The Parallel Stochastic Gradient Descent

Rishabh Ganesh (rg3478), Yealin Park (yp2611), Yue Sun (ys3535)

Problem Statement

Classifying data is a common task in machine learning. When we need to perform linear/non-linear classification problems on a large dataset, we usually use support vector machine algorithms, in which a data point is usually viewed as a p-dimensional vector, and we want to know whether we can separate a bunch of points with a hyperplane. There are many hyperplanes that can be used to classify the data, but we want to pick one such that the distance from it to the nearest data point on each side is minimized. It is usually difficult to determine the best hyperplane, since its goal is to minimize a function that may be analytically complex, so we can use gradient descent to find the optimal weights. In gradient descent, it requires a moving direction and a learning rate, then the algorithm can slowly arrive at the local or global minimum.

Algorithm

Overview

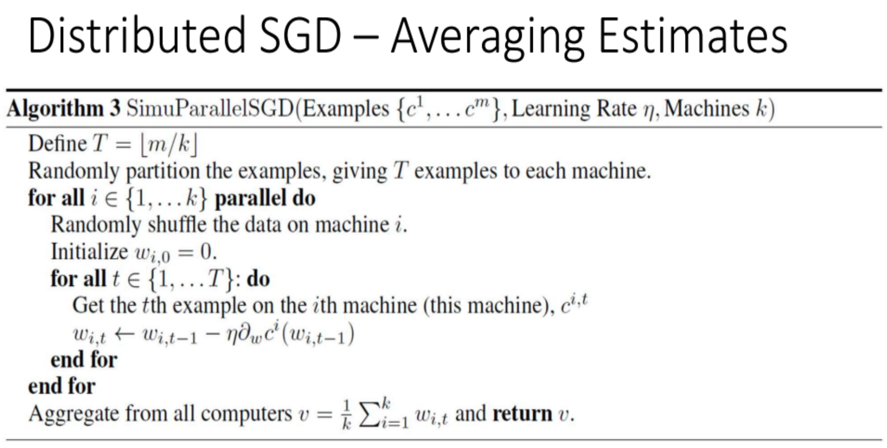
Gradient descent is used to minimize an objective function, in our case, the objective function of a support vector machine algorithm is

$$f(\mathbf{w}) \overset{\text{def}}{=} \underbrace{\frac{\lambda}{2}\|\mathbf{w}\|^2}_{\text{Regularization term}} + \underbrace{\frac{1}{m}\sum_{i=1}^{m} \max\{0, 1 - y_i\langle \mathbf{w}, \mathbf{x}_i\rangle\}}_{\text{Empirical loss}}$$

Standard gradient descent method would perform the following iterations:

$$w = w - \eta \nabla f(w) = w - \frac{\eta}{n} \nabla \sum_{i=1}^{n} f_i(w)$$

W is the model parameters, the weights of features , and η is the learning rate. Usually it requires multiple iterations until the weights start to converge to the optimal value.
Because gradient descent requires the calculation of the derivative to each datapoint in the dataset, and then sum them up, so the time complexity is O(Iteration * Data Dimension), one way to utilize the multiple cores of the computer to speed up the calculating process is stochastic gradient descent, instead of using the whole dataset to update the weight, it randomly pick a smaller batch of datasets, update the weights, then calculate the average weights, use the average as the starting point for the next iteration. The pseudo code is as following:



Distributed SGD – Averaging Estimates

Algorithm 3 SimuParallelSGD(Examples $\{c^1, \dots c^m\}$, Learning Rate η, Machines k)

```
Define $T = \lfloor m/k \rfloor$
Randomly partition the examples, giving $T$ examples to each machine.
for all $i \in \{1, \dots k\}$ parallel do
  Randomly shuffle the data on machine $i$.
  Initialize $w_{i,0} = 0$.
  for all $t \in \{1, \dots T\}$: do
    Get the $t$th example on the $i$th machine (this machine), $c^{i,t}$
    $w_{i,t} \leftarrow w_{i,t-1} - \eta \partial_w c^i(w_{i,t-1})$
  end for
end for
Aggregate from all computers $v = \frac{1}{k} \sum_{i=1}^{k} w_{i,t}$ and return $v$.
```

Implementation

We started by using a sequential implementation of stochastic gradient descent on GitHub that runs on a small data set. The Train.hs file initially contained the sequential implementation, where three type aliases were defined:

```
type Sample = R.Sample Int
type Model = (FullVector, Double, Double)
type PredLoss = (Double, Double, Double)
```

The Sample type represents one data point, and it is defined by both the sample defined in Read.hs, which consists of a vector of weights and a Double for the response, and an Int. The Model type defines the model that our algorithm determines as the best fit for the given data set. Model is a 3-tuple that has a Vector which represents the weights for the parameters for the curve that fits the data set, a Double that represents the scaling factor for the weights, and another Double which represents the offset term for the model. The PredLoss type tracks the value for the estimated error of our model. It is a 3-tuple with three Double values representing the value of the loss function, the cost of the model, and the predicted error value.

Our goal for this project is to find ways to parallelize this implementation by looking for relevant parts of the sequential implementation from GitHub that would likely speed up the algorithm, and use techniques learned during the course to make these computations run in parallel. We achieve the best results from parallelizing the `train` function, which performs iterations of the stochastic gradient descent algorithm and returns a model that makes accurate predictions on the input data set.

```
train :: Loss                             -- loss and dloss function
     -> [Sample]                         -- list of samples
     -> Double                           -- regularizer
     -> Int                              -- epochs
     -> Model
train l x lambda epochs = foldl (go) wParam0 [1..epochs]
 where
   go wParam _  = trainMany (dloss l) x lambda wParam eta

   dim = R.dimSample x
   wParam0 = initParam dim

   n = min 1000 . length $ x

   n=length $ x
   fTrain =  trainMany (dloss l) (take n x) lambda wParam0
   fTest =  testMany (loss l) (take n x) lambda
   eta0 = determineEta0 (fTest . fTrain) (2, 1, 2)
   eta = map (\t -> eta0 / (1 + lambda * eta0 * t)) [0..]
```

Our main focus is this implementation of the train function, which takes Loss function l, list of Samples x, a Double as a lambda value for regularization, an Int number of epochs or iterations, and returns a classification Model. It trains the Model using the trainMany function, which takes a derivative of the loss function, a list of samples, a regularization value, and the Model at the current iteration. We removed the testing functionality when parallelizing this problem because the bulk of the computation happens during training epochs. Another change we make between the sequential and parallel implementations is the determination of the Eta value, which represents the learning rate, or the rate of descent at which the Model parameters change after each SGD iteration. Eta can be tuned in various ways as the algorithm progresses, but we choose to keep it constant. Note that the SGD implementation is found in the `trainMany` function below, which uses the `trainOne` function.

```
trainMany :: (Double -> Double -> Double)      -- dloss function
         -> [Sample]                          -- list of samples
         -> Double                            -- regularizer
         -> Model                             -- current model parameter
         -> [Double]                          -- sgd gain eta for each iteration (or
sample)
         -> Model

trainMany dloss x lambda wParam0 eta= foldl go wParam0 $ zip x eta
 where
   go wParam (xt, etat) = trainOne dloss xt lambda wParam etat


trainOne :: (Double -> Double -> Double) -> Sample -> Double -> Model -> Double ->
Model
trainOne dloss (x, y) lambda (w, wDiv, wBias) eta = (w'', wDiv'', wBias')
 where
   s = (dotFS w x) / wDiv + wBias
   d = dloss s y
   wDiv' = wDiv / (1 - eta * lambda)
   (w', wDiv'') = renorm w wDiv'
   w'' = addFS w' . mul x $ eta * d * wDiv''
   wBias' = wBias + d * eta * 0.01
```

`trainOne` takes the derivative of the loss function, one Sample, one regularizer, the current Model, and the learning rate Eta, and returns the model after one epoch. The SGD computation happens when calculating the `s` value, which takes the dot product of the Model weights `w` and the current sample, normalizes it by dividing by `wDiv`, and adds `wBias` as the bias term. This value is used to find `d`, which is a value that is necessary to adjust the Model weights. The new normalizing term `wDiv'` is adjusted by the learning rate and the lambda value provided as input. The final Model parameters are calculated after renormalizing the weights and adjusting the bias term.

Dataset

The dataset utilized for this project is the Red Wine Quality Dataset, sourced from Kaggle, comprising 1599 samples of red wines. Originally a multi-class classification task, the dataset was transformed into a binary classification problem by introducing a threshold at a wine quality rating of 6. This threshold defines two quality categories: 'good' (quality ratings greater than or equal to 6) and 'not good' (quality ratings below 6). To prepare the data for analysis, the Min-Max Scaler was employed to normalize the physicochemical properties of the wines. The dataset was further formatted into a sparse vector format, consistent with the original implementation, and saved as 'normalized_data_binary.dat'. The dataset exhibits a class imbalance, with 'good' quality wines being in the minority, which poses a challenge for classification. However, it is considered suitable for the project's goal of parallelizing the stochastic gradient descent (SGD) algorithm to enhance its speed and efficiency. This dataset offers a real-world scenario for evaluating the impact of parallelization techniques on machine learning algorithms.

Parallelism

Repa

We tried to use Repa, a package for high performance parallel array calculations to improve the performance of the code, since our code involves a lot of vector add, multiplication and dot product calculation.

```
trainOne  :: (Double -> Double -> Double) -- dloss function
         -> SampleFull                   -- single input and response (x, y)
         -> Double                       -- regularizer
         -> IO Model                     -- current model parameter
         -> Double                       -- sgd gain eta
         -> IO Model
trainOne dloss (x, y) lambda para eta = do
       (w, wDiv, wBias)<-para
       array <- R.computeUnboxedP $ R.zipWith (*) w x
       let summ =R.sumAllS array
       let s = summ / wDiv + wBias
       let d = dloss s y
```

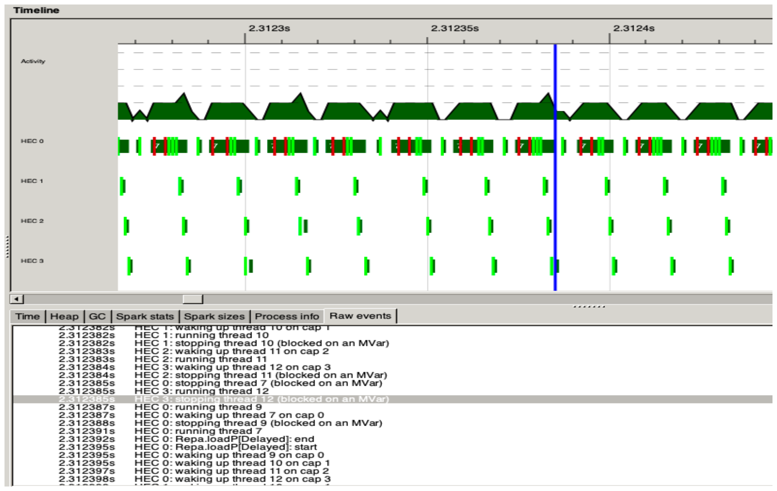
```
        let wDiv' = wDiv / (1 - eta * lambda)
        (w', wDiv'') <- renorm w wDiv'
        sc <- scale x (eta * d * wDiv'')
        sc' <- now $ sc
        w'' <- addFF w' sc'
        w'''<-now $ w''
        let wBias' = wBias + d * eta * 0.01;
        return (w''',wDiv'',wBias')
```

Since repa uses a different data type than the original code which uses vector, we need to redefine all the basic array calculations, in trainOne function, we need to calculate the dot product of two arrays, scaling an array by an integer multiplier, add two arrays element-wise, all those calculations will be performed using the repa parallel function, however in repa, to avoid nested parallelism, we have to evaluate the value after each calculation. Below is a list of the methods we use for array calculation.

```
import Data.Array.Repa as R
import Control.Monad

type FullArray = Array U DIM1 Double

dotFF :: FullArray -> FullArray -> IO Double
dotFF u v = R.sumAllP $ R.zipWith (*) u v
addFF :: FullArray -> FullArray -> IO FullArray
addFF u v= R.computeUnboxedP $ R.zipWith (+) u v
scale :: FullArray -> Double -> IO FullArray
scale u d = R.computeUnboxedP $ R.map (*d) u
l2norm :: FullArray -> IO Double
l2norm v = sqrt <$> (dotFF v v)
```



```
2.312382s     HEC 1: waking up thread 10 on cap 1
2.312382s     HEC 1: running thread 10
2.312382s     HEC 1: stopping thread 10 (blocked on an MVar)
2.312383s     HEC 2: waking up thread 11 on cap 2
2.312383s     HEC 2: running thread 11
2.312384s     HEC 3: waking up thread 12 on cap 3
2.312384s     HEC 2: stopping thread 11 (blocked on an MVar)
2.312385s     HEC 0: stopping thread 7 (blocked on an MVar)
2.312385s     HEC 3: running thread 12
2.312385s     HEC 3: stopping thread 12 (blocked on an MVar)
2.312387s     HEC 0: running thread 9
2.312387s     HEC 0: waking up thread 7 on cap 0
2.312388s     HEC 0: stopping thread 9 (blocked on an MVar)
2.312391s     HEC 0: running thread 7
2.312392s     HEC 0: Repa.loadP[Delayed]: end
2.312395s     HEC 0: Repa.loadP[Delayed]: start
2.312395s     HEC 0: waking up thread 9 on cap 0
2.312395s     HEC 0: waking up thread 10 on cap 1
2.312397s     HEC 0: waking up thread 11 on cap 2
2.312398s     HEC 0: waking up thread 12 on cap 3
```

However when we test the code on multiple cores, the running time increases as the core number increases. If we look at the threadscope for the implementation on 4 cores, as we zoom in, it seems each thread wakes up but got blocked immediately. We will investigate and experiment more in the future.

Strategies

As our next approach of parallelizing the algorithm, we use Haskell's Control.Parallel.Strategies module, particularly the parMap rdeepseq function, to efficiently process chunks of the training dataset in parallel. The parMap rdeepseq function applies a specified function, in our case trainBatch, to each chunk of data concurrently across multiple processor cores. This function updates the model parameters for each sample within the chunk using the trainOne function, which implements the core SGD logic.

The parallel execution is controlled by Haskell's runtime system, which schedules and runs these computations on available processor cores. The rdeepseq strategy ensures that each chunk is completely evaluated before moving to the next step, which is crucial for correctness and performance in a parallel setting.

```
train :: Loss -> [Sample] -> Double -> Int -> Model
train l x lambda epochs = foldl' go wParam0 [1..epochs]
 where
   go wParam _ = trainEpoch wParam
   wParam0 = initParam $ R.dimSample x
   eta = calculateEta lambda epochs

   --trainEpoch wParam = trainBatch wParam (x, eta) -- seq

   trainEpoch wParam =
       let chunks = chunkData chunkSize x
           updatedModels = parMap rdeepseq (trainBatch wParam) (zip chunks (repeat
eta))
       in combineModels updatedModels wParam0
       -- parallel

   trainBatch wParam (samples, etaBatch) =
       foldl' (\wParam' sample -> trainOne (dloss l) sample lambda wParam' etaBatch)
wParam samples
```

After processing all chunks, the updated models from each chunk are combined. The combination is not a simple summation; rather, it involves averaging the parameters across all models. This step is essential to accurately represent the collective learning from all data chunks. The averaging step involves summing the corresponding parameters from each model and then scaling them down by the number of models. This results in a final, averaged model that is representative of the entire dataset. This method allows for a more efficient and scalable training process compared to a purely sequential approach, especially with large datasets.

```
combineModels :: [Model] -> Model -> Model
combineModels models wParam0 =
   let numModels = fromIntegral $ length models
       summedModels = foldl' addModels wParam0 models
   in scaleModel summedModels numModels

addModels :: Model -> Model -> Model
addModels (w1, div1, bias1) (w2, div2, bias2) =
   (addFF w1 w2, div1 + div2, bias1 + bias2)

scaleModel :: Model -> Double -> Model
scaleModel (w, divisor, bias) scaleFactor =
   (scale w (1 / scaleFactor), divisor / scaleFactor, bias / scaleFactor)
```

In the original SGD implementation, the learning rate (eta) was dynamically adjusted using a complex method involving the determineEta0 function (see original github repo). This approach fine-tuned eta based on model performance across varying eta values. For detailed implementation and context, please refer to the Train.hs file of the original GitHub repository.

In this revised implementation, a simplified method for calculating the learning rate (eta) was adopted. This approach, termed calculateEta, determines eta statically for each epoch based on a predetermined initial value (eta0) and the total number of epochs. This simplification was crucial for efficient parallelization of the training process, as it reduced the computational complexity, allowing for more consistent and scalable performance across multiple processing units. The trade-off here was the flexibility of dynamic eta adjustment for potential performance gains, a common consideration in parallel algorithm design where simplicity often leads to better scaling and performance, particularly with large datasets.

```haskell
calculateEta :: Double -> Int -> Double
calculateEta lambda epochs =
   let eta0 = 1 -- Modify this to determine the initial eta
   in eta0 / (1 + lambda * eta0 * fromIntegral epochs)
```

Based on our evaluations, detailed in the following section, this parallelization approach proved effective. The performance improvements were notably consistent with increasing number of cores, underscoring the efficiency and scalability of the adopted parallel strategy in the SGD algorithm.

Evaluation

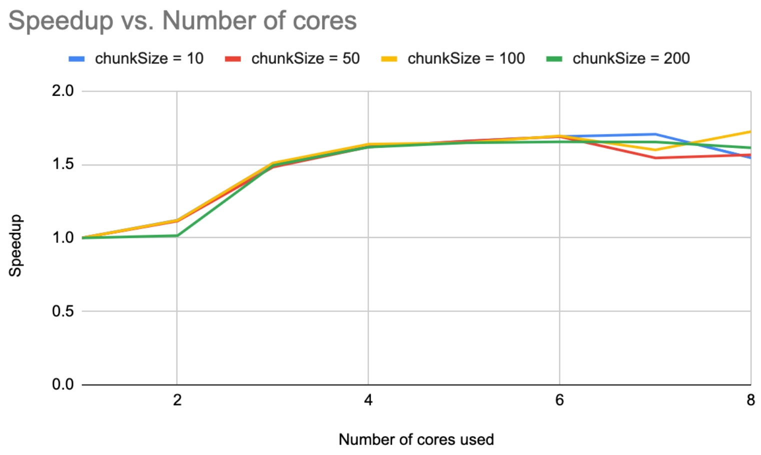
For evaluation, we ran our program on a MacBookPro with the processor of 2 GHz Intel Core i5 and hyperthreaded 4 cores.

On the command line:

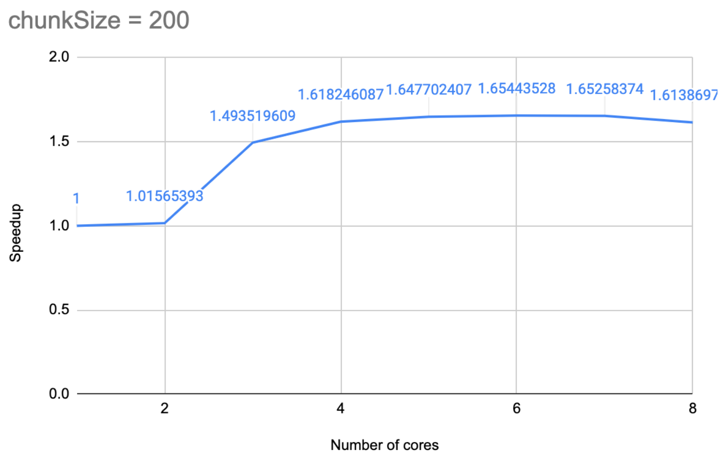
```
stack exec -- svm-sgd-parallel-exe normalized_data_binary.dat --lambda 0.00001
--epochs 200 +RTS -l -N4 -RTS
```

The above runs our executable after building it using stack build in the project directory, using the red wine quality data set that is preprocessed, taking in the lambda value (default 1e-5), the number of epochs (default 200) for training, and the number of cores to use as inputs.

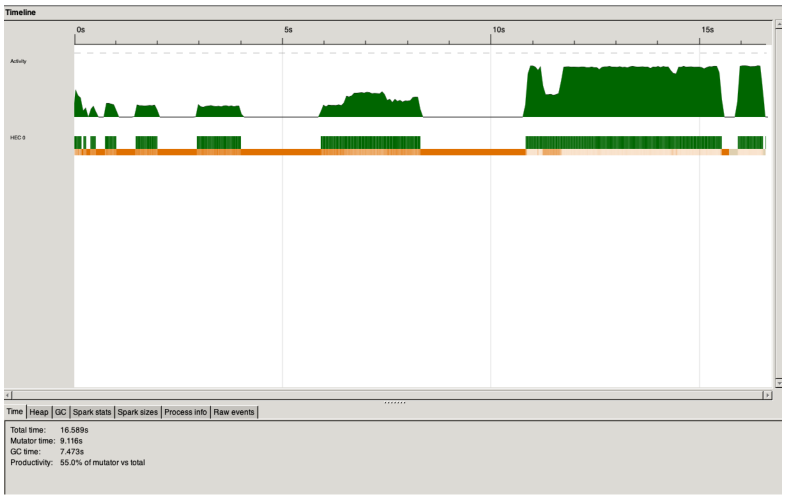
Experiment



Our performance evaluation of the parallelized SGD training function with various chunk sizes revealed that a chunk size of 200 optimally balances workload and processing efficiency, especially evident in the marked speedup when increasing from 2 to 3 cores. Larger chunk sizes allow for more extensive computation per thread, assuming uniform distribution across cores. This results in reduced overhead and maximizes the utilization of parallel processing capabilities, making chunk size 200 a particularly effective choice for our implementation.

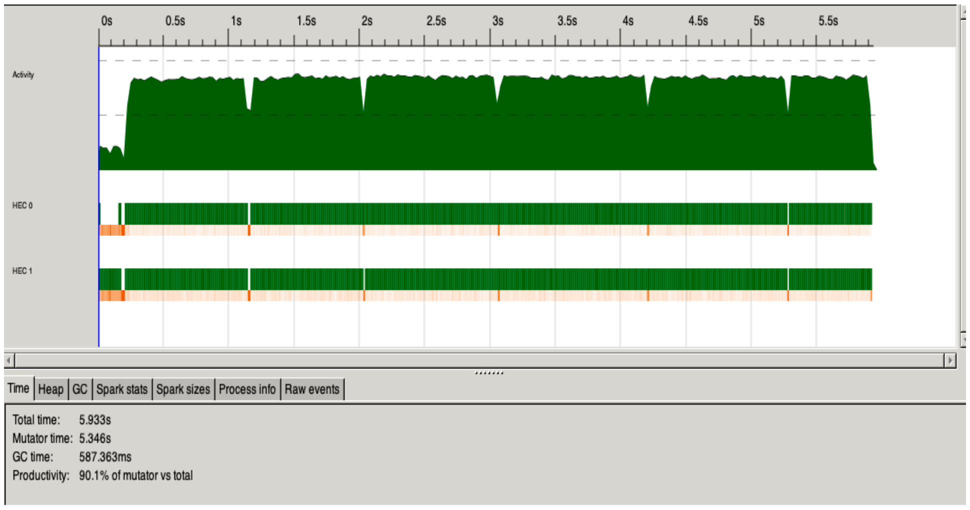


The figure in the report displays the speedup analysis, derived from averaging execution times of ten consecutive runs for varying core counts, using the identified optimal chunk size of 200. This graph effectively demonstrates how the number of cores influences the acceleration of our parallelized SGD training. It provides insightful data on how effectively our parallelization scales, reflecting the benefits and limitations of adding more cores in terms of performance enhancement in our training algorithm. As we can see, after four cores, the speedup is not improving any more, because the maximum number of cpu cores on our laptop is four.

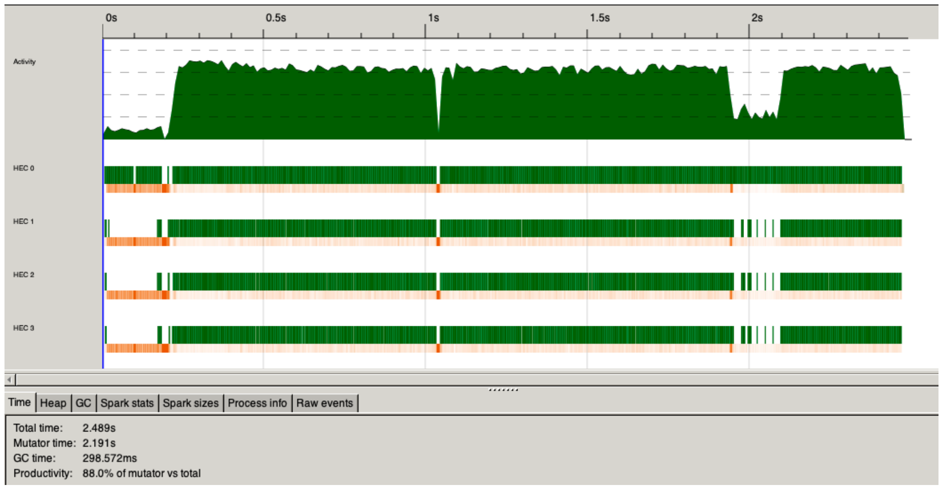


```
Total time:    16.589s
Mutator time:  9.116s
GC time:       7.473s
Productivity:  55.0% of mutator vs total
```

This figure shows the sequential execution of the SGD training algorithm, achieved by running a non-parallelized version of the training function (`trainEpoch wParam = trainBatch wParam (x, eta) -- seq`). This sequential approach, utilized for the same dataset as the parallel version, exhibits a consistent training duration of approximately 15-16 seconds. This timing serves as a baseline to compare and contrast with the performance of the parallelized version, illustrating the efficiency gains achieved through parallel processing.



```
Total time:    5.933s
Mutator time:  5.346s
GC time:       587.363ms
Productivity:  90.1% of mutator vs total
```



```
Total time:     2.489s
Mutator time:   2.191s
GC time:        298.572ms
Productivity:   88.0% of mutator vs total
```

The behavior of the program using 4 cores suggests that there is a heavy load on the garbage collector earlier during execution. When running the training in parallel, the bulk of the computation happens after the initial garbage collection, and this behavior is consistent in every thread. This suggests that there is a lot of space allocated for the functions that are used within the `train` function, since there are many subroutines that are part of the parallelized computation. For the sequential one, it loads all the datapoints into the function, that's why it spends a significant amount of time on garbage collection. For parallel versions, it only loads smaller chunks of data into each time, so it wastes less time on garbage collection. And if we compare the productivity, it is gradually getting slower with more cores, but the mutator time is reduced significantly, so the parallelism is still performing well.

References

[1]Original Haskell Code without parallelism: Github repo :

https://github.com/daz-li/svm_sgd_haskell

[2]Dataset: https://www.kaggle.com/datasets/ryanholbrook/dl-course-data?select=red-wine.csv

[3]Martin A. Zinkevich. Parallelized Stochastic Gradient Descent. NeurIPS 2010

Code Listing

As mentioned earlier, we prioritize efficient parallelization of the SGD training process in this project. As a result, the testing phase present in the original implementation has been commented out to focus solely on optimizing the training procedure. This approach allows for a more concentrated evaluation of the parallelized training's effectiveness and performance.

app/Main.hs

```haskell
{-# LANGUAGE DeriveDataTypeable, RecordWildCards #-}
module Main where

import qualified Data.ByteString.Char8 as C
import System.Console.CmdArgs
import System.Exit
import Control.Monad (when)
import qualified Read as R
import qualified Train as T
import qualified LossFun as L
import Data.Time

data SgdOpts = SgdOpts
   { trainFile :: String
--   , testFile :: String
   , lambda :: Double
   , epochs :: Int
   } deriving (Data, Typeable, Show, Eq)

sgdOpts :: SgdOpts
sgdOpts = SgdOpts
   { trainFile = def &= argPos 0 &= typ "TRAINING-FILE"
   --, testFile = def &= typ "TESTING-FILE" &= help "Testing data" &= name "T" &= name
"test"
   , lambda = 1e-5   &= help "Regularization parameter, default 1e-5"
   , epochs = 200      &= help "Number of training epochs, default 200"
   }

getOpts :: IO SgdOpts
getOpts = cmdArgs $ sgdOpts
   &= versionArg [explicit, name "version", name "v", summary _PROGRAM_INFO]
   &= summary (_PROGRAM_INFO ++ ", " ++ _COPYRIGHT)
   &= help _PROGRAM_ABOUT
   &= helpArg [explicit, name "help", name "h"]
   &= program _PROGRAM_NAME
```

```
_PROGRAM_NAME :: String
_PROGRAM_NAME = "svmsgd"
_PROGRAM_VERSION :: String
_PROGRAM_VERSION = "0.0.1"
_PROGRAM_INFO :: String
_PROGRAM_INFO = _PROGRAM_NAME ++ " version " ++ _PROGRAM_VERSION
_PROGRAM_ABOUT :: String
_PROGRAM_ABOUT = "svm + sgd + haskell"
_COPYRIGHT :: String
_COPYRIGHT = "(C) Dazhuo Li 2013"
main :: IO ()
main = do
   opts <- getOpts
   optionHandler opts

optionHandler :: SgdOpts -> IO ()
optionHandler SgdOpts{..}  = do
   when (null trainFile) $ putStrLn "--trainFile is blank!" >> exitWith (ExitFailure
1)
   when (lambda <= 0 || lambda > 10000) $ putStrLn "--lambda must be in (0, 1e4]" >>
exitWith (ExitFailure 1)
   when (epochs <= 0 || epochs > 1000000) $ putStrLn "--epochs must be in (0, 1e6]" >>
exitWith (ExitFailure 1)
   exec SgdOpts{..}

exec :: SgdOpts -> IO ()
exec SgdOpts{..} = do
   contents <- C.readFile trainFile
   let dat = case R.read contents :: Maybe [R.Sample Int] of
               Nothing -> error "Wrong input format"
               Just x  -> x

   start <- getCurrentTime

   let model = T.train L.logLoss dat lambda epochs
   --let predLoss = T.testMany (L.loss L.logLoss ) dat lambda model
   print model
   --print predLoss

   end <- getCurrentTime
   let diff = diffUTCTime end start
   putStrLn $ "Execution Time: " ++ (show diff)
```

src/Train.hs

```
module Train (train) where

import qualified Read as R
import Vector
import LossFun
import Control.Parallel.Strategies
import Data.List (foldl')

type Sample = R.Sample Int
type Model = (FullVector, Double, Double)       -- model parameter (w, wDivsor, wBias)
--type PredLoss = (Double, Double, Double)        -- prediction error on the Model
(los, cost, num.error)
type Chunk a = [a]

-- Function to chunk the data
chunkData :: Int -> [a] -> [Chunk a]
chunkData _ [] = []
chunkData n xs = let (chunk, rest) = splitAt n xs in chunk : chunkData n rest

-- Constants
chunkSize :: Int
chunkSize = 200 -- chosen through multiple tests

initParam :: Int -> Model
initParam dimVar = (rep (dimVar + 1)  0, 1, 0)

train :: Loss -> [Sample] -> Double -> Int -> Model
train l x lambda epochs = foldl' go wParam0 [1..epochs]
 where
   go wParam _ = trainEpoch wParam
   wParam0 = initParam $ R.dimSample x
   eta = calculateEta lambda epochs

   --trainEpoch wParam = trainBatch wParam (x, eta) -- seq

   trainEpoch wParam =
       let chunks = chunkData chunkSize x
           updatedModels = parMap rdeepseq (trainBatch wParam) (zip chunks (repeat
eta))
```

```
       in combineModels updatedModels wParam0
       -- parallel

   trainBatch wParam (samples, etaBatch) =
       foldl' (\wParam' sample -> trainOne (dloss l) sample lambda wParam' etaBatch)
wParam samples

combineModels :: [Model] -> Model -> Model
combineModels models wParam0 =
   let numModels = fromIntegral $ length models
       summedModels = foldl' addModels wParam0 models
   in scaleModel summedModels numModels

addModels :: Model -> Model -> Model
addModels (w1, div1, bias1) (w2, div2, bias2) =
   (addFF w1 w2, div1 + div2, bias1 + bias2)

scaleModel :: Model -> Double -> Model
scaleModel (w, divisor, bias) scaleFactor =
   (scale w (1 / scaleFactor), divisor / scaleFactor, bias / scaleFactor)

calculateEta :: Double -> Int -> Double
calculateEta lambda epochs =
   let eta0 = 1 -- Modify this to determine the initial eta
   in eta0 / (1 + lambda * eta0 * fromIntegral epochs)

trainOne :: (Double -> Double -> Double)
         -> Sample
         -> Double
         -> Model
         -> Double
         -> Model
trainOne dloss (x, y) lambda (w, wDiv, wBias) eta = (w'', wDiv'', wBias')
 where
   s = (dotFS w x) / wDiv + wBias
   d = dloss s y
   wDiv' = wDiv / (1 - eta * lambda)
   (w', wDiv'') = renorm w wDiv'
   w'' = addFS w' . mul x $ eta * d * wDiv''
   wBias' = wBias + d * eta * 0.01

renorm :: FullVector -> Double -> (FullVector, Double)
renorm w wDiv
 | wDiv == 1.0 || wDiv <= 1e5  = (w, wDiv)
 | otherwise                   = (scale w $ 1 / wDiv, 1.0)

{-wnorm (w, wDiv, wBias) = (Vector.dot w w ) / wDiv / wDiv

trainMany :: (Double -> Double -> Double)     -- dloss function
         -> [Sample]                         -- list of samples
         -> Double                           -- regularizer
         -> Model                            -- current model parameter
         -> [Double]                         -- sgd gain eta for each iteration (or
sample)
         -> Model
trainMany dloss x lambda wParam0 eta= foldl go wParam0 $ zip x eta
 where
   go wParam (xt, etat) = trainOne dloss xt lambda wParam etat

testMany :: (Double -> Double -> Double)      -- loss function
         -> [Sample]                         -- list of samples
         -> Double                           -- regularizer
         -> Model                            -- model parameter
         -> PredLoss
testMany loss x lambda wParam = (los, cost, nerr)
 where
   los = ploss / fromIntegral (length x)
   nerr = (fromIntegral pnerr) / fromIntegral (length x)
   cost = los + 0.5 * lambda * (wnorm wParam)
   (ploss, pnerr) = (\(t1, t2, _) -> (sum t1, sum t2)) . unzip3 . map go $ x
     where
       go x = testOne loss x lambda wParam

testOne :: (Double -> Double -> Double)    -- loss function
         -> (SparseVector, Double)        -- single input and response (x, y)
         -> Double                        -- regularizer
         -> Model                         -- model parameter
         -> (Double, Int, Double)
testOne loss (x, y) lambda (w, wDiv, wBias) = (ploss, pnerr, s)
 where
   s = (dotFS w x) / wDiv + wBias
   ploss = loss s y
   pnerr = if s * y <= 0 then 1 else 0
-}
```

src/Read.hs

```haskell
{-# LANGUAGE TypeSynonymInstances #-}

module Read where

import qualified Data.ByteString.Char8 as C
import Data.ByteString.Lex.Fractional (readDecimal)
import qualified Vector as V
import Data.Array.Repa as R hiding (map)

type SparseVector a = [(a, Double)]
type Response = Double
type Sample a = (SparseVector a, Response)
type FullArray = Array U DIM1 Double
type SampleFull=(FullArray,Response)



-- | On error handling
-- Error handling is done by Maybe. There are pros and cons
-- of any approach: Maybe, Either, Excpetion. e.g. facing
-- error, Maybe returns Nothing, which is un-informative about
-- which part of which line has the wrong format. More ref:
-- http://book.realworldhaskell.org/read/error-handling.html
-- http://blog.ezyang.com/2011/08/8-ways-to-report-errors-in-haskell-revisited/

class Reads a where
 read :: C.ByteString -> Maybe [Sample a]
 read = sequence . map readSample . C.lines

 readSample :: C.ByteString -> Maybe (Sample a)
 readSample str = fmap ((,)) feas <*> response
   where
     tokens = C.words str
     feas = sequence . map readPair . tail $ tokens
     response = readDouble' . head $ tokens



 readPair :: C.ByteString -> Maybe (a, Double)
 readPair str = let xs = C.split ':' str
                  in if length xs /= 2
                      then Nothing
                      else let (x:y:_) = xs
```

```
                              in fmap (,) (readId x) <*> readDouble' y
  readId :: C.ByteString -> Maybe a

-- | On TypeSynonymInstances
-- http://stackoverflow.com/a/2125769/1311956

readFull :: C.ByteString -> Maybe [SampleFull]
readFull = sequence . map readSampleFull . C.lines

readSampleFull:: C.ByteString -> Maybe (SampleFull)
readSampleFull str = do
   let tokens = C.words str
   let feas = listToUnboxedArray . map readDouble' . tail $ tokens
   response <- readDouble' . head $ tokens
   return (feas,response)

listToUnboxedArray :: [Maybe Double] -> Array U DIM1 Double
listToUnboxedArray values = fromListUnboxed (Z :. length values) (map handleMaybe
values)
 where
   handleMaybe :: Maybe Double -> Double
   handleMaybe (Just x) = x
   handleMaybe Nothing  = 0.0  -- Choose a default value for Nothing

instance Reads Int where
 readId = readInt'
instance Reads C.ByteString where
 readId = Just . id
 readDouble' :: C.ByteString -> Maybe Double
readDouble' bs =
 case C.uncons bs of
   Just ('-', rest) -> fmap negate (fmap fst (readDecimal rest))
   _ -> fmap fst (readDecimal bs)

readInt' :: C.ByteString -> Maybe Int
readInt' str = case C.readInt str of
               Nothing -> Nothing
               Just (x, y) -> if C.null y then Just x else Nothing

dimSample :: [Sample Int] -> Int
dimSample = foldl (\x y -> max x . V.dim $ y ) 0 . fst . unzip
```

src/LossFun.hs
```
module LossFun where

data Loss = Loss {
   loss :: Double -> Double -> Double
   --  -dloss(a,y)/da
 , dloss :: Double -> Double -> Double
 }
 -- logloss(a,y) = log(1+exp(-a*y))
logLoss :: Loss
logLoss = Loss {
   loss = loss'
 , dloss = dloss'
 }
 where
   loss' a y
     | z > 18       = exp (-z)
     | z < -18      = -z
     | otherwise    = log $ 1 + exp (-z)
     where z = a*y
   dloss' a y
     | z > 18       = y * exp (-z)
     | z < -18      = y
     | otherwise    = y / (1 + exp z)
     where z = a*y
```

src/Vector.hs
```
module Vector where

--
http://stackoverflow.com/questions/17892065/mutable-random-access-array-vector-with-hi
gh-performance-in-haskell
--
http://haskell.1045720.n5.nabble.com/fishing-for-ST-mutable-Vector-examples-td4333461.
html

-- import qualified Data.IntMap as IntMap
import qualified Data.Vector.Unboxed as VU

-- type SparseVector = IntMap
type SparseVector = [(Int, Double)]
type FullVector = VU.Vector (Double)
```

```haskell
-- type Sample = (SparseVector, Double)          -- (input, response)

dotFS :: FullVector -> SparseVector -> Double
dotFS u v = foldl go 0 v
 where
   go :: Double -> (Int, Double) -> Double
   go acc (ind, val) = acc + u VU.! ind * val

dotFS' :: FullVector -> SparseVector -> Double
--dotFS' u v = sum . zipWith (*) vals . VU.toList . map (\x -> u VU.! x) $ inds
dotFS' u v = sum . zipWith (*) vals . slice u $ inds
 where
   (inds, vals) = unzip v
   slice :: FullVector -> [Int] -> [Double]
   slice u v = map (\x -> u VU.! x) $ v


addFS :: FullVector -> SparseVector -> FullVector
addFS = VU.accum (+)

addFF :: FullVector -> FullVector -> FullVector
addFF=VU.zipWith (+)

dot :: FullVector -> FullVector -> Double
dot u v = VU.sum $ VU.zipWith (*) u v

scale :: FullVector -> Double -> FullVector
scale u d = VU.map (*d) u


dim :: SparseVector -> Int
dim = foldl1 max . fst . unzip


rep :: Int -> Double -> FullVector
rep l x = VU.replicate l x

mul :: SparseVector -> Double -> SparseVector
mul x d = map (\(a1, a2) -> (a1, a2*d)) x

normalizeL2 :: SparseVector -> SparseVector
normalizeL2 v = mul v $ 1 / (l2norm v)
```

```
l2norm :: SparseVector -> Double
l2norm v = sqrt . sum . zipWith (\x y -> snd x * snd y) v $ v
```

src/Vector.hs

```
module Vector where

--
http://stackoverflow.com/questions/17892065/mutable-random-access-array-vector-with-hi
gh-performance-in-haskell
--
http://haskell.1045720.n5.nabble.com/fishing-for-ST-mutable-Vector-examples-td4333461.
html

-- import qualified Data.IntMap as IntMap
import qualified Data.Vector.Unboxed as VU

-- type SparseVector = IntMap
type SparseVector = [(Int, Double)]
type FullVector = VU.Vector (Double)
-- type Sample = (SparseVector, Double)          -- (input, response)

dotFS :: FullVector -> SparseVector -> Double
dotFS u v = foldl go 0 v
 where
   go :: Double -> (Int, Double) -> Double
   go acc (ind, val) = acc + u VU.! ind * val

dotFS' :: FullVector -> SparseVector -> Double
--dotFS' u v = sum . zipWith (*) vals . VU.toList . map (\x -> u VU.! x) $ inds
dotFS' u v = sum . zipWith (*) vals . slice u $ inds
 where
   (inds, vals) = unzip v
   slice :: FullVector -> [Int] -> [Double]
   slice u v = map (\x -> u VU.! x) $ v

addFS :: FullVector -> SparseVector -> FullVector
addFS = VU.accum (+)

addFF :: FullVector -> FullVector -> FullVector
addFF=VU.zipWith (+)
```

```haskell
dot :: FullVector -> FullVector -> Double
dot u v = VU.sum $ VU.zipWith (*) u v

scale :: FullVector -> Double -> FullVector
scale u d = VU.map (*d) u

dim :: SparseVector -> Int
dim = foldl1 max . fst . unzip

rep :: Int -> Double -> FullVector
rep l x = VU.replicate l x

mul :: SparseVector -> Double -> SparseVector
mul x d = map (\(a1, a2) -> (a1, a2*d)) x

normalizeL2 :: SparseVector -> SparseVector
normalizeL2 v = mul v $ 1 / (l2norm v)

l2norm :: SparseVector -> Double
l2norm v = sqrt . sum . zipWith (\x y -> snd x * snd y) v $ v
```