word-histogram

Claude Heiland-Allen

2015

Contents

1 .gitignore 2
2 Histogram.hs 2
3 LICENSE 4
4 Main.hs 5
5 README.md 6
6 Setup.hs 10
7 TakeSort.hs 10
8 Utf8Chunk.hs 11
9 word-histogram.cabal 12

1 .gitignore

```
.cabal-sandbox
cabal.sandbox.config
dist
*.aux
*.hp
*.ps
```

2 Histogram.hs

```
{-# LANGUAGE RoleAnnotations #-}
module Histogram
  ( Histogram()
  , fromFrequencyList
  , toFrequencyList
  , fromList
  , fromMap
  , toMap
  , split
  , splits
  , Frequency(..)
  , fromListsPar
  , fromListsParIO
  , fromFrequencyListsPar
  , fromFrequencyListsParIO
  )
  where

import Control.Parallel.Strategies (parMap, rseq)       -- parallel
import Control.Concurrent.Spawn (parMapIO)              -- spawn
import Control.Exception (evaluate)
import qualified Data.Map.Strict as M                   -- containers
```

```
import Data.Monoid (Monoid(..))

-- | A histogram.
type role Histogram nominal
newtype Histogram a = Histogram{ getHistogram :: M.Map a Int }
  deriving (Read, Show, Eq, Ord)

newtype Frequency a = Frequency{ getFrequency :: (a, Int) }
  deriving (Read, Show, Eq)

instance Ord a => Ord (Frequency a) where
  compare (Frequency (x, a)) (Frequency (y, b)) = compare a b `mappend` compare x y

instance Ord a => Monoid (Histogram a) where
  mempty = Histogram M.empty
  mappend (Histogram a) (Histogram b) = Histogram (M.unionWith (+) a b)
  mconcat = Histogram . M.unionsWith (+) . map getHistogram

fromFrequencyList
  :: Ord a
  => [(a, Int)]
  -> Histogram a
fromFrequencyList = Histogram . M.fromListWith (+)
{-# INLINE fromFrequencyList #-}

toFrequencyList
  :: Histogram a
  -> [(a, Int)]
toFrequencyList = M.toList . getHistogram
{-# INLINE toFrequencyList #-}

fromList
  :: Ord a
  => [a]          -- ^ values
  -> Histogram a  -- ^ histogram
fromList = fromFrequencyList . map (\w -> (w, 1))
{-# INLINE fromList #-}

fromMap
  :: M.Map a Int
  -> Histogram a
fromMap = Histogram
{-# INLINE fromMap #-}

toMap
  :: Histogram a
  -> M.Map a Int
toMap = getHistogram
{-# INLINE toMap #-}


-- | Split a histogram into disjoint pieces.  How many pieces and their sizes
--   depends on the implementation of 'Data.Map.splitRoot', which is iterated
--   'depth' times.
splits
  :: Int            -- ^ depth must be non-negative
```

```
  -> Histogram a    -- ^ histogram
  -> [Histogram a]  -- ^ histogram pieces
splits depth
  = map Histogram
  . (!! depth) . iterate (concatMap M.splitRoot) . (:[])
  . getHistogram
{-# INLINE splits #-}

-- | Split a histogram once.
--
--   > split == splits 1
split
  :: Histogram a    -- ^ histogram
  -> [Histogram a]  -- ^ histogram pieces
split = splits 1
{-# INLINE split #-}

fromListsPar :: Ord a => [[a]] -> Histogram a
fromListsPar = mconcat . parMap rseq fromList
{-# INLINE fromListsPar #-}

fromFrequencyListsPar :: Ord a => [[(a, Int)]] -> Histogram a
fromFrequencyListsPar = mconcat . parMap rseq fromFrequencyList
{-# INLINE fromFrequencyListsPar #-}

fromListsParIO :: Ord a => [[a]] -> IO (Histogram a)
fromListsParIO = fmap mconcat . parMapIO (evaluate . fromList)
{-# INLINE fromListsParIO #-}

fromFrequencyListsParIO :: Ord a => [[(a, Int)]] -> IO (Histogram a)
fromFrequencyListsParIO = fmap mconcat . parMapIO (evaluate . fromFrequencyList)
{-# INLINE fromFrequencyListsParIO #-}
```

3 LICENSE

4 Main.hs

```
module Main (main) where

import Data.Coerce (coerce)
import Data.Char (isSpace)
import Data.Ord (Down(Down))
import qualified Data.ByteString    as B        -- bytestring
import qualified Data.Text          as T        -- text
import qualified Data.Text.Encoding as T        -- text
import GHC.Conc (getNumCapabilities)
import System.Environment (getArgs)
import System.Exit (exitFailure)
import Text.Read (readMaybe)

import qualified Histogram as H
import TakeSort (takeSort, takeSortsPar, takeSortsParIO)
import Utf8Chunk (utf8Chunk)

data Parallel = Sequential | Strategies | Threads
  deriving (Read)

data Count = Chars | Words | Lines
  deriving (Read)

usage :: IO a
usage = do
  putStrLn "usage: word-histogram cores hpar tpar type count utf8.txt"
  putStrLn "hpar/tpar can be Sequential | Strategies | Threads"
  putStrLn "type can be Chars | Words | Lines"
  exitFailure

parseArgs :: IO (Int, Parallel, Parallel, Count, Int, FilePath)
parseArgs = do
  args <- getArgs
  case args of
    [coresS, histS, topS, countS, topCountS, fileName] ->
      case (,,,,) <$>
           readMaybe coresS <*>
           readMaybe histS <*>
           readMaybe topS <*>
           readMaybe countS <*>
           readMaybe topCountS <*>
           Just fileName of
        Just s -> return s
        Nothing -> usage
    _ -> usage
```

```
main :: IO ()
main = do
  (cores0, histIO, topIO, count, topCount, fileName) <- parseArgs
  cores1 <- getNumCapabilities
  let cores
        | cores0 > 0 = cores0
        | otherwise = cores1
      log2cores = ceiling (logBase (2 :: Double) (fromIntegral cores))
  -- serial:   read the input file
  fileContents <- B.readFile fileName                -- O(total bytes)
  -- serial:   split it into 1 chunk for each core
  let fileLength = B.length fileContents             -- O(1)
      chunkSize = div fileLength cores               -- O(1)
      strings = utf8Chunk canBreak chunkSize fileContents -- O(cores)
      canBreak = case count of
        Chars -> const True
        Words -> isSpace
        Lines -> ('\n' ==) -- FIXME check this works
      unpack = case count of
        Chars -> map T.singleton . T.unpack -- FIXME optimize, type system ouch
        Words -> T.words
        Lines -> T.lines
      -- O(total words / cores + unique words)
      histogram = case histIO of
        Sequential -> return . mconcat . map H.fromList
        Strategies -> return . H.fromListsPar
        Threads    -> H.fromListsParIO
      -- O(unique words * topCount / cores + cores * topCount)
      top = case topIO of
        Sequential -> \n -> return . takeSort n . concatMap (takeSort n)
        Strategies -> \n -> return . takeSortsPar n
        Threads    -> \n -> takeSortsParIO n
      source = map T.decodeUtf8 strings
  -- parallel: compute histogram for each chunk
  freq <- histogram $ map unpack source
  -- serial:   split the final histogram into disjoint submaps
  let freqs = map H.toFrequencyList $ H.splits log2cores freq
  -- parallel: extract the peaks
  result <- coerceOnF (map (Down . H.Frequency)) (top topCount) freqs
  putStr . unlines . map show $ result               -- O(topCount)

coerceOnF f g
  = fmap (coerce `asTypeOfReversed` f) . g . fmap (coerce `asTypeOf` f)
  where
    asTypeOfReversed :: (b -> a) -> (a -> b) -> (b -> a)
    asTypeOfReversed = const
```

5 README.md

```
word-histogram
==============

Count frequencies of chars/words/lines in UTF-8 text files using three kinds of
parallelism (sequential, Control.Parallel.Strategies, Control.Concurrent.Spawn).

Inspired by: <https://github.com/apauley/parallel-frequency>
```

```
Usage
-----

    word-histogram cores hpar tpar type count utf8

    cores  -- the amount of parallelism to use
              (0 uses getNumCapabilities)

    hpar   -- the parallel method for histogram creation
              (Sequential | Strategies | Threads)

    tpar   -- the parallel method for "top N" extraction
              (Sequential | Strategies | Threads)

    type   -- what to histogram on
              (Chars | Words | Lines)

    count  -- how many items to extract
              (the N in "top N")

    utf8   -- the input filename
              (must contain valid UTF-8 encoded text)

    All arguments are mandatory and must occur in order.


Benchmarks
----------

AMD Athlon(tm) II X4 640 Processor (quad core, 3GHz, 8GB RAM)
ghc-7.10.1 (Linux amd64 bindist)


    $ ls -lsh artamene.txt
    11M artamene.txt
    $ word-histogram +RTS -N -s -A64M -RTS 4 Strategies Sequential Words 10 ↙
        ↳ artamene.txt
    ("de",87178)
    ("que",67057)
    ("et",47254)
    (":",44569)
    ("la",42343)
    ("\224",34063)
    ("ne",32839)
    ("vous",29862)
    ("le",29532)
    ("je",29121)
       1,489,660,416 bytes allocated in the heap
          34,407,248 bytes copied during GC
          27,761,352 bytes maximum residency (1 sample(s))
           2,389,512 bytes maximum slop
                 305 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max ↙
                                        ↳ pause
      Gen  0         6 colls,     5 par    0.132s   0.042s     0.0071s    0.0155↙
```

```
    ↳ s
  Gen  1         1 colls,     1 par    0.020s   0.006s     0.0057s    0.0057↙
    ↳ s

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

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

  SPARKS: 4 (4 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.004s elapsed)
  MUT     time    2.524s  (  0.783s elapsed)
  GC      time    0.152s  (  0.048s elapsed)
  EXIT    time    0.004s  (  0.004s elapsed)
  Total   time    2.680s  (  0.839s elapsed)

  Alloc rate    590,198,263 bytes per MUT second

  Productivity  94.3% of total user, 301.4% of total elapsed

gc_alloc_block_sync: 8943
whitehole_spin: 0
gen[0].sync: 2081
gen[1].sync: 12


$ ls -1sh books.txt
29M books.txt
$ word-histogram +RTS -N -s -A64M -RTS 4 Strategies Sequential Words 10 ↙
    ↳ books.txt
("the",158933)
("and",107854)
("of",89036)
("de",87430)
("que",67091)
("to",66370)
("a",50123)
("et",47304)
(":",44569)
("in",43127)
   4,246,763,552 bytes allocated in the heap
     125,486,528 bytes copied during GC
      63,881,184 bytes maximum residency (2 sample(s))
       4,052,400 bytes maximum slop
             365 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max ↙
                                       ↳ pause
  Gen  0        18 colls,    18 par    0.512s   0.174s     0.0097s    0.0164↙
    ↳ s
  Gen  1         2 colls,     1 par    0.024s   0.008s     0.0041s    0.0067↙
    ↳ s

  Parallel GC work balance: 58.09% (serial 0%, perfect 100%)

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)
```

```
  SPARKS: 4 (3 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.000s  (  0.004s elapsed)
  MUT     time    8.080s  (  2.522s elapsed)
  GC      time    0.536s  (  0.183s elapsed)
  EXIT    time    0.004s  (  0.002s elapsed)
  Total   time    8.620s  (  2.710s elapsed)

  Alloc rate    525,589,548 bytes per MUT second

  Productivity  93.8% of total user, 298.3% of total elapsed

gc_alloc_block_sync: 109141
whitehole_spin: 0
gen[0].sync: 1796
gen[1].sync: 0


$ ls -1sh gutenberg.rdf
673M gutenberg.rdf
$ word-histogram +RTS -N -s -A64M -RTS 4 Strategies Sequential Words 10 ↙
    ↳ gutenberg.rdf
("</rdf:Description>",1051760)
("<rdf:Description",1051752)
("<dcam:memberOf",966656)
("<rdf:value",830516)
("rdf:resource=\"http://purl.org/dc/terms/IMT\"/>",781337)
("<dcterms:format>",781337)
("</dcterms:format>",781337)
("<pgterms:file",661624)
("<dcterms:modified",661624)
("<dcterms:isFormatOf",661624)
  22,842,886,616 bytes allocated in the heap
   1,746,765,656 bytes copied during GC
   1,417,475,456 bytes maximum residency (3 sample(s))
      19,399,528 bytes maximum slop
            2388 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max ↙
                                        ↳ pause
  Gen  0        95 colls,    95 par    7.340s   2.716s     0.0286s    0.2101↙
     ↳ s
  Gen  1         3 colls,     2 par    0.056s   0.098s     0.0328s    0.0894↙
     ↳ s

  Parallel GC work balance: 43.40% (serial 0%, perfect 100%)

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

  SPARKS: 4 (4 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.004s  (  0.004s elapsed)
  MUT     time   75.564s  ( 24.771s elapsed)
  GC      time    7.396s  (  2.814s elapsed)
  EXIT    time    0.028s  (  0.090s elapsed)
  Total   time   82.992s  ( 27.678s elapsed)
```

```
        Alloc rate    302,298,536 bytes per MUT second

        Productivity  91.1% of total user, 273.1% of total elapsed

      gc_alloc_block_sync: 260218
      whitehole_spin: 0
      gen[0].sync: 21626
      gen[1].sync: 0


    Notes
    -----

    The main low-level hack that achieves impressive parallel speed-up is chunking
    the input UTF-8 text file into an appropriate number of pieces using O(1)
    ByteString operations before even starting any histogram generation.

    Map.fromListWith (+) is about as fast as any sequential method to apply to
    each chunk (possibly in parallel), which are then merged sequentially with
    Map.unionsWith (+).

    Extracting the "top N" values is parallelized by splitting the histogram into
    disjoint pieces and finding the top N of each piece in parallel.  The "top N"
    of the whole histogram is then the "top N" of concatenation of the piece
    "top N"s.  But this operation doesn't seem to get as much parallel speedup.


    --
    http://code.mathr.co.uk/word-histogram
```

6 Setup.hs

```
import Distribution.Simple
main = defaultMain
```

7 TakeSort.hs

```
module TakeSort
  ( takeSort
  , takeSortsPar
  , takeSortsParIO
  ) where

import Data.Coerce (coerce)
import Data.List (foldl', insert, transpose)
import Data.Maybe (catMaybes)
import Data.Ord (Down(Down))
import Control.Parallel.Strategies (parMap, rseq)       -- parallel
import Control.Concurrent.Spawn (parMapIO)              -- spawn
import Control.Exception (evaluate)

-- | Sort a list keeping only the first few values.
--   O(n * length xs)
takeSort
  :: Ord a
  => Int        -- ^ n
  -> [a]        -- ^ xs
```

```
  -> [a]        -- ^ take n (sort xs)
-- invariant: xs contains the (exactly) n largest values seen so far
-- head xs is the smallest value that makes the top n
-- if x is less than that, it will be pruned by the tail call
-- otherwise the new smallest value will be pruned
takeSort n
  = coercively (map getDown)
  . reverse
  . catMaybes
  . foldl' (\xs x -> tail (insert (Just x) xs)) (replicate n Nothing)
  . coercively (map Down)
{-# INLINE takeSort #-}

getDown :: Down a -> a
getDown (Down a) = a

coercively f = coerce `asTypeOf` f
{-# INLINE coercively #-}


takeSortsPar :: Ord a => Int -> [[a]] -> [a]
takeSortsPar n = takeSort n . concat . transpose . parMap rseq (takeSort n)
{-# INLINE takeSortsPar #-}

takeSortsParIO :: Ord a => Int -> [[a]] -> IO [a]
takeSortsParIO n = fmap (takeSort n . concat . transpose) . parMapIO (evaluate .↙
    ↳ takeSort n)
{-# INLINE takeSortsParIO #-}
```

8 Utf8Chunk.hs

```
{-# LANGUAGE BangPatterns #-}

-- | Split a UTF-8 ByteString into chunks.
module Utf8Chunk (utf8Chunk) where

import Data.Bits ((.&.))
import Data.Word (Word8)
import qualified Data.ByteString    as B       -- bytestring
import qualified Data.Text          as T       -- text
import qualified Data.Text.Encoding as T       -- text

-- | Split a strict ByteString containing UTF-8 into chunks, each no smaller
--   than the desired size.
utf8Chunk
  :: (Char -> Bool)   -- ^ which characters are allowable break points
  -> Int              -- ^ desired chunk size in bytes
  -> B.ByteString     -- ^ must be valid UTF-8
  -> [B.ByteString]   -- ^ valid UTF-8 chunks split at allowable points
utf8Chunk canBreak approxChunkBytes = go
  where
    go utf8Input
      | B.null utf8Input = []
      | otherwise =
          let -- skip the target number of bytes
              post = B.drop approxChunkBytes utf8Input
              -- ensure multi-byte characters are synchronized
```

```
                (cont, rest) = B.span isUtf8ContinuationByte post
                -- ensure it is safe to break (eg: not in hte middle of a word)
                block = breakLengthUtf8 canBreak rest
                -- find the true chunk length taking the above into account
                chunkBytes = approxChunkBytes + B.length cont + block
                -- split the input without copying
                (prefix, suffix) = B.splitAt chunkBytes utf8Input
                -- decode the chunk and continue with the remainder
            in  prefix : go suffix

isUtf8ContinuationByte :: Word8 -> Bool
-- https://en.wikipedia.org/wiki/UTF-8#Description
isUtf8ContinuationByte w = w .&. 0xC0 == 0x80

breakLengthUtf8 :: (Char -> Bool) -> B.ByteString -> Int
-- keep getting text until it's safe to break
-- return the number of bytes consumed
breakLengthUtf8 canBreak = go 0
  where
    go !count s = case fromUtf8 s of
      Nothing -> count
      Just (t, bytes)
        | canBreak (T.last t) -> count
        | otherwise -> go (count + bytes) (B.drop bytes s)

fromUtf8 :: B.ByteString -> Maybe (T.Text, Int)
-- feed in a byte at a time until some text can be decoded or there are no more
fromUtf8
  = go 0 (T.Some T.empty B.empty T.streamDecodeUtf8)
  . map B.singleton . B.unpack
  where
    go !count (T.Some t _ next) bs
      | T.null t = case bs of
          [] -> Nothing
          (b:rest) -> go (count + 1) (next b) rest
      | otherwise = Just (t, count)
```

9 word-histogram.cabal

```
name:                word-histogram
version:             0.1.0.0
synopsis:            parallel word frequency histograms
description:         Lists the most frequent words in a UTF-8 text file.
license:             BSD3
license-file:        LICENSE
author:              Claude Heiland-Allen
maintainer:          claude@mathr.co.uk
copyright:           2015 Claude Heiland-Allen
category:            Text
build-type:          Simple
cabal-version:       >=1.10

executable word-histogram
  main-is:             Main.hs
  other-modules:       Histogram
                       TakeSort
                       Utf8Chunk
```

```
    other-extensions:    BangPatterns
                         RoleAnnotations
    build-depends:       base >=4.6 && <4.9
                       , containers >=0.5.4 && <0.6
                       , bytestring >=0.10 && <0.11
                       , text >=1.2 && <1.3
                       , spawn >= 0.3 && <0.4
                       , parallel >= 3.2 && < 3.3
    default-language:    Haskell2010
    ghc-options:         -Wall -O2 -threaded -rtsopts -with-rtsopts -N
```