Parallelized Stochastic Gradient Descent

Wen-Ching Miao
Institute of Network Engineering
National Chiao Tung University
Hsinchu, Taiwan
mm0503885@gmail.com

JIA-MING CHANG
Institute of Network Engineering
National Chiao Tung University
Hsinchu, Taiwan
vincent191228@gmail.com

Pei-Chi Mai
Institute of Network Engineering
National Chiao Tung University
Hsinchu, Taiwan
a103291@yahoo.com.tw

ABSTRACT

Stochastic gradient descent (SGD) is the most popular optimized method in machine learning, most importantly forms the basis of Neural Networks. To handle large-scale problems, researchers have proposed several methods for multicore systems. However, due to its iterative nature, it is hard to achieve satisfactory performance. In this project, we perform data-parallel SGD and update shared parameters by both synchronous and asynchronous methods. Results show that our asynchronous can made improvements like other related works in terms of convergence rate and computation time.

1 INTRODUCTION

Gradient descent (GD) and stochastic gradient descent (SGD) are optimization methods to train analytics models like the backpropagation algorithm for deep neural networks. To summarize, these algorithms are used to find the solution by updating its parameter.

$$x^{t+1} = x^t - \gamma * Gradient(x^t) \tag{1}$$

t is the number of times that we update the parameter, γ is the learning rate, and $Gradient$ is a function that we calculate the gradient of the dataset. The difference between GD and SGD is that SGD approximate the gradient using only one or some data points rather than the overall dataset. Thus, evaluating gradient saves a lot of time especially when training data sets are huge. By uniformly choosing a subset of training data as a mini-batch at each iteration, we can also find the optimal values of the parameters and reach the minimum of the loss function. Nowadays, this method is implemented in a form or another by every modern analytics system, such as Google's Brain, Microsoft's Project Adam and Vowpal Wabbit, IBM's SystemML, Pivotal's MADlib and Spark MLlib[3]. Since SGD is widely applied by numerous enterprises, it is important to understand its optimal behavior and limitations on modern computing architectures.

At the same time, if we want to get higher accuracy when training models in deep learning, it's inevitable to increase the amount of training data. By now, some datasets are even approaching PetaBytes or more. The size and complexity of the model combined with the size of the training dataset makes the training process very computationally and temporally expensive. As a result, many researchers have tried to use distributed and parallel systems to solve the problem. Despite of its limited scalability by its inherently sequential nature, there are many works try to break the parallelization barrier and develop several clever data-parallel schemes for SGD recently. In general, data parallelism is to partition and distribute training data evenly across several nodes of a system. All nodes or workers will concurrently train on their partition of the data and aggregate their results in some way to have the model produce a model that was trained on the entire dataset. By increasing the number of nodes or workers, it effectively reduces the computational cost due to the reduced workload each node needs to process. However, the communication costs to share and synchronize huge gradient vectors and parameters also increase dramatically and may thwart the anticipated benefits of reducing computational costs. In this project, we will first perform the comprehensive study of data-parallel SGD. Then we will propose our own methods and discuss the results. The experimental results show that our implementation can overcome the weakness caused by communication overhead and achieve speedup while attaining desired accuracy, too.

2 RELATED WORKS

In data-parallel SGD, a large dataset is partitioned into several subsets. These subsets are processed together to minimize an objective function. Each process has its own local parameter vector of the model. At each iteration, each process computes an updated stochastic gradient using its own local data. Then it shares the gradient update with its peers. The processes collect and aggregate stochastic gradients to compute the updated shared parameter vector. Since the communication time required to share stochastic gradients and parameters is the main performance bottleneck, there are many different data parallel approaches, mainly differing by how the workers' contributions are aggregated. They can be categorized into two main groups: synchronous and asynchronous data parallelism.

2.0.1 *Synchronous parallel SGD.*

The main difficulty to parallelize SGD is that the chain dependency on model updates across epochs, where the current gradient relies on the previous model. Thus, in order to preserve the concurrency of gradients within each iteration, we can utilize a series of locking mechanisms during the training runtime. In [1], it proposes a global Top-k (gTop-k) sparsification approach for synchronous SGD (S-SGD). Top-k sparsification techniques can be used to reduce the volume of data to be exchanged among workers and thus alleviate the network pressure. It zero-outs a significant portion of gradients without impacting the model convergence. However, this paper identified that the Top-k sparsification is inefficient in averaging the gradients from all workers because the indices of the Top-k gradients are not the same. By the algorithm it proposed, its results show that their communication complexity can be reduced to $O(klogP)$. [5] proposes another method to minimize the training iteration time. The researchers develop an optimal solution named merged-gradient WFBP (MG-WFBP) and implement it in their deep learning platform B-Caffe. Based on the fact that merging some short communication tasks into a single one may reduce the overall communication time, they formulate an optimization problem and propose two algorithms to solve it. Their experimental results show that the MG-WFBP algorithm can achieve good scaling efficiency.

2.0.2 *Asynchronous parallel SGD.*

Usually, when there are stragglers (i.e., slower workers) in the parallel SGD system, which is common especially at the scale of hundreds of devices, asynchronous approaches are more robust. Multiple model updates are executed without mutexes or locks. Moreover, access to the model in gradient computation can also interface with the update. Hogwild [4] is the most representative algorithm in this category which allows processors access to shared memory with the possibility of overwriting each other's work. Although it seems to cause many problems, the researchers prove this method to work without any negative effect and provide all the benefits of having multiple threads. The results show that when the associated optimization problem is sparse, Hogwild achieves a nearly optimal rate of convergence. Besides, since it only executes the following loop in parallel.

for $i = 1$ to N **do** in parallel

This makes the parallel implementation of Hogwild (such as add directive in OpenMP) very simple, too. Another paper [2] describes that while asynchronous algorithms using a parameter server, it may suffer from two weaknesses: communication bottleneck when workers are many and significantly worse convergence when the traffic to parameter server is congested. To solve these problems, they propose an asynchronous decentralized stochastic gradient descent algorithm (AD-PSGD) that keeps the advantage of both asynchronous SGD and decentralized SGD. In AD-PSGD, each worker maintains a local model x in its local memory and compute and update gradients without any global synchronous. This method reduces the idle time of each worker and the training process will still be fast even if part of the network or workers slow down. After that, all workers run the averaging procedure simultaneously with a virtual counter k to denote the iteration counter. Also, researchers design the communication network as a bipartite graph to prevent the deadlock. Their design is theoretically justified to have the same convergence rate as synchronous and centralized models and can achieve linear speedup with respect to number of workers.

3 PROPOSED SOLUTION

3.0.1 *Synchronous parallel SGD.*

Our first solution is to parallelize SGD synchronously. We divide data evenly to all the processes we have and let each process use its own data to calculate the gradient in every iteration. Within each iteration, processes will wait until all of them have finished their calculations. Then, the main process will collect gradients that each process computed and take an average. After the main process uses the averaged gradient to update its parameter, the last step is to broadcast the new parameter to all other processes before next iteration starts. Figure 2 below illustrates these steps.

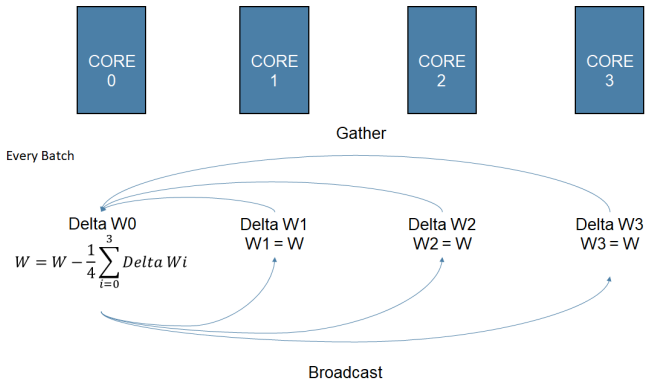

Figure 1: Synchronous parallel SGD

Since each of the iteration is blocking, i.e., the next iteration is invoked only after all processes finish execution and get the updated parameter, this introduces a clear boundary between gradient computation and model update. The advantage of synchronous SGD is that synchronization will be strictly enforced between iterations so that every process will have the same information at each iteration. However, the process to synchronize entails more work significantly,

which makes the parallelization takes more computation time than the serial one. As a consequence, we design an alternative method to calculate SGD.

3.0.2 parallel SGD with timeline.

Since each process may calculate gradients with different speed according to its computation power, processes that take less time for calculation need to waste time waiting for others. Thus, to solve this problem, we consider different conditions for processes who wants to update the shared parameter to take corresponding actions. At each time a process finishes its calculation and wants to update the shared parameter, six conditions it may occur as Table 1 shows.

Table 1: Parallel SGD with timeline

shared parameter	shared parameter is	action toward shared parameter
Unupdated	being read	None
Unupdated	being written	None
Unupdated	idle	update and read
Updated	being read	read
Updated	being written	update and read
Updated	idle	update and read

If no other processes have updated the shared parameter since the process last updated, it means that the process can calculate faster than others. Thus, if the shared parameter is idle then update it; otherwise, do nothing. On the other hand, if the shared parameter has been updated by others, the process needs to check the current state of the shared parameter and acts correspondingly. If some process is reading then read it as its local parameter, too. If some process is writing, then update the shared parameter, too. If the shared parameter is idle now, the process can also update it. We consider all possible situations and try to apply proper actions to reduce waiting time. Nevertheless, to accomplish this algorithm is not easy, and we are not sure whether in this way can save time while ensure the synchronicity and correctness, so we design the third algorithm.

3.0.3 Asynchronous parallel SGD.

We assume that all the cores have similar computation power. Thus, the time for each process to calculate the gradient is almost the same. Based on this prerequisite, we can use a for loop for each of them to update the shared parameter in turn rather than collect all of them at every iteration. We use four cores as an example to illustrate our idea in Figure 2. Firstly, all the training data will be distributed equally to Core 1 to Core 3. The only function of Core 0 is to receive the gradient from other processes and update the shared parameter. Core1, Core 2 and Core 3 will take turns to communicate with Core 0 at different iterations. To avoid waiting at Core0, it must be the fastest. For this purpose, we do not assign data for Core 0 to reduce its workload. Secondly, to make sure Core 1, Core 2, Core 3 will not communicate with Core 0 at the same time, they must have similar computation power and same amount of data. By this way, we can reduce the communication overhead dramatically. Although processes cannot always get the updated gradient at every iteration so that we may get lower accuracy, it is still within acceptable boundary.

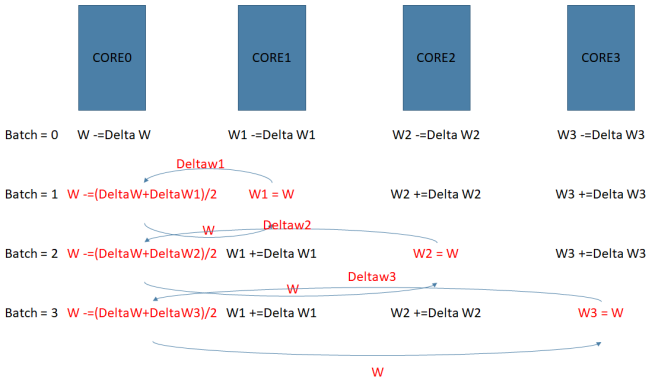

Figure 2: Asynchronous parallel SGD

4 EVALUATION

4.0.1 Experimental methodology. To validate the proposed algorithms compared to serial one, we conduct experiments in the following section. Figure 3 illustrates our architecture. We use MNIST as the evaluation dataset and construct a neural network architecture with one hidden layer as our deep learning framework. MNIST dataset, a popular dataset used for machines to recognize handwritten digits, consists of 60000 training samples and 10000 testing samples. Methods that used in the experiments are mounted in the same physical machine with Intel i7-8700 3.20GHz CPU and 16GB system memory. All the code is written in Python and we use mpi4py for multi-process programming on the CPU. To measure under the same circumstance, we set hyper-parameters that the learning rate is set to be 1 initially and reduced by 1% after each epoch. Also, we fix the batch size to 100 for the dataset and apply sigmoid as the activation function. When training models with different methods, we will use just 5 epochs to see if they converge or not.

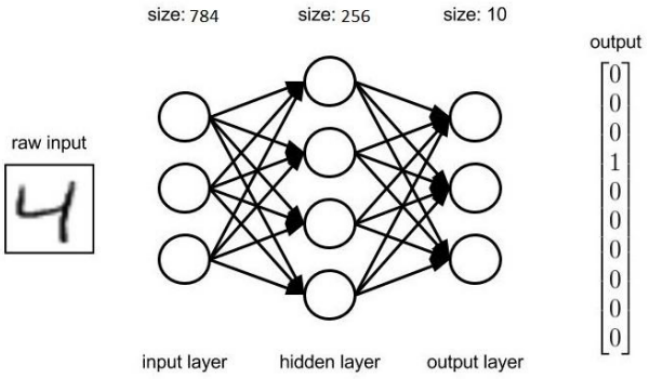

Figure 3: Our experimental architecture

For synchronous and asynchronous SGD, process 0 will initialize two feature vectors at first and broadcast to other processes. Then each process will use the assigned samples to train the model just like the serial one. At the end of each epoch, synchronous SGD will gather all gradients to get their average. After update the shared parameters, these parameters will be broadcast and processes will start another epoch until the end of the for loop. On the other hand, in asynchronous SGD method, processes will take turn sending its gradient and receive the updated parameters. For other processes that are not their turns, they will just use their own parameters to do calculation.

4.0.2 Experimental results.

Table 2 shows the experimental results on both synchronous SGD and asynchronous SGD. We evaluate different number of processes 5 times to take the average and compare their training loss, accuracy and training time with serial SGD.

Table 2: synchronous SGD and asynchronous SGD performance with different number of processes

Method	# Process	Loss	Accuracy	Time (s)
Serial SGD	1	0.3726	0.8880	9.0821
Sync	2	0.2709	0.9210	10.8651
	3	0.2744	0.9213	9.5284
	4	0.2977	0.9185	9.7092
Async	2	0.3204	0.9111	7.5638
	3	0.6277	0.8426	6.4041
	4	0.4981	0.8596	5.9964

As we can see in the table, synchronous SGD can get less training loss and higher accuracy than the serial one. Because we only update the parameter for 5 times, the parallel implements will overall take more data into account and thus get higher accuracy. However, no matter how many processes we use, synchronous method performs a litter worse than the serial program in execution time. We think the reason lies in the high communication overhead. The size of the parameters that we need to synchronize in each iteration are too large (both $w0$ and $w1$ are big matrices). Thus, we think of the second method. Asynchronous SGD can reduce the data size needs to be transmitted and the times that each process needs to communicate with process 0. As a result, we can get speedup compared with the serial one. Figure 4 shows the speedup of two methods. As the number of processes increases, speedup also increase with method 2. But speedup cannot get linear growth along with the number of process. Acceleration effect is not obvious when we increase the number of process because we need to synchronize parameters between processes. That is, communication overhead will be higher. In the aspect of accuracy, we should not just compare each method in specific epoch because the convergence rate is different in each method. Figure 5 shows the convergence of loss (lower loss means higher accuracy) of each method. We can find that the serial method and the synchronous method can get lower than 0.3 loss within a litter epoch, this is the reason why they perform better in accuracy than the asynchronous method in 5 epochs. However, their convergence rate are lower than the asynchronous method. Asynchronous SGD gets the lowest loss in about 10 epochs. On the other hand, the lowest loss in asynchronous SGD is a little higher than the serial method and the synchronous method due to delay gradients, this is a problem that we need to overcome in the future.

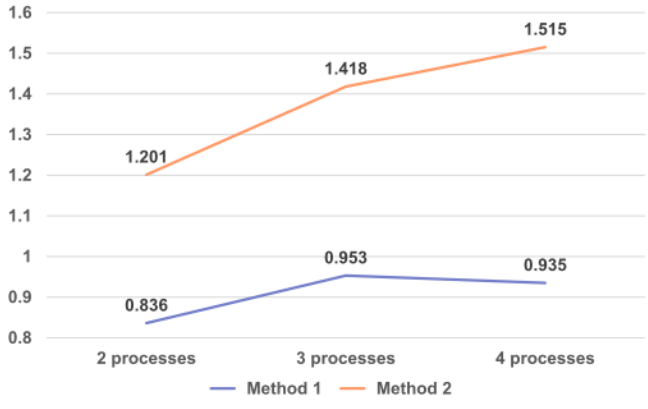

Figure 4: The speedup of our two methods

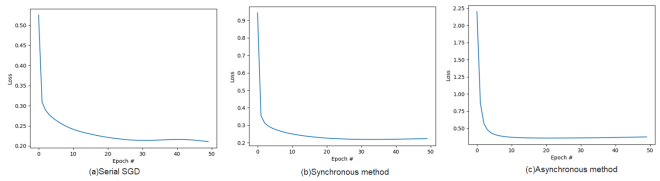

Figure 5: The convergence rate of serial, synchronous and asynchronous SGD

5 CONCLUSIONS

In this project, we perform a comprehensive study of synchronous and asynchronous parallel SGD and design our methods as well. For synchronous SGD, the performance is a little worse than serial one. To improve this method, we may need to drop the slow workers, use backup workers or larger batch-size to avoid wasting time in the future. Also, we can also apply methods mentioned in relate works such as using linear algebra instead of for loop. For asynchronous SGD, we can see that our implementation outperforms than the serial one. However, accuracy dropped a bit due to delay gradients. To solve this problem, a lot of related works have proposed methods that we should apply in the future. One paper leverages Taylor expansion of the gradient function [7], and another paper uses a decentralized token-based protocol to ensure convergence and minimize conflicts [6]. Since SGD is a popular technique for solving large-scale machine learning problems, we propose our methods to parallelize it. And how to enhance the model's performance is our future work.

REFERENCES

[1] Quoc Le, Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, and Andrew Ng. 2011. On Optimization Methods for Deep Learning. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* 2011, 265–272.

[2] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2017. Asynchronous Decentralized Parallel Stochastic Gradient Descent. (10 2017).

[3] Yujing Ma, Florin Rusu, and Martin Torres. 2018. Stochastic Gradient Descent on Highly-Parallel Architectures. *CoRR* abs/1802.08800 (2018). arXiv:1802.08800 http://arxiv.org/abs/1802.08800

[4] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 693–701. http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf

[5] Shaohuai Shi and Xiaowen Chu. 2018. MG-WFBP: Efficient Data Communication for Distributed Synchronous SGD Algorithms. *CoRR* abs/1811.11141 (2018). arXiv:1811.11141 http://arxiv.org/abs/1811.11141

[6] H. Zhang, C. Hsieh, and V. Akella. 2016. HogWild++: A New Mechanism for Decentralized Asynchronous Stochastic Gradient Descent. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 629–638. https://doi.org/10.1109/ICDM.2016.0074

[7] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhiming Ma, and Tie-Yan Liu. 2016. Asynchronous Stochastic Gradient Descent with Delay Compensation for Distributed Deep Learning. *CoRR* abs/1609.08326 (2016). arXiv:1609.08326 http://arxiv.org/abs/1609.08326