Parallel Cryptanalysis

A Study of the Vigenère Cipher in Haskell

Final Project: Parallel Functional Programming

Fall 2022

Alexander Nicita
an2582@columbia.edu

Introduction

This paper presents a parallelized Vigenère cipher solver in Haskell. As Marlow’s parallelized Haskell RSA encoder and decoders\(^1\) have shown, the process of parallelizing encryption and decryption in Haskell is a trivial task, meaning that there is not an inherent aspect of the algorithm that can be accelerated by leveraging multiple cores. Alternatively, the process of parallelizing cryptanalysis, the formal technique of breaking encryption schemes, often relies on algorithms which do, in fact, have aspects that can be parallelized. For certain ciphers, including the Vigenère cipher, algorithms to search for the encryption key underlying a ciphertext behave akin to graph algorithms, which parallelize rather well.

----

Vigenère Cipher

The Vigenère cipher was created in the 16th century as a successor to the Caesar cipher, which was among the earliest substitution ciphers in history. As techniques to break the Caesar cipher, including early forms of frequency analysis, began to rise in popularity, the puzzling Vigenère cipher remained unbreakable for over 300 years after originally being published. The fundamental premise of the Vigenère cipher is a repeating keyword. Quite literally, plaintext messages are encrypted by performing an XOR with a chosen keyword that repeats as long as the plaintext message. More formally, the algorithm can be represented as follows:

\[ \text{ALGORITHM} \]

\text{INPUT:} \text{ Plain text } m [ ]

\hspace{1cm} \text{Key } e [ ]

\text{OUTPUT:} \text{ Cipher text } c [ ]

\begin{align*}
\text{Step1} & \quad i = 0; \\
\text{Step2} & \quad n = \text{length} (m); \\
\text{Step3} & \quad c = 00; \\
\text{Step4} & \quad \text{for } i \leq n \text{ do;} \\
\text{Step5} & \quad c[i] = m[i] \oplus [i \mod n] \mod 26; \\
\text{Step6} & \quad \text{end for;} \\
\text{Step7} & \quad \text{return } c.
\end{align*}

\[ \text{https://link.springer.com/article/10.1007/s40305-020-00320-x} \]
In the context of this project, we will consider a cryptanalytic attack on the cipher to be a modified version of ciphertext only attack. The COA\(^3\) setting is where a malicious actor can only access a ciphertext yet must deduce the encryption key used to encrypt the ciphertext. The modifications to COA made in this project are that we will be responsible for both creating and breaking ciphertexts, meaning that we know the correct encryption keys. However, this knowledge is not used during the process of analyzing ciphertexts and outputting a deduced key. Instead, this knowledge is only used to verify that our solver’s output is, in fact, the original correct encryption key.

There exists a number of known solutions to finding the encryption key for a Vigenère cipher provided only with a ciphertext\(^4\). One such solution, the Kasiski Test\(^5\), relies on the fact that identical plaintext separated by the length of the key will map to identical ciphertext. However, this approach is rather fickle and inelegant, as it relies on brute force frequency analysis rather than probabilistic or graphical approaches. The second popularized solution, which is the technique implemented in this project, is the Index of Coincidence algorithm\(^6\). Relying on the distribution of letter usage in an alphabet, this algorithm will first deduce the length of the encryption key by comparing potential alphabet distributions according to key size (limited to 20 for this project). These steps of calculating distributions for key length do not depend on any other such calculations by key length, meaning that it can be parallelized. After determining an optimal key length, the algorithm then will output an encryption key in order to maintain the letter frequency distribution for a given alphabet.

---

3 [https://en.wikipedia.org/wiki/Ciphertext-only_attack](https://en.wikipedia.org/wiki/Ciphertext-only_attack)
4 [https://www.cs.purdue.edu/homes/ninghui/courses/Fall05/lectures/355_Fall05_lect04.pdf](https://www.cs.purdue.edu/homes/ninghui/courses/Fall05/lectures/355_Fall05_lect04.pdf)
Haskell Solver

The solver implementation in haskell is contained in two files, Main.hs and Cipher.hs, both of which are inside the vigenere project directory. Main.hs contains the 3 cipher texts being studied in this project and Cipher.hs contains the sequential and parallel algorithms. The output of the solver algorithm can be printed in the terminal console. Below is one such example of the algorithm’s output, where the key and decrypted text are outputted. This example is a paragraph from the Columbia University wikipedia page encrypted under the key “Columbia”, which the solver is able to successfully return even though it was provided only with the encrypted ciphertext of the below paragraph. Running make test inside the vigenere directory will also output this result.

Key: COLUMBIA
Decrypted Text:
Above is the threadscope result of running the sequential algorithm to break Vigenere ciphers on 3 ciphertexts. The code takes 2.18 seconds to find the encryption keys successfully for each ciphertext, with the 1 CPU running the algorithm maintaining near peak performance throughout the entire process.

The step of the algorithm which calculates letter distributions for each possible key length can be parallelized. The next step of the algorithm which determines the encryption key after calculating key size can also be parallelizing. Using a combination of `parMap rdeepseq` and `parList rseq` for each of those steps leads to the next result of the project, where a parallelized version of the sequential algorithm is run on 2 cores, leading to an algorithm which runs in 1.34 seconds. This is a 1.6x speedup, with some time lost due to garbage collection introduced when running the algorithm on multiple cores. Below is the threadscope result for running the parallelized algorithm on 2 cores.
However, when this same parallelized algorithm is running on 8 cores, there is a significant amount of sparking. See below for the threadscope results of running the algorithm on 8 cores.
The approach taken in this project to potentially solve this problem with 8 cores is chunking, which can be applied to the `parList rseq` parallelization by instead using `parListChunk 100 rseq` where 100 is the size of the chunk processed by that step in the algorithm. Unfortunately, even the chunking approach did not solve the sparking problem, as is shown in the below threadscope result for parallelized chunking of the Vigenère solver:

![Threadscope result](image)

While there appears to be a more even load among the 8 cores when utilizing chunking, there is also more garbage collection toward the end of the algorithm, leading to roughly the same results as the non-chunking algorithm.
Overall, a direction for future work in this project is to experiment with additional chunking sizes, which will be further discussed in the conclusion of this document. For now, there was considerable speedup around 3 cores for the parallelized implementations, but then performance plateaus, with a monotonically increasing time to run the algorithm between 4 and 8 cores. These results are visualized below for both the chunking and on-chunking implementations of the solver.

![Parallel Vigenere: Chunking vs. No Chunking](chart.png)

**Conclusion**

In conclusion, this project has studied the parallelization of the Index of Coincidence algorithm for solving Vigènere ciphers. This algorithm was able to achieve a roughly 2x speedup versus its sequential alternative when run on 3 cores. Beyond 3 cores, significant sparking was experienced, which was slightly alleviated by chunking, but also led to more garbage collection. Future work should explore additional chunking sizes as well as a larger set of ciphertexts to further explore performance results.
module Main (main) where

import Cipher (vigenere, parallelVigenere, parallelChunkingVigenere)

main :: IO ()
main = do
    let crypt1 = "LOEDSVIDPSWILSYECLSZISGHIHQWNAOIEPGBVZXLNDMJTTLRCSTJSWUTREZEVLSOVLEPATILOUXHWGE
NUPANHRACKKEEXWYTRWTAIAKGZLSPMJXWZUXOWRIFHHLACPOLTRWTRQEVEVXSNASRACXHLRVLVMYWXN2RYVRSJR
PMNJUVNNAAIIGHNWUTRIJVLFPMLWYDEMLNUTRITXTCHSDKXLCHWFLRLIBHOCXZT2HTGMAFVEVVAHOG
WWISFJYMIBWLLMTTLOEWEILWIPCEUSOGCHNWZXUOZSFILYISAHDLGHZWZUOTJTXODIWASZXBECISLO
FSDKIAIKZGFQCWVPAQSSSKSRTKTOEOUPTMLHPHDFGCVJKPVILLIHPOPTMLSCTSNRPASVYAJOLPSUDPSMAMSE
AYHYZWGXIISMNHDULWUGSRVVMXZNEIIPLXJXKRINVMACHQKCIJAIGYROWSAEPLQOPTMFRTVVBNNMEMBE
TSDDYRSAFAZDLNSXPMTYELAEHXSZCDWAXXSQFEVBHUGCJQVJAEVXBEVMNEILFAKEFDVONCRPIRXZHTRIP
RRVTQYKXKMTXVIKZRMKPEIXLILVDFLWGVFLVPFAHORWILSGKLYHTQOSAGUQLOICMVKILJNENEKLCIOCMCLO
ASENULSYDIWAGAHVVXUQVLLIEWILYTMEMHVZQASPTGEVVJPEXHKPCFXZLXKRYILEBGRDLOHYSFKLMR
YMPDWTSMKINNLAMILAIAASOYENKHBJDMJAMTJMTJXHZNHTUAMNXXGNDLWGENUGZ
VKOXZTMRRLAHQOWWGGICIKZMKGVDRWRPVODBXWNXOSISHYELIDUSRKLLOADXZTQUXAWLIMFVCIFPSIOWGURAK
YJLDYWOPVMTGUHSQXSGSKMWDEXCWGRIJLPWZOPONGRISRRVNAVPLKLRWKLSSWKJTPCMMKAXHDVLTRVLRNNMKSE
MEIVXUKJTPSDMPNWZDADCMCLABITTPSLNGELYLLPLJXGVOEOCUSRSVSECEUWUITYAWSLRTVTEIWZTREL
CEMVHADKMFQKRGSJIAIXQOXXYMRKRRVXHUMLEVAZHGJQCRENAILBCSRCYORBWYRT2FSFS11NFOPLVZTIX
MCIGGAIZIKWEOZXPWPKHGOXAAAPHVJDJXENZTXTLR2ZDHGZCE1E1ZVOLOSCHSEJSJIANBPVNDSTYLABICPZL
UXWWFDQYMEMERYQVLWINRZWAODLWRLAGUSDMPFSLIJJLWHDETGCUSIFSATVIERTJTJHMUCRGKELZEFACYKXRWRF
LOYQIFIPFBZKBBWPIPOEOXKWIILEJYNXNIZXRWJVLCDMCGKWH2TOYMOAOCKSK5TJRSKVRSMJRNTSFISRI
JFLJLAUWGVDUXZTHXTJCRNLGNWUCQOBGEIOJXKRLWHLQDODIPPVMMUSRSK_OWLVVEPKBFNOPWUDLRLRVTABOFD
WOFRWYJEWLXGEVYIDLPOLNPOLSYMNGGTECPVAHOHWHOIGTAUGIJCFWXJBSBZTMRYIWSSLYLILESJDEIE
FASRUWWMVOCACKVRRLUHGOAIALLFLTVPSRHQSSRQVUGMETLSIKIWFIPIQGRRSMGZPWSRYDADERPRZDEWEHY
NRGGXZHGBRMHEHEIEWEHCLILWENEFUSQUSIESKTVGFVFAYONSHFWRZJWCISVFESJAHOSLGITYIDFOXMMXWAFFG
VTRMEJWGIHTVRDXTZEVVWNVSNFSLNLRSMKNGUEAASSHMOPETSJKCWGILEPHGVBVCJTHOLELDCXJDOEJ
YHVNDLUSEVBUPLDZTMCMEFATYSILEJRJLEUMFVAOQLRZOBQWBSRZWHHNYXZTVGFYFTVREARENESLAEVTPXJ
IRQHXHENKAJLWZCEBDXVHPKUHGSWCWONIDSTRDVDFJWNJOAWLIERWLOPTGYSMCHKLWFZUESYSFVFUBFG
ILGFKWAHSQKJGVSFZEHMLHIRXISUTXCLTRVHOOMSETTWVNLIDJGGJQSSCYPEILESJAHAXZRFJVSIC
FWCROOAZHKTLSXEDSRLSDSSDRSKLASSOIEHOJTWREDLACKSJKJXJHQQ1JJDWSXZV
AFLWIKZRYKUXGSAHVRLCKQWXHTYSMDOBDZXXHRRWYOCXWGSXMXXLGBISIOEKEOBIIWNVOLKPYKTRPRNVV
RMLYYLILEJQOHNNJSCSUWTVPTVIPUPDSDJAEILABWCEOCOPTRXNGVISPSWWUFWGWJAZWTOHTXHRXEVSDAHPO
PEDARKMLDVTOIOAXIGUGAHNGYBOKSCEDZWEHLSFUPITLXLPVZPXBPDVNKTGMVDKVJWOPSETZDHRYOYJUD
let crypt2 = "BXNSTLQXMTHCFMKXSZFWJGQTMYFKMTCVDCVTKEUBVTNCZTBXNWGWVRQIORSVGVHTDSAROBKFMTPRTHFZWPFVXRQZGZBCHBGIMKYWBUPCMICQAHPACCTICTIYKBIADQTNNVUFQMWHQQBOLBJWBNMABBIGVZXDZBQWQGDIEUWYVMFNTNVWACMCVMNZBQGDSVRQGMJZ2RMMLWVNWNUNUHMQUPBTXRNSBREKAGGETESYHJZXFHCSMOSQBEQPCUNSACFVWKBQJFXKQDFWFTLJQC2ZGMVGCUBFLHWPTRXZXPQRAXHQPKTPRNCLEQOMXTHFCFMKPSJBPZ2PDMVAFBPQFSGJUXUHIQZAMTOVFBBMKVCFCGACPGVOMJSUOVBHCBAWHBWPUBKBCQTPNTUCMGQQ2HTQSTWXKMTWENQIOIRZDWKFHNUPKTPWGCJMOYTFPLPKPBVGBKRWOMJSAWJQEPWYQITEZKFQABUFTTUMFXHQ2WSDWXRCERSILNCVTBATOOBSQMCANPCIQKMEJAVQBEETPERSRJLQOWGGGBUHTTSVPUBBMMBCQSGINGHHEBSSMKJGXYMOXPSZJNMJSTBOOXBHKUBQCGCBLBVBCQMNUBLTMMETBAGMBXKWMBNFJFTYTSHYJXUJUAJUTNGBEVWKFMOZ8M1CUMFZXXDFWNEKTHUWABXUXHSQGKVPIFJATPRVSBELEMEOVIFIOGOVQSMCQWV1MGHHEBSSTVQYMIVESXGJZVVMYPZGGBCSWHHCNJPZ2DEVIVOILRFWBGWWYVIGIIERSVREA282M1MYS2RHWGG"
putStrLn ""

vigenere crypt3
putStrLn ""

putStrLn ""

parallelVigenere crypt1
putStrLn ""

putStrLn ""

parallelVigenere crypt2
putStrLn ""

putStrLn ""

parallelVigenere crypt3
putStrLn ""

putStrLn ""

parallelChunkingVigenere crypt1
putStrLn ""

putStrLn ""

parallelChunkingVigenere crypt2
putStrLn ""

putStrLn ""

parallelChunkingVigenere crypt3
putStrLn ""
module Cipher (vigenere, parallelVigenere, parallelChunkingVigenere)
where

import Data.List (transpose, nub, sort, maximumBy)
import Data.Ord (comparing)
import Data.Char (ord)
import Data.Map (Map, fromListWith, toList, findWithDefault)
import GHC.Conc (par, pseq)
import Control.Parallel.Strategies (Strategy, evalList, using, parMap, rdeepseq, rparWith, parListChunk, rseq)
import Data.Ratio (denominator)

vigenere :: [Char] -> IO ()
vigenere crypt = do
  let filteredCrypt = filter (/= ' ') crypt
      dists = fmap (wrap filteredCrypt) [1..length filteredCrypt `div` 20]
      optimalDist = maximumBy (comparing rate) dists
      key = fmap (deriveCharacter engFreqs) optimalDist
      chars a b = ['A'..'Z'] !! ((ord b - ord a) `mod` 26)
      putStrLn key

parList :: Strategy a -> Strategy [a]
parList strat = evalList (rparWith strat)

parallelVigenere :: [Char] -> IO ()
parallelVigenere crypt = do
  let filteredCrypt = filter (/= ' ') crypt
      dists = parMap rdeepseq (wrap filteredCrypt) [1..length filteredCrypt `div` 20]
optimalDist = maximumBy (comparing parallelRate) dists
  `using` parList rseq
key = parMap rdeepseq (deriveCharacter engFreqs) optimalDist
putStrLn key

parallelChunkingVigenere :: [Char] -> IO ()
parallelChunkingVigenere crypt = do
  let filteredCrypt = filter (/= ' ') crypt
  dists = parMap rdeepseq (wrap filteredCrypt) [1..length filteredCrypt `div` 20]
  optimalDist = maximumBy (comparing parallelRate) dists
  `using` parListChunk 100 rseq
key = parMap rdeepseq (deriveCharacter engFreqs) optimalDist
putStrLn key

engFreqs :: [Double]
engFreqs = [0.081, 0.014, 0.027, 0.042, 0.127, 0.022, 0.020, 0.060, 0.069, 0.001,
  0.007, 0.040, 0.024, 0.067, 0.075, 0.019, 0.000, 0.059, 0.063, 0.090,
  0.027, 0.009, 0.023, 0.001, 0.019, 0.000]

aggregate :: Ord a => [a] -> Map a Int
aggregate = fromListWith (+) . fmap (, 1)

avg :: Fractional a => [a] -> a
avg as = sum as / fromIntegral (length as)

parse :: Int -> [a] -> [[a]]
parse _ [] = []
parse n as = let (h, r) = splitAt n as in h:parse n r

wrap :: [a] -> Int -> [[a]]
wrap as n = transpose $ parse n as

count :: (Ord a, Fractional b) => [a] -> b
count str = let charCounts = snd <$> toList (aggregate str)
            l = length str
denominator = fromIntegral $ l * (l - 1)
numerator = fromIntegral $ sum $ fmap (\c -> c * (c-1)) charCounts
            in numerator / denominator
parallelCount :: (Ord a, Fractional b) => [a] -> b
parallelCount str = let charCounts = snd <$> toList (aggregate str)
                    l = length str
                    denominator = fromIntegral $ l * (l - 1)
                    numerator = fromIntegral $ sum $ parMap rdeepseq \c -> c * (c-1) charCounts
                in numerator / denominator

rate :: (Ord a, Fractional b) => [[a]] -> b
rate d = avg (fmap count d) - fromIntegral (length d) / 3000.0

parallelRate :: (Ord a, Fractional b) => [[a]] -> b
parallelRate d = avg (fmap parallelCount d) - fromIntegral (length d) / 3000.0

sumMult :: Num a => [a] -> [a] -> a
sumMult v0 v1 = sum $ zipWith (*) v0 v1

swap :: Num a => [a] -> [a] -> Char -> a
swap v0 v1 letter = sumMult v0 (drop (ord letter - ord 'A') (cycle v1))

deriveCharacter :: RealFrac a => [a] -> String -> Char
deriveCharacter possible sample = let charCounts = aggregate sample
                                       samp c = findWithDefault 0 c charCounts
                                       real = fmap (fromIntegral . samp) ['A'..'Z']
                                  in maximumBy (comparing $ swap possible real) ['A'..'Z']
all: vigenere

vigenere:
  stack build

run: vigenere
  stack exec vigenere-exe

test: vigenere
  stack exec -- vigenere-exe -t

threadscope: vigenere
  stack exec -- vigenere-exe -t +RTS -N1 -ls
  threadscope vigenere-exe.eventlog

threadscope2: vigenere
  stack exec -- vigenere-exe -t +RTS -N2 -ls
  threadscope vigenere-exe.eventlog

threadscope3: vigenere
  stack exec -- vigenere-exe -t +RTS -N3 -ls
  threadscope vigenere-exe.eventlog

threadscope4: vigenere
  stack exec -- vigenere-exe -t +RTS -N4 -ls
  threadscope vigenere-exe.eventlog

threadscope6: vigenere
  stack exec -- vigenere-exe -t +RTS -N6 -ls
  threadscope vigenere-exe.eventlog

threadscope8: vigenere
  stack exec -- vigenere-exe -t +RTS -N8 -ls
  threadscope vigenere-exe.eventlog

clean:
  stack clean