Parallelization of Linear Regression

A PROJECT REPORT

Submitted by

15BCB0049 - Debashis Karmakar

15BCE0519 – Rajesh Kumar Singh

NOVEMBER 2017

ABSTRACT:

In this paper we propose a strategy for tackling the direct minimum squares issue, through parallelization of the normal Gradient Descent calculation. We try to present the defense for the iterative Gradient Descent approach over various investigative techniques. The issue of parallelizing direct minimum squares minimization is investigated, alongside examination of current systems. A parallel Gradient Descent arrangement is proposed and the strategies utilized in our usage are nitty gritty. The case for our calculation is made through examination of procedures for theoretical runtime situations, with qualities and shortcomings of each point by point. The proposed parallel inclination plummet strategy is actualized in MPI and is benchmarked against a practically equivalent to consecutive form, to show the enhanced productivity.

INTRODUCTION:

In a significant number of the most widely recognized calculations in measurable examination and machine adapting, for example, different direct relapse, neural systems, and numerous others, the minimization of a mistake esteem is understood. A typical measure of mistake is the SSE(Sum of Squared Error) metric. Characterizing the SSE as the mistake metric makes the motivation behind such calculations to diminish the SSE however much as could reasonably be expected. This is a minimization issue and is accomplished via hunting the space down the worldwide minima. Various arrangements exist for this issue, and every fall in to one of two classes: iterative or systematic. Expository strategies commonly have extremely poor execution for expansive datasets, where the iterative arrangements have a tendency to perform better. Be that as it may, on the planet we live in today, information stores are proceeding to progress and the volumes of information close by are ending up less wieldy, hence making both minimization approaches deficient when keep running in a totally consecutive way. Parallelization is critical for managing, for example, C programming.

The way to parallelize minimization calculations is significantly more clear on account of the investigative procedures, as most are finished utilizing some mix of straightforward network operations, for example, duplication, which loan themselves to parallelization decently obviously. The way isn't so direct with iterative procedures, for example, Gradient Descent. The innate successive nature appears not to fit parallelization, as each progression must be performed in strict request. Be that as it may, a little element of the Gradient Descent calculations, in particular the learning rate, which firmly decides the execution of the calculation, makes an unmistakable entrance to parallelization.

In this paper we detail a strategy of actualizing a parallel inclination drop utilizing a SPMD (Single Processor Multiple Data) engineering, which will beat parallel investigative renditions for expansive datasets. The strategy will be investigated with regards to the issue of Multiple Linear Regression.

Multiple Linear Regression

In uncountable situations, the accompanying issue emerges: How would we figure out how to take in the relations of a vector of mindependent factors (x) to a solitary ward variable (y)? By and by, the issue is understood byattempting to discover a vector of coefficients (θ), to such an extent that the speculation $h_\theta(x) = \theta_0 + \theta_1 x_1 + \cdots + \theta_m x_m$ limits the SSE between h_θ (x) and y. The SSE is characterized as$(h_\theta(x) - y)^2$.

The reason for existing is to take in a capacity which, when given another x vector precisely predicts the relating y esteem, inside worthy limits. Practically speaking, a calculation would be given an arrangement of preparing information, comprising of a network X ofm highlights and n illustrations, and a vector yof n reaction factors. Its will probably yield the θ vector, with which to make ensuing expectations.

The current calculations for this issue are either incremental or explanatory in nature. The expository arrangements understand for the θ esteems that limit the SSE in one go, utilizing diverse varieties of the Normal Equation:

$$\theta = (X^T X)^{-1} X^T y$$

The incremental arrangements make stages each one in turn towards the base, assessing their advance at every walk en route.

LITERATURE REVIEW:

Minimization Methods

In each vein of arrangements various varieties exist. Each vein has its advantages and inside them there is variety in critical thinking strategies for shifting issue sort.

Arrangements of the diagnostic sort limit by registering the ordinary condition. Registering the typical condition comes down to playing out various lattice operations, which prompts expansive computational multifaceted nature, considering grid augmentation itself is an $O(n^3)$ operation. Practically speaking, the typical condition isn't explained for straightforwardly as the $(X^T X)^{-1}$ factor is entirely restrictive. One arrangement is to figure a covariance framework for X and y, and decide the eigenvalues, eigenvectors, and eigenvector of best fit. This strategy is utilized as a part of a contemporary GPU parallelization [3]. Another arrangement, called a QR disintegration, is to break down the X grid into a mix of an orthogonal lattice Q and an upper triangle network R. This technique is utilized as a part of two more contemporary parallel calculations for CPUs [1] and GPUs [2]. Techniques that utilization the ordinary condition are extremely clear, and stay helpful for littler volumes of information, however information volumes develop the framework operations in that wind up plainly restrictive.

Arrangements of the iterative sort, for example, Gradient Descent, Stochastic Gradient Descent, and other GD varieties limit by processing the fractional subsidiaries of θ esteems, regarding the SSE, and venturing down the slope toward the base at every emphasis. Inclination plummet keeps running in O(pn+kn) where p is the quantity of highlights, and k is the quantity of steps it takes to meet.

Slope plummet has the hindrance of succumbing to imperfect minima union, and practically speaking has a tendency to be less exact than the expository techniques. In any case, practically speaking GD tends to create comes about substantially speedier, and it isn't hard to perceive any reason why: an $O(n^3)$ totally overshadows an O(n) arrange calculation as information volumes develop.

In contrasting the two strategies, Gradient Descent appears like the undeniable decision for preparing on huge datasets. For slope plummet to keep running as inadequately as alternate strategies, the quantity of ventures for it to join would need to approach n^2, which turns out to be progressively improbable, as n develops substantial. Notwithstanding, with the present sizes of information that are getting to be plainly standard for this kind of issue, neither of the arrangements is adequate in a solitary procedure, successive frame. Parallelization is urgent, and the typical condition loans itself splendidly to it, settling on it the more evident decision for parallelization.

Parallelization can offer some factor of speedup to consecutive forms of calculations, and since Gradient Descent is characteristically successive, contemporary parallel arrangements actualize the ordinary condition.

Two strategies, one actualized utilizing the MapReduce demonstrate [1] and one executed utilizing Nvidia's CUDA dialect for use on their GPUs [2], utilize cunning dividing methods to run a QR decay crosswise over procedures. QR disintegration is a technique for breaking down a framework into a result of an orthogonal lattice Q, and an upper triangle network R. The QR decay can be played out various ways, as a rule utilizing a progression of projections or reflections. These projection techniques present a component of grouping that expels a portion of the charm of the ordinary condition for parallelization. Astute hacks are currently important to run the calculations in parallel, rather than being an impeccable fit. For example, a typical strategy is to utilize Householder reflections, a procedure that makes various middle frameworks on its approach to creating Q and R. This is the strategy utilized as a part of [2]. Utilizing Householder reflections however presents the need of creating grids in strict arrangement. The creators of the GPU usage figured out how to enhance the reflections a bit by playing out the appearance in squares, yet the full advantage of the GPU is quite recently not ready to be acknowledged for QR deteriorations, and their execution delivered speedup by about a factor of 5, over the successive rendition. For substantial issue sizes the runtime is as yet restrictive.

An expository technique that does not utilize the QR disintegration exists and furthermore can be executed in parallel. A covariance network is figured amongst X and y, and from that, eigenvalues are resolved, eigenvectors made, and the eigenvector of best fit picked. This is the technique utilized as a part of another GPU and CUDA based calculation [3]. The creators could accomplish speedup of a factor of ≈1000 over the successive rendition. In any case, the way toward processing a covariance network alone is $O(n^3)$. By and by, as issue sizes become tremendous, the impact of division by a steady begins to vanish.

Along these lines, for enormous issue sizes Gradient Descent is the undeniable decision. In an issue of size $n = 10^6$, for GD to keep running as ineffectively as ordinary condition strategies, it would need to find a way to achieve its objective, a far-fetched situation. At the point when the distinction is by a factor of a trillion, any steady speedup prone to be accomplished through parallelization is probably going to be exceeded.

All things considered, the characteristically consecutive nature of Gradient Descent is disturbing, and a few methods for accelerating execution through parallelization is wanted. Through abuse of the learning rate, and its rate of progress, nonlinear speedup is conceivable.

METHODOLOGY:

➢ Method for parallelizing Gradient Descent

The update step for the Gradient Descent algorithm is as follows:

$$\theta_{iteration\ i} = \theta_{iteration\ i-1} - \alpha(\nabla(SSE))$$

where α is the size of step to be taken down the gradient

The progression size of the Gradient Descent usage is pivotal to its execution. On the off chance that too huge of a stage measure is picked, the calculation experiences a re-rectification issue. At minima, if the progression measure is too extensive, the calculation will keep on overcorrect, becoming further and further from appropriate working at each progression, as the θ esteems and relating blunder esteem dispatch off to vastness. In the event that the progression measure is picked too little, the calculation will take unnecessarily long to focalize, and consequently act sub-ideally. The best decision of learning rate is one that is sufficiently huge to not shoot off to endlessness. On the off chance that it evades that issue, it will unite in less emphasess than if it had a littler esteem. Another component of the learning rate in Gradient Descent is its flexibility constants. Angle Descent experiences the issue of getting to be noticeably stuck in neighborhood minima. One answer for that issue is to execute a versatile learning rate, which changes in light of the past cycle's conduct. On the off chance that a stage down the inclination in the past cycle brought about the blunder developing, the learning rate is most likely too substantial and ought to be diminished by a factor. In the event that a stage down the inclination created appropriate outcomes, and the blunder contracted, at that point it may be sheltered to develop the learning rate by a factor. Appropriate decision of these constants likewise enormously influences the execution of Gradient Descent. Be that as it may, deciding legitimate esteems for these factors early is as yet a theme of research. A contemporary calculation for deciding appropriate advance size, in view of the information, called FASSA (Fast Automatic Step Size Estmiation) [4] exists which makes a decent showing with regards to of picking as expansive a stage estimate as it supposes is shrewd. It gives speedup by a factor of around 2-3, yet those outcomes are just affirmed tentatively, and it isn't ensured to be an ideal esteem. The calculation is likewise a result of the way that with a steady advance size all through, Gradient Descent is ensured to merge, and in this way does not represent variety of the learning rate between emphasess.

With a sensible decision for the learning rate and sensible decisions for the adjustment constants, a near ideal usage can be found. By changing these rates around foreordained sensible esteems, crosswise over handling hubs, one is ensured to have preferable execution over the rest. As the quantity of hubs builds, the hub which merges the speediest will probably have nearer to ideal esteems, with which speedup is significant and exists over a wide range. With a parallel model, in which nodal intercommunication is conceivable, the runtime of the most ideal arrangement turns into the runtime of the whole gathering.

➢ Parallel implementation

Our execution, which is composed in C, utilizing the MPI interface, gives extensive speedup over the successive simple, likewise written in C. Our implementation also manages to get more accurate results.

MPI (Message Passing Interface) is a SPMD programming model that utilizations multi-purpose message going between hubs running their own particular rendition of a program. Our calculation requires a message passing structure, with the goal that all hubs know to stop execution when the primary hub joins on what it considers to be an ideal arrangement, and therefore we picked MPI. Any parallel design in which all hubs have a few methods for passing data, regardless of whether that be through message passing or shared memory get to, would do the trick. The points of interest of the calculation are as per the following.

For a solitary hub, sensible and un-extraordinary esteems for the learning rate, the lessening steady, and the expansion consistent are picked, inclining towards slower running, to keep away from uniqueness. No usefulness is right now display for automatic selection of qualities for the primary hub. For each hub after the primary, the qualities advance gradually toward the more outrageous, in light of the hub's procedure number. All start their own particular variant of slope plunge, varying just in their progression esteems, and run at the same time. Every hub is furnished with a greatest number of cycles. At each progression, every hub meets a boundary to such an extent that they all keep running in bolt step. Upon a hub's merging, it watches that it has a lower blunder an incentive than its kin, and assuming this is the case, communicates a stop message to all hubs in the communicator.

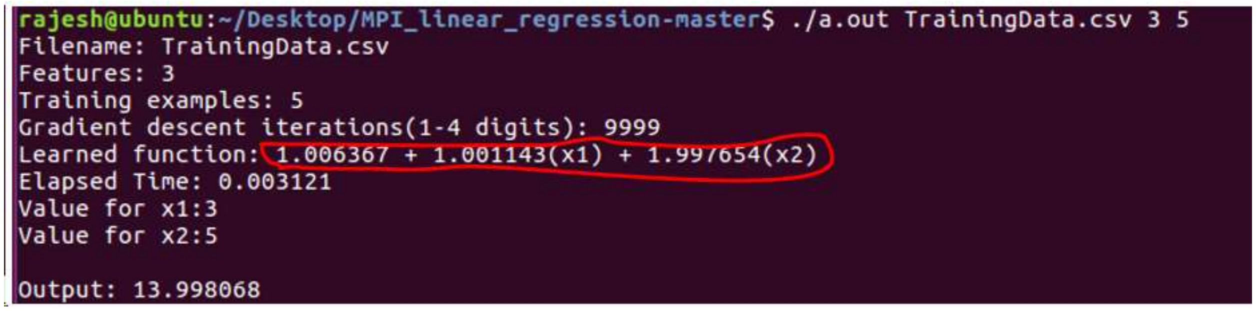
On account of a solitary hub occasion of the MPI rendition, the qualities will be the same with respect to the consecutive form and in this manner run comparatively, short the overhead cost of correspondences. Since deciding the ideal esteems from the earlier is mystery, the parallel execution isn't ensured to create any speedup as hubs are included; the likelihood of expansive speedups just increments in a way difficult to anticipate. In this way, speedup is nonlinear and must be affirmed tentatively.

Our parallel usage was benchmarked against the successive form for a couple of datasets extending from 5 to 100 illustrations. The two adaptations focalized on exact arrangement in all illustrations. The outcomes are as per the following:

Implementation	**Processes**	**Features**	**Examples**	**Runtime(seconds)**	**Speedup**
Sequential	1	3	5	.0138	-
MPI	2	3	5	.00148	9.32
MPI	4	3	5	.0057	2.42
Sequential	1	3	100	.075	-
MPI	2	3	100	.063	1.19
MPI	4	3	100	.060	1.25

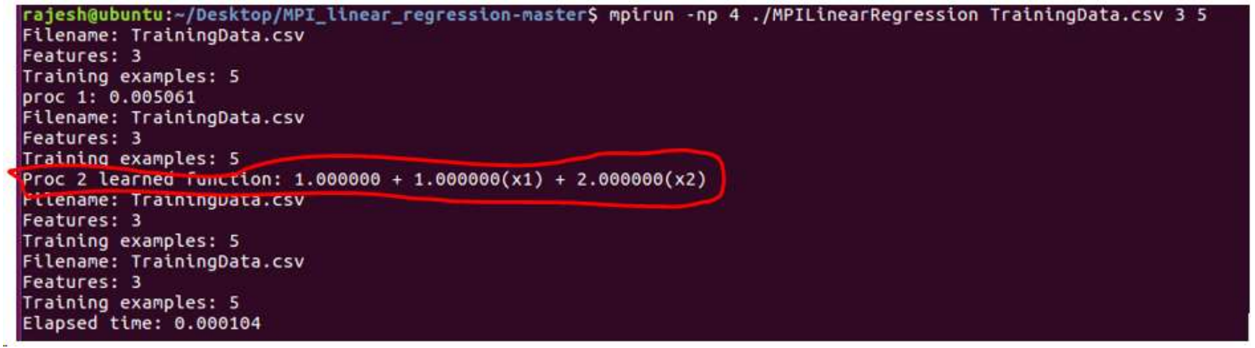
RESULTS:

SERIAL IMPLEMENTATION:



```
rajesh@ubuntu:~/Desktop/MPI_linear_regression-master$ ./a.out TrainingData.csv 3 5
Filename: TrainingData.csv
Features: 3
Training examples: 5
Gradient descent iterations(1-4 digits): 9999
Learned function: 1.006367 + 1.001143(x1) + 1.997654(x2)
Elapsed Time: 0.003121
Value for x1:3
Value for x2:5

Output: 13.998068
```

PARALLEL IMPLENTATION (USING MPI):



```
rajesh@ubuntu:~/Desktop/MPI_linear_regression-master$ mpirun -np 4 ./MPILinearRegression TrainingData.csv 3 5
Filename: TrainingData.csv
Features: 3
Training examples: 5
proc 1: 0.005061
Filename: TrainingData.csv
Features: 3
Training examples: 5
Proc 2 learned function: 1.000000 + 1.000000(x1) + 2.000000(x2)
Filename: TrainingData.csv
Features: 3
Training examples: 5
Filename: TrainingData.csv
Features: 3
Training examples: 5
Elapsed time: 0.000104
```

In the results for both serial and parallel implementations we note two things:

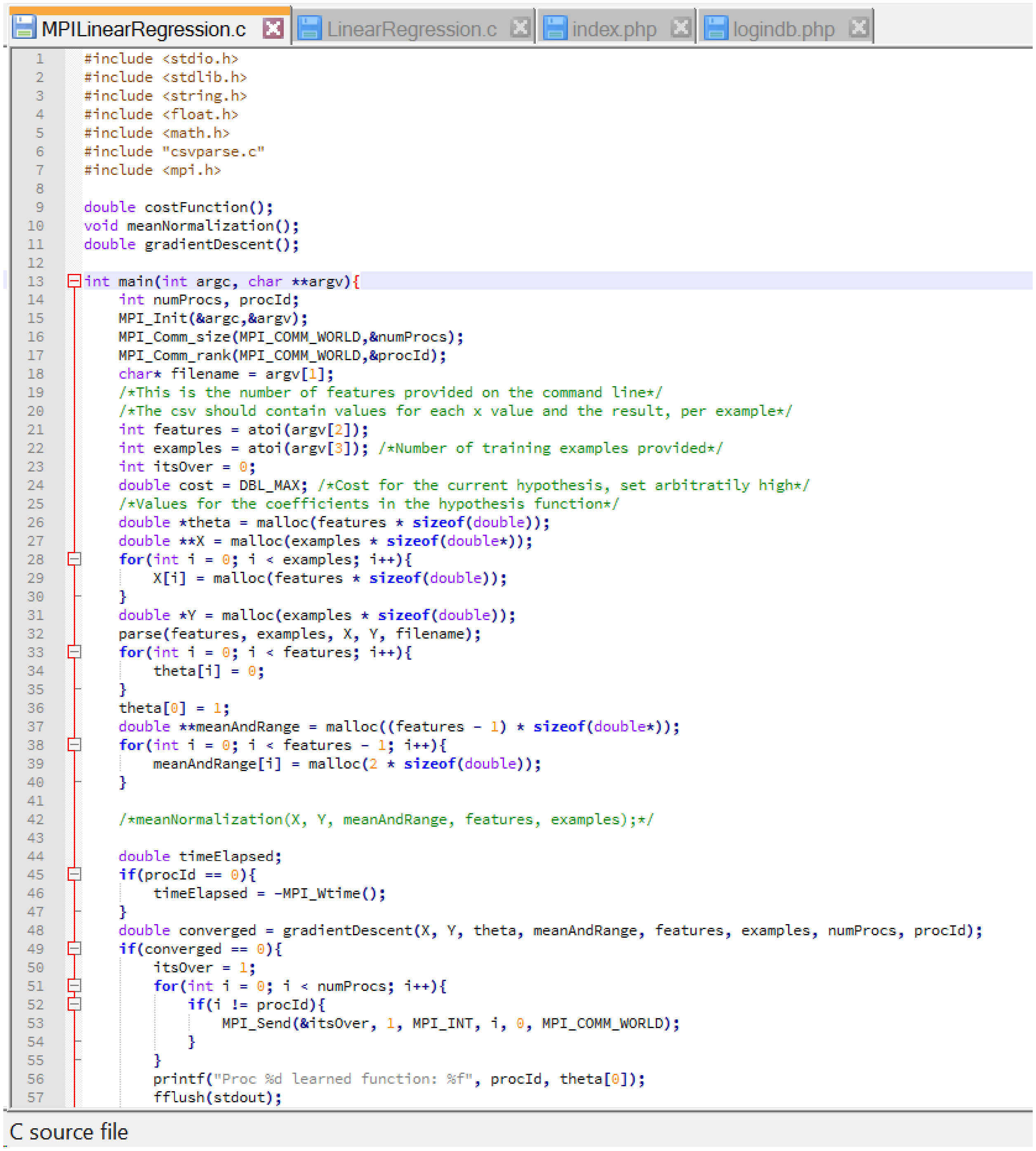
- The elapsed time : for parallel, 0.000104 and serial, 0.003121
- The accuracy of the equation: for serial and parallel, in red marked equation , accuracy is more for the parallel implementation.

CONCLUSION:

Many parallel calculations exist for getting an answer for the Linear Least Squares issue. Be that as it may, most are alterations of scientific arrangements utilizing the ordinary condition. While they offer colossal speedup over their successive analogs, the hidden calculations are still woefully deficient for application to tremendous volumes of information. In spite of the fact that it isn't an issue fit for fulfilling parallelization, the learning rate exhibit in the calculation shows the likelihood, and Gradient Descent ought to be the decision for Multiple Linear Regression. For the correct size of issue, a consecutive form of Gradient Descent will run more effectively than any parallel typical condition execution. With adequate registering power, and appropriate misuse of the learning rate, Gradient Descent can be made to run substantially quicker.

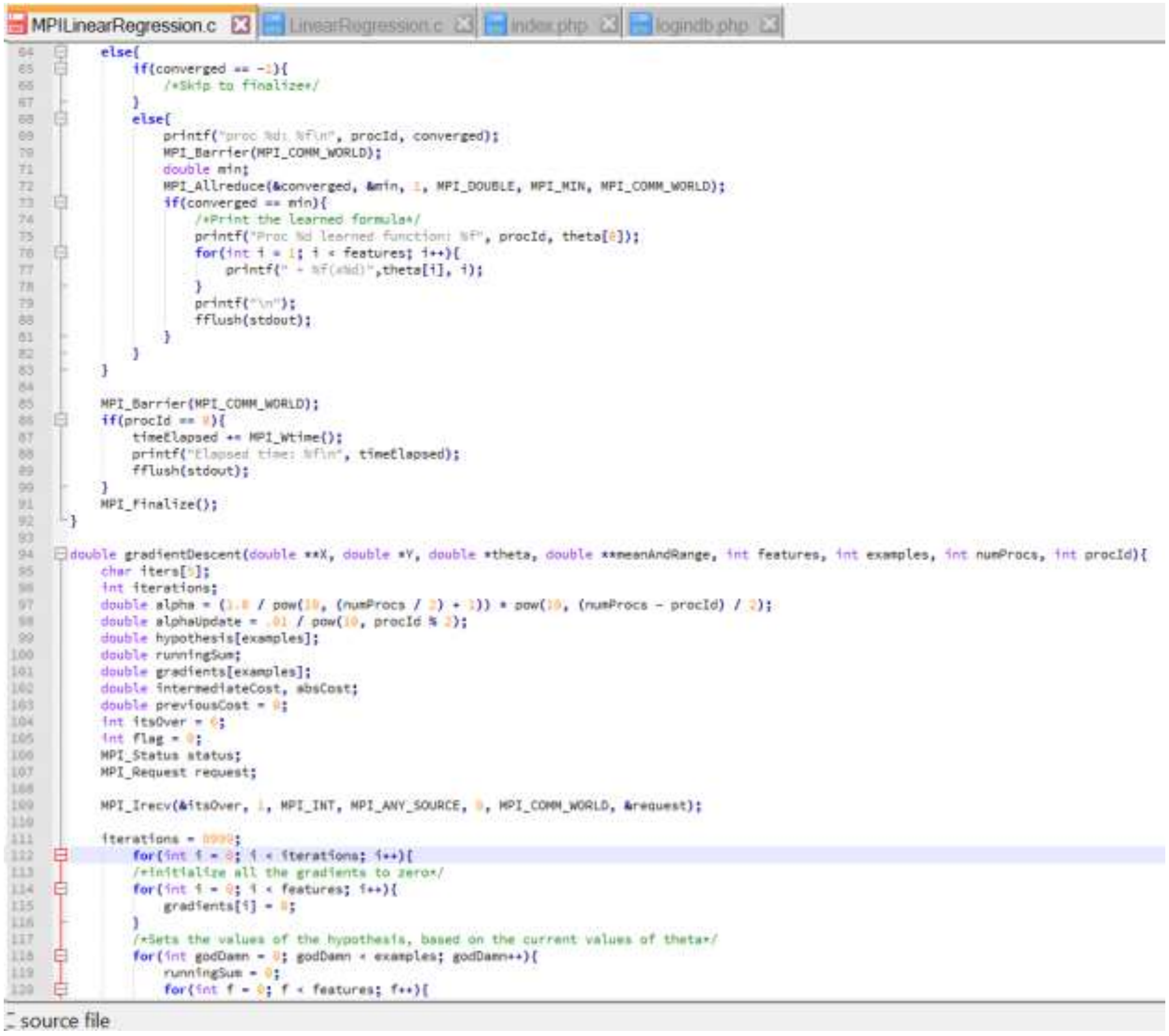
Code:

MPILinearRegression:



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
#include <math.h>
#include "csvparse.c"
#include <mpi.h>

double costFunction();
void meanNormalization();
double gradientDescent();

int main(int argc, char **argv){
    int numProcs, procId;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD,&procId);
    char* filename = argv[1];
    /*This is the number of features provided on the command line*/
    /*The csv should contain values for each x value and the result, per example*/
    int features = atoi(argv[2]);
    int examples = atoi(argv[3]); /*Number of training examples provided*/
    int itsOver = 0;
    double cost = DBL_MAX; /*Cost for the current hypothesis, set arbitratily high*/
    /*Values for the coefficients in the hypothesis function*/
    double *theta = malloc(features * sizeof(double));
    double **X = malloc(examples * sizeof(double*));
    for(int i = 0; i < examples; i++){
        X[i] = malloc(features * sizeof(double));
    }
    double *Y = malloc(examples * sizeof(double));
    parse(features, examples, X, Y, filename);
    for(int i = 0; i < features; i++){
        theta[i] = 0;
    }
    theta[0] = 1;
    double **meanAndRange = malloc((features - 1) * sizeof(double*));
    for(int i = 0; i < features - 1; i++){
        meanAndRange[i] = malloc(2 * sizeof(double));
    }

    /*meanNormalization(X, Y, meanAndRange, features, examples);*/

    double timeElapsed;
    if(procId == 0){
        timeElapsed = -MPI_Wtime();
    }
    double converged = gradientDescent(X, Y, theta, meanAndRange, features, examples, numProcs, procId);
    if(converged == 0){
        itsOver = 1;
        for(int i = 0; i < numProcs; i++){
            if(i != procId){
                MPI_Send(&itsOver, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            }
        }
        printf("Proc %d learned function: %f", procId, theta[0]);
        fflush(stdout);
```

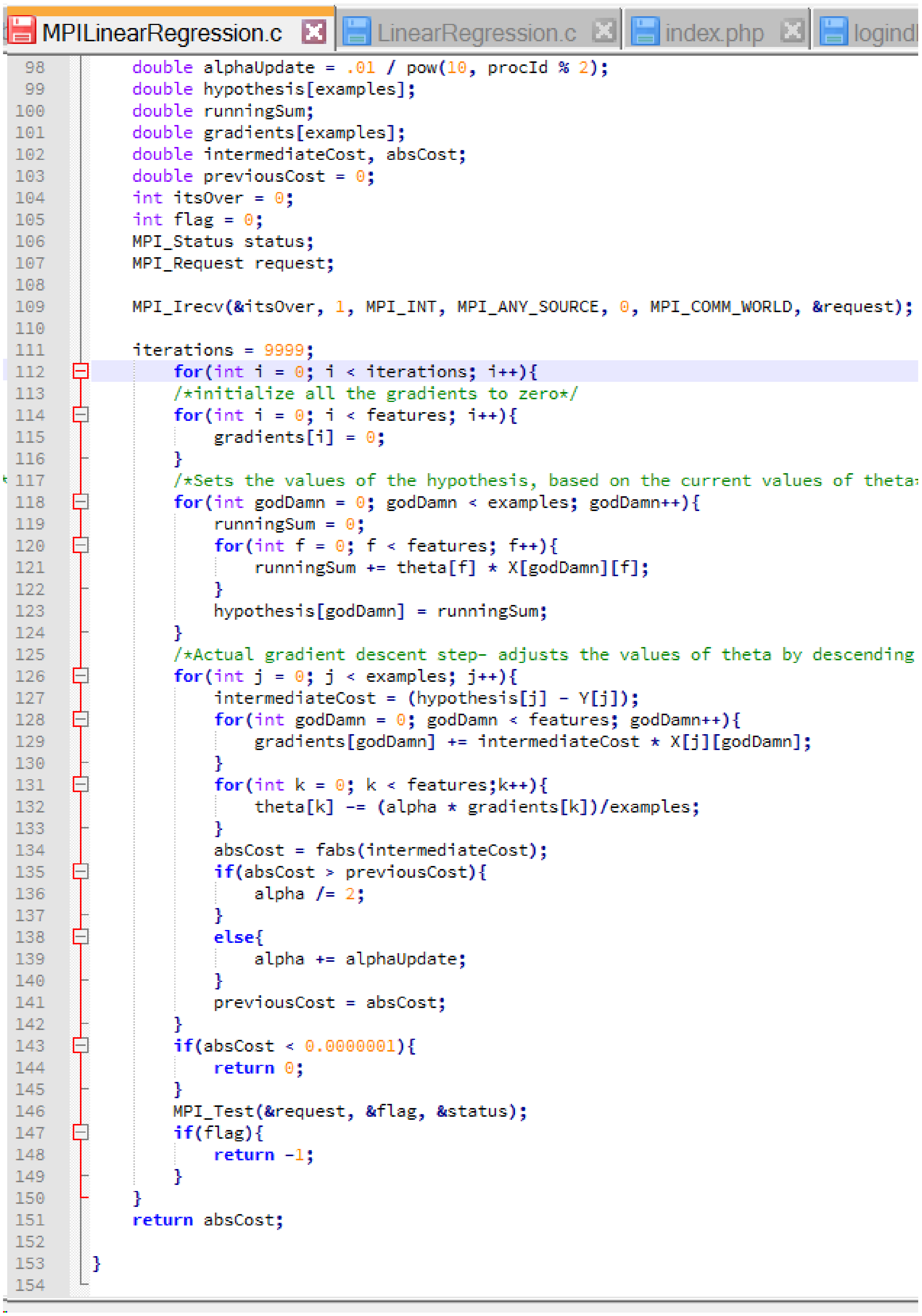
C source file



MPILinearRegression.c | LinearRegression.c | index.php | logindb.php

```c
    else{
        if(converged == -1){
            /*Skip to finalize*/
        }
        else{
            printf("proc %d: %f\n", procId, converged);
            MPI_Barrier(MPI_COMM_WORLD);
            double min;
            MPI_Allreduce(&converged, &min, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
            if(converged == min){
                /*Print the learned formula*/
                printf("Proc %d learned function: %f", procId, theta[0]);
                for(int i = 1; i < features; i++){
                    printf(" + %f(x%d)",theta[i], i);
                }
                printf("\n");
                fflush(stdout);
            }
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
    if(procId == 0){
        timeElapsed += MPI_Wtime();
        printf("Elapsed time: %f\n", timeElapsed);
        fflush(stdout);
    }
    MPI_Finalize();
}

double gradientDescent(double **X, double *Y, double *theta, double **meanAndRange, int features, int examples, int numProcs, int procId){
    char iters[5];
    int iterations;
    double alpha = (1.0 / pow(10, (numProcs / 2) + 1)) * pow(10, (numProcs - procId) / 2);
    double alphaUpdate = .01 / pow(10, procId % 2);
    double hypothesis[examples];
    double runningSum;
    double gradients[examples];
    double intermediateCost, absCost;
    double previousCost = 0;
    int itsOver = 0;
    int flag = 0;
    MPI_Status status;
    MPI_Request request;

    MPI_Irecv(&itsOver, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &request);

    iterations = 9999;
        for(int i = 0; i < iterations; i++){
        /*initialize all the gradients to zero*/
        for(int i = 0; i < features; i++){
            gradients[i] = 0;
        }
        /*Sets the values of the hypothesis, based on the current values of theta*/
        for(int godDamn = 0; godDamn < examples; godDamn++){
            runningSum = 0;
            for(int f = 0; f < features; f++){
```

C source file



```
    double alphaUpdate = .01 / pow(10, procId % 2);
    double hypothesis[examples];
    double runningSum;
    double gradients[examples];
    double intermediateCost, absCost;
    double previousCost = 0;
    int itsOver = 0;
    int flag = 0;
    MPI_Status status;
    MPI_Request request;

    MPI_Irecv(&itsOver, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &request);

    iterations = 9999;
        for(int i = 0; i < iterations; i++){
        /*initialize all the gradients to zero*/
        for(int i = 0; i < features; i++){
            gradients[i] = 0;
        }
        /*Sets the values of the hypothesis, based on the current values of theta
        for(int godDamn = 0; godDamn < examples; godDamn++){
            runningSum = 0;
            for(int f = 0; f < features; f++){
                runningSum += theta[f] * X[godDamn][f];
            }
            hypothesis[godDamn] = runningSum;
        }
        /*Actual gradient descent step- adjusts the values of theta by descending
        for(int j = 0; j < examples; j++){
            intermediateCost = (hypothesis[j] - Y[j]);
            for(int godDamn = 0; godDamn < features; godDamn++){
                gradients[godDamn] += intermediateCost * X[j][godDamn];
            }
            for(int k = 0; k < features;k++){
                theta[k] -= (alpha * gradients[k])/examples;
            }
            absCost = fabs(intermediateCost);
            if(absCost > previousCost){
                alpha /= 2;
            }
            else{
                alpha += alphaUpdate;
            }
            previousCost = absCost;
        }
        if(absCost < 0.0000001){
            return 0;
        }
        MPI_Test(&request, &flag, &status);
        if(flag){
            return -1;
        }
    }
    return absCost;

}
```

References

1. Moufida Adjout Rehab and F. Boufares, “Scalable Massively Parallel Learning of Multiple Linear Regression Algorithm with MapReduce” 2015 IEEE Trustcom/BigDataSE/ISPA conference.
2. Andrew Kerr, Dan Campbell and Mark Richards, “QR Decomposition on GPUs”.
3. Jyoti B. Kulkarni, A. A. Sawant, and Vandana S. Inamdar, “Database Processing by Linear Regression on GPU using CUDA” 2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies (ICSCCN 2011).