Passage Search by Similarity

Samhit Chowdary Bhogavalli (sb4845), Mooizz Abdul (ma4496).

December 19, 2024

1 Background & Objective

In this project, we aim to tackle the challenge of parallelizing the similarity search in the context of passages using embeddings. The problem is as follows given a text query, the program should retrieve best passage that matches the query using Embeddings, ie, a representation of text we can solve this problem in two stages.

- Creation of text embeddings
- Query Search

1.1 Text Embedding Creation

Text embeddings are numerical representations of text that capture the semantic meaning of words, phrases, or documents in a continuous vector space. There are various methods to generate text embeddings, ranging from statistical approaches to sophisticated machine learning techniques.

1.1.1 Statistical Methods

- TF-IDF (Term Frequency-Inverse Document Frequency): This approach represents text as sparse vectors based on the frequency of words in a document relative to their occurrence across a corpus.

1.1.2 Neural Network-Based Embeddings

- Word2Vec
- Glove
- Bert Embeddings

In this stage, we aim to parallelize tf-idf embedding creation. To create TF-IDF embeddings

- Term Frequency (TF): The Term Frequency (TF) measures how often a word appears in a document relative to the total number of words in the document. It is calculated using the formula:

$$\text{TF}(t, d) = \frac{\text{Number of times } t \text{ appears in } d}{\text{Total number of words in } d}$$

  where t is a term in the document and d is the document.

- Inverse Document Frequency (IDF): IDF measures how important a word is by reducing the weight of commonly occurring words across the corpus and emphasizing rare words. It is calculated using the formula:

$$\text{IDF}(t, D) = \log\left(\frac{|D|}{\text{Number of documents where } t \text{ appears}}\right)$$

  where t is the term in corpus and $|D|$ is the number of documents in the corpus.

- TF-IDF Calculation: The TF-IDF score for a term t in a document d is the product of its TF and IDF:

$$\text{TF-IDF}(t, d, D) = \text{TF}(t, d) \times \text{IDF}(t, D)$$

1.2 Embedding Search

In this part of the project, we aim to perform parallel embedding search on this embedding data [link].

We search for the closest passage embedding for a given query embedding, this closeness is measured by a similarity measures. These measures can be Manhattan Distance, Cosine Similarity or Euclidean Distance. We consider cosine similarity as our closeness measure.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Figure 1: Cosine Similarity Formula

Also, we perform absolute search instead of approximate, many leading embeddings searches are approximate in nature like faiss embedding search library.

2 Sequential Implementations

2.1 Embedding Creation Stage

Input for this stage is multiple files in txt format each containing approximately 10 passages mapped to their passage id. The goal is to get an embedding vector for each passage and store the final result in a CSV file.

We used (Map String String) in Haskell to store the passage id and passage mapping.

- Step 1: Reading the data from all the files and parsing each line and storing the results in a map. To parse each line of the file, it is split based on the first "," character. The string before the comma is treated as the ID, and the string after it represents the passage.

```
parseLine :: String -> (String, String)
parseLine line =
    let parts = splitOn "," line
    in case parts of
        (key:rest) -> (key, unwords rest)
        _ -> error $ "Invalid line format: " ++ line
```

- Step 2: Calculate the IDF Values:

  Building WordMap: This step involves creating a map to store the final IDF values of words in the passages. To achieve this, we read a dictionary file containing the relevant words and initialize a map with these words as keys and 0 as their initial values. Here we are building WordMap from external dictionary because our dataset has approx 10000 words and using these many words to calculate TF-IDF vectors will be too expensive. So we limited the words in the dictionary to 500.

  To compute the IDF values, all passages are first converted into sets of unique words. These sets are then used to update the word map by incrementing the values for words that appear in each passage. Finally, the values in the map are normalized using the total passage count to calculate the IDF values.

```
idf passages initMap =
    let passageSets = map sentenceToSet passages
        idf_count = foldr addSetToMap initMap passageSets
    in idf_count

sentenceToSet (_, sentence) =
    Set.fromList $ map (map toLower) (words sentence)

addSetToMap passage_set doc_count =
```

```
    foldr (\word acc' -> Map.adjust (1.0 +) word acc')
        doc_count (Set.elems passage_set)

idfNormalise x totalDocCount = logBase 2 ((fromIntegral totalDocCount) / x)
```

- Step 3: Calculate TF values and combine it with the IDF values:

  To calculate TF values for each passage, I used list comprehension to add each word to a list and aggregated the results into a map. This process was repeated for all passages. Afterward, the values were normalized and multiplied by their respective IDF values to produce the final TF-IDF scores.

```
tfIdf :: [String] -> [String] -> Map.Map String Double -> Map.Map String Double
tfIdf wordsList passageWords idf =
  let totalWords = fromIntegral (length passageWords)
      wordCounts = Map.fromListWith (+) [(word, 1) | word <- passageWords]
  in Map.fromList
      [ ( word,
          let wordCount = fromMaybe 0 (Map.lookup word wordCounts)
              idfValue = fromMaybe 0 (Map.lookup word idf)
           in (wordCount * idfValue / totalWords)
        )
      | word <- wordsList
      ]

tfIdfForAll :: [String] -> [(String, String)] ->
    Map.Map String Double -> Map.Map String String
tfIdfForAll wordsList passages idf =
    let passageMap = Map.fromList passages
    in Map.map (\passage ->
        let passageWords = words (map toLower passage)
            tfValues = tfIdf wordsList passageWords idf
        in mapToCsvRow wordsList tfValues) passageMap
```

  The final output is converted to proper format which can be later consumed in the search phase of the algorithm.

2.2 Embedding Search Stage

The inputs for this stage are query embedding e_q and list of passage embeddings p_i, i ranges from 1 to 200k. The goal of this stage is to find the passage embedding that is closer to the given query embedding.

For representing embedding in haskell, we use the type Vector of double values

```
import qualified Data.Vector as V
type Embedding = Vector Double
```

The closenesss ie similarity is measured by cosine similarity as described below.

```
cosineSimilarity :: Embedding -> Embedding -> Double
cosineSimilarity v1 v2 = let dotProduct = V.sum $ V.zipWith (*) v1 v2
    norm1 = sqrt $ V.sum $ V.map (** 2) v1
    norm2 = sqrt $ V.sum $ V.map (** 2) v2
in dotProduct / (norm1 * norm2)
```

This stage mainly consists of two parts.

- Parsing the embeddings csv into memory

- Finding the Best Passage: Performing Similarity Computation for a given query across all passages

The Parsing part that consists IO runs in the beginning of the program, ie loading the embeddings csv line by line into the memory and then parsing it into embedding ie Vector of double values. Refer Appendix for parsing code.

The Second part,ie finding the best passage is where most of the computation happens.

```
computeSimilarities :: Embedding -> [IdEmbedding] -> [(Int, Double)]
computeSimilarities queryEmbedding passages =
    let compute idEmb = ( fst idEmb , cosineSimilarity queryEmbedding (snd idEmb))
    in map compute passages
```

The above function computes the similarities for each passage and creates a list of Id and Similarity Value pairs

```
findBestPassage :: Embedding -> [IdEmbedding] -> Int
findBestPassage queryEmbedding passages =
    let similarities = computeSimilarities queryEmbedding passages
     in fst $ maximumBy (comparing snd) similarities
```

The above function finds the best passage's Id, given a list of similarities.

3 Parallel Implementations

3.1 Embedding Creation Stage

The main idea is to use the MapReduce framework. we have applied this framework in 2 situations, IDF calculation and TF Calculation

- Approach 1: We created a new spark for each passage and combined the results appropriately, To get the IDF we added the Maps together and for TF we just concatenated the Maps.
- Approach 2: We split our input passages into multiple chunks and created sparks for each chunk and combined the final output appropriately, To get the IDF we added the Maps together and for TF we just concatenated the Maps.

```
parIdf passage_chunks word_map passage_count =
  let par_output_idf =
        map (\input -> idf input word_map) passage_chunks
        `using` parList rdeepseq
      reduced_output =
        Map.unionsWith (+) par_output_idf
  in Map.map
       (\x -> idfNormalise x passage_count)
       reduced_output

parTf passage_chunks wordOrder norm_idf =
  let par_output =
        map
          (\t_p_input -> tfIdfForAll wordOrder t_p_input norm_idf)
          passage_chunks
        `using` parList rdeepseq
  in Map.unions par_output
```

3.2 Embedding Search Stage

To parallelize Embedding search, we tried multiple strategies.

- Parallelizing the computation across embedding dimension, ie when parallelizing cosineSimilarity function

- Parallelizing the computation across list of passages, ie the search load.
  - Basic Strategy
  - Chunk List Strategy

3.2.1 Parallel Cosine Similarity

This parallel implementation didn't really workout, because huge number of sparks were being created. For 200k passages, the scale of sparks was around 10 power 8 or 9, because of the scale the time taken for a parallel implementation resulted in negligible gains in speedup, there were some experiment instances in which the time taken was higher than sequential time.

```
parDotProduct :: [Double] -> [Double] -> Double
parDotProduct xs ys = sum (zipWith (*) xs ys `using` parList rdeepseq)

parMagnitude :: [Double] -> Double
parMagnitude xs =  sqrt (sum  ((map (**2) xs) `using` parList rdeepseq))

cosineSimilarity :: Embedding -> Embedding -> Double
cosineSimilarity vec1 vec2
    | null vec1 || null vec2 = 0.0
    | otherwise =
        let dotProd = parDotProduct vec1 vec2
            mag1 = parMagnitude vec1
            mag2 = parMagnitude vec2
        in dotProd / (mag1 * mag2)
```

3.2.2 Parallel Computation of Passage Embedding Similarities: Basic

To parallelize computation of similarity for each passage in the list, this strategy create a spark for every similarity computation in the passage list. We use parMap to create dynamic sparks and rdeepseq strategy. Refer to the code below

```
computeSimilarities :: Embedding -> [IdEmbedding] -> [(Int, Double)]
computeSimilarities queryEmbedding passages =
let compute idEmb = ( fst idEmb , cosineSimilarity queryEmbedding (snd idEmb))
in parMap rdeepseq compute passages
```

This method has its shortcomings too, most of the sparks were being overflowed. These speed insights suggests us to decrease the number of sparks, we use chunking to do that.

3.2.3 Parallel Computation of Passage Embedding Similarities: Chunk-Based

In this method, we create chunks of passage list and then create a spark for each chunk. Observe that the number of sparks is less and work done in each spark is higher than the previous strategy.

```
-- Chunking
chunkList :: Int -> [a] -> [[a]]
chunkList n = f
  where
    f [] = []
    f list = let (chunk, rest) = splitAt n list in chunk : f rest

-- Compute cosine similarity for a list of passages in parallel
computeSimilarities :: Embedding -> [IdEmbedding] -> [(Int, Double)]
computeSimilarities queryEmbedding passages =
    let compute idEmb = (fst idEmb,
    cosineSimilarity queryEmbedding (snd idEmb)) in map compute passages
```

```
-- Find the best match in a chunk of passages for a given query
findBestInChunk :: Embedding -> [IdEmbedding] -> (Int, Double)
findBestInChunk queryEmbedding passages =
    let similarities = computeSimilarities queryEmbedding passages
     in maximumBy (comparing snd) similarities

findBestPassage :: Embedding -> [IdEmbedding] -> Int
findBestPassage queryEmbedding passages =
    let chunks = chunkList chunkSize passages
        -- local maximum
        bestInChunks = parMap rdeepseq
        (findBestInChunk queryEmbedding) chunks
    -- Global maximum
    in fst $ maximumBy (comparing snd) bestInChunks
```

In the above code, the function chunkList divides the passage list into multiple chunks. The function computeSimilarities maps the similarity score for a passage in the given list. The function findBestInChunk does the job of finding the local maxima ie the closest passage in the given chunk. findBestPassage iterates through these local chunk maximas and outputs the global maximum.

4 Experiments & Results

4.1 How to change the document language and spell check settings

4.2 Embedding Creation Stage

Time taken to create embedding for 200k passages with 1000 chunk size by both sequential algorithm and parallel algorithm with different cores.

Implementation	Time Taken
Sequential	108 sec
Parallel TF-IDF - 2cores	76 sec
Parallel TF-IDF - 4cores	51 sec
Parallel TF-IDF - 8cores	41 sec
Parallel TF-IDF - 12cores	36 sec
Parallel TF-IDF - 16cores	40 sec

From this table you can see that the max speedup we achieve for this setting is 3. More experimental results with different chunk sizes on different cores are plotted in [2].

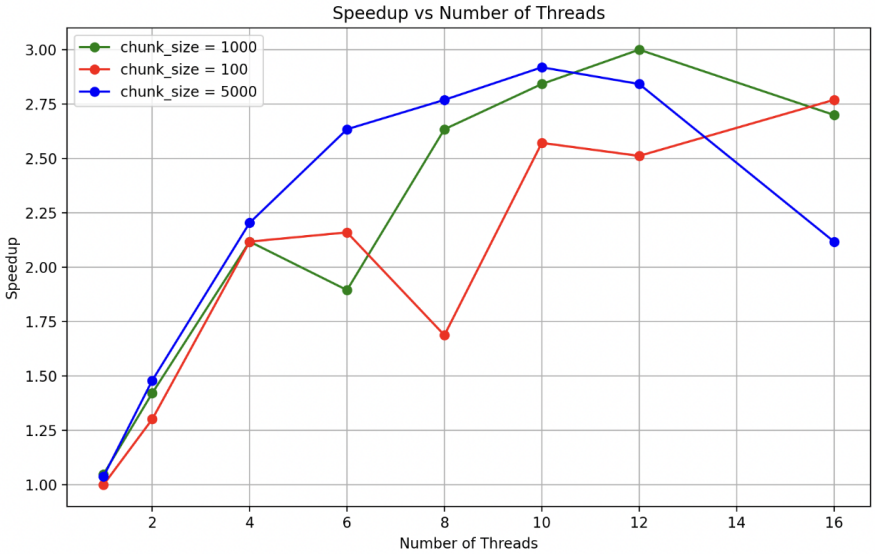

Figure 2: Speedup vs Chunk Size for Different Cores

Eventlogs on threadscope for both sequential and parallel runs can be see in [3] and [4]. From these we can see that in parallel execution 25s of 40s is IO i.e.: reading from file and writing final output to csv. and if we consider similar IO time in sequential execution we can say that just tf-idf vector computation not including IO has > 4 speedup. we can also see there are some pauses because of garbage collection. to resolve this we tried to remove redundant variable consisting of same values and we tried using multiple optimisations but they didn't result in any significant changes in runtime.

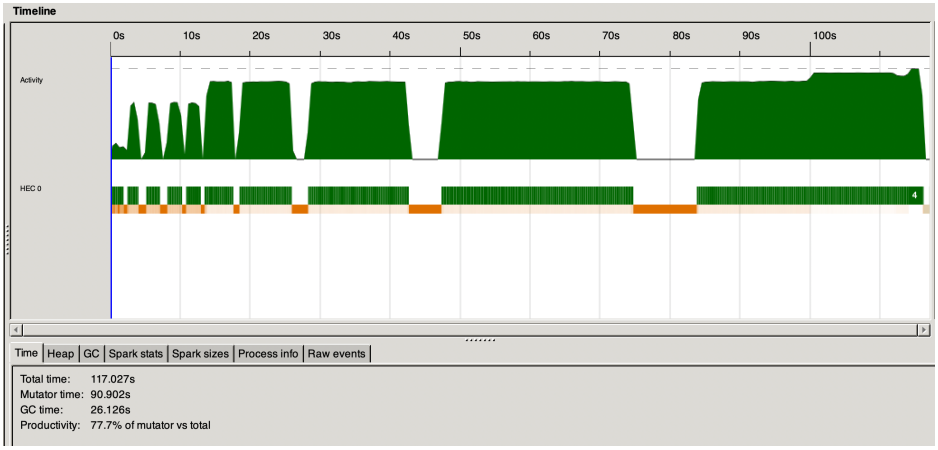

Figure 3: Singlecore eventlog on threadscope for creating TF-IDF vectors

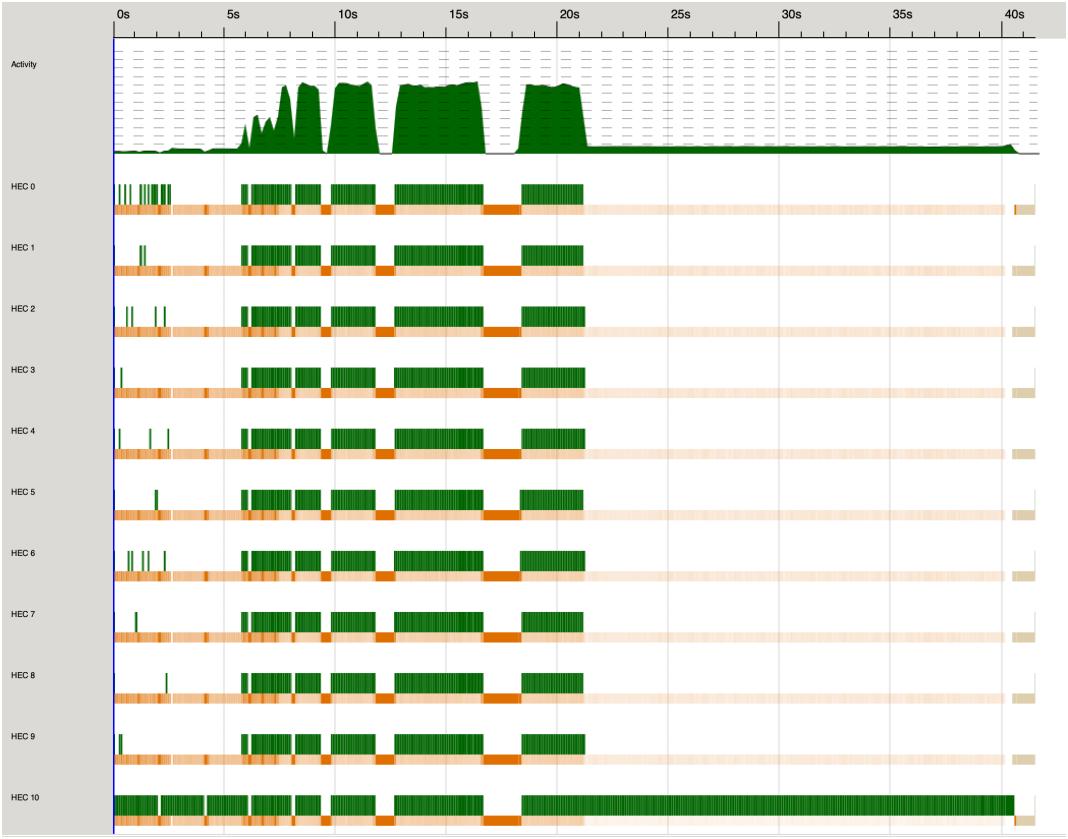

Figure 4: Multicore eventlog on threadscope for creating TF-IDF vectors

4.3 Embedding Search Stage

This section includes experiment results for different parallel strategies

4.3.1 Parallel Cosine Similarity

For 10k passages and 1 query, the time taken was as follows

Implementation	Time Taken
Sequential	14 sec
Parallel CosSim - 2cores	9.5 sec
Parallel CosSim - 4cores	7.4 sec
Parallel CosSim - 8cores	8.5 sec

Even though there was speedup, there are an enormous amount of sparks created even for 10k passages.
Sparks Created: 23 x 10^6, More than 50 percent of them were fizzled for 4 cores.
Consider the case for 200k passages

Implementation	Time Taken	Speed Up
Sequential	300 sec	1.0
Parallel CosSim - 8cores	192 sec	1.5

There were 45 x 10^7 sparks created, but the speedup was 1.5, The computation for each sparks was minimal ie just multiplication or power, this minimal computation for each spark didn't help the case. The overhead of creating sparks far exceeded the gain from parallelism.

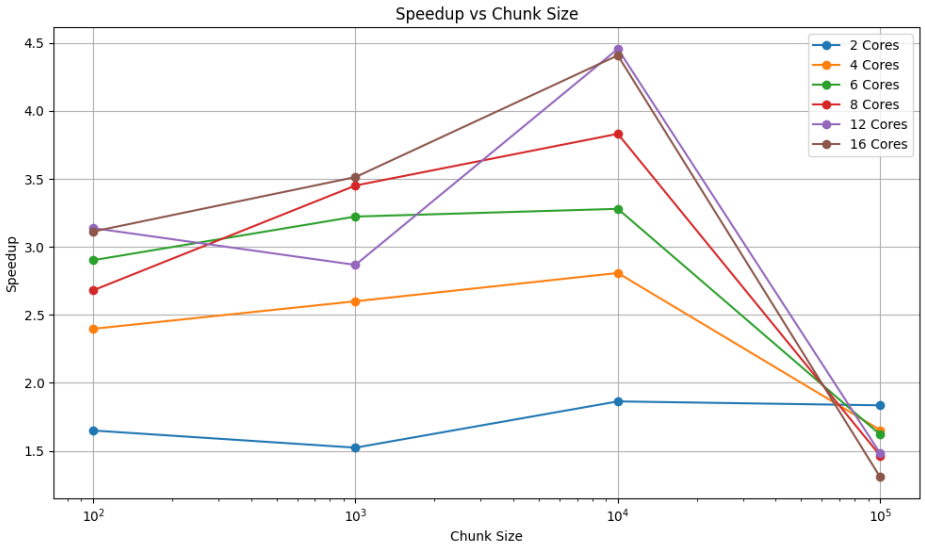

Figure 5: Speedup vs Chunk Size for Different Cores

4.3.2 Parallel Computation of Passage Similarity: Basic

Cores	Sparks Created	Converted	Overflowed	Speed up
2 cores	1995100	74138	1781698	<1
4 cores	1995100	70770	1744106	<1

From the table, we can observe that most of the sparks were overflowed. Also that the speedup is less than 1. The overflow of sparks, the overhead of creating huge number of sparks contributed to this.

4.3.3 Parallel Computation of Passage Similarity: Chunk-Based

In this computation we create a spark for each chunk of passages, Using chunking helped because the speedup was healthy and all the sparks created were converted. We perform experiments on chunk size, query load, number of passages.

Chunk size experiment result [5] indicates that speedup increases as the chunk size increases and at some point it declines.

This decline is due to the fact that as chunk size increases, the number of chunks decreases, this in-turn reduces the number of sparks. Minimal number of sparks doesn't take advantage of the parallel core causing a decline in speedup.

For the optimal chunk size, consider the Speedups for different cores below. For 10 queries and 200k passages.

Implementation	Time Taken	Speed Up
Sequential	410 sec	1.0
Chunk - 2cores	220 sec	1.86
Chunk - 4cores	146 sec	2.8
Chunk - 6cores	125 sec	3.28
Chunk - 8cores	107 sec	3.8
Chunk - 12cores	92 sec	4.45

Below table shows the spark pool data for parallel run with chunks of size 1000. As you can observe all the sparks are converted unlike the basic parallelism approach.

Cores	Sparks Created	Converted	Overflowed	Speed up
2 cores	2000	2000	0	1.58
4 cores	2000	2000	0	2.57

Consider the threadscope output for 10 test queries and 200k passages run on 4 cores in fig [6]. There is IO overhead at the beginning of the program and it impacts the speedup from parallelism.

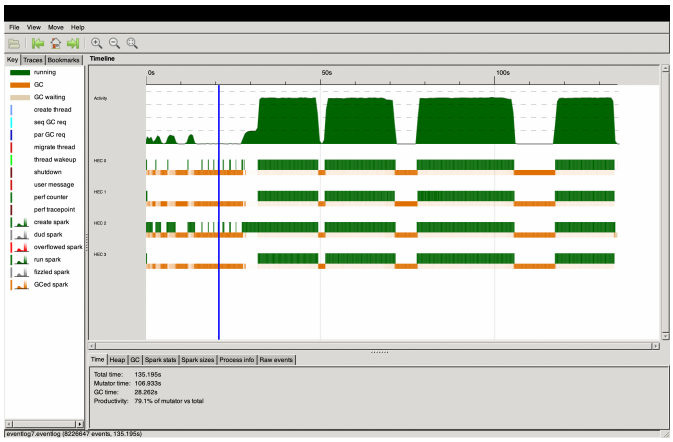

Figure 6: IO Overhead for 10 test queries on 4 cores.

To diminish the IO overhead and measure the speedup gain from parallelism on computation we perform experiments on computation load.

- Number of queries
- Number of passages

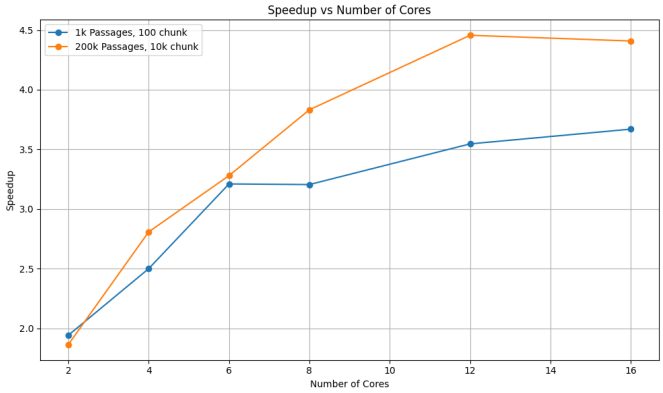

Figure 7: Speedup for 10 queries with different passage loads

As you can see in fig[7], as the number of passages increase, ie the computation load increases, the speedup is higher.

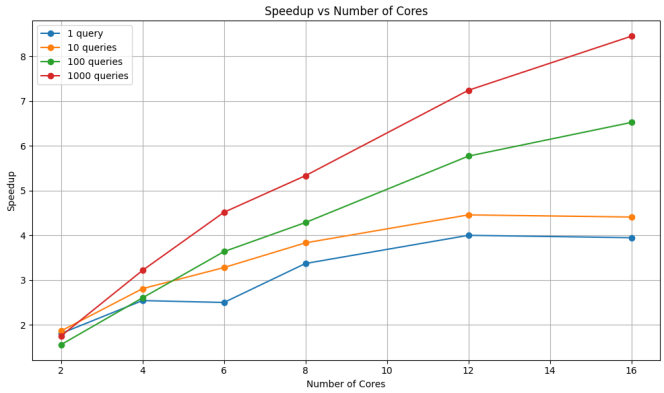

Figure 8: Speedup for 200k passages with different query loads

We also experiment with number of queries, by increasing query load the speedup we get increases as expected. This increase in computation diminishes the IO overhead contribution in speedup calculation. The threadscope eventlog for 1k queries on 200k passages in the figure below indicates this.

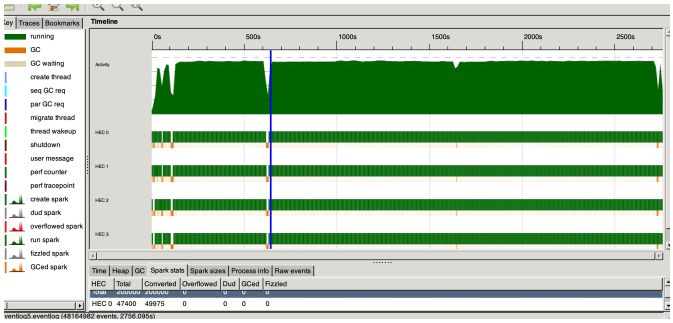

Figure 9: 1k queries, 200k passages

5 Future Optimization Considerations

- Instead of finding the optimal chunk size by experimenting, we could have used number of parallel cores as an indication to configure the chunk size, ie for n number of cores, chunk size could have been configured as 200k/n.
- The Speedup can further be increased by using unboxed vectors during creation of TF-IDF vectors to improve memory efficiency instead of Map.

6 Appendix

[GIT-REPO]

6.1 Embedding Search Stage

Running Tests, The main function which is the entry point for each query search is the findbestPassage method, that is tested by the function below

```
printTuple :: (Int, Int) -> IO ()
printTuple (queryId, bestPassageId) =
putStrLn $ "(" ++ show queryId ++ ", " ++
show bestPassageId ++ ")"

runTests :: [IdEmbedding] -> [IdEmbedding] -> IO ()
runTests testQueryEmbeddings passageEmbeddings = do
    putStrLn "Testing..."
    let results = [ (queryId, findBestPassage queryEmbedding passageEmbeddings)
                  | (queryId, queryEmbedding) <- testQueryEmbeddings ]
    mapM_ printTuple results
    putStrLn "Done."
```

IO at the beginning of the program and the main function, which is common for all the implementations below

```
main :: IO ()
main = do
    args <- getArgs
    progName <- getProgName
    case args of
        [queryEmbeddingsPath, passageEmbeddingsPath] -> do
            putStrLn "Loading test query embeddings..."
            testQueryEmbeddings <- readEmbeddings queryEmbeddingsPath
            putStrLn "Loading passage embeddings..."
            passageEmbeddings <- readEmbeddings passageEmbeddingsPath
            putStrLn "Embeddings loaded. Ready for input."
            runTests testQueryEmbeddings passageEmbeddings
        _ ->
            putStrLn $ "Usage: " ++ progName ++
            "<test_embeddings_file>
            <passage_embeddings_file>"
```

Sequential Version

```
module Main where

import System.IO (hFlush, stdout)
import Data.List (maximumBy, transpose)
import Data.Ord (comparing)
import Data.Vector (Vector)
import qualified Data.Vector as V
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Text.Read as TR
import System.Environment (getArgs, getProgName)

type Embedding = Vector Double
type IdEmbedding = (Int, Embedding)

-- Parsing the CSV file into an (Id, Embedding) tuple
```

```
parseCSVLine :: T.Text -> Either [Char] IdEmbedding
parseCSVLine line = case T.splitOn (T.pack ",") line of
    [idText, embText] -> case TR.decimal idText of
        Right (id, _) -> Right (id, parseEmbeddingText embText)
        Left _ -> Left ("Invalid ID format " ++ show idText)
    _ -> Left ("Invalid line format " ++ show line)

parseEmbeddingText :: T.Text -> Embedding
parseEmbeddingText embText = V.fromList $ map (read . T.unpack)
(T.splitOn (T.pack ":") embText)

readEmbeddings :: FilePath -> IO [IdEmbedding]
readEmbeddings filePath = do
    content <- TIO.readFile filePath
    let parsedLines = map parseCSVLine (T.lines content)
    case sequence parsedLines of
        Right embeddings -> return embeddings
        Left err -> error err

-- Compute the cosine similarity between two embeddings
cosineSimilarity :: Embedding -> Embedding -> Double
cosineSimilarity v1 v2 =
    let dotProduct = V.sum $ V.zipWith (*) v1 v2
        norm1 = sqrt $ V.sum $ V.map (** 2) v1
        norm2 = sqrt $ V.sum $ V.map (** 2) v2
    in dotProduct / (norm1 * norm2)

computeSimilarities :: Embedding -> [IdEmbedding] -> [(Int, Double)]
computeSimilarities queryEmbedding passages =
    let compute idEmb = ( fst idEmb , cosineSimilarity queryEmbedding (snd idEmb))
    in map compute passages

-- Find the best passage for a given query
findBestPassage :: Embedding -> [IdEmbedding] -> Int
findBestPassage queryEmbedding passages =
    let similarities = computeSimilarities queryEmbedding passages
     in fst $ maximumBy (comparing snd) similarities

printTuple :: (Int, Int) -> IO ()
printTuple (queryId, bestPassageId) =
putStrLn $ "(" ++ show queryId ++ ", " ++
show bestPassageId ++ ")"
```

Parallel Cosine Similarity

```
module Main where

import System.IO (hFlush, stdout)
import Data.List (maximumBy, transpose)
import Data.Ord (comparing)
import Data.Vector (Vector)
import Control.Parallel.Strategies (parMap, rdeepseq, parList, withStrategy, using)
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Text.Read as TR
import System.Environment (getArgs, getProgName)
```

```
type Embedding = [Double]
type IdEmbedding = (Int, Embedding)

-- Parsing the CSV file into an (Id, Embedding) tuple
parseCSVLine :: T.Text -> Either [Char] IdEmbedding
parseCSVLine line = case T.splitOn (T.pack ",") line of
    [idText, embText] -> case TR.decimal idText of
        Right (id, _) -> Right (id, parseEmbeddingText embText)
        Left _ -> Left ("Invalid ID format " ++ show idText)
    _ -> Left ("Invalid line format " ++ show line)

parseEmbeddingText :: T.Text -> Embedding
parseEmbeddingText embText = map (read . T.unpack) (T.splitOn (T.pack ":") embText)

readEmbeddings :: FilePath -> IO [IdEmbedding]
readEmbeddings filePath = do
    content <- TIO.readFile filePath
    let parsedLines = map parseCSVLine (T.lines content)
    case sequence parsedLines of
        Right embeddings -> return embeddings
        Left err -> error err

-- Dot product calculation with parallel strategy
parDotProduct :: [Double] -> [Double] -> Double
parDotProduct xs ys =
    sum (zipWith (*) xs ys `using` parList rdeepseq)

-- Magnitude calculation with parallel strategy
parMagnitude :: [Double] -> Double
parMagnitude xs =
    sqrt (sum  ((map (**2) xs) `using` parList rdeepseq))

-- Parallel Cosine Similarity
cosineSimilarity :: Embedding -> Embedding -> Double
cosineSimilarity vec1 vec2
    | null vec1 || null vec2 = 0.0
    | otherwise =
        let dotProd = parDotProduct vec1 vec2
            mag1 = parMagnitude vec1
            mag2 = parMagnitude vec2
        in dotProd / (mag1 * mag2)

computeSimilarities :: Embedding -> [IdEmbedding] -> [(Int, Double)]
computeSimilarities queryEmbedding passages =
    let compute idEmb = ( fst idEmb , cosineSimilarity queryEmbedding (snd idEmb))
    in map compute passages

-- Find the best passage for a given query
findBestPassage :: Embedding -> [IdEmbedding] -> Int
findBestPassage queryEmbedding passages =
    let similarities = computeSimilarities queryEmbedding passages
     in fst $ maximumBy (comparing snd) similarities
```

Basic Parallel Strategy for Passage Similarity Computation

```
module Main where

import Data.List (maximumBy)
```

```
import Data.Ord (comparing)
import Data.Vector (Vector)
import Control.Parallel.Strategies (parMap, rdeepseq, rpar)
import qualified Data.Vector as V
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Text.Read as TR
import System.Environment (getArgs, getProgName)

type Embedding = Vector Double
type IdEmbedding = (Int, Embedding)


-- Parsing the CSV file into an (Id, Embedding) tuple
parseCSVLine :: T.Text -> Either [Char] IdEmbedding
parseCSVLine line = case T.splitOn (T.pack ",") line of
    [idText, embText] -> case TR.decimal idText of
        Right (id, _) -> Right (id, parseEmbeddingText embText)
        Left _ -> Left ("Invalid ID format " ++ show idText)
    _ -> Left ("Invalid line format " ++ show line)

parseEmbeddingText :: T.Text -> Embedding
parseEmbeddingText embText = V.fromList $ map (read . T.unpack) (T.splitOn (T.pack ":") embText)



readEmbeddings :: FilePath -> IO [IdEmbedding]
readEmbeddings filePath = do
    content <- TIO.readFile filePath
    let parsedLines = map parseCSVLine (T.lines content)
    case sequence parsedLines of
        Right embeddings -> return embeddings
        Left err -> error err

-- Compute the cosine similarity between two embeddings
cosineSimilarity :: Embedding -> Embedding -> Double
cosineSimilarity v1 v2 =
    let dotProduct = V.sum $ V.zipWith (*) v1 v2
        norm1 = sqrt $ V.sum $ V.map (** 2) v1
        norm2 = sqrt $ V.sum $ V.map (** 2) v2
    in dotProduct / (norm1 * norm2)

computeSimilarities :: Embedding -> [IdEmbedding] -> [(Int, Double)]
computeSimilarities queryEmbedding passages =
    let compute idEmb = ( fst idEmb , cosineSimilarity queryEmbedding (snd idEmb))
    in parMap rdeepseq compute passages

-- Find the best passage for a given query
findBestPassage :: Embedding -> [IdEmbedding] -> Int
findBestPassage queryEmbedding passages =
    let similarities = computeSimilarities queryEmbedding passages
     in fst $ maximumBy (comparing snd) similarities
```

Chunk-Based Parallel Strategy for Passage Similarity Computation

```
module Main where

import System.IO (hFlush, stdout)
```

```
import Data.List (maximumBy, transpose)
import Data.Ord (comparing)
import Data.Vector (Vector)
import Control.DeepSeq (force)
import Control.Parallel.Strategies (parMap, rdeepseq, rpar)
import qualified Data.Vector as V
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Text.Read as TR
import System.Environment (getArgs, getProgName)

type Embedding = Vector Double
type IdEmbedding = (Int, Embedding)

chunkSize :: Int
chunkSize = 10000

-- Parsing the CSV file into an (Id, Embedding) tuple
parseCSVLine :: T.Text -> Either [Char] IdEmbedding
parseCSVLine line = case T.splitOn (T.pack ",") line of
    [idText, embText] -> case TR.decimal idText of
        Right (id, _) -> Right (id, parseEmbeddingText embText)
        Left _ -> Left ("Invalid ID format " ++ show idText)
    _ -> Left ("Invalid line format " ++ show line)

parseEmbeddingText :: T.Text -> Embedding
parseEmbeddingText embText = V.fromList $ map (read . T.unpack) (T.splitOn (T.pack ":") embText)



readEmbeddings :: FilePath -> IO [IdEmbedding]
readEmbeddings filePath = do
    content <- TIO.readFile filePath
    let parsedLines = map parseCSVLine (T.lines content)
    case sequence parsedLines of
        Right embeddings -> return embeddings
        Left err -> error err

-- Compute the cosine similarity between two embeddings
cosineSimilarity :: Embedding -> Embedding -> Double
cosineSimilarity v1 v2 =
    let dotProduct = V.sum $ V.zipWith (*) v1 v2
        norm1 = sqrt $ V.sum $ V.map (** 2) v1
        norm2 = sqrt $ V.sum $ V.map (** 2) v2
    in dotProduct / (norm1 * norm2)

-- Chunking
chunkList :: Int -> [a] -> [[a]]
chunkList n = f
  where
    f [] = []
    f list = let (chunk, rest) = splitAt n list in chunk : f rest

-- Compute cosine similarity for a list of passages in parallel
computeSimilarities :: Embedding -> [IdEmbedding] -> [(Int, Double)]
computeSimilarities queryEmbedding passages =
```

```
        let compute idEmb = (fst idEmb, cosineSimilarity queryEmbedding (snd idEmb))
        in map compute passages

    -- Find the best match in a chunk of passages for a given query
    findBestInChunk :: Embedding -> [IdEmbedding] -> (Int, Double)
    findBestInChunk queryEmbedding passages =
        let similarities = computeSimilarities queryEmbedding passages
         in maximumBy (comparing snd) similarities

    findBestPassage :: Embedding -> [IdEmbedding] -> Int
    findBestPassage queryEmbedding passages =
        let chunks = chunkList chunkSize passages
            -- local maximum
            bestInChunks = parMap rdeepseq (findBestInChunk queryEmbedding) chunks
        -- Global maximum
        in fst $ maximumBy (comparing snd) bestInChunks
```

6.2 TF-IDF

Code for generating TF-IDF vectors for 200k passages. if there are any issues with indentation, please refer: link

```
import Data.Word
import Data.Char (toLower)
import qualified Data.Map as Map
import qualified Data.Set as Set
import System.IO ( hClose, hPutStrLn, openFile, IOMode(WriteMode),
    hGetContents, withFile, IOMode(ReadMode), hGetContents )
import Data.List.Split (splitOn)
import System.Directory (listDirectory)
import System.FilePath ((</>))
import Control.DeepSeq (deepseq)
import Control.Parallel.Strategies (using, parList, rdeepseq)
import Data.Time.Clock (getCurrentTime, diffUTCTime)
import Data.Time (diffUTCTime)
import Control.Concurrent.Async (mapConcurrently)
import qualified Data.Csv as Csv
import qualified Data.ByteString.Lazy as BL
import qualified Data.Vector as V
import Data.List (intercalate)
import Data.Maybe (fromMaybe)
import System.Environment (getArgs)


createWordList :: FilePath -> IO [String]
createWordList filePath =
    withFile filePath ReadMode $ \handle -> do
        content <- hGetContents handle
        content `deepseq` return ()
        return $ lines content

-- Type alias for better readability
type PassageMap = Map.Map String String

-- Function to read all passages from files in a directory
readPassagesFromDirectory :: FilePath -> IO PassageMap
readPassagesFromDirectory dir = do
    fileNames <- listDirectory dir
```

```
    let fileNameFullPath = map (\file_name -> dir ++ file_name) fileNames
    passageMaps <- mapM readPassagesFromFile fileNameFullPath
    return $ Map.unions passageMaps

readPassagesFromFile :: FilePath -> IO PassageMap
readPassagesFromFile fileName =
    withFile fileName ReadMode $ \handle -> do
        content <- hGetContents handle
        content `deepseq` return ()
        let linesOfFile = lines content
        let keyValuePairs = map parseLine linesOfFile
        return $ Map.fromList keyValuePairs

parseLine :: String -> (String, String)
parseLine line =
    let parts = splitOn "," line
    in case parts of
        (key:rest) -> (key, unwords rest)  -- Combine the rest into the passage
        _ -> error $ "Invalid line format: " ++ line

tfIdf :: [String] -> [String] -> Map.Map String Double -> Map.Map String Double
tfIdf wordsList passageWords idf =
  let totalWords = fromIntegral (length passageWords)
      wordCounts = Map.fromListWith (+) [(word, 1) | word <- passageWords]
  in Map.fromList [(word, (fromMaybe 0 (Map.lookup word wordCounts))
    * (fromMaybe 0 (Map.lookup word idf)) / totalWords) | word <- wordsList]

-- Convert a map of TF values to a CSV row format
mapToCsvRow :: [String] -> Map.Map String Double -> String
mapToCsvRow wordsList tfMap =
  let tfValues = [show (fromMaybe 0 (Map.lookup word tfMap)) | word <- wordsList]
  in intercalate "," tfValues

tfIdfForAll :: [String] -> [(String, String)] ->
    Map.Map String Double -> Map.Map String String
tfIdfForAll wordsList passages idf =
    let passageMap = Map.fromList passages
    in Map.map (\passage ->
        let passageWords = words (map toLower passage)
            tfValues = tfIdf wordsList passageWords idf
        in mapToCsvRow wordsList tfValues) passageMap


readMapFromCsv :: FilePath -> IO (Map.Map String Double)
readMapFromCsv filePath = do
  csvData <- BL.readFile filePath
  case Csv.decode Csv.HasHeader csvData of
    Left err -> error err
    Right vec -> return $ Map.fromList [(key, value) | (key, value) <- V.toList vec]

saveMapToCSV :: FilePath -> Map.Map String String -> [String] -> IO ()
saveMapToCSV path mapData wordList = do
    let csvData = Map.foldrWithKey (\key value acc -> [key, value] : acc) [] mapData
        headers = ["passageId", intercalate "," wordList] -- Headers for CSV
    BL.writeFile path $ Csv.encode (headers : csvData) -- Write to file
```

```
chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = let (ys, zs) = splitAt n xs in ys : chunk n zs

-- IDF LOGIC ############################
createWordMap :: FilePath -> IO (Map.Map String Double)
createWordMap filePath =
    withFile filePath ReadMode $ \handle -> do
        content <- hGetContents handle
        content `deepseq` return ()
        let wordsInFile = lines content
        let wordMap = Map.fromList [(word, 0) | word <- wordsInFile]
        -- Return the resulting Map
        return wordMap

idf :: [(String, String)] -> (Map.Map String Double) -> (Map.Map String Double)
idf passages initMap =
    let passageSets = map sentenceToSet passages
        idf_count = foldr addSetToMap initMap passageSets
    in idf_count

sentenceToSet (_, sentence) = Set.fromList $ map (map toLower) (words sentence)

addSetToMap passage_set doc_count = foldr
    (\word acc' -> Map.adjust (1.0 +) word acc') doc_count (Set.elems passage_set)

addDocToMap docMap count =
    let passages = Map.elems docMap
        passageSets = map sentenceToSet passages
    in foldr addSetToMap count passageSets

idfNormalise x totalDocCount = logBase 2 ((fromIntegral totalDocCount) / x)
-- ######################################

parIdf passage_chunks word_map passage_count =
    let par_output_idf = map (\input -> idf input word_map)
            passage_chunks `using` parList rdeepseq
        reduced_output = Map.unionsWith (+) par_output_idf
    in Map.map (\x -> idfNormalise x passage_count) reduced_output

parTf passage_chunks wordOrder norm_idf =
    let par_output = map (\t_p_input -> tfIdfForAll wordOrder t_p_input norm_idf)
            passage_chunks `using` parList rdeepseq
    in Map.unions par_output

-- time ./tf_idf_v2_par +RTS -N10 -ls -s
-- stack ghc -- -O2 -Wall -threaded -rtsopts -eventlog tf_idf_v2_par
main :: IO ()
main = do
    args <- getArgs
    case args of
        [dir_path, chunkSizeString] -> do
            let filePath = "output.txt"
            let chunkSize = read chunkSizeString
            wordOrder <- createWordList filePath
            word_map <- createWordMap filePath
```

```
all_passages <- readPassagesFromDirectory dir_path
let passage_chunks = chunk chunkSize (Map.toList all_passages)

-- IDF logic
let passage_count = Map.size all_passages
let norm_idf = parIdf passage_chunks word_map passage_count

--TF logic
let output = parTf passage_chunks wordOrder norm_idf
saveMapToCSV "tf_idf_par_output.csv" output wordOrder
```