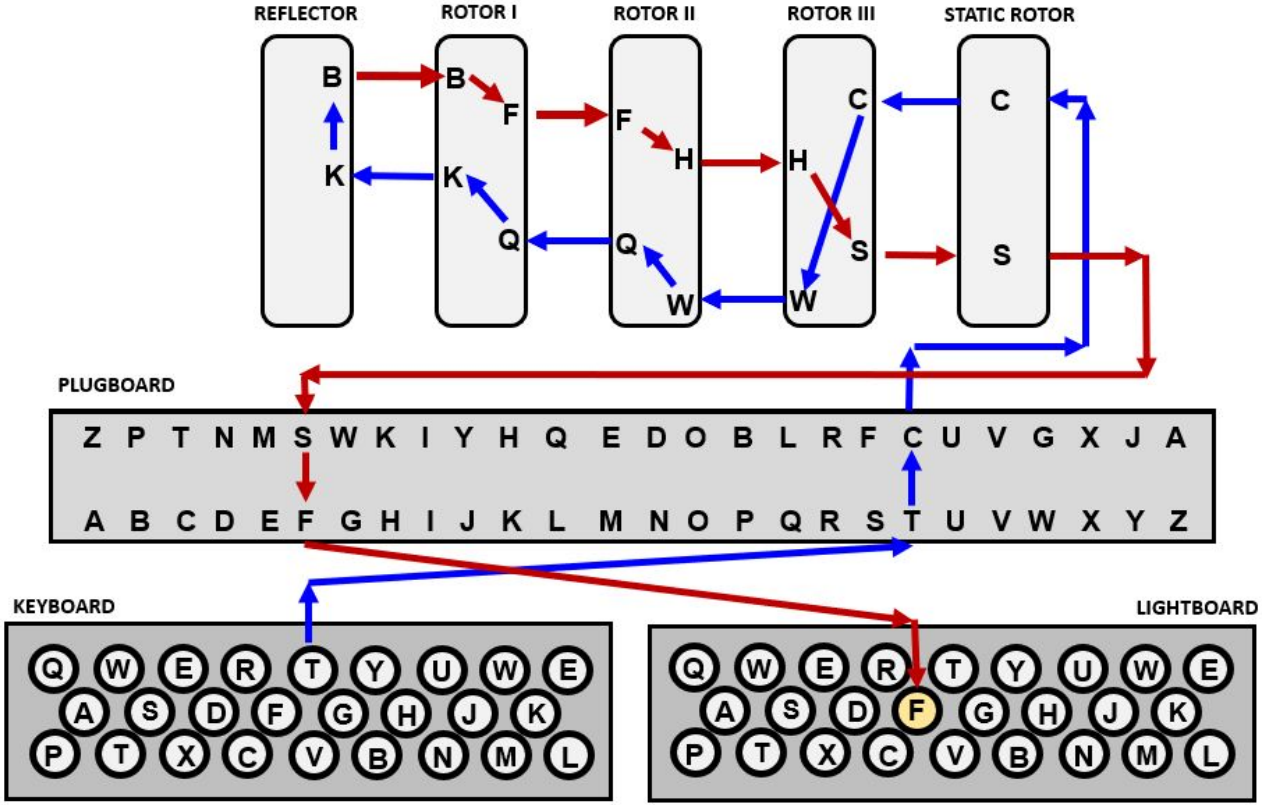
Parallel Enigma Cracker

Samuel Meshoyrer and Evan Mesterhazy

Enigma Machine



The path of current through the Enigma Machine [1]

Haskell Implementation

Data structures model the structure of the physical Enigma machine

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


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

Haskell Implementation

Rotor wiring is represented with an immutable Array for O(1) indexing

```
type Wiring = Array Int Char

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

Example:
Wiring for rotor “I” is represented by the array “EKMFLGDQVZNTOWYHXUSPAIBRCJ”

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

“Inverted” wiring stored for O(1) lookups in both directions

Keyspace

The Enigma keyspace is comprised of several settings:

- The plugboard
- The rotors
  - IDs and relative order
  - Key settings
  - Ring settings
- Reflector choice

Keyspace

The Enigma keyspace is comprised of several settings:

- The plugboard
- The rotors
  - IDs and relative order
  - Key settings
  - Ring settings
- Reflector choice

We focus on recovering the rotor order and key settings, which is the first step to recovering the full configuration.

Keyspace

The M3 Enigma used three rotors chosen from a set of five:

```
Rotor I    "EKMFLGDQVZNTOWYHXUSPAIBRCJ"
Rotor II   "AJDKSIRUXBLHWTMCQGZNPYFVOE"
Rotor III  "BDFHJLCPRTXVZNYEIWGAKMUSQO"
Rotor IV   "ESOVPZJAYQUIRHXLNFTGKDCMWB"
Rotor V    "VZBRGITYUPSDNHLXAWMJQOFECK"
```

5 nCr 3 = 10 possible combinations of rotors

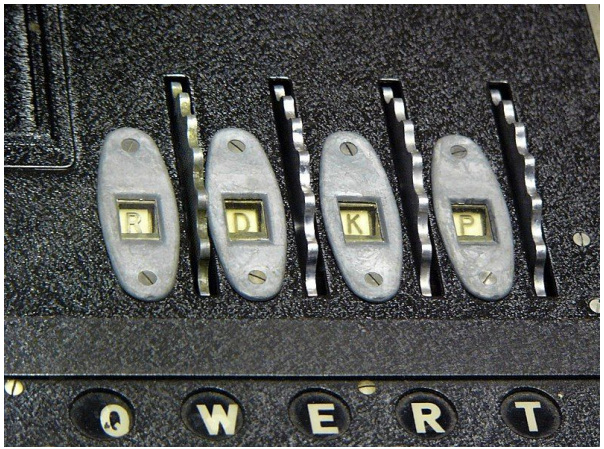
3 nPr 3 = 6 possible permutations for each rotor set

$10 \cdot 6 = 60$ possible rotor permutations

Keyspace

Each rotor has an initial key setting denoted by the top character visible on the rotor.

- 26 possible key settings, one for each letter A - Z
- 3 rotors: 26 · 26 · 26 = 17,576 possible initial key settings
- 60 possible rotor permutations
- 60 · 17,576 = 1,054,560 initial rotor configurations



Key settings visible on a 4-rotor Enigma [2]

Decrypting Messages

We leverage Haskell's lazy evaluation to generate a list of all possible rotor configurations

```
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')
```

Highly efficient for sequential strategies, but presents challenges for partitioning

Decrypting Messages

We decrypt the ciphertext with each possible rotor configuration

Each candidate plaintext is scored by calculating its Index of Coincidence (IC) [3], which represents the likelihood of randomly selecting two identical letters from the text:

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

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

The IC of each candidate is compared to the IC of a ~900,000 word corpus of articles from *The Economist*, which produced an IC of 0.0651

The candidate with the closest IC is selected as the best decryption.

Parallelization

Sequential Solving

```
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 $ minimum $ parChunks allPermutations ctext
      BufferChunks -> print $ minimum $ bufferChunks ctext
```

```
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]
```

Choosing the Parallel Strategy

```
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 $ minimum $ parChunks allPermutations ctext
      BufferChunks -> print $ minimum $ bufferChunks ctext
```

```
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
```

Static Partitioning

```
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 $ minimum $ parChunks allPermutations ctext
      BufferChunks -> print $ minimum $ bufferChunks ctext
```

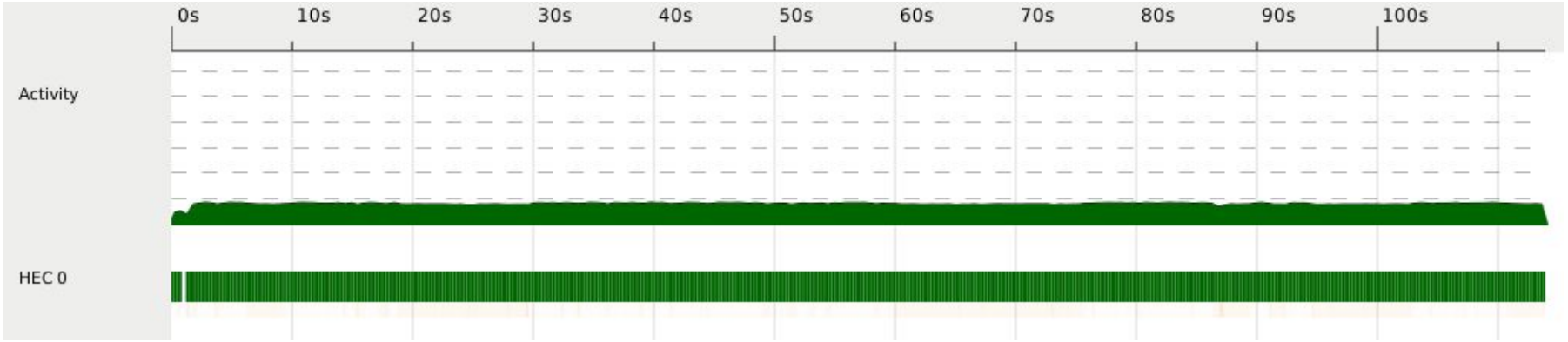
```
sixWayDecrypt :: String -> String
sixWayDecrypt msg = do
  let [as, bs, cs, ds, es, fs] = chunker $ chunksOf (length allPermutations `div` 6) allPermutations
      solutions = runEval $ do
      as' <- rpar (force $ minimum $ map (\c -> (icDistance refIC c, c)) $ solve as msg)
      ...
      fs'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

Chunking

```
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 $ minimum $ parChunks allPermutations ctext
      BufferChunks -> print $ minimum $ bufferChunks ctext
```

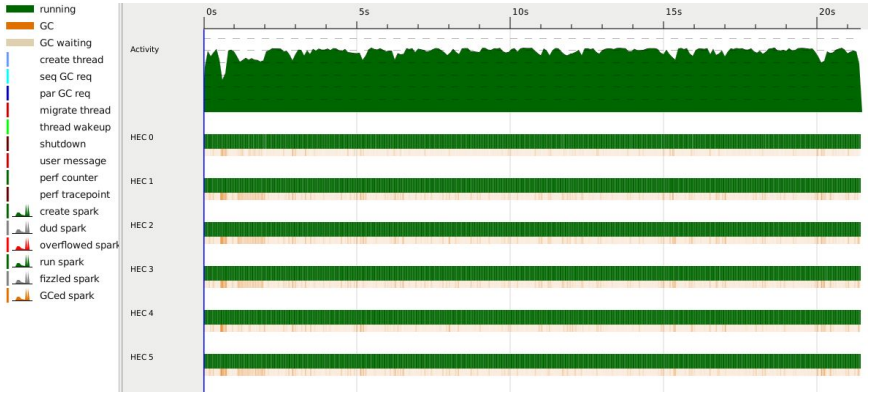
```
parChunks :: [[OrientedRotor]] -> [Char] -> [(Double, String)]
parChunks rots ctext = map (solveIC ctext) rots `using` parListChunk 1000 rdeepseq

bufferChunks ctext = concat $ withStrategy (parBuffer 128 rdeepseq) (map (map (solveIC ctext))
chunks)
  where chunks = chunksOf 1000 allPermutations
```

Results

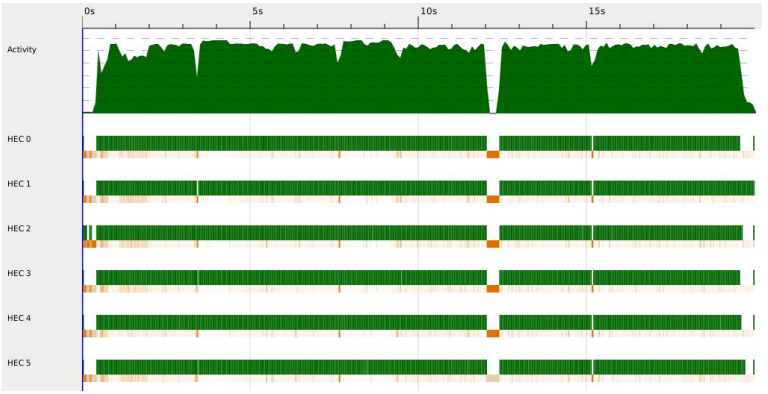


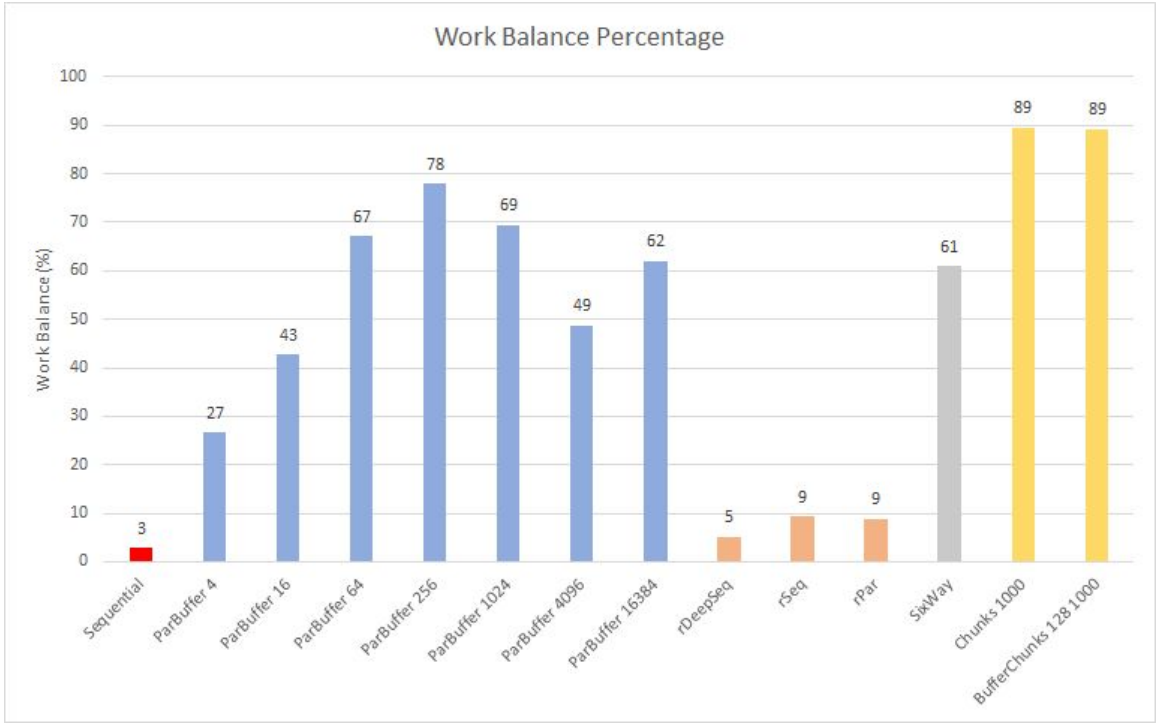
Sequential

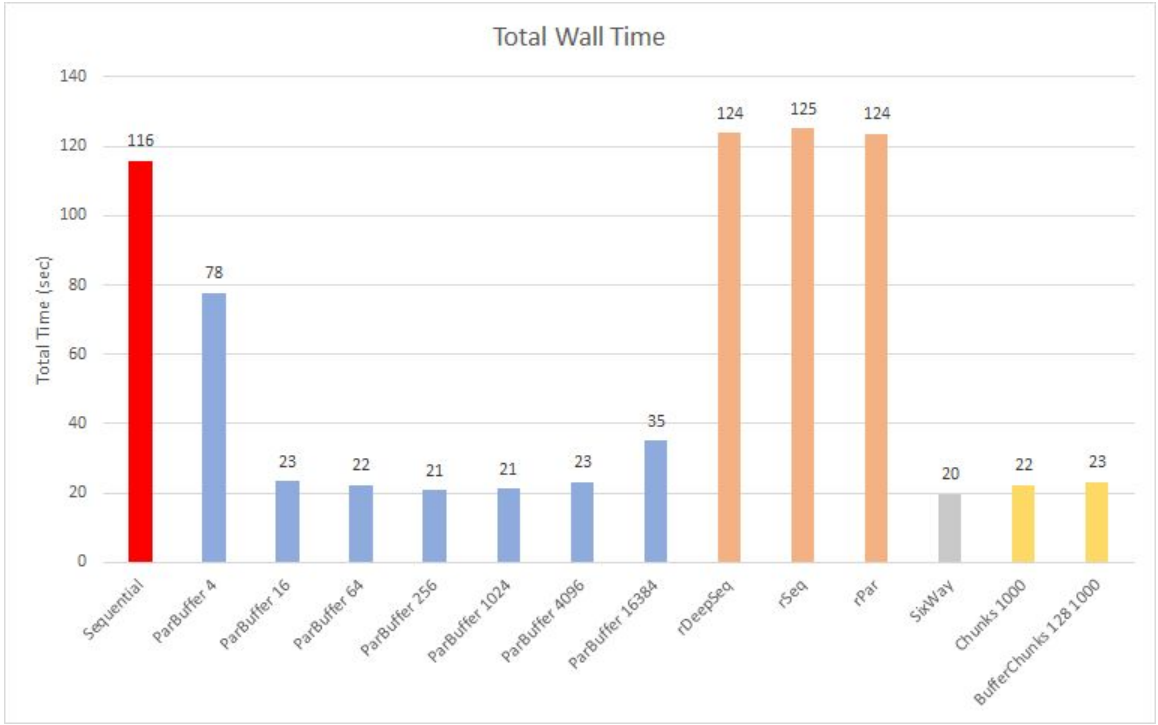
ParBuffer 1024

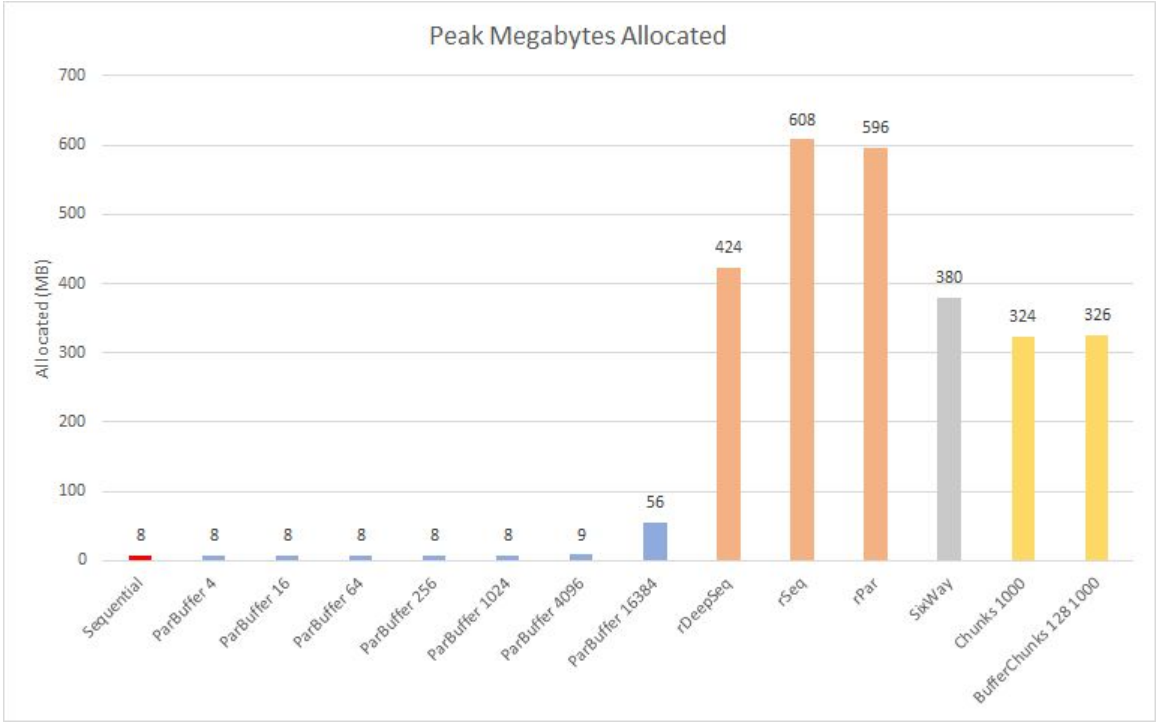


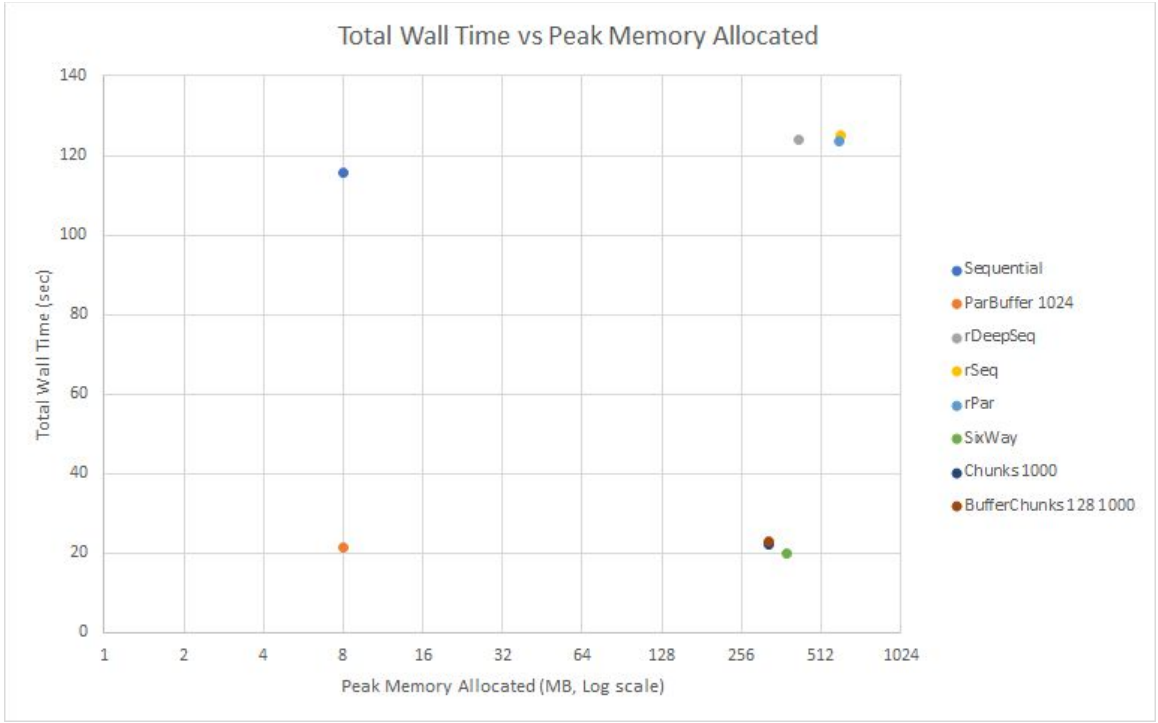
Six-Way

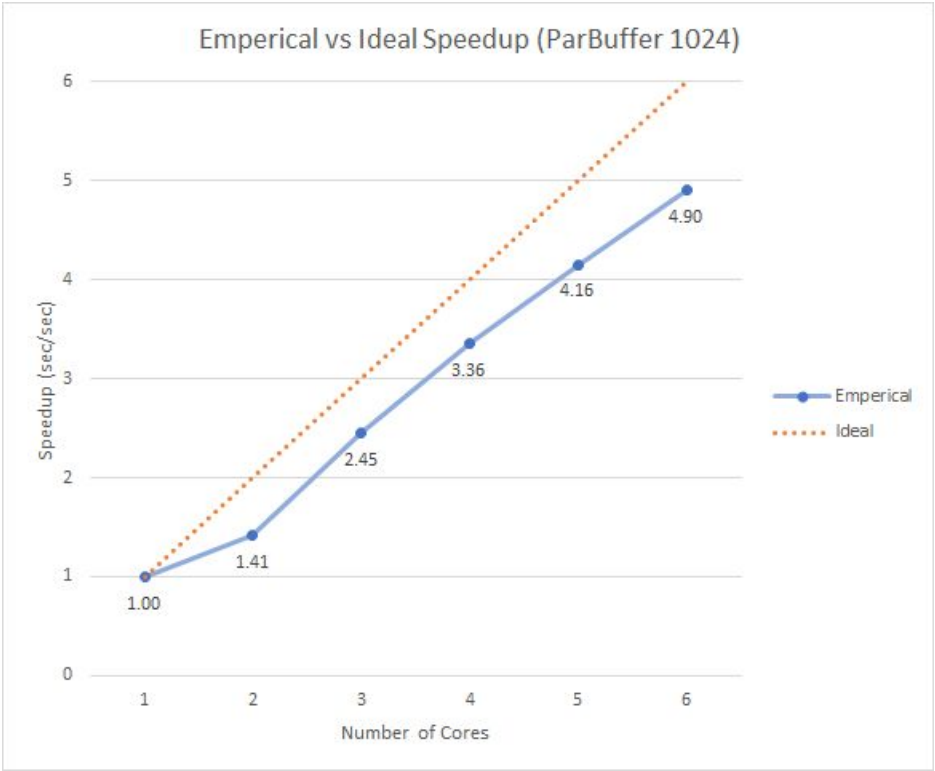


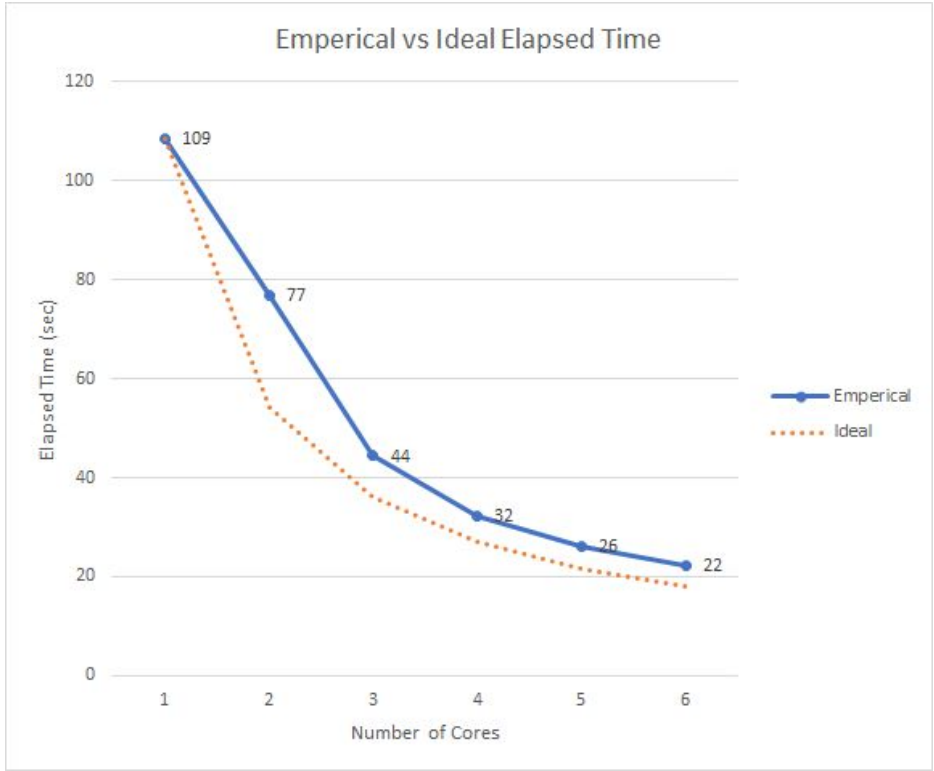

Work Balance Percentage
Work Balance (%)
100
90
80
70
60
50
40
30
20
10
0
3
27
43
67
78
69
49
62
5
9
9
61
89
89
Sequential
ParBuffer 4
ParBuffer 16
ParBuffer 64
ParBuffer 256
ParBuffer 1024
ParBuffer 4096
ParBuffer 16384
rDeepSeq
rSeq
rPar
SixWay
Chunks 1000
BufferChunks 128 1000


Total Wall Time
Total Time (sec)
140
120
100
80
60
40
20
0
116
78
23
22
21
21
23
35
124
125
124
20
22
23
Sequential
ParBuffer 4
ParBuffer 16
ParBuffer 64
ParBuffer 256
ParBuffer 1024
ParBuffer 4096
ParBuffer 16384
rDeepSeq
rSeq
rPar
SixWay
Chunks 1000
BufferChunks 128 1000


Peak Megabytes Allocated
Allocated (MB)
700
600
500
400
300
200
100
0
8
8
8
8
8
8
9
56
424
608
596
380
324
326
Sequential
ParBuffer 4
ParBuffer 16
ParBuffer 64
ParBuffer 256
ParBuffer 1024
ParBuffer 4096
ParBuffer 16384
rDeepSeq
rSeq
rPar
SixWay
Chunks 1000
BufferChunks 128 1000


Total Wall Time vs Peak Memory Allocated
Total Wall Time (sec)
140
120
100
80
60
40
20
0
1
2
4
8
16
32
64
128
256
512
1024
Peak Memory Allocated (MB, Log scale)
Sequential
ParBuffer 1024
rDeepSeq
rSeq
rPar
SixWay
Chunks1000
BufferChunks128 1000


Emperical vs Ideal Speedup (ParBuffer 1024)
Speedup (sec/sec)
0
1
2
3
4
5
6
1.00
1.41
2.45
3.36
4.16
4.90
1
2
3
4
5
6
Number of Cores
Emperical
Ideal


Emperical vs Ideal Elapsed Time
Elapsed Time (sec)
0
20
40
60
80
100
120
109
77
44
32
26
22
1
2
3
4
5
6
Number of Cores
Emperical
Ideal

References

[1] 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 12/21/2020).

[2] Wikipedia. *Enigma Rotor Windows*, url: https://commons.wikimedia.org/wiki/File:Enigma-rotor-windows.jpg (visited 12/22/2020).

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