HipgRap - Report

Bicheng Gao (bg2640) \ Kangwei Ling (kl3076)

Introduction

Grep is a very useful command line tool for search patterns in text-based files. We would like to implement such a tool in haskell, called HipgRap, for purposes of both practising programming with haskell and gaining experience on building system tools focusing on performance. A typical grep tool has three parts: find the files to search, search patterns in those files, and print the result. Our implementation will focus on the search of string literals.

Motivation

Current haskell implementation of grep in the library is not efficient enough, especially when you compare it to other implementation in C(GNU grep) and Rust(ripgrep). The basic motivation is to use the parallelism feature of Haskell to boost the execution of reading files, finding matches in each line.

Implementation

We have implemented six versions in total, each focuses on different level of parallelism, check out the following table for their differences.

File	Description
SeqGrap.hs	Sequential version of the grep, read the entire file and run the BM algorithm line by line.
StaPar.hs	Static partitioning version, read the entire file as SeqGrap.hs, however, it splits the input into several chunks (we choose 4) and run BM algorithm on each chunk in parallel.
DynPar.hs	Dynamic partitioning version, similar to StaPar.hs, it sparks a thread on every line instead of a chunk of input.

ParIO.hs	Parallel IO version, this one is similar to the StaPar.hs too. The new thing is that we add parallel support for the input, we split the input file in many chunks, multiple threads will start at different offset for the input file.
Grap.hs	Multiple-file parallel version, it tries to solve the problem in another level of parallelism. Many files can be searched at same time.
SeqHipGrap.hs	Another version of SeqGrap that searches in the whole directory recursively if a directory is passed as the argument

There are two levels of parallelism in our implementations: file-level parallelism and line-based/chunk-based parallelism.

The file-level parallelism is used in Grap.hs, since it must efficiently search in multiple files, which makes us want to introduce concurrency here, so we can search different files concurrently. We have used explicit parallelism with Channels, forkIO. Specifically, we create a dedicated thread to traverse the directories and add files that need to be searched to the channel, and we create several worker threads that read from the channel to grab files to work on. We spent some time on learning how to implement such a worker threadpool solution as it is quite different than other imperative language since we must deal with the monad environment.

Line-based/chunk-based parallelism is used for single file searching, as implemented in StaPar.hs, DynPar.hs and ParIO.hs. We split the search on each line/chunk by using the Parallel library, specifically, rpar, parMap, rseq.

At first, we only implemented the version without the parallelism on the IO, it turns out they are pretty slow, and even can not compete with the sequential version. Since we chose Boyer Moore algorithm for string matching, the running time is linear to the input size, which makes the IO become the bottleneck. After some investigations, we found Haskell supports the POSIX way of reading. It's possible to read like `pread`, we can specify an offset to a file descriptor, and reading a specific number of bytes from that position. By this way, we can make multiple threads starting from different offset and

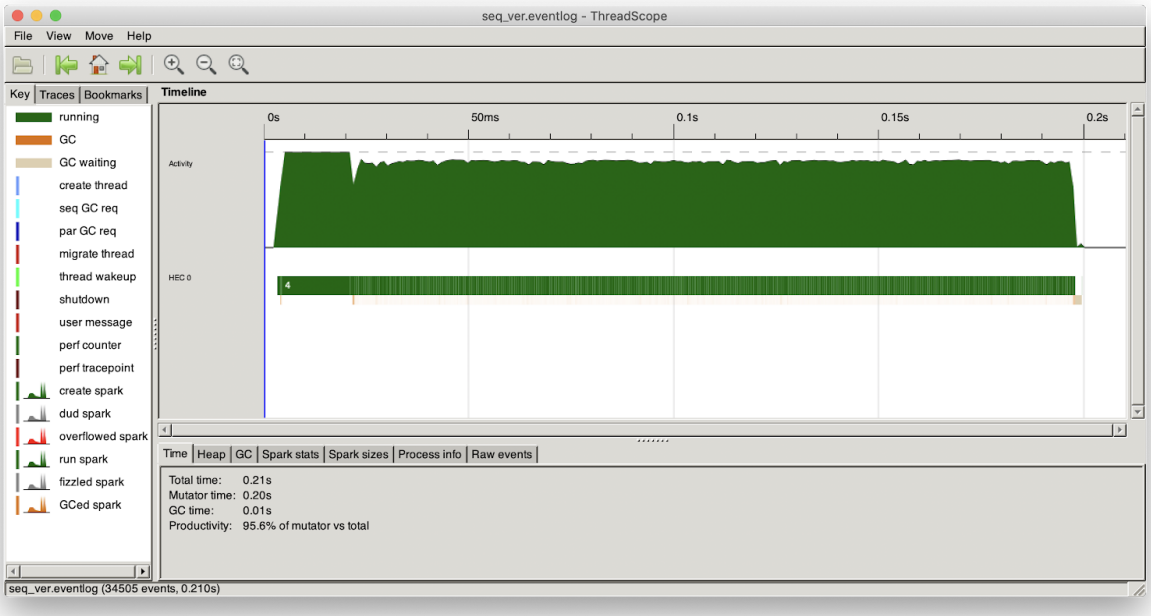
reading their own parts. It makes a pretty good performance improvement, and beaten all other versions. The following table is the performance test result.

Single File Test Results

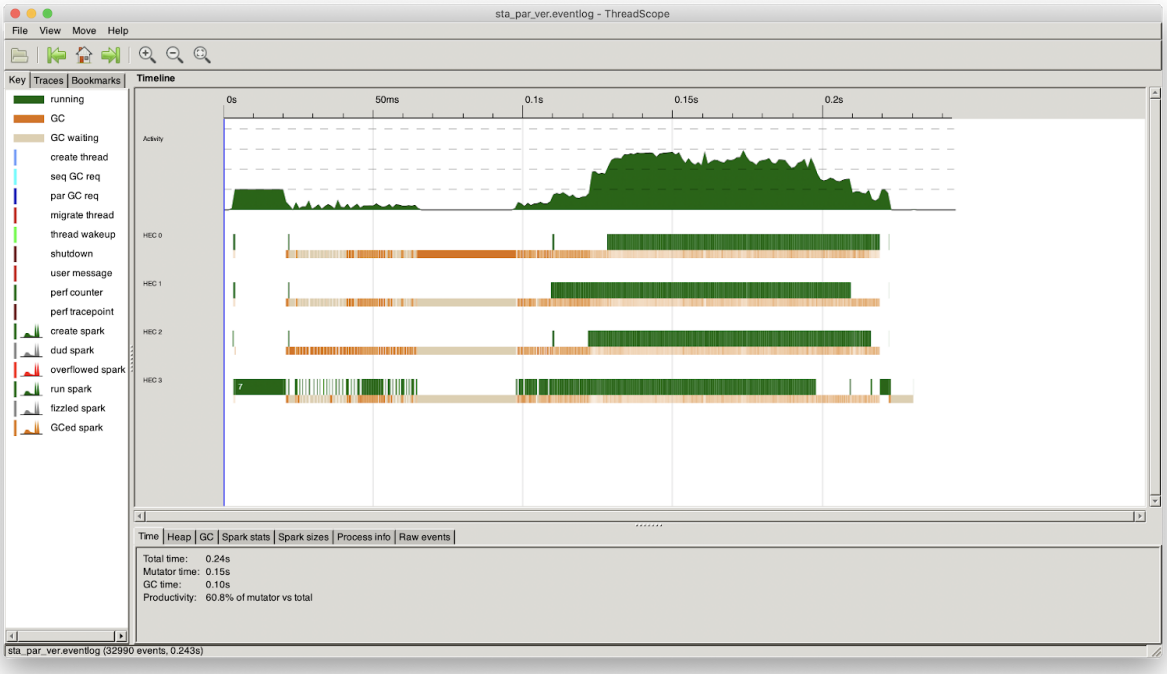
Implementation	Total time	Test parameters	All running on a file with size around 25MB.
SeqGrap.hs	0.224 s	+RTS -N1 -ls -s	
StaPar.hs	0.263 s	+RTS -N4 -ls -s	
DynPar.hs	0.514 s		
ParIO.hs	0.139 s		
Grap.hs	0.627 s		

Single File Test Thread Scope Results

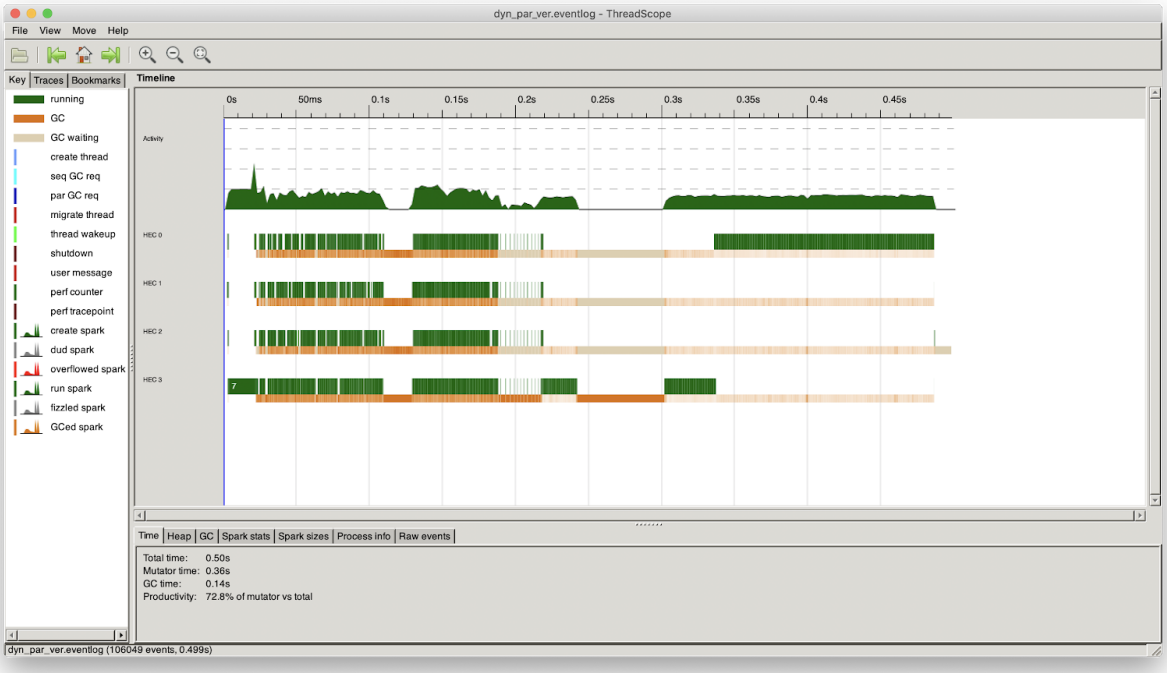
- SeqGrap.hs



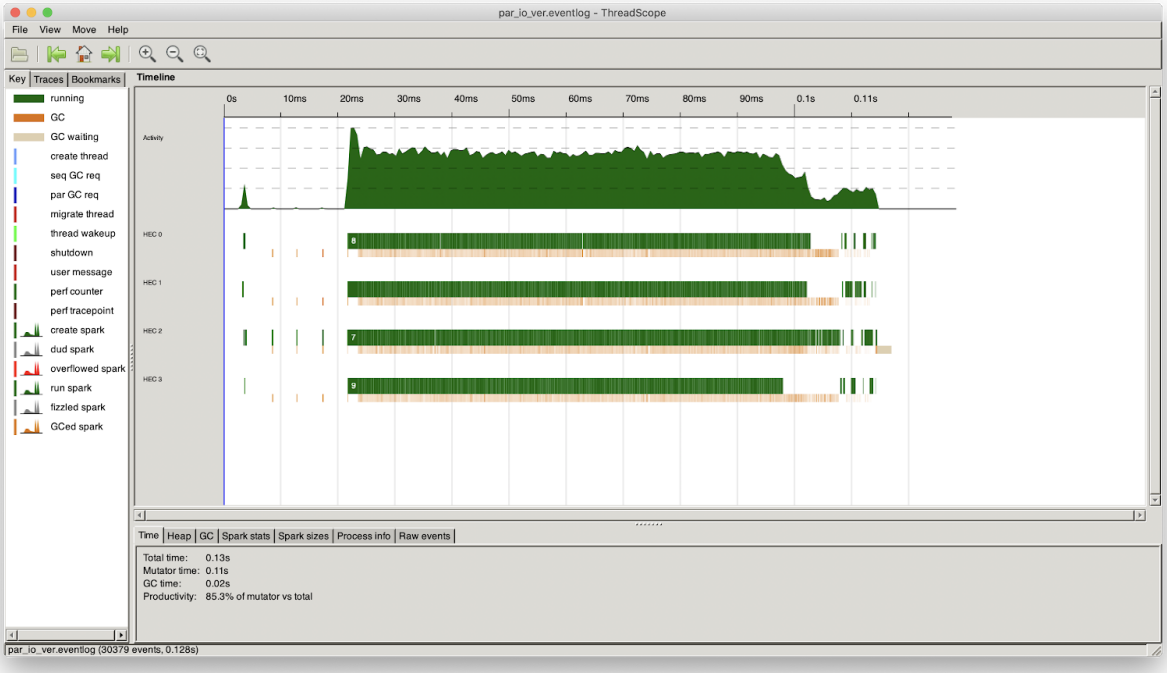
- StaPar.hs



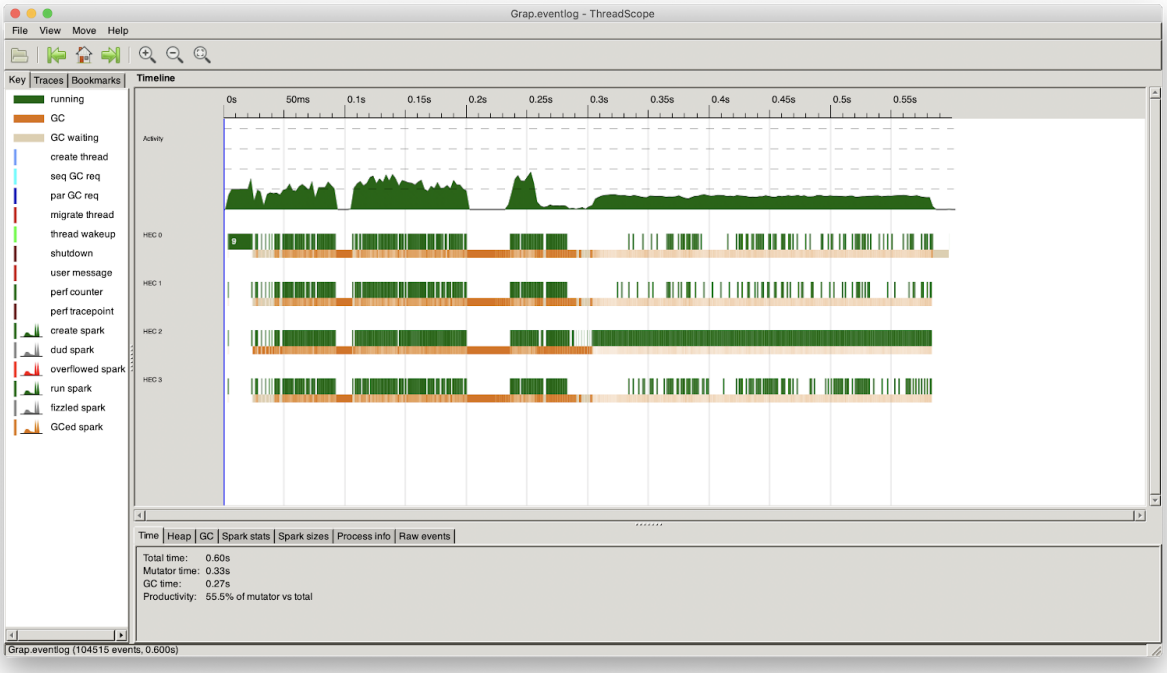
- DynPar.hs



- ParIO.hs



- Grap.hs



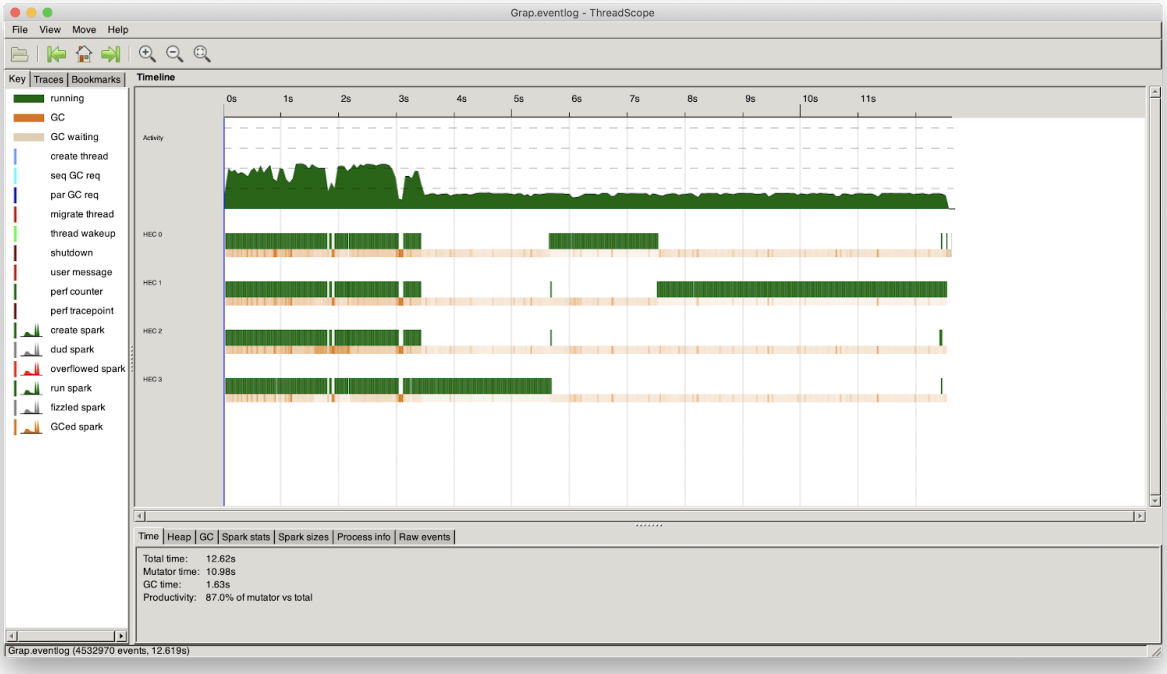
As we can see from the results, the ParIO version which parallels the chunk reading of the file runs fastest. The IO part is indeed the bottleneck of grep since the string matching algorithm runs pretty fast enough compared with IO.

Multi-files Searching Test Results

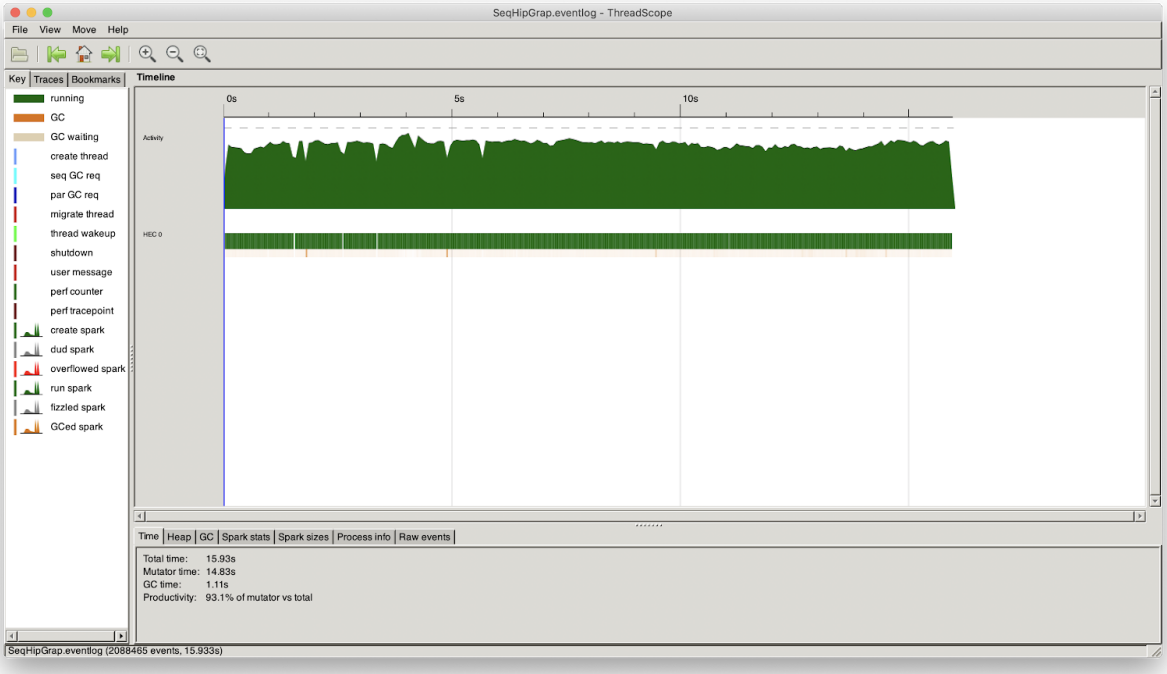
Implementation	Total time	Parameter	Searching for “fair” in linux kernel source
Grap.hs	12.737 s	+RTS -N4 -ls -s	
SeqHipGrap.hs	15.955 s	+RTS -N1 -ls -s	

Multi-Files Test Thread Scope Results

- Grap.hs



- SeqHipGrap.hs



For recursively searching in multiple files and directories, our parallel or concurrent version Grap.hs slightly beat the sequential version. We actually expect its performance to be much better than the sequential version. We speculate it is because our implementation brings too much overhead as to the sequential version.

Remarks

One thing remaining is to solve the corner cases that one line might be split into multiple chunks for our Parallel IO version, the simple way is to concatenate the last line of i-th chunk and first line of i+1-th chunk, and check it separately. Since we only split the input into 4 chunks, here we ignore its influence to the performance.

Code Listing

SeqGrap.hs

```haskell
import System.Environment
import Data.Maybe
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BC8
import qualified Data.ByteString.Search as BS

solve :: BC8.ByteString  -> BC8.ByteString -> Maybe BC8.ByteString
solve pat text
 | check == True = Just text
 | otherwise = Nothing
 where check = not . null $ BS.indices pat text

printMaybe :: Maybe BC8.ByteString -> IO ()
printMaybe (Just x) = BC8.putStrLn x
printMaybe Nothing = return ()

main :: IO ()
main = do
 [pat, filename] <- getArgs
 contents <- B.readFile filename
 let res = map (solve (BC8.pack pat)) $ BC8.lines contents
 mapM_ printMaybe (filter isJust res)
```

StaPar.hs

```haskell
import System.Environment
import Data.Maybe
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BC8
import qualified Data.ByteString.Search as BS
import Control.Parallel.Strategies
import Control.DeepSeq

solve :: BC8.ByteString  -> BC8.ByteString -> Maybe BC8.ByteString
solve pat text
```

```
 | check = Just text
 | otherwise = Nothing
 where check = not . null $ BS.indices pat text

printMaybe :: Maybe BC8.ByteString -> IO ()
printMaybe (Just x) = BC8.putStrLn x
printMaybe Nothing = return ()

main :: IO ()
main = do
 [pat, filename] <- getArgs
 contents <- B.readFile filename
 let as = BC8.lines contents
     len = length as `div` 4
     (a, bs) = splitAt len as
     (b, cs) = splitAt len bs
     (c, d) = splitAt len cs
     sol = runEval $ do
       a' <- rpar (force (map (solve (BC8.pack pat)) a))
       b' <- rpar (force (map (solve (BC8.pack pat)) b))
       c' <- rpar (force (map (solve (BC8.pack pat)) c))
       d' <- rpar (force (map (solve (BC8.pack pat)) d))
       _ <- rseq a'
       _ <- rseq b'
       _ <- rseq c'
       _ <- rseq d'
       return (a' ++ b' ++ c' ++ d')
       -- return (length a' + length b' + length c' + length d')
 mapM_ printMaybe (filter isJust sol)
 -- print sol
```

DynPar.hs

```
import System.Environment
import Data.Maybe
import qualified Data.ByteString as B
```

```haskell
import qualified Data.ByteString.Char8 as BC8
import qualified Data.ByteString.Search as BS
import Control.Parallel.Strategies hiding (parMap)

solve :: BC8.ByteString  -> BC8.ByteString -> Maybe BC8.ByteString
solve pat text
 | check = Just text
 | otherwise = Nothing
 where check = not . null $ BS.indices pat text

printMaybe :: Maybe BC8.ByteString -> IO ()
printMaybe (Just x) = BC8.putStrLn x
printMaybe Nothing = return ()

parMap :: (a -> b) -> [a] -> Eval [b]
parMap _ [] = return []
parMap f (a:as) = do b <- rpar (f a)
                     bs <- parMap f as
                     return (b:bs)

main :: IO ()
main = do
 [pat, filename] <- getArgs
 contents <- B.readFile filename
 let res = runEval (parMap (solve (BC8.pack pat)) $ BC8.lines contents)
 mapM_ printMaybe (filter isJust res)
 -- print $ length res
```

ParIO.hs

```haskell
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE PackageImports #-}
{-# LANGUAGE ScopedTypeVariables #-}

import System.Environment
import Data.Maybe
```

```haskell
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BC8
import qualified Data.ByteString.Search as BS
import Control.Parallel.Strategies
import Control.DeepSeq
import qualified System.Posix.IO as PIO
import qualified "unix-bytestring" System.Posix.IO.ByteString as PIOB
import System.Posix.Types
import qualified System.Posix as P

solve :: BC8.ByteString  -> BC8.ByteString -> Maybe BC8.ByteString
solve pat text
 | check = Just text
 | otherwise = Nothing
 where check = not . null $ BS.indices pat text

printMaybe :: Maybe BC8.ByteString -> IO ()
printMaybe (Just x) = BC8.putStrLn x
printMaybe Nothing = return ()

getFileSize :: String -> IO FileOffset
getFileSize path = do
   stat <- P.getFileStatus path
   return (P.fileSize stat)

main :: IO ()
main = do
 [pat, filename] <- getArgs
 filesize <- getFileSize filename
 fd <- PIO.openFd filename PIO.ReadOnly (Just (CMode 0440)) PIO.defaultFileFlags

 let chunk_size_bt::ByteCount = fromIntegral (filesize `div` 4)
 let rm_bt::ByteCount = fromIntegral filesize - 3 * chunk_size_bt
 let chunk_size_off::FileOffset = filesize `div` 4
 ca <- PIOB.fdPread fd chunk_size_bt 0
 cb <- PIOB.fdPread fd chunk_size_bt chunk_size_off
 cc <- PIOB.fdPread fd chunk_size_bt (chunk_size_off * 2)
 cd <- PIOB.fdPread fd rm_bt (chunk_size_off * 3)
```

```
 let sol = runEval $ do
       a' <- rpar (force (map (solve (BC8.pack pat)) $ BC8.lines ca))
       b' <- rpar (force (map (solve (BC8.pack pat)) $ BC8.lines cb))
       c' <- rpar (force (map (solve (BC8.pack pat)) $ BC8.lines cc))
       d' <- rpar (force (map (solve (BC8.pack pat)) $ BC8.lines cd))
       _ <- rseq a'
       _ <- rseq b'
       _ <- rseq c'
       _ <- rseq d'
       return (a' ++ b' ++ c' ++ d')
       -- return (length a' + length b' + length c' + length d')
 mapM_ printMaybe (filter isJust sol)
 -- print sol
```

Grap.hs

```
import Control.Monad(forM_, forever)
import Control.Concurrent.STM
import Control.Concurrent(forkIO, forkFinally, threadDelay)
import Control.Parallel
import Control.Parallel.Strategies(parMap, rpar)
import System.Directory (doesDirectoryExist, getDirectoryContents)
import System.FilePath ((</>))
import qualified Data.ByteString.Char8 as B
import qualified Data.ByteString.Search as BS
import System.Environment(getArgs, getProgName)

numOfWorkers = 4


grabFiles :: FilePath -> Bool -> TChan (Maybe FilePath) -> IO ()
grabFiles fpath recursive chan = do
   walkDir fpath recursive
   -- add terminators
   forM_ [0..numOfWorkers-1] $ \_ -> atomically $ writeTChan chan Nothing
 where
```

```haskell
    fileFilter fname = head fname /= '.'
    walkDir :: FilePath -> Bool -> IO ()
    walkDir path recursive = do
        isDir <- doesDirectoryExist path
        if isDir
            then do
                names <- getDirectoryContents path
                let properNames = filter fileFilter names
                forM_ properNames $ \fname -> walkDir (path </> fname) recursive
            else atomically $ writeTChan chan (Just path)


runGrap :: String -> FilePath -> IO ()
runGrap pat filepath = do
    jobChan <- newTChanIO
    outChan <- newTChanIO
    let bpat = B.pack pat

    forkIO $ grabFiles filepath True jobChan

    -- start workers
    forM_ [0..numOfWorkers-1] $ \i -> forkIO $ runWorker i jobChan outChan bpat

    -- gather result and print
    printResults outChan


runWorker :: Int -> TChan (Maybe FilePath) -> TChan Output -> B.ByteString -> IO ()
runWorker wid jobChan outChan pat = runLoop
    where
        runLoop = do
            filename <- atomically $ readTChan jobChan
            case filename of
                Just fname -> do
                    searchInFile pat fname outChan
                    runLoop
                Nothing -> atomically $ writeTChan outChan Terminated

searchInFile :: B.ByteString -> FilePath -> TChan Output -> IO()
```

```
searchInFile pat fname outChan = do
   content <- B.readFile fname

   let augLines = zip [1..] $ B.lines content
       matches = filter (\al@(_, line) -> not . null $ BS.indices pat line) augLines
   atomically $ writeTChan outChan (Matches fname matches)

printResults :: TChan Output -> IO ()
printResults outChan = loop 0
   where
       loop i =
           if i == numOfWorkers
               then return ()
           else do
               output <- atomically $ readTChan outChan
               case output of
                   Terminated -> loop (i + 1)
                   Matches fpath results -> do
                       forM_ results $ \(ln, txt) -> putStrLn $ fpath ++ ":" ++ show
ln ++ ": " ++ B.unpack txt
                       loop i


data Output = Terminated | Matches FilePath [(Int, B.ByteString)]


main :: IO ()
main = do
   [pat, filename] <- getArgs
   runGrap pat filename
```

SeqHipGrap.hs

```haskell
import System.Environment
import Data.Maybe
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BC8
import qualified Data.ByteString.Search as BS
import System.Directory (doesDirectoryExist, getDirectoryContents)
import System.FilePath ((</>))
import Control.Monad(forM_, forever)

solve :: BC8.ByteString  -> BC8.ByteString -> Maybe BC8.ByteString
solve pat text
 | check == True = Just text
 | otherwise = Nothing
 where check = not . null $ BS.indices pat text

printMaybe :: Maybe BC8.ByteString -> IO ()
printMaybe (Just x) = BC8.putStrLn x
printMaybe Nothing = return ()

main :: IO ()
main = do
 [pat, filename] <- getArgs
 let bpat = BC8.pack pat

 grapFiles filename True bpat

grapFiles :: FilePath -> Bool -> BC8.ByteString -> IO ()
grapFiles fpath recursive pat = walkDir fpath recursive
 where
   fileFilter fname = head fname /= '.'
   walkDir :: FilePath -> Bool -> IO ()
   walkDir path recursive = do
       isDir <- doesDirectoryExist path
       if isDir
           then do
               names <- getDirectoryContents path
               let properNames = filter fileFilter names
```

```
                forM_ properNames $ \fname -> walkDir (path </> fname) recursive
            else grap path pat

grap :: FilePath -> BC8.ByteString -> IO ()
grap fpath pat = do
   contents <- B.readFile fpath
   let res = map (solve pat) $ BC8.lines contents
       linedRes = zip [1..] res
       finalRes = filter (isJust . snd) linedRes
   forM_ finalRes $ \(ln, txt) -> putStrLn $ fpath ++ ":" ++ show ln ++ ": " ++
BC8.unpack (fromJust txt)
```