PFP Project Report: Autocomplete

Michael Jan (mj2886), Maÿlis Whetsel (mw3391)

Fall 2021

1 Introduction

Our task is as follows: Given an unfinished word in context, predict the remainder of the word. The best prediction will depend on the given characters of the current word as well as the previous words. For example, given the input "I love functional prog", our program's output might be "programming".

Because autocomplete is typically useful only in real time, making the algorithm run fast is of utmost importance, which is why we decided to parallelize it.

Our program is split into two parts (essentially two different programs): BuildKnowledgeBase and AutoCorrect. The former takes a raw English corpus and turns it into n-grams. The latter uses the n-grams to perform autocomplete.

2 BuildKnowledgeBase

BuildKnowledgeBase (BKB) takes a corpus (large file of English text) and converts it into n-gram frequencies. These frequencies are then used by AutoComplete to automatically complete sentences.

In particular, the BKB program does the following:

1. Read the corpus into memory
2. Break it into sentences
3. Preprocess (make lowercase, keep only alphanumeric characters, etc.) and tokenize the sentences
4. Generate n-grams for $n = 1, 2, 3$.
5. Count the frequency of each n-gram
6. Write the n-gram frequencies to text files. 1-gram frequencies (single word frequencies) are stored in 1.txt, 2-gram frequencies are stored in 2.txt, and so on.

2.1 Sequential

Below is the sequential implementation, and Figure 1 shows its Threadscope output. We tested the program on a corpus of 40000 words (5 megabytes) on a 2018 Macbook Pro with an i7 6-core processor and 16 GB of RAM. From the Threadscope output, we found that the execution took 13.78 seconds and used only one core (which is expected since this is the sequential implementation).

```
isStringNumber :: String -> Bool
isStringNumber = foldr ((&&) . isNumber) False

getFreqs :: (Ord a) => [a] -> [(a, Int)]
getFreqs xs = M.toList (M.fromListWith (+) [(x, 1) | x <- xs])

processSentence :: String -> [String]
```

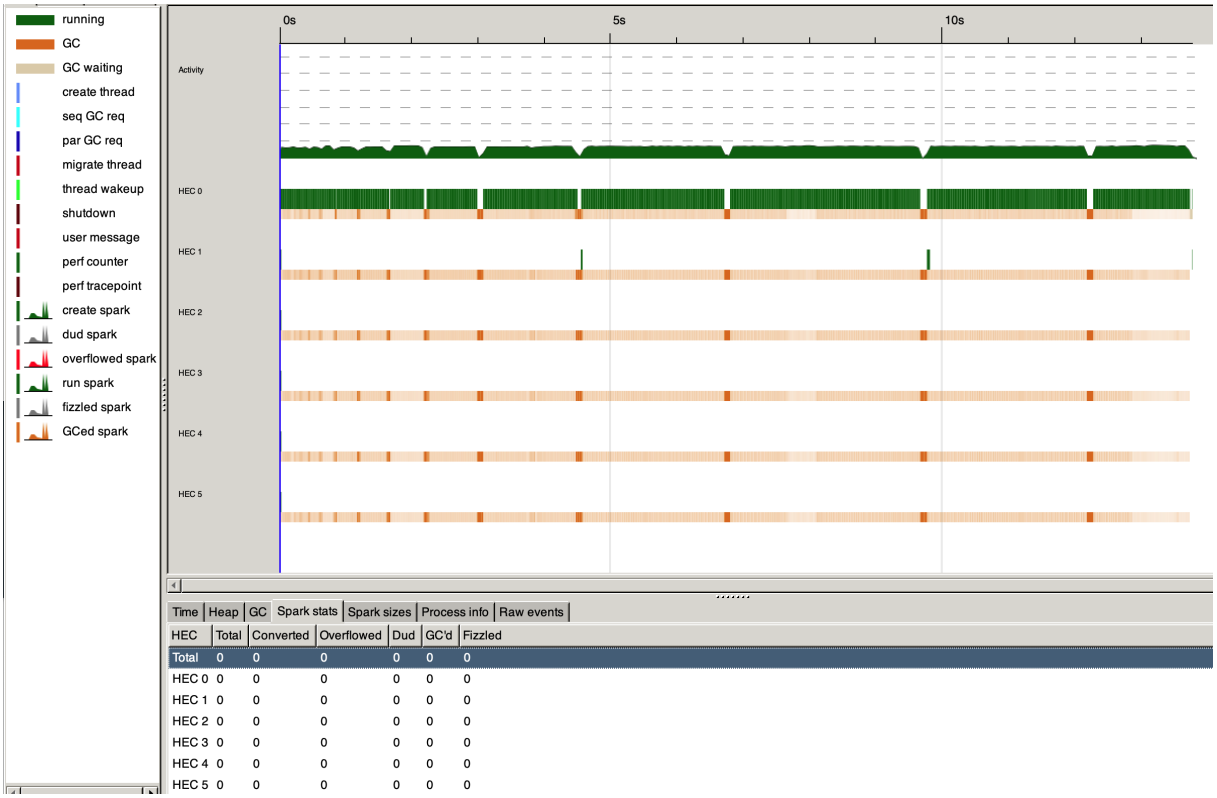

Figure 1: Threadscope output for sequential implementation.

```
processSentence sentence = toks'''
  where
    -- split into words
    toks = splitOneOf " .,;:!?()[]{}<>-/\"\\\n" sentence
    -- make lowercase and keep only letters and numbers
    toks' = map (map toLower . filter (\c -> isLetter c || isNumber c)) toks
    -- convert each number to "[NUM]"
    toks'' = map (\t -> if isStringNumber t then "[NUM]" else t) toks'
    -- remove empty strings
    toks''' = filter (not . null) toks''

getNGrams :: Int -> [String] -> [[String]]
getNGrams n toks = filter (\ts -> length ts == n) \$ map (take n) . tails $ toks

getNGramFreqs:: Int -> [[String]] -> [([String], Int)]
getNGramFreqs n sentences = getFreqs \$ concatMap (getNGrams n) sentences

writeNGramFreqs :: [([String], Int)] -> String -> IO ()
writeNGramFreqs freqs filename = writeFile filename \$ unlines lines
  where
    lines = map (\(ts, c) -> show c ++ " " ++ unwords ts) freqs

buildKnowledgeBase :: String -> String -> IO ()
buildKnowledgeBase inputFile outputDir = do
  input <- readFile inputFile
```

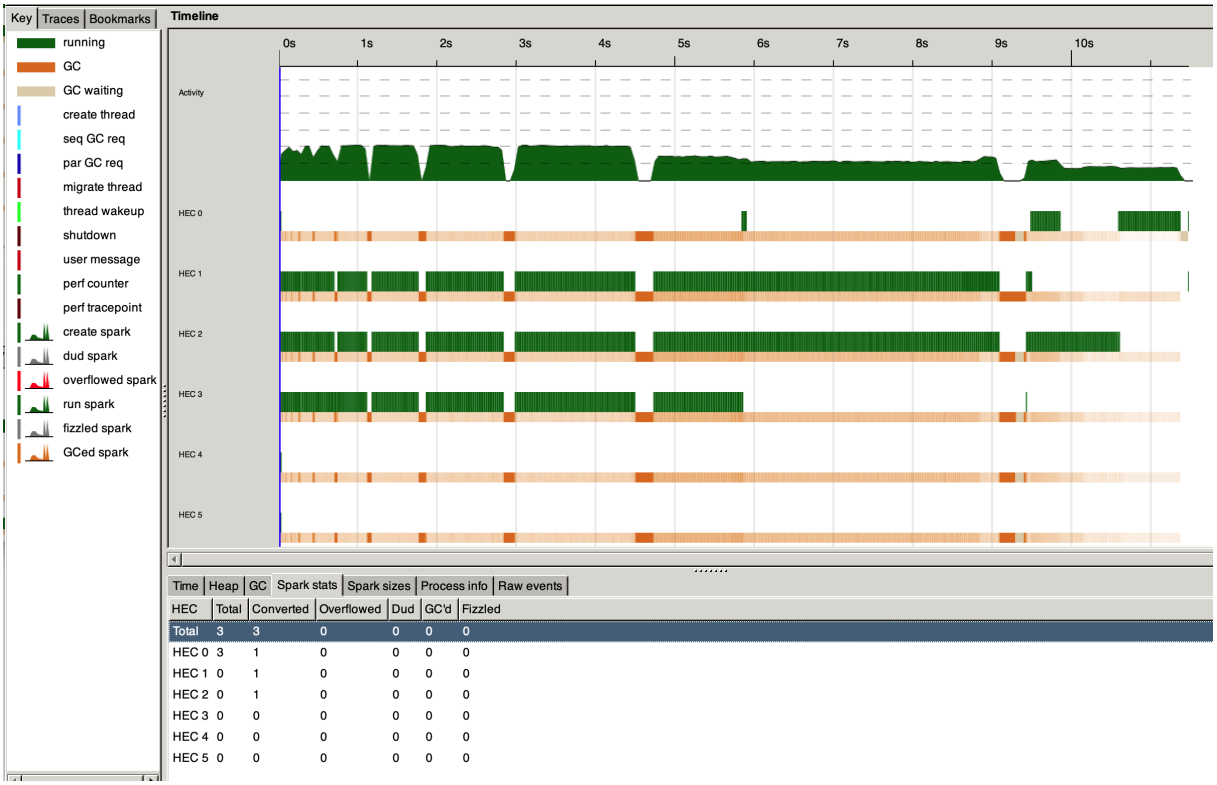

Figure 2: Threadscope output for BKB parallel strategy 1.

```
let sentences = splitOneOf ".?!" input
    sentences' = map processSentence sentences

    sentences'' = map (take 200) \$ chunksOf 200 sentences'
    ngram_freqs n = concat (
      map (getNGramFreqs n) sentences'' `using` parList rdeepseq)
    freqs = map ngram_freqs [1..3]

mapM_ (\(n, f) -> writeNGramFreqs f (outputDir ++ "/" ++ show n ++ ".txt")) \$
 zip [1..] freqs
```

2.2 Parallel Strategy 1

Our first attempt at parallelization was to generate 1-grams, 2-grams, and 3-grams concurrently. To accomplish this, we changed the following line of our main BKB function following to:

```
freqs = map (`getNGramFreqs` sentences') [1..3] `using` parList rdeepseq
```

We achieved a 1.2x performance boost, which is not much. Furthermore, the Threadscope output in Figure **??** indicates that a maximum of three cores were used at a time. This was expected because our intent was to have one core generate 1-grams, another core generate 2-grams, and yet another core generate 3-grams.

2.3 Parallel Strategy 2

We then tried to preprocess each sentence correctly, which gave a 1.7x speedup. To this end, we modified another line of code:

```
sentences' = map processSentence sentences `using` parListChunk 16 rdeepseq
```

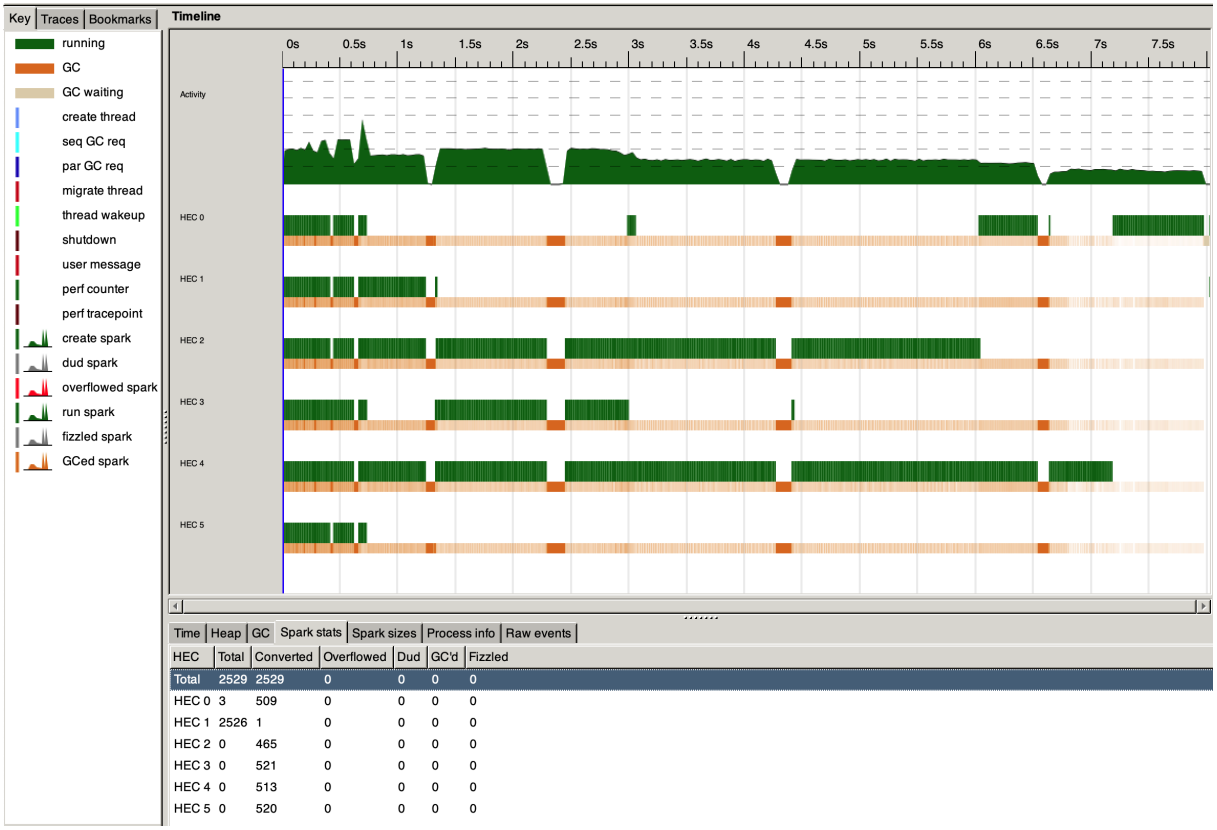

Figure 3: Threadscope output for BKB parallel strategy 2.

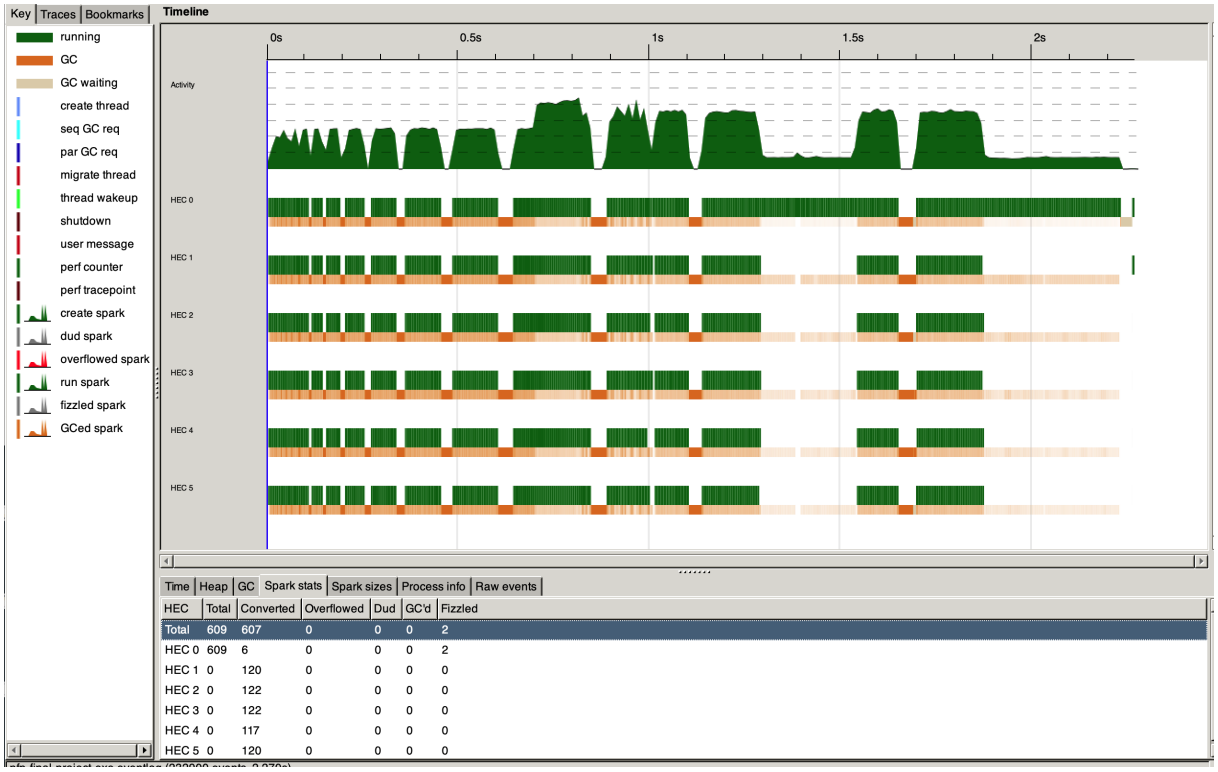

Figure 4: Threadscope output for BKB parallel strategy 3.

The Threadscope output (given in Figure 3) differs from the previous version in that for the first half-second, all cores are being used. This indicates that the sentence preprocessing took only a small fraction of the total running time, which we suspect is why this parallelization attempt did not speed our program up by much. Note also that we used `parListChunks` instead of `parList` because processing a single sentence takes so little time (and there are so many sentences) that it would be unreasonable to generate one spark per sentence.

2.4 Parallel Strategy 3

Our final attempt at parallelizing BKB was much more successful. From Strategy 2, we found that the sentence preprocessing was not the bottleneck—so the bottleneck had to be the n-gram generation. Strategy 1 parallelized this portion of our program but did so naively so that only a maximum of 3 cores were used at a time.

This time, we broke the sentences into batches and generated n-grams from them in parallel. As you can see from the Threadscope output in Figure 4, this led to all cores being used and a speedup of 5.7x, which is near the theoretical limit.

To accomplish this, we modified the main BKB function as follows:

```
sentences' = map processSentence sentences
sentences'' = map (take 200) \$ chunksOf 200 sentences'
ngram_freqs n = concat (
  map (getNGramFreqs n) sentences'' `using` parList rdeepseq)
freqs = map ngram_freqs [1..3]
```

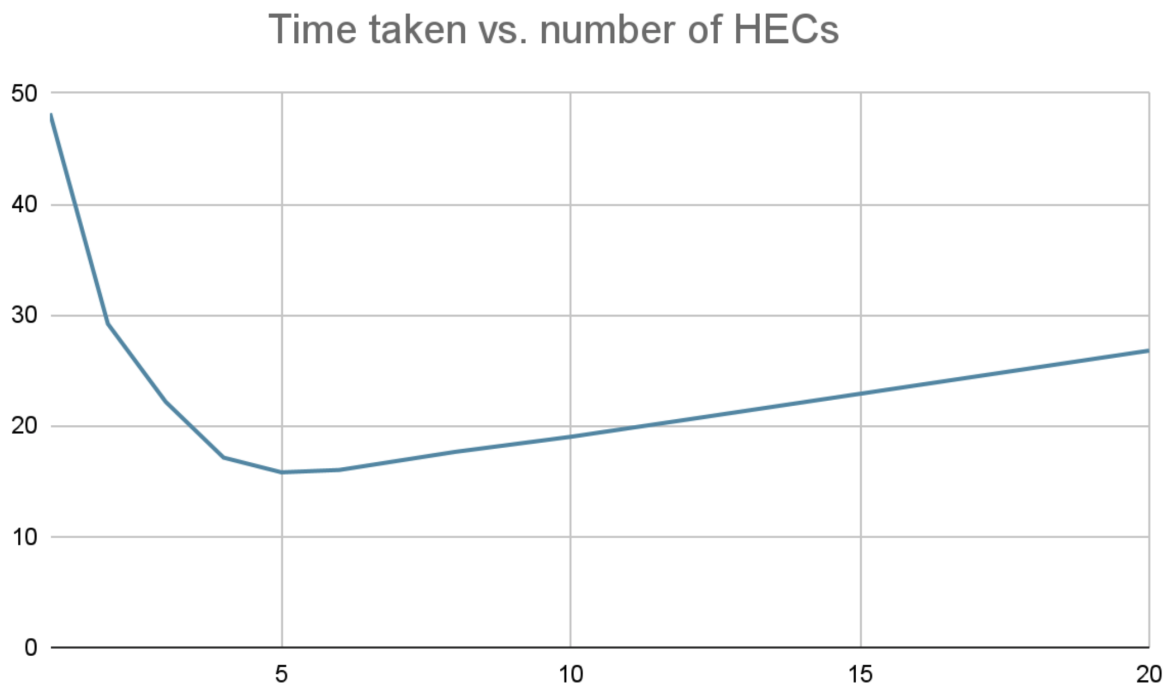

Figure 5: Test results on a corpus of 250k words.

2.5 Further testing

For completeness, we tested our final parallelized implementation on a larger corpus (250k words, 35 mb). On the same (6-core) machine, we varied the number of threads from 1 to 20, and the running time is graphed in Figure 5. As shown, the best speedup was 3x, using 5 threads. As we continued adding threads (to the point where there were many more threads than available physical cores), performance decreased, most likely due to the added overhead of context switching.

3 AutoComplete

AutoComplete (AC) takes a folder containing N-gram files (generated by BKB) and an input.txt file containing the unfinished word (in-context). It will the output the most likely word to finish the sentence based on word N-gram frequency.

In particular the AC program does the following:

1. Read the 1-grams (word frequency) into a Trie
2. Read the 2 and 3-grams into a map (gram to frequency)
3. Extract the prefix (word to complete), the second to last word (start of the 2-gram), and two second to last words (start of the 3-gram)
4. Run a lookup on the Trie to get all valid words completing the prefix
5. Generate 2-grams and 3-grams with each of the valid completing words and get the associated score by looking them up in their respective map
6. Combine 1,2,3-grams to get (word, score) pairs – we ended up only using the 2 and 3-grams scores since otherwise nonsensical but super common words overpower all other possibilities

7. Traverse the scores list and print the word with the maximum score

3.1 Sequential

Below is the sequential implementation, and Figure 6 shows its Threadscope output. We tested the program on a corpus of 5264944 words (31 megabytes) on a 2020 Macbook Pro with an i5 4-core processor and 16 GB of RAM. From the Threadscope output, we found that the execution took 12.641 seconds and used only one core (which is expected since this is the sequential implementation).

```
readNGramFreqs :: B.ByteString -> M.Map [B.ByteString] Int
readNGramFreqs contents = M.fromList $ map getListTuple (B.lines contents)
  where
    getListTuple x = case B.words x of
      [freq, a, b, c] -> ([a, b, c], fst $ fromJust $ B.readInt freq)
      [freq, a, b] -> ([a, b], fst $ fromJust $ B.readInt freq)
      _ -> ([], 0)

buildTrie :: B.ByteString -> T.Trie Int
buildTrie contents = T.fromList $ map getTuple (B.lines contents)
  where
    getTuple s = case B.words s of
      [freq, a] -> (a, fst $ fromJust $ B.readInt freq)
      _ -> (B.empty, 0)

autoComplete :: String -> String -> IO ()
autoComplete knowledgeBaseDir inputFile = do
  ifile <- readFile inputFile
  let input = processSentence ifile

  let trie = buildTrie f1 -- f1 is 1-gram file contents as ByteString
  let words = T.toList $ T.submap prefix trie -- prefix is last "word" in contents

  let m2 = readNGramFreqs f2 -- f2 is 2-gram file contents as ByteString
  let m3 = readNGramFreqs f3 -- f3 is 3-gram file contents as ByteString

  let scores = zipWith3 comb words (calcF m2 g2 words) (calcF m3 g3 words)
  print $ fst $ maximumBy (comparing snd) scores

  where
    makeGrams words g = map (makeNGrams g) words
    calcF m g w = map (getFreq m) (makeGrams w g)
```

Note: For brevity some code was omitted

- fname - gets n-gram file path
- processSentence - cleans up text
- prefix - the word to autocomplete
- comb - combines scores for three tuples
- getFreq - does map lookup to get tuple
- makeNGram - appends word to generic N-gram

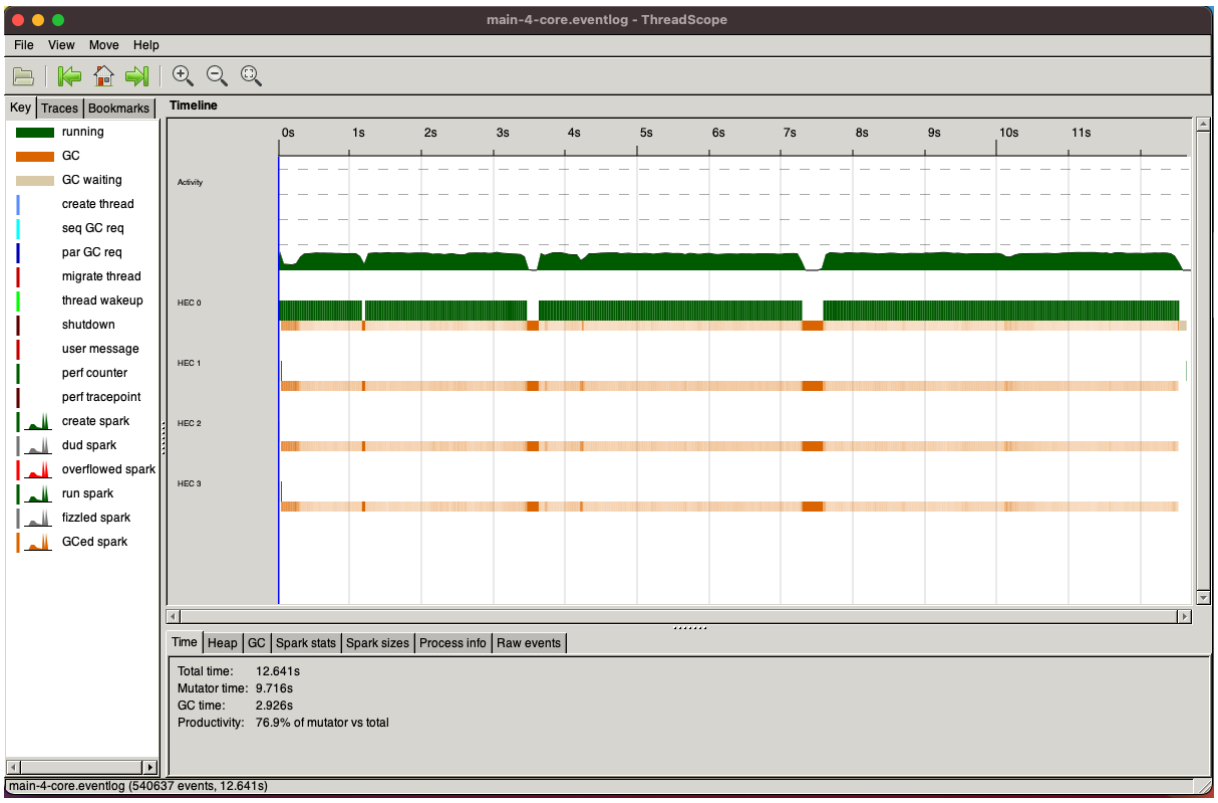

Figure 6: Threadscope output for AC sequential implementation.

3.2 Naive Parallel: `parList` on `calcF`

We started by played around with printing and found that the `zipWith3` statement was the most time consuming part of the program. As such it made sense to focus on parallelizing the `calcF` function since it does the majority of the work calculating the scores for each of the resulting words.

We started using `parList` to create a spark for each part of the list and then evaluate them to normal form using the `rdeepseq` strategy.

```
makeGrams words g = map (makeNGrams g) words `using` parList rdeepseq
calcF m g w = map (getFreq m) (makeGrams w g) `using` parList rdeepseq
```

This resulted in some speedup (around x1.4 for 2 cores) but the Threadscope output (Figure 7) showed that the strategy was not as parallel as expected. In particular we were not able to get the program to use all of the cores, instead it ran the two `calcF` calculations (2-grams and 3-grams) in parallel and then finished the rest of the program sequentially.

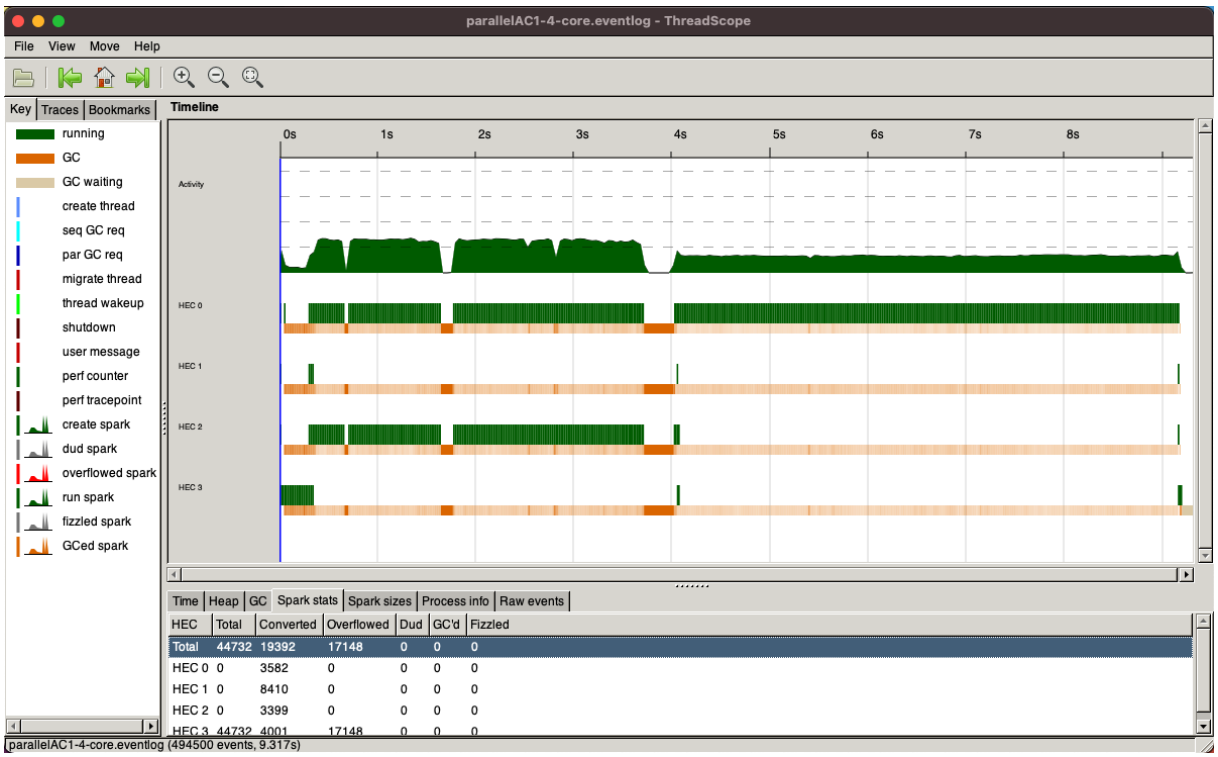

Figure 7: Threadscope output for AC parallel strategy 1.

Cores	Time
1 core	12.016s
2 cores	8.810s
3 cores	9.445s
4 cores	9.371s

The timing on the different cores also corroborated the Threadscope results. Two cores was the fastest since the `parList` strategy only resulted in the two `calcF` calls being done in parallel and as a result it would use two cores at most.

3.3 Parallel 2: `parList` everywhere

After running the code from our first parallel strategy we realized that much of the work that appeared to have been done before was actually being done lazily and it was not until the `parList` call for `calcF` that the setup code (reading in the files, creating the trie, etc...) was actually being done. So we added the same `parList rdeepseq` strategy to the `readNGramFreqs` and `buildTrie` functions as well to prompt the program to begin those calculations earlier.

```
readNGramFreqs :: B.ByteString -> M.Map [B.ByteString] Int
readNGramFreqs contents = M.fromList (map getListTuple (B.lines contents) `using` parList rdeepseq)
  where
    getListTuple x = case B.words x of
      [freq, a, b, c] -> ([a, b, c], fst $ fromJust $ B.readInt freq)
      [freq, a, b] -> ([a, b], fst $ fromJust $ B.readInt freq)
      _ -> ([], 0)

buildTrie :: B.ByteString -> T.Trie Int
buildTrie contents = T.fromList (map getTuple (B.lines contents) `using` parList rdeepseq)
  where
```

```
    getTuple s = case B.words s of
      [freq, a] -> (a, fst $ fromJust $ B.readInt freq)
      _ -> (B.empty, 0)

autoComplete :: String -> String -> IO ()
autoComplete knowledgeBaseDir inputFile = do
  ifile <- readFile inputFile
  let input = map B.pack (processSentence ifile)

  let trie = buildTrie f1 -- f1 is 1-gram file contents as ByteString
  let words = T.toList $ T.submap prefix trie -- prefix is last "word" in contents

  let m2 = readNGramFreqs f2 -- f2 is 2-gram file contents as ByteString
  let m3 = readNGramFreqs f3 -- f3 is 3-gram file contents as ByteString

  let scores = zipWith3 comb words (calcF m2 g2 words) (calcF m3 g3 words)
  print $ fst $ maximumBy (comparing snd) scores

  where
    makeGrams words g = map (makeNGrams g) words `using` parList rdeepseq
    calcF m g w = map (getFreq m) (makeGrams w g) `using` parList rdeepseq
```

As you can see from the Threadscope output in Figure 8 there is now a lot more work being done upfront to process the files. However the sparks are out of control with (roughly) 5 million being generated and only 1/2 a million of them being converted. As a result this strategy is even slower than before despite being able to utilise all four cores.

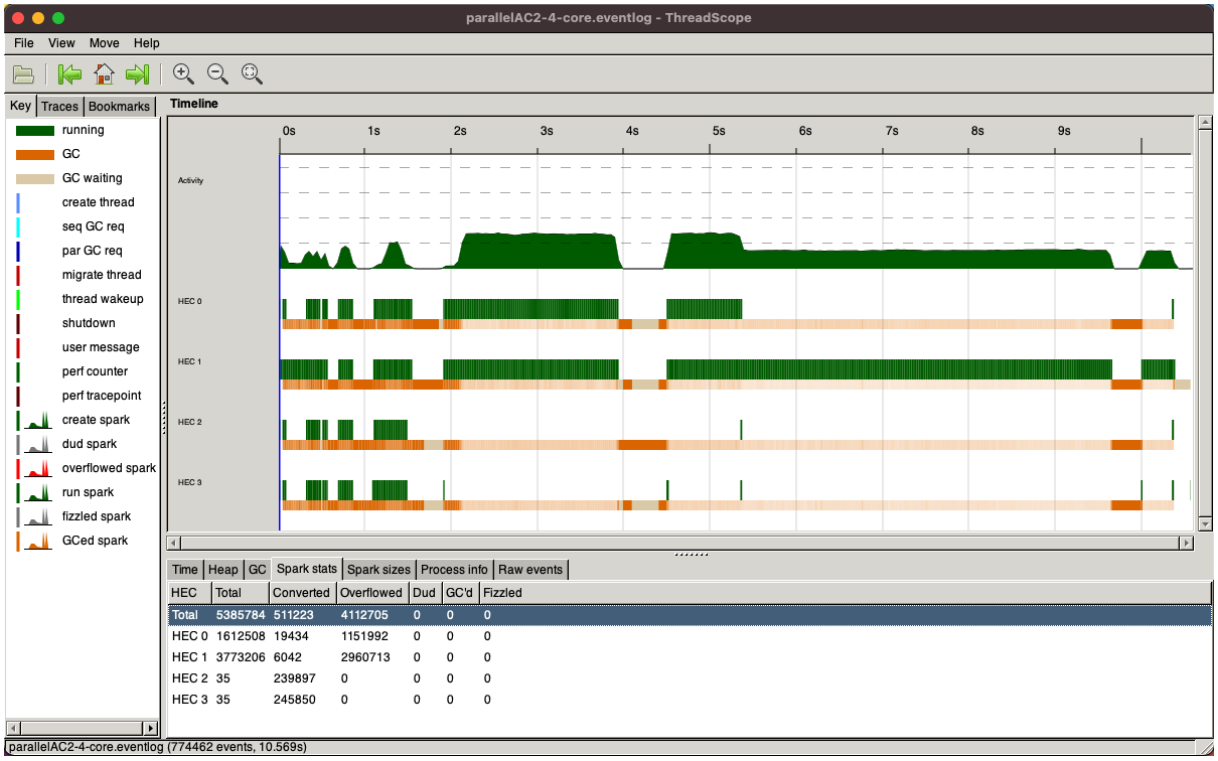

Figure 8: Threadscope output for AC parallel strategy 2.

Cores	Time
1 core	13.438s
2 cores	14.152s
3 cores	11.044s
4 cores	10.569s

3.4 Parallel 3: `parListChunk`

Since the last attempt at speeding up the program resulted in huge amounts of spark overflow we decided to try and regulate the number of sparks being created to avoid having such high overhead. Using `parListChunk` and chunk sizes of 100 we were able to significantly reduce the overflow and sparks being created. This resulted in a much better rate of overflow and garbage collection but 3/5ths of the sparks were still not being converted. Furthermore, the two parallel runs of `calcF` were no longer running on two separate cores.

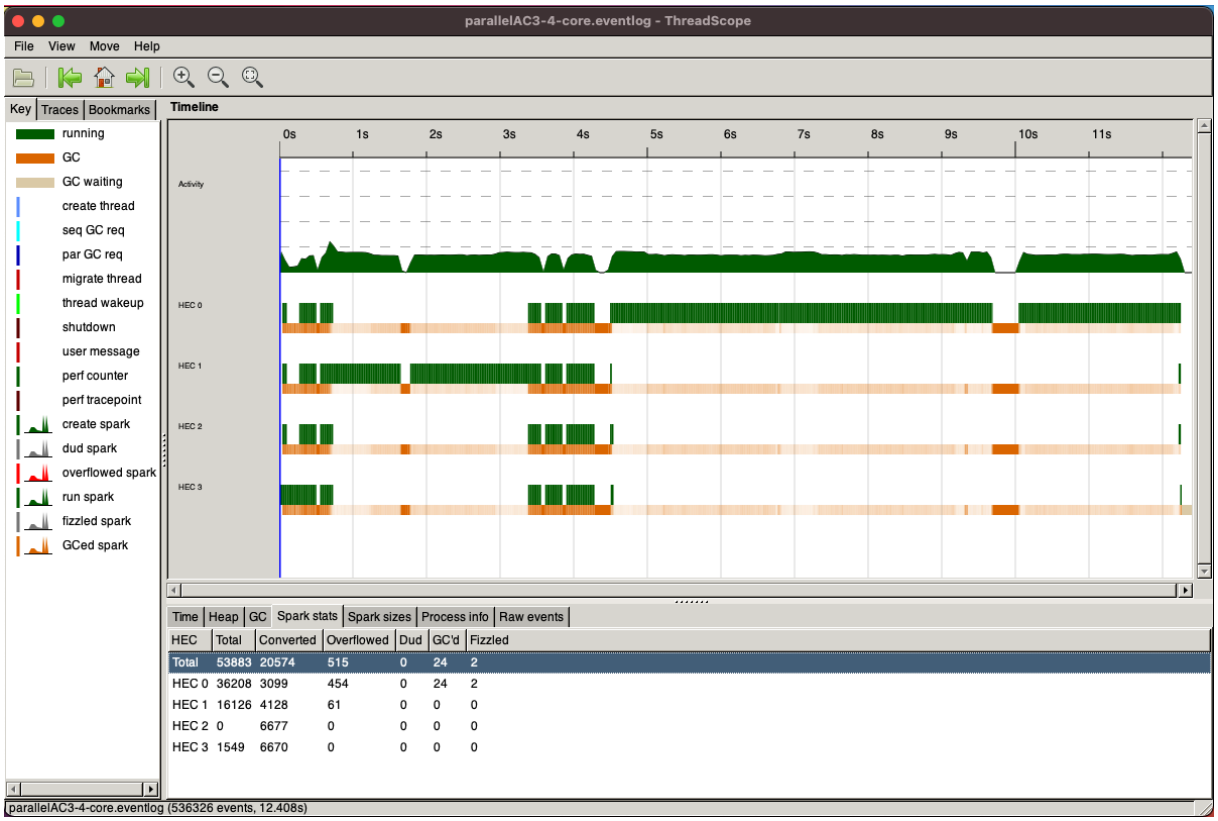

Figure 9: Threadscope output for AC parallel strategy 3.

Cores	Time
1 core	13.191s
2 cores	12.707s
3 cores	12.405s
4 cores	12.408s

3.5 Parallel 4: `parBuffer` and two chunk sizes

Our final parallel strategy involved using `parBuffer` and two chunk sizes. Using `parBuffer` was much more space efficient than `parList` since it is lazy and waits for sparks to finish before computing more of the list. This meant that the sparks themselves were much more efficient. Additionally since the IO work (`readNGramFreqs` and `buildTrie`) were operating on huge files we tailored the chunk sizes to be more

appropriate to their work the IO chunks sizes were 5000 while the score chunk sizes were 2000. With this combination we were able to get nice parallelization across all the cores with the speed peaking at 8.275 seconds on 3 cores which was a x1.53 speedup from pure sequential and a x1.66 speedup from running the same program on just one core.

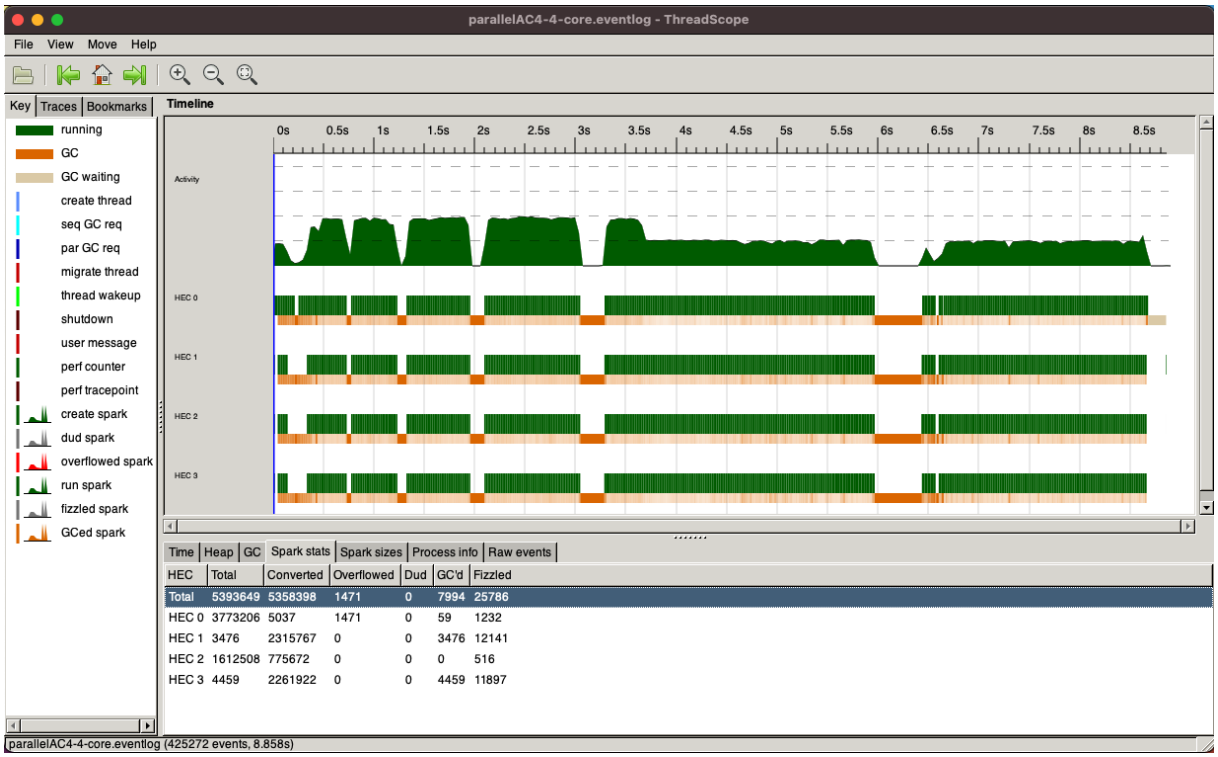

Figure 10: Threadscope output for AC parallel strategy 4.

Cores	Time
1 core	13.783s
2 cores	10.522
3 cores	8.275s
4 cores	8.858s