CS4012: Assignment 1

Matt Donnelly – 11350561

Installation

All parts of my solution to this assignment can be installed using cabal by simply cd-ing into the respective folder and running the following command:

```
cabal install --only-dependecies && cabal build
```

This will create a folder called 'dist' which will contain the program binary. The program should be run with a path to a text file passed as the first argument. I provided a couple of example ones I used during my own testing

Part 1

For part 1, I began by simply taking the the non-parallel MapReduce function type signature from the lecture slides and built the function body out from there. I then started working on the mapper and reducer functions for calculating the word frequency table. Initially, my mapper function was designed to be applied to each character in the file with type `Char -> Char` and would simply covert each character to lowercase and filter all non-alphabetical characters. However, in order to to take advantage of future parallelism I decided to change this to a function to be applied to each line in the file's contents instead. This changes the type of mapper to `String -> FrequencyTable`, where FrequencyTable is a type alias for a list of tuples containing a String representing a word in the table and an Int for the number of times the word occurs. Inside the function, I apply the same stripping before to the string, then split on each word, sort the words, group them and map a function returning the head and length of each group to get a frequency table for a line.

Inside my reducer I take in a list of FrequencyTables (i.e. a FrequencyTable for each line in the file) as input. I then concatenate the items in the list of FrequencyTables and apply map a function converting each element to Data.Map singleton. Using this list of Data.Map singletons I perform a fold using `unionWith`, to join all the maps and sum the non-unique elements, and finally convert the list of Maps back to a list of String Int pairs.

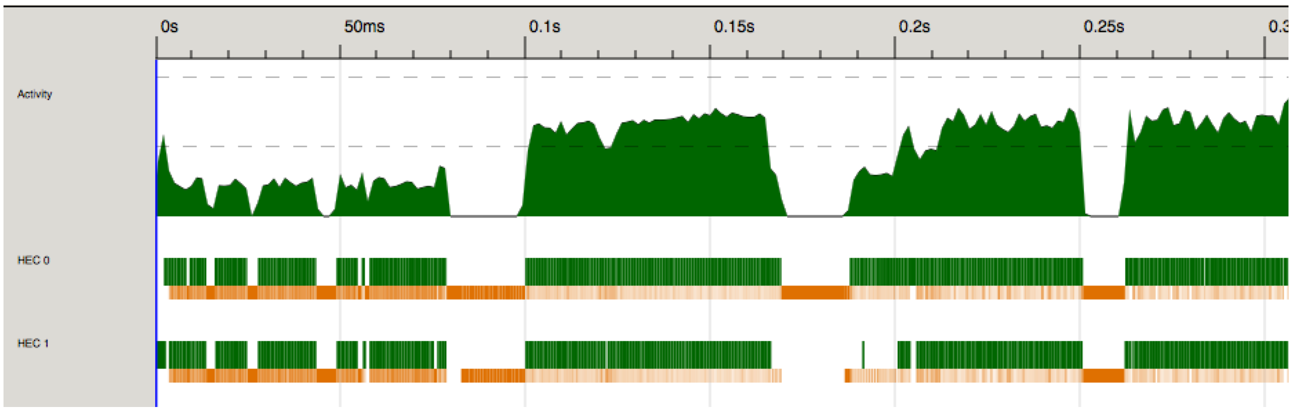
Part 2

For part 2, I began by converting my MapReduce to accept Strategies for the mapper and reducer functions passed as input. The strategy I selected for my mapper function was ``rpar `dot` rdeepseq`` – this means the function will be applied in parallel to each item in the list and will then fully evaluate its result rather than evaluate it in WHNF. For the reducer function I simply passed it `rseq` as its strategy as we don't need to fully evaluate the result immediately.

I then converted my reducer function to perform its fold in parallel which is possible due to Data.Map being a monoid. To achieve this I created a function called `pfold` which will recursively call its self in parallel and split the inputted array in half to be handled on different threads.

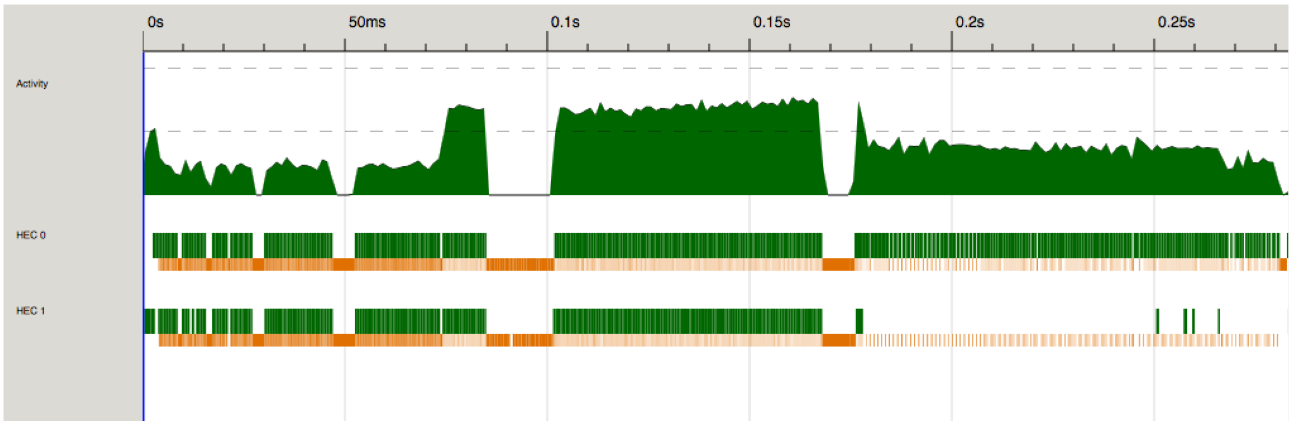
Part 3

For part 3, I began by simply benchmarking my code from part 3 using Threadscope which the results for are show below:



Looking at this graph it's clear that a decent amount of work is being done across both threads that have been spawned, with a total running time of 0.31 seconds. The mapper function finishes around 0.17s which means that roughly equal times are spent on it and the reducer function. A large part of the reason the reducer doesn't take longer is because I'm Data.Map which I found in initial benchmarkings to be significantly faster than using a List of String Int Pairs to perform the union of the list of frequency tables. This is because its implementation is based on size balanced binary trees, which means it can perform a lookup in O(log n) time rather than O(n) time as is the case with lists.

I was then able to improve the speed of my program again by reverting to a non-parallel fold in my reducer function. I believe this is because the overhead for creating new threads made it much slower to perform unions on small maps, as the fold descend recursively through the list of Maps. It's possible that implementing some kind of queue instead of recursively dividing the list in half inside the parallel fold could spread work more evenly and reduce some of the slowness, but I decided to revert to a non-parallel fold for the sake of simplicity. The new graph is pictured below with the running time now coming at around 0.28s:



Part 4

Finally, for Part 4 I started by simply implementing the MapReduce monad as described in Julian Porter’s paper and his example word frequency counter. I decided to replace his parallel map function `pmap` with a concurrent version called `concurrentMap` making use of `forkIO` and channels. However, because `forkIO` needs to be wrapped in an IO monad, this forces us to use `unsafePerformIO`.

The concurrent map function starts by creating two channels – one for inputting values into worker threads and one for outputting values. It then spawns a defined number threads using `forkIO` which will infinitely read a value in from the input channel, map it and then send it to the output channel. Once the threads have been spawned I end each element in the list to a worker thread `forM_`. Finally, I collect the mapped values from the threads by calling `readChan` repeatedly with using `replicateM` and return them.

Using this function I was also able to replace his mapper function with a concurrent version.