Parallel Stochastic Gradient Descent

Biplab Kumar Pradhan, Siddhant Panda
Department of Computational and Data Sciences
Indian Institute of Science, Bangalore, India
bippradhan@grads.cds.iisc.ac.in, siddhantpanda22@gmail.com

***Abstract*—With increasingly large datasets, stochastic gradient descent is becoming more convenient over gradient descent, as performing gradient descent for learning over entire dataset at each time step becomes very expensive. In this report we survey several SGD parallelization algorithms, and present our own implementation of a combined algorithm.**

I. Introduction

Gradient Descent is a method used for finding local minima of a function iteratively. One of its most significant use is in training machine learning models, like neural networks, by applying gradient descent on a convex cost function.

A machine learning model is defined by a set of parameters that are fine tuned based on an input dataset, so that it gives accurate "predictions" for data not in the input set.

Given a machine learning model defined by a set of parameters, gradient descent starts with an initial set of parameter values and iteratively moves toward a set of parameter values that minimize the cost function.

$$w_{k+1} = w_k - \eta \Sigma \nabla C \tag{1}$$

$$C = \frac{1}{2N} \sum_{i=1}^{N} C_i \tag{2}$$

- A commonly used cost function for training a machine learning model, with an input data size N is

$$C(W, B) = \frac{1}{2N} \sum_{i=1}^{N} ||Y_{actual}(x_i) - Y_{predicted}(x_i)||^2 \tag{3}$$

- It is easy to parallelize. As all the terms of the sum are independent of each other, ideally, we can assign N processing units to compute each term of the series independently.

Stochastic Gradient Descent: Here the gradient of the function in each iteration is approximated by gradient of a single example x_i and label Y_i, where we choose x_i randomly from the given batch of samples for each iteration, then update the parameter before the next iteration. The total number of iterations required are more than that for gradient descent, but as one sample is chosen in each iteration, hence total gradient computations is significantly less, for large datasets.

$$w_{k+1} = w_k - \eta \nabla C_i \tag{4}$$

The cost function simply becomes:

$$C_i(W_i, B_i) = \frac{1}{2} ||Y_{actual}(x_i) - Y_{predicted}(x_i)||^2 \tag{5}$$

for some random $i \in 1 : N$.

- Useful and faster when datasize is very large. Is commonly used nowadays in machine learning.

II. Related Work

In [1], the authors discuss the challenges faced while working on industrial scale datasets on a given bandwidth and capacity of hard disk. Computing SGD would take hours for large datasets on a single hard disk.

Here the use of Distributed memory architecture is optimized by assigning datasets on different hard disks to different processors. They perform independent SGD on their respective data based on a fixed learning rate η for a fixed number of steps T and a cost function as suitable. Then averaging the computed gradients by each processor at the end. The averaging reduces the variances by $O(k^{\frac{-1}{2}})$ where k is the number of processors. Here it has been proved that for large sized datasets the accuracy is good and solution converges by this method.

In [2] delayed stochastic gradient is used. Here the observations are partitioned among 'N' processing units on a per-instance basis. Each of them computes its own gradient. After computing the gradient, each processor would update the shared parameter vector in a round robin manner. Thus causing a delay of 'N-1' between computing the gradient, and applying to the shared parameter vector.

For optimization, two templates are suggested: Asynchronous stochastic gradient descent and pipelined optimization. In the paper pipelined optimization was also implemented which gives a delay τ, which is much smaller than 'N-1' and mainly depends on latency of the communication channel.

III. Methodology

Our implementation of asynchronous stochastic gradient descent using CUDA.

Algorithm 1 Asynchronous SGD(CUDA)

NUMSTEPS ← total number of steps
MAXITER ← *NUMSTEPS/(NumThreads)*
All threads share the parameter vector W
for i: 0 to MAXITER:
Get random x_i from input set
compute $grad(x_i)$
$atomicAdd(W - grad(x_i))$.
$endfor$.

Our implementation of distributed memory stochastic gradient descent using MPI.

Algorithm 2 Distributed SGD(MPI)

NUMSTEPS ← total number of steps
MAXITER ← *NUMSTEPS/(NumProcs)*
Each processor has its own parameter vector W_j, $j \in$ [1 to NumProcs]
Divide input set into NumProcs subsets.
for i: 0 to MAXITER:
Get random x_i from input set j.
compute $grad(x_i)$
$W_j = W_j - grad(x_i)$.
$endfor$.
$W \leftarrow \frac{1}{NumProcs}\Sigma W_j$. (Sum using MPI_Reduce)

Here we propose a combination of algorithms proposed in [1] and [2]. Each block has whole dataset and independently performs asynchronous SGD, where each thread in a processor computes and updates parameter vector with a delay of τ. At the end parameter is averaged over all processors.

Algorithm 3 Combined Algorithm(CUDA)

NUMSTEPS ← total number of steps
MAXITER ← *NUMSTEPS/(NumBlocks*NumThreads)*
Parameter vectors W_j for $j \in$ [1 to NumBlocks]
All threads in block j share the parameter vector W_j
for i: 0 to MAXITER:
Get random x_i from input set
compute $grad(x_i)$
$atomicAdd(W_j - grad(x_i))$.
$endfor$.
$W \leftarrow \frac{1}{NumBlocks}\Sigma W_j$.

IV. Experiments and Results

A. Experiment Setup

The aforementioned algorithms were tested on OpenMP, MPI and CUDA.

Dataset:Pima Indians Diabetes Data Set (UCI Machine Learning Repository) - Binary classification problem(0,1).

Logistic regression model was used.

$$Y_i = h(z_i) = \frac{1}{1 + e^{-z_i}}$$

where $z_i = W^T X_i + B_i$

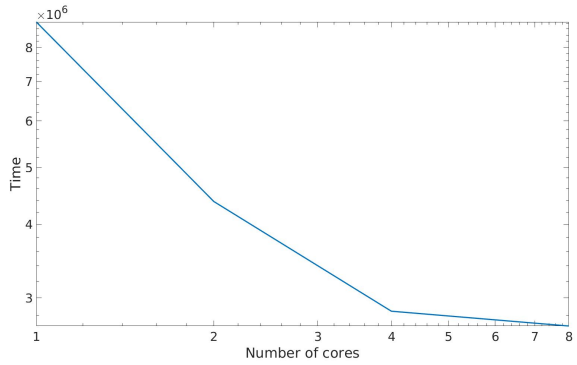
Independent Variables taken - Diastolic blood pressure (mm Hg), Body mass index (weight in kg/(height in m)^2).

B. Results

OpenMP: The asynchronous SGD was implemented and tested using multi-threading in OpenMP.
For 10 threads, time taken was 1.989 seconds
Sequential time taken was: 0.890 seconds.

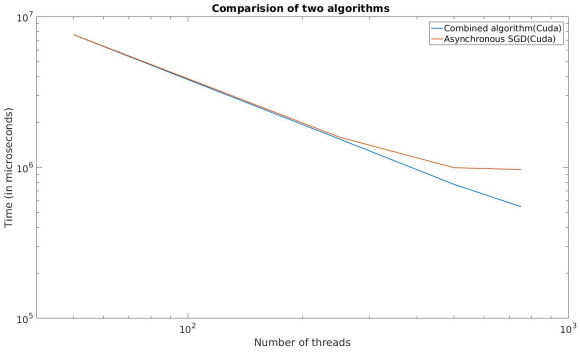
MPI: Asynchronous SGD implementation on MPI framework was done and tested upto 8 cores.It showed scalable speedup.

Fig. 1. MPI implementation of Distributed SGD



CUDA: The CUDA implementation of asynchronous SGD and the 'Combined Algorithm' was compared for a total number of threads: [50,250,500,750]. The Combined algorithm scaled linearly and performed slightly better in comparison to Asynchronous one.

Fig. 2. Combined algorithm vs Asynchronous SGD in CUDA



Error: Root mean square error was calculated. Since we used logistic model for (0,1) classification, error was not very significant provided convergence is achieved.

Fig. 3. MPI distributed algorithm error

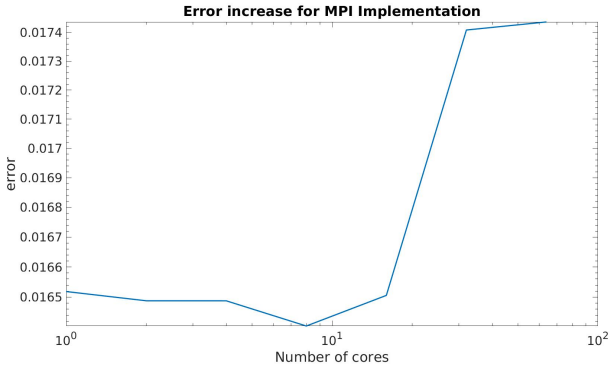
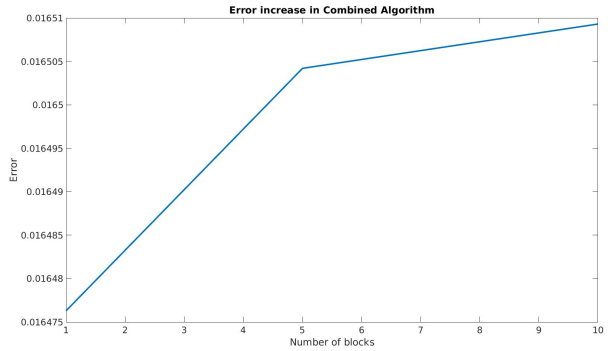


Fig. 4. Combined Algorithm error



V. Conclusions

Shared memory architecture provides good platform for our modeled algorithm. However it is limited by size of data. For very large dataset, algorithm proposed in [1] is better. As it showed convergence as well as the communication step would be minimal, i.e. at the end of individual computation.

The error for MPI algorithm decreased initially, with increasing number of cores, which goes against intuition and has scope for investigation. Larger datasets and more complex machine learning models (neural networks for example) provide better avenue for experiments with parallelization in future.

Our combined algorithm performed better than the asynchronous one, thus there is scope for further investigation in this direction.

References

[1] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[2] J. Langford, A. J. Smola, and M. Zinkevich, "Slow learners are fast," *Advances in Neural Information Processing Systems*, vol. 22, pp. 2331–2339, 2009.