Parallelized Decision Tree Algorithm

(COMS 4995 - Parallel Functional Programming)

Fall 2021

Phan Anh Nguyen (pn2363) Azhaan Zahabee (az2641)

1 Introduction

Decision Trees are a class of supervised machine learning algorithms that are often used to solve both regression and classification problems. At a high level, the Decision Tree algorithm takes in a set of input data with corresponding labels, greedily builds a lookup-tree where data is grouped at each level according to a 'best-feature' to split the data on, and recursively applies this splitting down the branches until a stop criterion is met. During test time, a new input is classified by following the decision tree nodes until a leaf is reached. The predicted label is that associated with the leaf that the input data ultimately lands on.

The choice of which feature to best split on is based on two concepts: **Entropy** and **Information Gain**. **Entropy** for a set of data that is split on a feature that has c classes measures the degree of 'impurity' associated with a set of data. It is defined as:

$$Entropy(Node) = \sum_{i=1}^{c} -p_i log_2(p_i),\ p_i = \text{proportion of data with label } i \tag{1}$$

Information gain measures the expected reduction in entropy caused by partitioning the examples/data according to an attribute/feature. It is defined as:

$$Gain(Node, A) = Entropy(Node) - \sum_{c \in A} \frac{\mid Node_c \mid}{\mid Node \mid} Entropy(Node_c) \tag{2}$$

From these two equations, we can identify the 'best attribute/feature' to split our data on by selecting the attribute that gives us the most information gain at any node.

For our implementation, we will be focusing on Decision Tree algorithms for classification problems as regresison-based problems require extra pre-processing in the form of data discretization, which we will not focus on in this project. The bottleneck with Decision Tree algorithms lies within the initial building of the tree due to the potential exponential branching factor. The prediction process is trivial as we simply follow the tree until we reach a leaf, going through branches based on what our new data point is. As such, we will only be focusing on implementing and optimizing the tree-building algorithm.

In our report, we discuss several parallelization strategies that were attempted, and discuss the affects that different datasets can have on the runtime and speed up achieved. We were able to achieve anywhere between 1.3x and 16x speedup for our decision tree algorithm.

2 Sequential Implementation

For our project, we referenced a sequential decision tree algorithm implemented by Cortland Walker as the starting point with which to modify and parallelize.

2.1 Data Structure

Label and Feature of the DataSet are stored as strings. Each row of the DataSet comprises of a list of strings (features) and a label. To store the final constructed decision tree, we created a data structure DTree that keeps record of the feature used at each node used for branching as well as the list of children trees.

```
type Label = String
type Feature = String
type Entropy = Double
type DataSet = [([String], Label)]

data DTree = DTree { feature :: String, children :: [DTree] }
    | Node String String deriving Show
```

2.2 Tree Construction

In this part, we begin with the root feature to construct the decision tree. The dataTrees function helps in splitting up the dataset by the attributes that contributes the highest information gain. This function returns a map of key value pairs, where key corresponds to each class of the selected feature based on highest information gain and value corresponds to all other feature values except the selected feature. Then foldrWithKey proceeds with the construction of child trees based on the new information from dataTrees function.

```
dtree :: String -> DataSet -> DTree
dtree f d
    | allEqual (labels strat d) = Node f $ head (labels strat d)
    | otherwise = DTree f $ M.foldrWithKey (\k a b -> b ++ [dtree k a] ) [] (datatrees d)

datatrees :: DataSet -> M.Map String DataSet
datatrees d = foldl (\m (x,n) -> M.insertWith (++) (x!!i) [((x `dropAt` i), fst (cs!!n))] m)
              M.empty (zip (samples d) [0..])
    where
        i = highestInformationGain d
        dropAt xs i = let (a,b) = splitAt i xs in a ++ drop 1 b
        cs = zip (labels d) [0..]
```

2.3 Entropy Calculation

This is a simple function where the entropy calculation is done based on the formula provided in the introduction section. The input is the list of labels corresponding to a particular class of a feature. Later, in the report we have tried to parallelize this calculation of entropy.

```
entropy :: (Eq a) => [a] -> Entropy
entropy xs = sum $ map (\x -> prob x * into x) $ nub xs
    where
        prob x = (length' (elemIndices x xs)) / (length' xs)
        into x = negate $ logBase 2 (prob x)
        length' xs = fromIntegral $ length xs
```

2.4 Information Gain Calculation

The function defined here handles the core functionality of the algorithm. From the given DataSet we construct the list **attr** that contains pair of (class, label) for each feature in the highestInformationGain

function. This list is used to calculate information gain value by selecting each feature separately and the highest information gain value is returned as a result.

In the informationGain function, we make use of helper function splitAttr to split the **attr** list into separate lists having labels corresponding to a particular class of a feature. Then this list is used to calculate the entropy and resultant information gain value.

```
informationGain :: [Label] -> [(Feature, Label)] -> Double
informationGain s a = entropy s - newInformation
    where
        eMap = splitEntropy $ splitAttr a
        m = splitAttr a
        toDouble x = read x :: Double
        ratio x y = (fromIntegral x) / (fromIntegral y)
        sumE = M.map (\x -> (fromIntegral.length) x / (fromIntegral.length) s) m
        newInformation = M.foldWithKey (\k a b -> b + a*(eMap!k)) 0 sumE

-- Determine which attribute contributes the highest information gain
highestInformationGain :: DataSet -> Int
highestInformationGain d = snd $ maximum $ zip (map ((informationGain . labels) d) attrs) [0..]
    where
        attrs = map (attr d) [0..s-1]
        attr d n = map (\(xs,x) -> (xs!!n,x)) d
        s = (length . fst . head) d
```

3 Proof of Concept

We show the correctness of this sequential algorithm by comparing the output of the sequential program to mathematically deduced results that we expect a correct decision tree to output. We use the toy example provided from Cortland Walker's repository that builds the classifier using 14 data points, with 4 features each contain between 2-3 possible classes. The labels in this data is binary, i.e., either Yes or No.

Table: toy data

Sunny	**Hot**	**High**	**Weak**	**No**
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

Below is the output of the sequential algorithm ran on our toy example:

```
DTree {feature= "root", children = [
        DTree {feature="Sunny", children=[
            Node "Normal" "Yes",
            Node "High""No"]},
        DTree {feature="Rain", children=[
            Node "Weak" "Yes",
            Node "Strong" "No"]},
        Node "Overcast" "Yes"]}
```

We first calculate the feature used for splitting at the "root" level:

$Entropy(root) = -\frac{5}{14}(log_2\frac{5}{14}) - \frac{9}{14}(log_2\frac{9}{14}) = 0.940$

$Gain(Root, Feature"Sunny", "Rain", "Overcast")$
$= Entropy("Root") - [Entropy("Sunny") + Entropy("Rain") + Entropy("Overcast")]$
$= 0.94 - (\frac{5}{14}(-\frac{3}{5}(log_2\frac{3}{5}) - \frac{2}{5}(log_2\frac{2}{5}))) - (\frac{5}{14}(-\frac{3}{5}(log_2\frac{3}{5}) - \frac{2}{5}(log_2\frac{2}{5}))) - (\frac{4}{14}(-1(log_2 1) - 0))$
$= 0.246$

$Gain(Root, Feature"Hot", "Cool", "Mild")$
$= Entropy("Root") - [Entropy("Hot") + Entropy("Cool") + Entropy("Mild")]$
$= 0.94 - (\frac{4}{14}(-\frac{2}{4}(log_2\frac{2}{4}) - \frac{2}{4}(log_2\frac{2}{4}))) - (\frac{6}{14}(-\frac{2}{6}(log_2\frac{2}{6}) - \frac{4}{6}(log_2\frac{4}{6}))) - (\frac{4}{14}(-\frac{3}{4}(log_2\frac{3}{4}) - \frac{1}{4}(log_2\frac{1}{1}))))$
$= 0.0289$

$Gain(Root, Feature"Weak", "Strong")$
$= Entropy("Root") - [Entropy("Weak") + Entropy("Strong")]$
$= 0.94 - (\frac{7}{14}(-\frac{4}{7}(log_2\frac{4}{7}) - \frac{3}{7}(log_2\frac{3}{7}))) - (\frac{7}{14}(-\frac{1}{7}(log_2\frac{1}{7}) - \frac{6}{7}(log_2\frac{6}{7}))))$
$= 0.152$

Based on this, we would choose to split the root by the feature whose classes are "Sunny","Rain","Overcast". This is inline with what the algorithm predicts.

We perform the same calculations on the next level for each of the "Sunny", "Rain" and "Overcast" nodes. We see that for the "Sunny" node, we obtain the highest gain by splitting the feature whose classes are "High","Normal". For the "Rain" node, highest gain is obtained by splitting on the "Weak","Strong" feature. We notice that next level nodes are pure so we stop there. For the "Overcast" node, we realize that the node is pure as all rows have a label of "Yes" so no further splitting is needed and we stop there.

Thus, we see that our algorithm has produced the correct decision tree.

4 Effect of Dataset on Algorithmic Runtime

The size and depth of a decision tree, and therefore algorithm runtime, is highly influenced by the underlying dataset being used. At each node, we choose the highest features to split the dataset and continue to build the tree until the leaf nodes are pure or all features have been used. Since no feature is re-used in the implementation of this algorithm, the number of features thus creates an upper bound on the depth of the tree.

Furthermore, the number of children (branching factor) that is created after we decide on a feature to split by is also dependent on the number of possible classes that exist for that feature in the dataset. For a binary feature, there are only two children, whereas a multi-class feature could generate up to hundreds or thousands of children. Since we have to go through every data point to calculate

the entropy at each node, the runtime for the algorithm would be: $O(nb^d + k)$, where n is the number of data points, b is the average branching factor, d is the depth of the final decision tree and k is the remaining time spent on I/O or reading in of data.

By Amdahl's Law, the theoretical maximum speedup is dependent on the proportion of work that is parallelizable. Depending on the dataset, the proportion of time spent on building the tree versus reading in the data can thus affect the degree of speedup. In our report, we test our strategies with 2 different datasets to explore how changes in feature number and number of classes within a feature affect speedup gain from parallelization.

The first dataset is the "Cat in the Dat II" dataset from Kaggle, which contains 300,000 training samples with 17 features. Each feature may have up to 5 different classes and there are 2 possible Y labels to be predicted.

The second dataset is a subset of the "A-Z MNIST Letter Recognition: dataset. This is a letter-classification dataset with roughly 25,000 data points. The training data consists of 28x28 images that have been vectorized to obtain 784 features for each datapoint, which each feature having potentially 256 different classes that correspond to a pixel RGB value between 0-255. There are 26 possible y labels for each datapoint which correspond to each letter of the alphabet.

5 Parallelization Strategies

We have identified several potential sources of parallelization that can be done:

1. **Multi-core** parallelization: We can try to increase the number of cores accessible by Haskell in running the program. In our report, we attempt to use up to 8 cores to run both sequential and parallelized versions of the algorithm. We expect to see minor to no speed-ups in the sequential version. For the parallelized algorithm, we expect to see speed-up to scale acccordingly as we increase core count until a maximum whereby any increased performance gain from more cores will be outweighed by increased overhead costs.

2. **Entropy** parallelization: The sequential implementation of the Entropy calculation for a node maps the entropy equation over each possible class that the label can have. For a multi-label dataset, we could potentially parallelize the entropy calculation for each class. Since the entropy equation solely depends on values in a current class, as well as the number of datapoints that have a specific class within a feature, we can potentially parallelize this process.

3. **Information Gain** parallelization: The sequential implementation of the Highest Information Gain calculation for a node maps over each possible attribute that we could split on and chooses the attribute that results in the highest gain. Since the information gain calculations self-contained within the columns (i.e: does not depend on information gain results from other columns), we can also potentially parallelize this process.

4. **Miscellaneous row-based operations** parallelization: Lastly, the sequential implementation has several helper functions that map on the row/datapoint level such as the **'samples'** or the **'labels'** function. These functions are used deeper within the entropy/information gain function call and potentially provide a more granular level of parallelization in addition to the two main sources of parallelization previously described.

Based on the above sources of parallelization, we define 4 potential parallelization strategies that can be applied to our 2 datasets:

5.1 Increasing Core Count

With this strategy, we simply increase the number of cores/Haskell Execution Contexts (HECs) used. There is no change to the underlying decision tree algorithm. This strategy serves as a baseline to determine efficacy of other parallelization strategies with 1, 2, 4 and 8 core.

Cat in the Dat II Dataset

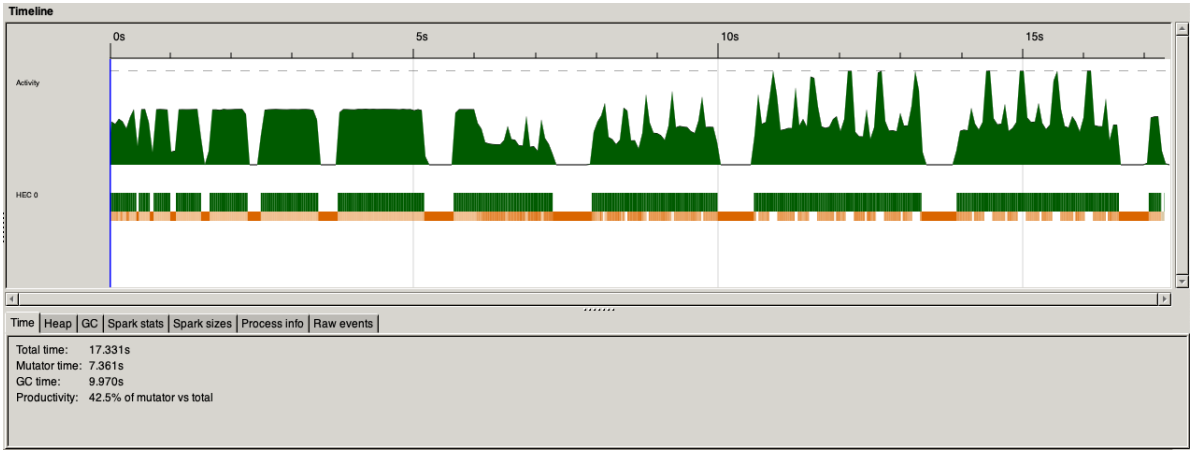


Fig 1. Sequential Algorithm/1-Core, Cat-in-the-dat dataset

A-Z Handwritten Dataset

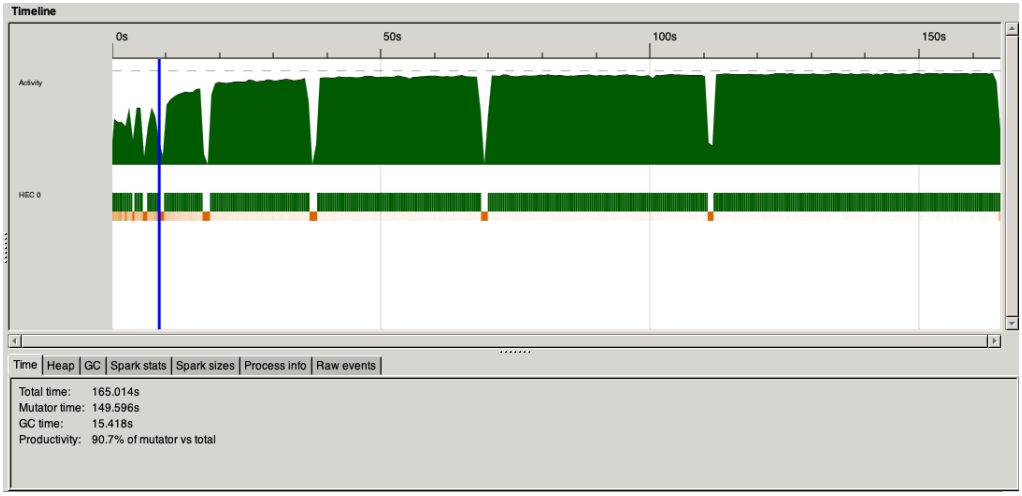


Fig 2. Sequential Algorithm/1-Core, A-Z handwritten dataset

5.2 Single-source parallelization

With this strategy, in addition to using multiple cores, we attempted to obtain performance gains by either parallelizing the Entropy calculation OR Information Gain. This serves to determine which source of parallelization results in more performance gains.

Entropy is parallelized by implementing the following code snippet:

```
entropy :: (Eq a) => String -> [a] -> Entropy
entropy strat xs | strat `elem` ["2","4","5"] = sum $ parMap rseq (\x -> prob x * into x) $ nub xs
                 | otherwise = sum $ map (\x -> prob x * into x) $ nub xs
        where
            prob x = (length' (elemIndices x xs))/(length' xs)
            into x = negate $ logBase 2 (prob x)
            length' xs = fromIntegral $ length xs
```

Information Gain is parallelized by implementing the following code snippet:

```
highestInformationGain :: String -> DataSet -> Int
highestInformationGain strat d = snd $ maximum $ infoGains
  where
    infoGains
       | strat `elem` ["3", "4", "5"] = zip (parMap rseq (((informationGain strat)
            . (classes strat)) d) attrs) [0..]
       | otherwise = zip (map (((informationGain strat) . (classes strat)) d) attrs) [0..]
    attrs = map (attr d) [0..s-1]
    attr d n = map (\(xs,x) -> (xs!!n,x)) d
    s = (length . fst . head) d
```

Entropy - Cat in the Dat II Dataset

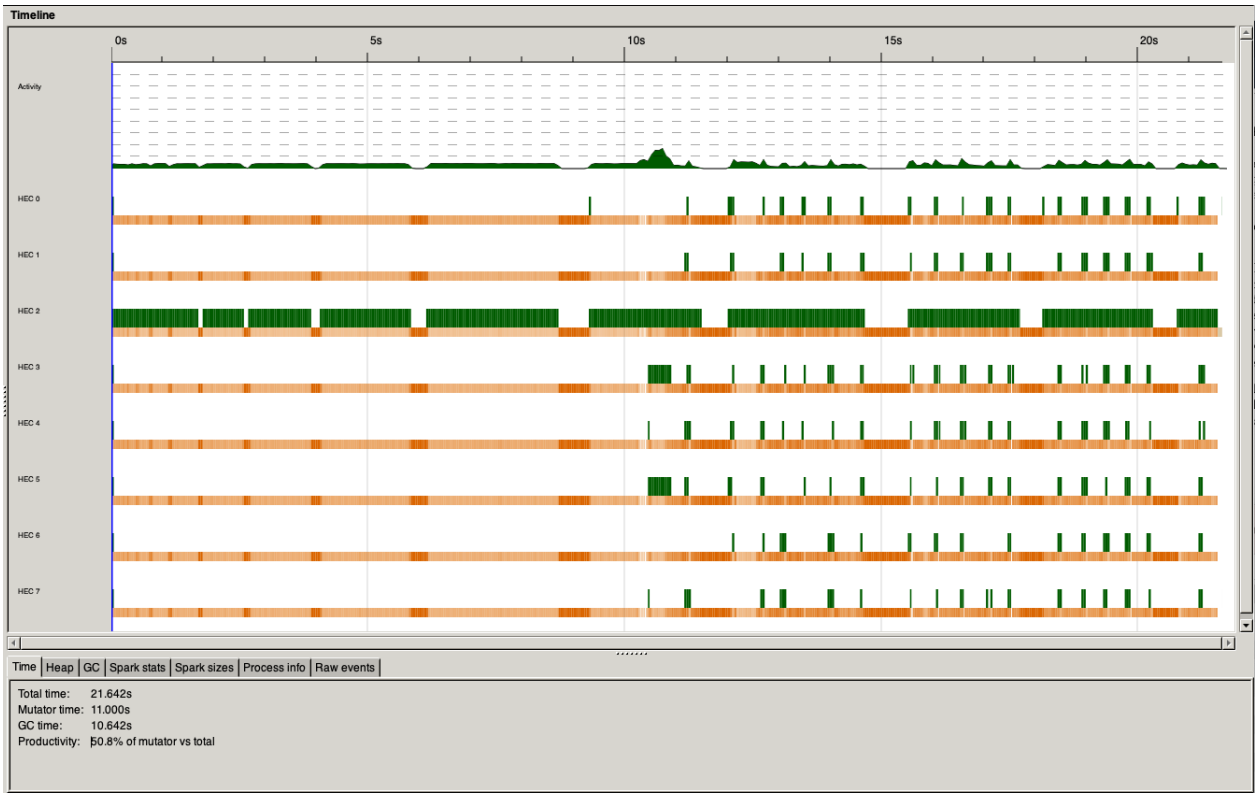


Fig 3. Entropy Parallelization/8-Core, Cat-in-the-dat dataset

Entropy - A-Z Handwritten Dataset

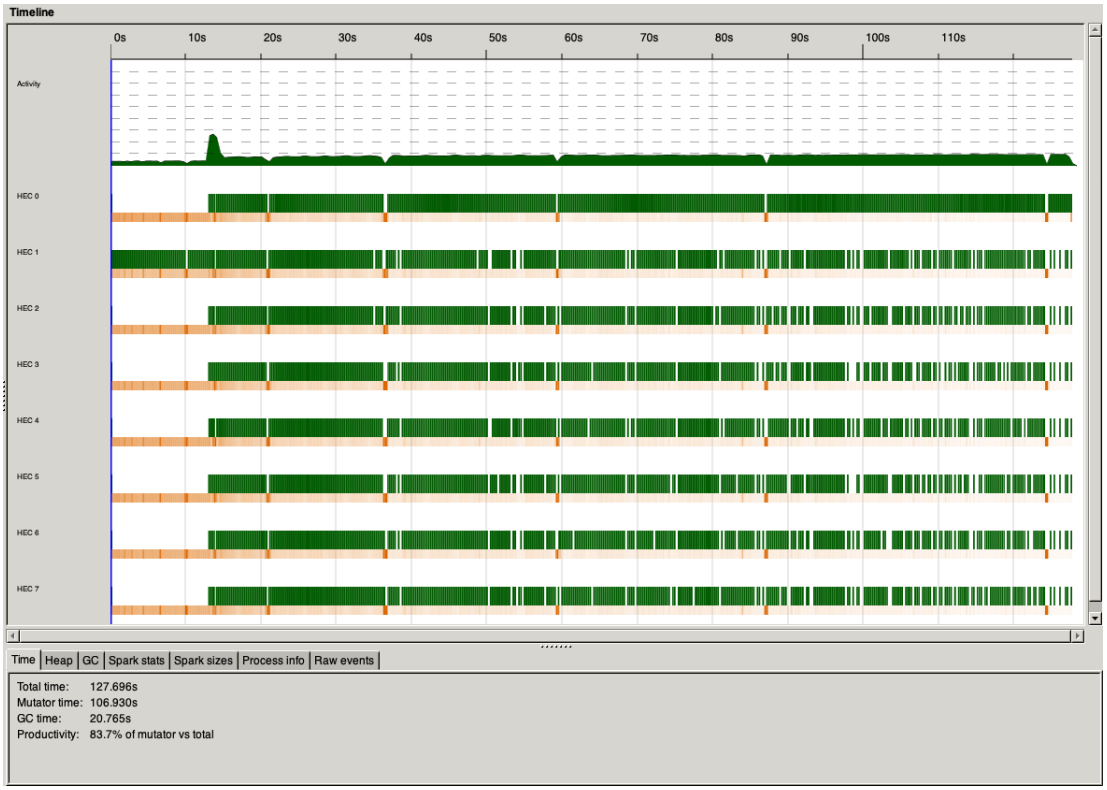


Fig 4. Entropy Parallelization/8-Core, A-Z Handwritten dataset

Information Gain - Cat in the Dat II Dataset

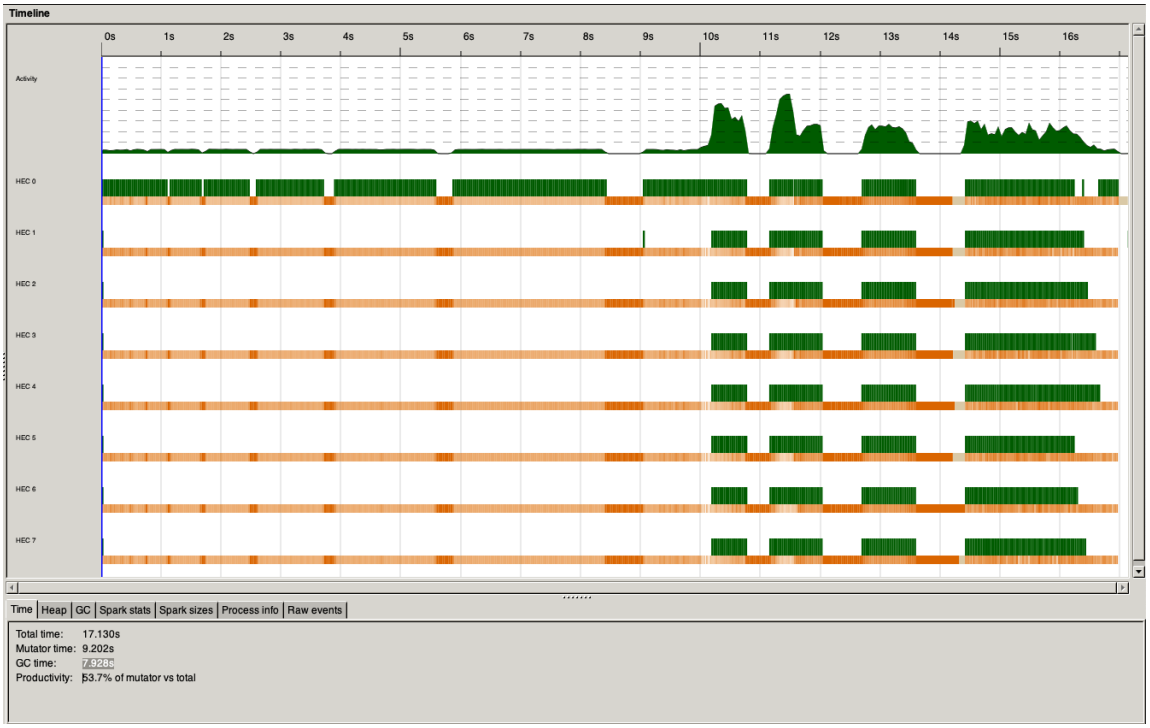


Fig 5. Information Gain Parallelization/8-Core, Cat-in-the-dat dataset

Information Gain - A-Z Handwritten Dataset

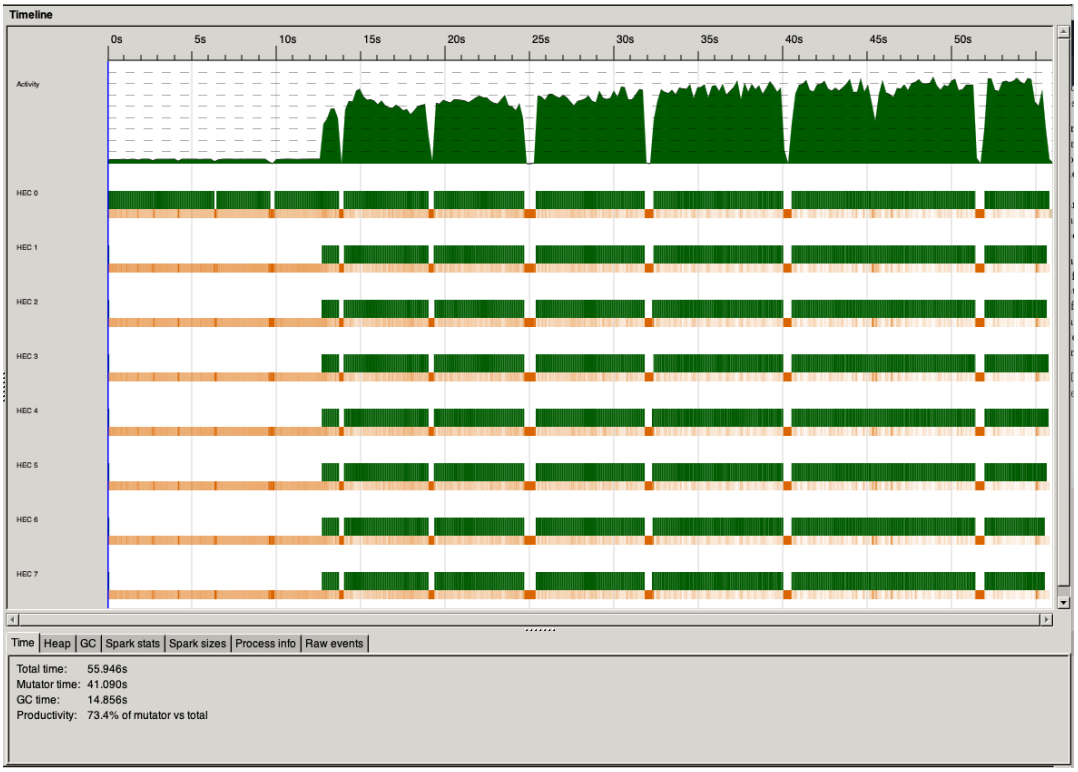


Fig 6. Information Gain Parallelization/8-Core, A-Z Handwritten dataset

5.3 Combined parallelization

It seeks to combine parallelization from both Entropy and Information Gain to maximize performance.

Combined - Cat in the Dat II Dataset

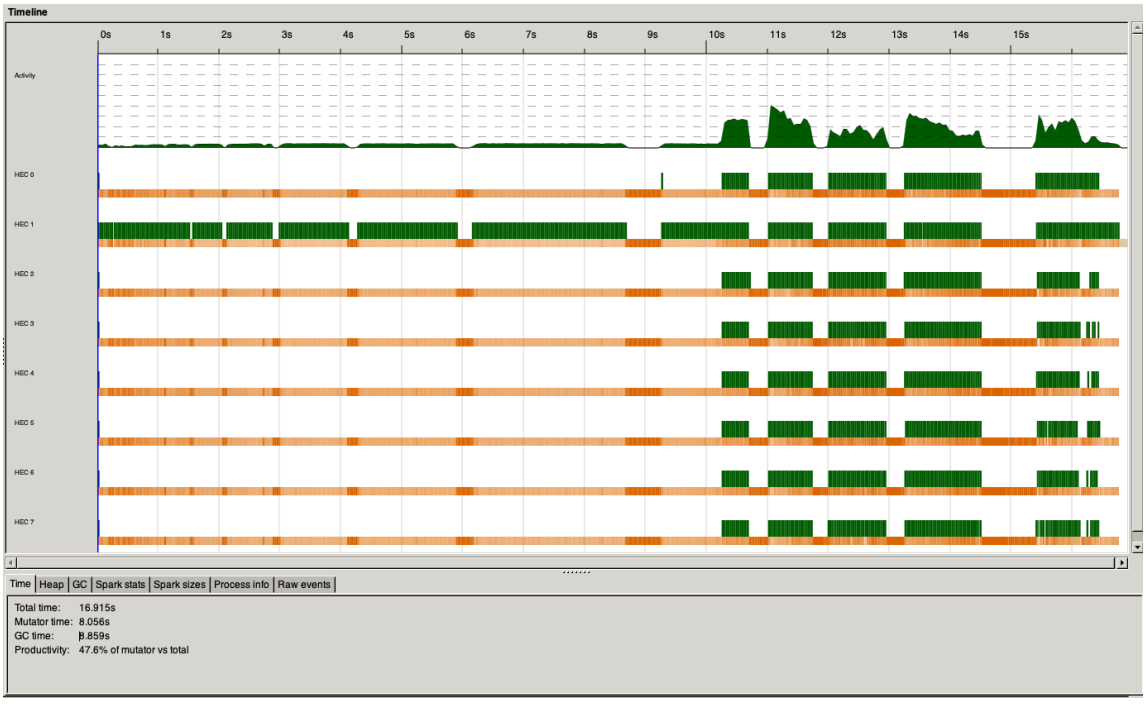


Fig 7. Combined Parallelization/8-Core, Cat-in-the-dat dataset

Combined - A-Z Handwritten Dataset

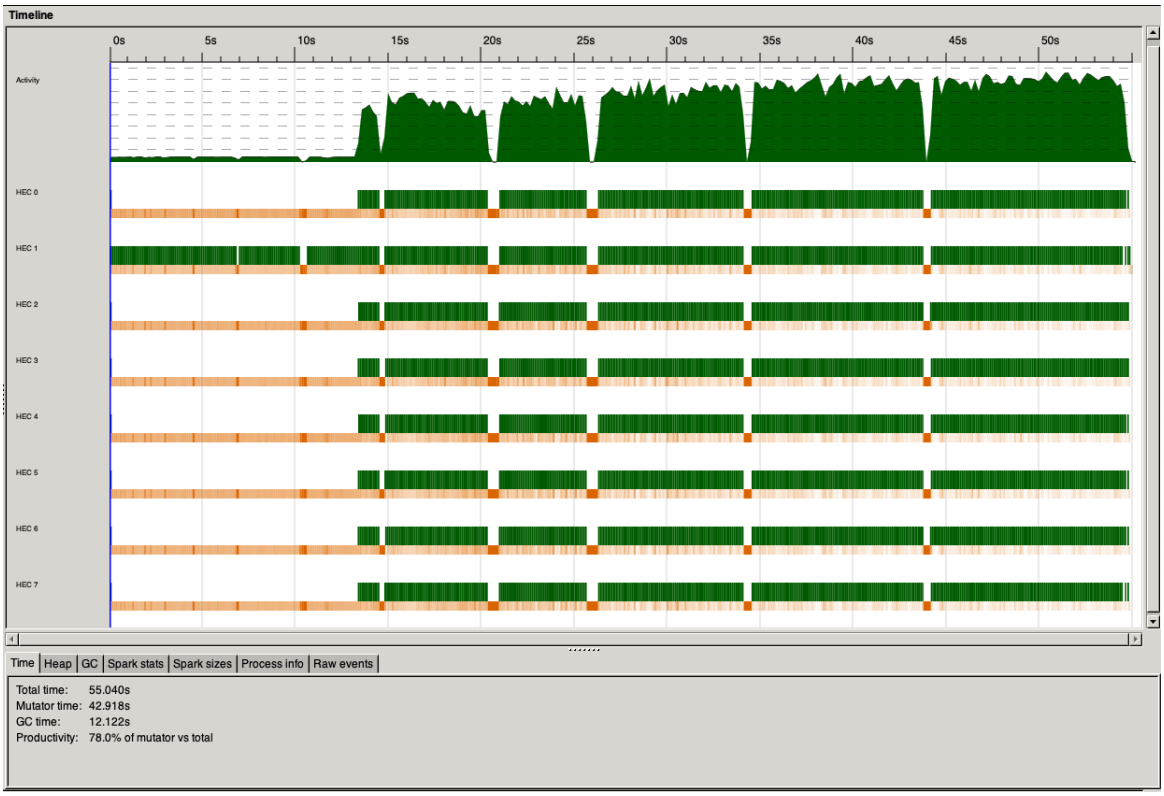


Fig 8. Combined Parallelization/8-Core, A-Z Handwritten dataset

5.4 Combined + Miscellaneous row-based operations parallelization

This strategy seeks to combine parallelization from both Entropy and Information Gain to maximize performance gains as well as attempts to leverage more granular parallelization in the form of downstream row-based operations such as 'samples' and 'labels' functions.

'Samples' function is parallelized by implementing the following code snippet:

```
samples :: String -> DataSet -> [[String]]
samples strat sdata | strat == "5" = withStrategy (parBuffer 100 rdeepseq) (map fst sdata)
                    | otherwise = map fst sdata
```

'Labels' function is parallelized by implementing the following code snippet:

```
labels :: String -> DataSet -> [Class]
labels strat sdata | strat == "5" = withStrategy (parBuffer 100 rdeepseq) (map snd sdata)
                   | otherwise = map snd sdata
```

Combined + Misc - Cat in the Dat II Dataset

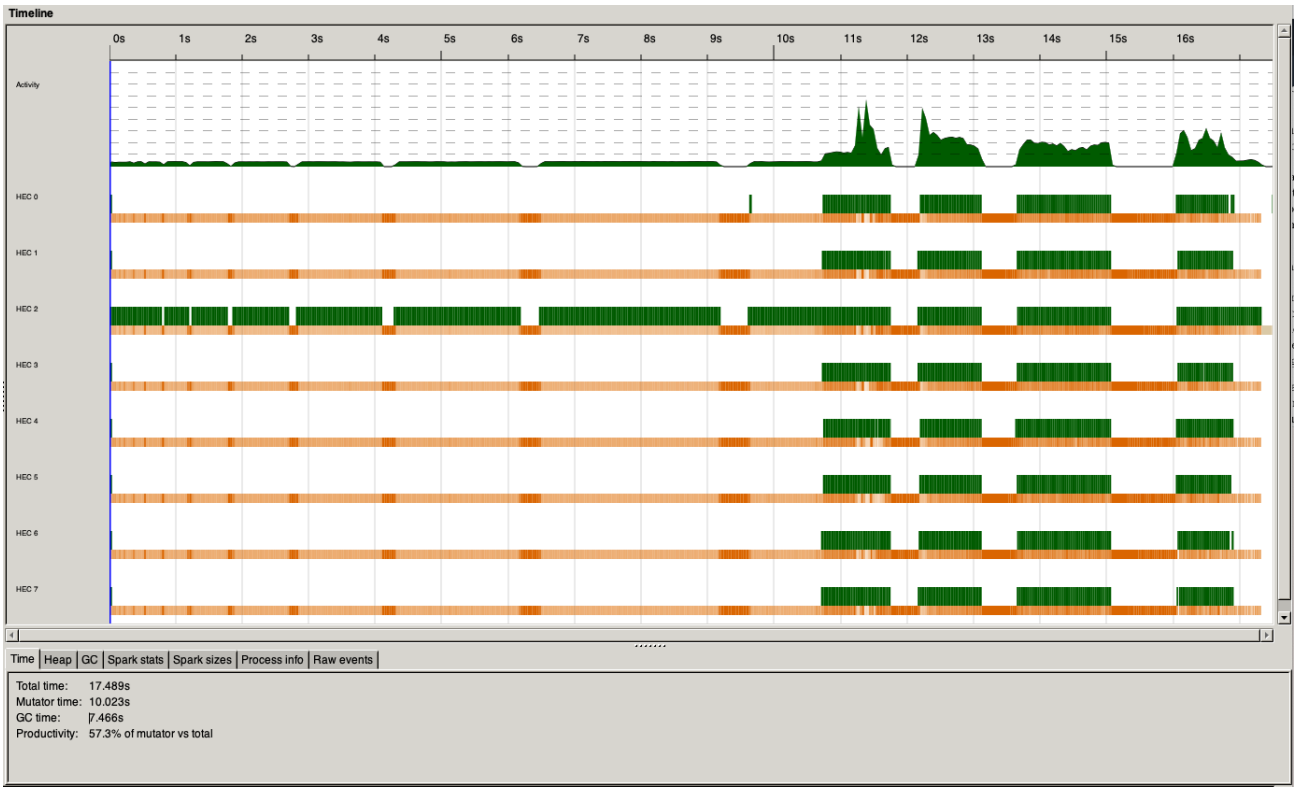


Fig 9. Combined + Misc Parallelization/8-Core, Cat-in-the-dat dataset

Combined + Misc - A-Z Handwritten Dataset

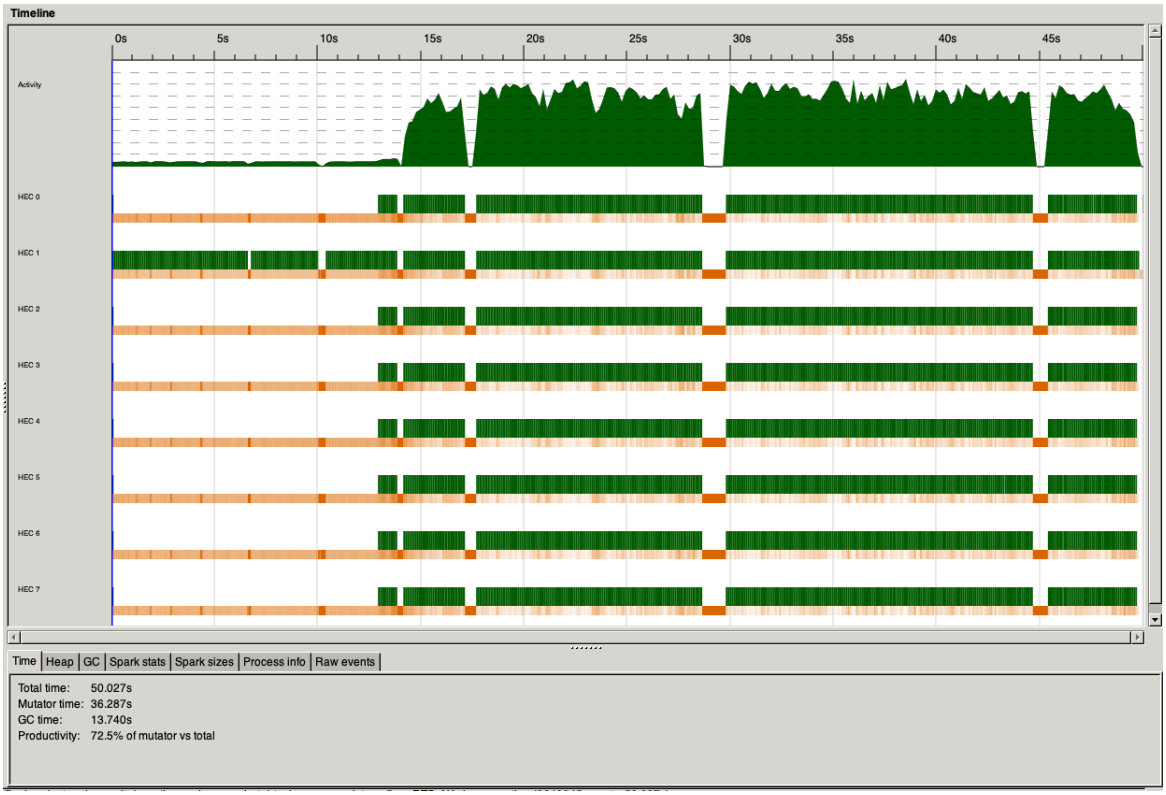


Fig 10. Combined + Misc Parallelization/8-Core, A-Z Handwritten dataset

6 Discussion

We first look at the use of multiple cores for the sequential algorithm. Looking at the threadscope event log for both datasets, we see that most of the work is still completed by only one core. This suggests that the initial algorithm is highly sequential and was implemented in a way such that the system is unable to utilize parallelism even if we throw more cores at the program. As such, there is almost no improvement in runtimes as seen in **Fig.11**, and **Fig.12**.

Overall Performance - Cat in the Dat II Dataset

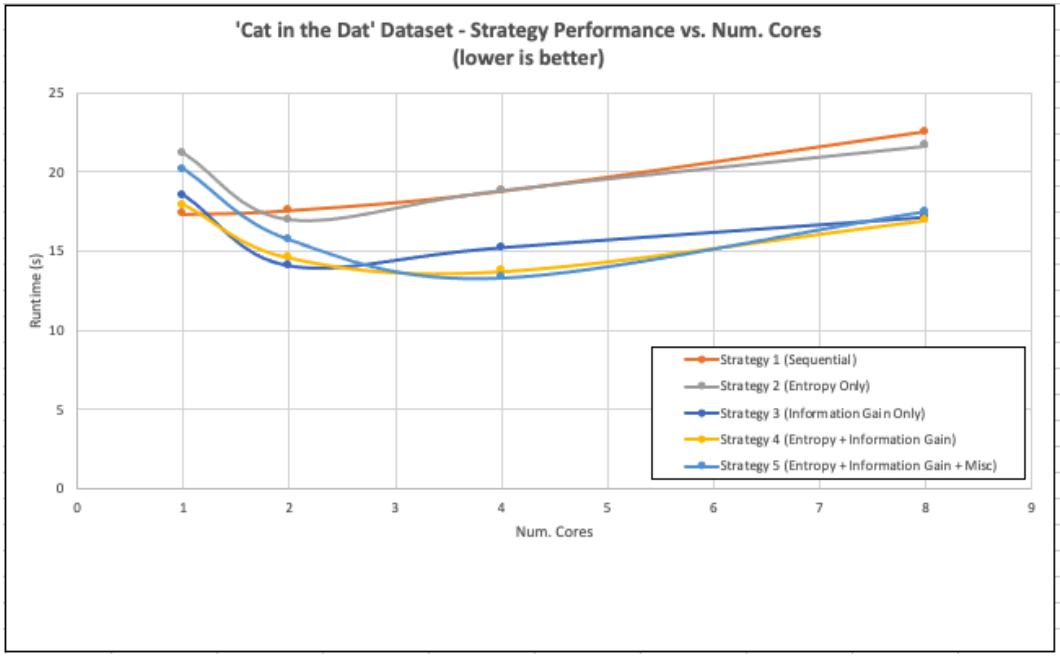


Fig 11. Overall Performance of Strategies across cores, Cat in the Dat II dataset

Overall Performance - A-Z Handwritten Dataset

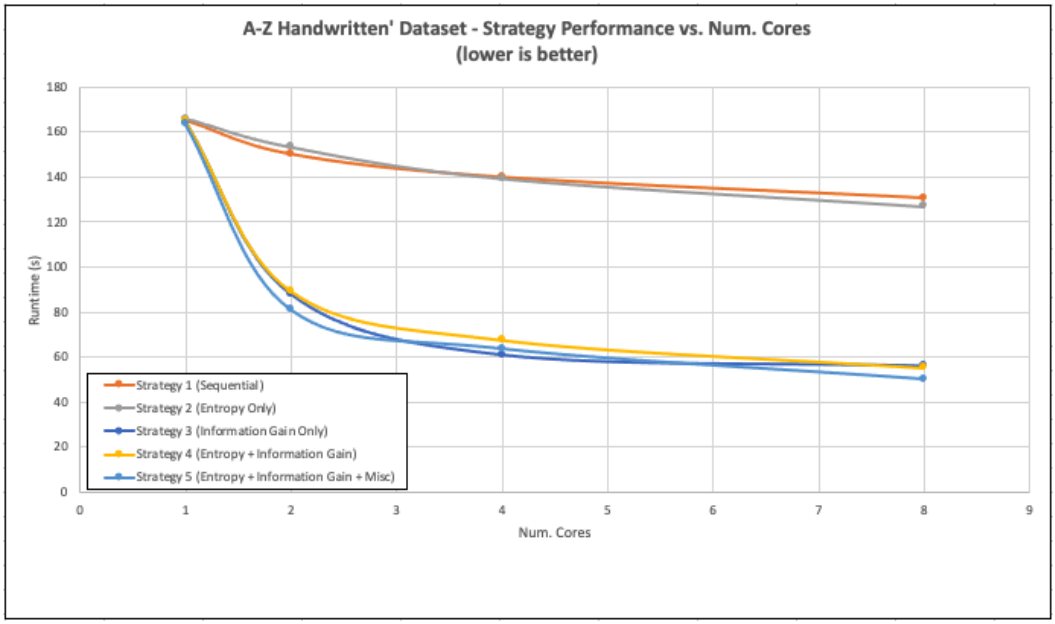


Fig 12. Overall Performance of Strategies across cores, A-Z Handwritten dataset

We only start to notice multiple cores being used by the program once parallelized **Entropy** and **Information Gain** calculations are implemented. Looking at graphs of **Fig.3, Fig.4, Fig.5, and Fig.6**, we see multiple cores being used which suggests that parallelization is working.

To calculate the theoretical speedup limit for each dataset, we need to look at what proportion of total work is parallelizable. We noticed that for all parallel strategies implemented, there is always a fixed ∼11 -12 seconds of sequential work being done. Given that paralellism is implemented relatively early on in the actual building of the Decision Tree, we suspect that this sequential portion is due to I/O from reading in the input data. Since we are using the external 'Text' library for this part of the algorithm, we did not implement parallelism here. This seemed to be the only sequential portion. Since the sequential version of the 'Cat in the Dat' dataset takes ∼17.3 seconds and that of the 'A-Z Handwritten' dataset takes ∼165 seconds, we estimate potential theoretical maximum speedup of ∼**1.5x** for the 'Cat in the Dat' dataset and ∼**14x** for the 'A-Z Handwritten' dataset.

In reality, the maximum speedup we achieved for the 'Cat in the Dat' dataset at was close to expected at ∼**1.3x**, while far from expected for the 'A-Z Handwritten' dataset at ∼**3.3x**. For the 'A-Z Handwritten' dataset, we suspect that additional performance gains may be obtained by increasing the number of cores used as shown in **Fig.12**. However, notice that we do see a tapering off of performance gains from these figures which suggests that the additional overhead from increasing core count will negate any additional performance gains as we reach a certain core count. Hence, it seems like there can be improvements made to our implementation of paralellization. We discuss an additional source of parallelization that can be implemented as future work in the **Conclusion** section.

Next, we wanted to compare the different sources of parallelization and see how much they contribute to the overall performance gain. Firstly, we look at parallelizing Entropy. For the 'Cat in the Dat' dataset, we see only minor gains (∼**1.04x**) from parallelizing Entropy with 2 cores. With 4 and 8 cores, we actually obtain worse performance than the sequential counterpart. Since there are only 2 possible classes to parallelize for each Entropy calculation, we suspect that the higher overhead associated with scheduling parallelization did not result in any significant improvement. The situation is only slightly improved when we look at the 'A-Z Handwritten' dataset with 26 possible classes. The maximum speedup obtained was (∼**1.3x**). This seems to suggest that parallelizing Entropy may not have a significant affect on performance. However, we realize that these results are only preliminary. In practice, it is not uncommon for datasets to have much more than 26 possible labels. Several Natural Language Processing tasks such as Named Entity Recognition or Next-word Prediction may have up to several million possible Y labels. A possible improvement on this work would be to test this strategy against such datasets. Unfortunately, due to time and resources constraints, we were unable to find and test such a dataset while completing our project.

Parallelizing Information Gain seems to tell a different story, however. With the 'Cat in the Dat' dataset, we were able to obtain a ∼**1.24x** speedup, compared to the ∼**1.5x** theoretical maximum. With the 'A-Z handwritten' dataset, we were able to obtain a ∼**2.95x** speedup, which is significantly better than the gains obtained from parallizing Entropy.

Overall, the best strategy for both datasets seemed to be Strategy 5. With the 'Cat in the Dat' dataset, we were able to obtain a ∼**1.3x** speedup, while the 'A-Z Handwritten' dataset obtained ∼**3.3x**. In this strategy, we apply all sources of parallelization from both Entropy and Information Gain, as well as parallelized row-based operations such as the 'samples' and 'labels' function which extract the features and labels from a data row respectively. We notice that the optimal number of cores varies between dataset. For the the 'Cat in the Dat' dataset, we seem to reach optimal cores much earlier on at 4 cores (**Fig.11**), while with the 'A-Z Handwritten' dataset, it seems that we are still obtaining performance gains as we approach 8 cores (**Fig.12**). This suggests that in practice, exact number of cores to provide will need some fine-tuning of hyperparameters.

7 Conclusion

Overall, we see that the Decision Tree algorithm is a highly parallelizable algortihm as we were able to obtain up to ∼**3.3x** speedup through parallelization of **Entropy**, **Information Gain**, **Row-based operations**, as well as increasing the number of cores provided to the program. We also determined that the exact degree of speedup, as well as how much each source of parallelization contributes to the overall performance gain is dependent on the underlying data being used.

8 Future Work

Throughout our experiements, we have noted that the data input process was mostly sequential throughout all strategies implemented. Given further resources, we would like to potentially explore ways to parallelize this process in the future. This is an important consideration as for specific datasets, reading in the data constitutes the majority of the work being done, and therefore, sets a limit of the degree of parallelization by Amdahl's law.

Lastly, we would like to point out another source of parallelization that could have been implemented. We realized that since different feature branches do not rely on each other for any calculations, it is theoretically possible to parallelize the recursive creation of branches in a Decision Tree. The sequential algorithm, however, uses the 'M.foldrwithkey' function to achieve this, which we have found is not easily parallelizable. As such, if we were to re-implement this algorithm from scratch, we will try to design the tree-building functions with parallelization in mind.

9 References and Acknowledgements

We would like to thank Professor Edwards and the TAs for your guidance throughout the semester!

In completion of our report, we have consulted the following references:

- Sequential Decision Tree Classifier implementation, Cortland Walker, Github: https://github.com/Cortlandd/Haskell-Data-Tree-Classifier
- Dataset 1: https://www.kaggle.com/c/cat-in-the-dat-ii/overview
- Dataset 2: https://www.kaggle.com/sachinpatel21/az-handwritten-alphabets-in-csv-format
- *Intelligence, A Modern Approach (Fourth Edition)*, Stuart Russel, Peter Norvig
- *Parallel and Concurrent Programming in Haskell*, Simon Marlow
- COMS 4995 - Parallel Functional Programming Lecture Notes
- COMS 4701 - Artificial Intelligence Lecture Notes
- https://www.kdnuggets.com/2020/01/decision-tree-algorithm-explained.html

10 Code Listing

```
module Main where

import Data.List(nub, elemIndices)
import qualified Data.Map as M
import Text.CSV
import Control.Parallel.Strategies
import System.Exit(exitFailure)
import System.Environment(getArgs)
```

```haskell
-- Better type synonyms for better understanding
-- of what data is being passed around.
type Class = String
type Feature = String
type Entropy = Double
type DataSet = [([String], Class)]

-- Define a data structure for a decision tree that'll be constructed
data DTree = DTree { feature :: String, children :: [DTree] }
    | Node String String deriving Show

main :: IO ()
main = do
    args <- getArgs
    [filename, strat] <- case args of
        [f, s] -> return [f, s]
        _ -> do
                error "Usage: stack run <filename> <strat> -- +RTS -N<numHEC> -ls -s"
                exitFailure
    strategy <- case strat of
        x | x `elem` ["1", "2", "3", "4", "5"] -> do return strat
        _ -> do
                error "Usage: Valid strat options are: <1> - Sequential, <2> - Single-Choice (Entropy),
                <3> - Single-Choice (InformationGain), <4> - Both (w/o Misc.), <5> - Both (with Misc)"
                exitFailure
    rawCSV <- parseCSVFromFile ("src/" ++ filename)
    either handleError doWork rawCSV strategy

handleError = error "invalid file"

-- IF file is read successfully
-- THEN remove any invalid CSV records and construct a decision tree out of it
doWork :: CSV -> String -> IO ()
doWork fcsv strat = do
    let removeInvalids = filter (\x -> length x > 1)
    -- #1
    let myData = map (\x -> (init x, last x)) $ removeInvalids fcsv
    let result = dtree strat "root" myData
    let firstNodeChildren = length b
          where DTree a b = result
    print "Final Decision Tree:"
    print result

-- Helper functions to break up the DataSEt tuple into
-- a list of samples or list of classes.
-- #2
samples :: String -> DataSet -> [[String]]
samples strat sdata | strat == "5" = parMap rseq fst sdata
                    | otherwise = map fst sdata

-- #3
classes :: String -> DataSet -> [Class]
classes strat sdata | strat == "5" = parMap rseq snd sdata
                    | otherwise = map snd sdata

-- Calculate the entropy of a list of values
-- #4
entropy :: (Eq a) => String -> [a] -> Entropy
entropy strat xs | strat `elem` ["2","4","5"] = sum $ parMap rseq (\x -> prob x * into x) $ nub xs
```

```
                | otherwise = sum $ map (\x -> prob x * into x) $ nub xs
    where
        prob x = (length' (elemIndices x xs))/(length' xs)
        into x = negate $ logBase 2 (prob x)
        length' xs = fromIntegral $ length xs

-- Split an attribute by its features
splitAttr :: [(Feature, Class)] -> M.Map Feature [Class]
splitAttr dc = foldl (\m (f,c) -> M.insertWith (++) f [c] m) M.empty dc

-- Obtain each of the entropies from splitting up an attribute by its features.
splitEntropy :: String -> M.Map Feature [Class] -> M.Map Feature Entropy
splitEntropy strat m = M.map (entropy strat) m

-- Compute the information gained from splitting up
-- an attribute by its features
informationGain :: String -> [Class] -> [(Feature, Class)] -> Double
informationGain strat s a = entropy strat s - newInformation
    where
        eMap = splitEntropy strat $ splitAttr a
        m = splitAttr a
        toDouble x = read x :: Double
        ratio x y = (fromIntegral x) / (fromIntegral y)
        sumE = M.map (\x -> (fromIntegral.length) x / (fromIntegral.length) s) m
        newInformation = M.foldrWithKey (\k a b -> b + a*(eMap M.! k)) 0 sumE

-- Determine which attribute contributes the highest information gain
-- #5
highestInformationGain :: String -> DataSet -> Int
highestInformationGain strat d = snd $ maximum $ infoGains
    where
        infoGains
            | strat `elem` ["3", "4", "5"] =
                zip (parMap rseq (((informationGain strat) . (classes strat)) d) attrs) [0..]
            | otherwise = zip (map (((informationGain strat) . (classes strat)) d) attrs) [0..]
        attrs = map (attr d) [0..s-1]
        attr d n = map (\(xs,x) -> (xs!!n,x)) d
        s = (length . fst . head) d

-- Split up the dataset by the attributes that contributes the highest
-- information gain
datatrees :: String -> DataSet -> M.Map String DataSet
datatrees strat d = foldl (\m (x,n) -> M.insertWith (++) (x!!i) [((x `dropAt` i), fst (cs!!n))] m)
                    M.empty (zip (samples strat d) [0..])
    where
        i = highestInformationGain strat d
        dropAt xs i = let (a,b) = splitAt i xs in a ++ drop 1 b
        cs = zip (classes strat d) [0..]


-- A helper function to determine if all  elements of a list are equal.
-- Used to check fi further splitting of a dataset is necessary by checking
-- if its classes are identical.
allEqual :: Eq a => [a] -> Bool
allEqual [] = True
allEqual [_] = True
allEqual (x:xs) = x == (head xs) && allEqual xs

-- Construct the decision tree from a labeling and a dataset of samples
```

```
dtree :: String -> String -> DataSet -> DTree
dtree strat f d
    | allEqual (classes strat d) = Node f $ head (classes strat d)
    | otherwise = DTree f $ M.foldrWithKey (\k a b -> b ++ [dtree strat k a] ) [] (datatrees strat d)
```

11 Appendix and Notes

Our code can be ran by first unpacking the zip file. Following that, we can run:

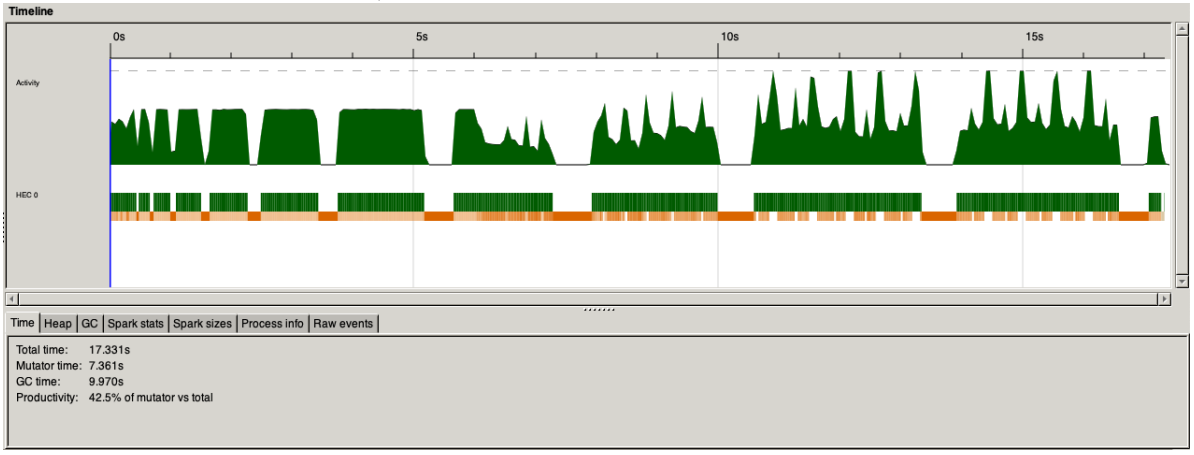
```
cd final-project
stack build
stack run <input-filename> <strategy> -- +RTS -N<Num. cores> -ls -s
```

An example run would be:

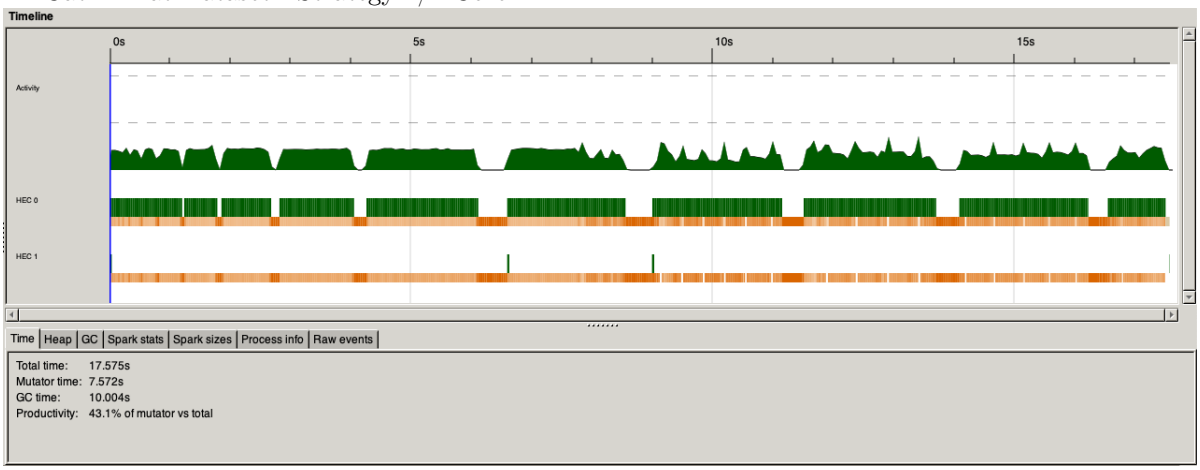
```
stack run a-z-mnist.csv 3 -- +RTS -N4 -ls -s
```

Raw Eventlog Graphs

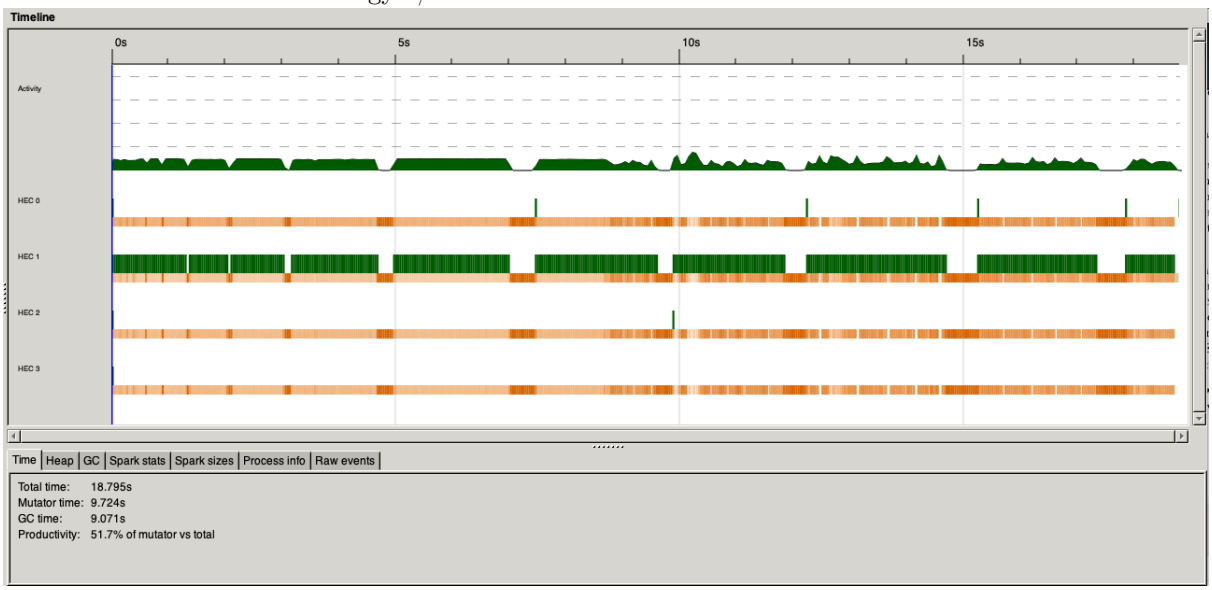
Cat-In-Dat Dataset - Strategy 1/1-Core



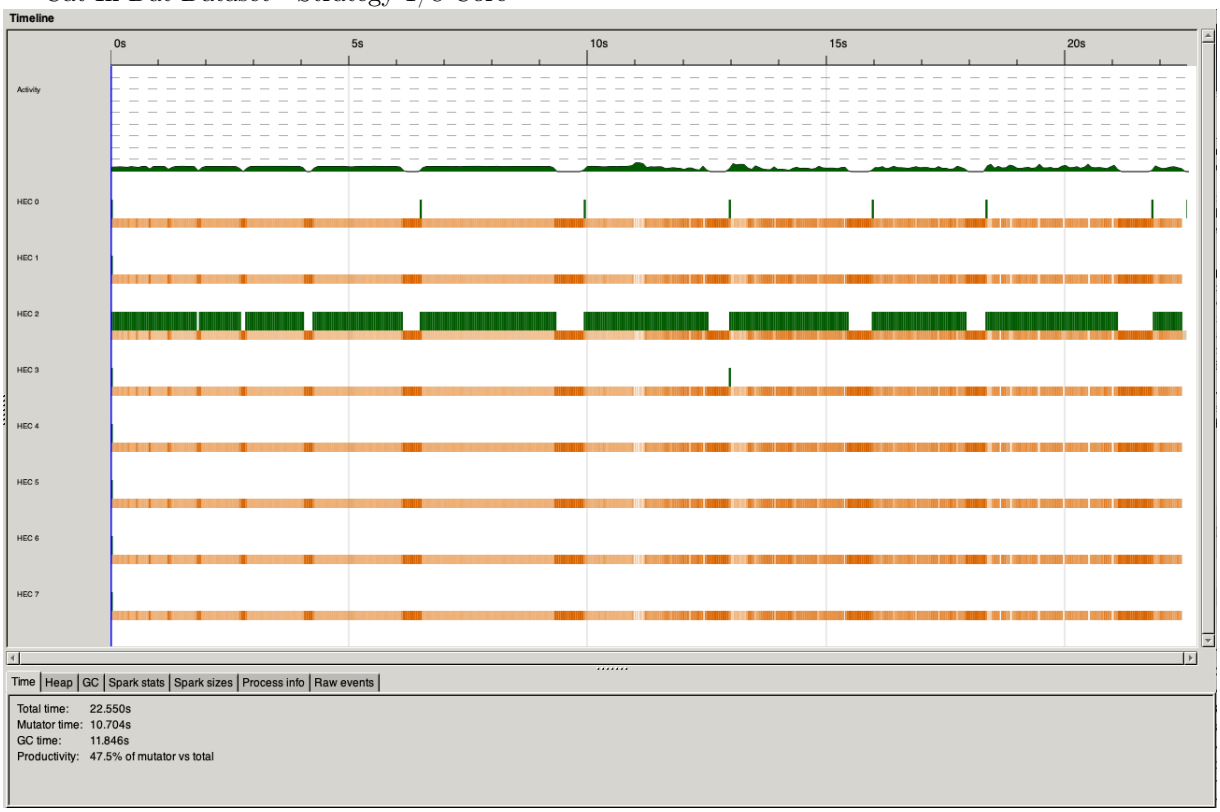
Cat-In-Dat Dataset - Strategy 1/2-Core



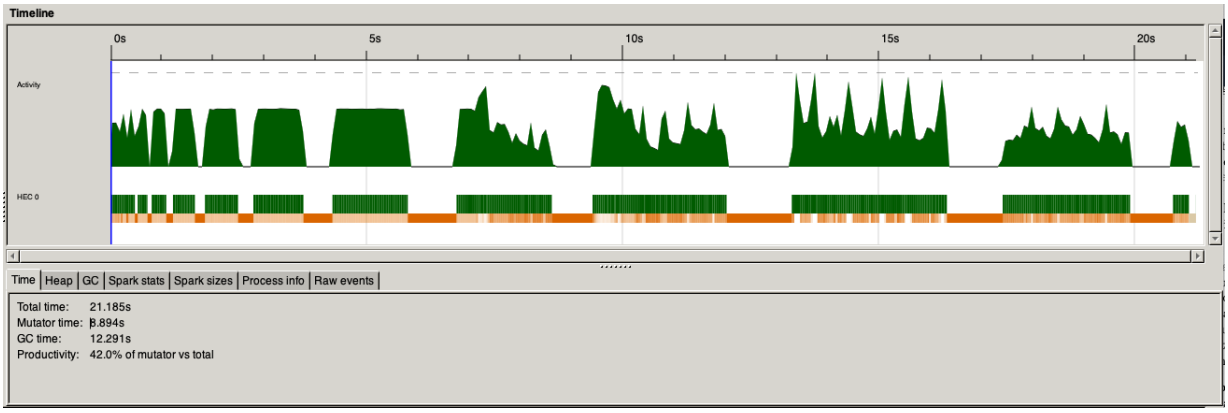
Cat-In-Dat Dataset - Strategy 1/4-Core



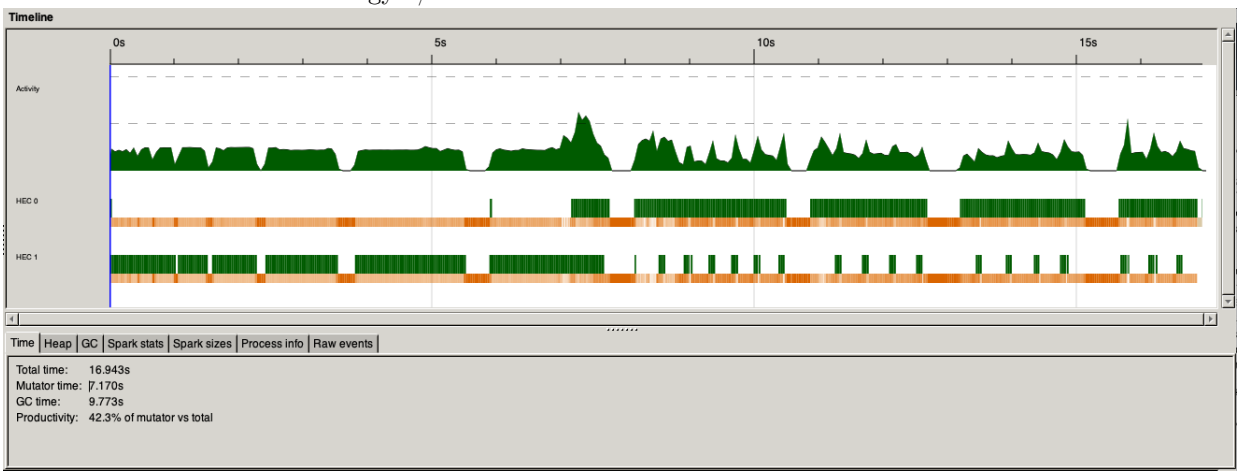
Cat-In-Dat Dataset - Strategy 1/8-Core



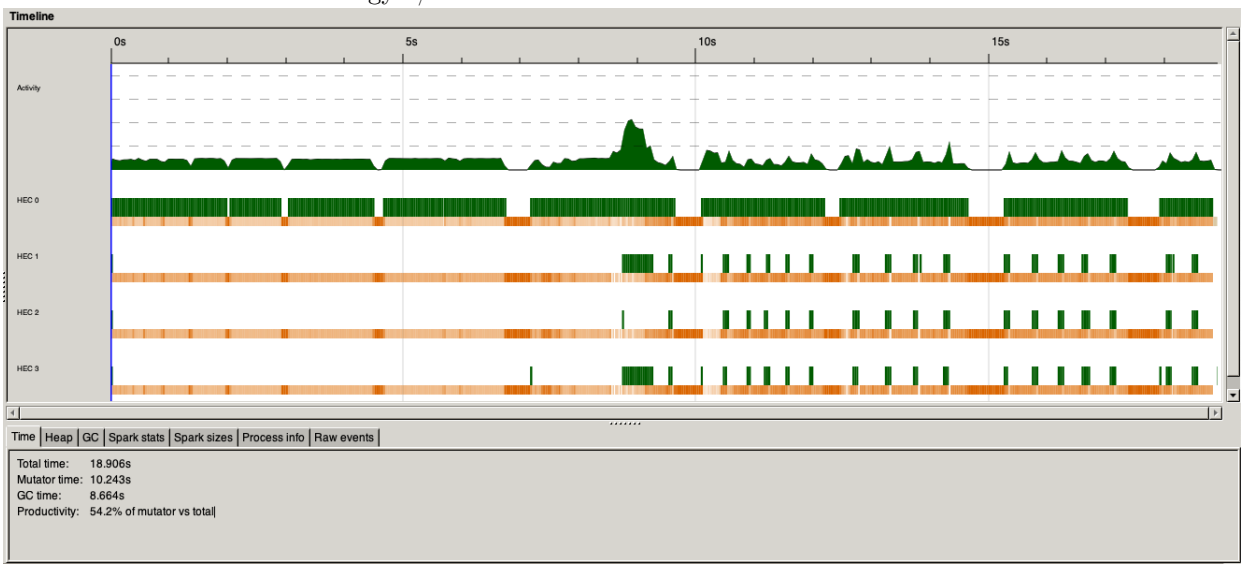
Cat-In-Dat Dataset - Strategy 2/1-Core



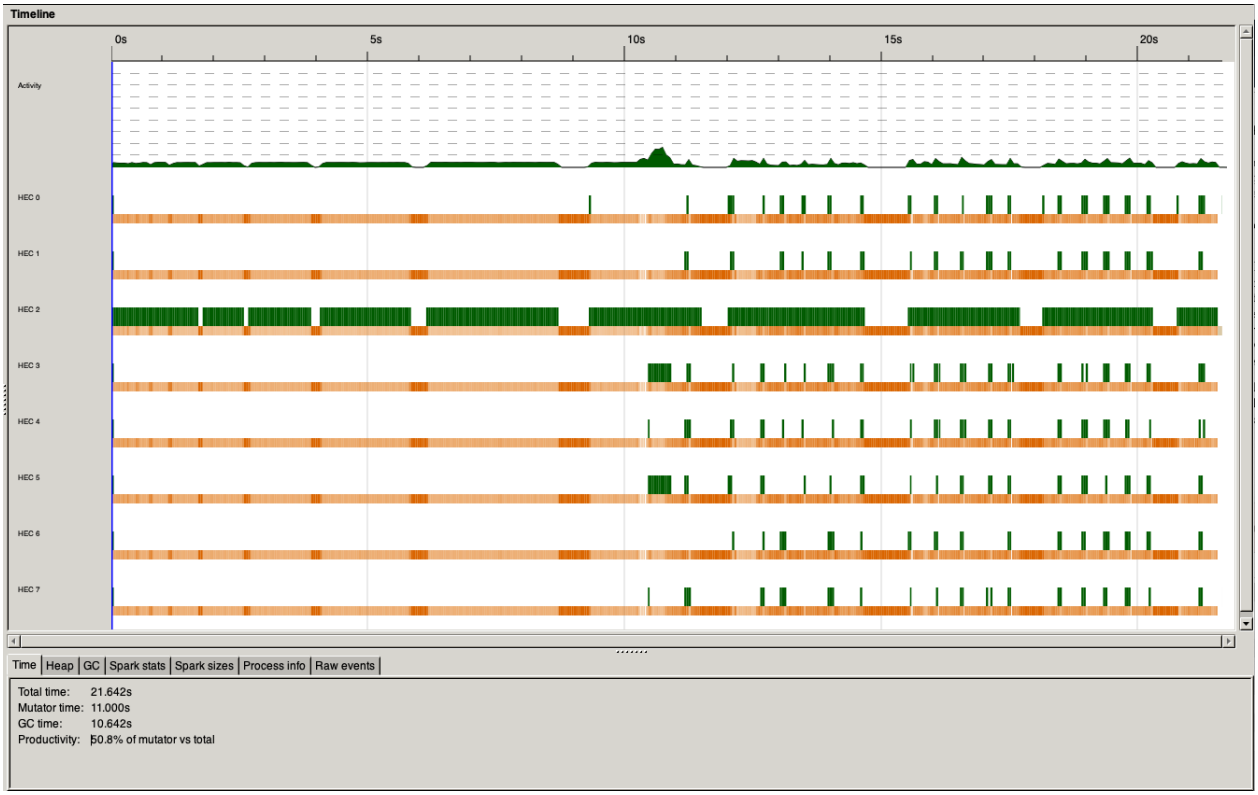
Cat-In-Dat Dataset - Strategy 2/2-Core



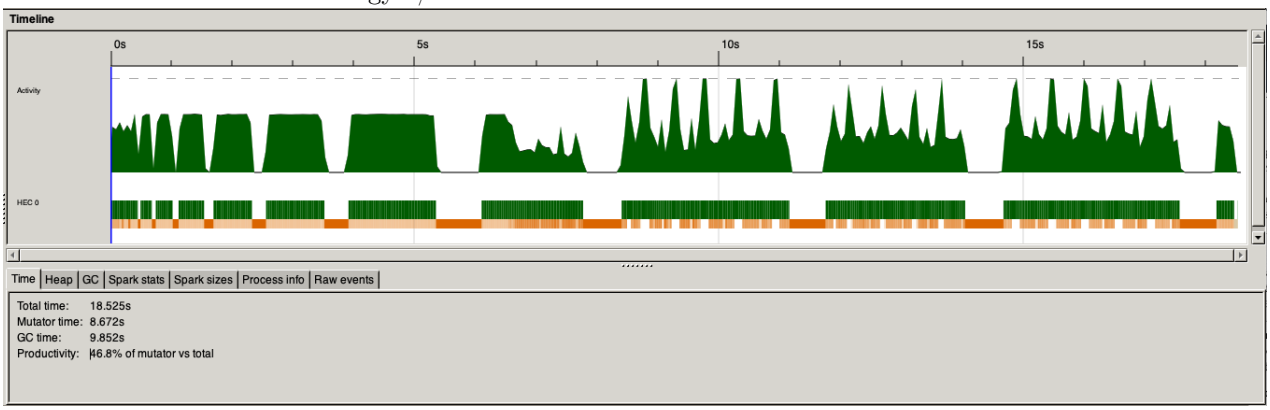
Cat-In-Dat Dataset - Strategy 2/4-Core



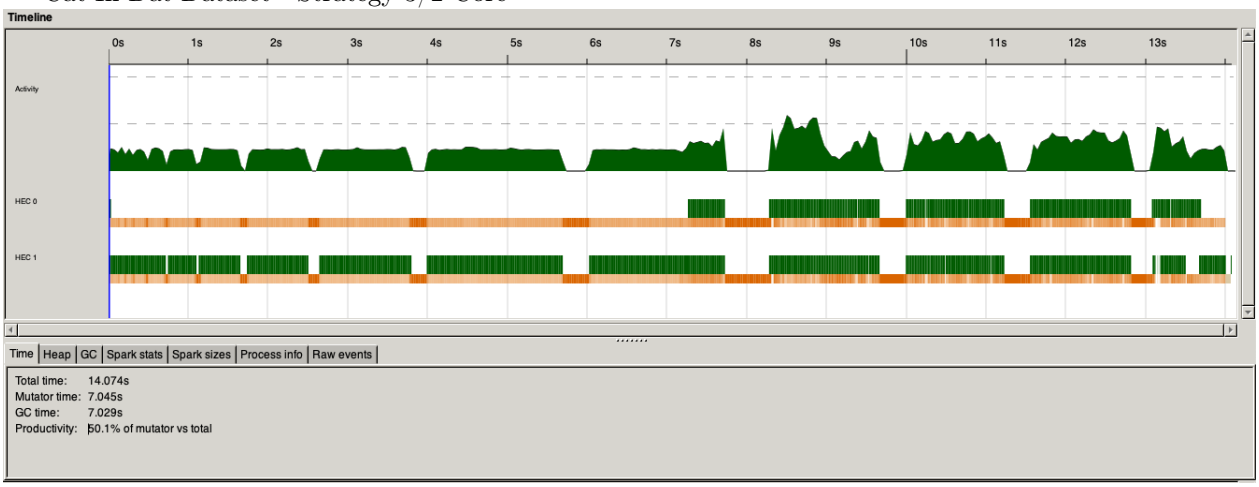
Cat-In-Dat Dataset - Strategy 2/8-Core



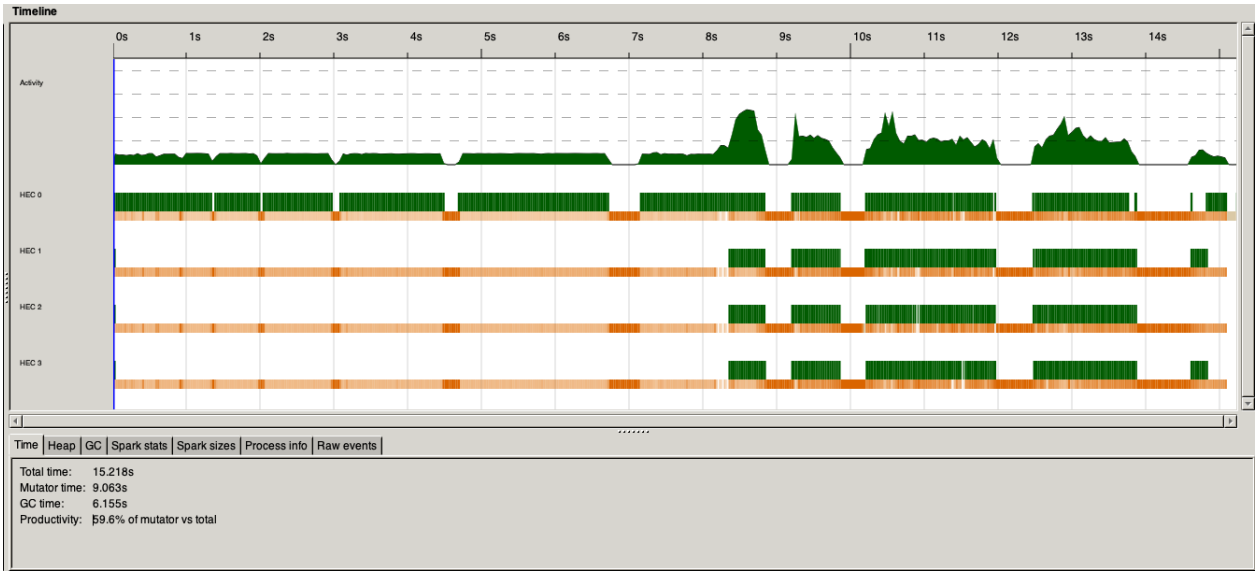
Cat-In-Dat Dataset - Strategy 3/1-Core



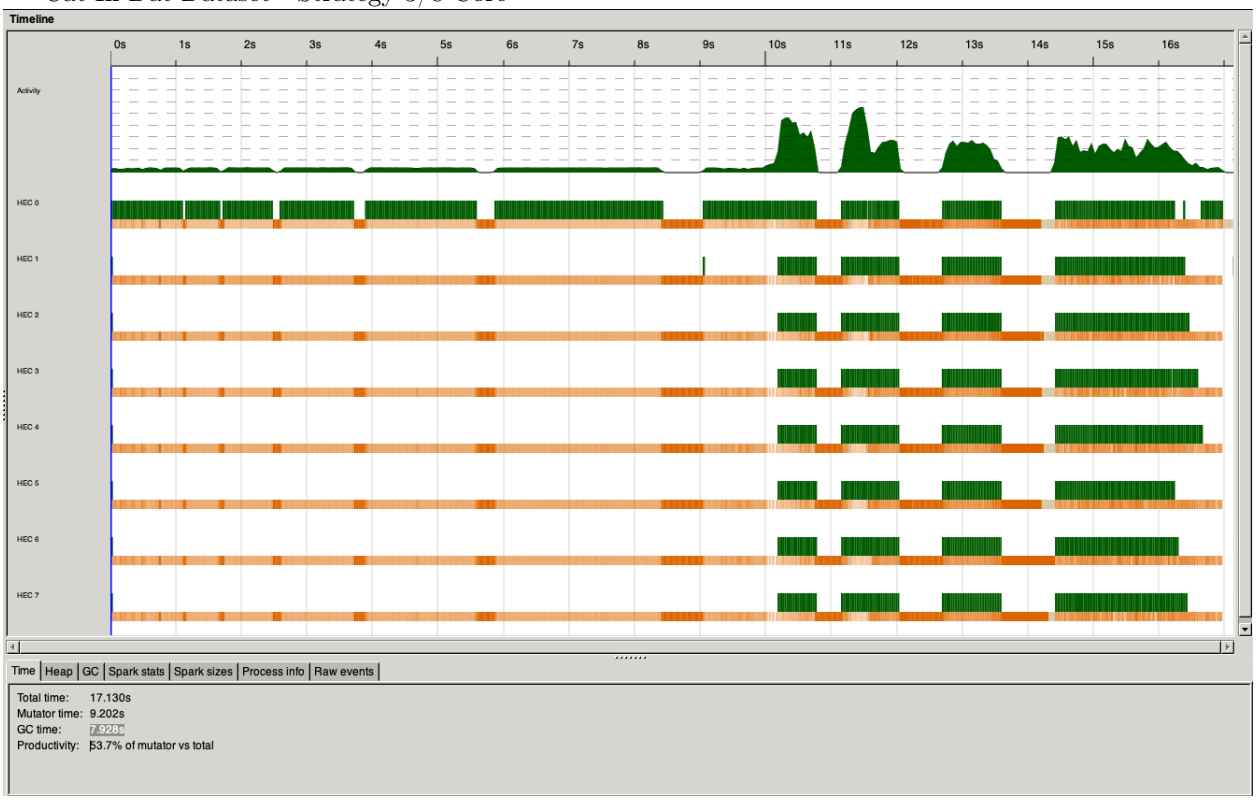
Cat-In-Dat Dataset - Strategy 3/2-Core



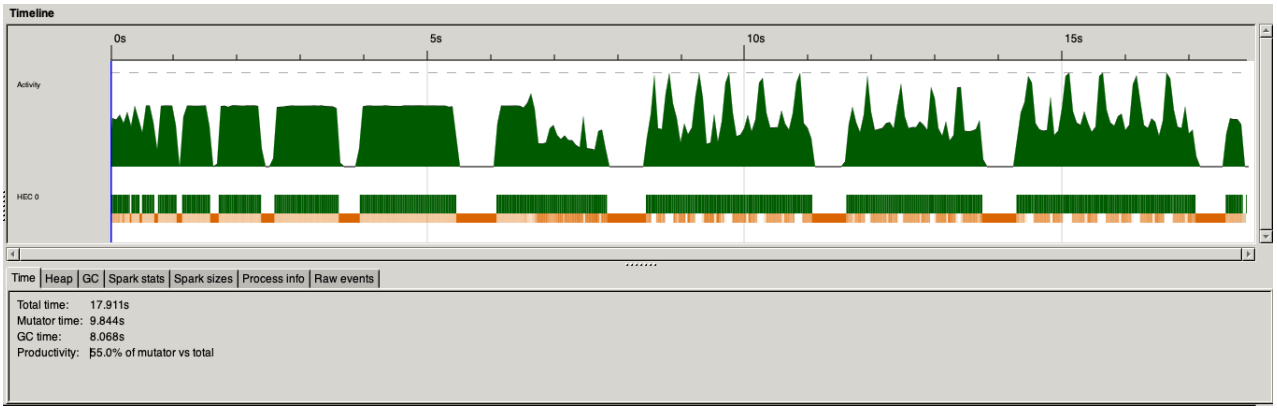
Cat-In-Dat Dataset - Strategy 3/4-Core



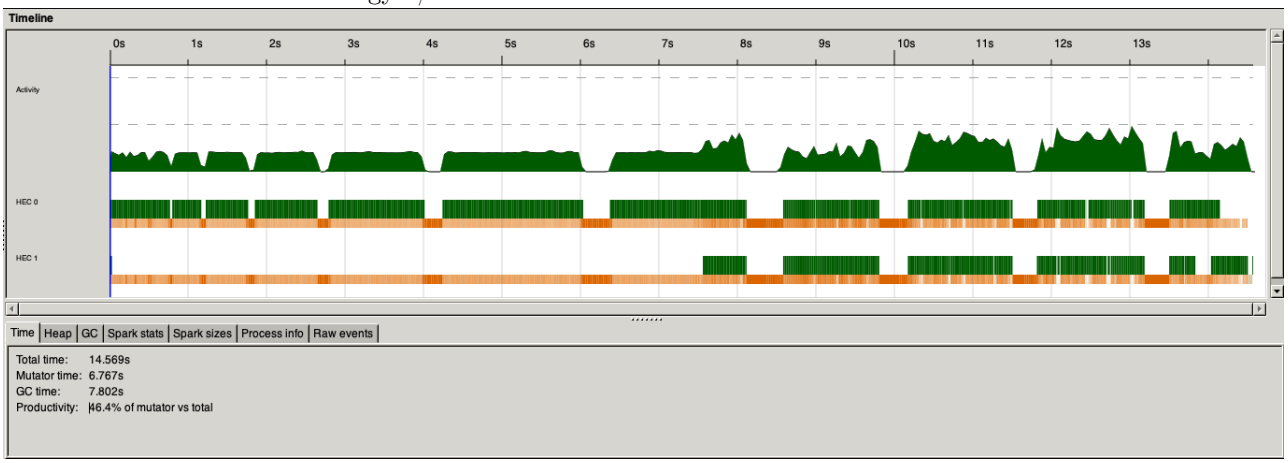
Cat-In-Dat Dataset - Strategy 3/8-Core



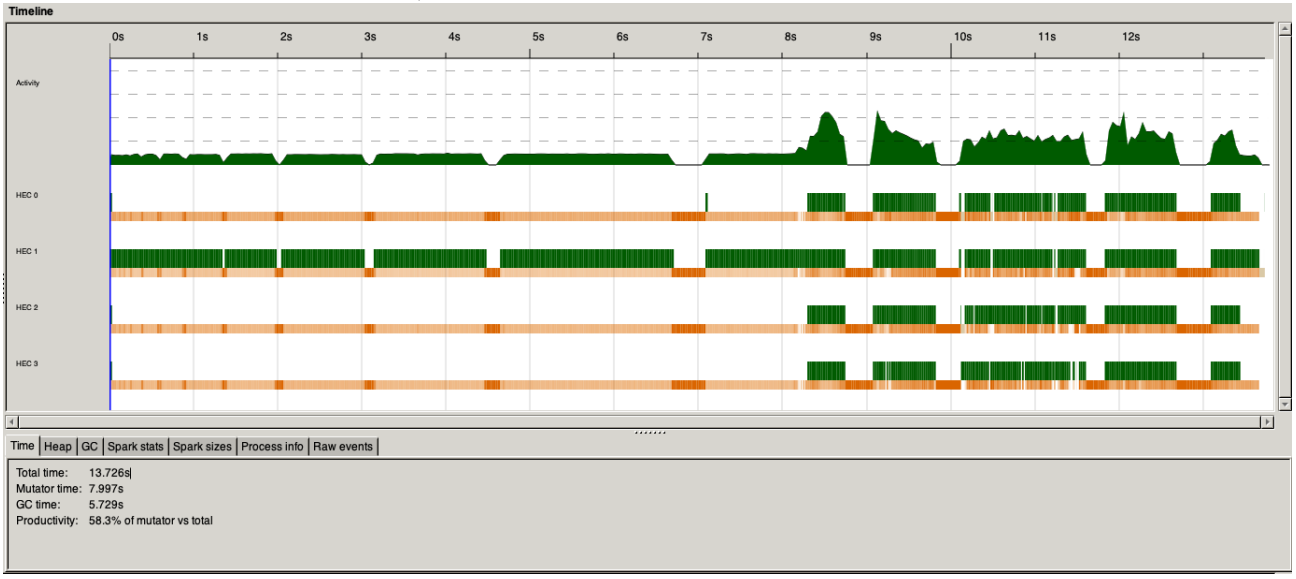
Cat-In-Dat Dataset - Strategy 4/1-Core



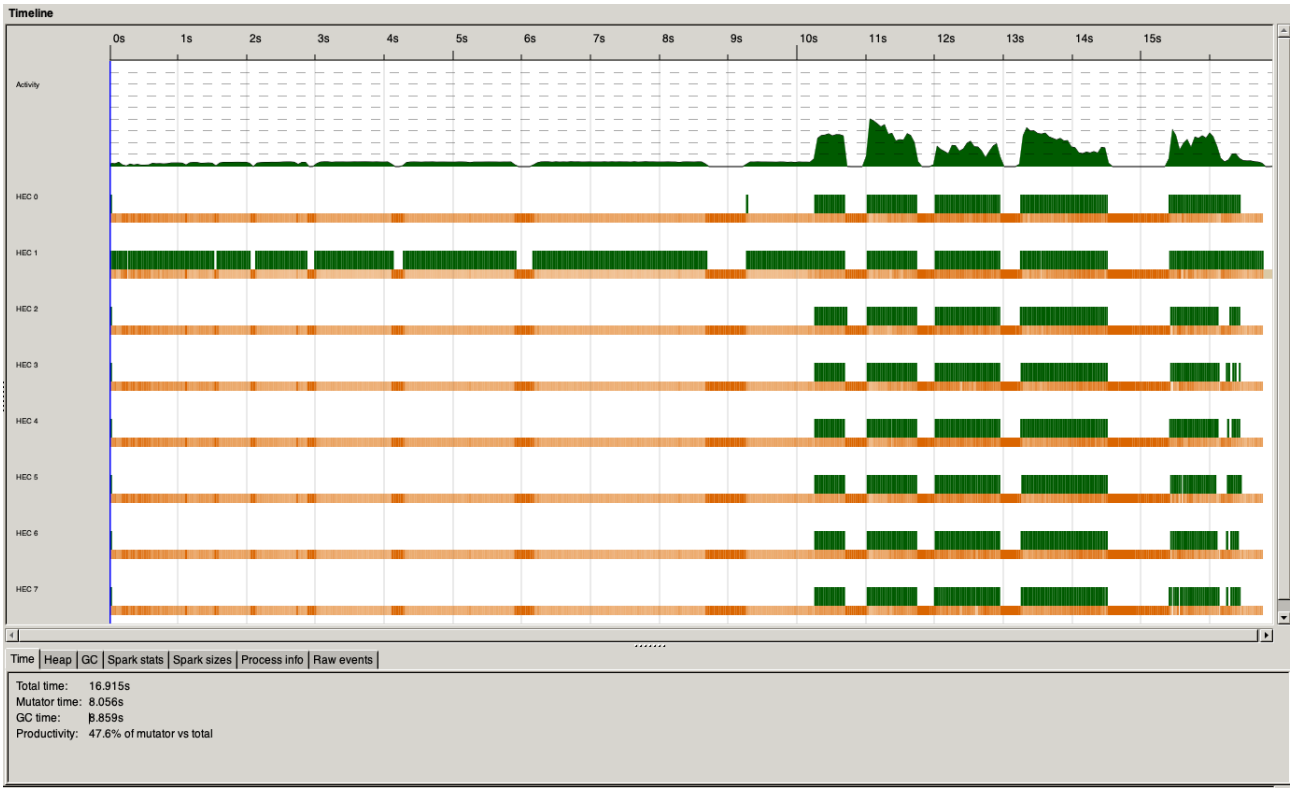
Cat-In-Dat Dataset - Strategy 4/2-Core



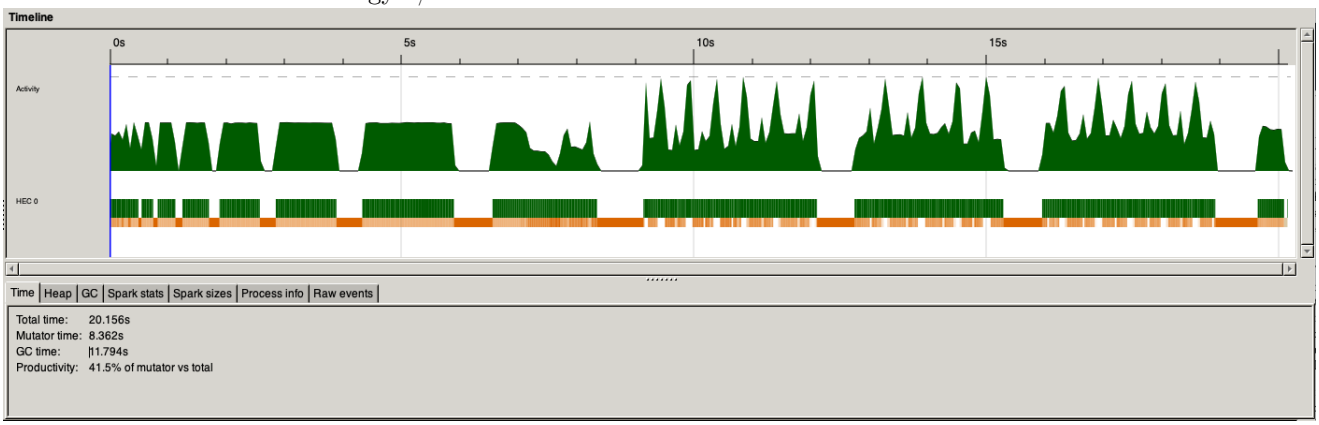
Cat-In-Dat Dataset - Strategy 4/4-Core



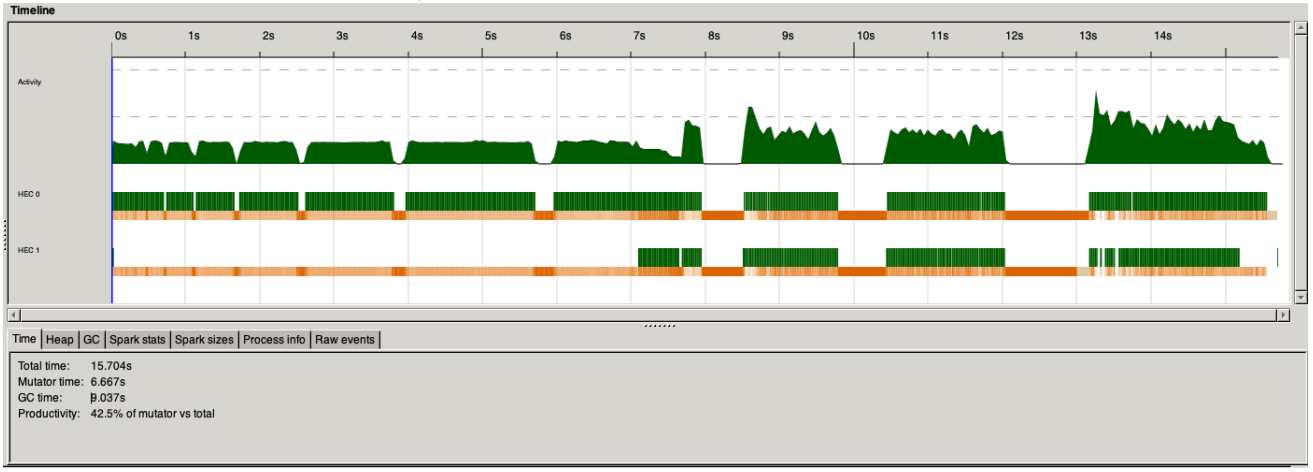
Cat-In-Dat Dataset - Strategy 4/8-Core



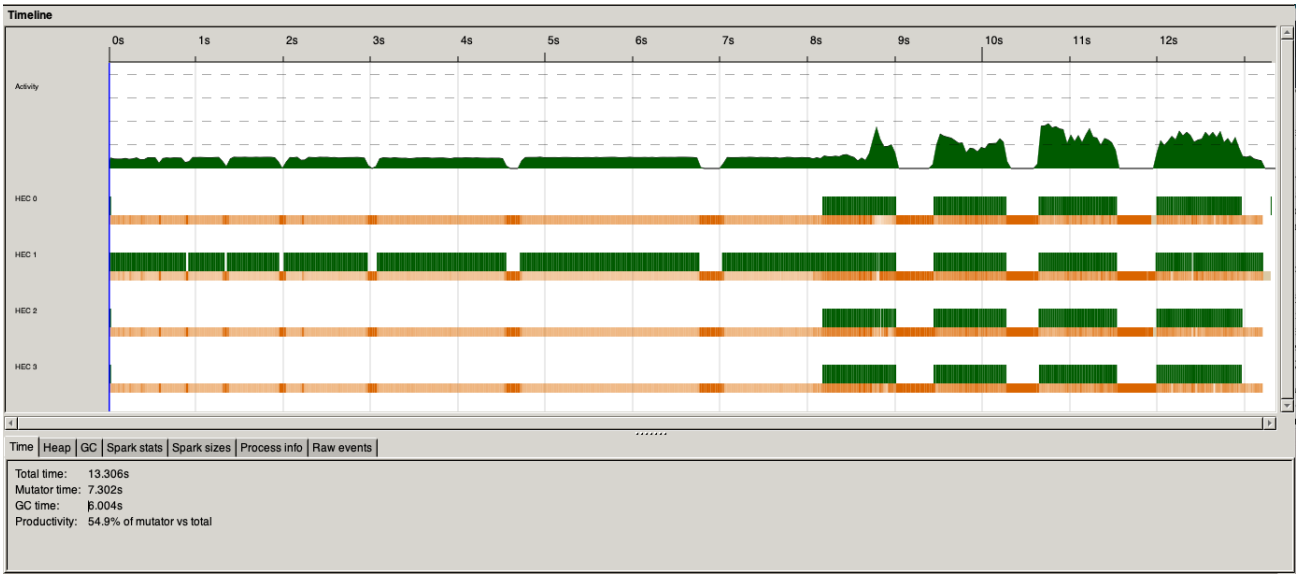
Cat-In-Dat Dataset - Strategy 5/1-Core



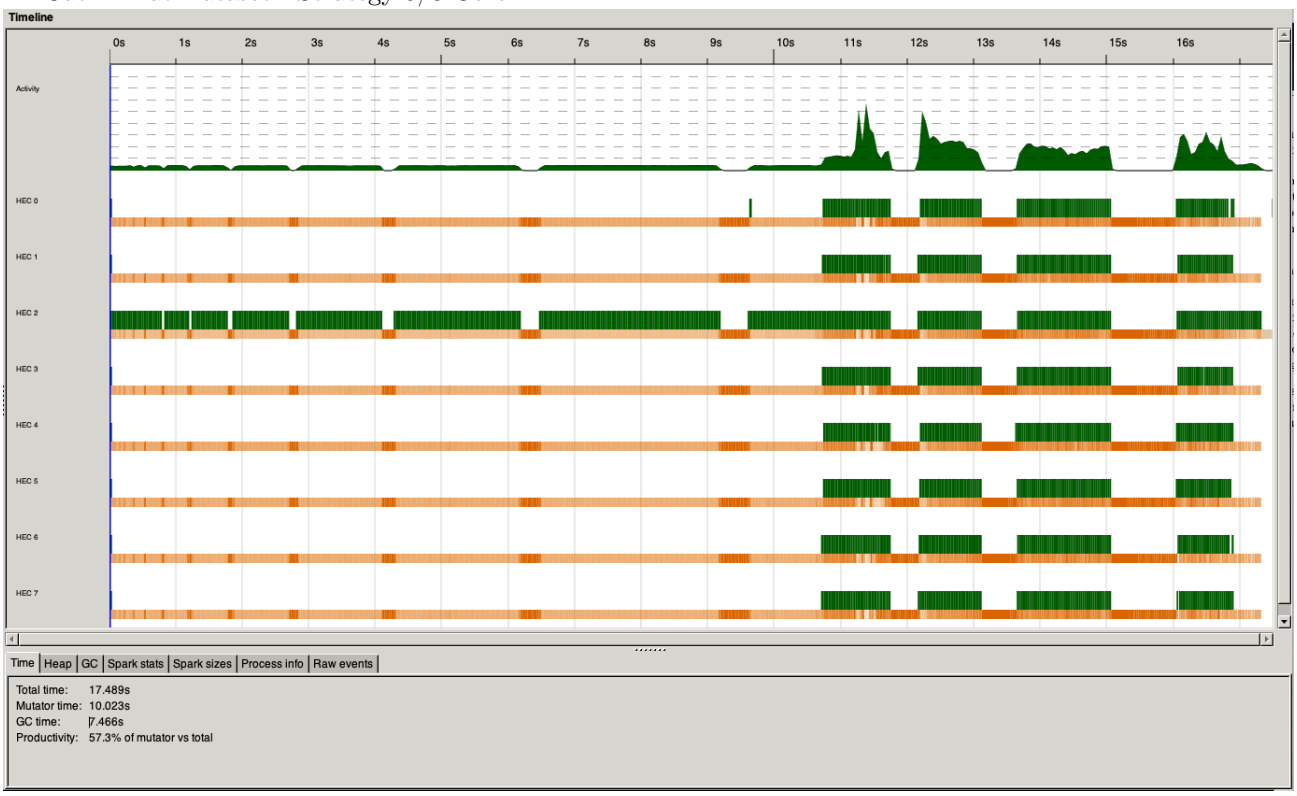
Cat-In-Dat Dataset - Strategy 5/2-Core



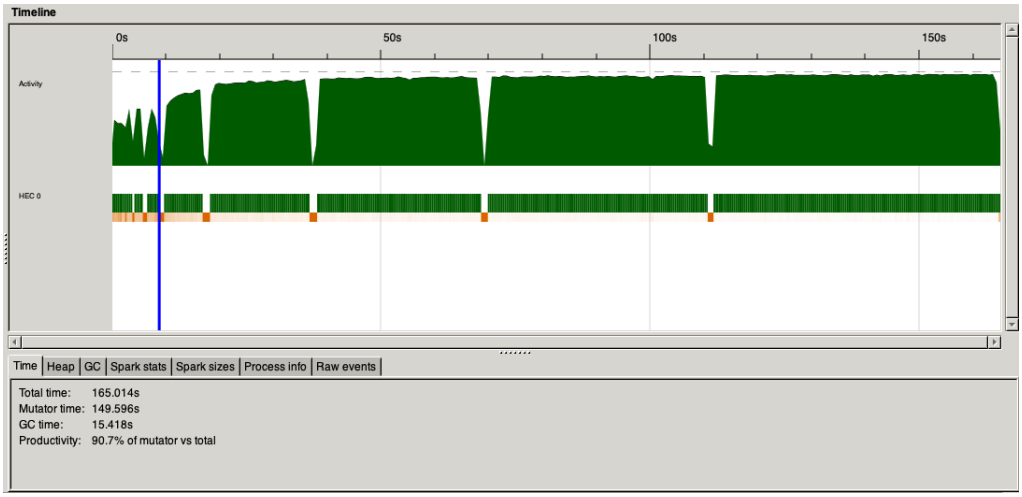
Cat-In-Dat Dataset - Strategy 5/4-Core



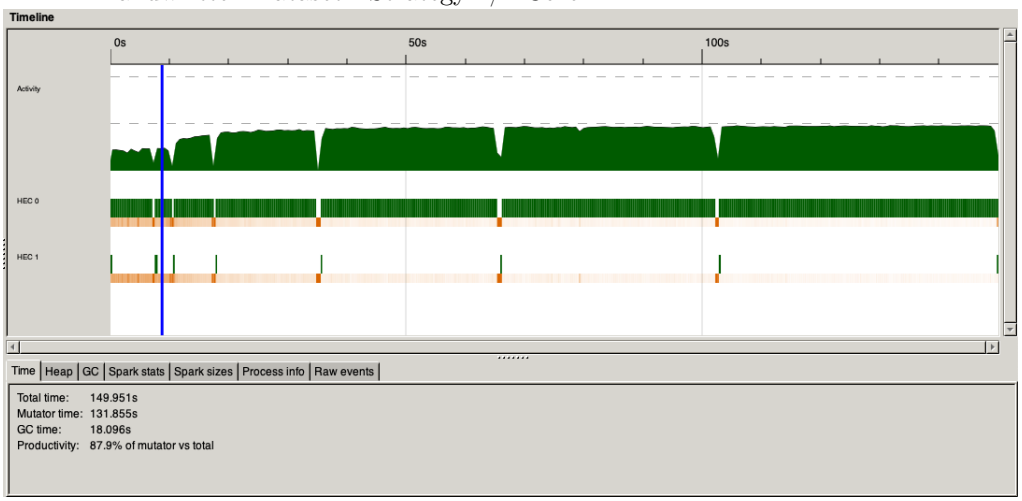
Cat-In-Dat Dataset - Strategy 5/8-Core



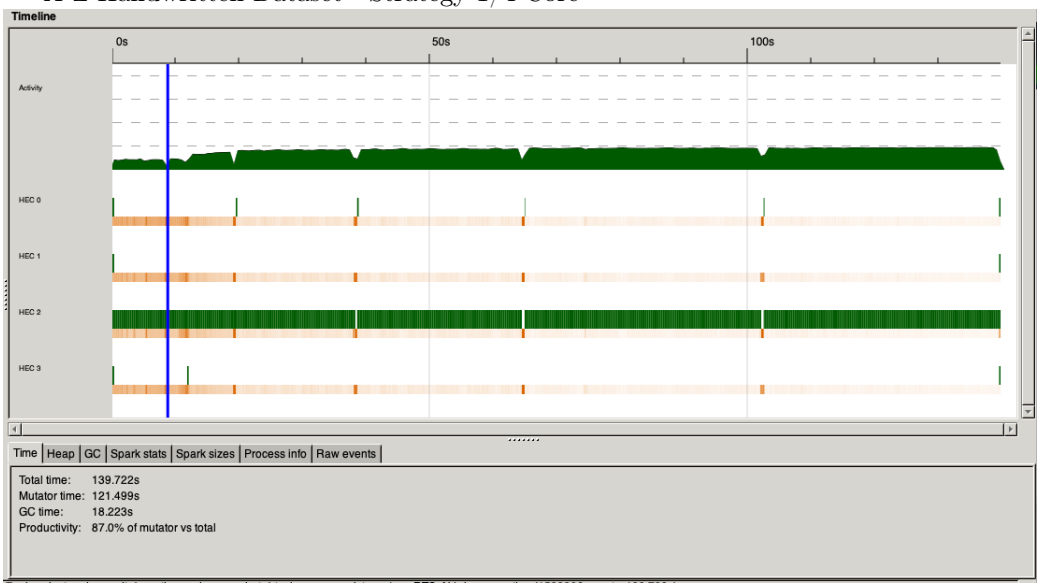
A-Z Handwritten Dataset - Strategy 1/1-Core



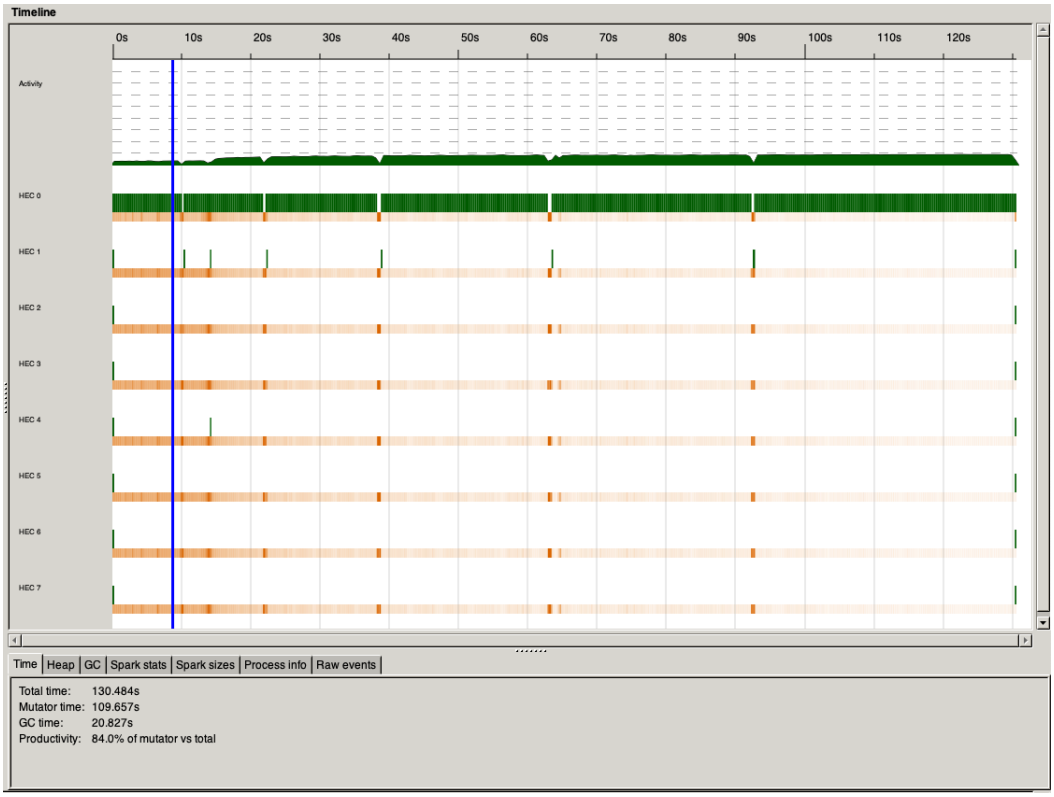
A-Z Handwritten Dataset - Strategy 1/2-Core



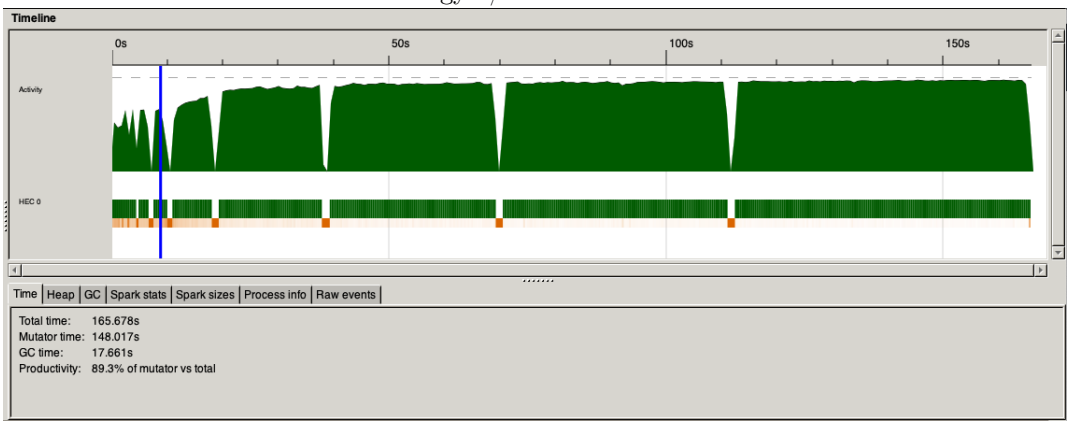
A-Z Handwritten Dataset - Strategy 1/4-Core



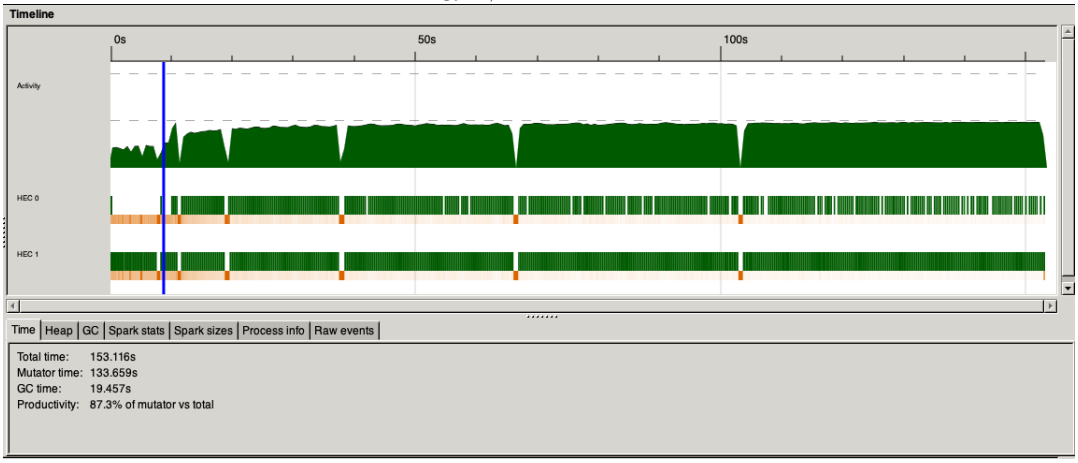
A-Z Handwritten Dataset - Strategy 1/8-Core



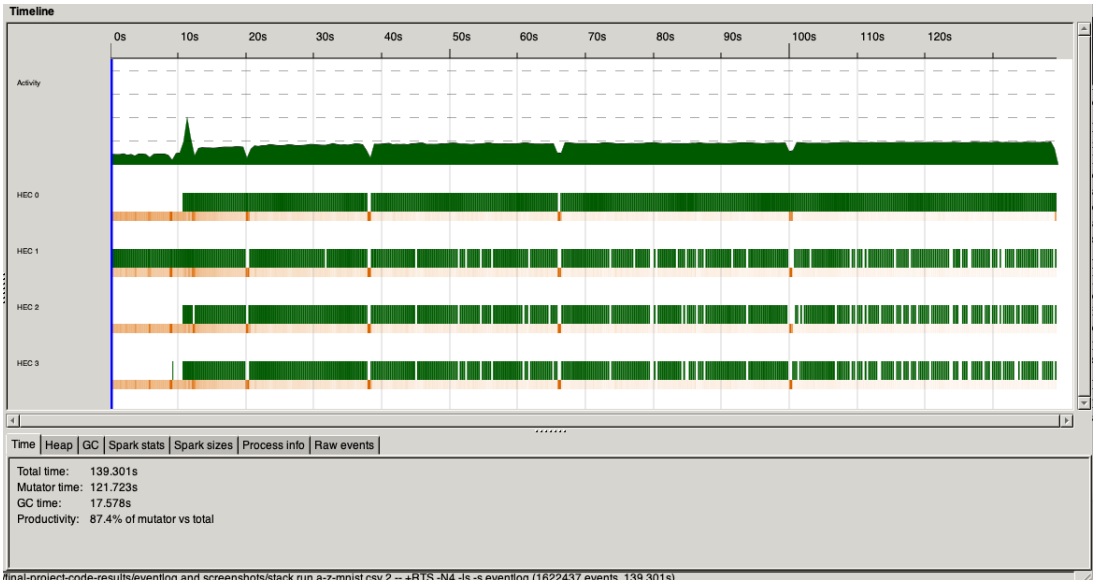
A-Z Handwritten Dataset - Strategy 2/1-Core



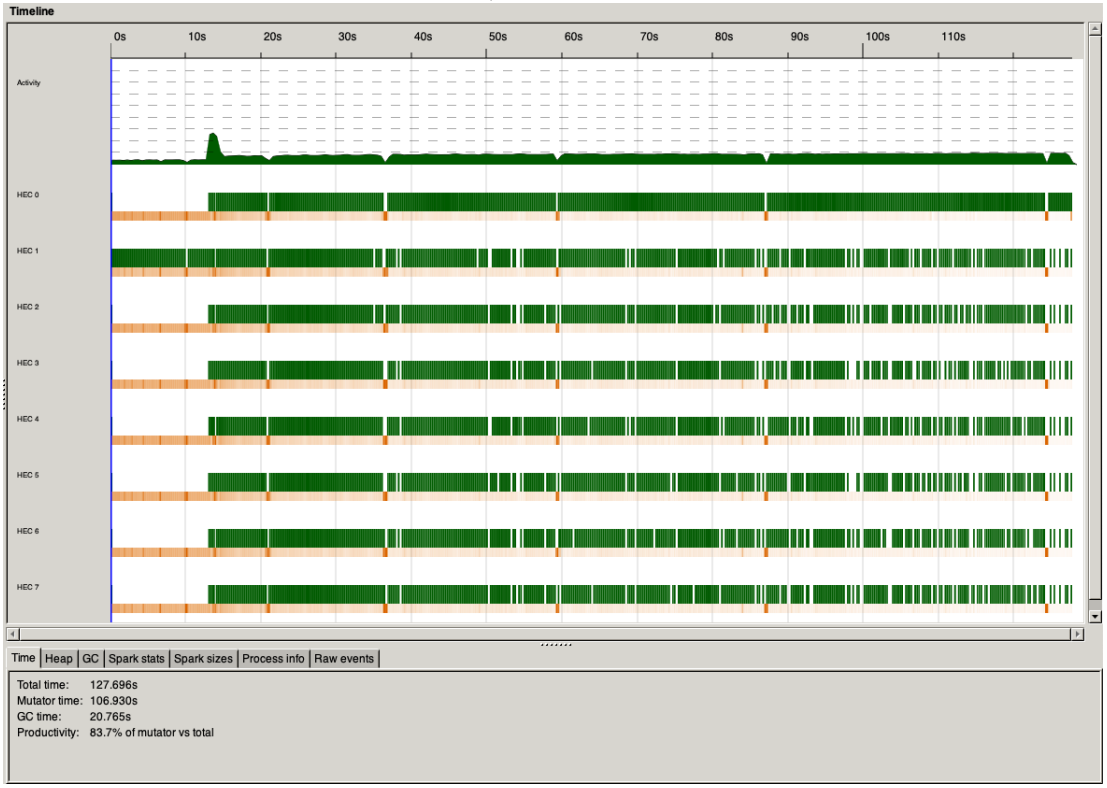
A-Z Handwritten Dataset - Strategy 2/2-Core



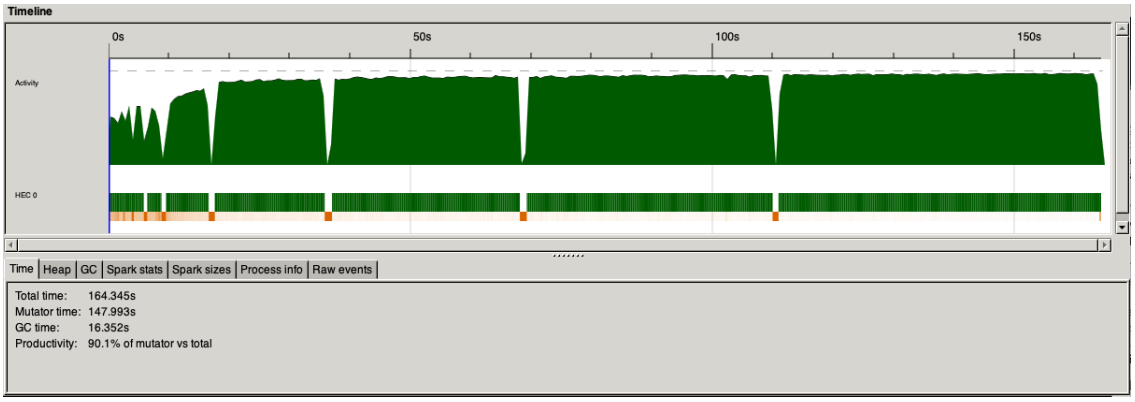
A-Z Handwritten Dataset - Strategy 2/4-Core



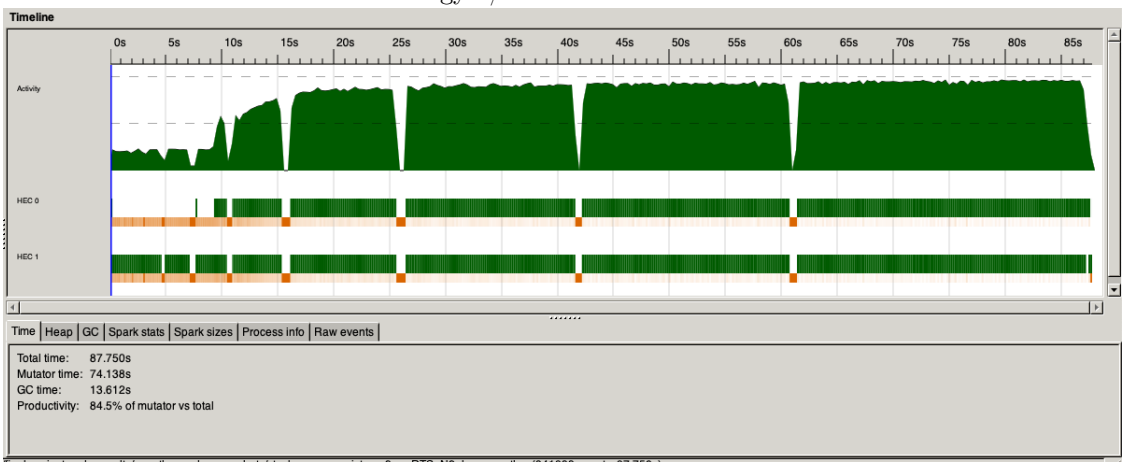
A-Z Handwritten Dataset - Strategy 2/8-Core



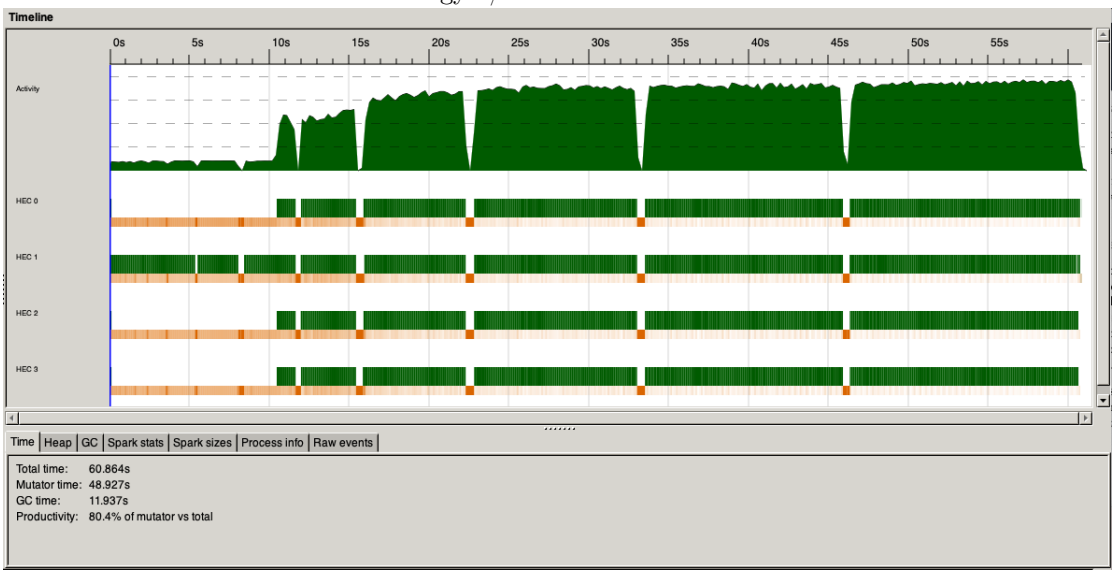
A-Z Handwritten Dataset - Strategy 3/1-Core



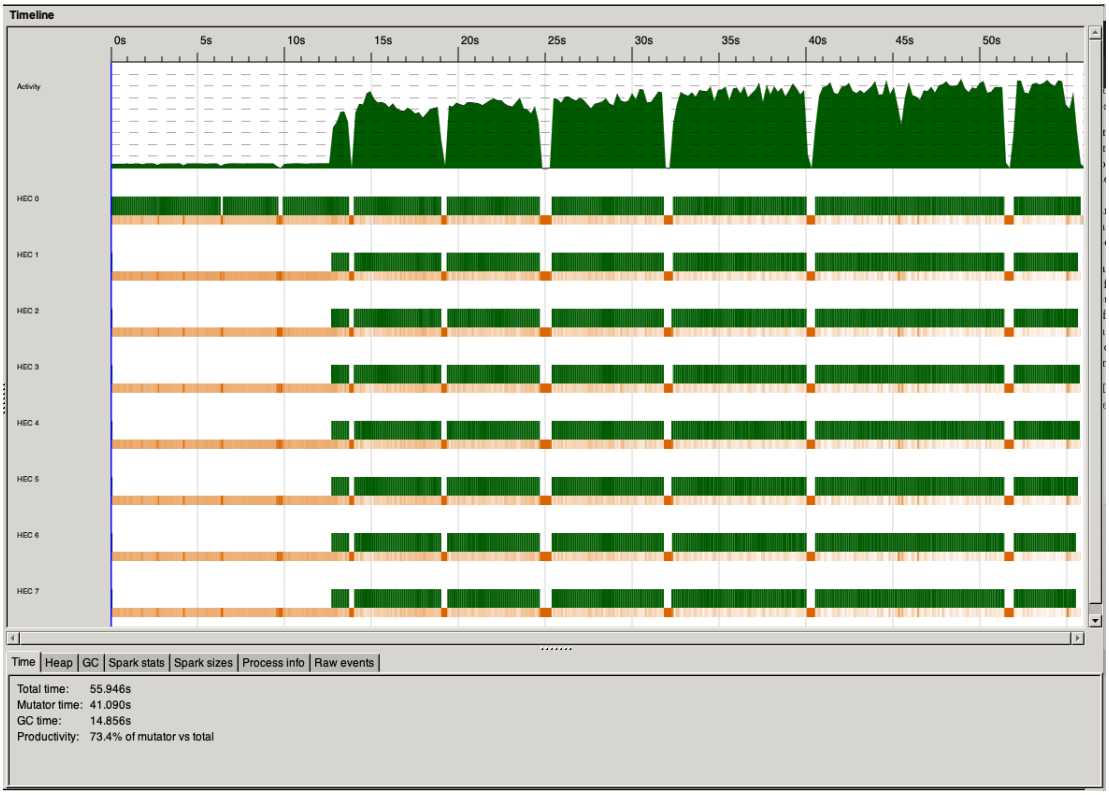
A-Z Handwritten Dataset - Strategy 3/2-Core



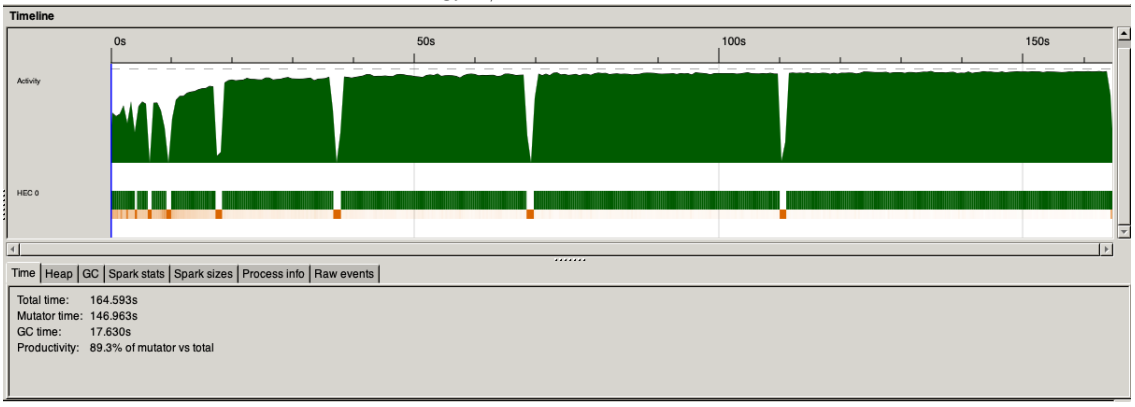
A-Z Handwritten Dataset - Strategy 3/4-Core



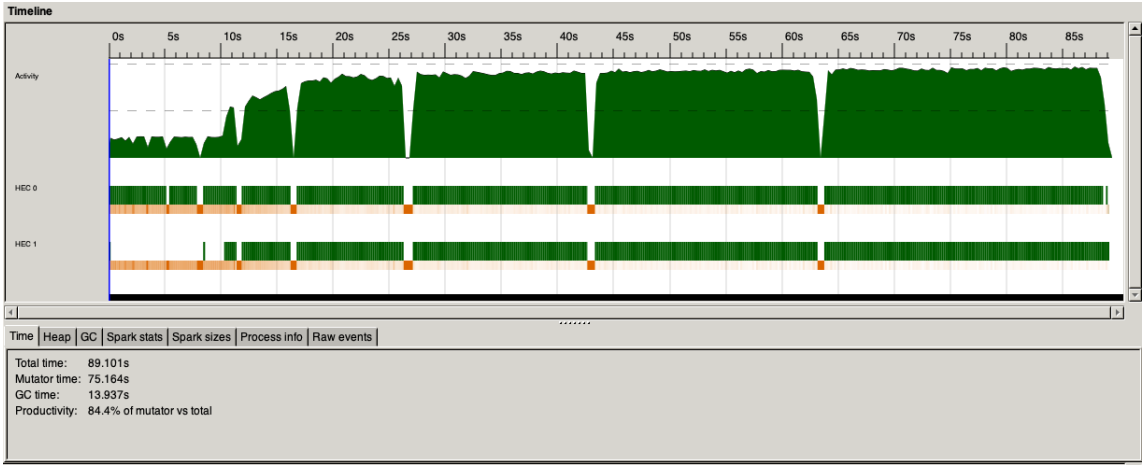
A-Z Handwritten Dataset - Strategy 3/8-Core



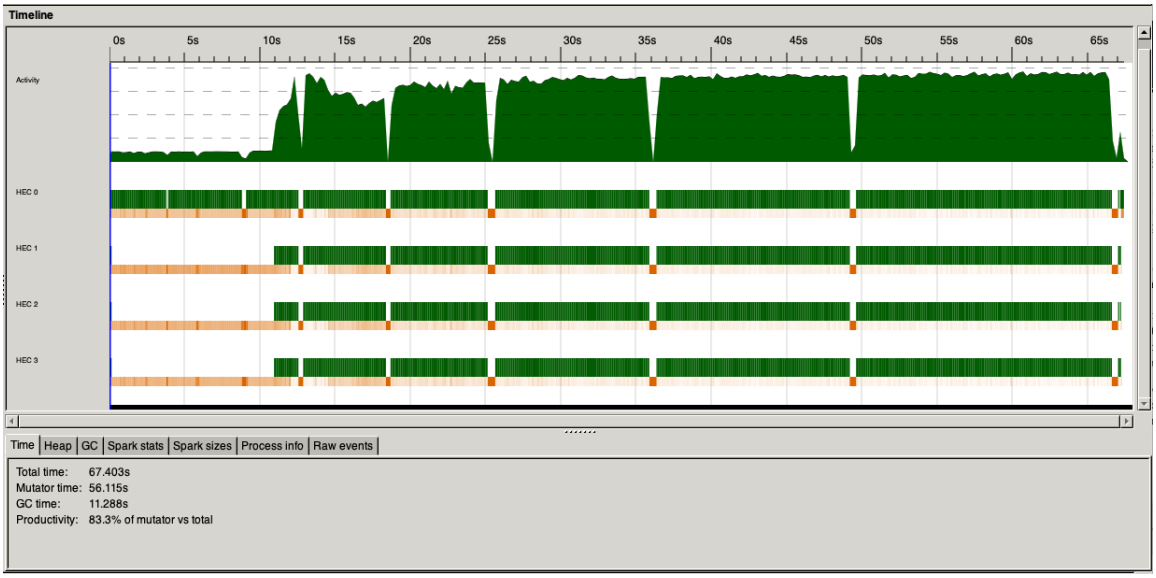
A-Z Handwritten Dataset - Strategy 4/1-Core



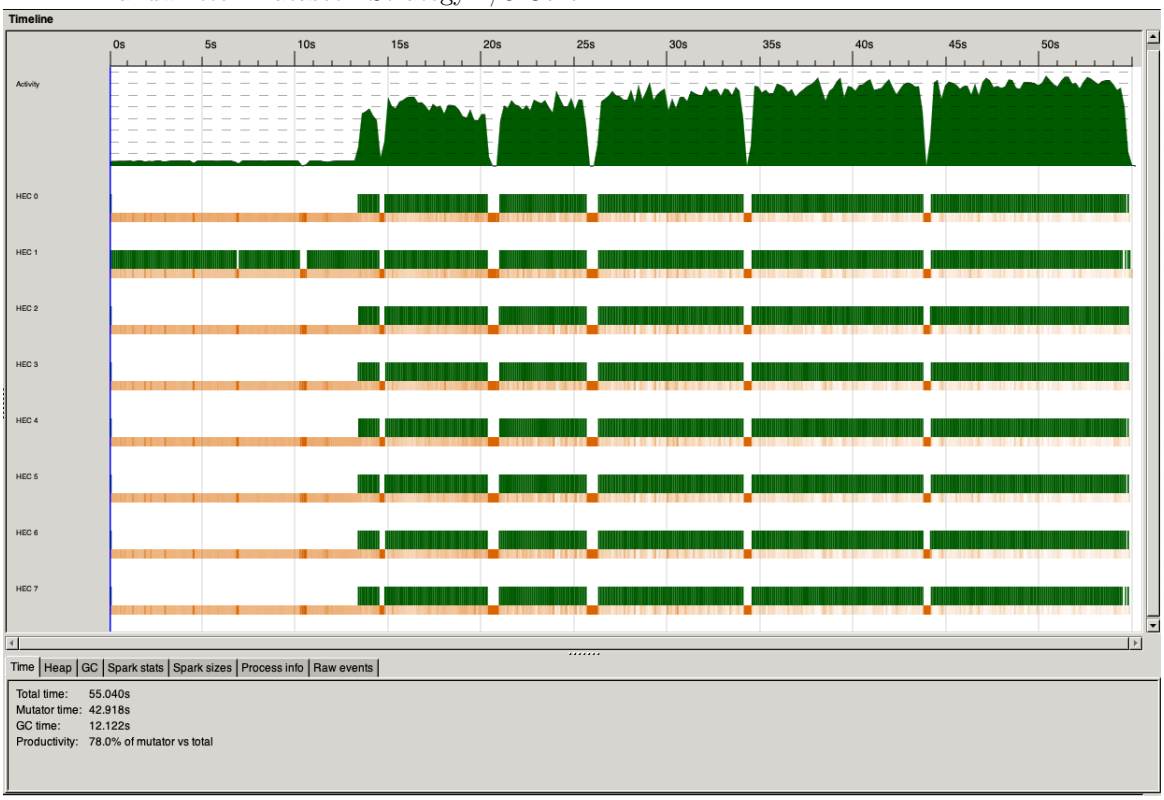
A-Z Handwritten Dataset - Strategy 4/2-Core



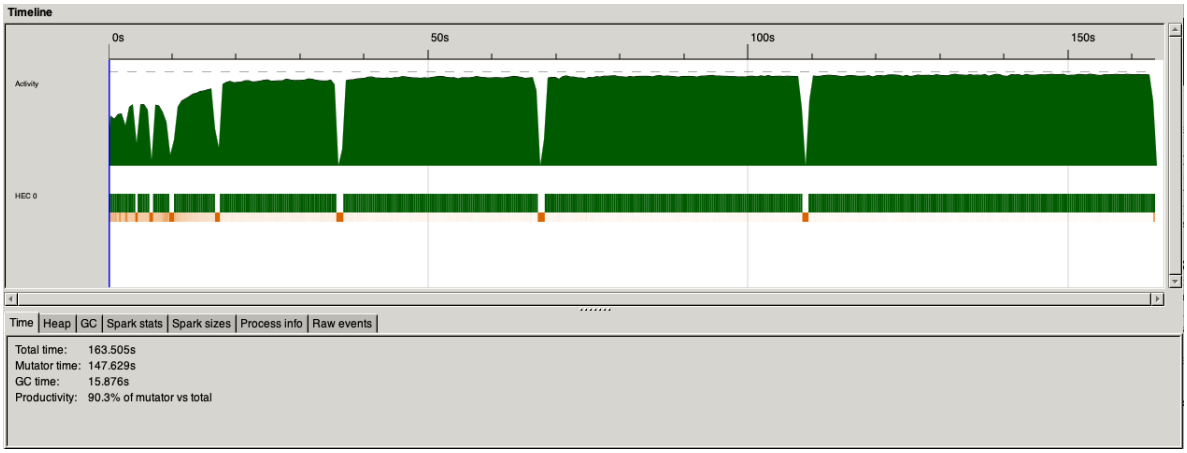
A-Z Handwritten Dataset - Strategy 4/4-Core



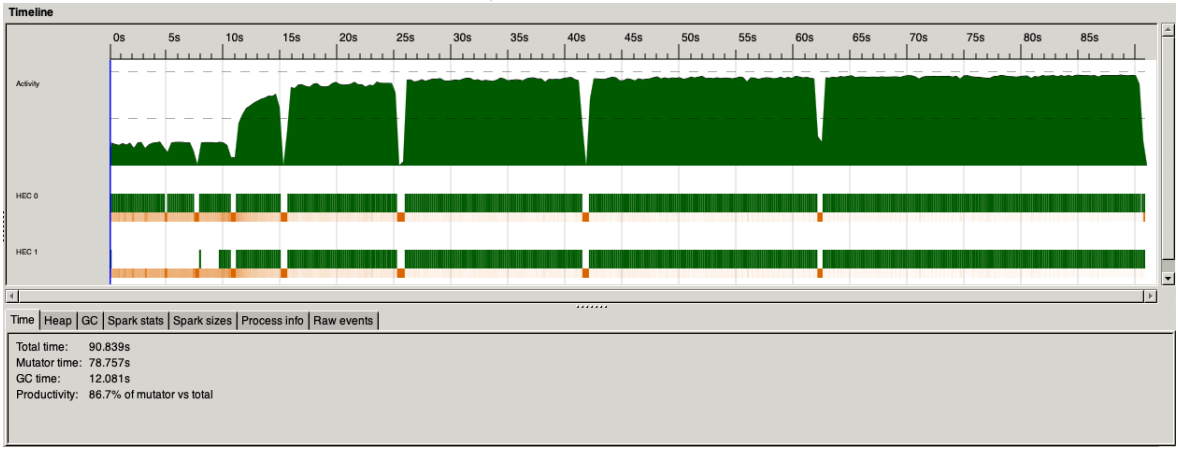
A-Z Handwritten Dataset - Strategy 4/8-Core



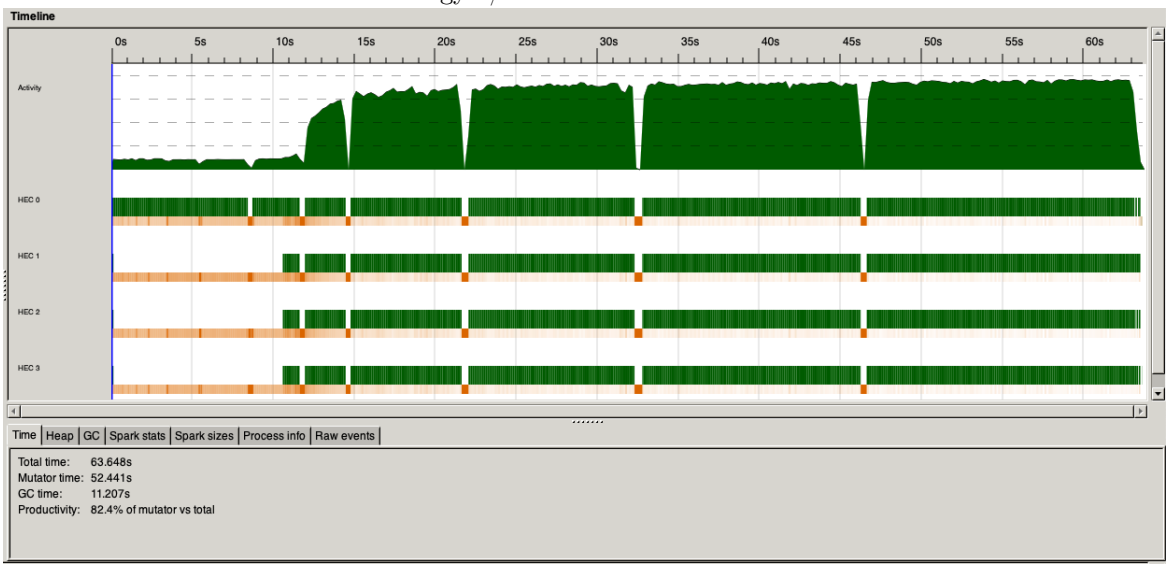
A-Z Handwritten Dataset - Strategy 5/1-Core



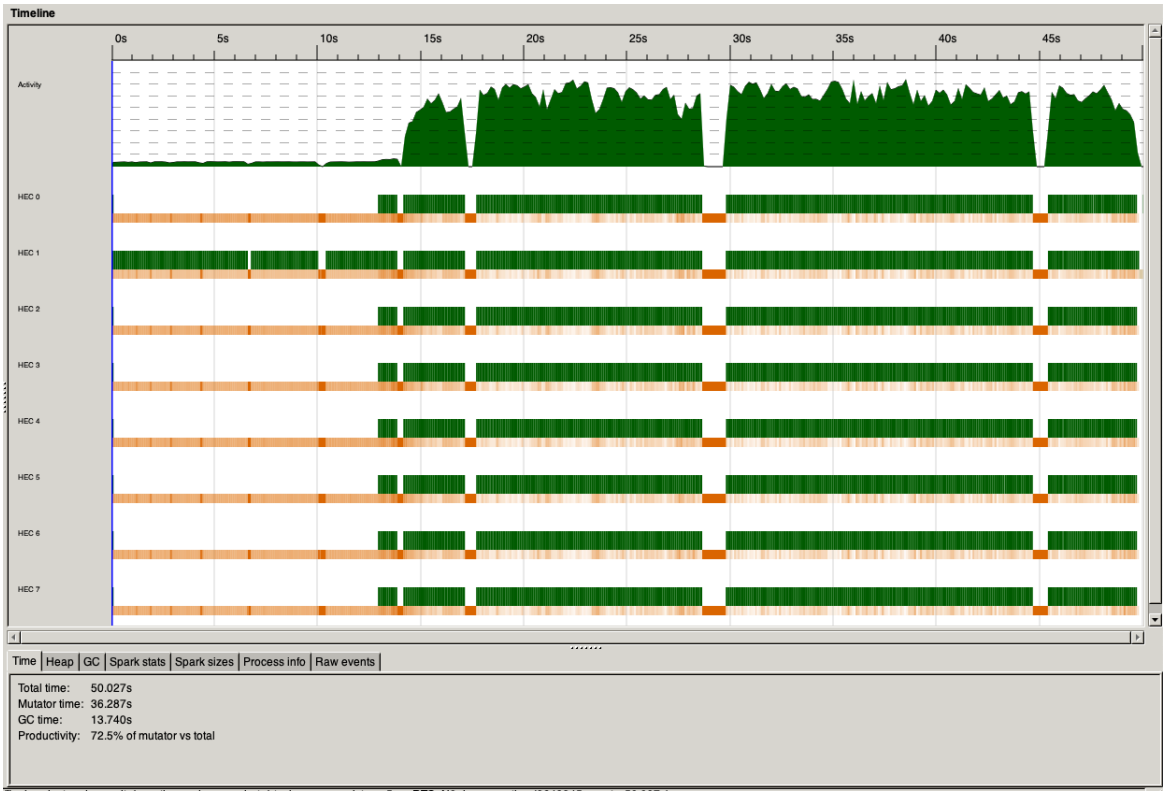
A-Z Handwritten Dataset - Strategy 5/2-Core



A-Z Handwritten Dataset - Strategy 5/4-Core



A-Z Handwritten Dataset - Strategy 5/8-Core


Timeline
0s
5s
10s
15s
20s
25s
30s
35s
40s
45s
Activity
HEC 0
HEC 1
HEC 2
HEC 3
HEC 4
HEC 5
HEC 6
HEC 7
Time
Heap
GC
Spark stats
Spark sizes
Process info
Raw events
Total time: 50.027s
Mutator time: 36.287s
GC time: 13.740s
Productivity: 72.5% of mutator vs total