COMS 4995 - Project Report - WordEmb

Yang, Yifan
`yy3185@columbia.edu`

Huang, Erik
`th2925@columbia.edu`

December 22, 2021

1 Introduction

For our final project, we are looking into a fundamental component for many natural language processing tasks, word embeddings. More specifically, we are revisiting, from the perspective of functional parallel processing, the static word embeddings derived from word co-occurrence matrix with a fixed-size context window and the truncated SVD method.

Word embeddings, or word vectors, aim to encode a word's meaning in a fixed-size vector, whose dimension is typically magnitudes smaller than the size of the vocabulary. The intuition behind the use of co-occurrence matrix is the distributional hypothesis [1]: the meaning of a word can be determined from the context it appears in. And words with similar meaning would occur in similar context.

To represent a word's context, we slide a fixed-size context window over the corpus and count all pairs of words that occurred in each other's context. The result is a co-occurrence matrix M, where element M_{ij} represents how many times a word i occurred in the context of the word j. For example, for the following corpus:

1. I enjoy flying.
2. I like NLP.
3. I like deep learning.

A context window of size 1 would produce the following co-occurrence matrix.

$$X = \begin{array}{c|cccccccc} & \textit{I} & \textit{like} & \textit{enjoy} & \textit{deep} & \textit{learning} & \textit{NLP} & \textit{flying} & . \\ \hline \textit{I} & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ \textit{like} & 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ \textit{enjoy} & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \textit{deep} & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \textit{learning} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ \textit{NLP} & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ \textit{flying} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ . & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}$$

Figure 1: Example co-occurrence matrix taken from [3]

Building upon the co-occurrence matrix, studies show that raw counts on occurrences are often not a great measure of association between words as they tend to be very skewed [2]. Therefore, to obtain the vector representation of a word, we compute what is called a Positive Pointwise Mutual Information matrix from the co-occurrence matrix, using the equation below.

$$\text{PPMI}(\text{word}_1, \text{word}_2) = \max(\log_2(\frac{P(\text{word}_1, \text{word}_2)}{P(\text{word}_1)P(\text{word}_2)}), 0) \tag{1}$$

Finally, because the vector representations we have obtained from the co-occurrence matrix has very high dimension ($d = |V|$, where V is the vocabulary set), we want to obtain an approximate representation using fewer dimensions. One way to do so is the truncated Singular Value Decomposition method, in which we truncate a matrix M to the top k singular values.

$$\underset{|V| \times |V|}{\begin{bmatrix} \\ \quad X \quad \\ \\ \end{bmatrix}} = \underset{|V| \times k}{\begin{bmatrix} \\ \quad W \quad \\ \\ \end{bmatrix}} \underset{k \times k}{\begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_k \end{bmatrix}} \begin{bmatrix} \quad C \quad \\ k \times |V| \end{bmatrix}$$

Figure 2: Visualization of truncated SVD matrix [2]

The $|V| \times k$ matrix W would be the final word embedding, where k is the dimension of the embedding. The obtained embedding can then be used for downstream NLP tasks such as sentiment analysis, named entity recognition, etc.

2 Objective

As outlined in the introduction, we aim to implement a parallel algorithm in computing the static word embedding. The procedure that we will be mainly focus on are:

1. Parallelizing the computation of co-occurrence matrix, which involves how do we partition the corpus and combine intermediate results. This would be challenging because the co-occurence matrix would be incredibly sparse, and threads/workers cannot afford maintain an entire co-occurrence matrix in the memory.
2. Parallelizing the computation of the PPMI matrix. This poses similar challenges as above.

We have decided to use an existing solver for SVD (SVDLIBC by Doug Rohde) because it is overly complicated to demonstrating Haskell's parallelism mechanics in a complex algorithm such as SVD.

3 Problem Formulation

In this section we outline the input and output of our program.

The program takes 4 inputs. The first input is a file path to the input corpus file. The corpus file are sentences separated by a new line character. The second input is a tuneable depth parameter that dictates the parallelization depth of the program. The third parameter is the dimensionality of the final word embedding. The last parameter is the path to which the program writes the final output to.

The output is a text file where each line is the computed word embedding for a given word. Each word is converted to lower case and the number of dimension matches the dimension specified in the input. Each dimension is represented by a real number (possibly in scientific notation). The format is as follows:

word [dim1] [dim2] [dim3] ...

4 Implementation

In this section we describes our implementation for each step in computing the word embeddings.

4.1 Corpus Pre-processing

There are two tasks in our preprocessing step. The first task is to tokenize, clean, and format our corpus. We split the corpus by new lines to obtain a list of sentences, then tokenize each sentence into a list of tokens. We then filter out invalid tokens, such as punctuations and tokens that include non-alphabetical characters. Lastly, we converted all the tokens to lower case for good measure.

The second task is to build a word-index dictionary that maps each word to a integer index. This is done because in our later computations, we are going to index our matrix with word indices instead of strings.

4.2 Computing Co-occurrence Matrix

Word co-occurrences are very sparse in languages, which means that each word would only co-occur with a very limited set of other words. Therefore, if we represent co-occurrences in a dense matrix, most entries in the matrix will be 0 (we can see this effect in the previous toy corpus). Therefore, dense matrix representation for co-occurrences are bound to be wasteful, and sometimes not practical. For instance, in the Brown Corpus we tested our program on, there are well over 40,000 unique words. Representing a 40,000 $\times$ 40,000 integer matrix in a dense representation would have required 11.92 GB on a 64-bit machine. In addition, less than 0.2% of the co-occurrence matrix have non-zero values.

Thus, we decided to represent our co-occurrence data in a sparse matrix representation using a map of matrix coordinate to value: `Map (Int, Int) Int`. This is done to reduce the memory profile and increase the efficiency of our program.

The computation of co-occurrence matrix can be parallelized because the computation for each sentence is independent of each other, and the final co-occurrence matrix can be combined by adding together the matrices for each sentence.

We parallelize this step by recursively partitioning our corpus into two halves until a certain depth d, and compute each partition in parallel. After the computation for both partitions has finished, we sum the two sparse matrices using `Map.UnionWith (+)` to get the final result.

4.3 Computing PPMI Matrix from the Co-occurrence Matrix

This step transforms each value of the co-occurrence matrix, which is the raw count of co-occurrences of a word with another word, to a better measure that gives more information about whether a context word is particularly indicative of a target word.

Again, each cell in the co-occurrence matrix can be transformed without dependency on any other cells. We effectively perform a parallel `map` operation. Again, we recursively partition the co-occurrence matrix until a certain depth, apply our transformation at the root, then recursively build up to the final PPMI matrix by combining each sub-map.

4.4 SVD Truncation

As previously established, our resulting matrix would be very sparse, and also not every dimension would be indicative or useful in the downstream tasks. Therefore, we build a more compact vector representation of words by truncating the original matrix down to the top k singular value and their respective dimension using SVD. We relied on an existing solver for this step because solving SVD is not naturally parallelizable and they are inappropriate in demonstrating Haskell's parallel mechanism.

As a result, we project the original matrix, which might have over 40,000 dimensions, down to the top k (e.g., 100) dimensions and we consider this as our final word embeddings.

5 Evaluation

Hardware

The processor which we perform our analysis on is *2.3 GHz 4-Core Intel Core i5* with Hyper Threading.

Evaluation Method

We used *Brown University Standard Corpus of Present-Day American English* as our input text file. There are two parameters we can tune 2 variables to analyze the performance: **N**, **d** (the recursive depth level of partition).

Balancing and Granularity Test

In Table 1, we experimented our implementation with depth fixed at 5 and different **N** values. We found that the program is most efficicent when $\mathbf{N} = 8$, twice as much as the number of cores we have. We think that this is because of the hyper-threading of the CPU. As **N** goes higher, the time cost starts to increase again due to the increased amount of overhead. Our conversion rate remains constant when $\mathbf{N} > 1$. The reason might be that the depth is rather shallow at $\mathbf{d} = 5$. So that we will further test the effect of different **d** values.

Threadscope Analysis

Figure 3 and 4 demonstrates the eventlog from running the program with $\mathbf{N} = 4$, $\mathbf{d} = 5$ and $\mathbf{N} = 8$, $\mathbf{d} = 5$. We can observe the sequential preprocessing of data in the beginning. Following is the parallelization of co-occurrence matrix formation. Since the final folding must happen on one HEC, we can see that towards the end of this step the task becomes unbalanced. At the very end we have the PPMI matrix conversion, which is also parallelized.

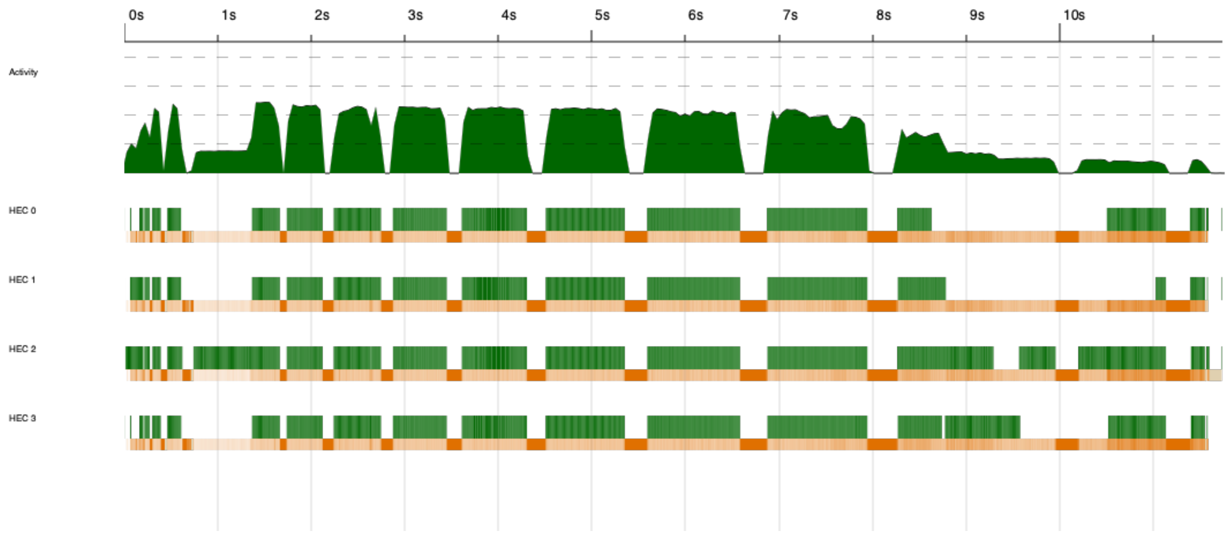

Figure 3: Eventlog of N=4 d=5

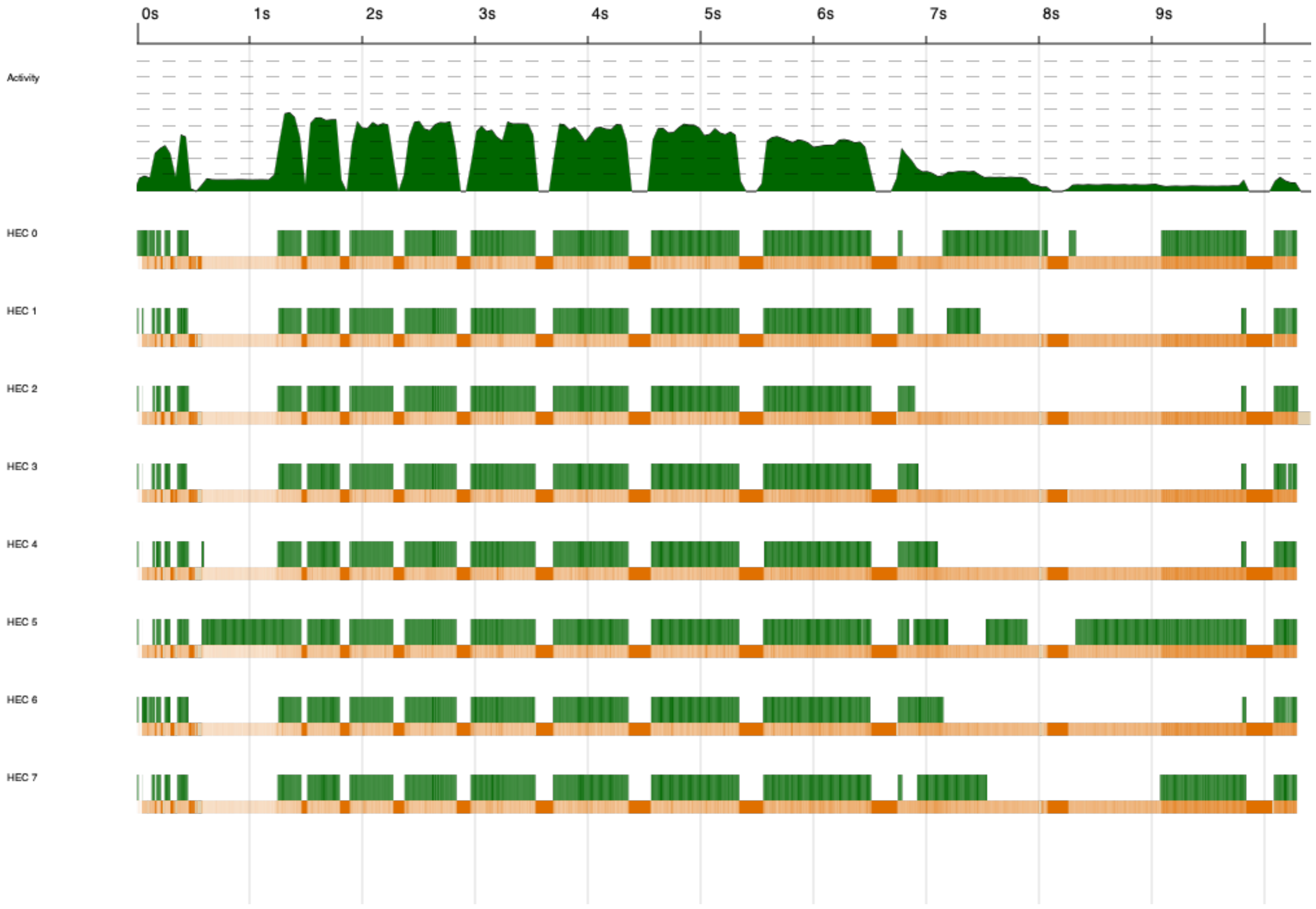

Figure 4: Eventlog of N=8 d=5

Table 1: Experiment with different **N** and **d** = 5

N	Speedup	time	total	converted	overflow	GC'd	fizzled
1	1	28.565	127	0	0	2	125
2	1.66	17.159	127	64	0	0	63
4	2.44	11.689	127	64	0	0	63
6	2.59	11.021	127	64	0	0	63
8	2.69	10.620	127	64	0	0	63
10	2.36	12.071	127	64	0	0	63
16	1.97	14.520	127	64	0	0	63

In table 2 we fixed our **N** value to be 8 and test out different values for **d**. The number of total sparks is approximately $2^{\mathbf{d}+2} - 1$. The conversion rate stays around 50% until the value of **d** becomes very large. At $\mathbf{d} = 20$, we have a conversion rate of 6% and more than half of the sparks created are garbage collected. The optimal value for depth **d** seems to be 5.

Table 2: Experiment with different **d** and **N** = 8

d	Speedup	time	total	converted	overflow	GC'd	fizzled
0	1	33.940	3	2	0	0	1
1	1.52	22.324	7	4	0	0	3
2	2.25	15.083	15	9	0	0	6
3	2.78	12.206	31	18	0	0	13
4	2.71	12.520	63	32	0	0	31
5	3.18	10.674	127	64	0	0	63
6	3.11	10.892	255	128	0	0	127
8	3.00	11.290	1055	530	0	0	525
10	2.8	12.132	4619	2320	0	1	2298
20	2.45	13.841	4352473	263946	951	2629877	1392163

6 Conclusion

In this project, we presented the parallel computation schemes for two important steps in the computation of word embedding. We demonstrated a sparse matrix representation for word co-occurrences that both drastically reduce memory usage and increased our computation efficiency. We leveraged parallelization and lazy constructs in Haskell, such as `Control.DeepSeq` and `Control.Parallel.Strategies`, to build a parallel program that gained noticeable speedups from the sequential counterpart.

7 Code

app/Main.hs

```haskell
module Main where

import Lib
import System.Exit(die)
import System.IO
import System.Environment(getArgs, getProgName)
import qualified Data.Text.IO as TIO
import qualified Data.Text as T
import qualified Data.Map as M
import qualified Numeric.LinearAlgebra as LA
import qualified Numeric.LinearAlgebra.Data as NL
import qualified Numeric.LinearAlgebra.Devel as NLD
import qualified Numeric.LinearAlgebra.SVD.SVDLIBC as SVD
import qualified Control.DeepSeq as DS
import Foreign.C.Types(CInt)

toCInt :: Int -> CInt
toCInt x = (fromIntegral x) :: CInt


main :: IO ()
main = do
  args <- getArgs
  case args of
    [input, depth, vector_length, output] -> do
      outfile  <- openFile output WriteMode
      contents <- TIO.readFile input
      let corpus      = lineTokenize contents
          vocab       = vocabulary corpus
          cm          = parCooccurenceMatrix (read depth::Int) corpus
           ↪  vocab 5
          dim         = M.size vocab
          pmi         = ppmi (read depth::Int) cm dim
          pmis        = NLD.mkCSR $ M.toList pmi
          (u, s, vt)  = SVD.sparseSvd (read vector_length::Int) pmis
          w           = DS.force (NL.tr' u) LA.<> (NL.diag s)
          embs        = map (\i -> T.unwords $ map (T.pack . show) $
           ↪  NL.toList $ w NL.! i) [0..dim-1]
          word_embs   = zip (M.keys vocab) embs

      -- putStr $ show $ M.size pmi
      mapM_ (TIO.hPutStrLn outfile) $ map (\(word, emb) -> T.unwords
       ↪  [word, T.pack " ", emb]) word_embs
      hClose outfile
    _ -> do
      pn <- getProgName
      die $ "Usage: "++pn++" <input_filename> <depth> <vector_length>
       ↪  <output_filename>"
```

src/Lib.hs

```
module Lib
     ( module Utils,
       module WordEmbSeq,
       module WordEmbPar,
      module Matrix
    ) where

import           WordEmbSeq
import           WordEmbPar
import           Utils
import           Matrix
```

src/Matrix.hs

```
module Matrix
    (
      ppmi
    ) where

import qualified Data.Map as M
import qualified Data.Vector.Unboxed as V
import qualified Control.Parallel.Strategies as ST

rowSum :: M.Map (Int, Int) Int -> Int -> V.Vector Int
rowSum m dim = V.accum (+) acc items
    where items = map (\(k, v) -> (fst k, v)) $ M.toList m
          acc   = V.generate dim (\i -> 0)

colSum :: M.Map (Int, Int) Int -> Int -> V.Vector Int
colSum m dim = V.accum (+) acc items
    where items = map (\(k, v) -> (snd k, v)) $ M.toList m
          acc   = V.generate dim (\i -> 0)

-- Note: Int potential overflow? yes.
-- However, wc -w brown.txt -> 1161192 << 2^64 so we are in the safe
↪  zone for now
matSum :: M.Map (Int, Int) Int -> Int
matSum = sum . M.elems

ppmi :: Int -> M.Map (Int, Int) Int -> Int -> M.Map (Int, Int) Double
ppmi depth m dim = _ppmiMatPar depth pwc pw pc
    where
        n = matSum m
        (pw, pc, pwc) = ST.runEval $ do
            rs' <- ST.rpar $ V.map (`floatDiv` n) $ rowSum m dim
            cs' <- ST.rpar $ V.map (`floatDiv` n) $ colSum m dim
            pwc <- ST.rpar $ M.map (`floatDiv` n) m
            return (cs', rs', pwc)

_ppmiMatPar :: Int -> M.Map (Int, Int) Double -> V.Vector Double ->
↪  V.Vector Double -> M.Map (Int, Int) Double
_ppmiMatPar 0 pwc pw pc = _ppmiMat pwc pw pc
```

```haskell
_ppmiMatPar d pwc pw pc = M.unionWith (+) m1 m2
    where
        maxKey = fst . fst $ M.findMax pwc
        minKey = fst . fst $ M.findMin pwc
        mid = (minKey + maxKey) `div` 2
        (x,y) = M.split (mid, -1) pwc
        m1 = ST.runEval $ ST.rpar (_ppmiMatPar (d-1) x pw pc)
        m2 = ST.runEval $ ST.rpar (_ppmiMatPar (d-1) y pw pc)

_ppmiMat :: M.Map (Int, Int) Double -> V.Vector Double -> V.Vector
 ↪  Double -> M.Map (Int, Int) Double
_ppmiMat pwc pw pc = M.mapWithKey eval pwc
    where eval = _ppmiCell pw pc

_ppmiCell :: V.Vector Double -> V.Vector Double -> (Int, Int) -> Double
 ↪  -> Double
_ppmiCell pw pc (i, j) pwc | pwpc == 0 = 0 -- ppmi is +infinite
                           | pwc  == 0 = 0 -- ppmi is -infinite
                           | otherwise = max 0 (logBase 2 pmi)
    where pwi  = pw V.! i
          pcj  = pc V.! j
          pwpc = pwi * pcj
          pmi  = pwc / pwpc

floatDiv :: Int -> Int -> Double
floatDiv a b = (fromIntegral a) / (fromIntegral b)
```

src/Utils.hs

```haskell
module Utils
   (
      lineTokenize,
      vocabulary
   )
   where

import qualified Data.Text as T
import qualified Data.Set as S
import qualified Data.Map as M
import qualified Data.Text.IO as TIO
import Data.Char ( isSpace, isAlpha, toLower )

validToken :: T.Text -> Bool
validToken = T.all isAlpha

cleanTokens :: [T.Text] -> [T.Text]
cleanTokens = filter validToken . map T.toLower
```

```
wrapStartEnd :: [T.Text] -> [T.Text]
wrapStartEnd seq = [T.pack "<START>"] ++ seq ++ [T.pack "<END>"]

-- Drop empty sequences
validSequence :: [T.Text] -> Bool
validSequence = not . null

-- Seperate text into list of list of words
lineTokenize :: T.Text -> [[T.Text]]
lineTokenize contents = map wrapStartEnd $ filter validSequence $ map
↪  cleanTokens $ map T.words $ T.lines contents

-- Output all words into a map vocabulary
vocabulary :: [[T.Text]] -> M.Map T.Text Int
vocabulary corpus = M.fromList $ zip (S.toAscList . S.fromList $ concat
↪  corpus) [0..]
```

src/WordEmbSeq.hs

```
module WordEmbSeq
   (
      cooccurence,
      cooccurenceMatrix
   )
   where

import qualified Data.Text as T
import qualified Data.Vector as V
import qualified Data.Map as M
import Data.Maybe(fromMaybe)

-- create cooccurence sparse matrix
cooccurenceMatrix :: [[T.Text]] -> M.Map T.Text Int -> Int -> M.Map
↪  (Int, Int) Int
cooccurenceMatrix corpus vocab window = foldr (M.unionWith (+)) M.empty
↪  cooccurences
   where cooccurences = map (\line -> cooccurence (V.fromList line)
    ↪  vocab window) corpus

-- cooccurence(sentence, vocab, window_size) -> cooccurence matrix
cooccurence :: V.Vector T.Text -> M.Map T.Text Int -> Int -> M.Map
↪  (Int, Int) Int
cooccurence sentence vocab window =
  foldr (M.unionWith (+)) M.empty $ map (_cooccurence sentence vocab
   ↪  window) [0..V.length sentence -1]

_cooccurence :: V.Vector T.Text -> M.Map T.Text Int -> Int -> Int ->
↪  M.Map (Int, Int) Int
```

```haskell
_cooccurence sentence vocab window i = M.fromListWith (+) $ V.toList $
 ↪  left_cooccur V.++ right_cooccur
   where center_word                        = sentence V.! i
         (left_context, right_context) = _context sentence window i
         left_cooccur                       = V.map (\left -> ((vocab M.!
          ↪  center_word, vocab M.! left), 1)) left_context
         right_cooccur                      = V.map (\right -> ((vocab M.!
          ↪  center_word, vocab M.! right), 1)) right_context

_context :: V.Vector T.Text -> Int -> Int -> (V.Vector T.Text, V.Vector
 ↪  T.Text)
_context sentence window i = (V.slice left_start left_len sentence,
 ↪  V.slice (i + 1) right_len sentence)
   where left_start = max 0 (i - window)
         left_len   = i - left_start
         right_end  = min (i + window) (V.length sentence - 1)
         right_len  = right_end - i

toWordsMatrix :: M.Map (Int, Int) Int -> M.Map T.Text Int -> M.Map
 ↪  (T.Text, T.Text) Int
toWordsMatrix cm vocab = M.fromList $ map (\(k,v) -> ((toWords M.! (fst
 ↪  k), toWords M.! (snd k)), v)) $ M.toList cm
   where toWords = revertMap vocab
         revertMap m = M.fromList $ map (\(x,y) -> (y,x)) $ M.toList m
```

src/WordEmbPar.hs

```haskell
{-# LANGUAGE BlockArguments #-}
module WordEmbPar
   (
      parCooccurenceMatrix,
   )
   where

import qualified WordEmbSeq as WES
import qualified Utils as U
import qualified Control.Parallel.Strategies as ST hiding(parMap)
import qualified Data.Text as T
import qualified Data.Vector as V
import qualified Data.Map as M

parCooccurenceMatrix :: Int -> [[T.Text]] -> M.Map T.Text Int -> Int ->
 ↪  M.Map (Int, Int) Int
parCooccurenceMatrix 0 corpus vocab window = WES.cooccurenceMatrix
 ↪  corpus vocab window
parCooccurenceMatrix d corpus vocab window = M.unionWith (+) m1 m2
   where
      (x,y) = splitAt ((length corpus) `div` 2) corpus
```

```
        m1 = ST.runEval $ ST.rpar (parCooccurenceMatrix (d-1) x vocab
         ↪  window)
        m2 = ST.runEval $ ST.rpar (parCooccurenceMatrix (d-1) y vocab
         ↪  window)
```

References

[1] Association for Computational Linguistics. *Distributional Hypothesis*. Dec. 2010. URL: https://aclweb.org/aclwiki/Distributional_Hypothesis.

[2] D. Jurafsky. *Vector Semantics*. 2019. URL: https://web.stanford.edu/~jurafsky/li15/lec3.vector.pdf.

[3] C. Maning. *CS224n: Natural Language Processing with Deep Learning*. 2019. URL: http://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes01-wordvecs1.pdf.