PARALLEL STOCHASTIC GRADIENT DESCENT USING MPI

Monish Kumar V (CE18B118)
Department of Civil Engineering
Indian Institute of Technology, Madras

ABSTRACT

In the era of Bigdata, parallelizing Machine Learning and Deep Learning algorithms has become an increasingly pressing problem. As a substitute for traditional Gradient Descent algorithm, Stochastic Gradient Descent (SGD) is being used due to its faster convergence rate. In this paper a C++ codebase for parallel implementation of stochastic gradient descent algorithm is developed using MPI library specification. A sample regression problem is solved to validate the code. The main objective of this study is to compare the time taken, speedup, efficiency for solving regression problems between the serial and parallel implementation of the stochastic gradient descent algorithm.

NOMENCLATURE

x	Regressor
y	Target variable
m	Length of data set
n	No. of regressors
θ	Model parameter (weights)
$J(\theta)$	Loss function
α	Learning rate
p	No. of threads (machines)
e	Epochs

INTRODUCTION

Over the past few decades, with tremendous increase in data, we have realized even though a good machine learning algorithm yields good results in general, sometimes the better performant algorithm is for those who has larger data set rather than those who has the better algorithm. The main challenge of using gradient descent to improve the predictive ability of the computer for training on large data set is that it is very computationally expensive due to its slow convergence nature. This is where stochastic gradient descent comes in handy [1]. In Stochastic Gradient Descent (SGD), a few samples are selected randomly instead of the whole data set for each iteration of descent i.e., the batch size is equal to one whereas the batch size for gradient descent is equal the length of the whole data.

However, SGD is inherently a serial algorithm. It updates the parameters after seeing every example. Thus, SGD's scalability is limited by its inherently sequential nature and as a result it is difficult to parallelize. [2] proposed an elegant technique to parallelize SGD. This paper aims to implement the technique proposed in [2] with the help of Message-Passing Interface (MPI), a library specification used for parallel computing on distributed systems and review its performance by measuring the speedups achieved on a sample experimental data.

GOVERNING EQUATIONS

Given a data set $\{y^{(i)}, x_0^{(i)}, x_1^{(i)}, x_2^{(i)}, ...x_{n-1}^{(i)}\}_{i=1}^{m}$, let the model to be predicted take the form,

$$\hat{y}^{(i)}(\theta, x) = \sum_{j=0}^{n-1} \theta_j x_j^{(i)} \tag{1}$$

where,

$$x_0^{(i)} = 1 \ \forall i \in \{1, 2, ...m\}$$

A generic regression problem is solved by minimizing the sum squared errors as shown in Eqn. (3).

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 \tag{2}$$

$$\min_{\theta} \quad J(\theta) \tag{3}$$

The parameter update is computed by taking a step in the opposite direction of the cost gradient for a generic gradient descent algorithm where the magnitude and direction of the update is given by,

$$\begin{aligned}\Delta\theta_j &= -\alpha\frac{\partial}{\partial\theta_j}\left(J(\theta)\right) \\ &= -\alpha\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)x_j^{(i)}\end{aligned} \tag{4}$$

EXPERIMENTAL SCENARIO

To review the experiment, a custom generated dataset of various lengths was used where the regressors $x_j^{(i)}$ are a random numbers in the range $[-10, 10]$.

TABLE 1: SAMPLE MODEL PARAMETERS

θ_0	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7	θ_8	θ_9
1.0	-2.0	8.0	4.0	-9.0	5.0	7.0	2.0	-3.0	6.0

The target variable $y^{(i)}$ corresponding to a training sample $x^{(i)}$ is calculated as shown in Eqn. (1). with sample model parameters θ_j given in Table 1. Further, to add some stochasticity to $y^{(i)}$ a gaussian white noise, $\varepsilon \sim \mathcal{N}(0,1)$ is added to it.

$$y^{(i)}(\theta, x) = \sum_{j=0}^{n-1}\theta_j x_j^{(i)} + \varepsilon \tag{5}$$

ALGORITHM

Using the generated data, a linear regression problem was solved as described in Algorithm 1 [2], using the MPI library specification in C++. As we know, in bigdata analytics, the amount of data to be processed by one machine is usually very high. As a result. for computational efficiency the data is randomly shuffled and scattered across p machines as described in Algorithm 1.

Algorithm 1 ParallelSGD (Learning Rate α, Machines p, Epochs e)

Define $T = [m/p]$
Randomly partition the data, giving T examples to each machine.
for all $k \in \{1, 2, ...p\}$ **parallel do**
  Randomly shuffle the data on machine k.
  Initialize $\theta_{j,k} = 0$
  for all $i \in \{1, 2, ...e\}$ **do**
    Randomly draw a sample (x^*, y^*) with replacement
    $\theta_{j,k}^{i} \leftarrow \theta_{j,k}^{i-1} - \alpha(\hat{y}(\theta_{j,k}^{i-1}, x^*) - y^*)x_j^*$
  end for
  Aggregate from all computers, $\theta_j = \frac{1}{k}\sum_{k=1}^{p}\theta_{j,k}^{e}$
end for
return θ_j

The regression problem was solved by making use of the High Performance Computing Environment (HPCE), a super-computing facility available at IIT Madras.

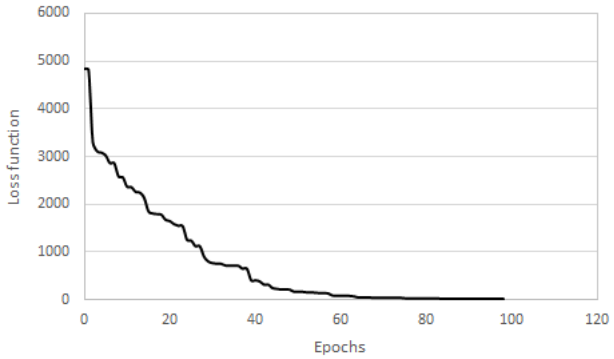

FIGURE 1: Loss function $J(\theta)$ at the end of every epoch (e) for $m = 10^3$

RESULTS AND DISCUSSION

Investigation was carried out using two different data sets each of length $m = 10^3$ and $m = 10^4$ respectively to check if the model has a good accuracy. The parameters estimated by running a serial program i.e., $p = 1$ are shown

TABLE 2: ESTIMATED MODEL PARAMETERS

	θ_0	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7	θ_8	θ_9
$m = 10^3$	0.92	-1.99	8.0	4.02	-8.97	5.0	6.99	2.0	-2.99	6.0
$m = 10^4$	0.99	-2.03	8.02	3.97	-8.98	5.03	7.01	2.0	-2.97	5.97

in Table 2. It is evident that the estimated parameters are very close to the true values. As a result, one can conclude that the model is performing reasonably well.

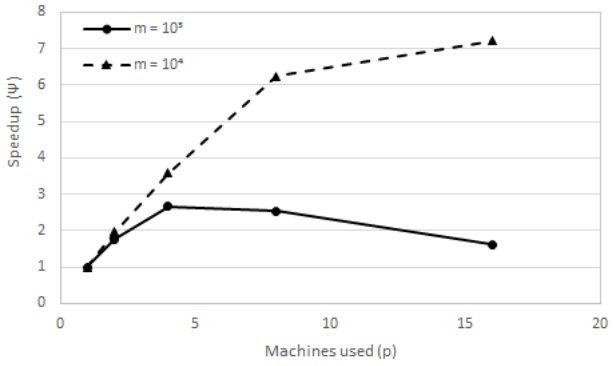

FIGURE 2: Speedup (ψ) as a function of length of data set m and no. of machines p

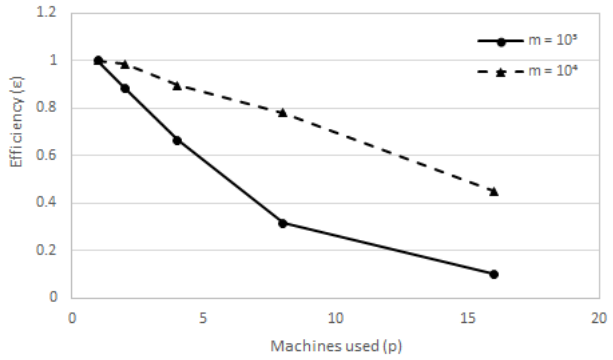

FIGURE 3: Efficiency (ϵ) as a function of length of data set m and no. of machines p

On analysing the loss function (Fig. 1.) it can be seen that around 100 epochs is enough to ensure convergence of the model. Since, 100 epochs consume very less time it is difficult and pointless to review the parallelism performance because time saved is very less in comparison to the overhead cost of parallelism. As a result, the model was allowed to run for a large no. of epochs (5000) which generally is the case of bigdata analytics.

The model's ability to parallelize is evaluated by comparing the speedup and efficiency for different values of m and p. The speedup (Fig. 2.) increases with increase in no. of threads (machines) for $m = 10^4$ as expected whereas for $m = 10^3$ the speedup reduces significantly for $p > 4$. This reduction in the latter case is because the parallel overheads are higher than the time saved through parallelism. This also explains why efficiency (Fig. 3.) is significantly higher for the former while the efficiency is nearly equal to zero for later for large p's. Therefore, trying to parallelize programs which consume very less time when run in serial manner is not recommended.

CONCLUSION

A C++ code using MPI specification is developed to solve a regression problem using parallel stochastic gradient descent. Model parameters were estimated with two different data sets of lengths 10^3 and 10^4 and compared with true values. Speedup and efficiency of the parallel code for different values of p where investigated.

REFERENCES

[1] Bottou, Léon. "Large-scale machine learning with stochastic gradient descent." Proceedings of COMPSTAT'2010. Physica-Verlag HD, 2010. 177-186.

[2] Zinkevich, Martin, et al. "Parallelized Stochastic Gradient Descent." NIPS. Vol. 4. No. 1. 2010.