A Parallelized Gradient Descent Algorithm for Regression Coefficient Estimation on Massive Data

COMS 4995 Parallel Functional Programming Final Project Report

Max Helman (mhh2148) and Riya Chakraborty (rc3242)

For our final project in COMS 4995 Parallel Functional Programming at Columbia University, we propose a parallelized Gradient Descent Algorithm that easily scales to massive inputs. We implemented the algorithm in Haskell for both linear and logistic regression coefficient estimation. The most expensive operation in our sequential approach was a fold on a list-based data structure, which was parallelized using static partitioning. Our parallel algorithm on 6 cores had roughly a 40x speedup over the sequential algorithm, and the speedup increases with input size. We also achieved a roughly 4x speedup on the parallel algorithm by giving the program access to 6 cores instead of 1, which approached our theoretical upper bound of a 5x speedup for the parallel algorithm.

I. Introduction

For our final project in **COMS 4995 Parallel Functional Programming** (Fall 2020), we aimed to use Parallel Haskell to speed up a Gradient Descent algorithm in a way that was scalable to massive datasets. Gradient Descent is an optimization technique used to estimate parameters of a function, and it is especially useful when the function does not have an analytical solution. It is commonly used in Statistical Machine Learning. The Gradient Descent algorithm starts out with an initial guess of the parameters θ_0 and some sort of convex loss function $L : \theta \times \{y_i, x_i\}_{i=1}^k \to \mathbb{R}$ that takes the estimated parameters $\theta \in \mathbb{R}^d$ and the data itself ($\{y_i, x_i\}_{i=1}^k$) as its parameters; we fit functions that have the same number of estimated parameters as the data x_i (which is a vector quantity) has dimensions, and we refer to this dimensionality as d. Then, the algorithm computes the gradient of the loss function ∇L given the current estimated parameters and subtracts the gradient, multiplied by some learning rate α, from the previous estimate; the new estimate θ_n is given as $\theta_n = \theta_{n-1} - \alpha \nabla L(\theta_{n-1}, \{y_i, x_i\}_{i=1}^k)$. This continues until either the numerical solution converges or a predetermined number of iterations occur. The final estimate is the output. We were particularly excited by the potential for parallelism when computing the gradient, since it often involves summing up derivatives of every data point, and we noticed that the sums across data points can be done in parallel. Where we saw less of an application was with the sequence of iterative steps, since each step depends on the value obtained in the prior step.

II. Building a Regression Framework

Our first priority was ensuring that we had a robust and correct system for estimating regression coefficients sequentially. This involved processing input and running the actual algorithm itself to produce an output. As a proof-of-concept, we included two loss functions, one which is solvable analytically (linear regression) and one which is not (logistic regression).

A. Processing Input

Our first task was to come up with some sort of I/O paradigm; we settled on using CSV files, since these are common in practice, and most other formats (such as Excel, Google Sheets, and Relational Databases) can easily be converted to CSVs. Since each datapoint is used at least once in each step of the iteration, it does make sense to store the data in memory rather than asking for a new I/O operation at each step. Therefore, we created a functional analog to something like a Pandas Dataframe in Python, designed to be efficient primarily with map and fold operations. The actual data structure was simpler than it sounds: we decided that a list of lists would suffice (where the inner lists represent rows), and in the rare case that we needed direct access to an element in one of the rows, we could incur a small $O(d)$ penalty: we would already be performing an operation on the row of interest, and for large data, there is normally a relatively negligible number of columns by comparison. We read all data in as **Double**s, so the Dataframe was ultimately of type **[[Double]]**. A small amount of input preprocessing was necessary, but this was a one time linear operation, and will probably not come back to bite us except with Amdahl's Law (Section III.A).

B. Computing the Gradient for Linear Regression

For this project, we started with the linear least-squares model, where the loss function was the sum of the squared residuals in the data. While there actually is an analytical solution to this loss function, it served as an excellent proof of concept because we could easily check if our answer was correct, and it fundamentally operates the same way as most other loss functions. Here, the loss function is given as:

$$L(\theta_{n-1}, \{y_i, x_i\}_{i=1}^k) = \sum_{i=1}^{k} \left(y_i - (\theta_{n-1} \cdot x_i)\right)^2$$

Please note that this notation assumes a 1 is appended to the beginning of x_i as to represent the intercept term. Correspondingly, the gradient is given as:

$$\frac{\partial L}{\partial \theta_1} = \sum_{i=1}^{k} -2\left(y_i - (\theta_{n-1} \cdot x_i)\right)$$

$$\frac{\partial L}{\partial \theta_j} = \sum_{i=1}^{k} -2\left(y_i - (\theta_{n-1} \cdot x_i)\right)(x_{i,j})$$

Luckily, this also lends itself well towards mapping, and we created a list (with the function **computeGradRowLinear**)

of the function for computing the parameter corresponding to the intercept (**gradIntLinear**) followed by a repeating (but specified) general function (**gradSlopeLinear**) for computing derivatives of the parameters corresponding to slopes (where slope and intercept correspond to parameters b_z of the form of some linear function, for example: $b_0 + b_1x_1 + b_2x_2 + ...$). These functions are all in **Grad.hs**. Computing the overall gradient at each step is done by mapping the list of these partially applied functions to each row in the Dataframe (of type **[[Double]]**), and then applying a fold operation (referred to as **sequentialMegaFold**) that essentially sums across each row and returns a list; this is done in a function called **computeGrad**. **sequentialMegaFold** and **computeGrad** are found in **Grad.hs**, and **sequentialMegaFold** is the core part of the algorithm that we parallelized (to be discussed soon).

C. Computing the Gradient for Logistic Regression

We also decided to extend this model for logistic regression, which is actually a problem that cannot be solved analytically. Logistic functions are especially useful for binary classification in Machine Learning, since they are continuous and differentiable. A logistic curve is of the form:

$$h_\theta(\theta_{n-1}, \{x_i\}_{i=1}^k) = \frac{1}{1 + e^{(\theta_{n-1} \cdot x_i)}}$$

Please again note that this notation makes the same assumption as before (that a 1 is appended to the beginning of the data for the intercept term). The loss function here is chosen as cross-entropy, which has an incredibly unwieldy formula but a somewhat more simple gradient, computed as:

$$\frac{\partial L}{\partial \theta_1} = \sum_{i=1}^{k} h_\theta(\theta_{n-1}, \{x_i\}_{i=1}^k) - y_i$$

$$\frac{\partial L}{\partial \theta_j} = \sum_{i=1}^{k} \left(h_\theta(\theta_{n-1}, \{x_i\}_{i=1}^k) - y_i\right)(x_{i,j})$$

The functions we used to compute these values are all found in **Grad.hs**, and are analogs to the functions for linear regression: **hTheta** and **g** helped us compute the logistic curve, and then **computeGradRowLogistic** computed a row of the gradient from a row of the data. What is especially nice here is that we wrote the overall program to be modular between the two loss functions, as will be seen in the next section. Moreover, as part of the I/O of our program, we added the functionality that allows the user to choose whether they want a linear or logistic-based loss function for their parameter optimization/fitting.

D. Optimization of Original Guess

Due to the convexity of the least squares loss function for both the linear and logistic functions, the original estimate of the parameters can be arbitrary and the subsequent estimates will still converge numerically to the optimal solution. There are two primary stopping conditions that are used in practice: numerical convergance (**descendTolerance**) and number of steps (**descendSteps**). Both were written to be agnostic to any particular loss function, and thus workable with either linear or logistic regression, and any future loss functions should we add them. Essentially, both take in either a parallel/sequential choice from the user, number of chunks (for parallelization), Dataframe, a function for computing gradient, estimated parameters, and a learning rate, and tolerance level or number of steps (depending on which descent function is being used), and at each step, they perform identical calculations by computing the gradient with the current estimated parameters and then forming a new estimate. However, the function signatures and stopping conditions are different: **descendSteps** takes in a number of steps and stops once that many steps have been taken, and **descendTolerance** takes in a tolerance and stops once the maximum value of any component of the gradient fails to exceed said tolerance. Only one of these algorithms needs to be used in practice, but we figured we would give either option. We did run into some sort of strange convexity with the logistic function where the solution did not always converge numerically, although it continued to optimize the loss function. Since this loss function is not solvable analytically, there is no one solution that is necessarily most optimal, so our suspicion is that the gradient would cycle at one point. Despite this behavior, we did verify both graphically and computationally that the value of the loss function decreased at each step, and that the results it gave were very reasonable. However, this perhaps makes the logistic function much better suited for use with **descendSteps** than with **descendTolerance**, since numerical convergence cannot be guaranteed. Another thing we noticed was that, depending on the size of data we were processing, we would have to manually tweak the learning rate (step size in our case) and the parameters for number of steps and/or tolerance. This is due to the fact that, as the size of the data changes, the learning rate must also be modified, as one too high would lead to an "exploding" gradient (by which the gradient computation would not converge and would lead to a numerical overflow for our parameters – a.k.a a lovely **NaN**), and one too low would lead to (what we suspect) is a program that continuously runs/runs longer than it should to yield the parameter values. Thus, for our purposes, we established a short "optimized parameters" table that serves as a baseline, but handy guide for inputting step size/number of steps when using **descendSteps** as the desired gradient function based on the size of data (number of rows in data). This can be found in our project directory (under the "report" sub-directory).

III. The Parallel Gradient Descent Algorithm

We parallelized the "fold" operation (**sequentialMegaFold**), which is essentially a large summation of the columns in our Dataframe. This took roughly 85% of the computation time since it was a large repeated calculation, so we figured it would be prime for parallelizing. Our original approach used the **REPA** library, but due to the limits of Amdahl's Law and the task at hand (make the sequential algorithm run faster by parallelizing it), this was not suitable for the project. Instead, we ultimately used static partitioning, which proved to be a solid choice.

A. REPA and the Limits of Amdahl's Law

Our first attempt at parallelism involved converting our Dataframe to be based off of **REPA arrays**. **REPA arrays** are high performance, shape polymorphic, and parallel, so they figured to be an excellent choice. The **REPA** API offers a multitude of functions that have both sequential and parallel implementations, and we originally decided to transform our **sequentialMegaFold** into a parallel, **REPA**-based implementation. This was roughly accomplished by first reading in our Dataframe structure and converting it to a corresponding, "flattened" version (i.e. **[[Double]]** to **[Double]**). This is due to the fact that **REPA** works inherently as a wrapper for 1-dimensional Haskell lists. We then used **REPA** to get a matrix representation for our Dataframe, and as one may notice, this was an extra back-and-forth, $O(n)$ conversion which caused our **REPA** implementation to already lend itself to a sense of unwieldiness and intensive (sequential) computation, as the flattening operation was not - and we believe cannot be - parallelized (it was a simple concatenation of nested lists). Working with **REPA** caused other issues that we noticed during this implementation, as well. Though **REPA** now provided us with our desired matrix representation of data, the existing **foldp** function, which is a **REPA**-based parallel fold, would not work in the way we wanted it to. Namely, the fold was not written such that we could fold the sum *across* rows of our Dataframe. Thus, we had to perform yet another expensive conversion, a 2D transpose using **REPA**'s **backpermute**, that could not inherently be parallelized if we were to use the **REPA** API. We will mention, though, that the **computeP** function was subsequently applied to bring the **REPA** resultant array out of its "delayed" representation, and finally, used the aforementioned **foldp** to perform the desired sum across rows *in parallel*. Finally, we converted our **REPA array** back into a Haskell list, yet another $O(n)$ operation, to output from our then-parallel fold implementation. As is apparent, though **REPA** offered a wide variety of opportunities to parallelize using existing functions and though **REPA arrays** are intrinsically efficient representations of big data, the above conversions and transformations of data proved necessary if our gradient descent algorithm were to work with **REPA**. Since one of our main goals was to be able to work with and parallelize computation on large sets of data, these inherently expensive operations did not mesh well with our algorithm. We explain in greater detail below.

Although **REPA arrays** are naturally much more efficient than lists, we ran into two problems: converting from lists to **REPA arrays** took a significant amount of computation which lowered the percentage that could be parallelized, and **REPA arrays** are so efficient on their own that parallelizing their operations had very little effect on performance, since they took up so little of the computational time being spent. This brings us to Amdahl's Law:

$$S(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

Here, $S(s)$ is the speedup, s is the number of threads that the parallel portion is given, and p is the proportion of execution time in parallel. Since our sequential list-based algorithm spent over 99% of its time on **sequentialMegaFold**, our theoretical maximum speedup assuming perfect parallelism (where $s = \infty$) was rather large (our eventual parallel algorithm spent roughly 80% of its time on this, which meant we should see a speedup approaching 5x when giving it more cores). However, the same operation with **REPA arrays** took roughly 20% of the total computational time, so our theoretical maximum speedup was 1.25x. Moreover, upon analysis with Threadscope, we found that a very small percentage (around 12%) of our overall code was parallelized - i.e. the effects of **REPA** were minimal. Therefore, for the purposes of this project, it made much more sense to figure out how to parallelize a list-based Dataframe and have a higher percentage of code that is actually parallelized, rather than using **REPA arrays**, although in the real world, **REPA arrays** are awesome.

B. Parallelization Strategy

Having had our hands forced by Amdahl's Law into parallelizing the Dataframe with list operations, we figured static partitioning would be our best bet. Static partitioning was an appropriate choice here because we have assurance that each parallelized unit of work would take equal time, since they were all fold operations on equally-sized lists. We wrote a function **parallelMegaFold** in **Main.hs** that essentially splits the original Dataframe into chunks, maps **sequentialMegaFold** onto the chunks in parallel using **parMap** with **rdeepseq**, and then performs one final **sequentialMegaFold** on a list of the resulting arrays.

C. Parallel vs. Sequential Performance

First, we decided to test whether the parallel algorithm could even beat the sequential algorithm, or if we had made things worse for ourselves as we did in an earlier attempt. All tests were performed on a machine with a 6th generation Intel Core i5 (6 cores at 4.19 GHz), 16GB of 2133 MHz DDR4 RAM, and a Samsung 850 EVO SSD running Ubuntu 20.04. We used 128 chunks and 1,000 iterations; the dataset was 2 columns by 100,000 rows. We did not add more columns (add more dimensions/variables to optimize) because our algorithm is parallelized on rows (which represent the number of data points/size of dataset), which often vastly outnumber columns, so we would expect and predict a less appreciable difference if we were to test on a dataset with significantly more columns.

TABLE I
UNIT TESTING - SEQUENTIAL AND PARALLEL RUNTIMES

Rows	Sequential Runtime (s)	Parallel Runtime (s)
100	6.390e-2	2.848e-2
1000	2.465	0.223
10000	200.268	4.629

Above, we offer a "sampler" of the observable differences captured by our unit tests, which ran and timed our **descendSteps** function using a linear loss function. We were able to achieve nearly a 45x speedup when using 6 cores and 128 chunks on a 10,000 line input. This is consistent with

our observation that **sequentialMegaFold** took over 99% of the work in the sequential algorithm; it also means that the algorithm lends itself very well to parallelism. We did notice that the speedup increased with input size; this is because as the input size increases, **sequentialMegaFold** does a larger percentage of the work, and this is the operation that we parallelize. Simply put, the algorithm scales quite nicely with parallelism. Our approach clearly worked, but we wanted to see if we could tune its performance even further.

D. Multicore Performance

We now wanted to compare the performance of the parallel algorithm when we varied the number of cores. Our results were as follows:

TABLE II
CORES AND RUNTIME

Cores	Runtime (s)
1	129
2	98
4	50
6	45

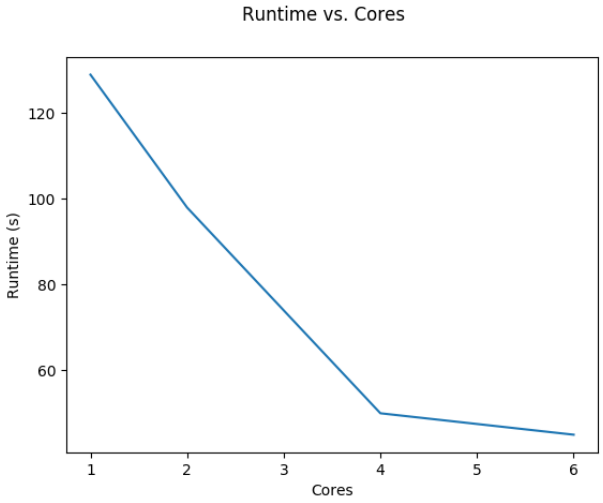

Fig. 1. Runtime vs. Cores of our parallelized Gradient Descent Algorithm (1,000 iterations) with 128 chunks on a 2 column by 100,000 row input.

As can be seen in Table II and Fig. 1, we were able to decrease runtime from 129s on 1 core with 128 chunks to 45s on 6 cores with 128 chunks. This was nearly a 3x speedup! Returns appeared to diminish after the 4th core was given, but excessive garbage collection did not seem to be much of a problem; our threadscope output for the 6 core case is given in Fig. 2:

Overall, this algorithm parallelized quite nicely with respect to cores, but there was still further tweaking to be done with respect to chunk size, especially since our speedup of 3x was not yet near the theoretical bound of 5x.

E. Chunking and Granularity

As a result, we added to our I/O functionality an input parameter that represents the number of chunks that the user

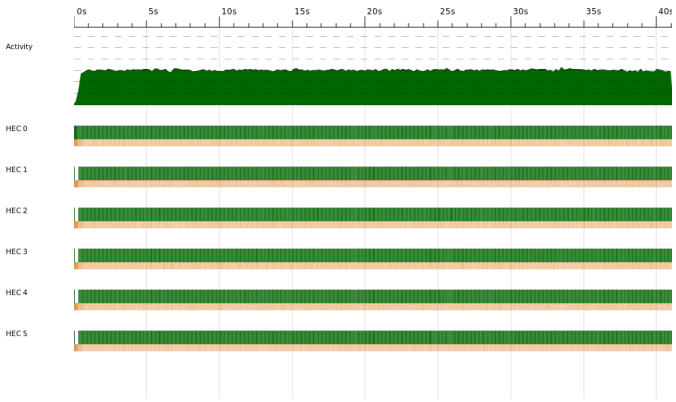

Fig. 2. Threadscope output of parallelized Gradient Descent Algorithm (1,000 iterations) with 128 chunks on a 2 column by 100,000 row input and 6 cores.

wishes to "break" their parallel computation into. While static partitioning ultimately worked great, we now needed to make a design choice due to this addition of chunking; specifically, we needed to figure out the ideal choice for number of chunks, or how many chunks to split the original list into. While we strongly suspect that the optimal answer depends on both the size of the input and the number of cores given to the program, we held those constant (at 2 columns, $100,000$ rows and 6 cores, respectively), and examined the effect chunk size had on runtime. All tests were performed on the same machine that was used for testing multicore performance, with the same input and same number of iterations:

TABLE III
CHUNKS AND RUNTIME

Chunks	Runtime (s)
8	489
16	259
32	122
64	73
128	45
256	35
512	31
1024	29
2048	28
4096	62
6144	176

What is particularly interesting here is that although the runtime initially decreases with granularity, it once again increases after the granularity hits a certain point. We came up with two hypotheses for this phenomenon: either too many sparks were being created, or using more chunks beyond a certain point actually decreases the amount of work done in parallel, because the result of all of the parallel computations is computed sequentially (and thus after a certain number of chunks, we reach a bottleneck due to the amount of sequential computation). Upon examining Threadscope (the output for 6144 chunks is shown in Fig. 4), it appeared to be the latter: every spark that was created converted, which means that there was a significant increase in the amount of sequential work

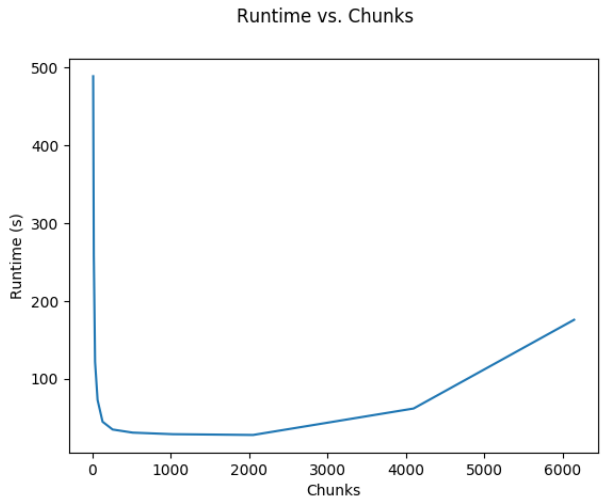

Fig. 3. Runtime vs. Chunks of our parallelized Gradient Descent Algorithm (1,000 iterations) with 6 cores on a 2 column by 100,000 row input.

being done. In fact, once the granularity reaches its maximum, the parallel algorithm essentially becomes the sequential algorithm.

Fig. 4. Threadscope output of parallelized Gradient Descent Algorithm (1,000 iterations) with 6144 chunks on a 2 column by 100,000 row input and 6 cores.

Again, we do strongly suspect that the exact optimal number of chunks depends on the CPU, the number of threads, and the input size. However, it appears that roughly 2048 chunks would be the most optimal given these conditions. Our runtime for this number of chunks hovered just under 30s, which gives slightly more than a 4x speedup and approaches the elusive 5x theoretical bound.

IV. Conclusion

The Gradient Descent algorithm proved to lend itself very well to parallelism. We were able to get it working for two loss functions, one of which is solvable analytically and the other of which is not, and it worked quite well with our CSV input. We got roughly a 40x speedup over the sequential algorithm, which we consider to be a massive success. The speedup increased with input size, as more and more of the work done by the algorithm could be parallelized. Using Amdahl's Law, we calculated a 5x theoretical upper bound for the speedup when given more cores (since the fold operation comprised roughly 80% of the computational workload), and by giving the program access to more cores and tweaking the granularity of the parallelism with the chunk size, we were able to achieve slightly over a 4x speedup. It is likely not possible to achieve the 5x speedup in real life, since Amdahl's Law assumes perfect parallelism, which does not exist. We noticed some interesting phenomena, such as the runtime initially dropping but then increasing as the granularity of the parallelism is increased. **REPA** appeared to be a good choice at the beginning, but turned out to eliminate the need for parallelism in the first place due to Amdahl's Law. Furthermore, the benefit of adding more cores seemed to slow down significantly around 4 or so. While we ultimately did not have time to do so, it would have been interesting to test the effects of parallelism on high dimensional data, which would involve parallelizing "horizontally" as well as vertically (i.e. across columns of our Dataframe), and dealing with a whole host of new granularity issues. Furthermore, we note that we had to tweak gradient-related parameters such as step size (our learning rate), number of steps, and tolerance level by ourselves in order to avoid numerical overflow and other potential runtime issues. Given the time, this is something that we would like to investigate more in the future. Finally, we plan on making the project open source and encourage anyone to contribute to it.

V. Usage and Testing

The program can be built using the **stack build** command from inside the project directory. Then, the **stack run** command can be used in the following manner: **stack run** <**filename**> <**loss function: linear/logistic**> <**guess array**> <**parallel/sequential**> <**number of chunks**>, where each option corresponds to a parameter of the user's choice. The parameters are, filename (with correct path), choice of loss function (using keywords linear/logistic, case-insensitive), a guess input for parameters, keyword "parallel" or "sequential" for the method of computation, and the corresponding number of chunks (Note: this number is ignored if a sequential implementation is chosen; however, it must be passed in). We have also integrated an automatic data generation script that can allow the user to build their own datasets of any size for testing/experimentation purposes. For more details on usage, please consult our **README**.

One thing that we noticed immediately upon initial testing and comparison of outcomes post-parallelization was just *how much faster* our parallelized algorithm was! We therefore included a mini test suite of unit tests (run **stack test**) that use the **TimeIt** Haskell library. Here, we have provided a scaffolding wherein we output the amount of time it takes to run the actual descent algorithm. We have provided a few cases, each with its own particular dataset size, and have run the sequential and parallel implementations for each, as an opportunity for comparison. We wanted to include this to just appreciate the differences in runtime that we have observed - and as the dataset size increases, so do the differences between our sequential and parallel implementations. See Section III.C for a snapshot of these results.

```haskell
module Main where

import Grad
import System.Environment(getArgs)
import System.Exit(die)
import Data.List(isInfixOf)

{- |
Module      :  <File name or $Header$ to be replaced automatically>
Description :  Parallelized Gradient Descent algorithm for linear regression
Copyright   :  (c) <Max Helman, Riya Chakraborty>
License     :  BSD 3-Clause

Maintainer  :  mhh2148@columbia.edu, rc3242@columbia.edu
Stability   :  stable
Portability :  portable
-}

main :: IO()
main = do
        args <- getArgs
        input <- case args of
           [f, method, guess, parseq, chunks] -> return [f, method, guess, parseq, chunks]
           _ -> do
                die $ "Usage: grad-descent <filename> <loss function: linear/logistic> <guess array> <parallel/sequential>
<number of chunks>"
        csvData <- getCSVData (head input)

        let linMatch = or $ map ($ (head $ tail input)) (map isInfixOf ["Linear", "linear", "LINEAR"])
        let logMatch = or $ map ($ (head $ tail input)) (map isInfixOf ["Logistic", "logistic", "LOGISTIC"])

        appLoss <- case (linMatch || logMatch) of
            True -> if linMatch then (return computeGradRowLinear) else (return computeGradRowLogistic)
            False -> do
                    die $ "Choose either Linear or Logistic loss functions"

        let guess = read (head $ tail $ tail input) :: [Double]
        let choice = last $ init input
        let chunkNum = read $ last input :: Int

        print $ descendSteps choice chunkNum csvData appLoss guess (1000::Int) (0.0000001::Double)
--          print $ descendSteps choice chunkNum csvData appLoss guess (10000::Int) (0.001::Double)
```

```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
```

```
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))
```

```haskell
import Test.Hspec
import Grad
import System.TimeIt



main :: IO ()
main = hspec $ do

  describe "Testing Gradient Descent" $ do
    it "Parallel - 100000 rows" $ do
        csvData <- getCSVData "data/test-5.csv"
        output <- timeItT $ (descendSteps "parallel" 64 csvData computeGradRowLinear [0,0] (1000::Int)
(0.00000000000000001::Double)) `seq` return ()
        let computeTime = fst output
        print $ computeTime
        computeTime `shouldSatisfy` (<=(300.0::Double))


    it "Parallel - 10000 rows" $ do
        csvData <- getCSVData "data/test-4.csv"
        output <- timeItT $ (descendSteps "parallel" 64 csvData computeGradRowLinear [0,0] (1000::Int)
(0.0000000000001::Double)) `seq` return ()
        let computeTime = fst output
        print $ computeTime
        computeTime `shouldSatisfy` (<=(10.0::Double))


    it "Sequential - 10000 rows" $ do
        csvData <- getCSVData "data/test-4.csv"
        output <- timeItT $ (descendSteps "sequential" 64 csvData computeGradRowLinear [0,0] (1000::Int)
(0.0000000000001::Double)) `seq` return ()
        let computeTime = fst output
        print $ computeTime
        computeTime `shouldSatisfy` (<=(300.0::Double))


    it "Parallel - 1000 rows" $ do
        csvData <- getCSVData "data/test-3.csv"
        output <- timeItT $ (descendSteps "parallel" 64 csvData computeGradRowLinear [0,0] (1000::Int) (0.0000000001::Double))
`seq` return ()
        let computeTime = fst output
        print $ computeTime
        computeTime `shouldSatisfy` (<=(10.0::Double))


    it "Sequential - 1000 rows" $ do
        csvData <- getCSVData "data/test-3.csv"
        output <- timeItT $ (descendSteps "sequential" 64 csvData computeGradRowLinear [0,0] (1000::Int)
(0.0000000001::Double)) `seq` return ()
        let computeTime = fst output
        print $ computeTime
        computeTime `shouldSatisfy` (<=(300.0::Double))


    it "Parallel - 100 rows" $ do
        csvData <- getCSVData "data/test-2.csv"
        output <- timeItT $ (descendSteps "parallel" 64 csvData computeGradRowLinear [0,0] (1000::Int) (0.0000001::Double))
`seq` return ()
        let computeTime = fst output
        print $ computeTime
        computeTime `shouldSatisfy` (<=(10.0::Double))


    it "Sequential - 100 rows" $ do
```

```
    csvData <- getCSVData "data/test-2.csv"
    output <- timeItT $ (descendSteps "sequential" 64 csvData computeGradRowLinear [0,0] (1000::Int) (0.0000001::Double))
`seq` return ()
    let computeTime = fst output
    print $ computeTime
    computeTime `shouldSatisfy` (<=(10.0::Double))
```