Yefri Gaitan
Ecenaz Ozmen

MapReduce Word Count

1. Map Reduce for Word Count

We implemented a word counter in haskell using the map-reduce framework. We first designed a sequential implementation in which the map and reduce operations happen sequentially. We also implemented two different parallel implementations of the word counter using two different monads. We achieved 2x speedup when ran on 1 vs 4 cores.

The map phase involves getting each word in a text file and making a list of tuples including the word and the integer 1. (eg. [(word, 1)]) In the reduce step, we treat the tuple as a key value pair and sum up the numbers corresponding to the same key. At the end we get a mapping of each word to the number of times it was used in the file.

To parallelize this map-reduce framework, we split up our data into chunks, both for the map step and the reduce step. For mapping, we split up a list of words into chunks, resulting in a list of list of words. Each chunk went through the map step in parallel. After the map phase, each chunk in the list became a list of tuples (with (word, 1) as mentioned above) and the reduce function was applied to these same chunks. The reduce step involves building a Map and using the (+) operator as a combining function for the same keys. Reducing the chunks ran in parallel, resulting in each chunk becoming a map containing the mapping of a word to the number of times it was used. To merge these multiple maps we used a final reduce phase that merges all the maps and again uses (+) as the combining function for the same keys. At the end we convert the map into a list of tuples again, and to get the result, we sort it based on the second element of the tuples and print the 10 most repeated words.

2. How we implemented it

We ended up implementing our word counter using two different frameworks of parallelism in haskell. The first, used the **Eval** monad and basic strategies to parallelize the file split into chunks. Also, we used to **Par** monad with the Stream module borrowed from [https://github.com/simonmar/parconc-examples/blob/master/Stream.hs] Marlow's textbook. Both implementations used the lazy **ByteStrings** to improve both the performance and memory footprint of our program.

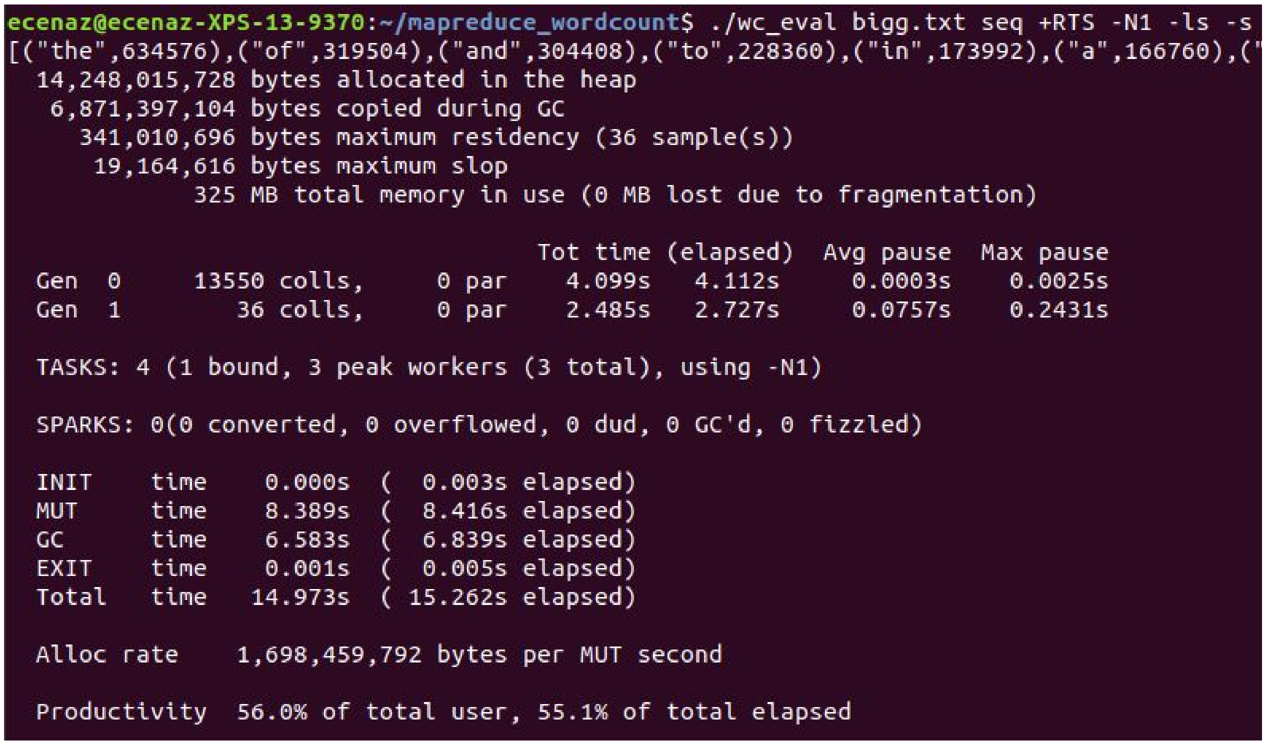
First, we designed a generic map-reduce helper function that applied provided map and reduce functions with a provided **Strategy**. The strategies were applied using **parBuffer** to take advantage of the laziness of **ByteStrings**. This allowed to regulate the amount of outstanding sparks used while the program was executing. **parBuffer** with this number sparked the full evaluation of each map and reduce operation using the **rdeepseq** strategy.

The maps operations were forced to occur first by using the **pseq** function and helper functions were added to strip non-alphabetic characters from the file split into words, all done lazily by haskell by default. We decided to hardcode the number of words per chunk as 100,000 since chunks with less words did not warrant the creation of a spark for them as the computation for map was just building a tuple. We also needed to find a threshold for this size since if we increased it too much, the heap would overflow with too much data, but if we made the chunks too small, they would be too cheap to warrant a spark each and the pool size would stay close to 0.

NOTE: In order to force the evaluation of the code, we took and printed to stdout the length of the final list of tuples of words and their final count.

Second, we used the Par monad in a different implementation as Eval did not provide the results we had anticipated. Also, the Par monad provided a higher level of abstraction since we no longer had to keep track of spark limits, laziness, and strategies while trying to understand and diagnose the tricky performance of our program. Mostly, the logic remains the same, but the list data constructor, **[]**, was replaced with a lazy implementation of Ivars represented as lists using Streams (provided by Marlow). This was done in order to provide abstracted lists that could have each elemented evaluated by a core in parallel. Since the lists were replaced by streams, the map and reduce functions were updated accordingly using the helper functions in the Stream module. Lastly, a pipeline function was used to set the order of each computation in parallel.

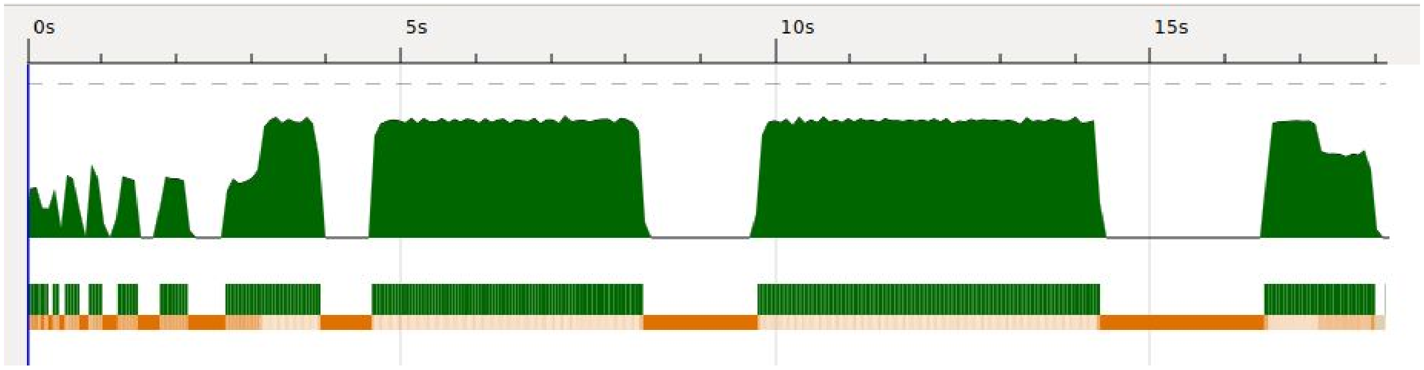
3. **Performance figures**
    a. **Sequential performance for primitive parallelism**

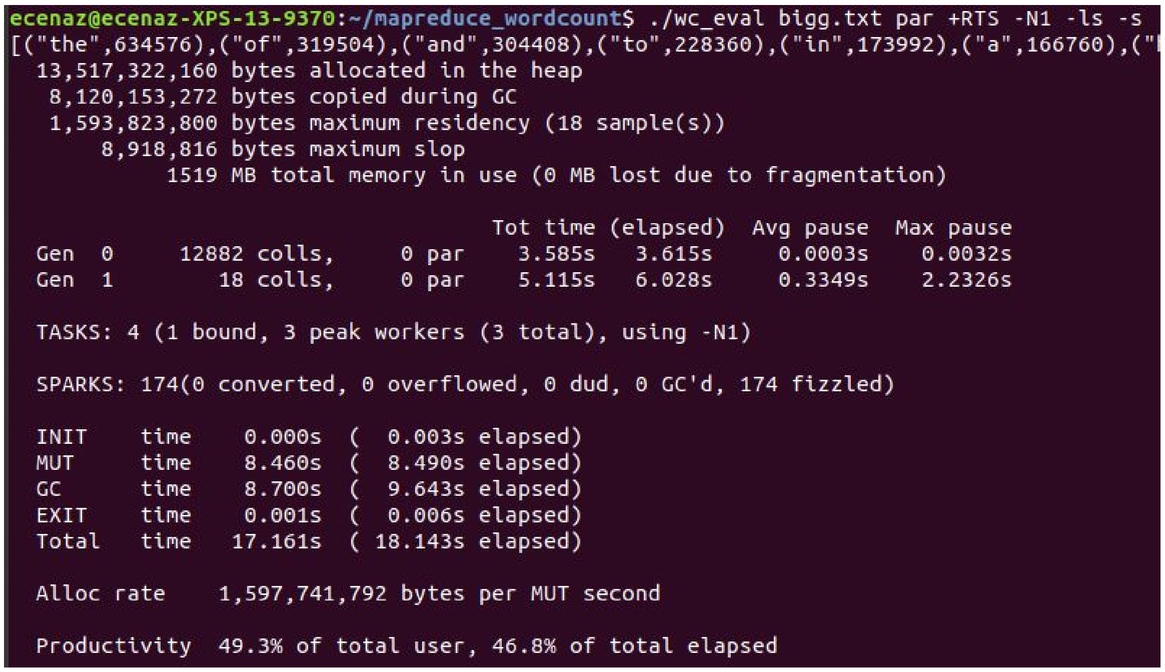


```
ecenaz@ecenaz-XPS-13-9370:~/mapreduce_wordcount$ ./wc_eval bigg.txt seq +RTS -N1 -ls -s
[("the",634576),("of",319504),("and",304408),("to",228360),("in",173992),("a",166760),("
  14,248,015,728 bytes allocated in the heap
   6,871,397,104 bytes copied during GC
     341,010,696 bytes maximum residency (36 sample(s))
      19,164,616 bytes maximum slop
             325 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     13550 colls,     0 par    4.099s   4.112s     0.0003s    0.0025s
  Gen  1        36 colls,     0 par    2.485s   2.727s     0.0757s    0.2431s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

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

  INIT    time    0.000s  (  0.003s elapsed)
  MUT     time    8.389s  (  8.416s elapsed)
  GC      time    6.583s  (  6.839s elapsed)
  EXIT    time    0.001s  (  0.005s elapsed)
  Total   time   14.973s  ( 15.262s elapsed)

  Alloc rate    1,698,459,792 bytes per MUT second

  Productivity  56.0% of total user, 55.1% of total elapsed
```

    b. **Parallel performance primitive parallelism**
        i. Speed up: 2.38x between 1 core and 4 cores
        ii. Ran with a 50.6 MB file

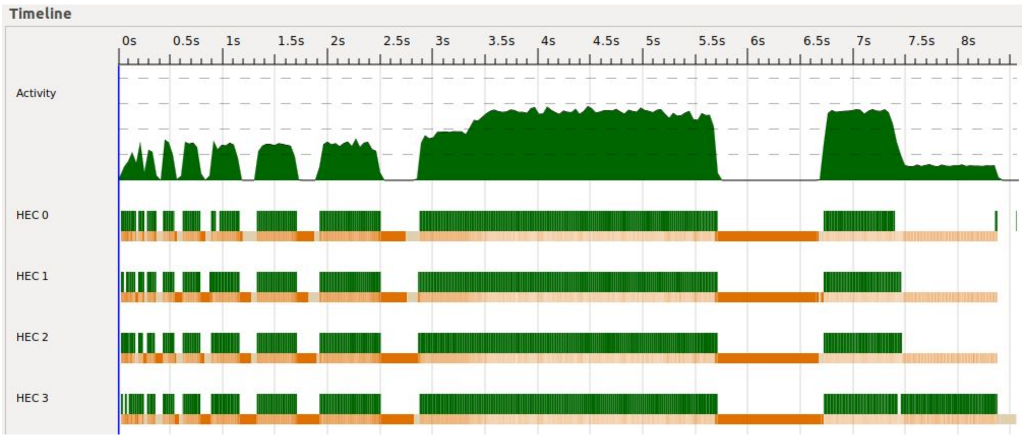
1 core:

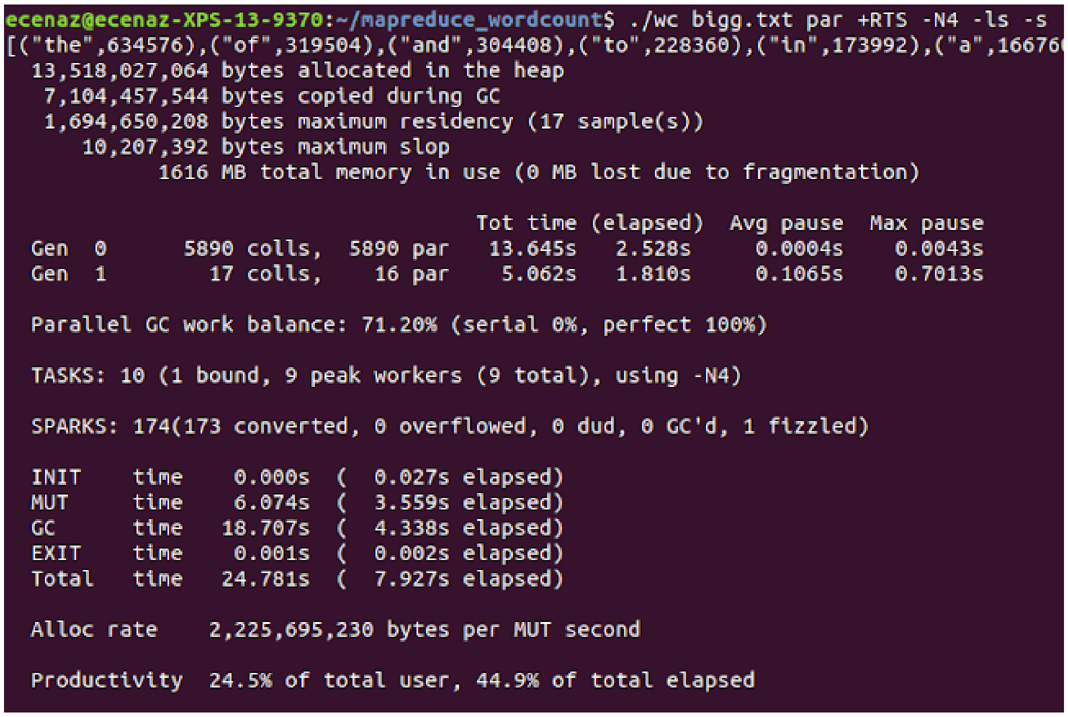




```
ecenaz@ecenaz-XPS-13-9370:~/mapreduce_wordcount$ ./wc_eval bigg.txt par +RTS -N1 -ls -s
[("the",634576),("of",319504),("and",304408),("to",228360),("in",173992),("a",166760),("
  13,517,322,160 bytes allocated in the heap
   8,120,153,272 bytes copied during GC
   1,593,823,800 bytes maximum residency (18 sample(s))
       8,918,816 bytes maximum slop
            1519 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     12882 colls,     0 par    3.585s   3.615s     0.0003s    0.0032s
  Gen  1        18 colls,     0 par    5.115s   6.028s     0.3349s    2.2326s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

  SPARKS: 174(0 converted, 0 overflowed, 0 dud, 0 GC'd, 174 fizzled)

  INIT    time    0.000s  (  0.003s elapsed)
  MUT     time    8.460s  (  8.490s elapsed)
  GC      time    8.700s  (  9.643s elapsed)
  EXIT    time    0.001s  (  0.006s elapsed)
  Total   time   17.161s  ( 18.143s elapsed)

  Alloc rate    1,597,741,792 bytes per MUT second

  Productivity  49.3% of total user, 46.8% of total elapsed
```

4 cores:



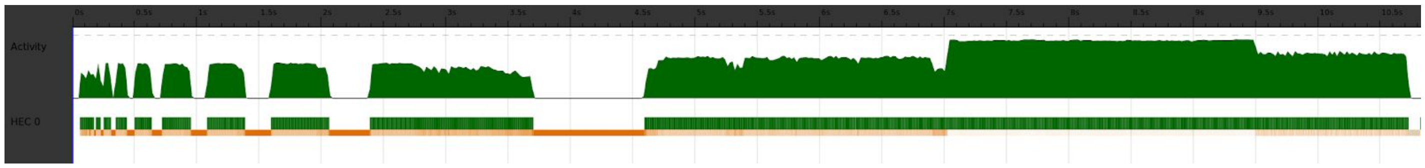


```
ecenaz@ecenaz-XPS-13-9370:~/mapreduce_wordcount$ ./wc bigg.txt par +RTS -N4 -ls -s
[("the",634576),("of",319504),("and",304408),("to",228360),("in",173992),("a",16676
  13,518,027,064 bytes allocated in the heap
   7,104,457,544 bytes copied during GC
   1,694,650,208 bytes maximum residency (17 sample(s))
      10,207,392 bytes maximum slop
            1616 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      5890 colls,  5890 par   13.645s   2.528s     0.0004s    0.0043s
  Gen  1        17 colls,    16 par    5.062s   1.810s     0.1065s    0.7013s

  Parallel GC work balance: 71.20% (serial 0%, perfect 100%)

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

  SPARKS: 174(173 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.000s  (  0.027s elapsed)
  MUT     time    6.074s  (  3.559s elapsed)
  GC      time   18.707s  (  4.338s elapsed)
  EXIT    time    0.001s  (  0.002s elapsed)
  Total   time   24.781s  (  7.927s elapsed)

  Alloc rate    2,225,695,230 bytes per MUT second

  Productivity  24.5% of total user, 44.9% of total elapsed
```

c. **Sequential Performance for Par Monad**

   Ran with a 25 MB file and only 1 core:

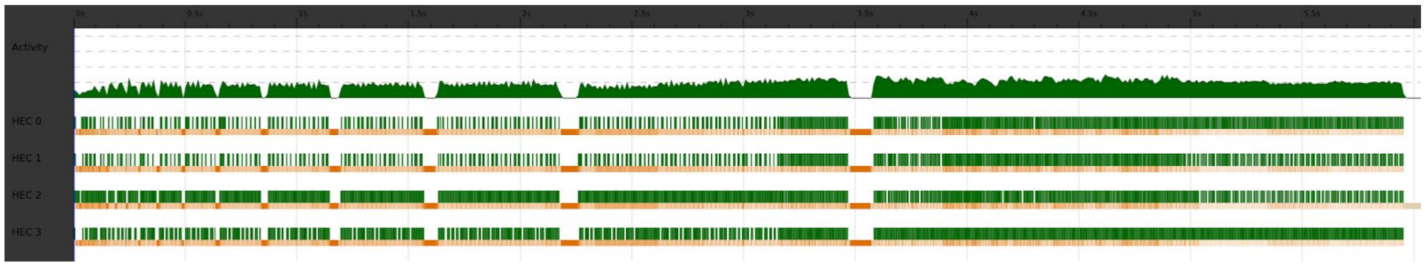
   - MUT time 6.010s (6.060s elapsed)



d. **Parallel performance for Par Monad (4 cores)**

   Ran with a 25 MB file and 4 cores

   - MUT time 4.036s (3.274s elapsed)



As seen above, it was 1.85x speedup.

4. Full Listing of our Code

wc_eval.hs

```
{-# LANGUAGE TupleSections #-}
import Control.Parallel(pseq)
import Control.Parallel.Strategies
import Data.Char(isAlpha, toLower)
import Data.Map(Map,fromListWith, toList, unionsWith)
import qualified Data.ByteString.Lazy.Char8 as B
import Data.List(sortBy)
import Data.Function(on)
import System.Environment(getArgs, getProgName)
import System.Exit(die)
main :: IO()
main = do
    args <- getArgs
    case args of
        [filename, "par"] -> do
            content <- B.readFile filename
            print $ take 10 $ sort $ wcpar content
        [filename, "seq"] -> do
            content <- B.readFile filename
            print $ take 10 $ sort $ wcseq content
        _ -> do
            pn <- getProgName
            die $ "Usage: " ++ pn ++ " <filename> <par/seq>"

wcseq :: B.ByteString -> [(B.ByteString, Int)]
wcseq = seqMapReduce wcmap wcreduce . split 100000

wcpar :: B.ByteString -> [(B.ByteString, Int)]
wcpar = finalreduce . parMapReduce rdeepseq wcmap rseq parwcreduce . split 100000



-- word count helper functions

wcmap :: [B.ByteString] -> [(B.ByteString, Int)]
wcmap = map (, 1)

parwcreduce :: [(B.ByteString, Int)] -> Map B.ByteString Int
parwcreduce = fromListWith (+)

finalreduce :: [Map B.ByteString Int] -> [(B.ByteString, Int)]
finalreduce = toList . unionsWith (+)

wcreduce :: [[(B.ByteString, Int)]] -> [(B.ByteString, Int)]
wcreduce  = toList . fromListWith (+) . concat
```

```haskell
-- map reduce library

seqMapReduce :: (a   -> b) -> ([b] -> c) -> [a] -> c
seqMapReduce mf rf = rf . map mf

parMapReduce
    :: Strategy b  -- for mapping
    -> (a   -> b)  -- map func
    -> Strategy c  -- for reducing
    -> (b -> c)  -- reduce func
    -> [a]         -- init list
    -> [c]
parMapReduce mstrat mf rstrat rf xs =
    mres `pseq` rres
  where mres = map mf xs `using` parBuffer 200 mstrat
        rres = map rf mres `using` parBuffer 200 rstrat

-- Helper functions

sort :: Ord b => [(a,b)] -> [(a,b)]
sort = sortBy (flip compare `on` snd)

split :: Int -> B.ByteString -> [[B.ByteString]]
split n bs = chunk n $ map removeNonLetters $ B.words bs

chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = let (as,bs) = splitAt n xs in as : chunk n bs

removeNonLetters :: B.ByteString -> B.ByteString
removeNonLetters = B.filter isAlpha . B.map toLower
```

wc_par.hs

```haskell
{-# LANGUAGE TupleSections #-}
import Data.Char(isAlpha, toLower)
import Data.Map(Map, toList, unionWith, insert, empty)
import qualified Data.ByteString.Lazy.Char8 as B
import Data.List(sortBy)
import Data.Function(on)
import System.Environment(getArgs, getProgName)
import System.Exit(die)
import Stream
import Control.Monad.Par
main :: IO()
main = do
    args <- getArgs
    case args of
        [filename] -> do
            content <- B.readFile filename
            print $  take 10 $ sort $ pipeline 10000 content
        _ -> do
            pn <- getProgName
            die $ "Usage: " ++ pn ++ " <filename>"

wcmap :: Stream B.ByteString -> Par (Stream (B.ByteString, Int))
wcmap = streamMap (\bs -> (bs, 1))

wcreduce :: Stream (B.ByteString, Int) -> Par (Map B.ByteString Int)
wcreduce = streamFold (insertTuple) empty

finalreduce :: Stream (Map B.ByteString Int) -> Par (Map B.ByteString Int)
finalreduce = streamFold (unionWith (+)) empty

pipeline :: Int -> B.ByteString -> [(B.ByteString, Int)]
pipeline n bs = runPar $ do
    s0 <- streamFromList (chunk n (map removeNonLetters (B.words bs)))
    s1 <- streamMap (runPar . streamFromList) s0 --using runPar to unbox
    s2 <- streamMap (runPar . wcmap) s1
    s3 <- streamMap (runPar . wcreduce) s2
    s4 <- finalreduce s3
    return $ toList s4


-- helper functions
chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = let (as,bs) = splitAt n xs in as : chunk n bs

removeNonLetters :: B.ByteString -> B.ByteString
removeNonLetters = B.filter isAlpha . B.map toLower


insertTuple :: Map B.ByteString Int -> (B.ByteString, Int) -> Map B.ByteString Int
insertTuple m (k,v) = insert k v m
```

```haskell
sort :: Ord b => [(a,b)] -> [(a,b)]
sort = sortBy (flip compare `on` snd)
```