PHEX – Parallel Haskell Examples

Ludwik Ciechański
Wojciech Wańczyk

Akademia Górniczo Hutnicza
im. Stanisława Staszica w Krakowie

lciechan@student.agh.edu.pl
wwanczyk@student.agh.edu.pl

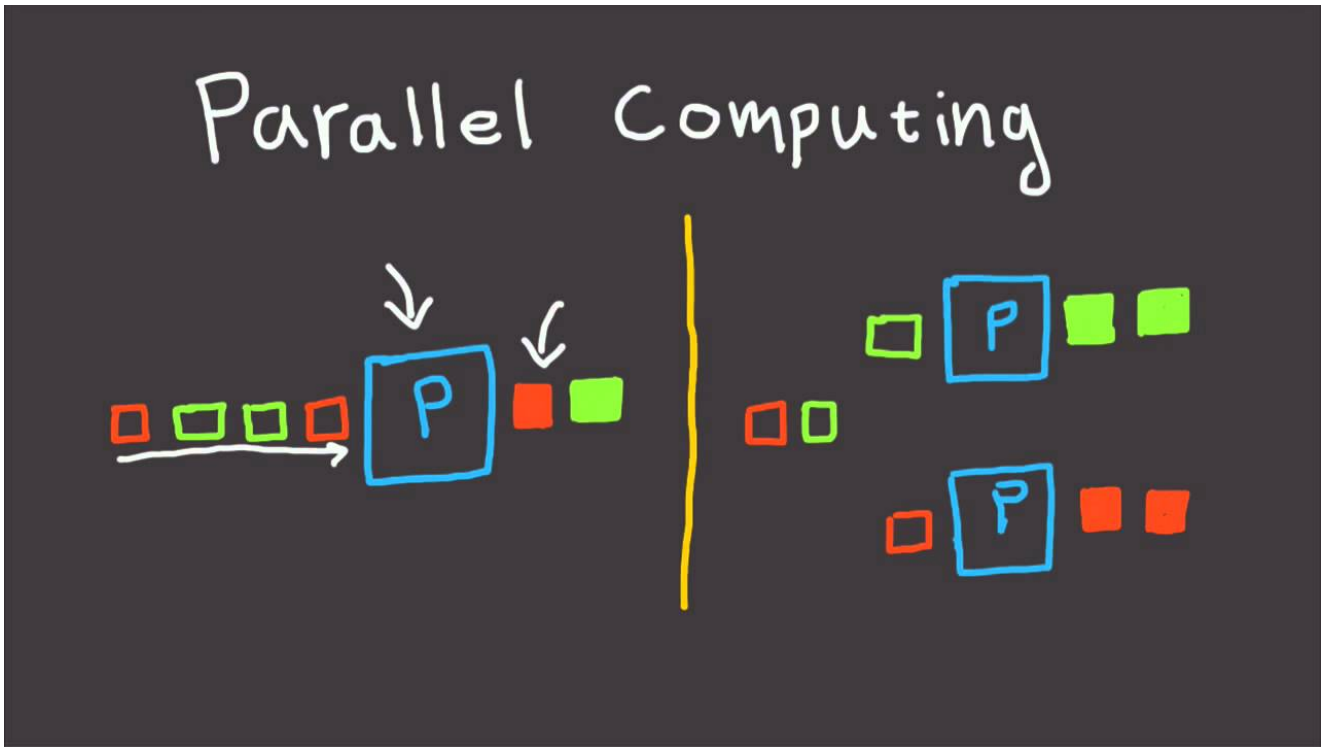
25 stycznia 2018

Plan prezentacji

1. Wprowadzenie
2. Przykłady
   - Fibonacci
   - Quicksort
   - SumPrime
   - ClosestPoint
   - ImageConversion
3. Zakończenie

Wprowadzenie

- Celem projektu jest zbadanie czasu wykonania programów równoległych w stosunku do programów sekwencyjnych.
- Wybór przykładów okazał się dość trudny, aczkolwiek jest on nieprzypadkowy. Są one raczej proste i dobrze pokazują badany temat.



Fibonacci

Fibonacci

fib

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

parFib

```
parFib :: Integer -> Integer
parFib 0 = 0
parFib 1 = 1
parFib n = par nf (parFib (n - 2) + nf)
    where nf = parFib(n - 1)
```

parFib''

```
parFib'' :: Integer -> Integer -> Integer
parFib'' 0 n = fib n
parFib'' _ 0 = 0
parFib'' _ 1 = 1
parFib'' d n = par nf1 (pseq nf2 (nf1 + nf2))
    where nf1 = parFib'' (d - 1) (n - 1)
          nf2 = parFib'' (d - 1) (n - 2)
```

parFib'

```
parFib' :: Integer -> Integer
parFib' 0 = 0
parFib' 1 = 1
parFib' n = par nf1 (pseq nf2 (nf1 + nf2))
    where nf1 = parFib'(n - 1)
          nf2 = parFib'(n - 2)
```

Fibonacci - czasy wykonania

Wersja	**-N1**	**-N2**	**-N4**
fib	2.52	2.17	2.18
parFib	3.29	3.34	3.06
parFib'	1.45	1.45	1.45
parFib''	0.98	0.99	0.98

Tabela: Badanie ciągu Fibonacciego - element nr 35.

Quicksort

seqSort

```
seqSort :: Ord a => [a] -> [a]
seqSort []    = []
seqSort (x:xs) = (seqSort lesser) ++ [x] ++ (seqSort greater)
    where
        lesser = filter (< x) xs
        greater = filter (>= x) xs
```

parSort

```
parSort :: (Ord a) => [a] -> [a]
parSort []     = []
parSort (x:xs) = force greater `par` (force lesser `pseq` (lesser ++ x:greater))
    where
        lesser = parSort [y | y <- xs, y < x]
        greater = parSort [y | y <- xs, y >= x]
```

Quicksort

optParSort

```
optParSort :: (Ord a) => Int -> [a] -> [a]
optParSort _ [] = []
optParSort d list@(x:xs)
  | d <= 0   = seqSort list
  | otherwise = force greater `par` (force lesser `pseq` (lesser ++ x:greater))
    where
        lesser = optParSort dn [y | y <- xs, y < x]
        greater = optParSort dn [y | y <- xs, y >= x]
        dn = d - 1
```

Quicksort - czasy wykonania

Wersja	**-N1**	**-N2**	**-N4**
seqSort	2.86	2.71	3.02
parSort	3.42	2.76	3.11
optParSort	2.66	2.56	2.87

Tabela: Badanie quicksorta.

SumTwoPrimes oraz SumPrimesInRange

sumTwoPrimes

```
sumTwoPrimes :: Int -> Int -> Int
sumTwoPrimes x y = a + b
 where
  a = if elem x (eS [2..x]) then x else 0
  b = if elem y (eS [2..y]) then y else 0
```

sumPrimesInRange

```
sumPrimesInRange :: [Int] -> Int
sumPrimesInRange [] = 0
sumPrimesInRange (x:xs) = a + b
 where a = if elem x (eS [2..x]) then x else 0
       b = (sumPrimesInRange xs)
```

sumTwoPrimesPar

```
sumTwoPrimesPar :: Int -> Int -> Int
sumTwoPrimesPar x y =
       par a (pseq b (a + b))
 where
  a = if elem x (eS [2..x]) then x else 0
  b = if elem y (eS [2..y]) then y else 0
```

sumPrimesInRangePar

```
sumPrimesInRangePar :: [Int] -> Int
sumPrimesInRangePar [] = 0
sumPrimesInRangePar (x:xs) =
       par a (pseq b (a + b))
 where a = if elem x (eS [2..x]) then x else 0
       b = (sumPrimesInRangePar xs)
```

SumTwoPrimes oraz SumPrimesInRange - czasy wykonania

Wersja	**-N1**	**-N2**	**-N4**
sumTwoPrimes	13.65	13.24	18.00
sumTwoPrimesPar	13.41	6.95	9.78

Tabela: Badanie sumTwoPrimes.

Wersja	**-N1**	**-N2**	**-N4**
sumPrimesInRange	68.04	66.55	75.93
sumPrimesInRangePar	76.47	42.42	24.58

Tabela: Badanie sumPrimesInRange.

ClosestPoint

closestPar

```
closestPar :: (Float, Float) ->
    [(Float, Float)] -> (Float, Float)
closestPar p (x:y:[]) = a
 where
  a = minPair2 p x y
closestPar p (x:y:z:[]) = a
 where
  a = minPair3 p x y z
closestPar p list = final
 where
  len = length list
  mid = div len 2
  l = take mid list
  r = drop mid list
  a = closestPar p l
  b = closestPar p r
  final = par a (pseq b (minPair2 p a b))
```

```
distance :: (Float, Float) ->
    (Float, Float) -> Float
distance a b = sqrt ((fst a - fst b)^2 +
    (snd a - snd b)^2)

minPair2 p a b
 | distance p a < distance p b = a
 | otherwise = b

minPair3 p a b c
 | distance p a < distance p b &&
       distance p a < distance p c = a
 | distance p b < distance p a &&
       distance p b < distance p c = b
 | otherwise = c
```

ClosestPoint - czasy wykonania

Wersja	**-N1**	**-N2**	**-N4**
closest	3.46	3.74	3.97
closestPar	2.98	2.24	1.81

Tabela: Badanie ClosestPoint.

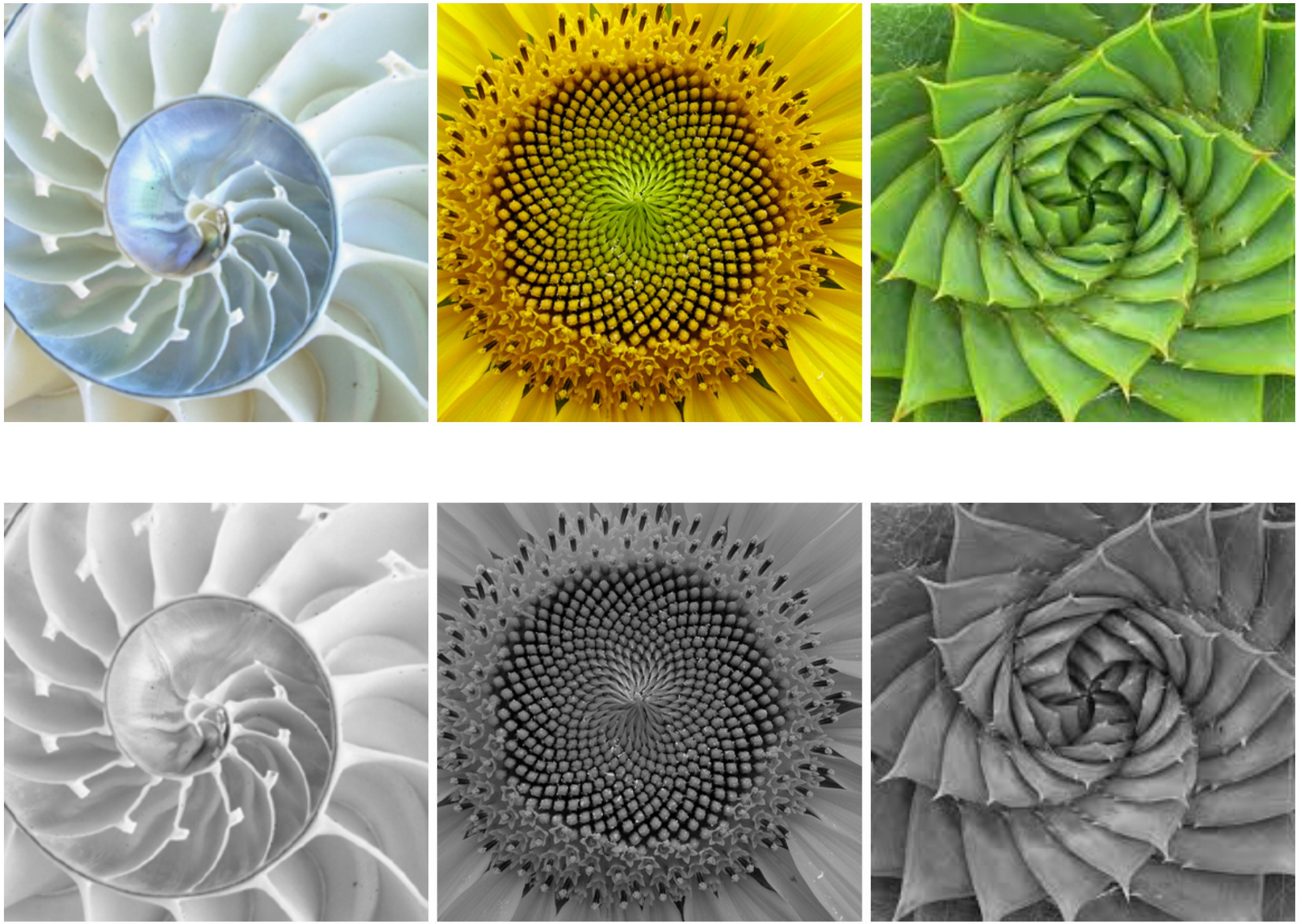
ImageConversion

toGreyScale

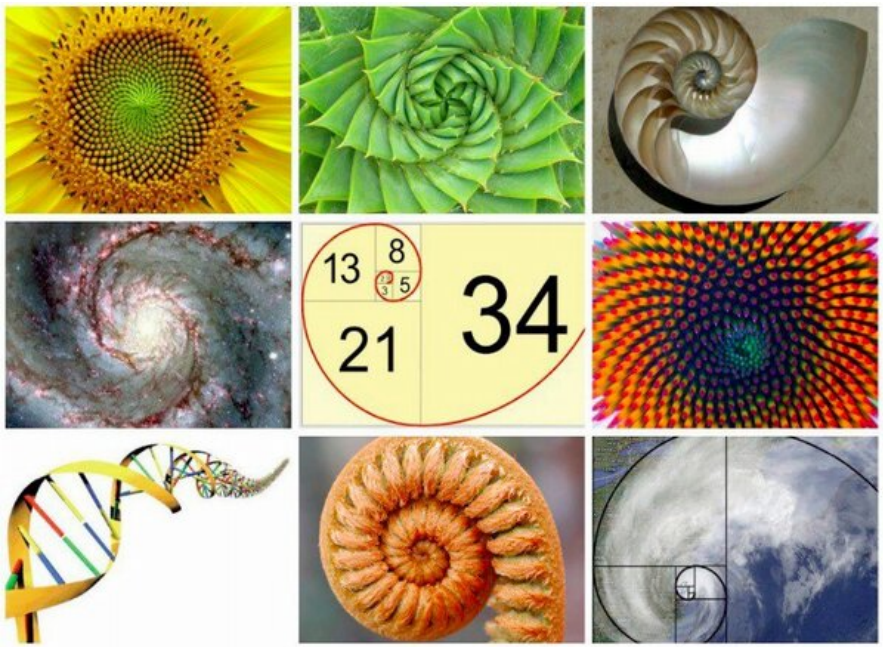
```
toGreyScale :: Image PixelRGB8 -> Image Pixel8
toGreyScale = pixelMap pixelGrey

pixelGrey :: PixelRGB8 -> Pixel8
pixelGrey (PixelRGB8 r g b) =
    (div r 3 + div g 3 + div b 3)

dynToPix :: DynamicImage -> Image PixelRGB8
dynToPix (ImageRGB8 img) = img
```

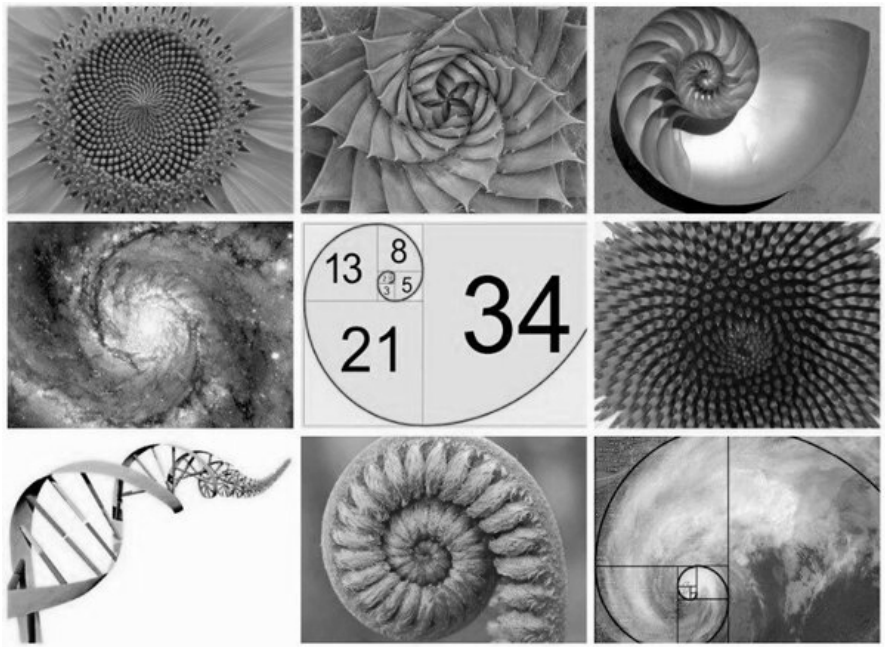
```
imageCon = do
    [pathIn] <- getArgs
    start <- getCurrentTime
    Right imgIn <- readPng pathIn
    let pathGrey = ("grey-" ++ pathIn)
    let imgGrey =
        toGreyScale . dynToPix $ imgIn
    writePng pathGrey imgGrey
    end <- getCurrentTime
    putStrLn $ "Image converted to grey
         scale, saved in " ++ pathGrey
    putStrLn $ show (end `diffUTCTime`
         start) ++ " elapsed."
```

ImageConversion - efekty



ImageConversion - efekty





Podsumowanie

- Odpowiednio użyte techniki programowania równoległego dają zadowalające efekty.
- Zaprezentowane przez nas przykłady pokazują, że możliwe jest kilku- a nawet kilkunasto- krotne przyspieszenie wykonania programu.

Źródła

Simon Marlow (2013)

Parallel and Concurrent Programming in Haskell

O'Reilly Media

Simon Peyton Jones, Satnam Singh

A Tutorial on Parallel and Concurrent Programming in Haskell

John Hughes

Parallel Functional Programming Lecture 1

Chalmers, University of Gothenburg

Dziękujemy za uwagę!