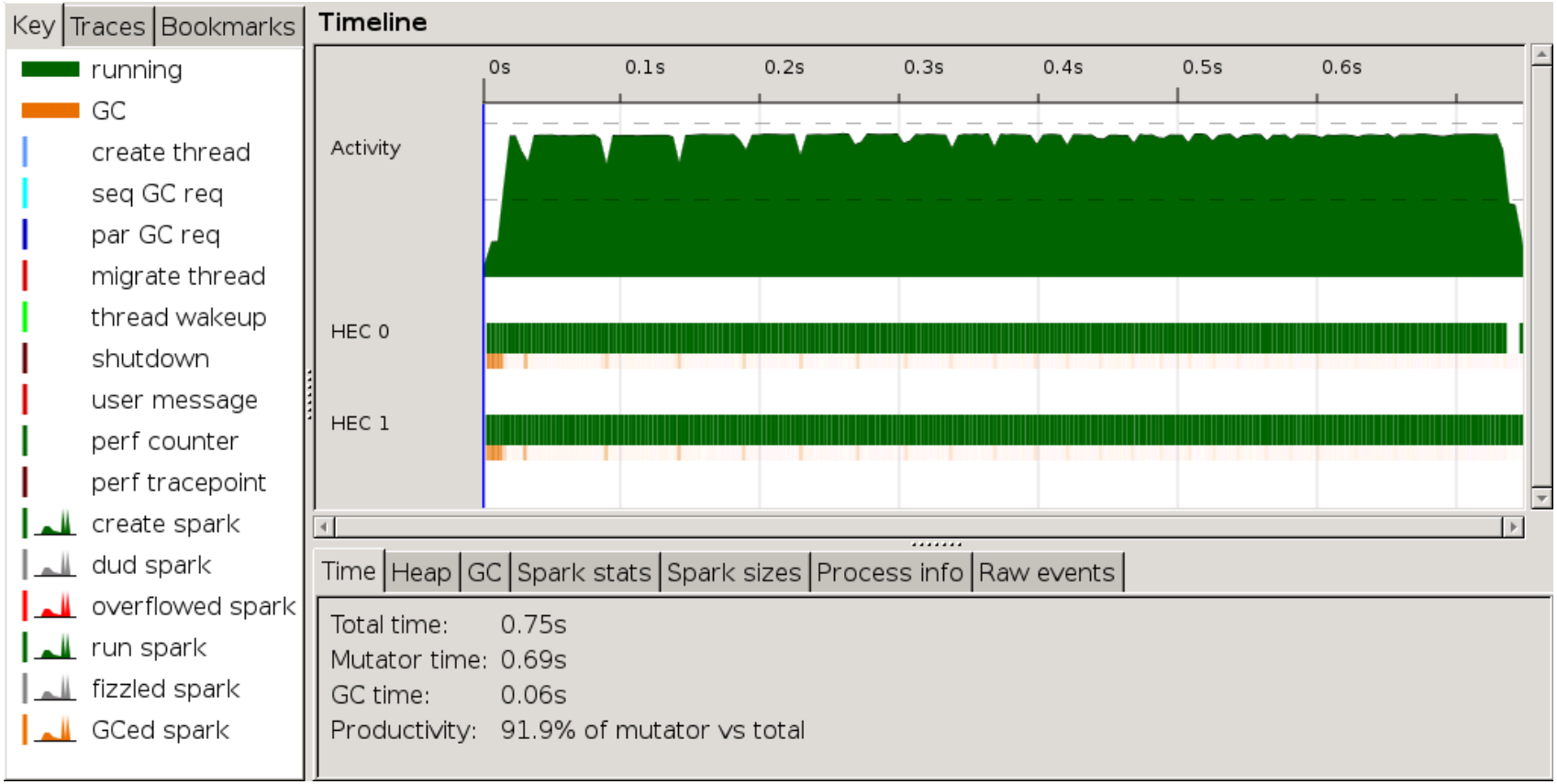
Sudoku: dinamikus párhuzamosítás

```
main :: IO ()
main = do
  [f]  ← getArgs
  file ← readFile f
  let puzzles   = lines file
  let solutions = parallelMap solve puzzles
  print $ length [ s | Just s ← solutions ]
  where
    parallelMap f input = runEval $ parMap f input

    parMap f [] = return []
    parMap f (x : xs) = do
      x1  ← rpar (f x)
      xs1 ← parMap f xs
      return (x1 : xs1)
```

Sudoku: dinamikus párhuzamosítás (profil)

```
$ ghc -O Solver3.hs -rtsopts -threaded -main-is Solver3
...
$ ./Solver3 sudoku17.1000.txt +RTS -N2 -s
1000
   2,389,659,224 bytes allocated in the heap
      63,520,136 bytes copied during GC
       2,163,224 bytes maximum residency (29 sample(s))
          88,008 bytes maximum slop
               8 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2450 colls,  2450 par    0.49s    0.05s     0.0000s    0.0002s
  Gen  1        29 colls,    28 par    0.03s    0.01s     0.0004s    0.0013s

  Parallel GC work balance: 67.15% (serial 0%, perfect 100%)

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

  SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    0.96s  (  0.68s elapsed)
  GC      time    0.52s  (  0.06s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    1.48s  (  0.75s elapsed)

  Alloc rate    2,487,207,502 bytes per MUT second

  Productivity  64.9% of total user, 128.7% of total elapsed
```

Sudoku: dinamikus párhuzamosítás (ThreadScope)



Meddig fokozható?

Mennyire lehet párhuzamosítani?

A választ Amdhal törvénye adja meg:

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

ahol:

- S: a gyorsulás mértéke („speedup”)
- N: a feldolgozó egységek száma
- P: a program párhuzamosítható részeinek aránya ($0 \leq P \leq 1$)

Annotációk absztrakciója: stratégiák

$$\textbf{type}\ \mathit{Strategy}\ \alpha\ =\ \alpha\ \rightarrow\ \mathit{Eval}\ \alpha$$

$$\begin{aligned}
&\mathit{parallelPair}\ ::\ \mathit{Strategy}\ (\alpha,\ \beta)\\
&\mathit{parallelPair}\ (x, y)\ =\ \textbf{do}\\
&\quad x_1\ \leftarrow\ \mathit{rpar}\ x\\
&\quad y_1\ \leftarrow\ \mathit{rpar}\ y\\
&\quad \mathit{return}\ (x_1,\ y_1)
\end{aligned}$$

```
> runEval (parallelPair (f x, f y))
```

$$\textbf{infixl}\ 0\ `\mathit{using}`$$

$$\begin{aligned}
&\mathit{using}\ ::\ \alpha\ \rightarrow\ \mathit{Strategy}\ \alpha\ \rightarrow\ \alpha\\
&x\ `\mathit{using}`\ s\ =\ \mathit{runEval}\ (s\ x)
\end{aligned}$$

```
> (f x, f y) `using` parallelPair
```

Identikusság

Elméletileg az annotáció tetszőlegesen elhagyható:

```
x `using` s  -->  x
```

Gyakorlatilag ehhez viszont a stratégiának identikusnak kell lennie:

```
x `using` s  ==  x
```

Ez a felhasználók által definiált, tetszőleges stratégiák esetében nem feltétlenül garantálható (akárcsak a monádtörvényeknél).

Óvatosan kell bánni a lustasággal:

```
snd ((1 `div` 0, "Hello!") `using` rdeepseq)
```

Paraméterezett stratégiák

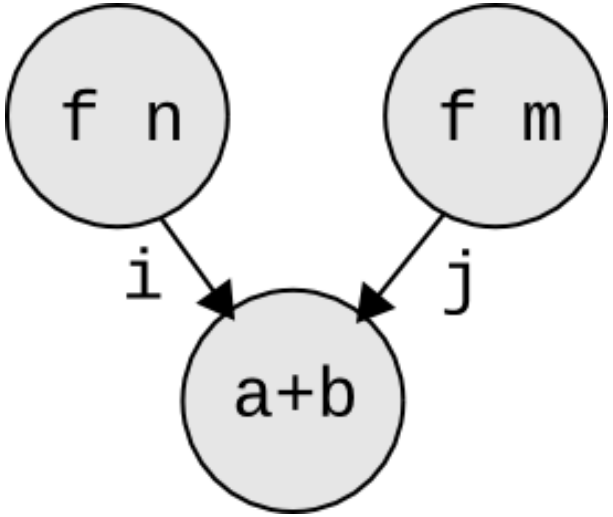
```
import Control.Parallel.Strategies

evalList :: Strategy α → Strategy [α]
evalList _ [] = return []
evalList s (x : xs) = do
  x1  ← rpar (x `using` s)
  xs1 ← evalList s xs
  return (x1 : xs1)

parallelMap f xs = map f xs `using` evalList rseq
```

Párhuzamos adatfolyamok, a *Par* monád

- A stratégiák és az *Eval* számítások nem mindig megfelelőek a párhuzamosítás kifejezésére.
- A *Par* monád lehetőséget ad arra, hogy pontosabban meg tudjuk adni a szemcsézettséget és az adatfüggőségeket, nem függ a lusta kiértékeléstől.
- Továbbra is determinisztikus modell!

Par monád: primitívek

```
-- cabal install monad-par
import Control.Monad.Par

-- runPar :: Par α → α
runPar $ do
  -- new :: Par (IVar α)
  i ← new
  j ← new
  -- fork :: Par () → Par ()
  fork $ put i (f n)
  -- put :: NFData α ⇒ IVar α → α → Par ()
  fork $ put j (f m)
  -- get :: IVar α → Par α
  a ← get i
  b ← get j
  return (a + b)
```



`Par` kombinátorok

$$put\_ :: IVar\ \alpha \rightarrow \alpha \rightarrow Par\ ()$$

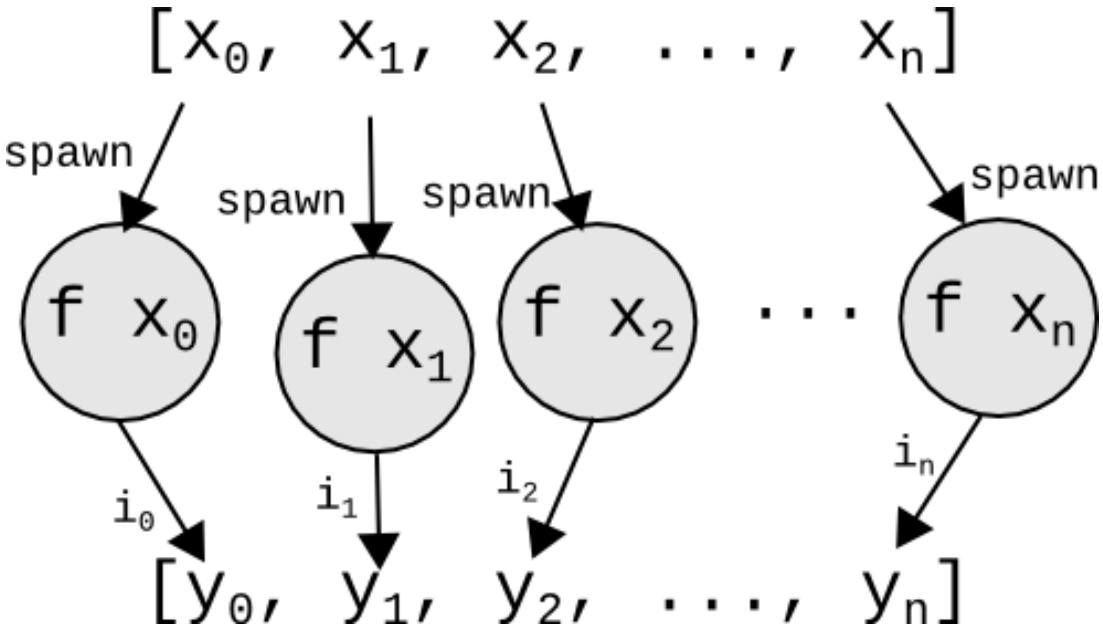
$$\begin{array}{l}
spawn :: NFData\ \alpha \Rightarrow Par\ \alpha \rightarrow Par\ (IVar\ a) \\
spawn\ p = \mathbf{do} \\
\quad i \leftarrow new \\
\quad fork\ \$\ \mathbf{do} \\
\qquad x \leftarrow p \\
\qquad put\ i\ x \\
\quad return\ i
\end{array}$$

$$spawn\_ :: Par\ \alpha \rightarrow Par\ (IVar\ \alpha)$$

$$\begin{array}{l}
spawnP :: NFData\ \alpha \Rightarrow \alpha \rightarrow Par\ (IVar\ \alpha) \\
spawnP = spawn \circ return
\end{array}$$

Párhuzamos leképezés au `Par`

$$\mathit{parMapM} :: \mathit{NFData}\,\beta \Rightarrow (\alpha \to \mathit{Par}\,\beta) \to [\alpha] \to \mathit{Par}\,[\beta]$$
$$\mathit{parMapM}\ f\ as = \mathbf{do}$$
$$\quad \mathit{ibs} \leftarrow \mathit{mapM}\,(\mathit{spawn} \circ f)\ as$$
$$\quad \mathit{mapM}\ \mathit{get}\ \mathit{ibs}$$



Sudoku: adatfolyam párhuzamosítás

```
main :: IO ()
main = do
  [f]  ← getArgs
  file ← readFile f
  let puzzles   = lines file
  let solutions = parallelMap solve puzzles
  print $ length [ s | Just s ← solutions ]
  where
    parallelMap f input = runPar $ do
      ibs ← mapM (spawnP ∘ f) input
      mapM get ibs
```

Sudoku: adatfolyam párhuzamosítás (profil)

```
$ ghc -O Solver4.hs -rtsopts -threaded -main-is Solver4
...
$ ./Solver4 sudoku17.1000.txt +RTS -N2 -s
1000
   2,377,124,296 bytes allocated in the heap
      51,987,992 bytes copied during GC
       2,883,888 bytes maximum residency (11 sample(s))
         245,272 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2480 colls,  2480 par    0.58s    0.05s     0.0000s    0.0001s
  Gen  1        11 colls,    10 par    0.02s    0.01s     0.0007s    0.0015s

  Parallel GC work balance: 69.65% (serial 0%, perfect 100%)

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    0.90s  (  0.70s elapsed)
  GC      time    0.60s  (  0.06s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    1.50s  (  0.76s elapsed)

  Alloc rate    2,631,399,014 bytes per MUT second

  Productivity  60.1% of total user, 119.1% of total elapsed
```

Sudoku: adatfolyam párhuzamosítás (ThreadScope)