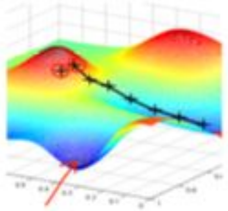
A Parallelized Gradient Descent Algorithm for Regression Coefficient Estimation on Massive Data

Max Helman and Riya Chakraborty



Motivation

-Commonly used in statistical machine learning for estimating regression coefficients

-Big data is taking over the world, and one of the most common things to do is to fit regression lines to these datasets and find trends

-Computing a gradient involves taking a large sum; this computation is repeated many times

The Gradient Descent Algorithm

-We focused on 2 classes of loss functions

-Linear least squares (analytical solution as a sanity check)

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

-Cross entropy logistic (no analytical solution)

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

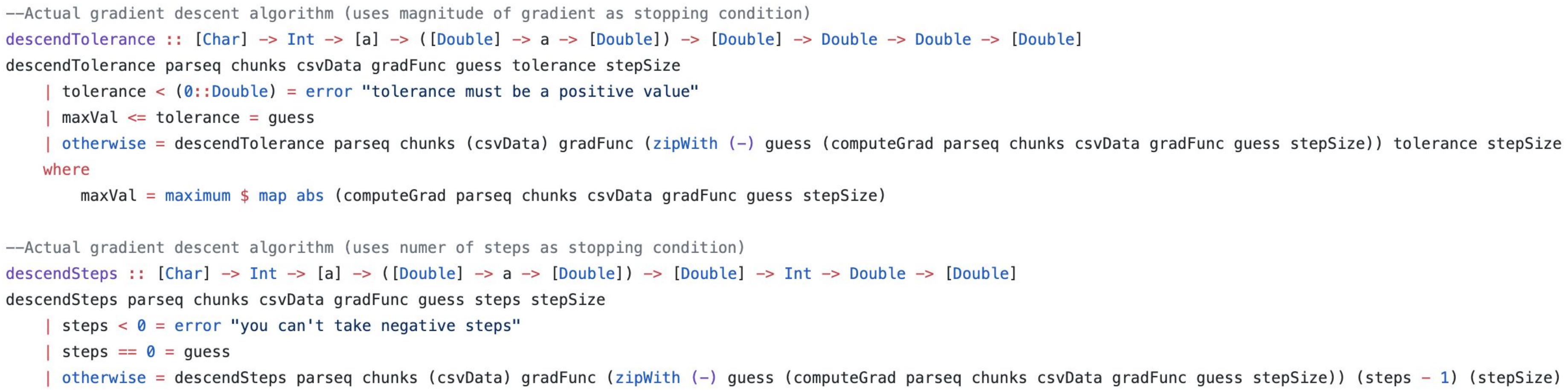
-”Pick any arbitrary point for some convex function, subtract the gradient of the loss function (times some step size), and repeat for a finite number of steps or until you get convergence”

The Gradient Descent Algorithm

-Interesting note about logistic: no numerical convergence, but loss function decreases

-Spent time fine-tuning the learning rate (step size), number of steps/tolerance

-Otherwise, all behavior is as expected



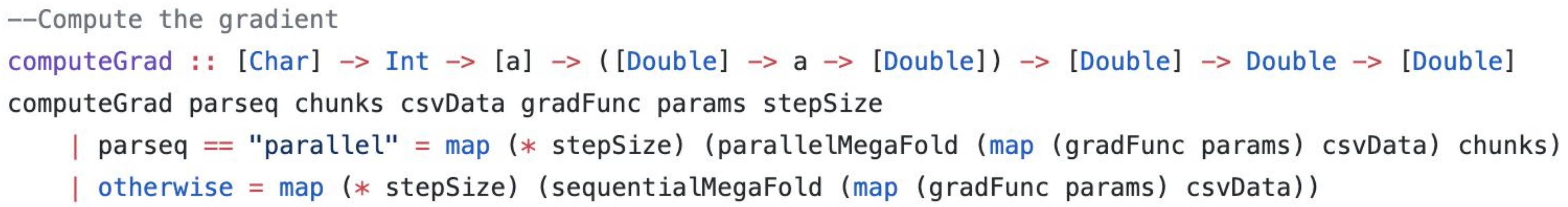
```haskell
--Actual gradient descent algorithm (uses magnitude of gradient as stopping condition)
descendTolerance :: [Char] -> Int -> [a] -> ([Double] -> a -> [Double]) -> [Double] -> Double -> Double -> [Double]
descendTolerance parseq chunks csvData gradFunc guess tolerance stepSize
    | tolerance < (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)
```

Our Program

-Reads in data from CSV and stores in memory as **[[Double]]** (which we refer to as a Dataframe)

-At each step, computes the gradient with a **map** (**gradFunc**, which is essentially a function that computes a row of the gradient) and a **fold** (**sequentialMegaFold**, which essentially sums up all of the columns)



```haskell
--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))
```

Parallelism - First Attempt

-Transformed Dataframe “back and forth” using REPA arrays, transpose to accommodate the fold across rows

```haskell
parallelComputeSum :: [[Double]] -> [Double]
parallelComputeSum nested@(x:xs:xss) = do
                    let x = fromListUnboxed (Z :. ((length nested)::Int) :. ((length $ (head nested))::Int) ) (concat nested)
                    let xTranspose = transpose2D x
                    let [xNDTranspose] = computeP xTranspose :: [Array U DIM2 Double]
                    let mResult = foldP (+) 0 xNDTranspose
                    result <- mResult
                    toList result


transpose2D :: (Source r e) => Array r DIM2 e -> Array D DIM2 e
transpose2D a = backpermute (swap e) swap a
     where
       e = extent a
       swap (Z :. i :. j) = Z :. j :. i
```

Parallelism - New Attempt

-Gave it away in the last slide, but we parallelized the fold (aka the summation) in a function called **parallelMegaFold**

-**sequentialMegaFold** took up **~ 98%** of the computation in the sequential algorithm*, and **parallelMegaFold** took up ~ **82%** of the computation in the parallel algorithm

-Theoretical speedup of sequential algorithm using parallelism ~ **50x**

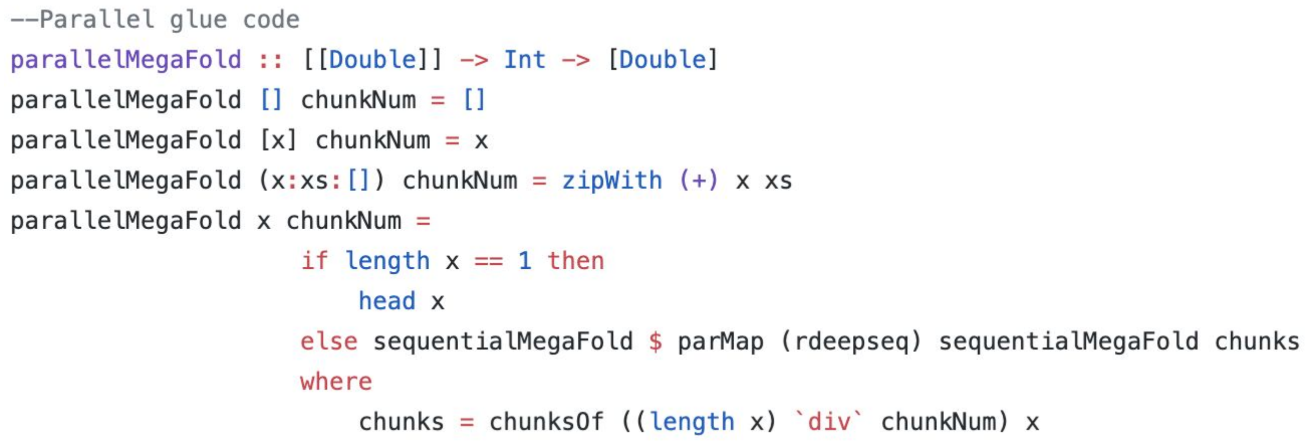
-Theoretical speedup of parallel algorithm (using Amdahl's law) ~ **5x**

*increases with input size; tested on 10,000 row input

Parallelism

-Settled on **static partitioning**, since we know each chunk of work takes the same amount of time

-Basically, partition list, compute sums of partitions with **parMap** and **sequentialMegaFold**, and then **sequentialMegaFold** the results



```haskell
--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
```

Performance

-128 Chunks, 2 columns x 100,000 rows

-6th Generation Intel Core i5 (6 cores, overclocked to 4.19 GHz)

-16GB 2133 MHz DDR4 RAM

-250GB Samsung 850 EVO SSD

-Ubuntu 20.04

-I know my PC is too powerful just to be used for tasks like this. I am a Fundies TA and spend too many of my TA paychecks on upgrades and watercooling. I like to think it’s in the spirit of the class?

Performance: Parallel vs. Sequential

-Nearly **45x** speedup over sequential algorithm (when both are given 6 cores)

-Speedup increases with input size

-This is a big speedup that seemed too good to be true. But we parallelized an operation that took **~98%** of the computation, which is a lot of work

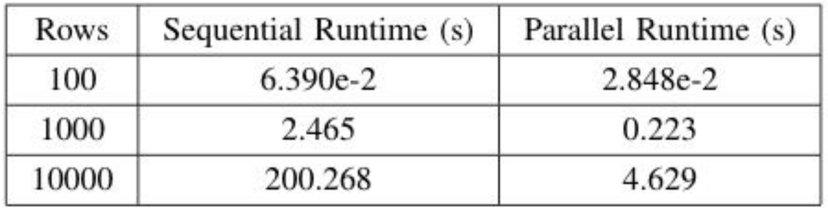
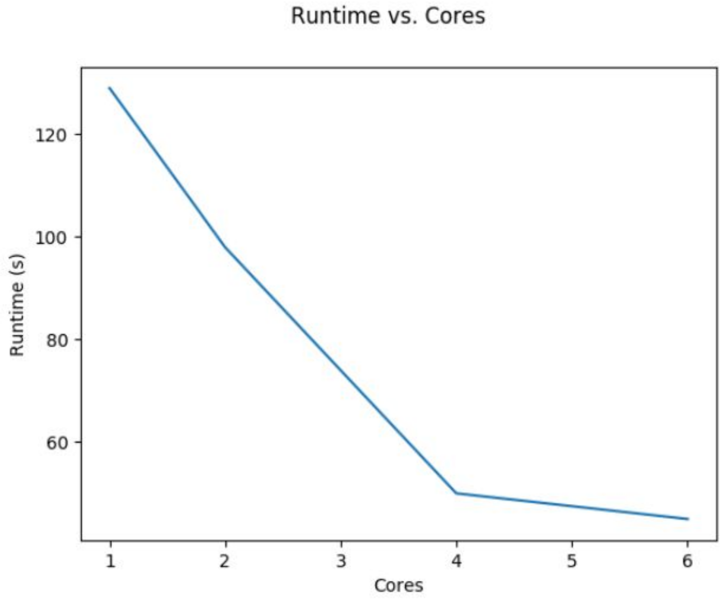


TABLE I
UNIT TESTING - SEQUENTIAL AND PARALLEL RUNTIMES

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

Performance: Cores

-Roughly a **3x** speedup for parallel algorithm just by giving it more cores



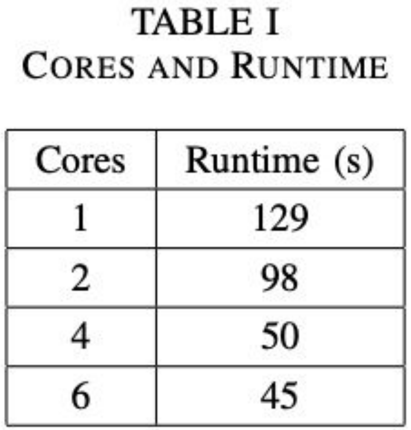


TABLE I
CORES AND RUNTIME

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

Performance: Cores

-Threadscope looks pretty legit to us! Even though it was a pain to install

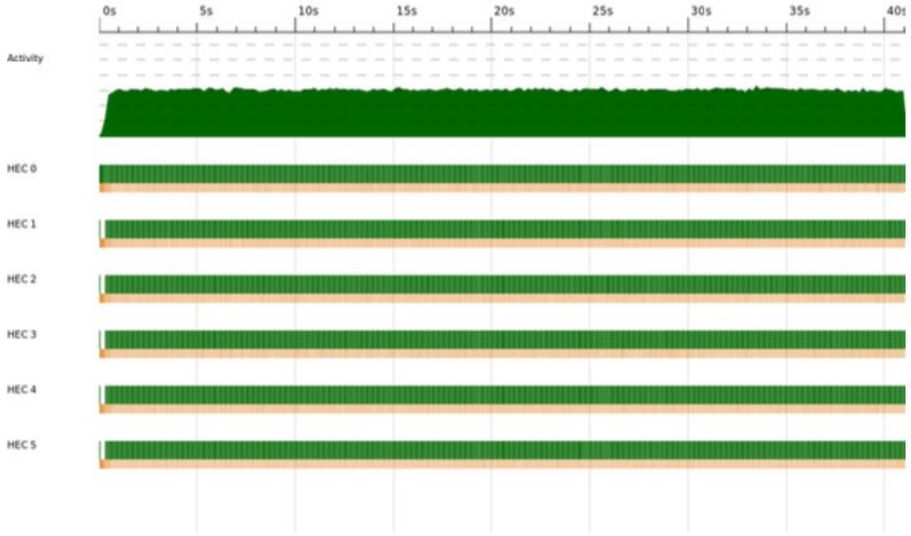


Fig. 2. Threadscope output of parallelized Gradient Descent Algorithm (1,000 iterations) with 128 chunks on a 2 column by 100,000 row input and 6 cores.

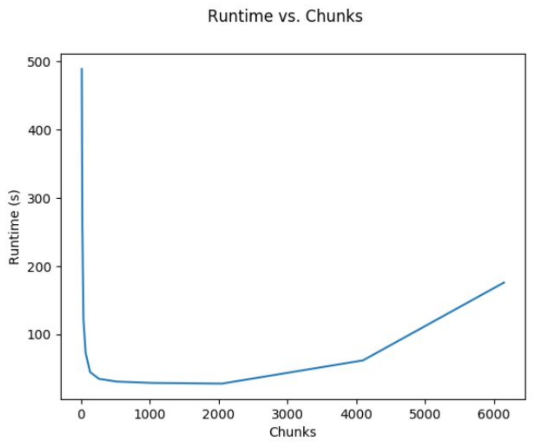
Performance: Cores

-Pretty good, but we can do better by playing with granularity

Performance: Chunking

-Interesting phenomenon: as we add more chunks, the runtime decreases, but then it increases again

-We were ultimately able to get a **4x** speedup over the sequential algorithm by playing around with granularity (**129 seconds -> 29 seconds**)



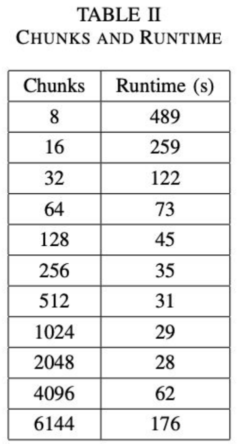


TABLE II
CHUNKS AND RUNTIME

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

Performance: Chunking

-As number of chunks increases, chunk size decreases

-Past a certain point, the parallelism becomes so fine grained that it is essentially the sequential algorithm (i.e. mapping to chunks that are only 1 row)

-We strongly suspect optimal number of chunks depends on specific setup: input size, number of cores, etc

Performance: Chunking

-Threadscope confirms that sparks are not an issue; rather, the work done in parallel is just very little (most work done on one core)

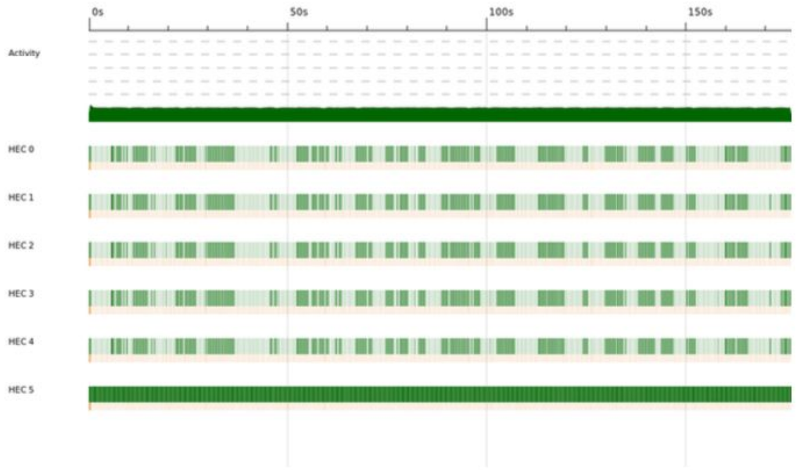


Fig. 4. Threadscope output of parallelized Gradient Descent Algorithm (1,000 iterations) with 6144 chunks on a 2 column by 100,000 row input and 6 cores.

REPA and Amdahl’s Law

-REPA is a good library. Sometimes, it’s a little too good.

-**REPA arrays** are much faster than Haskell lists, to the point where our fold operation only took up roughly **~12%** of our computation time, and our theoretical maximum speedup was **~1.2x**

-The conversions between lists and **REPA arrays** and transposes actually took up a majority of the runtime (rendering the effects of any parallel operations minimal)

-So, for the purposes of this project, we stuck with Haskell lists

Conclusion

-We got a **43x** speedup over the sequential algorithm with 128 chunks. That exceeded even our wildest expectations at the beginning of the project. Speedup increases with input size, and is thus VERY scalable

-Parallelism worked very well here! The parallel algorithm got a **4x** speedup by adding more cores and tweaking granularity, and our theoretical maximum was **5x**. More granularity was only better up to a certain point; more cores had diminishing returns

-Haskell stack is cool; installing Threadscope is not

-Amdahl’s Law came back to haunt us

Moving Forward

-Project will be made open source! But we wanted you guys to see it first

-Could try to parallelize across columns; might help with higher dimensional data, but also might generate too many sparks since there are still a lot of rows

-One thing that could be interesting to play with would be to optimize the parameters we are using for gradient descent (step size, number of steps, tolerance)

-We can invent perfect parallelism to achieve our maximum theoretical speedup! But in all seriousness, I do want to learn more about GPU computing

    -You should make a PFP part 2 :)

    -I also ordered a book on CUDA programming, per your and Prof. Kim’s recommendations

Acknowledgements

Thank you Prof. Edwards! We really appreciate that you took the time to meet with us. We are both juniors, and have found the opportunities for interaction with professors so far to be limited, so this was a great experience for us. Plus, we both signed up for your PLT class next semester… so we are not done with the suffering yet!

Please stay in touch!
Via FB/LinkedIn/etc

Acknowledgements

Benjamin Flin is a saint. He stayed on with us for 3 hours helping us determine a parallelization strategy and debugging our program. He deserves a raise.

Thank you again! And have a nice, safe, and restful break! Happy grading - Max and Riya