COMS 4995 W: Parallel Functional Programming Parallel PageRank with MapReduce

Xi Yang, Zefeng Liu - `xy2390, zl2715`

December 16, 2019

1 Introduction

PageRank (PR) is an algorithm used by Google Search to rank web pages in their search engine results.[1] PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.[2]

PageRank algorithm can be generalized to measure the importance of any type of recursive documents. It can be viewed as a node weight metric for complex networks including social networks, transportation networks, electricity networks, species networks, etc. The computing of PageRank is, therefore, a fundamental yet nontrivial problem.

In this project, we propose a parallel PageRank calculation program based on the MapReduce framework.

2 Problem Formulation

The PageRank algorithm simulates a random surfer traveling within a directed graph. Given the initial weight configuration of nodes, the algorithm outputs the probability (weight) distribution which represents the likelihood of a person randomly traveling through the edges will arrive at any particular node.

Now we formulate the Map/Reduce version of the PageRank problem.

The mapper receives the pair of node and pagerank as key, and the list of adjacent nodes as value. It maps those key-value pairs to either the pairs of node and pagerank increment or the pairs of node and list of adjacent nodes. The intermediate pairs are aggregated by key and fed to the reducers.

The reducer receives the pairs emitted by the mappers and aggregates the pagerank increments and calculates the updated pagerank value.

[1] Wikipedia contributors. "PageRank." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 15 Nov. 2019. Web. 21 Nov. 2019.

[2] "Facts about Google and Competition". Archived from the original on 4 November 2011. Retrieved 12 July 2014.

3 Implementation

3.1 Data Type Definitions

In this section, we would like to introduce some shared building blocks, the data types, upon which both our sequential and parallel solutions are implemented.

3.1.1 Nodes

Here we used *String* to represent a general node in the concerned graph. Intuitively, *Nodes* are a *set* of *Node*. An instance of *Nodes* might be {“a”, “b”, “c”}.

3.1.2 Edges

In our implementation, *Edges* are a map for which the key type is *Node*, while the value type is a *list* of nodes, representing the nodes connected to the corresponding key node. An instance of *Edges* might be {“a”: [“b”, “c”], “b”: [“c”]}.

3.1.3 Graph

The *Graph* data type represents a directed graph for whose nodes we would like to calculate the page rank values. The fields of this data type are,

- *nodes*. A set of all the nodes in this graph.
- *inEdges*. A map from a node to a list of nodes from which it is linked.
- *outEdges*. A map from a node to a list of nodes to which it links.

And there are some utility functions for this data type,

- *parseLine :: Graph -> String -> Graph*. Formulate an *inEdge* and an *outEdge* from the given *String*, add them to the given *Graph*, then return the newly constructed *Graph*. Each line of the input file should conform to the format ‘fromNode toNode’.
- *fromContent :: String -> Graph*. Given a file content, apply *parseLine* to every line of the file content to construct a Graph.
- *fromFile :: String -> IO Graph*. Given a file name, utilize *fromContent* to construct a *IO Graph*.

3.1.4 PageRank

A *PageRank* data type is a *map* from a *Node* to its current *PageRankValue*, which is a *double* in our case.

It also has some utility functions, such as the mapper and the reducer functions to compute the page rank values for a given graph with MapReduce, and also a sequential method to compute page rank. We will explain more about these utility functions in the following sections.

3.2 Sequential Solution

The function type is defined as *PageRank -> Graph -> Int -> Double -> PageRank*. We can interpret it as, "given initial PageRank, the corresponding graph, a number of iterations to compute, and a damping factor, returns the resulting PageRank after those iterations of computation in a sequential way".

Our sequential solution to compute the *PageRank* values for the next iteration works in this way,

```
for each node n in all the Nodes of the Graph do
    pr_n ← 0
    for each edge (m, n) of n's inEdges do
        num_of_out_nodes_m ← the number of nodes to which m links
        pr_previous_m ← the previous page rank value of m
        pr_delta_m ← pr_previous_m / num_of_out_nodes
        pr_n ← pr_n + pr_delta_m
    end
    update the new page rank value of node n in the new PageRank data
end
one iteration of computation is completed, return the updated PageRank data
```

3.3 Parallel Solution with customized MapReduce

The function type is also defined as *PageRank -> Graph -> Int -> Double -> PageRank*. And the interpretation is also similar, despite that this time the page rank values for the next iteration will be calculated in a parallel way.

The function type of the *mapper* is defined as *mapper :: (PageRankValue, [Node]) -> PageRank*. For each *Node* in the *Graph*, the *mapper* takes its current *PageRankValue* and the list of nodes in its *outEdges*, then produces a *map* for which the key is each of the node in its *outEdges*, and the value is its contribution to that node, defined as its current *PageRankValue* divided by the number of nodes in its *outEdges*.

The function type of the *reducer* is defined as *reducer :: [PageRank] -> PageRank*. The *reducer* merges all the outputs that the *mapper* produces. The merging rule is a simple addition for each same node.

With these definitions, our customized mapReduce function is implemented as,

```haskell
mapReduce :: (a -> b) -> ([b] -> c) -> [a] -> c
mapReduce mapper reducer input = pseq mapResult reduceResult
  where
    mapResult = parMap rpar mapper input
    reduceResult = runEval (rpar $ reducer mapResult)
```

Once the reducer completed its work in one iteration, we could simply update the page rank value for each node as $(base + d * pr)$, where $base = (1 - d)/num\_of\_nodes\_in\_graph$, d is the damping factor, pr is the corresponding value the reducer produced.

3.4 Benchmark based on External MapReduce Library

We wanted to have an external benchmark with which to compare and evaluate our MapReduce based parallel PageRank implementation.

A short web search yielded Haskell-MapReduce (`https://github.com/jdstmporter/Haskell-MapReduce`, `https://wiki.haskell.org/MapReduce_as_a_monad`) to be a promising general-purposed MapReduce library. Therefore we implemented a benchmark based on the mentioned library.

The library is implemented in a monadic fashion such that mappers and reducers can be viewed as generalized transformers of type signature `a -> ([(s,a)] -> [(s',b)])`. It provides a wrapper function `liftMR` that converts the map / reduce function into a monadic function.[3]

Given the aforementioned MapReduce library, we only need to implement conventional mapper and reducer.

According to the specification of the library, mapper should take the form of `[s] -> [(s', a)]`, where `s` is input data, `s'` is output data and `a` is output key. We implemented the mapper such that `s = (fromNode, (pageRankValue, toNodes))` and `s' = (toNode, pageRankIncrement)`. Each of the input data emits its pagerank increment contribution to all of its `toNodes`.

The reducer is implemented in a similar fashion, it takes input of the form `[(toNode, pageRankIncrement)]`. For a particular `toNode`, the pagerank increment contribution from all `fromNodes` are aggregated together, producing the pagerank value.

The evaluations to be given later in this report showed that this external benchmark has a vastly worse performance compared with our implementation.

4 Evaluation

4.1 Settings

We performed our experiments on a *MacBook Pro (15-inch, 2018)*, of which the processor is *2.2 GHz 6-core Intel Core i7*, and the memory is *16 GB 2400 MHz DDR4*.

4.2 Experiment Results

We performed our experiments by performing 10 iterations of page rank computation on two datasets with different sizes.

The first dataset is a larger fraction of the Wikipedia Note Network[4], which is 90Kb large with 11515 edges. Table 1 shows the experiment results of our MapReduce implementation. Table 2 shows the experiment results of our sequential implementation and the benchmark implementation.

[3] `https://github.com/jdstmporter/Haskell-MapReduce`
[4] `https://snap.stanford.edu/data/wiki-Vote.html`

Table 1: Experiment Result for a 90Kb Dataset (MapReduce)

N	time(s)	converted	gc'd	fizzled	total
1	59.92	0	4412	33038	37450
2	37.68	23322	858	13270	37450
3	35.86	28519	535	8396	37450
4	35.8	30830	346	4757	37450
5	36.09	32348	346	4757	37450
6	34.13	33376	300	3774	37450
7	35.62	33595	295	3560	37450
8	38.03	34186	256	3008	37450
9	40.4	34642	243	2565	37450
10	44.66	34850	226	2374	37450
11	44.29	35192	212	2046	37450
12	48.84	35487	191	1772	37450

Table 2: Experiment Result for a 90Kb Dataset (Sequential & Benchmark)

N	time(s)
seq	173.62
benchmark-1	1923.26
benchmark-6	906.79

The second dataset is a smaller fraction of the Wikipedia Note Network, which is 40Kb large with 5508 edges. Table 3 shows the experiment results of our MapReduce implementation. Table 4 shows the experiment results of our sequential implementation and the benchmark implementation.

Table 3: Experiment Result for a 40Kb Dataset (MapReduce)

N	time(s)	converted	gc'd	fizzled	total
1	26.23	0	2371	24879	27250
2	21.96	15890	844	10516	27250
4	20.38	21898	407	4945	27250
6	19.71	23888	286	3076	27250
8	24.64	24641	238	2371	27250
10	26.32	25345	196	1709	27250
12	27.99	25883	141	1226	27250

4.3 Performance Analysis

From the results, we can conclude that our MapReduce implementation is much more efficient both than the sequential version and than the benchmark implementation.

Table 4: Experiment Result for a 40Kb Dataset (Sequential & Benchmark)

N	time(s)
seq	88.26
benchmark-1	804.94
benchmark-6	472.71

We can also observe that when $N = 6$, which is equal to the number of cores, the performance of our implementation is the best. If N is set to be larger, even the convertion rate is increased, the overhead for parallelism is also increased, hence the consumed time becomes longer.

For furthur analysis, we scrutinized the event log for our MapReduce implementation running with the 40Kb dataset using *ThreadScope*. From the figure, we can observe that the bottleneck is the GC waiting time.

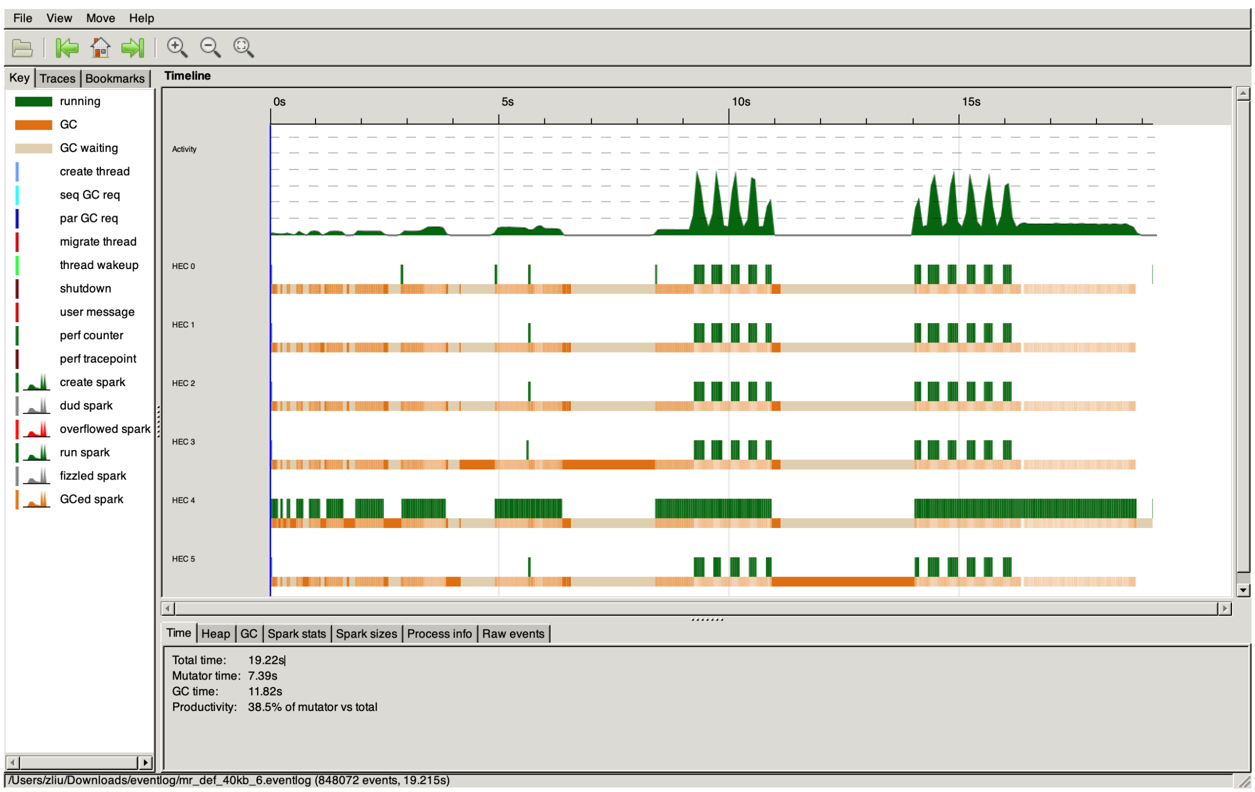

Figure 1: Eventlog for MapReduce experiment with 40Kb Dataset

A Code Listing

```haskell
{-

The main application program

Command line arguments: infilePath, outfilePath, itrs, [mode]

infilePath:  path of input file, which should be in the format of lines
             consisting of `fromNode toNode`
outfilePath: path of output file
itrs:        number of iterations in the PageRank computation
mode:        optional, mode of PageRank computation, one of {seq, mr_def,
             mr_ext}, default to mr_def
             seq:    non-parallel sequential computation
             mr_def: parallel implementation based on default MapReduce
             mr_ext: benchmark parallel implementation based on external opensourced MapReduce library

-}

module Main (main) where

import Control.Monad (when)
import System.IO (openFile, IOMode(WriteMode), hPutStrLn, hClose)
import System.Environment (getArgs, getProgName)
import System.Exit
import Data.Map as M (toList)

import ProcessData (processData)
import PageRank (computePageRankSeq, computePageRankMR)
import PageRankExt (computePageRankMRext)

main :: IO()
main = do
    progName <- getProgName
    args <- getArgs

    when (length args /= 3 && length args /= 4) $
        die $ "Usage: " ++ progName ++ " <infilePath> <outfilePath> <itrs> [mode], where\
        \ mode is one of {seq, mr_def, mr_ext}, default to mr_def"

    let infilePath : outfilePath : itrs : mode = args
        computePageRank = case mode of
            [] -> computePageRankMR
            ["mr_def"] -> computePageRankMR
            ["seq"] -> computePageRankSeq
            ["mr_ext"] -> computePageRankMRext
            _ -> error $ "Usage: " ++ progName ++ " <infilePath> <outfilePath> <itrs> [mode], where\
            \ mode is one of {seq, mr_def, mr_ext}, default to mr_def"
```

```
    (graph, pageRank) <- processData infilePath
    let resPageRank = computePageRank pageRank graph (read itrs) 0.85
    h <- openFile outfilePath WriteMode
    mapM_ (hPutStrLn h) [ n ++ ": " ++ show pr | (n, pr) <- M.toList resPageRank ]
    hClose h
```

Listing 1: app/Main.hs

```
{-

    This module contains a utility function, which
    1) reads in a graph from a input file
    2) initializes a PageRank data from the given graph
    3) returns the graph and the initial page rank

-}

module ProcessData
(processData) where

import Graph (Graph, fromFile)
import PageRank (PageRank, initFromGraph)

processData :: String -> IO (Graph, PageRank)
processData filename = do
    graph <- fromFile filename
    let pageRank = initFromGraph graph
    return (graph, pageRank)
```

Listing 2: src/ProcessData.hs

```
{-

This module defines the Graph data type. The fields are,
1) nodes:    set of all the nodes in this graph
2) inEdges:  map from a node to a list of nodes from which it is linked
3) outEdges: map from a node to a list of nodes to which it links

This module also contains some utility functions for this data type.

-}

module Graph
( Graph(..)
, Node
```

```haskell
, fromFile
) where

import qualified Data.Map as M (Map, insertWith, empty, keysSet)
import qualified Data.Set as S (Set, union, fromList, empty, toList, difference)
import System.IO (readFile)

type Node = String
type Nodes = S.Set Node
type InEdges = M.Map Node [Node]
type OutEdges = M.Map Node [Node]

data Graph = Graph { nodes    :: Nodes
                   , inEdges  :: InEdges
                   , outEdges :: OutEdges } deriving Show

-- Initial state of an empty graph
empty :: Graph
empty = Graph S.empty M.empty M.empty

-- Read in a graph from a file
fromFile :: String -> IO Graph
fromFile filename = do
    content <- readFile filename
    return $ fromContent content

fromContent :: String -> Graph
fromContent content =
    let ls = lines content
    in postProcess $ foldl parseLine empty ls
    where
        postProcess :: Graph -> Graph
        postProcess graph = foldl parseLine graph newLines
            where
                ns = nodes graph
                sinkNodes = S.difference ns $ M.keysSet $ outEdges graph
                newLines = [ n1 ++ " " ++ n2 |
                  n1 <- S.toList sinkNodes, n2 <- S.toList ns, n1 /= n2 ]

{-
  Parse each line of the input file as an edge in the graph.
  Each line of the input file should conform to the format `fromNode toNode`.
-}
parseLine :: Graph -> String -> Graph
parseLine graph line =
    let ws = words line
    in case ws of
```

```haskell
        [fromNode, toNode] ->
            Graph ns iEdges oEdges
            where
                ns = S.union (S.fromList ws) (nodes graph)
                iEdges = M.insertWith (++) toNode [fromNode] (inEdges graph)
                oEdges = M.insertWith (++) fromNode [toNode] (outEdges graph)
        _ -> error "All lines of the input  file  \
        \should be in the format of `fromNode toNode`"
```

Listing 3: src/Graph.hs

```haskell
module MapReduce
( mapReduce )

where

import Control.Parallel (pseq)
import Control.Parallel.Strategies (rpar, runEval, parMap)

mapReduce ::
    (a -> b)           -- map function
    -> ([b] -> c)      -- reduce function
    -> [a]             -- list to map over
    -> c
mapReduce mapper reducer input = pseq mapResult reduceResult
  where mapResult = parMap rpar mapper input
        reduceResult = runEval (rpar $ reducer mapResult)
```

Listing 4: src/MapReduce.hs

```haskell
{-

This module defines the PageRank data type, which is a map from a node to its
current page rank value.

This module also contains the empty definition and some utility functions for
this data type, such as the mapper and the reducer functions to compute the page
rank values for a given graph with MapReduce, and also a sequential method to
compute page rank.

-}

module PageRank
( PageRank
, initFromGraph
, computePageRankSeq
```

```haskell
, computePageRankMR
) where

import Graph (Graph(..), Node)
import MapReduce (mapReduce)
import qualified Data.Map as M (Map, empty, fromList, lookup, unionWith, toList)
import qualified Data.Set as S (toList, size)
import Data.Maybe (fromJust)

type PageRankValue = Double
type PageRank = M.Map Node PageRankValue

-- Initial state of a PageRank data for an empty graph
empty :: PageRank
empty = M.empty

{-
  Initial state of a PageRank data for a given graph, the page rank value
  of each node is the reciprocal of the number of nodes in this graph
-}
initFromGraph :: Graph -> PageRank
initFromGraph graph =
    let ns = nodes graph
        pr = 1.0 / (fromIntegral $ S.size ns) in
    M.fromList [ (n, pr) | n <- S.toList ns ]

mapper :: (PageRankValue, [Node]) -> PageRank
mapper (pr, outNodes) =
    let pr_ = pr / (fromIntegral $ length outNodes) in
    M.fromList [ (n, pr_) | n <- outNodes ]

reducer :: [PageRank] -> PageRank
reducer [] = empty
reducer [x] = x
reducer (x:xs) = M.unionWith (+) x (reducer xs)

{-
  Given initial PageRank and the corresponding graph, a number of iterations
  to compute, and a damping factor, returns the resulting PageRank after those
  iterations of computation in a parallel way with MapReduce
-}
computePageRankMR :: PageRank -> Graph -> Int -> Double -> PageRank
computePageRankMR pageRank _ 0 _ = pageRank
computePageRankMR pageRank graph itrs damping =
    let nextPageRank = computeNextPageRankMR pageRank
    in computePageRankMR nextPageRank graph (itrs-1) damping
    where
```

```haskell
        computeNextPageRankMR :: PageRank -> PageRank
        computeNextPageRankMR curPR =
            let ns = S.toList $ nodes graph
                input = map produceInput ns
                produceInput n = (pr, outNodes)
                    where
                        pr = fromJust $ M.lookup n curPR
                        outNodes = fromJust $ M.lookup n $ outEdges graph
                mrResult = mapReduce mapper reducer input
                base = (1 - damping) / (fromIntegral $ length ns)
            in M.fromList [ (n, base + damping * pr) | (n, pr) <- M.toList mrResult ]

{-
  Given initial PageRank and the corresponding graph, a number of iterations
  to compute, and a damping factor, returns the resulting PageRank after those
  iterations of computation in a sequential way
-}
computePageRankSeq :: PageRank -> Graph -> Int -> Double -> PageRank
computePageRankSeq pageRank _ 0 _ = pageRank
computePageRankSeq pageRank graph itrs damping =
    let nextPageRank = computeNextPageRank pageRank
    in computePageRankSeq nextPageRank graph (itrs-1) damping
    where
        computeNextPageRank :: PageRank -> PageRank
        computeNextPageRank curPR =
            M.fromList [ (n, computePRValue n) | n <- ns ]
            where
                ns = S.toList $ nodes graph
                iEdges = inEdges graph
                oEdges = outEdges graph
                computePRValue :: Node -> PageRankValue
                computePRValue n =
                    let inNodes = fromJust $ M.lookup n iEdges
                    in (1 - damping) / (fromIntegral $ length ns) + damping * (foldl sumUp 0 inNodes)
                    where
                        sumUp acc node =
                            let numOutNodes = length $ fromJust $ M.lookup node oEdges
                                prValue = fromJust $ M.lookup node curPR
                            in acc + prValue / (fromIntegral numOutNodes)
```

Listing 5: src/PageRank.hs

```haskell
{-
External MapReduce Lib used to implement a benchmark
GitHub repository of the MapReduce library: https://github.com/jdstmporter/Haskell-MapReduce
-}

```

```haskell
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}

-- | Module that defines the 'MapReduce' monad and exports the necessary functions.
--
--  Mapper / reducers are generalised to functions of type
--  @a -> ([(s,a)] -> [(s',b)])@ which are combined using the monad's bind
--  operation.  The resulting monad is executed on initial data by invoking
--  'runMapReduce'.
--
--  For programmers only wishing to write conventional map / reduce algorithms,
--  which use functions of type @([s] -> [(s',b)])@ a wrapper function
--  'liftMR' is provided, which converts such a function into the
--  appropriate monadic function.
module MapReduceLibExt (
-- * Types
        MapReduce,
-- * Functions
--
-- ** Monadic operations
        return, (>>=),
-- ** Helper functions
        run, distribute, lift ) where

import Data.List (nub)
import Control.Applicative ((<$>))
import Control.Monad (liftM)
import Control.DeepSeq (NFData)
import System.IO
import Prelude hiding (return,(>>=))
import Data.Digest.Pure.MD5
import Data.Binary
import qualified Data.ByteString.Lazy as B
import Control.Parallel.Strategies (parMap, rdeepseq)

-- | The parallel map function; it must be functionally identical to 'map',
--  distributing the computation across all available nodes in some way.
pMap :: (NFData b) => (a -> b) -- ^ The function to apply
        -> [a]                 -- ^ Input
        -> [b]                 -- ^ output
pMap = parMap rdeepseq

-- | Generalised version of 'Monad' which depends on a pair of 'Tuple's, both
--  of which change when '>>=' is applied.
class MonadG m where
        return :: a                 -- ^ value.
                -> m s x s a        -- ^ transformation that inserts the value
                                    --  by replacing all
```

```
                                          -- the key values with the specified
                                          -- value, leaving the data unchanged.


      (>>=) :: (Eq b,NFData s'',NFData c) =>
            m s a s' b                    -- ^ Initial processing chain
            -> ( b -> m s' b s'' c )      -- ^ Transformation to append to it
            -> m s a s'' c                -- ^ Extended processing chain


-- | The basic type that provides the MapReduce monad (strictly a generalised monad).
-- In the definition
-- @(s,a)@ is the type of the entries in the list of input data and @(s',b)@
-- that of the entries in the list of output data, where @s@ and @s'@ are data
-- and @a@ and @b@ are keys.
--
-- 'MapReduce' represents the transformation applied to data by one or more
-- MapReduce staged. Input data has type @[(s,a)]@ and output data has type
-- @[(s',b)]@ where @s@ and @s'@ are data types and @a@, @b@ are key types.
--
-- Its structure is intentionally opaque to application programmers.
newtype MapReduce s a s' b = MR { runMR :: [(s,a)] -> [(s',b)] }

-- | Make MapReduce into a 'MonadG' instance
instance MonadG MapReduce where
      return = ret
      (>>=) = bind

-- | Insert a value into 'MapReduce' by replacing all the key values with the
-- specified value, leaving the data unchanged.
ret :: a                                  -- ^ value
      -> MapReduce s x s a                -- ^ transformation that inserts the value
                                          -- into 'MapReduce' by replacing all
                                          -- the key values with the specified
                                          -- value, leaving the data unchanged.
ret k = MR (\ss -> [(s,k) | s <- fst <$> ss])

-- ^ Apply a generalised mapper / reducer to the end of a chain of processing
-- operations to extend the chain.
bind :: (Eq b,NFData s'',NFData c) =>
            MapReduce s a s' b            -- ^ Initial state of the monad
      -> (b -> MapReduce s' b s'' c)      -- ^ Transformation to append to it
      -> MapReduce s a s'' c              -- ^ Extended transformation chain
bind f g = MR (\s ->
      let
            fs = runMR f s
            gs = map g $ nub $ snd <$> fs
```

```haskell
        in
        concat $ pMap (`runMR` fs) gs)

-- | Execute a MapReduce MonadG given specified initial data. Therefore, given
--  a 'MapReduce' @m@ and initial data @xs@ we apply the processing represented
--  by @m@ to @xs@ by executing
--
--  @run m xs@
run :: MapReduce s () s' b            -- ^ 'MapReduce' representing the required processing
                -> [s]                -- ^ Initial data
                -> [(s',b)]           -- ^ Result of applying the processing to the data
run m ss = runMR m [(s,()) | s <- ss]

-- | The hash_ function. Computes the MD5 hash_ of any 'Hashable' type
hash_ :: (Binary s) => s        -- ^ The value to hash_
        -> Int                  -- ^ its hash_
hash_ s = sum $ map fromIntegral (B.unpack h)
        where
        h = encode (md5 $ encode s)

-- | Function used at the start of processing to determine how many threads of processing
--  to use. Should be used as the starting point for building a 'MapReduce'.
--  Therefore a generic 'MapReduce' should look like
--
--  @'distribute' '>>=' f1 '>>=' . . . '>>=' fn@
distribute :: (Binary s) => Int           -- ^ Number of threads across which to distribute initial data
                -> MapReduce s () s Int   -- ^ The 'MapReduce' required to do this
distribute n = MR (\ss -> [(s,hash_ s `mod` n) | s <- fst <$> ss])

-- | The wrapper function that lifts mappers / reducers into the 'MapReduce'
--  monad. Application programmers can use this to apply MapReduce transparently
--  to their mappers / reducers without needing to know any details of the implementation
--  of MapReduce.
--
--  Therefore the generic 'MapReduce' using only traditional mappers and
--  reducers should look like
--
--  @'distribute' '>>=' 'lift' f1 '>>=' . . . '>>=' 'lift' fn@
lift :: (Eq a) => ([s] -> [(s',b)])      -- traditional mapper / reducer of signature
                                         -- @([s] -> [(s',b)]@
        -> a                             -- the input key
        -> MapReduce s a s' b            -- the mapper / reducer wrapped as an instance
                                         -- of 'MapReduce'
lift f k = MR (\ss -> f $ fst <$> filter (\s -> k == snd s) ss)
```

Listing 6: src/MapReduceLibExt.hs

```haskell
{-
The benchmark PageRank computation implementation based on external opensourced MapReduce library
GitHub repository of the MapReduce library: https://github.com/jdstmporter/Haskell-MapReduce
-}

module PageRankExt
(
computePageRankMRext
) where

import Graph (Graph(..), Node)
import qualified Data.Map as M (Map, empty, fromList, lookup, unionWith, toList)
import qualified Data.Set as S (toList, size)
import Data.Maybe (fromJust)
import MapReduceLibExt (run,distribute,lift,(>>=))

type PageRankValue = Double
type PageRank = M.Map Node PageRankValue

empty :: PageRank
empty = M.empty

initFromGraph :: Graph -> PageRank
initFromGraph graph =
    let ns = nodes graph
        pr = 1.0 / (fromIntegral $ S.size ns) in
    M.fromList [ (n, pr) | n <- S.toList ns ]

mr :: Double -> Double -> Int -> [(Node, (PageRankValue, [Node]))] -> [(Node, (PageRankValue, [Node])
mr damping numNodes n state = run f state
    where
        f = distribute n MapReduceLibExt.>>= lift mapper MapReduceLibExt.>>= lift (reducer damping numNo

-- According to the specification of Haskell-MapReduce lib
-- mapper should take the form of [s] -> [(s', a)]
-- where s is input data, s' is output data and a is output key
mapper :: [(Node, (PageRankValue, [Node]))] -> [((Node, (PageRankValue, [Node])), Node)]
mapper [] = []
mapper (x:xs) = parse x ++ mapper xs
    where
        parse (n, (pr, outNodes)) =
            let pr_ = pr / (fromIntegral $ length outNodes)
            in ((n, (0, outNodes)), n) : [ ((n_, (pr_, [])), n_) | n_ <- outNodes ]

-- According to the specification of Haskell-MapReduce lib
-- reducer should take the form of [s'] -> [s'']
-- where s' is output data of mapper, s'' is output data of reducer
```

```
reducer :: Double -> Double -> [(Node, (PageRankValue, [Node]))] -> [(Node, (PageRankValue, [Node]))]
reducer _ _ [] = []
reducer damping numNodes xs@(x:_) =
    [ foldl f (fst x, ((1 - damping) / numNodes, [])) xs ]
    where f x y = (
            fst x,
            (
                (fst $ snd x) + damping * (fst $ snd y),
                (snd $ snd x) ++ (snd $ snd y)
            )
            )

computePageRankMRext :: PageRank -> Graph -> Int -> Double -> PageRank
computePageRankMRext pageRank _ 0 _ = pageRank
computePageRankMRext pageRank graph itrs damping =
    let ns = S.toList $ nodes graph
        numNodes = fromIntegral $ length ns
        oEdges = outEdges graph
        initMRinput = map toMRinput ns
        toMRinput n =
            let pr = fromJust $ M.lookup n pageRank
                outNodes = fromJust $ M.lookup n oEdges
            in (n, (pr, outNodes))
        mrOutput = mrItr initMRinput damping numNodes itrs
    in M.fromList [ (n, pr) | (n, (pr, _)) <- mrOutput ]
    where
        mrItr :: [(Node, (PageRankValue, [Node]))] -> Double -> Double -> Int -> [(Node, (PageRankVa
        mrItr input _ _ 0 = input
        mrItr input damping numNodes itrs =
            let output = mr damping numNodes 1 input
            in mrItr output damping numNodes (itrs-1)
```

Listing 7: src/PageRankExt.hs

```
import Data.Map as M (toList)

import ProcessData (processData)
import PageRank (computePageRankMR)

main :: IO ()
main = do
    (graph, pageRank) <- processData "../data/sample_input.txt"
    let resPageRank = computePageRankMR pageRank graph 10 0.85
    mapM_ putStrLn [ n ++ ": " ++ show pr | (n, pr) <- M.toList resPageRank ]
```

Listing 8: test/Spec.hs