Topics in Functional Programming - Assignment 1

By Kevin Hennessy - 11726665

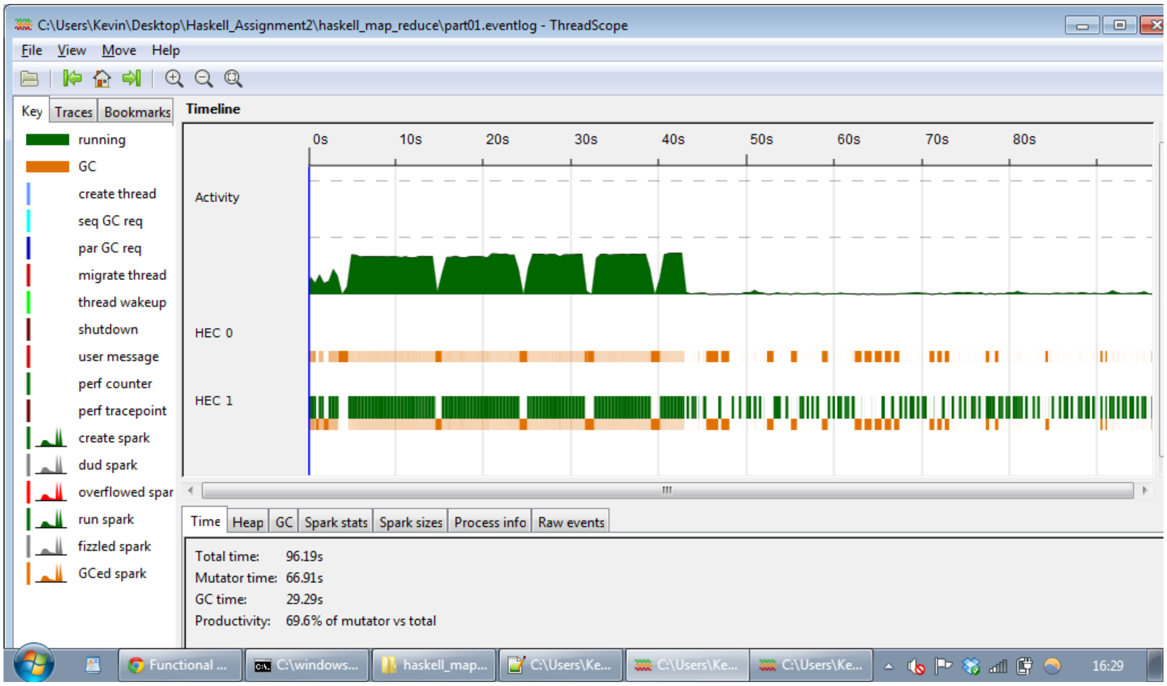
I started the assignment by writing the word count program sequentially, then creating a parallel version using the framework from Real World Haskell. This is my baseline for which I test the granularity of the parallelism.

For all tests I used a Intel Atom N570 CPU, which has 2 cores and 4 threads. For my file I used the Peter Norvig's file big.txt, a 6 MB text file containing all of Project Gutenbergs Free Ebooks.

Initial parallel program:

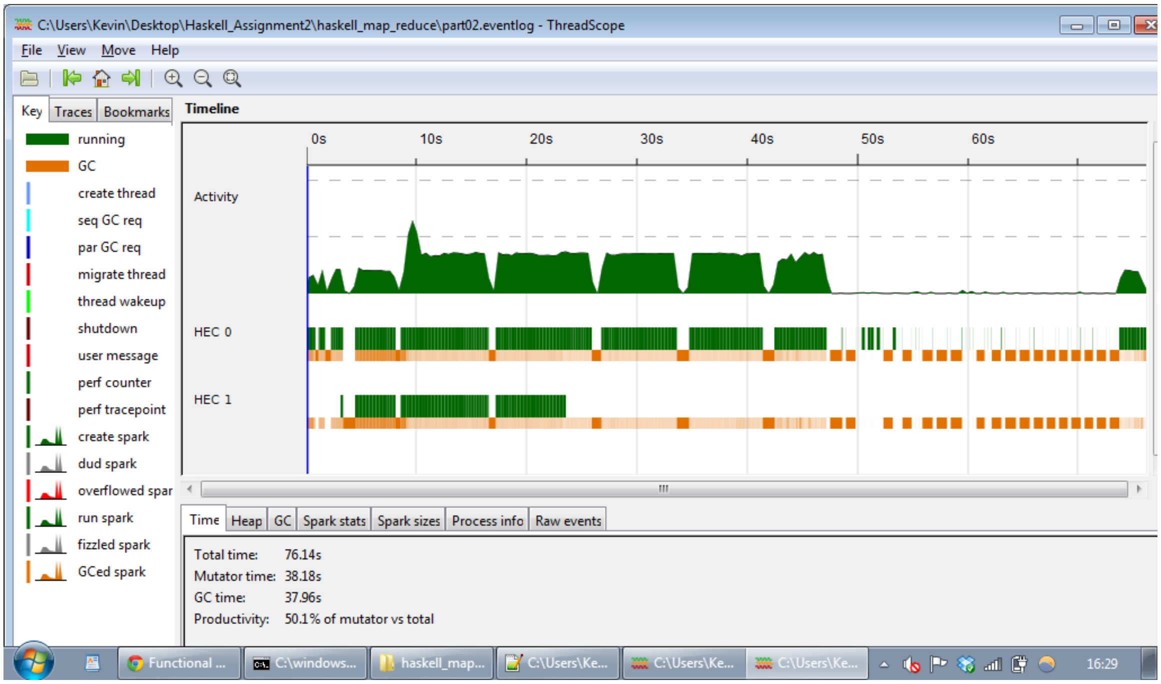
```
mapReduce
    :: Strategy b    -- evaluation strategy for mapping
    -> (a -> b)      -- map function
    -> Strategy c    -- evaluation strategy for reduction
    -> ([b] -> c)    -- reduce function
    -> [a]           -- list to map over
    -> c

mapReduce mapStrat mapFunc reduceStrat reduceFunc input =
    mapResult `pseq` reduceResult
  where mapResult    = parMap mapStrat mapFunc input
        reduceResult = reduceFunc mapResult `using` reduceStrat

stringToWordCountMap :: String -> Map.Map String Int
stringToWordCountMap  = Map.fromList . Prelude.map (head &&& length) . group . sort . words . Prelude.map toLower

combineWordCountMaps :: Map.Map String Int -> Map.Map String Int -> Map.Map String Int
combineWordCountMaps map1 map2 = Map.unionWith (+) map1 map2

reduceWordCountMaps :: [ Map.Map String Int] -> Map.Map String Int
reduceWordCountMaps  (x:[]) = x
reduceWordCountMaps  (x:xs) = combineWordCountMaps x (reduceWordCountMaps xs)

main = do (fileName:_) <- getArgs
          fileExists <- doesFileExist fileName
          if fileExists
              then do contents <- readFile fileName
                      let fileInLines = lines contents
                          result = mapReduce rpar stringToWordCountMap rpar reduceWordCountMaps fileInLines

                      putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
                      putStrLn $ "result = " ++ show result ++ "."
              else do putStrLn "The file doesn't exist!"
```

Sequential version:



```
Total time:     96.19s
Mutator time:   66.91s
GC time:        29.29s
Productivity:   69.6% of mutator vs total
```

Paralell version (code above:)



```
Total time:     76.14s
Mutator time:   38.18s
GC time:        37.96s
Productivity:   50.1% of mutator vs total
```

As you can see that while slightly faster than the sequential version the parallel version does not have a very good spread between the CPU's.

Tuning:

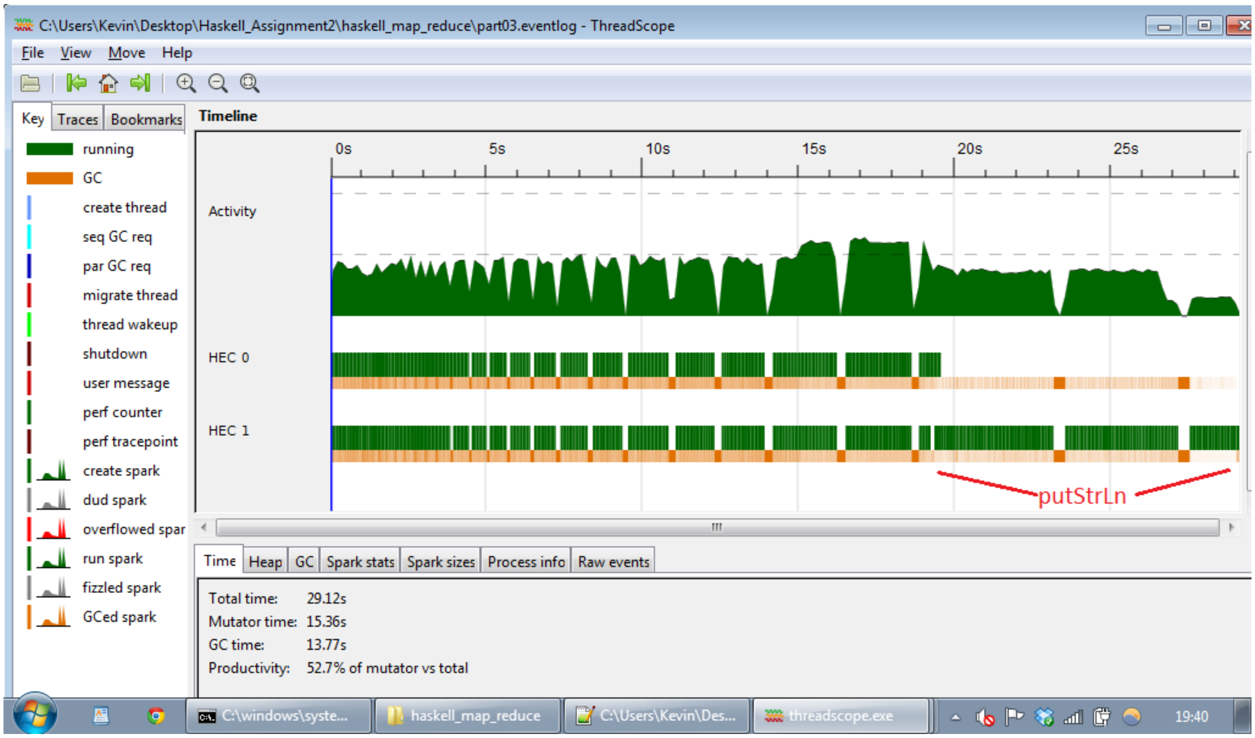
After doing some reading, I wrote a tuned version of the program. (See bolded parts):

```
mapReduce
    :: Strategy b    -- evaluation strategy for mapping
    -> (a -> b)      -- map function
    -> Strategy c    -- evaluation strategy for reduction
    -> ([b] -> c)    -- reduce function
    -> [a]           -- list to map over
    -> c

-- file: ch24/MapReduce.hs
mapReduce mapStrat mapFunc reduceStrat reduceFunc input =
    mapResult `pseq` reduceResult
  where mapResult    = parMap mapStrat mapFunc input
        reduceResult = reduceFunc mapResult `using` reduceStrat

chunkToWordCountMap :: [String] -> Map.Map String Int
chunkToWordCountMap  = Map.fromList . parMap rseq (head &&& length) . group . sort . words . Prelude.map toLower .
concat --Granularity too low for first map

combineWordCountMaps :: Map.Map String Int -> Map.Map String Int -> Map.Map String Int
combineWordCountMaps map1 map2 = Map.unionWith (+) map1 map2

reduceWordCountMaps :: [Map.Map String Int] -> Map.Map String Int
reduceWordCountMaps  (x:[]) = x
reduceWordCountMaps (x:xs) = combineWordCountMaps x (reduceWordCountMaps xs)

main = do (fileName:_) <- getArgs
          fileExists <- doesFileExist fileName
          if fileExists
              then do contents <- readFile fileName
                      let fileInChunks = chunksOf 500 $ lines contents
                          result = mapReduce rpar chunkToWordCountMap rpar reduceWordCountMaps fileInChunks
                      putStrLn $ "result = " ++ show result ++ "."
              else do putStrLn "The file doesn't exist!"
```

The main change I made was chunking the file into chunks of 500 lines and did my parallelization using each of these instead of each line. (File big.txt has 128457 lines, so this gave a dramatic increase in speed.)

I also adjusted the chunkToWordCountMap function to do the word-count tuple building from the groups in parallel. I did not do the mapping of toLower onto the concatenated chunks in parallel as it created too much parallelism and was not worth it.

This was the result:



Total time: 29.12s
Mutator time: 15.36s
GC time: 13.77s
Productivity: 52.7% of mutator vs total

As you can see this finished in only 30 seconds and has a very good spread between each core for the mapreduce operation. (The large sequential part at the end of HEC1 is the putStrLn operation, which took quite a long time for such a big file.)