Project Report

Shujun Liu (Username: liushuj)

Dec 2019

Logistic regression is a commonly used model in statistics. In this project, I parallelized a kind of training method, which uses gradient descent, of binary logistic regression model, implemented in MPI, and tested its performance on datasets. Experiments show that the parallel version can give considerable speedup compared to serial version without lose of correctness.

I. Logistic regression and its serial training

Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable. Mathematically, a binary logistic model has a dependent variable with two possible values, such as pass/fail, which is represented by an indicator variable, where the two values are labeled “0” and “1”. In the logistic model, the “log-odds” function for the value labeled “1” is a linear combination of one or more independent variables; the independent variables can each be a binary variable or a continuous variable. The corresponding probability of the value labeled “1” can vary between 0 and 1.[1] In short, after trained with some training data where each sample has several features and a label, it can give the probability that the label of a new sample equals some label, by the formula:

$$p(\mathcal{C}_1 \mid \boldsymbol{\phi}) = y(\boldsymbol{\phi}) = \sigma(\boldsymbol{w}^T \boldsymbol{\phi})$$

, where w is the parameter of the model estimated by training, φ is the vector consisting of values of the features of a sample, and σ is the sigmoid function:

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}.$$

Logistic regression is used in various fields, including machine learning, most medical fields, and social sciences. For example, logistic regression may be used to predict the risk of developing a given disease (e.g. diabetes; coronary heart disease), based on observed characteristics of the patient (age, sex, body mass index, results of various blood tests, etc.). Another example might be to predict whether a Nepalese voter will vote Nepali Congress or Communist Party of Nepal or Any Other Party, based on age, income, sex, race, state of residence, votes in previous elections, etc. It is also used in marketing applications such as prediction of a customer's propensity to purchase a product or halt a subscription, etc. In economics it can be used to predict the likelihood of a person's choosing to be in the labor force, and a business application would be to predict the likelihood of a homeowner defaulting on a mortgage.[1]

In this project, binary logistic regression, which deals with the situation in which the label can have only two possible types, is used; besides, the model is put a simple Gaussian prior

$$w \sim \mathcal{N}(0, \tfrac{1}{\alpha} I)$$

, to avoid overfitting. Then fitting the model is done by maximum a posteriori.[2] Because that there is no closed form solution for fitting the model, usually numerical optimization methods are used in training of logistic regression. This project chooses gradient descent to do the optimization, where in each iteration, the parameter w is updated as follows:

$$w_{n+1} \leftarrow w_n - \eta[\Phi^T(y-t) + \alpha w_n].$$

, where w is parameter of the model, η is the learning rate which decides at each step how far to walk in the direction of the gradient, Φ is the matrix of training data without labels, t is the vector containing all true labels of the training data, and y is calculated by sigmoid of dot product of current parameter w and a feature vector φ, or

$$\sigma(\boldsymbol{w}^T \boldsymbol{\phi})$$

.

The stopping criterion of iteration could be that the w before and after update is similar enough, or a limit of number of iterations could be specified. In this project, the stopping criteria as follows is used:

$$\frac{\|w_{n+1} - w_n\|_2}{\|w_n\|_2} < \varepsilon$$

, or number of iterations reaches 6000.

With such facts, a serial algorithm of training the model can be written:

Init w to zero vector

While(true):

For feature vector φ of every sample in the training set:

y[i]=sigmoid(w · φ)

w=w- η(Φ^T · (y-t) + α*w)*

if $\frac{\|w_{n+1} - w_n\|_2}{\|w_n\|_2} < \varepsilon$ *or n_iter>=max_iter:*

break

return w

II. Parallelization

It is obvious that in the steps calculating y in the serial training process:

For feature vector φ of every sample in the training set:

$$y[i]=sigmoid(w \cdot \varphi)$$

calculating for each element of y only needs w and feature vector of only one sample in the dataset. Thus, this part can be parallelized by dividing the training set such that each part has some samples, then dispatching each part to a process to calculate its corresponding part of vector *y*.

However, follows this is the updating of parameter w, which also contains a reference to y:

$$w=w- \eta*(\Phi^T \cdot (y-t) + \alpha*w)$$

noticing that the part that takes the most computation in this formula is actually the matrix-vector multiplication:

$$\Phi^T \cdot (y-t)$$

, which actually can be seen as a linear combination of feature vectors in the training set Φ. As a result, each process can take a part of the samples, and the part of y and t corresponding to its parts of samples, calculate its own linear combination, then the result of the multiplication can be obtained by summing up results of all processes. It's worth seeing that this division works well with the former division for calculating y. The code of these two step's computations for each process is as follows: (mydata[i] is feature vector of one sample and mydata[i][j] is the jth feature in the vector; *expit* is the sigmoid function)

```
for(int i=0;i<local_rows;i++)
{
        dotvv=0.0;
        for(int j=0;j<cols;j++){
                dotvv+=w[j]*mydata[i][j];
        }
        myy[i]=expit(dotvv);
}
for(int i=0;i<cols;i++)
   for(int j=0;j<local_rows;j++)
      dotmv[i]+=mydata[j][i]*(myy[j]-myt[j]);
```

The project uses MPI for parallelization. After this, the sum of all partial results of the matrix-vector multiplication can be simply obtained by a call to MPI_Reduce, thanks to its built-in MPI_SUM operation:

```
MPI_Reduce(dotmv,sumdotmv,cols,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

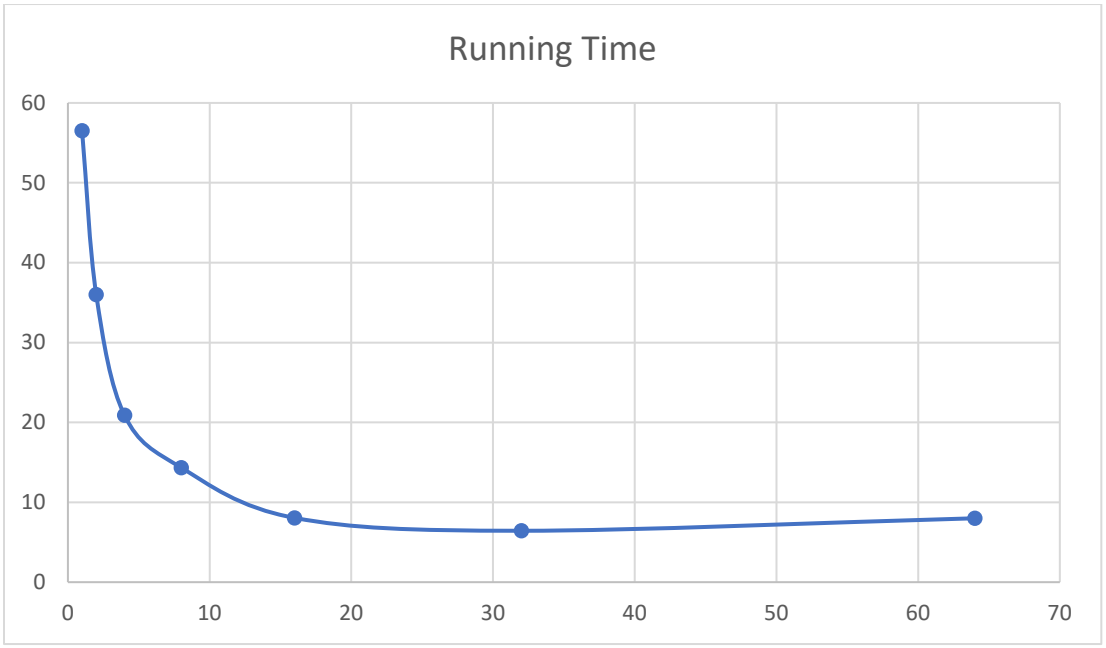
Then the update of w is performed by Process 0, the so-called "parameter server". With the remaining computation being scalar products, it will not take much computation resources.
After updating, the new parameter w is broadcasted to all processes to ensure that each process has the up-to-date w to perform next iteration. At the same time, the stopping criteria is also calculated by Process 0 and if it is satisfied, it will turn the stopping signal to 1 and also broadcasting it.
The corresponding code for Process 0 and broadcasting:

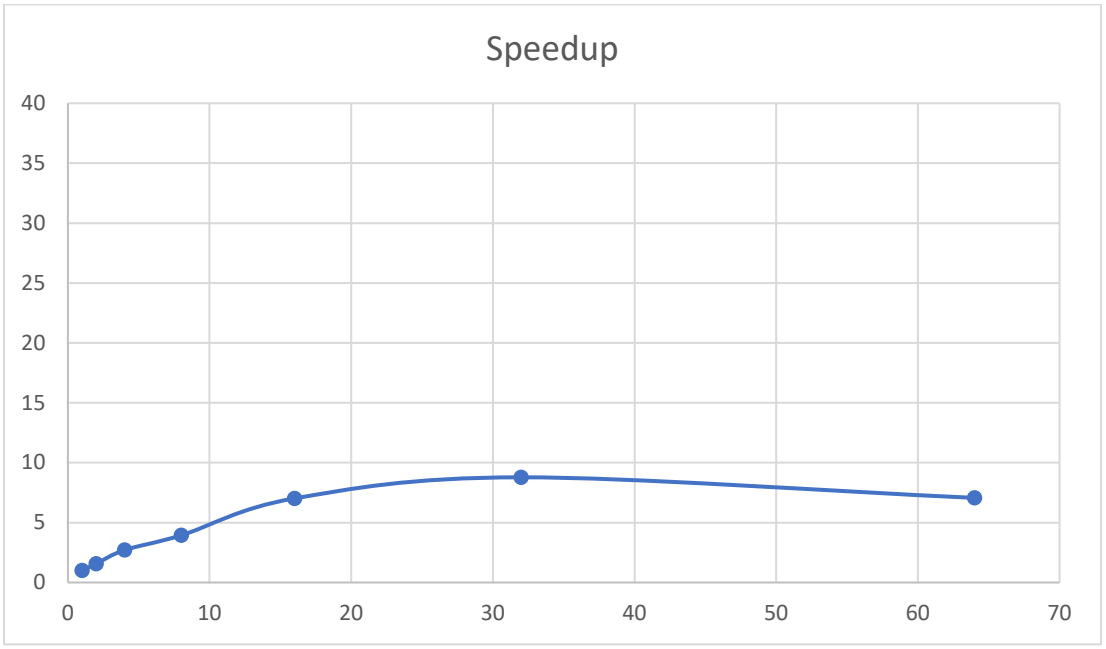
```
if(myrank==0)
{
        for(int i=0;i<cols;i++)
           new_w[i]=w[i]-eta*(sumdotmv[i]+alpha*w[i]);
        double sum1=0.0;
        double sum2=0.0;
        for(int i=0;i<cols;i++)
        {
           sum1+=pow((w[i]-new_w[i]),2);
           sum2+=pow(w[i],2);
        }
        if(sum1/sum2<precision)
             stopping=1;
        for(int i=0;i<cols;i++)
           w[i]=new_w[i];
}
MPI_Bcast(w,cols,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(&stopping,1,MPI_INT,0,MPI_COMM_WORLD);
```

III. Experiments

To test the performance of the parallelized training algorithm, experiments are carried out on Big Red II. The default cc in PrgEnv-Cray environment is used to compile the program, and the program is run on two nodes and 32 cores per node.
The dataset used in the experiments is made from part of the USPS dataset[3] , each vector represents a 16*16 bitmap, and each bitmap could be an image of character '3' or '5'. The task is to train the model to distinguish between the two characters. To make speedup more obvious while not affecting the training process, the data is duplicated to

11 copies, making a 16940*257 matrix. During all experiments, the hyperparameter α is set to 0.1, η set to 0.001, the stopping criteria ε set to 0.000001 (also for the purpose of emphasizing speedup) and max number of iterations is 6000.
The following chart shows the running time of the training program on the mentioned dataset in seconds, and speedup compared to runs with 1, 2, 4, 8, 16, 32, 64 processes (each averaged over 3 runs):





The results show that the parallel version successfully gave a considerable speedup compared with serial version. For the performance degradation, I came up with 3 possible sources:
1. The serial computation on process 0;
2. the communication overhead;
3. the starvation caused by waiting for all process to terminate this iteration.
To test 3, I compared two runs of the parallel program, both on 14 processes, and set the stopping criteria only to be number of iterations; with same number of samples, one run letting all the processes have exactly the same data, the other one with different data per process. Averaged over 6 tests, the results show that the run with all same data terminates ~1sec faster than the latter (~24secs vs. ~25secs).
To test 1 and 2, I compiled the program with *mpicc*, ran 32 processes with *mpirun*, and ran *perf* with them to record overhead; results show that overhead of function corresponding to serial computation of process 0 is very low – in fact it does not even showed up in the table; the highest-ranking call besides the main function is "opal_progress" and "mca_btl_vader_component_progress" which in total takes ~48% of overheads. While I don't know the details of these calls, I guess the time may be spent in waiting for events during communication. An possible evidence of this is that when less processes are used (decrease from 32 to 4), the overhead of these two calls decreases significantly (to ~8 percents). The comparison of performance when run on 32 processes vs. 64 processes is also worth noticing, in which the latter uses two nodes of the supercomputer and that might bring extra communication overhead.
Besides, to make sure that the algorithm is correctly parallelized, the accuracy of prediction of the obtained model is also tested. Since the goal of the project is to improve the computational performance, the model is simply tested against the same training data as it is trained with, just to ensure that the data is fitted correctly. The code for testing the accuracy, meanwhile also for performing prediction using the model, is as follows:

```
int correct=0;
 for(int i=0;i<rows;i++){
      dot=0.0;
      for(int j=0;j<cols;j++)
           dot+=mydata[i][j]*w[j];
      double p=expit(dot);
      if((p>=0.5 && myt[i]==1) || (p<0.5 && myt[i]==0))
           correct++;
}
printf("\n Corrects: %d out of %d\n",correct,rows);
```

The result shows that the model trained in parallel reached an accuracy of 98.05% on the training set, indicating that it has fitted the model correctly; at the same time the serial trained model on same data gives 98.3% accuracy, indicating the parallelization of the algorithm hardly affect the predicting performance of the model. The difference of the accuracy may be caused by error propagation under different order of computation compared with the serial version.

IV. Conclusions and Future Works

A parallel training algorithm for binary logistic regression is implemented in the project and it's proved that it can speed up the training process with little influence on accuracy. As some possible future work,

1. Inside one iteration the computation can be further parallelized using OpenMP, CUDA or other techniques, by dividing the data not only in rows but also in columns; this is especially important when dealing with data with millions of features.
2. Currently the reduce and broadcast operations are handled by MPI, which means that the topology and strategy of communication in the network is determined automatically in MPI. It may be worth doing to test how different MPI implementations, or how different low-level topology and strategies affect performance of communication. This is especially critical when the model is large and can have large amount of parameters.
3. Besides, for sparse data, it can be proved that in Stochastic Gradient Descent, with a different update formula using only one sample per iteration, each process can finish their each iteration asynchronously. Under such condition, the process(es) holding the parameter w acts more like a "server": it should update the parameter or send it to a server whenever receives a request for that operation. As one of the popular architectures in today's parallel machine learning, implementation and optimization of the server is also worth exploring.[4]

References:

[1] Wikipedia. 2019. Logistic Regression - Wikipedia. https://en.wikipedia.org/wiki/Logistic_regression

[2] Christopher Bishop. 2006. Pattern Recognition And Machine Learning. Springer. https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf

[3] USPS handwritten digit data. http://www.gaussianprocess.org/gpml/data/

[4] Mamidala, A. R., Kollias, G., Ward, C., and Artico, F.
MXNET-MPI: embedding MPI parallelism in parameter server task model for scaling deep learning. CoRR, abs/1801.03855, 2018. URL http://arxiv.org/abs/1801.03855.