2020 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC) and Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)

2020 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC) and Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S) | 978-1-6654-2291-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/MLHPCAI4S51975.2020.00008

EventGraD: Event-Triggered Communication in Parallel Stochastic Gradient Descent

Soumyadip Ghosh and Vijay Gupta

Department of Electrical Engineering, University of Notre Dame
{sghosh2,vgupta2}@nd.edu

Abstract— **Communication in parallel systems consumes significant amount of time and energy which often turns out to be a bottleneck in distributed machine learning. In this paper, we present EventGraD - an algorithm with event-triggered communication in parallel stochastic gradient descent. The main idea of this algorithm is to modify the requirement of communication at every epoch to communicating only in certain epochs when necessary. In particular, the parameters are communicated only in the event when the change in their values exceed a threshold. The threshold for a parameter is chosen adaptively based on the rate of change of the parameter. The adaptive threshold ensures that the algorithm can be applied to different models on different datasets without any change. We focus on data-parallel training of a popular convolutional neural network used for training the MNIST dataset and show that EventGraD can reduce the communication load by up to 70% while retaining the same level of accuracy.**

I. Introduction

Machine Learning workloads are increasing in size due to the massive growth in data collected from the internet, multiple connected sensors, etc. Hence training machine learning models in a parallel environment is becoming ever popular [1], [2]. Most parallel machine learning endeavours have focused on training in multi-core desktop environments or small commodity clusters. However, recently researchers have become interested in machine learning on large-scale clusters such as supercomputers [3]–[7].

One of the biggest challenges in such large-scale parallel environments is that communication is often more expensive in terms of time and energy than computation [8], [9]. Consequently there has been a lot of research aimed at reducing communication in parallel simulations. Most of the focus has been in the context of parallel numerical solutions of partial differential equations [10]–[12] but there has been recent work with respect to parallel training of machine learning models, mostly neural networks [13], [14]. In this paper, we introduce a novel algorithm for communication reduction in parallel machine learning.

Parallel or distributed machine learning can be data-parallel or model-parallel [15], [16]. In the former, the dataset is divided into multiple processors, with each processor having a copy of the entire model and averaging its model with others during the training. In the latter, the neural network model itself is divided among the multiple PEs, however the dataset is not divided. Note that the model decomposition in model parallelism is usually done across the neurons in each layer (width), not across layers (depth) because of the obvious sequential dependencies. There have been approaches to combine the two for hybrid parallelism.

In any kind of parallelism in training, the processors need to exchange the weights and biases with each other before moving to the next training epoch. For example, in data parallelism, the weights and biases among the different processors are averaged with each other before executing the next training epoch. Such an exchange usually happens by message passing at the end of every epoch. In this paper, we propose an algorithm where the exchange of these weights and biases happens in events only when a certain criterion is satisfied. In particular, the main contributions of this paper are:

- We propose a parallel stochastic gradient descent (SGD) training algorithm where the weights/biases are communicated to other processors only when the change in their norm exceeds a threshold; otherwise the intended receiver continues averaging using the last received values.
- We emphasize on a threshold that is adaptive to the slope of the parameter values, thus requiring no separate procedure for tuning the threshold for different models.
- We provide an HPC implementation of our algorithm using PyTorch and Message Passing Interface (MPI) in C++ and highlight the implementation challenges of this algorithm, particularly the need of advanced features such as MPI one-sided communication.

Using this algorithm, we show that the same accuracy can be achieved with 70% lesser messages sent between processors using a popular convolutional neural network on the MNIST dataset. This translates to savings in time and energy involved in communication, thus making the overall training more efficient. Our implementation is open-source and available at [17]. To the best of our knowledge, this algorithm is novel and has not been proposed before in literature in the context of parallel machine learning. Note that we proposed a similar algorithm in the context of parallel numerical simulations of partial differential equations in our previous work [12].

It is important to note that while we focus on data-parallel stochastic gradient descent in this paper, the idea of event-triggered communication can be generalized to model-parallel and hybrid configurations. Further it can be extended to training algorithms other than SGD such as

Adam, RMSProp, etc.

The paper is organised as follows:- Section II surveys related work and Section III introduces the necessary background. The idea of our algorithm is proposed in Section IV with implementation details in Section V. Section VI contains the experimental results followed by conclusion in Section VII. For notational convenience, we denote the abbreviation PE to be a processing element, either CPU or GPU.

II. Related Work

There are a number of approaches to reducing communication in parallel stochastic gradient descent in neural networks. Before we discuss some of the popular works in literature regarding this, we outline the main approaches for parallelizing SGD in Fig 1.

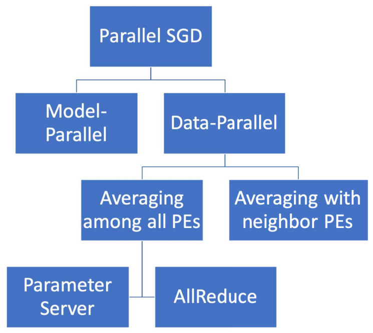

Fig. 1. Popular ways of parallelizing neural networks.

Since we are focused on data parallelism, we study related work in that context. In data-parallelism, every PE has a subset of the overall dataset and a copy of the neural network model. During the training process, the PEs exchange information among each other, namely the weights and biases, to ensure that the training is carried out on the entire dataset and not just the sub-datasets assigned to every PE.

During each epoch of the training process, the common approach is to average the gradients of the weights and biases among all the PEs. There are two approaches which build up on this idea. In the parameter server approach, there is a central parameter server PE that stores the neural network model. All the remaining PEs calculate the gradient in their sub-dataset and then send it to the parameter server. The parameter server averages all the gradients, updates the model and then the worker PEs continue to the next epoch of training, resulting in a synchronized algorithm. This requirement of synchronization was relaxed by the Hogwild algorithm [18] where the worker PEs can send gradients to the parameter server without any lock step. There are numerous other works improving the performance of parameter server based approaches [19], [20]. However, this approach is not very suitable for machine learning on HPC environments because of the central point of dependence on the parameter server, resulting in poor scalability.

The second approach does not involve a parameter server. Instead all PEs average their gradients using an AllReduce mechanism. Every PE has a copy of the same model which is updated using the averaged gradients in every iteration. At the end of training, the models in all PEs are averaged to do testing/inference. Instead of a normal AllReduce, there have been several optimized variants proposed in literature. The authors in [21] have proposed one-bit quantization where each gradient update is quantized to 1-bit, resulting in reduction of data volume to be communicated. Threshold quantization was developed by [22] where only gradient updates greater than a static threshold are encoded with a fixed value and sent. A hybrid approach combining both 1-bit and threshold quantization was given in the adaptive quantization proposed by [13]. Deep Gradient Compression in [23] compresses the size of gradients and accumulates the quantization error and momentum to maintain accuracy. There have been approaches to minimize communication by reducing the precision of gradients, e.g., using half precision (16-bit) for training [24] and mixed precision [25]. Sparsified methods that communicate only the top-k most significant values have been proposed by [14], [26]. Combining the two methods of quantization and sparsification is presented in [27]. A different approach based on changing MPI operations to reduce communication has been discussed in [28] in which the AllReduce operation is replaced by relocating the intermediate data and using an Allgather operation.

All the above works are improvements on the AllReduce based training where averaging is done among all the processes. Another version of data-parallel training has been proposed where gradient averaging can be done with few selected PEs instead of all the PEs. In particular, the PEs can be assumed to be connected in a ring topology and each PE averages its gradient with the two neighbors in the ring [29]. The intuition is that a processor will communicate its information to its neighbors, which will then communicate to its neighbors and so on. Thus information gets propagated through all the processors. While it might seem that such a scheme will converge slowly with lesser accuracy due to the delayed propagation of information, the authors in [30] showed that there is negligible drop in accuracy. While [30] considers a regular communication pattern with neighbors which remain fixed throughout training, there are interesting gossip algorithms [31]–[33] that choose neighbors randomly and exchange information. Whether the neighbors remain fixed or not, the main advantage of these methods is the replacement of collective communication with peer-to-peer communication, resulting in significant savings in communication time and energy.

In this paper, we build on such an algorithm proposed in [30]. In [30], communication of messages with neighbor PEs happen at every epoch of training. This might not be necessary since values might not change significantly in every epoch. The main idea of this paper is that instead of communicating the parameters with neighbors at every epoch of training, communication is triggered in events only when necessary. We show that our algorithm achieves the same level of accuracy on the test set using approximately 70% lesser communication.

III. Background

As stated before, we consider the decentralized stochastic gradient descent algorithm mentioned in [30] as our baseline. The PEs are assumed to be connected in a ring, with each PE having two neighbors shown in Fig 2. This topology stays fixed throughout the training.

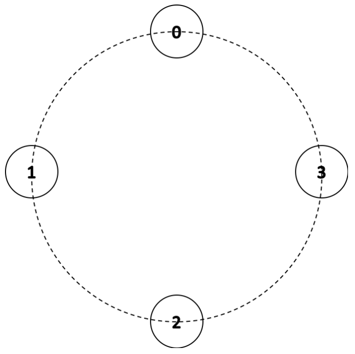

Fig. 2. Usual topology of processors executing training based on averaging with neighbors. For illustrative purposes, 4 PEs are shown here but the ring topology can be generalized to any number of PEs.

To mathematically formulate the training procedure of the algorithm in [30], consider x_I as a parameter in the I-th PE, $\mathcal{N}_I$ the set of neighbors of the I-th PE, F the loss function involved. Then $x_I(k+1)$ for epoch $k+1$ is updated as follows:

$$x_I(k+1) = W_{II}x_I(k) + \sum_{J\in\mathcal{N}_I} W_{IJ}x_J(k) - \gamma\nabla F_{I_k}(x_I(k)), \tag{1}$$

where $W = [W_{IJ}]_{1\leq I\leq N, 1\leq J\leq N}$ is the adjacency matrix. W_{II} corresponds to the weight of the state of the I-th PE while W_{IJ} corresponds to the weight of the state of J-th PE on the state of the I-th PE. For the topology in Fig 2, $\mathcal{N}_I = 2$. Note that in (1), the parameters of the neural network, i.e., the weights and biases are exchanged instead of the usual method of exchanging their gradients. The authors in [30] were able to show that such a scheme involving exchange of the parameters is almost equivalent to exchanging the gradients of the parameters. For details on how to choose W optimally, the reader is referred to [34]. Usually the values of W are taken to be $\frac{1}{\mathcal{N}_I+1}$, meaning the weight of the I-th PE is averaged with that of its neighbors after every epoch of training. After training concludes, the models in all the PEs are usually averaged to produce one model which is then evaluated on the testing dataset. This algorithm in [30] is stated in pseudo code in Algorithm A.

IV. Main Idea

The decentralized algorithm in (1) assumes that the model parameters in a PE are exchanged with neighbors in every epoch of the training. This might be a waste of resources since the parameters might not differ a lot in every epoch. The main idea behind our approach is to relax this requirement of communication with neighbors at every epoch of training. Every PE tracks the changes in the parameters of its model, i.e., the weights and biases. When the norm of a particular weight or bias in a PE has changed by some threshold, it is sent to the neighbors. Otherwise that particular parameter is not sent after that epoch to the neighbors and the neighbors continue averaging their own model with the last received parameters.

Algorithm A : Regular Communication in Data Parallel Machine Learning

```
for k = 0, 1, 2, ... K-1 do
    Randomly sample from dataset in $I$-th PE
    Compute the local stochastic gradient $\nabla F_{I_k}(x_I(k))$
    Communicate parameters to neighbors
    Update parameters using (1)
end for
Obtain averaged model from all PEs
```

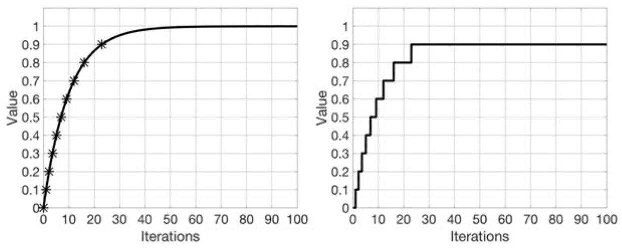

Fig. 3. Illustration of change in norm of parameters over iterations/epochs (taken from [12]). The left plot shows the norm of the parameter over iterations at the sender. The right plot shows the norm of that corresponding parameter used at the receiver.

Fig 3 illustrates this phenomenon. Usually in neural network training, the parameters change rapidly during the initial iterations, followed by sluggish changes before they reach an approximately steady value. Because of this behavior, we consider the concave curve in Fig 3 for illustration. The left plot shows the evolution of the norm of a parameter over training iterations. When this norm changes by more than a threshold (0.1 in Fig 3) from the norm of the previously communicated values, an event for communication is triggered as marked by an asterisk. The first event of communication is forced to take place at iteration $k = 0$ for convenience. The right plot shows the corresponding values that the receiving PE uses when averaging its parameters with parameters from this corresponding sending PE. When a new value is received due to an event of communication at the sender, that value is used by the receiver. Otherwise, the last communicated value is used at the receiver. Due to implementation challenges involving MPI one-sided communication which we describe in detail in the next section, this algorithm becomes completely asynchronous unlike the synchronous algorithm in [30]. So before we formulate our algorithm, let us define two sequences:

- $\lambda_{IJ}(k)$ - This is the sending sequence containing the current iteration number of the I-th PE when the I-th

PE sends a message to the J-th PE.

- $\tau_{IJ}(k)$ - This is the receiving sequence containing the current iteration number of the I-th PE when the I-th PE receives a message from the J-th PE.

Now, the sending PE sends a parameter x_I when it changes by a threshold δ. Thus the sending sequence $\lambda_{IJ}(k)$ is given by:

$$\lambda_{IJ}(k) = \begin{cases} k, & \text{if } (x_I(k) - x_I(\lambda_{IJ}(k-1))) \geq \delta \\ \lambda_{IJ}(k-1), & \text{otherwise.} \end{cases} \tag{2}$$

Consequently, the training algorithm at the receiving PE gets modified from (1) to

$$x_I(k+1) = W_{II}x_I(k) + \sum_{J \in \mathcal{N}_I} W_{IJ}x_J(\tau_{IJ}(k)) - \gamma \nabla F_{I_k}(x_I(k)). \tag{3}$$

The pseudo code for this algorithm is given in Algorithm B. It is important to have the distinction between the sending and receiving sequences because of the asynchronous nature of this algorithm. Note that the sending sequence $\lambda_{IJ}(k)$ for the I-th PE is the same for all $J \in \mathcal{N}_I$. However, the receiving sequence $\tau_{IJ}(k)$ for the I-th PE is different for different $J \in \mathcal{N}_I$.

Algorithm B : EventGraD - Event-Triggered Communication in Data Parallel SGD

```
for k = 0, 1, 2, ...K-1 do
    Randomly sample from dataset in $I$-th PE
    Compute the local stochastic gradient $F_{I_k}(x_I(k))$
    if $(x_I(k) - x_I(\lambda_{IJ}(k-1))) \geq \delta$ then
        Communicate parameter to neighbors
    end if
    Update parameters using (3)
end for
Obtain averaged model from all PEs
```

Choosing the threshold δ is a design problem. The efficiency of this algorithm depends totally on selecting appropriate thresholds. The simplest option would be to choose a constant static value. But selecting the appropriate value would involve a lot of trial and error. Further, when the neural network model changes, the process would have to be repeated all over again. In other words, if the number of neurons in a layer or the number/type of layers changes, a different value of the constant threshold would need to be chosen.

Instead, it is better to choose a dynamic threshold that is adaptive to the rate of change of the parameters, i.e., the slope of the parameters. This means that the threshold for each parameter is a function of the slope of the parameter. Whenever an event of communication is triggered, the slope is calculated between the current value and the last communicated value. This slope is multiplied by a horizon to calculate the threshold as shown in Fig 4. Thus the threshold has the following form:

$$\text{Threshold } \delta = \text{Slope} \times \text{Horizon}. \tag{4}$$

This threshold will be kept fixed until another event is triggered, resulting in calculation of a new threshold. The intuition behind making the threshold dependent on the slope is to save on communication as much as possible while ensuring that communication does not stop, i.e., happens once in a while. If a parameter is changing fast, it means that it will satisfy the criterion for communication soon - thus a high threshold (due to the high slope) can be suitable. However, if the parameter is changing slowly, there might a long period before the next communication happens which might slow down convergence of the overall algorithm. Hence the threshold is decreased (due to the low slope) to incentivize communication. The horizon is a hyperparameter that has to be chosen by the user. It might seem that the horizon hyperparameter requires tuning as well, thereby nullifying its advantages over the static threshold. However, we found empirically that the same horizon used for training different models worked well. This plays a huge role in keeping EventGraD portable across multiple models.

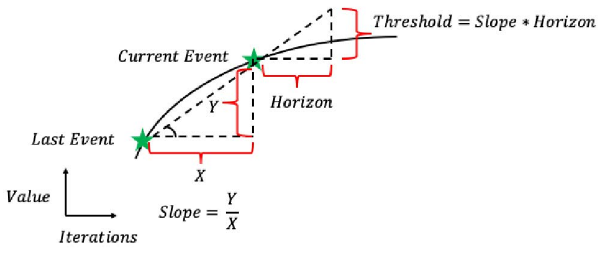

Fig. 4. Illustration of slope-based adaptive threshold. The right green star denotes the event of current communication while the left green star denotes the event of last communication. The slope is calculated between these two points which is then multiplied by the horizon to obtain the new adaptive threshold.

Often the parameters in a neural network have local minor oscillations due to the nature of the stochastic gradient descent algorithm. In other words, the change in parameters may not be as smooth as illustrated in Fig 4. This might be lead to calculation of a threshold that is not representative of the overall non-oscillatory trend in the parameter. In order for the threshold to reflect the aggregate trend in the parameter and not the local oscillations, the sender keeps a history of multiple previously communicated events instead of just one previous event. Then the average slope is calculated which is the mean of the slopes between two consecutive events in that history. This average slope is then multiplied by the horizon to obtain the threshold. The length of the history is a hyperparameter which is similar in notion to the length of a moving average filter. The higher the length, the smoother the trend but at the cost of increased computational complexity.

We note that the theoretical convergence properties of this algorithm have to be studied which we will address in our future work. However, in experiments, we noticed that the

EventGrad algorithm reaches the same level of accuracy as the algorithm in [30].

V. Implementation

There are a lot of popular frameworks for machine learning like PyTorch, TensorFlow, CNTK, etc. Almost all of these frameworks support parallel or distributed training. TensorFlow follows the parameter server approach for parallelization. PyTorch provides a module called DistributedDataParallel that implements AllReduce based training. Horovod is another framework developed by Uber that implements an optimized AllReduce algorithm. However, none of these frameworks provide native support for the training involving averaging with just neighbors. Hence we decide to implement them without using any of the distributed modules in these frameworks.

We use PyTorch and MPI for our implementation. Recently, PyTorch released a C++ frontend called Libtorch which is suitable for HPC environments unlike the usual Python interface. Libtorch has almost the same functionalities as PyTorch. Hence we combine the neural network training functionalities of Libtorch with communication routines in MPI to implement our algorithm. Note that the LBANN HPC toolkit introduced in [3] is also a great candidate for implementing distributed machine learning in HPC environments.

It is worth noting that the events for communication are dependent on the change in values of the parameters of the sender which is a local phenomenon. Thus, when an event is triggered in the sending PE, it can issue a MPI_Send operation. However, since the intended receiving PE is not aware of when the event is triggered at the sender, it does not know when to issue a MPI_Recv operation. So two-sided communication using MPI_Send and MPI_Recv cannot be used for our algorithm.

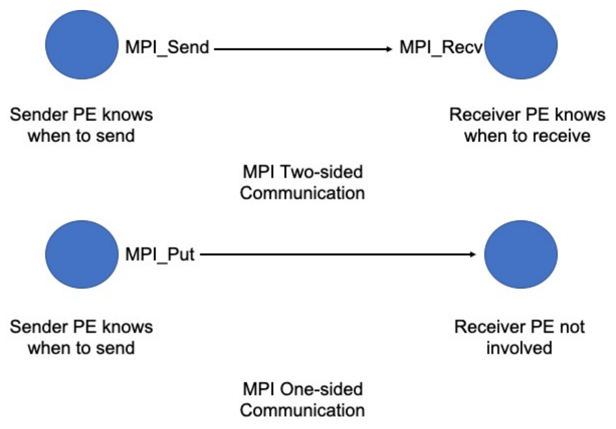
MPI_Send
MPI_Recv
Sender PE knows when to send
Receiver PE knows when to receive
MPI Two-sided Communication
MPI_Put
Sender PE knows when to send
Receiver PE not involved
MPI One-sided Communication

Fig. 5. Comparison of two-sided and one-sided communication.

We select one-sided communication for our purpose as was also the case in our previous work [12]. In one-sided communication, only the sending PE has to know all the parameters of the message for both the sending and receiving side and can remotely write/read from a portion of the memory of the receiver without the receiver's involvement - hence the alternate name of RMA (Remote Memory Access) [35]. That region of memory in the receiver is called *window* and can be publicly accessed. In our case, it is used to store the model parameters from the neighbors. So when an event for communication is triggered in the sending PE, it uses MPI_Put to write its model parameters directly into the window of the corresponding neighbor PE. An illustration of one-sided vs two-sided communication is provided in Fig 5.

MPI one-sided communication provides two options for establishing a communication channel between two PEs. The first is *active target synchronization* such as the fence mechanism or post/start/complete/wait (PSCW) mechanism. These mechanisms require active participation from the receiver which is not possible in our case since the receiver is not aware of the occurrences of events at the sender. Instead we select the other option of *passive target synchronization* where the receiving PE plays no role. The sending PE can start a RMA access epoch using MPI_Win_lock, transfer the message using MPI_Put and then end the access epoch by MPI_Win_unlock. Sometimes, before the unlock, MPI_Win_flush is used to ensure that RMA operations have completed.

In Libtorch, all the model parameters such as weights and biases are stored as tensors just like PyTorch. Using MPI one-sided operations to send tensors in Libtorch is challenging. At the moment, we convert the multidimensional tensors into contiguous unidimensional arrays, which are sent by the sender. At the receiver, these arrays are converted back into the corresponding multidimensional tensors again before they are used by the receiver. In future, we will look at how to allocate the memory of a Libtorch tensor as a MPI window so that one-sided operations can be done seamlessly without conversion into intermediate arrays.

VI. Evaluation

We perform experiments to evaluate the performance of our EventGraD algorithm. All our simulations are done on CPUs. We use an HPC cluster of nodes with each node having 2 CPU Sockets of AMD's EPYC 24-core 2.3 GHz processor and 128 GB RAM per node. The cluster uses Mellanox EDR interconnect. The MPI library chosen in Open MPI 4.0.1 compiled with gcc 8.3.0. The version of Libtorch used is 1.5.0. We choose the MNIST dataset for our experiments due its simplicity and popularity. Our neural network is made of convolutional and linear layers with max-pooling and dropout as shown in Fig 6. This is a simple model architecture which is popular for training the MNIST dataset and achieves around 98% accuracy [36]. For training this network, a learning rate of 0.05 is used with cross-entropy as the loss function.

As mentioned before, we focus on data-parallel training where the dataset is divided into multiple PEs, with each PE having a copy of the model in Fig 6. We run all our experiments on 4 PEs. The batch size on each PE is 64, making the effective batch size 256 on 4 PEs. The MNIST

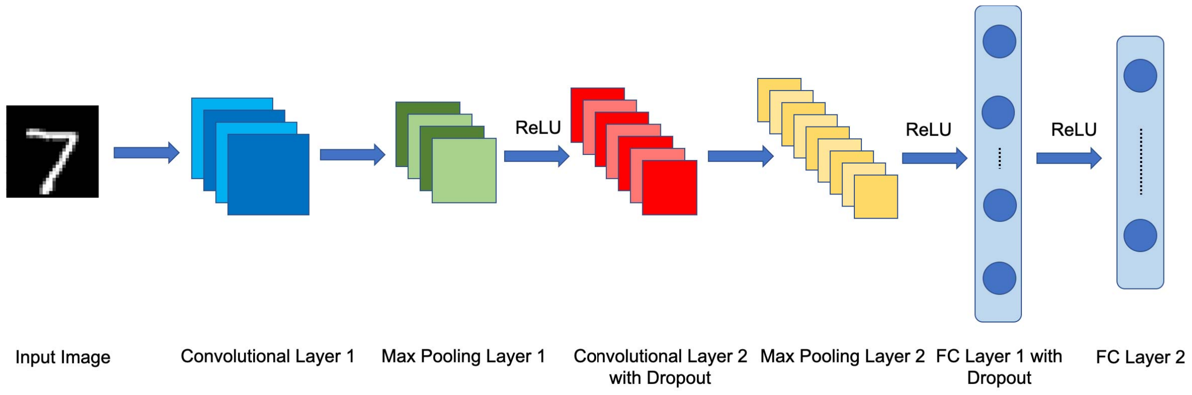

Fig. 6. Architecture of the Convolutional Neural Network used in our experiments. FC stands for Fully Connected (Linear) layers. ReLU is the rectified linear unit activation function.

dataset has 60000 training images, so each PE has 15000 images. The model is trained over 10 epochs. Hence we can estimate the total number of parameter update steps as $\frac{15000}{64} \times 10 \sim 2500$.

For the purpose of our experiments, we choose two different combinations of neurons and kernel sizes for the model in Fig 6. They are described in Tables I and II.

TABLE I
PARAMETERS OF LAYERS OF CNN-1

Layer	Parameters
ConvLayer1	InChannels= 1, OutChannels= 10, KernelSize= 5
MaxPool1	KernelSize= 2
ConvLayer2	InChannels= 10, OutChannels= 20, KernelSize= 5, DropoutProbability= 0.5
MaxPool2	KernelSize= 2
FCLayer1	Inputs= 320, Outputs= 100, DropoutProbability= 0.5
FCLayer2	Inputs= 100, Outputs= 10

TABLE II
PARAMETERS OF LAYERS OF CNN-2

Layer	Parameters
ConvLayer1	InChannels= 1, OutChannels= 10, KernelSize= 3
MaxPool1	KernelSize= 2
ConvLayer2	InChannels= 10, OutChannels= 20, KernelSize= 3, DropoutProbability= 0.5
MaxPool2	KernelSize= 2
FCLayer1	Inputs= 500, Outputs= 100, DropoutProbability= 0.5
FCLayer2	Inputs= 100, Outputs= 10

First we are interested in looking at how the parameters of the layers change over the update steps. As an example, Fig 7 shows the euclidean norm of the weights and biases of each layer for the model configuration in Table I in one PE. The change in weights for all the layers is similar to the illustration before in Fig 3 whereas biases stay almost flat throughout training. This means that not all parameters need to be communicated at every update step, thereby suggesting the possibility of saving messages by triggering communication in events.

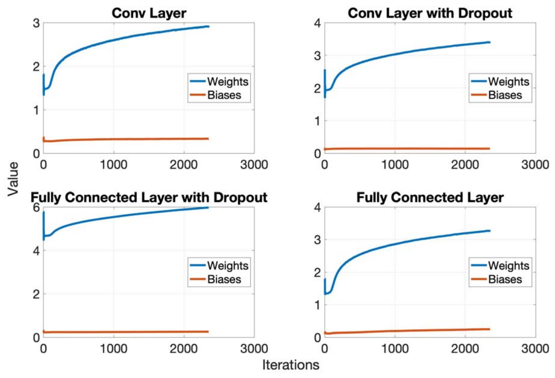

Fig. 7. Evolution of the euclidean norm of the weights and biases of the 4 corresponding layers in Fig 6 over the update steps for one PE.

Now we implement event-triggered communication to the training of the two models in Tables I and II. First, we implement a constant static threshold for both. We study how different thresholds affect the percentage of messages along with the training and testing accuracy of the network. The percentage of messages is measured with respect to the case with communication at every update step, equivalent to a threshold of 0 in the event-triggered communication algorithm. The plot in Fig 8 is for Table I and Fig 9 for Table II. We see that with increase in threshold, the number of messages decreases resulting in savings in communication while the accuracy stays practically same up to a point. After that point, the accuracy falls sharply indicating that thresholds above that are not suitable. The threshold from which accuracy sharply degrades is around 0.6×10^{-3} for Table I and around 2×10^{-3} for Table II. Since the two models are different, the range of thresholds for them are also different. It is not tractable to experiment with different thresholds for different models and then choose a suitable one for that specific model.

Instead we emphasize on the adaptive threshold proposed

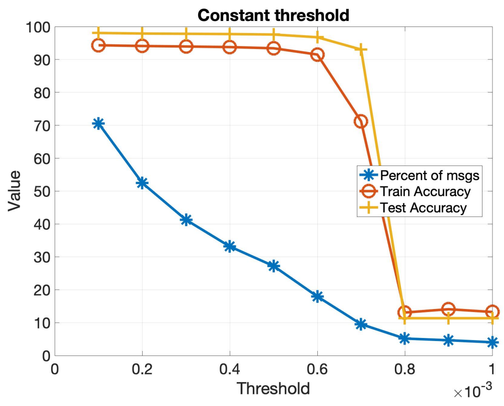

Fig. 8. Plot of percentage of messages and accuracy versus the threshold for the model configuration in Table I.

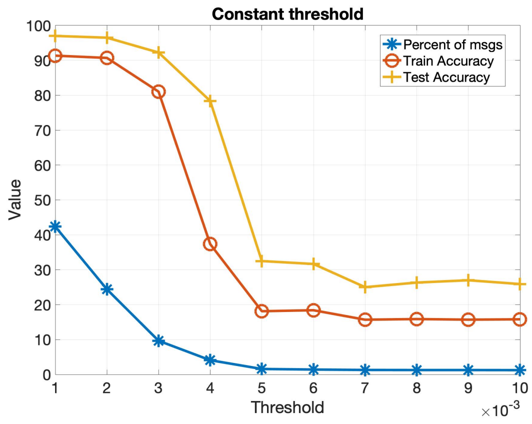

Fig. 9. Plot of percentage of messages and accuracy versus the threshold for the model configuration in Table II.

in Section IV. As a reminder, the adaptive threshold is determined by estimating the slope of the parameters multiplied by a horizon as shown before in Fig 4. A value of 1 for the horizon and a value of 2 for the length of the history of previously communicated values is chosen for our experiments. Using this threshold, the model in Table I achieves an accuracy of 97% with 40% of the messages. The model in Table II achieves an accuracy of 95% with 25% of the messages. No different thresholds have to be chosen here for different models unlike the constant threshold. Hence adaptive threshold is the better choice.

It is useful to study the accuracy vs the percentage of messages sent for both the types of threshold for both the models. Fig 10 shows that plot for the model in Table I and Fig 11 for Table II. We see that the slope-based adaptive threshold achieves a high level of accuracy with only 25 to 40% of the messages. However, there are values of the constant threshold that achieve the same accuracy with even lower percentage of the messages. To achieve that using the adaptive threshold would probably require the threshold to be a non-linear function of the slope instead of the linear function considered in this paper. Thus finding out the optimal way to choose the adaptive threshold is an open problem which we leave for future work.

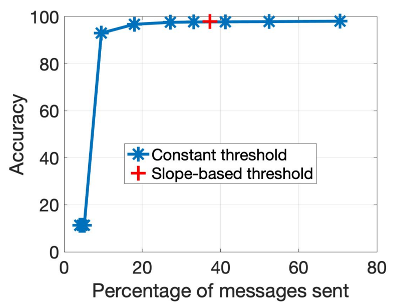

Fig. 10. Accuracy versus percentage of messages sent for the model configuration in Table I. The blue asterisks denote the values of the threshold in Fig 8, with the highest threshold on the left and the lowest threshold on the right.

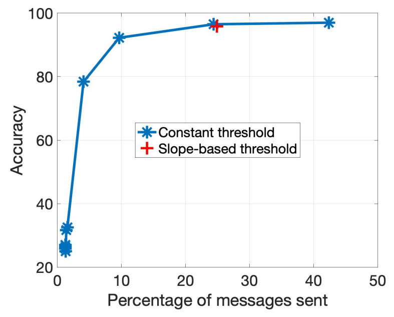

Fig. 11. Accuracy versus percentage of messages sent for the model configuration in Table II. The blue asterisks denote the values of the threshold in Fig 9, with the highest threshold on the left and the lowest threshold on the right.

VII. Conclusion

This paper introduces a novel communication mechanism in parallel training of neural networks using stochastic gradient descent. The proposed EventGraD algorithm based on event-triggered communication reduces communication up to around 70% while maintaining the same accuracy. The choice of the threshold for triggering events is chosen adaptively based on the slope of the parameter values, thus ensuring that it can be applicable for different neural network configurations without any change. The implementation challenges of the algorithm using MPI and PyTorch, specifically the need for MPI one-sided communication, are also discussed.

For future work, we want to apply this algorithm for training state-of-the-art neural network models such as ResNet-50 and VGG-16 on larger datasets such as ImageNet. We plan on using GPUs for those simulations, so we have to implement one-sided communication operations between GPUs using tools such as Nvidia GPUDirect. Further, we are working on characterizing theoretical properties of this algorithm such as the optimal way to choose the adaptive threshold and the rate of convergence with the adaptive threshold.

VIII. Acknowledgements

This research was supported in part by the University of Notre Dame Center for Research Computing through its computing resources. The work of the authors was supported in part by ARO grant W911NF1910483, DARPA FA8750-20-2-0502 and NSF CBET-1953090.

References

[1] Sujatha R Upadhyaya. Parallel approaches to machine learning—a comprehensive survey. *Journal of Parallel and Distributed Computing*, 73(3):284–292, 2013.

[2] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[3] Brian Van Essen, Hyojin Kim, Roger Pearce, Kofi Boakye, and Barry Chen. Lbann: Livermore big artificial neural network hpc toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, pages 1–6, 2015.

[4] Steven R Young, Derek C Rose, Travis Johnston, William T Heller, Thomas P Karnowski, Thomas E Potok, Robert M Patton, Gabriel Perdue, and Jonathan Miller. Evolving deep networks using hpc. In *Proceedings of the Machine Learning on HPC Environments*, pages 1–7, 2017.

[5] Jingoo Han, Luna Xu, M Mustafa Rafique, Ali R Butt, and Seung-Hwan Lim. A quantitative study of deep learning training on heterogeneous supercomputers. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[6] Arpan Jain, Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K Panda. Scaling tensorflow, pytorch, and mxnet using mvapich2 for high-performance deep learning on frontera. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, pages 76–83. IEEE, 2019.

[7] Junqi Yin, Shubhankar Gahlot, Nouamane Laanait, Ketan Maheshwari, Jack Morrison, Sajal Dash, and Mallikarjun Shankar. Strategies to deploy and scale deep learning on the summit supercomputer. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, pages 84–94. IEEE, 2019.

[8] Keren Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[9] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, et al. Doe advanced scientific computing advisory subcommittee (ascac) report: top ten exascale research challenges. Technical report, USDOE Office of Science (SC)(United States), 2014.

[10] AT Chronopoulos and Charles William Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153–168, 1989.

[11] Mark Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, UC Berkeley, 2010.

[12] Soumyadip Ghosh, Kamal K Saha, Vijay Gupta, and Gretar Tryggvason. Event-triggered communication in parallel computing. In *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*, pages 1–8. IEEE, 2018b.

[13] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 1–8. IEEE, 2016.

[14] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*, pages 5973–5983, 2018.

[15] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[17] `https://github.com/soumyadipghosh/eventgrad/tree/master/dmnist/event`.

[18] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[19] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015.

[20] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *Advances in neural information processing systems*, pages 685–693, 2015.

[21] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[22] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[23] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[24] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[25] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[26] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. Sparcml: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2019.

[27] Debraj Basu, Deepesh Data, Can Karakus, and Suhas Diggavi. Qsparse-local-sgd: Distributed sgd with quantization, sparsification and local computations. In *Advances in Neural Information Processing Systems*, pages 14695–14706, 2019.

[28] Sunwoo Lee, Ankit Agrawal, Prasanna Balaprakash, Alok Choudhary, and Wei-Keng Liao. Communication-efficient parallelization strategy for deep convolutional neural network training. In *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pages 47–56. IEEE, 2018.

[29] Kun Yuan, Qing Ling, and Wotao Yin. On the convergence of decentralized gradient descent. *SIAM Journal on Optimization*, 26(3):1835–1854, 2016.

[30] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 5330–5340, 2017.

[31] Michael Blot, David Picard, Matthieu Cord, and Nicolas Thome. Gossip training for deep learning. *arXiv preprint arXiv:1611.09726*, 2016.

[32] Peter H Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. How to scale distributed deep learning? *arXiv preprint arXiv:1611.04581*, 2016.

[33] Jeff Daily, Abhinav Vishnu, Charles Siegel, Thomas Warfel, and Vinay Amatya. Gossipgrad: Scalable deep learning using gossip communication based asynchronous gradient descent. *arXiv preprint arXiv:1803.05880*, 2018.

[34] Stephen Boyd, Persi Diaconis, and Lin Xiao. Fastest mixing markov chain on a graph. *SIAM review*, 46(4):667–689, 2004.

[35] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.

[36] Libtorch mnist example. `https://github.com/pytorch/examples/blob/master/cpp/mnist/mnist.cpp`. Accessed: 07-11-2020.