```haskell
module Grad where

import Control.Parallel.Strategies
import Data.List.Split


-- FUNCTIONS FOR PROCESSING DATA INPUT

--Creates the 'dataframe' structure (list of lists)
getCSVData :: FilePath -> IO [[Double]]
getCSVData filename = do
                    lns <- fmap lines (readFile filename)
                    return $ map (map (\x -> read x::Double)) (map words (map rep (tail lns)))

--Preprocessing for CSV files (turns all commas into spaces so we can use words)
rep :: [Char] -> [Char]
rep [] = []
rep (x:xs)
    | x == ',' = [' '] ++ (rep xs)
    | otherwise = [x] ++ (rep xs)





-- FUNCTIONS FOR GRADIENT DESCENT ALGORITHM

--Actual gradient descent algorithm (uses magnitude of gradient as stopping condition)
descendTolerance :: [Char] -> Int -> [a] -> ([Double] -> a -> [Double]) -> [Double] -> Double -> Double -> [Double]
descendTolerance parseq chunks csvData gradFunc guess tolerance stepSize
    | tolerance < (0::Double) = error "tolerance must be a positive value"
    | maxVal <= tolerance = guess
    | otherwise = descendTolerance parseq chunks (csvData) gradFunc (zipWith (-) guess (computeGrad parseq chunks csvData gradFunc guess stepSize)) tolerance stepSize
    where
        maxVal = maximum $ map abs (computeGrad parseq chunks csvData gradFunc guess stepSize)

--Actual gradient descent algorithm (uses numer of steps as stopping condition)
descendSteps :: [Char] -> Int -> [a] -> ([Double] -> a -> [Double]) -> [Double] -> Int -> Double -> [Double]
descendSteps parseq chunks csvData gradFunc guess steps stepSize
    | steps < 0 = error "you can't take negative steps"
    | steps == 0 = guess
    | otherwise = descendSteps parseq chunks (csvData) gradFunc (zipWith (-) guess (computeGrad parseq chunks csvData gradFunc guess stepSize)) (steps - 1) (stepSize)

--Compute the gradient
computeGrad :: [Char] -> Int -> [a] -> ([Double] -> a -> [Double]) -> [Double] -> Double -> [Double]
computeGrad parseq chunks csvData gradFunc params stepSize
    | parseq == "parallel" = map (* stepSize) (parallelMegaFold (map (gradFunc params) csvData) chunks)
    | otherwise = map (* stepSize) (sequentialMegaFold (map (gradFunc params) csvData))


--Applies a fold to each column in the dataframe
sequentialMegaFold :: [[Double]] -> [Double]
sequentialMegaFold [] = []
sequentialMegaFold [x] = x
sequentialMegaFold xx@(x:xs:xss)
    | (length xx) == 2 = zipWith (+) x xs
    | otherwise = sequentialMegaFold ((zipWith (+) x xs):xss)

--Parallel glue code
parallelMegaFold :: [[Double]] -> Int -> [Double]
parallelMegaFold [] chunkNum = []
parallelMegaFold [x] chunkNum = x
parallelMegaFold (x:xs:[]) chunkNum = zipWith (+) x xs
```

```haskell
parallelMegaFold x chunkNum =
            if length x == 1 then
              head x
            else sequentialMegaFold $ parMap (rdeepseq) sequentialMegaFold chunks
            where
              chunks = chunksOf ((length x) `div` chunkNum) x







-- FUNCTIONS FOR GRADIENT COMPUTATION

--Compute a row of gradient
computeGradRowLinear :: [Double] -> [Double] -> [Double]
computeGradRowLinear params dataList = computeGradRowLinearHelper 0 params dataList

--Helper function to compute row of gradient
computeGradRowLinearHelper :: Int -> [Double] -> [Double] -> [Double]
computeGradRowLinearHelper n params dataList
  | n == (length dataList) = []
  | n == 0 = [(gradIntLinear params dataList)] ++ (computeGradRowLinearHelper (n+1) params dataList)
  | otherwise = [(gradSlopeLinear params dataList n)] ++ (computeGradRowLinearHelper (n+1) params dataList)

--Linear gradient function with respect to intercept
gradIntLinear :: [Double] -> [Double] -> Double
gradIntLinear params dataList = -2 * ((head dataList) - ((head params) + (sum (zipWith (*) (tail params) (tail dataList)))))

--Linear gradient function with respect to slope
gradSlopeLinear :: [Double] -> [Double] -> Int -> Double
gradSlopeLinear params dataList var = -2 *
                    ((head dataList) - (head params) - (sum (zipWith (*) (tail params) (tail dataList)))) *
                    (dataList !! var)

--Compute a row of the gradient in a logistic function
computeGradRowLogistic :: [Double] -> [Double] -> [Double]
computeGradRowLogistic params dataList = [h0 - y]
                      ++ (zipWith (*) (xTail) (map (h0 -) (take (length xTail) (cycle [y]))))
                      where h0 = hTheta params dataList
                            xTail = tail dataList
                            y = head dataList

--Compute loss function exponential (needed for derivatives)
hTheta :: [Double] -> [Double] -> Double
hTheta params dataList = (/) 1.0 $ 1.0 + (exp (-1 * (g params dataList)))

--Compute exponential in denominator of logistic function
g :: [Double] -> [Double] -> Double
g params dataList = sum $ zipWith (*) params ([1.0::Double] ++ (tail dataList))
```