# Parallel Recovery of Enigma Rotor Settings

Samuel Meshoyrer Evan Mesterhazy

December 2020

## 1 Introduction

The Enigma cipher employed by the German army during WWII has fascinated amateur cryptographers for decades. In this paper we develop a Haskell implementation of the Enigma machine and use it to parallelize a cipher-text only statistical attack that recovers the rotor settings used to encrypt a message. Haskell's lazy evaluation, immutable data structures, and lightweight parallelism facilities make it an excellent medium to explore parallel work allocation strategies in this context. We develop seven parallelization strategies and identify two optimal approaches that achieve 82% of the ideal speedup expected by Amdhal's law on up to 6 Haskell execution contexts (HECs) with different trade-offs between memory usage and scalability as additional HECs are added.

## 2 The Enigma Machine

### 2.1 Bare Metal

The keyspace of the Enigma machine derives from settings selected for its three main components: the plugboard, the rotors, and the reflector. The machine also includes a keyboard for input, a lightboard that illuminates the enciphered output character on each keypress, and a static rotor that passes current between the plugboard and the first rotor.

When a key is pressed, current flows from the keyboard through the plugboard. Current leaving the plugboard passes through the static rotor and then through the rotors from right to left. After the last rotor, the reflector returns the current through the rotors from left to right and back through the static rotor. After leaving the static rotor, the current passes through the plugboard once more, illuminating the enciphered letter on the lightboard as shown in Figure 1.

The rotor positions and settings comprise a significant portion of the Enigma's keyspace since the encryption performed depends on the wiring of each rotor and its orientation. The rotor orientation is referred to as the rotor's "key setting" and is denoted by the top letter visible on the rotor before a key is pressed. Each rotor has 26 possible key settings labeled A - Z. Before enciphering each letter, Enigma steps the rightmost rotor by one

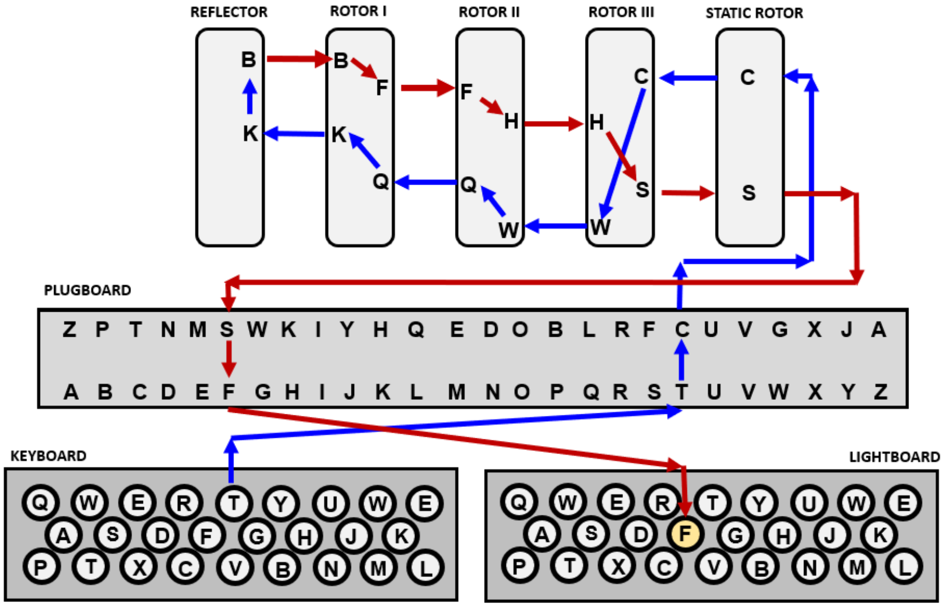

**Figure 1:** The path of current through the Enigma Machine [7]

position. The middle and leftmost rotors may also be stepped depending on the positions of the turnover notch on the rotor bodies.

Although our work does not attempt to recover the rotor ring settings or plugboard configuration, our Enigma model implements both features so we will address them briefly. In addition to the turnover notches, each rotor has a "ring setting" that rotates its internal wiring relative to the key settings marked on the outside of the rotor. Since the rotor stepping behavior is dictated by the turnovers, adjusting the ring setting changes the position of the wiring relative to the turnover notches. The plugboard allows the operator to swap inputs and outputs for a pair of letters. For example, with the plugboard setting ['K', 'T'], a key press of 'K' generates an input current at the "T" position of the static rotor and vice versa. Outputs to the lightboard are similarly swapped.

## 2.2 Haskell Implementation

Our Enigma implementation closely resembles the structure of the physical machine. The machine configuration is represented by an `EnigmaConfig` record, which encapsulates the reflector, rotor, and plugboard settings.

```haskell
data EnigmaConfig = EnigmaConfig
  { reflector :: Wiring,
    rotors :: [OrientedRotor],
    plugboard :: Plugboard
  }
  deriving (Show, Eq)
```

```
data OrientedRotor = OrientedRotor
  { rotor :: Rotor,
    topLetter :: Char,
    ringSetting :: Char
  }
  deriving (Show, Eq)

data Rotor = Rotor
  { rotId :: String,
    wiring :: Wiring,
    invWiring :: Wiring,
    turnovers :: [Char]
  }
  deriving (Show, Eq)

type Wiring = Array Int Char
type Plugboard = [(Char, Char)]
```

Rotor wirings are represented as a Haskell array of characters mapping a right-left rotor input to an output character with $\mathcal{O}(1)$ indexing. For example, the wiring of rotor "I" is represented by the array `"EKMFLGDQVZNTOWYHXUSPAIBRCJ"`. A right hand input at contact 'A' on the rotor maps to the output letter 'E' as shown in Figure 2. An "inverted" wiring mapping from left to right, invWiring, is also stored for each rotor to allow $\mathcal{O}(1)$ mapping of rotor inputs to outputs in both directions.

```
Rotor output character   ->  EKMFLGDQVZNTOWYHXUSPAIBRCJ
Rotor input position     ->  ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

**Figure 2:** Right-left wiring for rotor I

Since the Enigma cipher is reversible, both encryption and decryption use the same cipher function. A message encrypted with a given machine configuration can be decrypted by passing the ciphertext through the machine again with the same initial configuration.

```
cipher :: EnigmaConfig -> String -> String
cipher _ [] = []
cipher cfg [last] = [cipherChar (step cfg) last]
cipher cfg (hd : tl) =
  let stepped = step cfg
   in cipherChar stepped hd : cipher stepped tl
```

Matching the behavior of the mechanical Enigma, the rotors are stepped before each character is enciphered. We omit the details of the stepping function for brevity, but the algorithm is available in the code listing accompanying this paper. The cipherChar function implements character encryption by passing the input character and the corresponding outputs through each component of the Enigma machine.

```
cipherChar :: EnigmaConfig -> Char -> Char
cipherChar config c =
  let plugOut = index $ mapPlug plugs c
      reflectorIn = mapRotorsRightLeft rots plugOut
      reflectorOut = mapReflector (reflector config) reflectorIn
      plugIn = revIndex $ mapRotorsLeftRight rots reflectorOut
   in mapPlug plugs plugIn
  where
    plugs = plugboard config
    rots = rotors config
```

# 3 Decrypting Messages

To simplify the implementation we limit our attack to messages encrypted with an empty plugboard and all ring settings set to “A”. This constrains the keyspace to the selection and order of the rotors along with their key settings and is the first step in an extended method described by James Gillogy [4] that recovers the ring settings and plugboard configuration as well.

The M3 Enigma machine used by the German army during WWII employed three rotors chosen by operators from a set of five possible rotors for a total of 60 possible rotor permutations. Each rotor has 26 possible key settings labeled A - Z, yielding $26 \cdot 26 \cdot 26 = 17,576$ possible starting positions for each rotor permutation. Thus, stepping through each possible initial rotor configuration requires $60 \cdot 17,576 = 1,054,560$ decryptions.

## 3.1 Generating Rotor Permutations

We leverage Haskell’s lazy evaluation to generate a list of all possible initial rotor configurations by constructing the 60 rotor permutations and initializing their key settings with a list comprehension written in do notation.

```
m3RotorPermutations :: [[Rotor]]
m3RotorPermutations = concatMap permutations $ combinations 3 m3RotorSet

allPermutations :: [[OrientedRotor]]
allPermutations = do
  rots <- m3RotorPermutations
  lt <- ['A' .. 'Z']
  mt <- ['A' .. 'Z']
  rt <- ['A' .. 'Z']
  return $ zipWith3 OrientedRotor rots [lt, mt, rt] (repeat 'A')
```

Even though there are over one million starting positions, this representation is efficient since configurations are generated on demand. However, this approach results in high maximum memory utilization for parallelization strategies that are strict in the spine of the list, as we discuss in Section 4.6.

## 3.2 Index of Coincidence

To recover the message, we decrypt the ciphertext with each possible rotor configuration. Each candidate plaintext is scored by calculating its Incidence of Coincidence (IC) [3] and comparing it to the IC of a reference corpus. The IC represents the likelihood of randomly selecting two identical letters from the text and is given by:

$$\text{IC} = \frac{\sum_{i=A}^{Z} f_i(f_i - 1)}{N(N-1)}$$

Where $f_i$ is the number of occurrences of letter $i$ in the sample text, and $N$ is the total number of letters in the text.

As a reference, we calculated the IC of a ~900,000 word corpus of articles from *The Economist*, which produced an IC of 0.0651. During decryption, the candidate plaintext whose IC is closest to the reference IC is selected as the most likely decryption and output to the user. Since this technique is a statistical attack, the likelihood of success decreases as the message length decreases.

# 4 Parallelization

Decrypting Enigma messages presented the opportunity to analyze a special case of distributing parallel work - one where each piece of work is the same size. That is, as opposed to the sudoku solver discussed by Marlow [5], there are no "easy" or "hard" decryptions. Each decryption runs the same string through the same cipher function, just with different rotor configurations. We choose to implement a number of parallel decryption strategies in hopes of building a strong comparison and gaining an understanding of the strengths and weaknesses of the different strategies.

## 4.1 Methodology and Metrics

We tested all strategies with a 50 character sample message. This message was decrypted 5 times per method and the results for each method were averaged together. The primary metrics examined are total wall time and peak memory allocation. The data-points were gathered from the output of running the program with RTS options. There is a convenient "machine-readable" flag which outputs the relevant fields as an easily parsed list of tuples.

Several Python scripts were written to assist in data collection and analysis. Except when comparing parBuffer and parChunks with a varying number of cores, each run was given 6 cores. The decryptions were run on a Debian 10, Linux 4.19.0-11-amd64, guest virtual machine through Oracle VirtualBox on a Windows 10 host. The host machine has an Intel Core i7-8700K CPU 4.20 GHz processor and the guest VM was given direct access to 6 of the processor's 12 logical cores[1]. Both the guest and the host were lightly loaded when performing the tests.

---

[1] 12 cores were made available for tests utilizing more than 6 HECs

|  | Sparks Converted | Total Sparks | % Unconverted | Total Wall Time (sec) | Peak Memory Allocated (MB) |
|---|---:|---:|---:|---:|---:|
| sequential | - | - | - | 116 | 8 |
| rdeepseq | 10,876 | 1,054,560 | 99.0% | 124 | 423 |
| rseq | 1,017,917 | 1,054,560 | 3.5% | 125 | 608 |
| rpar | 1,822,924 | 2,109,120 | 13.5% | 124 | 596 |

**Table 1:** Sequential and parList results averaged over five runs

## 4.2 Sequential Baseline

On our test machine, the sequential implementation completed in approximately 116 seconds but allocated only 8 MB of peak memory. The sequential implementation decrypted each rotor configuration one at a time, using a single HEC, equivalent to one CPU core.

## 4.3 Strategies

For distributing the decryption work, we looked at seven different parallel Strategies [1]: parList using rdeepseq, rpar, and rseq; parBuffer with various sized buffers; a hand-crafted six-way static partition; chunking, first with parListChunk, second with distributing chunks of 1000 decodings to parBuffers of size 128. A summary of the results can be seen in the figures below. It is worth noting that none of the strategies imposed significant development costs. Haskell's clean and easy-to-use parallelization library made comparing a large number of strategies feasible within a short period of time.

## 4.4 First Steps: parList using rdeepseq, rpar, and rseq

All three of the naïve parList evaluation strategies, rdeepseq, rpar, and rseq, performed worse than the sequential implementation. Not only did they take longer in terms of wall time, but they also had significantly higher peak memory allocations. Table 1 demonstrates that each of these strategies, especially rdeepseq, generated a large number of unconverted sparks. The poor distribution of the sparks, shown in Figure 3 degraded the performance even further.

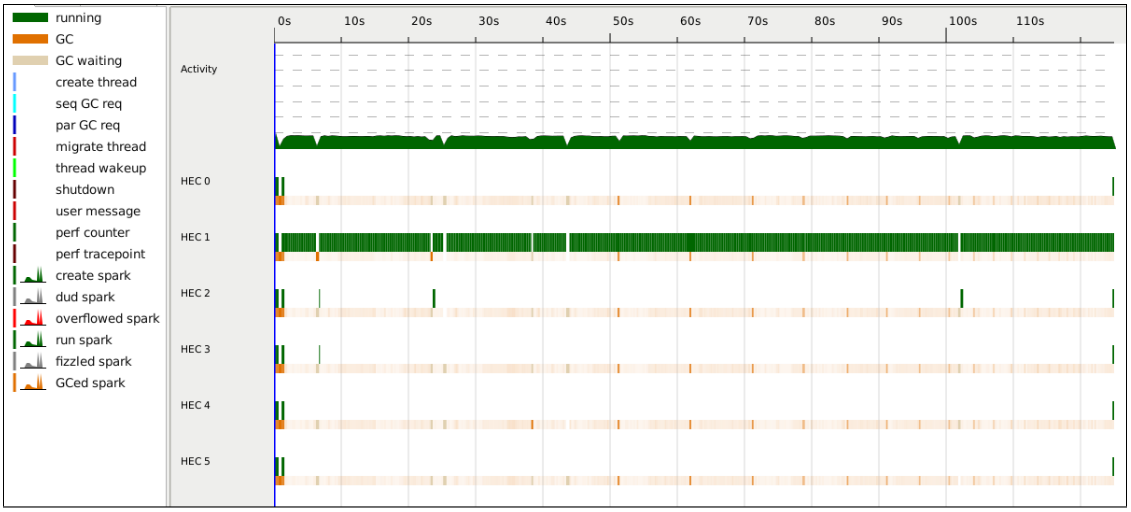

**Figure 3:** Threadscope result for parList rpar

## 4.5 Controlling Spark Generation with parBuffer

Parallelization with parBuffer turned out to be the most efficient strategy overall, achieving a near optimal spot in the bottom-left of the wall time vs. peak memory graph in Figure 4 when executing with 6 HECs. Using parBuffer with a buffer size of 1024 ("parBuffer 1024") hit a sweet spot between execution speed at 21 seconds and peak memory consumption at only 8 MB.

We decided to examine how different buffer sizes affected the performance of the parBuffer strategy and tested several different buffer sizes: 4, 16, 64, 256, 1024, 4096, and 16,348. We observed a moderate speedup as the buffer size increased to 1024. With smaller sizes the buffer capacity is insufficient to fully leverage the parallel capabilities of the cores. The strategy began to degrade with buffers larger than 1024, which we believe is due to the garbage collection (GC) overhead necessary to manage larger spark buffers. Peak memory usage remained stable regardless of the buffer size.

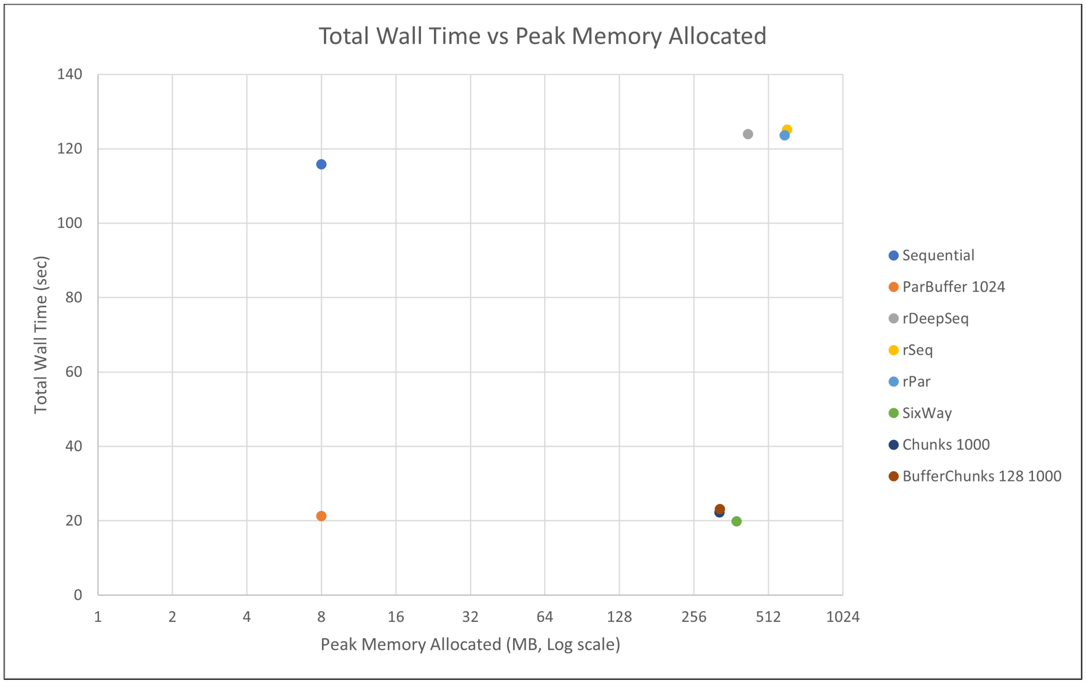

**Figure 4:** Wall Time vs. Peak Memory Allocated, 6 HECs

## 4.6 Six-Way Static Partitioning

Six-Way is our hand-crafted, static partition strategy, which works by breaking the rotor settings into six roughly equal length chunks and handing off each chunk to its own HEC. Breaking the code into chunks is accomplished via the `chunksOf` function from the split package [2] and is roughly equivalent to the code shown below.

```
let [as, bs, cs, ds, es, fs] =
  chunksOf ((length allPermutations `div` 6) + 1) allPermutations
```

Since each chunk is immediately evaluated in parallel using rpar, building the chunks in Six-Way is strict in the spine of the list. This results in significantly higher peak memory usage than observed with parBuffer as shown in Figure 5. Despite this, Six-Way achieved a marginal speed-up over parBuffer 1024 as shown later in Figure 6.

Ultimately this strategy scales poorly: the code must be written with direct knowledge of the hardware it runs on. On top of this, the static partitioning in Six-Way results in poor load-balancing. In each test we ran, HECs finished their work early and sat idle while others finished processing their queues. A more optimal strategy would load-balance between HECs so that all remain fully utilized.

Despite these drawbacks, Six-Way achieved the fastest runtime speed, clocking in at 20 seconds, which is marginally faster than parBuffer 1024. However, we do not consiger Six-Way more efficient because this slight increase in speed came at the cost of much higher peak memory usage: 380 MB vs 8 MB for parBuffer 1024, a nearly 48x increase.

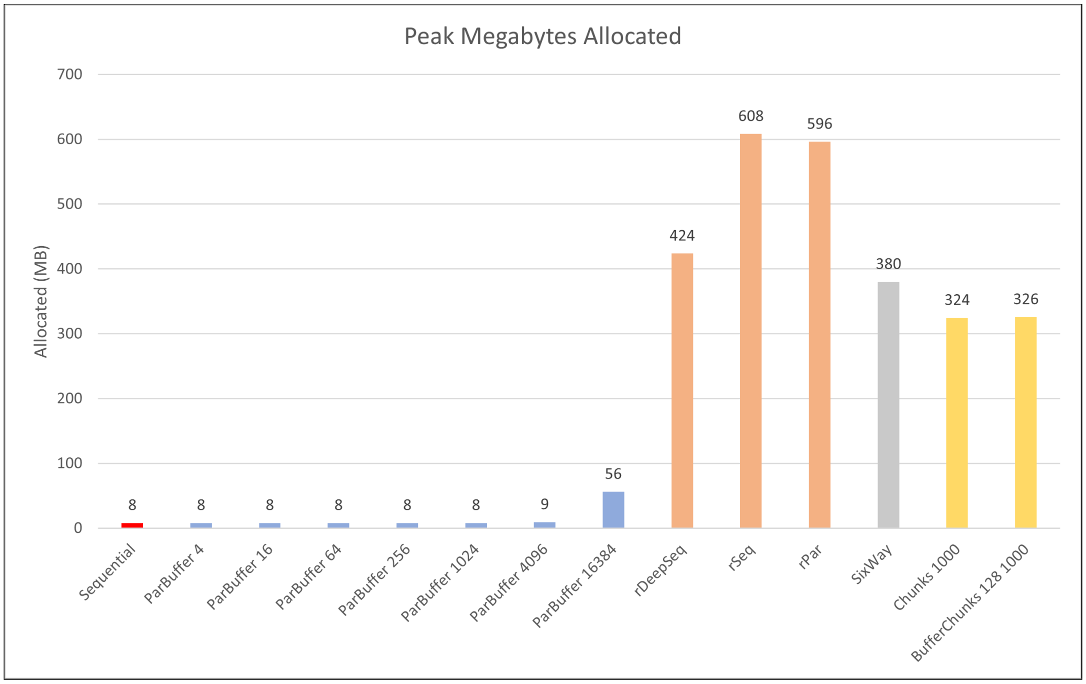

**Figure 5:** Peak megabytes allocated by strategy

## 4.7 Improved Chunking with parListChunk

Although static partitioning scales poorly, chunking the workload reduces the number of sparks generated, increasing throughput by reducing the number of GC pauses. By creating smaller chunks and scheduling them automatically on all available HECs via the parListChunk strategy we can leverage the speedup of chunking while avoiding the load balancing and scalability issues of static partitioning.

To explore this approach we tested two chunking strategies: parListChunk and parBuffer with separate chunking. In our implementation, the parListChunk strategy (parchunks below) begins by splitting the rotor permutations into approximately 1,000 chunks of size 1,000 before applying evalList rdeepseq to each chunk in parallel.

```
parChunks :: [[OrientedRotor]] -> [Char] -> [(Double, String)]
parChunks rots ctext =
  map (solveIC ctext) rots `using` parListChunk 1000 rdeepseq
```

The parBuffer-with-chunks, or “bufferChunks”, strategy applies the same parBuffer strategy discussed in Section 4.5, but increases the size of each unit of work from a single decryption to 1,000 decryptions per spark.

```
bufferChunks :: [Char] -> [(Double, String)]
bufferChunks ctext = concat $
  withStrategy (parBuffer 128 rdeepseq) (map (map (solveIC ctext)) chunks)
  where chunks = chunksOf 1000 allPermutations
```

Both strategies are among the fastest tested at the cost of high peak memory usage (see Figure 5) since they remain strict in the spine of the rotor configuration list. Despite the theoretical benefit of controlling spark creation in the bufferChunks strategy, we did not observe a material difference in running time compared to parChunks. This is likely due to the relatively minor difference of creating sparks 128 at a time instead of creating all 1,000 up front. In the general case with workloads of several million discrete units we expect the bufferChunks strategy to outperform parChunks.

## 5 Discussion

A comparison of total wall time for each strategy when executing with 6 HECs is shown in Figure 6. The parBuffer strategy achieved the best performance considering its speed and low memory overhead. Only the statically partitioned Six-Way strategy ran faster, but at the cost of significantly higher peak memory consumption as discussed previously in Section 4.6. Both chunking strategies, parChunks and parBuffer with chunking, ran nearly as fast as Six-Way and exhibited similar peak memory consumption as previously shown in Figure 5.

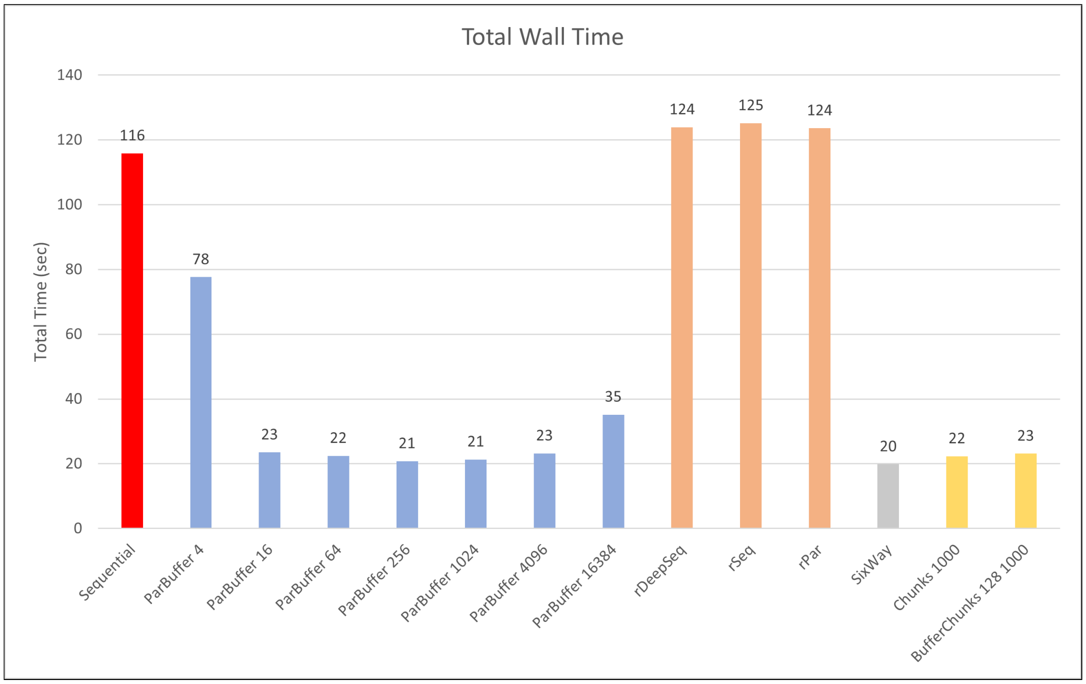

**Figure 6:** Total wall time by strategy, 6 HECs

We include the threadscope outputs for parBuffer and parChunks below in Figure 7 and Figure 8 to demonstrate that both strategies display excellent load balancing across HECs. For parChunks, the threadscope graph also visualizes the high initial GC activity as the spine of the rotor configuration list is evaluated and thunks are garbage collected.

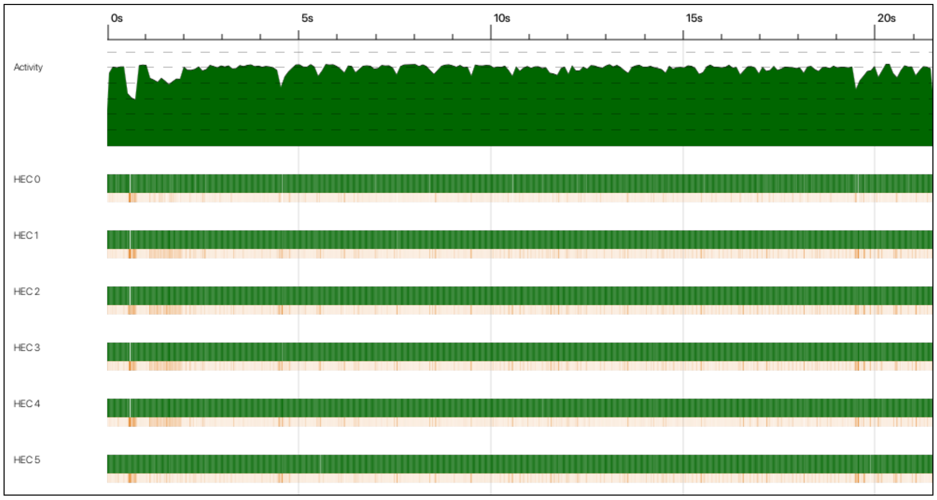

**Figure 7:** Threadscope result for parBuffer

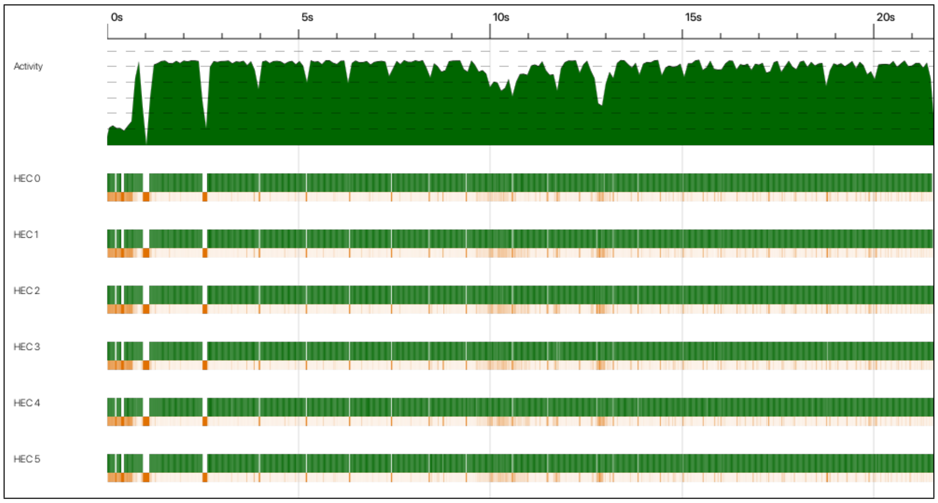

**Figure 8:** Threadscope result for parChunks

## 5.1 Multi-Core Scalability and Garbage Collection

Based on the results we observed, scalability as additional cores are added is highly dependent on the amount of memory allocated by the individual strategy and its interaction with Haskell's garbage collector. As show in in Figure 9, both parBuffer and parChunks exhibit close-to-ideal speedups as the number of HECs increases from 1 to 6. Since each decryption in our workload is independent, the observed speedup should scale nearly linearly with the number of HECs available. However, the actual speedup drops rapidly for parBuffer with the addition of a 7th core/HEC and the speedup curve of parChunks begins to flatten quickly after 8 HECs.

Although Haskell's garbage collector is *parallel*, it is not *concurrent*. This means that although the Haskell runtime can leverage multiple cores to speed-up collection,

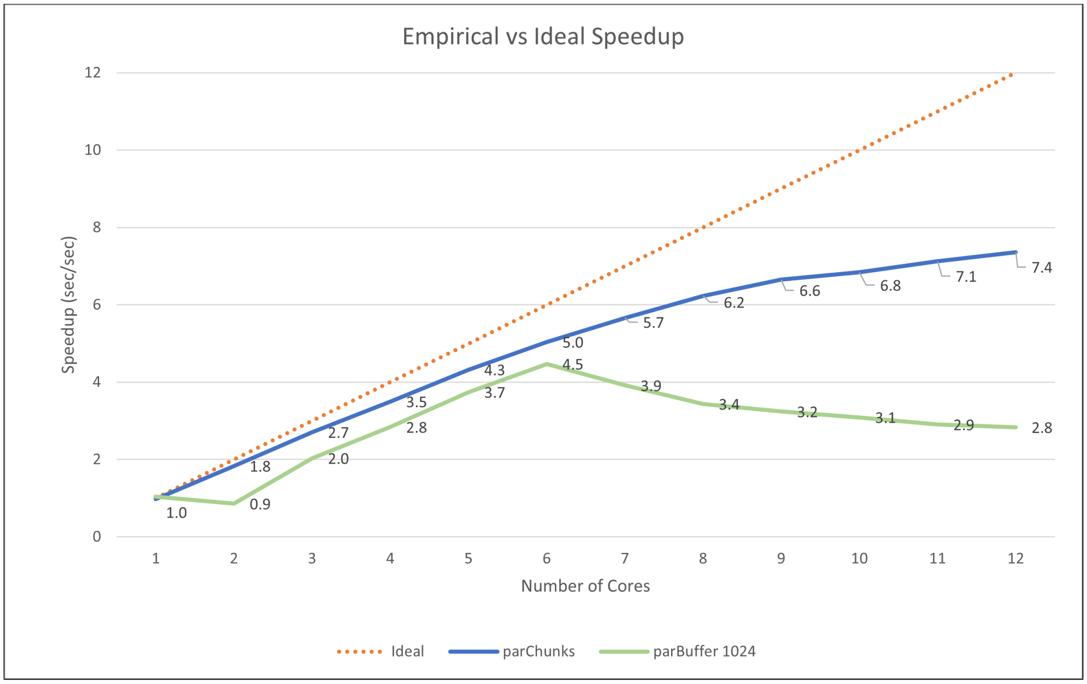

**Figure 9:** Empirical vs ideal speedup

mutators cannot run at the same time as the collector [6]. As a result, decryption across all threads is paused whenever garbage collection occurs. We believe the divergence shown in Figure 9 at higher core counts is due to this stop-the-world garbage collection and the poor parallelization of garbage collection as the number of cores increases as shown in Table 2.

| | 6 HECs | 12 HECs |
|---|---|---|
| parBuffer | 37.0% | 13.7% |
| parChunks | 80.0% | 65.6% |

**Table 2:** GC parallel work balance

# 6 Conclusion

Haskell's straightforward parallel facilities make it simple for programmers to parallelize existing algorithms and leverage the ubiquity of multi-core modern hardware. In decrypting Enigma messages we examined an algorithm where each unit of work is independent and requires the same amount of computation. For this "equal-work" use case, strategies like parBuffer and parChunks displayed the best performance by limiting the number of sparks generated and load-balancing them across available HECs.

Although parallelizing algorithms is easy in Haskell, we observed diminishing returns as the number of HECs increased. We largely attribute this to Haskell's stop-the-world garbage collector and believe that clever programmers can achieve additional performance gains through the careful application of strictness, GC tuning, and memory efficient data structures. Programmers requiring absolute speed are probably better served by languages with manual memory management like Rust, C++, and C. For everyone else, Haskell provides compelling parallelism alongside its expressive type system, lazy evaluation, and vibrant library ecosystem.

## 7 Future Work

This project can be expanded down two main avenues: increased functionality of the application, and a deeper analysis of the parallel evaluation strategies. Functional improvements range from simple, e.g. being able to choose the encoding configuration from the command line, to more complex. A fully fleshed out Enigma cracker would recover not only the rotor configurations, but also the plugboard settings. We could also make the command line interface more intuitive and add options for being able to choose parameters for all the parameterized evaluation methods, not only parBuffer.

In terms of parallelization, the first thing we could do is investigate how the performance of our strategies scales to much larger number of cores, e.g. 32 or even 64 cores. We can measure how each parallelization strategy scales when the size of the ciphertext increases. We could also better understand how the performance of a strategy changes with the external configuration, such as GC tuning. This work could lead to a more general theory of optimal parallelization strategies in the "equal-work" special case.

# References

[1] *Control.Parallel.Strategies.* url: `https://hackage.haskell.org/package/parallel-3.2.2.0/docs/Control-Parallel-Strategies.html` (visited on 12/21/2020).

[2] *Data.List.Split.* url: `https://hackage.haskell.org/package/split-0.2.3.4/docs/Data-List-Split.html#v:chunksOf` (visited on 12/21/2020).

[3] W. F. Friedman. *The Index of Coincidence and Its Applications in Cryptography.* Publication No. 22. Geneva IL: Riverbank Publications, 1922.

[4] James J. Gillogly. "CIPHERTEXT-ONLY CRYPTANALYSIS OF ENIGMA". In: *Cryptologia* 19.4 (1995), pp. 405–413. doi: `10.1080/0161-119591884060`. eprint: `https://doi.org/10.1080/0161-119591884060`. url: `https://doi.org/10.1080/0161-119591884060`.

[5] S. Marlow. *Parallel and Concurrent Programming in Haskell.* Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming. O'Reilly, 2013. isbn: 9781449335946.

[6] Simon Marlow et al. "Parallel Generational-Copying Garbage Collection with a Block-Structured Heap". In: *Proceedings of the 7th International Symposium on Memory Management.* ISMM '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 11–20. isbn: 9781605581347. doi: `10.1145/1375634.1375637`. url: `https://doi.org/10.1145/1375634.1375637`.

[7] University of New England. *Codes and Codebreaking - Modern Era.* url: `https://www.une.edu.au/info-for/visitors/museums/museum-of-antiquities/codebreaker-challenge/enigma` (visited on 12/21/2020).

# Source Code

## Main.hs

```
module Main where

import Data.List (permutations)
import Data.List.Split (chunksOf)
import Enigma
    ( EnigmaConfig(..),
      OrientedRotor(OrientedRotor),
      Rotor,
      getWiring,
      calculateIC,
      cipher,
      combinations,
      normalize )
import Enigma.Static (m3RotorSet, makeOriented, refIC)
import Options.Applicative
  ( Parser,
    ParserInfo,
    argument,
    command,
    helper,
    info,
    metavar,
    progDesc,
    str,
    subparser,
    prefs,
    showHelpOnEmpty,
    customExecParser,
  )
import Control.Parallel.Strategies
    ( parBuffer,
      parList,
      parListChunk,
      rdeepseq,
      rpar,
      rseq,
      runEval,
      using,
      withStrategy,
      Eval,
      Strategy )
import Control.DeepSeq ( force )

type FileName = String

data DecryptStrategy = Sequential | ParBuffer Int | RDeepSeq | RSeq | RPar | SixWay |
↪  Chunks | BufferChunks

data Command
  = Encrypt FileName
  | Decrypt DecryptStrategy FileName

makeParBuffer :: String -> Command
makeParBuffer s = Decrypt (ParBuffer n) fn
  where
   [sizeStr, fn] =
     case words s of
      a@[_, _] -> a
      _ -> error "Missing size or file argument"
   n = read sizeStr ::Int

withInfo :: Parser a -> String -> ParserInfo a
withInfo opts desc = info (helper <*> opts) $ progDesc desc

parseEncrypt :: Parser Command
parseEncrypt = Encrypt <$> argument str (metavar "filename")

parseDecryptSequential :: Parser Command
parseDecryptSequential = Decrypt Sequential <$> argument str (metavar "filename")

parseDecryptParBuffer :: Parser Command
parseDecryptParBuffer = makeParBuffer <$> argument str (metavar "bufferSize filename")

parseDecryptRDeepSeq :: Parser Command
parseDecryptRDeepSeq = Decrypt RDeepSeq <$> argument str (metavar "filename")

parseDecryptRSeq :: Parser Command
parseDecryptRSeq = Decrypt RSeq <$> argument str (metavar "filename")

parseDecryptRPar :: Parser Command
parseDecryptRPar = Decrypt RPar <$> argument str (metavar "filename")

parseDecryptSixWay :: Parser Command
parseDecryptSixWay = Decrypt SixWay <$> argument str (metavar "filename")

parseDecryptChunks :: Parser Command
parseDecryptChunks = Decrypt Chunks <$> argument str (metavar "filename")

parseDecryptBufferChunks :: Parser Command
parseDecryptBufferChunks = Decrypt BufferChunks <$> argument str (metavar "filename")

parseCommand :: Parser Command
parseCommand =
  subparser $
    command "encrypt" (parseEncrypt `withInfo` "Encrypt the file")
      <> command "decryptSequential" (parseDecryptSequential `withInfo` "Decrypt the file
       ↪  sequentially")
      <> command "decryptParBuffer" (parseDecryptParBuffer `withInfo` "Decrypt the file
       ↪  with a parBuffer limited to the given size; use quotes: e.g. \"1000
       ↪  /path/to/file\"")
      <> command "decryptRDeepSeq" (parseDecryptRDeepSeq `withInfo` "Decrypt the file
       ↪  with parList rDeepSeq")
      <> command "decryptRSeq" (parseDecryptRSeq `withInfo` "Decrypt the file with
       ↪  parList rSeq")
      <> command "decryptRPar" (parseDecryptRPar `withInfo` "Decrypt the file with
       ↪  parList rPar")
      <> command "decryptSixWay" (parseDecryptSixWay `withInfo` "Decrypt the file with a
       ↪  static six-way partition")
      <> command "decryptChunks" (parseDecryptChunks `withInfo` "Decrypt the file with
       ↪  parListChunk 1000 rdeepseq")
      <> command "decryptBufferChunks" (parseDecryptChunks `withInfo` "Decrypt the file
       ↪  in chunks of 1000 with parBuffer 128")


parseOptions :: Parser Command
parseOptions = parseCommand

main :: IO ()
main = run =<< customExecParser (prefs showHelpOnEmpty) (parseCommand `withInfo` "")

defaultConfig :: EnigmaConfig
defaultConfig =
  EnigmaConfig
    { reflector = Enigma.getWiring "refB",
      rotors =
        [ makeOriented "III" 'K' 'A',
          makeOriented "II" 'D' 'A',
          makeOriented "I" 'O' 'A'
        ],
      plugboard = []
    }

run :: Command -> IO ()
run cmd = case cmd of
  Encrypt fn -> do
    contents <- readFile fn
    let normalized = Enigma.normalize contents
    putStrLn $ Enigma.cipher defaultConfig normalized
  Decrypt strat fn -> do
    contents <- readFile fn
    let ctext = normalize contents
    case strat of
      Sequential -> print $ sequentialDecrypt ctext
      ParBuffer n -> print $ parListDecrypt (parBuffer n rdeepseq) ctext
      RDeepSeq -> print $ parListDecrypt (parList rdeepseq) ctext
      RSeq -> print $ parListDecrypt (parList rseq) ctext
      RPar -> print $ parListDecrypt (parList rpar) ctext
      SixWay -> print $ sixWayDecrypt ctext
      Chunks -> print $ snd . minimum $ parChunks allPermutations ctext
      BufferChunks -> print $ snd . minimum $ bufferChunks ctext

icDistance :: (Foldable t, Ord k) => Double -> t k -> Double
icDistance target plaintext = abs (target - Enigma.calculateIC plaintext)

m3RotorPermutations :: [[Rotor]]
m3RotorPermutations = concatMap permutations $ combinations 3 m3RotorSet

-- Generate all permutations of oriented rotors to test during decryption
allPermutations :: [[OrientedRotor]]
allPermutations = do
  rots <- m3RotorPermutations
  lt <- ['A' .. 'Z']
  mt <- ['A' .. 'Z']
  rt <- ['A' .. 'Z']
  return $ zipWith3 OrientedRotor rots [lt, mt, rt] (repeat 'A')

-- Like solve, but also calculates the IC
solveIC :: [Char] -> [OrientedRotor] -> (Double, String)
solveIC msg rot = (icDistance refIC ptext, ptext)
  where
    cfg = EnigmaConfig {reflector = getWiring "refB", rotors = rot, plugboard = []}
    ptext = cipher cfg msg

parChunks :: [[OrientedRotor]] -> [Char] -> [(Double, String)]
parChunks rots ctext = map (solveIC ctext) rots `using` parListChunk 1000 rdeepseq

bufferChunks :: [Char] -> [(Double, String)]
bufferChunks ctext = concat $ withStrategy (parBuffer 128 rdeepseq) (map (map (solveIC
 ↪  ctext)) chunks)
  where chunks = chunksOf 1000 allPermutations

sequentialDecrypt :: String -> String
sequentialDecrypt msg = snd $
  minimum $ map (\c -> (icDistance refIC c, c)) $ do
    rotorCfg <- allPermutations
    let cfg = EnigmaConfig {reflector = getWiring "refB", rotors = rotorCfg, plugboard =
     ↪  []}
    [cipher cfg msg]

solve :: [[OrientedRotor]] -> String -> [String]
solve rotorCfgs msg = do
  rotor <- rotorCfgs
  let cfg = EnigmaConfig {reflector = getWiring "refB", rotors = rotor, plugboard = []}
  [cipher cfg msg]

sixWayDecrypt :: String -> String
sixWayDecrypt msg = do
  let [as, bs, cs, ds, es, fs] = chunksOf ((length allPermutations `div` 6) + 1)
   ↪  allPermutations
      solutions = runEval $ do
        as' <- rpar (force $ minimum $ map (\c -> (icDistance refIC c, c)) $ solve as
         ↪  msg)
        bs' <- rpar (force $ minimum $ map (\c -> (icDistance refIC c, c)) $ solve bs
         ↪  msg)
        cs' <- rpar (force $ minimum $ map (\c -> (icDistance refIC c, c)) $ solve cs
         ↪  msg)
        ds' <- rpar (force $ minimum $ map (\c -> (icDistance refIC c, c)) $ solve ds
         ↪  msg)
        es' <- rpar (force $ minimum $ map (\c -> (icDistance refIC c, c)) $ solve es
         ↪  msg)
        fs' <- rpar (force $ minimum $ map (\c -> (icDistance refIC c, c)) $ solve fs
         ↪  msg)
        _ <- rseq as'
        _ <- rseq bs'
        _ <- rseq cs'
        _ <- rseq ds'
        _ <- rseq es'
        _ <- rseq fs'
        return [as', bs', cs', ds', es', fs']
  snd $ minimum solutions

parMapStrategy :: Strategy [b] -> (a -> b) -> [a] -> [b]
parMapStrategy strat f xs = map f xs `using` strat

parListDecrypt :: ([String] -> Eval [String]) -> [Char] -> [Char]
parListDecrypt strategy msg =
  snd $ minimum $ map (\c -> (icDistance refIC c, c)) solutions
  where solutions = parMapStrategy strategy (\cfg -> cipher (EnigmaConfig {reflector =
   ↪  getWiring "refB", rotors = cfg, plugboard = []}) msg) allPermutations
```

## Enigma.hs

---

```
module Enigma
  ( Wiring,
    Plugboard,
    Rotor (..),
    OrientedRotor (..),
    EnigmaConfig (..),
    cipher,
    normalize,
    getWiring,
    calculateIC,
    combinations,
    topLetters,
    ringSettings,
    rotorIDs,
  )
where

import Data.Char (isAlpha, toUpper)
import Data.List (foldl', subsequences)
import Enigma.Internal
    ( cipherChar,
      freqs,
      ic,
      step,
      EnigmaConfig(..),
      OrientedRotor(..),
      Plugboard,
      Rotor(..),
      Wiring )
import Enigma.Static (getWiring)
import GHC.Unicode (isAscii)

-- | Extract top letters from given EnigmaConfig
topLetters :: EnigmaConfig -> String
topLetters cfg = [topLetter rtr | rtr <- rotors cfg]
```

```
ringSettings :: EnigmaConfig -> String
ringSettings cfg = [ringSetting rts | rts <- rotors cfg]

rotorIDs :: EnigmaConfig -> [String]
rotorIDs cfg = [rotId $ rotor rts | rts <- rotors cfg]

-- |
--   Encipher a message using cfg, the starting configuration of
--   the Enigma machine
cipher :: EnigmaConfig -> String -> String
cipher _ [] = []
cipher cfg [last] = [cipherChar (step cfg) last]
cipher cfg (hd : tl) =
  let stepped = step cfg
   in cipherChar stepped hd : cipher stepped tl

-- | Uppercase and drop alpha characters from a string
--   Must be run on the input string before calling Enigma.Cipher
normalize :: String -> String
normalize = map toUpper . filter (\c -> isAscii c && isAlpha c)

-- |
--   Calculate the Index of Coincidence (IC) of a sequence of text
--   The IC of a natural language is typically much higher than
--   the IC of a uniformly random string.
calculateIC :: (Foldable t, Ord k) => t k -> Double
calculateIC = ic . freqs

combinations :: Int -> [a] -> [[a]]
combinations k ns = filter ((k ==) . length) $ subsequences ns
```

## Enigma/Internal.hs

---

```
module Enigma.Internal where

import Data.Array (Array)
import qualified Data.Array as A
import Data.Char (chr, ord)
import Data.List (elemIndex, foldl')
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as M

nRotorPos :: Int
nRotorPos = 26

type Wiring = Array Int Char

type Plugboard = [(Char, Char)]

data Rotor = Rotor
  { rotId :: String,
    wiring :: Wiring,
    invWiring :: Wiring,
    turnovers :: [Char]
```

```haskell
  }
  deriving (Show, Eq)

data OrientedRotor = OrientedRotor
  { rotor :: Rotor,
    topLetter :: Char,
    ringSetting :: Char
  }
  deriving (Show, Eq)

data EnigmaConfig = EnigmaConfig
  { reflector :: Wiring,
    rotors :: [OrientedRotor],
    plugboard :: Plugboard
  }
  deriving (Show, Eq)

index :: Char -> Int
index c = if i < 0 || i > 25 then error "invalid index" else i
  where
    i = ord c - ord 'A'

revIndex :: Int -> Char
revIndex i =
  if i < 0 || i > 25 -- 0 is min rotor ix; 25 is max rotor ix
    then error "invalid index"
    else chr $ i + ord 'A'

mapRotor :: Wiring -> Char -> Int -> Int
mapRotor wiring topLetter inputPos =
  (outputContact - offset) `mod` nRotorPos -- lh output position
  where
    offset = index topLetter
    inputContact = (inputPos + offset) `mod` nRotorPos
    outputContact = index $ wiring A.! inputContact

mapReflector :: Wiring -> Int -> Int
mapReflector wiring = mapRotor wiring 'A'

mapPlug :: Plugboard -> Char -> Char
mapPlug [] c = c
mapPlug ((a, b) : tl) c
  | c == a = b
  | c == b = a
  | otherwise = mapPlug tl c

mapRotors :: [OrientedRotor] -> (Rotor -> Wiring) -> (Wiring -> Char -> Int -> Int) ->
 ↪  Int -> Int
mapRotors [] _ _ pos = pos
mapRotors (curr : tl) wGetter mapper pos =
  mapRotors tl wGetter mapper outputPos
  where
    currWiring = wGetter $ rotor curr
    currTopletter = topLetter curr
    currRingSet = ord (ringSetting curr) - ord 'A'
    ringAdjPos = (pos - currRingSet) `mod` nRotorPos
    outputPos = (currRingSet + mapper currWiring currTopletter ringAdjPos) `mod`
     ↪  nRotorPos

mapRotorsRightLeft :: [OrientedRotor] -> Int -> Int
mapRotorsRightLeft rotors =
  mapRotors (reverse rotors) wiring mapRotor

mapRotorsLeftRight :: [OrientedRotor] -> Int -> Int
mapRotorsLeftRight rotors =
  mapRotors rotors invWiring mapRotor

-- Encipher a single character
cipherChar :: EnigmaConfig -> Char -> Char
cipherChar config c =
  let plugOut = index $ mapPlug plugs c
      reflectorIn = mapRotorsRightLeft rots plugOut
      reflectorOut = mapReflector (reflector config) reflectorIn
      plugIn = revIndex $ mapRotorsLeftRight rots reflectorOut
   in mapPlug plugs plugIn
  where
    plugs = plugboard config
    rots = rotors config

-- Step a single rotor
stepRotor :: OrientedRotor -> OrientedRotor
stepRotor rot =
  rot {topLetter = nextTL}
  where
    tl = topLetter rot
    nextTL = revIndex $ (index tl + 1) `mod` nRotorPos

-- Step all of the rotors in the machine
stepRotors :: [OrientedRotor] -> OrientedRotor -> [OrientedRotor]
stepRotors [] _ = []
stepRotors [curr] prev =
  if topLetter prev `elem` turnovers (rotor prev)
    then [stepRotor curr]
    else [curr]
stepRotors (curr : rest) prev =
  let curr' = if shouldStep then stepRotor curr else curr
   in curr' : stepRotors rest curr
  where
    shouldStep =
      topLetter prev `elem` turnovers (rotor prev)
        || topLetter curr `elem` turnovers (rotor curr)

-- Step the entire Enigma config based on the number of
-- rotors in the machine (3 or 4)
step :: EnigmaConfig -> EnigmaConfig
step cfg =
  case rotors cfg of
    [] -> cfg
    [left, ml, mr, right] ->
      let newRM = stepRotor right
```

```
          newRots = newRM : stepRotors [mr, ml] right
       in cfg {rotors = left : reverse newRots}
    [left, mid, right] ->
      let newRM = stepRotor right
          newRots = newRM : stepRotors [mid, left] right
       in cfg {rotors = reverse newRots}
    _ -> error "stepping behavior undefined for configs with < 3 or > 4 rotors"

freqs :: (Foldable t, Ord k) => t k -> [Int]
freqs = freqList . countItems
  where
    countItems :: (Foldable t, Ord k) => t k -> Map k Int
    countItems = foldl' (\counts item -> M.insertWith (+) item (1 :: Int) counts) M.empty
    freqList :: Map k Int -> [Int]
    freqList = M.foldl' (flip (:)) []

ic :: [Int] -> Double
ic freqs = numerator / (s * (s -1))
  where
    s = fromIntegral $ sum freqs
    numerator = fromIntegral $ sum [f * (f -1) | f <- freqs]
```

## Enigma/Static.hs

---

```
{-# LANGUAGE NamedFieldPuns #-}

module Enigma.Static where

import Data.Array (Array, accumArray, elems, listArray)
import Data.Char (chr, ord)
import Data.Maybe (fromJust)
import Enigma.Internal ( OrientedRotor(..), Rotor(..), Wiring )

-- NOTE: The order of this list is important. Do not modify
canonicalRotors :: [(String, (String, [Char]))]
canonicalRotors =
  [ ("I", ("EKMFLGDQVZNTOWYHXUSPAIBRCJ", "Q")),
    ("II", ("AJDKSIRUXBLHWTMCQGZNPYFVOE", "E")),
    ("III", ("BDFHJLCPRTXVZNYEIWGAKMUSQO", "V")),
    ("IV", ("ESOVPZJAYQUIRHXLNFTGKDCMWB", "J")),
    ("V", ("VZBRGITYUPSDNHLXAWMJQOFECK", "Z")),
    ("VI", ("JPGVOUMFYQBENHZRDKASXLICTW", "ZM")),
    ("VII", ("NZJHGRCXMYSWBOUFAIVLPEKQDT", "ZM")),
    ("VIII", ("FKQHTLXOCBJSPDZRAMEWNIUYGV", "ZM")),
    ("id", ("ABCDEFGHIJKLMNOPQRSTUVWXYZ", "")),
    ("refB", ("YRUHQSLDPXNGOKMIEBFZCWVJAT", "")),
    ("refC", ("FVPJIAOYEDRZXWGCTKUQSBNMHL", "")),
    ("beta", ("LEYJVCNIXWPBQMDRTAKZGFUHOS", "")),
    ("gamma", ("FSOKANUERHMBTIYCWLQPZXVGJD", ""))
  ]

invertWiring :: Wiring -> Wiring
invertWiring stringWiring =
  accumArray (\_ a -> a) '-' (0, 25) $ iv [] 0 (elems stringWiring)
```

```haskell
  where
    iv :: [(Int, Char)] -> Int -> String -> [(Int, Char)]
    iv accum _ [] = accum
    iv accum ix (hd : tl) =
      iv ((ord hd - ord 'A', chr $ ix + ord 'A') : accum) (ix + 1) tl

-- | Returns of list of the five rotors used in the original
-- M3 Enigma Machine
m3RotorSet :: [Rotor]
m3RotorSet = buildRotorSet "VI"

buildRotorSet :: String -> [Rotor]
buildRotorSet stopAt = map (getRotor . fst) $ takeWhile (\(n, _) -> n /= stopAt)
 ↪  canonicalRotors

-- The pre-calculated reference IC of the Economist corpus
refIC :: Double
refIC = 0.0651437

-- | Get a rotor wiring by name i.e. wiring "I"
getWiring :: String -> Wiring
getWiring name = listArray (0, length w - 1) w
  where
    w = fst $ fromJust $ lookup name canonicalRotors

getTurnovers :: String -> [Char]
getTurnovers name = snd $ fromJust $ lookup name canonicalRotors

getRotor :: String -> Rotor
getRotor name = Rotor {rotId = name, wiring = w, invWiring = iw, turnovers = getTurnovers
 ↪  name}
  where
    w = getWiring name
    iw = invertWiring w

makeOriented :: String -> Char -> Char -> OrientedRotor
makeOriented name topLetter ringSetting =
  OrientedRotor {rotor = getRotor name, topLetter, ringSetting}
```

## Spec.hs

---

```haskell
import qualified Data.Set as S
import Enigma
import Enigma.Internal
import EnigmaTestStatic
import Test.Hspec

-- Test helper
stepNTimes :: EnigmaConfig -> Int -> EnigmaConfig
stepNTimes cfg n =
  if n == 0 then cfg else stepNTimes (Enigma.Internal.step cfg) (n - 1)

main :: IO ()
main = hspec $ do
```

```
describe "Enigma.Internal.index" $ do
  it "Gets the 0-25 index of a rotor top letter" $ do
    Enigma.Internal.index 'A' `shouldBe` (0 :: Int)
    Enigma.Internal.index 'Z' `shouldBe` (25 :: Int)

describe "Enigma.Internal.revIndex" $ do
  it "Gets the top letter from an index num in 0-25" $ do
    Enigma.Internal.revIndex 0 `shouldBe` 'A'
    Enigma.Internal.revIndex 25 `shouldBe` 'Z'
    Enigma.Internal.revIndex (Enigma.Internal.index 'A') `shouldBe` 'A'

describe "Enigma.Internal.mapRotor" $ do
  it "Maps a rotor input to an output for a given wiring" $ do
    -- Right to left mappings
    Enigma.Internal.mapRotor rotorIDWiring 'A' 0 `shouldBe` 0
    -- Identity map should work regardless of rotor position
    Enigma.Internal.mapRotor rotorIDWiring 'B' 0 `shouldBe` 0
    -- Test input at the last contact
    Enigma.Internal.mapRotor rotorIWiring 'A' 25 `shouldBe` 9
    -- Output contact - offet is negative and the output pos needs
    -- to "roll" over to a high index
    -- We have offset = 20, rhContact = 22, lhContact = 1, lh_pos = 7
    Enigma.Internal.mapRotor rotorIWiring 'U' 2 `shouldBe` 7
    Enigma.Internal.mapRotor rotorIIIWiring 'Z' 1 `shouldBe` 2
    Enigma.Internal.mapRotor rotorIWiring 'A' 0 `shouldBe` 4
    Enigma.Internal.mapRotor rotorIWiring 'B' 0 `shouldBe` 9

    -- Left to right mappings
    -- Input at last contact
    Enigma.Internal.mapRotor invRotorIWiring 'A' 25 `shouldBe` 9
    -- Offset is negative and rolls over to an index at the start of the rotor
    Enigma.Internal.mapRotor invRotorIWiring 'U' 12 `shouldBe` 11
    -- Rotor top is Z and input is 25 - causing overflow and underflow in
    -- the lhContact and rh output calculation
    Enigma.Internal.mapRotor invRotorIIWiring 'Z' 25 `shouldBe` 22
    Enigma.Internal.mapRotor invRotorIWiring 'A' 0 `shouldBe` 20
    Enigma.Internal.mapRotor invRotorIWiring 'F' 10 `shouldBe` 14

describe "Enigma.Internal.mapReflector" $ do
  it "Maps reflector inputs to outputs" $ do
    Enigma.Internal.mapReflector rotorIDWiring 0 `shouldBe` 0
    -- Check that reflector is an involution
    Enigma.Internal.mapReflector refBWiring 4 `shouldBe` 16
    Enigma.Internal.mapReflector refBWiring 16 `shouldBe` 4

    -- Test the last position in the reflector
    Enigma.Internal.mapReflector refCWiring 11 `shouldBe` 25

    Enigma.Internal.mapReflector refCWiring 25 `shouldBe` 11
    Enigma.Internal.mapReflector refBWiring 0 `shouldBe` 24

describe "Enigma.Internal.mapPlug" $ do
  it "Maps plugboard IO" $ do
    Enigma.Internal.mapPlug plugboardA 'Z' `shouldBe` 'A'
    Enigma.Internal.mapPlug plugboardB 'Z' `shouldBe` 'A'
```

```haskell
    -- Plugboard mapping is an involution
    Enigma.Internal.mapPlug plugboardA (Enigma.Internal.mapPlug plugboardA 'X')
     ↪  `shouldBe` 'X'
    -- Character is missing from plugboard
    Enigma.Internal.mapPlug plugboardA 'Q' `shouldBe` 'Q'
    -- NB: Will need to change this test if representation of plugboard changes
    Enigma.Internal.mapPlug [] 'A' `shouldBe` 'A'

describe "Enigma.Internal.cipherChar" $ do
  it "Enciphers a single character" $ do
    Enigma.Internal.cipherChar configA 'O' `shouldBe` 'D'
    Enigma.Internal.cipherChar configA 'Z' `shouldBe` 'H'
    Enigma.Internal.cipherChar configA 'Q' `shouldBe` 'V'
    Enigma.Internal.cipherChar configA 'V' `shouldBe` 'Q'
    Enigma.Internal.cipherChar configB 'R' `shouldBe` 'V'
    Enigma.Internal.cipherChar configB 'J' `shouldBe` 'Q'
    -- Check the case where signal passes through the plugboard twice
    Enigma.Internal.cipherChar configB 'X' `shouldBe` 'E'

    Enigma.Internal.cipherChar idConfig 'A' `shouldBe` 'A'
    Enigma.Internal.cipherChar configA 'G' `shouldBe` 'P'

describe "Enigma.topLetters" $ do
  it "Extracts top letters from an EnigmaConfig" $ do
    Enigma.topLetters configA `shouldBe` "AAA"
    Enigma.topLetters configB `shouldBe` "ZQB"
    Enigma.topLetters stepCfgA `shouldBe` "KDO"
    Enigma.topLetters (stepNTimes stepCfgA 3) `shouldBe` Enigma.topLetters stepCfgA3

describe "Enigma.Internal.stepRotors" $ do
  it "Steps a set of rotors" $ do
    Enigma.Internal.step stepCfgA `shouldBe` stepCfgA'
    stepNTimes stepCfgA 3 `shouldBe` stepCfgA3
    head (Enigma.topLetters (stepNTimes stepCfgB 30000)) `shouldBe` head
     ↪  (Enigma.topLetters stepCfgB)

describe "Enigma.Internal.cipher" $ do
  it "Enciphers a string" $ do
    Enigma.cipher stepCfgA "OCAML" `shouldBe` "VOMUZ"
    Enigma.cipher stepCfgA "VOMUZ" `shouldBe` "OCAML"
    Enigma.cipher stepCfgA "HASKELL" `shouldBe` "ZLFIHSS"
    Enigma.cipher stepCfgA "ZLFIHSS" `shouldBe` "HASKELL"
    -- Long test cases verified against the Universal Enigma simulator
    -- http://people.physik.hu-berlin.de/~palloks/js/enigma/enigma-u_v25_en.html
    let plain =
     ↪  "OHMAMACANTHISREALLYBETHEENDTOBESTUCKINSIDEOFMOBILEWITHTHEMEMPHISBLUESAGAIN"
        ctext =
         ↪  "VGAHGCRJEZXTNQIXVACNAZMPYBZVJNYLIVAEWVNOMGQCZMQVWDCSYWRONWYEYSCCRFNPLEKILF"
    Enigma.cipher stepCfgA plain `shouldBe` ctext
    Enigma.cipher stepCfgA ctext `shouldBe` plain

    -- Tests incorporating the plugboard
    let plain2 = "COMEYOUMASTERSOFWARYOUTHATBUILDALLTHEGUNSYOUTHATBUILDTHEDEATHPLANES"
        ctext2 = "IHDVEWBUWODYNENLOUEZHHKKOLUBYOOKDFOPPKCWFZRANGWMNQAVLNUGISVXVDIMSPC"
    Enigma.cipher plugCfgA plain2 `shouldBe` ctext2
  describe "EnigmaInternal.freqs" $ do
    it "Returns a list of the number of times an item appears in a foldable" $ do
      S.fromList (Enigma.Internal.freqs "AABC") `shouldBe` S.fromList [2, 1, 1]
      Enigma.Internal.freqs "" `shouldBe` []
      Enigma.Internal.freqs "AAAAA" `shouldBe` [5]

  describe "Enigma.Internal.ic" $ do
    it "Calculates the index of coincidence from a list of frequencies" $ do
      Enigma.Internal.ic [1, 1] `shouldBe` 0
      Enigma.Internal.ic [2, 2] `shouldBe` 1 / 3
      Enigma.Internal.ic [2, 2, 2] `shouldBe` 1 / 5
      Enigma.Internal.ic (Enigma.Internal.freqs "AABBCC") `shouldBe` 1 / 5

  describe "Enigma.Internal.normalize" $ do
    it "Normalizes input text to [A..Z]" $ do
      Enigma.normalize "" `shouldBe` ""
      Enigma.normalize "ABCDE" `shouldBe` "ABCDE"
      Enigma.normalize ['a' .. 'z'] `shouldBe` ['A' .. 'Z']
      Enigma.normalize "é.~!@#$%^&*()" `shouldBe` ""
      Enigma.normalize ['0' .. '9'] `shouldBe` ""
      Enigma.normalize "This is a test string." `shouldBe` "THISISATESTSTRING"
```

## EnigmaTestStatic.hs

---

```
module EnigmaTestStatic where

import Enigma
import Enigma.Static

rotorIDWiring :: Wiring
rotorIDWiring = getWiring "id"

rotorIWiring :: Wiring
rotorIWiring = getWiring "I"

invRotorIWiring :: Wiring
invRotorIWiring = invertWiring rotorIWiring

rotorIIWiring :: Wiring
rotorIIWiring = getWiring "II"

invRotorIIWiring :: Wiring
invRotorIIWiring = invertWiring rotorIIWiring

rotorIIIWiring :: Wiring
rotorIIIWiring = getWiring "III"

rotorIVWiring :: Wiring
rotorIVWiring = getWiring "IV"

refBWiring :: Wiring
refBWiring = getWiring "refB"
```

```
refCWiring :: Wiring
refCWiring = getWiring "refC"

plugboardA :: Plugboard
plugboardA = [('A', 'Z'), ('X', 'Y'), ('O', 'E')]

plugboardB :: Plugboard
plugboardB = [('Z', 'A'), ('Y', 'X')]

rotorI :: Rotor
rotorI = getRotor "I"

rotorII :: Rotor
rotorII = getRotor "II"

rotorIII :: Rotor
rotorIII = getRotor "III"

rotorIV :: Rotor
rotorIV = getRotor "IV"

rotorBeta :: Rotor
rotorBeta = getRotor "beta"

idConfig :: EnigmaConfig
idConfig =
  EnigmaConfig
    { reflector = rotorIDWiring,
      rotors = [],
      plugboard = []
    }

configA :: EnigmaConfig
configA =
  EnigmaConfig
    { reflector = refBWiring,
      rotors =
        [ OrientedRotor {rotor = rotorI, topLetter = 'A', ringSetting = 'A'},
          OrientedRotor {rotor = rotorII, topLetter = 'A', ringSetting = 'A'},
          OrientedRotor {rotor = rotorIII, topLetter = 'A', ringSetting = 'A'}
        ],
      plugboard = []
    }

configB :: EnigmaConfig
configB =
  EnigmaConfig
    { reflector = refBWiring,
      rotors =
        [ OrientedRotor {rotor = rotorIII, topLetter = 'Z', ringSetting = 'A'},
          OrientedRotor {rotor = rotorII, topLetter = 'Q', ringSetting = 'A'},
          OrientedRotor {rotor = rotorI, topLetter = 'B', ringSetting = 'A'}
        ],
      plugboard = plugboardA
    }
```

```haskell
stepCfgA :: EnigmaConfig
stepCfgA =
  EnigmaConfig
    { reflector = refBWiring,
      rotors =
        [ OrientedRotor {rotor = rotorIII, topLetter = 'K', ringSetting = 'A'},
          OrientedRotor {rotor = rotorII, topLetter = 'D', ringSetting = 'A'},
          OrientedRotor {rotor = rotorI, topLetter = 'O', ringSetting = 'A'}
        ],
      plugboard = []
    }

stepCfgA' :: EnigmaConfig
stepCfgA' =
  EnigmaConfig
    { reflector = refBWiring,
      rotors =
        [ OrientedRotor {rotor = rotorIII, topLetter = 'K', ringSetting = 'A'},
          OrientedRotor {rotor = rotorII, topLetter = 'D', ringSetting = 'A'},
          OrientedRotor {rotor = rotorI, topLetter = 'P', ringSetting = 'A'}
        ],
      plugboard = []
    }

-- CfgA stepped 3 times
stepCfgA3 :: EnigmaConfig
stepCfgA3 =
  EnigmaConfig
    { reflector = refBWiring,
      rotors =
        [ OrientedRotor {rotor = rotorIII, topLetter = 'K', ringSetting = 'A'},
          OrientedRotor {rotor = rotorII, topLetter = 'E', ringSetting = 'A'},
          OrientedRotor {rotor = rotorI, topLetter = 'R', ringSetting = 'A'}
        ],
      plugboard = []
    }

stepCfgB :: EnigmaConfig
stepCfgB =
  EnigmaConfig
    { reflector = refBWiring,
      rotors =
        [ OrientedRotor {rotor = rotorIV, topLetter = 'I', ringSetting = 'A'},
          OrientedRotor {rotor = rotorIII, topLetter = 'K', ringSetting = 'A'},
          OrientedRotor {rotor = rotorII, topLetter = 'D', ringSetting = 'A'},
          OrientedRotor {rotor = rotorI, topLetter = 'O', ringSetting = 'A'}
        ],
      plugboard = []
    }

plugCfgA :: EnigmaConfig
plugCfgA =
  EnigmaConfig
    { reflector = refBWiring,
      rotors =
        [ OrientedRotor {rotor = rotorIII, topLetter = 'R', ringSetting = 'A'},
          OrientedRotor {rotor = rotorII, topLetter = 'F', ringSetting = 'A'},
          OrientedRotor {rotor = rotorI, topLetter = 'Z', ringSetting = 'A'}
        ],
      plugboard = plugboardA
    }

ringCfgA :: EnigmaConfig
ringCfgA =
  EnigmaConfig
    { reflector = refBWiring,
      rotors =
        [ makeOriented "II" 'F' 'D',
          makeOriented "I" 'K' 'F',
          makeOriented "III" 'L' 'B'
        ],
      plugboard = []
    }
```

## collect_stats.py

```
#!/usr/bin/env python3

import subprocess
import csv

strategies = ["ParBuffer", "Sequential", "RDeepSeq", "RSeq", "RPar", "SixWay",
 ↪  "Chunks","BufferChunks"]

c_filepath = "../corpus/very-small.txt"

def parse_rts_output(rts_out):
    exec("global out; out = {}".format(rts_out))
    d = dict()
    for k, v in out:
        d[k] = float(v) if '.' in v else int(v)
    return d

def all_decrypts():
    all_decrypts = []
    for strat in strategies:
        for _ in range(5):
            if strat == "ParBuffer":
                for buff_size_exp in range(2, 15, 2):
                    buff_size = 2**buff_size_exp
                    decrypt_param = 'decrypt{} "{} {}"'.format(strat, buff_size,
                     ↪  c_filepath)
                    all_decrypts.append(decrypt_param)
            else:
                decrypt_param = "decrypt{} {}".format(strat, c_filepath)
                all_decrypts.append(decrypt_param)
    return all_decrypts
```

```python
def all_out():
    all_out = []
    for i, decrypt in enumerate(all_decrypts()):
        print("Running {}".format(decrypt), i)
        cmd = "stack exec -- denigma-exe {} +RTS -N6 -t --machine-readable -l
         ↪  -ol{}.eventlog".format(decrypt, i)
        out = subprocess.run(cmd, capture_output=True, text=True, shell=True)
        d = parse_rts_output(out.stderr)
        d["decrypt"] = decrypt
        d["id"] = i
        all_out.append(d)
    return all_out

if __name__ == "__main__":
    all_out = all_out()
    with open("decrypt_data.csv", 'w', newline='') as csvfile:
        fieldnames = [k for k in all_out[0]]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for data in all_out:
            writer.writerow(data)
```

## avg_stats.py

---

```python
#!/usr/bin/env python3

import csv
from collections import defaultdict

if __name__ == "__main__":
    # get all rows
    rows = []
    with open("../data/decrypt_data.csv", newline='') as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            rows.append(row)

    # group by decrypt params
    cols = [k for k in rows[0]][:-2]
    grouped_rows = defaultdict(list)
    for row in rows:
        grouped_rows[row["decrypt"]].append(row)

    # take averages per group
    avgs = []
    for decrypt, group in grouped_rows.items():
        avg = dict()
        for c in cols:
            s = sum(float(run[c]) for run in group)
            avg[c] = s/len(group)
        avg["decrypt"] = decrypt
        avgs.append(avg)

    # persist
    with open('../data/avgs.csv', 'w', newline='') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=[k for k in avgs[0]])
        writer.writeheader()
        for a in avgs:
            writer.writerow(a)
```

## stack.yaml

```
resolver: lts-16.24

packages:
- .
```

## package.yaml

```
name:                denigma
version:             0.1.0.0
github:              "smeshoyrer/pfp-project"
license:             BSD3
author:              "Evan Mesterhazy & Samuel Meshoyrer"
maintainer:          "etm2131@columbia.edu"
copyright:           "2020 Evan Mesterhazy & Samuel Meshoyrer"

extra-source-files:
- README.md
- ChangeLog.md

description:         Please see the README

dependencies:
- base >= 4.7 && < 5
- hspec
- containers
- array
- optparse-applicative
- parallel
- deepseq
- split

library:
  source-dirs: src

executables:
  denigma-exe:
    main:                Main.hs
    source-dirs:         app
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N1
    - -eventlog
    - -O2
    - -Wall
    dependencies:
    - denigma

tests:
  denigma-test:
    main:                Spec.hs
    source-dirs:         test
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    dependencies:
    - denigma
    - hspec
```