REDSYNC : REDUCING SYNCHRONIZATION TRAFFIC FOR DISTRIBUTED DEEP LEARNING

Jiarui Fang[*], **Haohuan Fu, Guangwen Yang**
Tsinghua University
fjr14@mails.tsinghua.edu.cn, {haohuan,ygw}@tsinghua.edu.cn

Cho-Jui Hsieh
University of California, Davis
chohsieh@ucdavis.edu

ABSTRACT

Data parallelism has already become a dominant method to scale Deep Neural Network (DNN) training to multiple computation nodes. Considering that the synchronization of local model or gradient between iterations can be a bottleneck for large-scale distributed training, compressing communication traffic has gained widespread attention recently. Among several recent proposed compression algorithms, Residual Gradient Compression (RGC) is one of the most successful approaches—it can significantly compress the message size (0.1% of the original size) and still preserve accuracy. However, the literature on compressing deep networks focuses almost exclusively on achieving good compression rate, while the efficiency of RGC in real implementation has been less investigated. In this paper, we explore the potential of application RGC method in the real distributed system. Targeting the widely adopted multi-GPU systems, we proposed an RGC system design called RedSync, which includes a set of optimizations to reduce communication bandwidth while introducing limited overhead. We examine the performance of RedSync on two different multiple GPU platforms, including a supercomputer and a multi-card server. Our test cases include image classification and language modeling tasks on Cifar10, ImageNet, Penn Treebank and Wiki2 datasets. For DNNs featured with high communication to computation ratio, which have long been considered with poor scalability, RedSync shows significant performance improvement.

1 INTRODUCTION

The great success of deep neural networks (DNNs) is mainly powered by training big models on big data. However, the training process is extremely time-consuming and has become a critical debt to obtain a large-scale DNN model. In order to utilize resources from multiple computing nodes, data parallelism has emerged as the most popular choice to accelerate the training of DNNs. In a typical data parallel approach, each computing node holds an individual subset of training data, as well as a copy of model parameters. Optimizers, like Stochastic Gradient Descent (SGD), are performed locally on these node producing weight-updates after each iteration. A synchronization scheme collects all local weight-updates together to generate a global weight-update and applies it to every node.

However, the communication bandwidth of network fabric has become the bottleneck limiting data parallel performance. On the one hand, models of DNNs, which have already contained tens to hundreds of layers and totaling 10-20 million parameters today, continues to grow bigger. Due to the requirement of transmitting large weight-updates among all computing nodes at each iteration, model synchronization poses a higher challenge to network bandwidth. On the other hand, recent years have witnessed that the breakthrough of DNN training accelerators, which is shifting

[*]Work is done when he is a visiting scholar at UC Davis.

the bottleneck of training towards communication across models. To fully hide communication with computation, the required bandwidth for inter-accelerator communication scales up directly with raw hardware performance. As the evolution of the inter-connected network bandwidth is not as fast as computing hardware, synchronization overhead is increasingly becoming the bottleneck of data parallelism on distributed systems using new computing hardware.

A lot of studies focused on reducing the communication cost between nodes by shrinking the size of transmitting data. Targeting one type of most popular implementations of data-parallel SGD, where each worker communicate its entire gradient update to all other processors. Approaches based on Residual Gradient Compression (RGC) (Strom (2015); Aji & Heafield (2017); Chen et al. (2017); Lin et al. (2017); Sattler et al. (2018)) solve the communication bandwidth problem by compressing the transmitting gradients. In terms of theoretical compression ratio, current RGC variants, such as (Sattler et al. (2018); Lin et al. (2017)), are able to achieve a compression ratio as 0.1% and are still able to maintain no accuracy loss. The potential application scenarios in (Sattler et al. (2018); Lin et al. (2017)) are limited to systems including multiple mobile devices and suffering from lower network bandwidth. In this case, network communication is the only bottleneck in the system and the computing overhead brought by compression and decompression operations is ignored. As we all know, the most common scenario of neural network training is multi-GPU systems equipped high-quality network infrastructures. Unfortunately, few recent studies have discussed the potential performance gain after integrating the RCG method to these distributed GPU systems. The challenge of applying the RGC method on distributed GPU systems comes from two aspects. First, it is hard to propose efficient compression and decompression solutions for RGC on GPU architecture. Especially for the compression process, selecting important elements to transmit is inherently inefficient to be parallelized. Even if using the state-of-the-art radixSelect algorithm on GPU, the overhead of compression is much higher than the benefits of network bandwidth reduction. Second, synchronization of sparse data structures is nontrivial to be supported with existing efficient communication libraries, such as Message Passing Interface (MPI), which are designed for dense data structures.

Targeting multi-GPU systems, a highly-efficient RGC implementation called RedSync is proposed to reduce the communication traffic while taking compression and decompression overhead into account. First, we proposed a set of parallel-friendly top-0.1% selection methods to compress transmitting gradients, which are orders of magnitude faster than the state-of-the-art radixSelect method on GPU. We also applied an efficient quantization technique on compressed data to further reduce communication bandwidth requirement. Second, we proposed a sparse synchronization scheme to transmit compressed data. Third, using a cost model to analyze sparse and dense synchronization overhead, we pointed out potential performance gain and the bottleneck of our RGC implementation. Finally, we evaluated the performance of RedSync on the scale of the at most 128 GPUs for a variety of DNNs on different datasets.

2 BACKGROUND

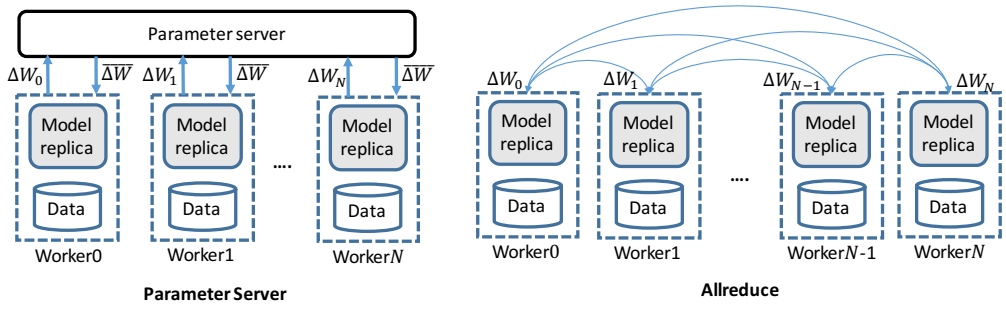

Figure 1: Two distributed DNN system implementations.

In this section, we first introduce the standard synchronized data parallelism for SGD and then present two popular distributed implementations for it.

2.1 DATA PARALLELISM OF SGD

We introduce a variant of data parallel method for SGD, whose communication target is gradients of each layer. We denote a DNN model as $f(\mathbf{w})$, where $\mathbf{w}$ is the vector of parameters. We assume a system employs N workers for data parallel. Each worker, say the k-th worker, holds a local dataset $\chi_k = \{(x_{k,1}, y_{k,1}), ..., (x_{k,m_k}, y_{k,m_k})\}$ with size m_k and a local copy of $\mathbf{w}$. It is worth noting that in the synchronous data parallel SGD method, each node has the same copy of the global weight. At iteration t, the SGD is implemented in the following three steps.

1. Local Training: each worker sample a mini-batch of data χ_k^t from local dataset χ_k. Calculating the gradients $\nabla f(\chi_k^t; \mathbf{w}_t)$ of loss function $f(\mathbf{w}_t, \chi_k^t)$ with forward and backward propagation.

2. Synchronization: Achieve a global weight updates G_k across all workers by averaging all local updates. $G_k \leftarrow \frac{1}{N}\sum_{k=1}^{N} \nabla f(\chi_k^t; \mathbf{w}_t)$.

3. Updating: Each worker applies synchronized updates. The vanilla SGD updates weight w by $\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha_t G_k$, where α_t is the learning rate of t iteration. To accelerate convergence, modern DNNs applies some SGD variants (Ruder (2016)), like momentum SGD and ADAM.

2.2 DISTRIBUTED IMPLEMENTATIONS

The synchronization step mentioned above requires different workers to exchange gradients data. In practice, two distributed implementations are adopted as shown in Figure 1.

One method is to perform the allreduce operation on the gradients among all nodes and to update the parameters on each node independently. It commonly works in a dead reckoning way: Each node maintains the same model replica. After every iteration, it communicates its gradients of individual weights to all other nodes with an allreduce operation. For all copies of the DNN to stay synchronized, all nodes receive exactly the same gradients and apply the averaged gradients to the local replica of DNN model. Such synchronization method is able to take advantage of highly-optimized allreduce operation on HPC systems (Thakur et al. (2005)). Consequently, it has been widely adopted in state-of-the-art large-scale CNN training tasks (Goyal et al. (2017) and You et al. (2017)).

The other method is using the parameter servers (Dean et al. (2012); Li et al. (2014)) as the intermediary which store the parameters among several nodes. The nodes push the gradients to the servers while the servers are waiting for the gradients from all nodes. Once all gradients are sent, the servers update the parameters, and then all nodes pull the latest parameters from the servers. If we set the parameter servers as independent computing node, the synchronization communication can be a bottleneck of the system. If the parameter server is distributed among local worker, the push operations become a scatter allreduce and pull operation became an allgather. In such case, it turns out to be a allreduce operation. Such a system is suitable asynchronous SGD method (Ho et al. (2013)), by which each worker may maintain delayed weights locally. However, asynchronous SGD may not result in the same accuracy as synchronous one and results vary each time. Additionally, most of the time it is not necessary for GPU training, because Sync SGD works best for less than a couple hundred GPU machines. Therefore, the research goal of this paper is to accelerate synchronized data parallel training using allreduce synchronization.

3 RELATED WORKS

Many works are proposed to reduce communication cost in data parallel training of DNN. These jobs can be divided into two major categories.

Sparsification : This type of method restricts weight-updates to a small subset of parameters. Strom (2015) proposed a threshold quantization to only send gradients larger than a predefined constant threshold for fully-connected layers. All the other relative small gradients are accumulated to residual. This is the first work introducing RGC to data parallel approach. Considering a predefined threshold is hard to be chosen appropriately, Aji & Heafield (2017) applied the DGC framework to RNN but proposed to use a sparsity rate instead of a fixed threshold. They selected top $p\%$ gradients to communicate according to their magnitude. AdaComp Chen et al. (2017) is a variant of RGC with a bin-based gradient compression scheme, which can self-adapts its compression rate across

minibatches and layers. Deep Gradient Compression proposed by Lin et al. (2017) pushed compression ratio to 0.01% by adopting some improvements to RGC while still ensuring the accuracy. They examine their method on a variety of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) with multiple datasets. Improvements include momentum correctness, gradient clipping, and warm-up scheme. Sparse Binary Compression proposed by Sattler et al. (2018) pushed the model compression ratio further by binarizing the sparsified gradients and combined an optimal weight update encoding. However, the recent design as mention in (Chen et al. (2017); Lin et al. (2017); Sattler et al. (2018)) remain at the level of theoretical analysis, and its performance is not verified on real systems.

Quantization : Another line of related research studies to reduce communication bandwidth is focused on quantizing the communication data to low-precision values. Seide et al. (2014) proposed 1-bit SGD to reduce gradients transfer data size and achieved $10\times$ speedup in traditional speech applications. Alistarh et al. (2017) proposed another approach called QSGD which balances the trade-off between accuracy and gradient precision. They integrated QSGD into CNTK and show 1.8x performance gain on ResNet-152. They test their system on Amazon EC2 instances with 10 Gbps peak peer-to-peer network bandwidth. However, the method proposed in this paper can be applied to high-quality network infrastructures. Similar to QSGD, a recent work called TernGrad proposed by Wen et al. (2017) used 3-level values to quantize gradients in 2 bit for CNNs, but not implemented in a real system. The theoretical compression ratio is limited in TernGrad. Two-bit quantization is only achievable when each worker pushes local gradients to the parameter. When each worker pulls parameters from the parameter server, the quantization level is increasing linearly with the number of workers. Currently, these quantitation methods can only be applied to parameter server systems. Because quantized bit-format data structure cannot do the reduction with average and summation on the fly, they can hardly using optimized allreduce operation to reduce communication traffic. Therefore, communication traffic reduction by quantitation-based methods is limited compared with allreduce based data parallel implementation.

4 Residual Gradient Compression

It is well established that the gradients generated by DNN training contain a lot of redundant information. Only a small fraction of the weights are required to be updated after each mini-batch. Elements of the gradient that are small enough can safely be delayed much longer than the typical mini-batch size. To avoid losing information, we locally accumulate these small remaining elements to a data pool called residual, which can be accessed in the next iteration. A recent work Lin et al. (2017) also implied the effectiveness of RGC in ways. First, accumulating the gradient of an individual parameter to residual through time is similar to increase the batch size for training. Second, updating the current model with staled gradients is equivalent to introducing synchronousness into training.

The general framework of RGC method is illustrated in Algorithm 1. As shown in line 3 and 4, gradients are calculated by forward-backward propagation on randomly sampled mini-batch of input data. We accumulate the gradients $\nabla_j f(x_k; w_t)$ of layer k to residual V_j^k of that layer. The key to reducing communication traffic in RGC is that it only transmits a subset of residuals, the indices of which are indicated as $Masks$ in Algorithm 1, instead of all the parameters in gradients during the training process. We refer the subset of the parameters in residual to be communicated during the parallel training as *communication set*. Communication-set is identified by selecting the important elements from residuals according to their absolute magnitude.

5 Design and Implementation of RedSync

5.1 Overall Workflow

We design RedSync to accelerate DNN training on multiple GPU systems, as illustrated in Figure 2 In this figure, W and ΔW are weights and gradients and R are residuals of each DNN layer. a single training iteration mainly consists of four steps. In Step 1, it calculates the gradients of each DNN layer by forward and backward propagation. In Step 2, after added with gradients, a subset of dense residuals is selected as communication-set and then is compressed into sparse data structures.

Algorithm 1 Residual Gradient Compression on each node

Input: node id k, the number of node N
Input: dataset χ
Input: mini batch size b per node
Input: initial learnable parameters $w = w[0], ..., w[\#layer]$
Input: density ratio D

$V^k \leftarrow 0$
for $t = 0, 1, ...max\_iter$ **do**
  sample b elements as x_k from χ
  for $j = \#layer, \#layer - 1, ..., 0$ **do**
    calculate $G_j^k \leftarrow \nabla f(x_k; w)$ with forward and backward propagation on x_k
    $V_j^k += G_j^k$
    $Masks \leftarrow$ communication-set-selection(V^k, D)
    $G_j \leftarrow \frac{1}{N} \sum_{k=1}^{N}$compress$(V_j^k \cdot Masks)$
    $V_j^k \leftarrow V_j^k \cdot (1 - Masks)$
  end for
  $w_{t+1} \leftarrow$ SGD(w_t, decompress(G^k))
end for

To reduce memory copy bandwidth between the host and the GPU device, the compression process should be efficiently done inside device memory. In Step 3, RedSync synchronizes the compressed communication-sets of each GPU. In Step 4, RedSync applies synchronized compressed residuals to update weight parameters. The existing DNN training framework has already provided with highly-efficient implementation for Step 1. Therefore, this paper focuses on the design of efficient communication-set compression/decompression methods and synchronization approaches for the compressed data structure.

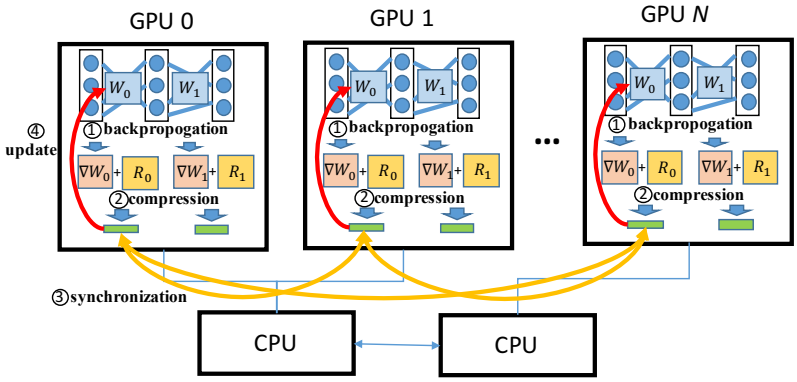

Figure 2: Workflow of DNN training with RedSyn.

5.2 Compression

Residual Compression with an efficient communication-set identification method is essential for the system's overall performance. Since a predefined threshold is difficult to determine, recent works (Aji & Heafield (2017); Lin et al. (2017); Sattler et al. (2018)) suggest to select top $p\%$ elements from residuals of each layer as the communication-set. It is well-known that, by applying *Quickselect* algorithm (Hoare (1961)), the time complexity of a top-k selection on a list of n elements using a single-core CPU is $O(n)$. However, such method is hardly parallelized on multi-thread architectures, like GPU. On GPU architecture, one of the most efficient top-k selection methods can be implemented based on *radixSelect* algorithm (Alabi et al. (2012)). This algorithm iterates from the most important digit by performing digit radix sorts to determine what each digit should be as the kth largest number. The core of the digit radix sort is a prefix-sum operation. Although time complex-

ity of a work efficient version prefix-sum is limited to $O(n)$ addition operations, *radixSelect* itself requires several prefix-sum operations for each bit and therefore can be extremely time-consuming to serve as a communication-set identifier. Despite Lin et al. (2017) has already achieved a compression ratio of up to 0.1%, the overhead to identify communication-set with a top-k selection is nontrivial to be implemented. As shown in Figure 3, we can see that the cost of the radixSelect algorithm grows linearly with the size of the parameter. The computation time for top-k on a Titan X GPU is even slightly higher than the synchronization time of these parameters with an allreduce operation through a 3.5 GBps network. To avoid performing a top-k operation on a large number of residual parameters, we propose two communication-set selection algorithms called *trimmed top-k selection* and *threshold binary search selection*, which are more efficient to be parallelized on GPU.

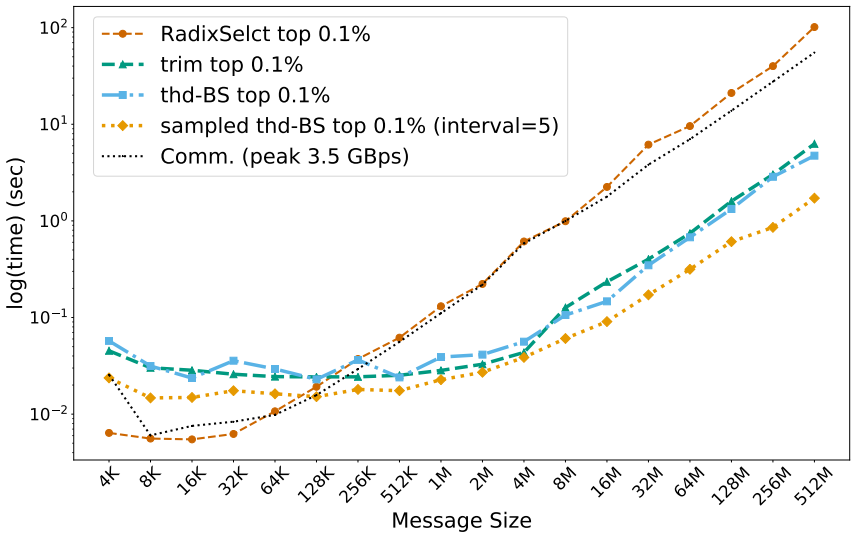

Figure 3: Performance of four communication-set selection methods under different parameters. Elements in the data list are generated randomly from standard uniform distribution. *Comm.* illustrates the time taken to synchronize these data through a network with a peak bandwidth of 3.5GBps by allreduce operation. Performance is measured as total time cost for 100 times independent operations.

5.2.1 TRIMMED TOP-k SELECTION

The feature of top-k used in the RGC algorithm is that we only select a very small part of the entire element list. We can use statistical features to remove most of the smaller elements and limit radixSelect operation on a relatively small number of elements. As shown in Algorithm 2, we first calculate the mean and maximum of residuals' absolute values of this layer. A relative large threshold value is chosen according to mean and maximum of all values, for example, $0.9 \times (max - mean) + mean$. With this threshold, we can get the number of elements whose absolute values are greater than it. If the number is smaller than k, we dynamically decrease the threshold until we find the number of parameters whose absolute value above the threshold is larger than k. Then we trim all elements that are less than the threshold and perform a top-k selection operation using radixSelect on the remaining elements. Finding mean and maximal values and counting the nonzero elements on an array can all be implemented with a single prefix-sum operation, which are well-optimized routines on GPU architecture. As shown in line 10 of Algorithm 2, we have to filter the elements larger than a specific threshold and obtain a continuous element list, containing indices of non-zero positions in X. It is a typical *stream compaction* problem, which can be implemented based on *prefix-sum algorithm* on GPU architecture by Sengupta et al. (2006).

5.2.2 THRESHOLD BINARY SEARCH SELECTION

For some layers with a large number of parameter elements, because the communication-set itself is large, radixSelect on a subset of elements will be still a very time-consuming operation, even if its size is slightly larger than commination-set. In order to completely avoid using radixSelect

operation on GPU, we propose a binary-search-based method for communication-set selection. Not really identifying the kth largest element, we find a threshold element which is between the kth and the $2k$th largest element. Therefore we can still ensure at least k largest elements included in the communication-set. As shown in Algorithm 3, we use a binary search algorithm to find the magnitude of the threshold element. The condition for the algorithm to terminate is that the number of elements greater than the threshold element is between k and $2k$. To avoid excessive searches, it will always be terminated when the difference of left bound and right bound is less than a small value ϵ.

For layers with large sizes, such as the first fully-connected layer in VGG16 and softmax layer in LSTM, even if we avoid using radixSelect operation on GPU, the time for `count_nonzero` operation is still not negligible. We further improve the efficiency of the selection algorithm by reducing the `count_nonzero` operations. We recommend that, after a threshold binary search for this layer, the threshold element can be reused in the next few iterations. The interval is empirically set to 5. On average, the selection algorithm introduces only one `nonzero_count` overhead.

Algorithm 2 trimmed top-k Selection

Input: tensor to be compressed X
Input: number of elements remained k
Output: $< indice, values >$

$mean \leftarrow$ mean(abs(X))
$max \leftarrow$ max(abs(X))
$\epsilon \leftarrow 0.2$
$ratio \leftarrow (1 - \epsilon)$
nnz = count_nonzero(abs(X) $> threshold$)
while $nnz > k$ **do**
  $threshold \leftarrow mean + ratio \times (max - mean)$
  nnz = count_nonzero(abs(X) $> threshold$)
  $ratio = ratio - \epsilon$
end while
$indice \leftarrow topk$(filter(abs(X) $> threshold$))
$values \leftarrow X[indice]$

Algorithm 3 Top-k selection with threshold binary search selection

Input: tensor to be compressed X
Input: number of elements remained k
Input: Termination condition parameters ϵ
Output: $< indice, values >$

$mean \leftarrow$ mean(abs(X))
$max \leftarrow$ max(abs(X))
$l \leftarrow 0.0$; $r \leftarrow 1.0$; $threshold = 0.0$
while $r - l > \epsilon$ **do**
  $ratio = l + (r - l)/2$
  $threshold \leftarrow mean + ratio \times (max - mean)$
  nnz = count_nonzero(abs(X) $> threshold$)
  if $nnz > k$ and $2k > nnz$ **then**
    break
  else if $nnz < k/2$ **then**
    $r = threshold$
  else
    $l = threshold$
  end if
end while
$indice \leftarrow$ nonzero_index(filter(abs(X) $> threshold$))
$values \leftarrow X[indice]$

In Figure 3, we compared the time cost of different selection approaches on parameter arrays of different sizes. Compared with directly performing radixSelect method, both methods we proposed can significantly reduce the selection time for large parameter sizes. Take a top-0.1% selection on elements list with a size of 64 MB as an example. Trimmed selection and sampled threshold binary search selection are 38.13 $\times$ and 16.17 $\times$ faster than radixSelect. Finally, we compared our proposed method to two other communication-set selection plans, which exist only in the design phase but are not really implemented. Lin et al. (2017) supposed to sample only 0.1% to 1% elements from residuals and perform top-k selection on the samples to estimate the threshold for the entire population. If the number of elements exceeding the threshold is far more than expected, they perform another top-k on the already-selected residual. On GPU, randomly sampled elements should also be gathered by a stream compression operation, time cost of which is similar to a filter operation. Additionally, as shown in Figure 3, top-k selection with radixSelect on a small amount of data (<32KB) is not as efficient as they imagined. Due to requiring multiple top-k selection operations and even repeated sampling, this method cannot be more efficient than our *trimmed top-k* method, which always requires once selection. Chen et al. (2017) supposed to divide the entire residual list for each layer

into several bins and add elements above an adaptive threshold in each bin to communication-set. Such method needs many small stream compression operations to collect selected elements in each bin and then combine them together, which is less efficient than a stream compression operation on entire list. Since it may miss important elements and pick some small elements, their method results in a smaller compression ratio than selecting top-0.1% elements. Additionally, threshold in their method is required to be fine-tuned for different DNN layer types.

5.2.3 QUANTIZATION OF COMPRESSED RESIDUALS

We further investigate the possibility of introducing quantization technique to compressed residuals. By setting the values of all elements of the same sign in the communication-set to their mean, we can halve the communication bandwidth requirement by transmitting only one floating-point number instead of k value information, as well as k integers as index information. In order to facilitate quantization compression, we slightly modify our communication-set selection to ensure elements in it are all of the same sign. It can be achieved by choosing the largest k elements and the smallest k elements as communication-set alternatively. In other words, if we select the top k elements in this layer as the communication-set at current iteration, we will choose bottom k elements as the communication-set for the next iteration. Consequently, *sampled threshold binary search selection* cannot be used with quantization. It is worth noting that we do not quantify the output layer of the DNN, in order to distinguish the correct classification information. Therefore, although the softmax layer in LSTM is large, we can not quantify it. Strom (2015) has also utilized similar quantization to compressed residual but introduced more communication traffic than us. Their method quantizes both positive and negative elements. For each element, at least 1 bit representing the sign of the element is required to be transmitted. Our approach guarantees that all elements have the same sign, thus reducing the amount of information that needs to be passed.

5.3 SPARSE SYNCHRONIZATION

Synchronization of dense gradient structures in traditional distributed DNN system can be simply implemented with an allreduce operations, which has been well-studied on multiple-GPU systems Awan et al. (2017). However, the design of a sparse allreduce in a distributed setting is not as simple because each worker may contribute different non-zero indices in its compressed residuals. For example, training VGG16 on Cifar10 dataset using 16 GPUs with a compression ratio as 0.1% for each node, the average density of synchronized residual of all layers is 1.55%. That is to say, there are very few overlapping indices of the communication-set distribution of different nodes. We utilize the allgather operation, an operation in which the data contributed by each node is gathered at all nodes, to implement sparse allreduce. The message representing a compressed residual of each node should include the information of indices and values of elements in communication-set. Instead of using two allgather operations for indices and values message separately, we package the indices and values into a single message to reduce latency. When using threshold binary search selection, the length of each node's message is different. As a result, the packaged message should include an initial element, which indicates the length of the compressed elements. Although Lin et al. (2017) suggest reducing indices message size by storing distances between the current non-zero element and previous one instead of the exact positions, we observe that the encoding process is hardly be parallelized on GPU architecture. Therefore we do not further compress indices information. We also apply *tensor fusion* technique to batch small allgather operations. It can reduce the time of communication initialization and increase the amount of data transferred at a time so as to increase the available bandwidth.

5.4 DECOMPRESSION

After allgather operation completed, each node achieves the compressed message of each layer, which contains N communication-sets from the different node. We add the compressed parameters to the corresponding position of the local model after scaling with learning rate. It can be seen as an operation that adds a sparse array to a dense array, which has been fully-optimized in Level 1 function axpyi() of cuSparse library [1] on GPU.

[1] https://docs.nvidia.com/cuda/cusparse/index.html

5.5 PERFORMANCE MODEL OF COMMUNICATION COST

To analyze the potential performance gain of sparse synchronization, we adopt a widely-used performance model to estimate the communication cost of synchronization in terms of latency and bandwidth use. We assume that the time taken to send a message between any two nodes can be modeled as $\alpha + n\beta$, where α is the latency (or startup time) per message, independent of message size, β is the transfer time per byte, and n is the number of bytes transferred. The nodes network interface is assumed to be single ported; i.e. at most one message can be sent and one message can be received simultaneously. M is the number of elements in residuals or gradients of this layer. D is density, which represents a ratio of the size of communication-set to the size of dense gradients. In the case of reduction operations, we assume that γ_2 is the computation cost performing the reduction operation for a message of size M locally on any process, and γ_1 is the cost to decompress the collected sparse message M. For the case where the density distribution is uneven after using the binary search method, D represents the average density of all nodes.

Suppose that we use recursive doubling for allgather and Rabenseifners algorithm mentioned in Thakur et al. (2005) for allreduce communication. The cost of quantized sparse and dense synchronization is illustrated Equation 1 and 2, respectively. The derivations are left in Appendix B.

$$T_{sparse} = T_{select} + lg(p)\alpha + (p-1)(MD)\beta + p\gamma_1 \tag{1}$$

$$T_{dense} = 2lg(p)\alpha + 2\frac{p-1}{p}M\beta + \frac{p-1}{p}\gamma_2 \tag{2}$$

Some interesting conclusions could be drawn by comparing the above two equations. First, **the compression rate for the model is not equal to the compression rate for communication bandwidth.** Considering the term in front of β, even if the sparseness D is 0.1% for all p node. When p is 128, the communication bandwidth for sparse synchronization will be 12.8% of dense synchronization rather than 0.1% of dense synchronization. Second, the overhead to do reduction may be a new bottleneck when scaling RedSync to a large number of GPU nodes. The last term $p\gamma_1$ in Equation 1 indicating the overhead to do reduction increase linearly with the number of computing nodes. However, in Equation 2, reduction overhead almost does not increase with parallel scale.

According to our performance model, RedSync follows the following rules to identify the communication-set. Take a network configuration with a device-to-device bandwidth of 3.5 GB/s as an example. When the parameter size is less than 128KB, we use allreduce operation to do synchronization, because only the compression overhead is greater than the original communication overhead. When the parameter size is larger than 128KB and less than 4MB, we use *trimmed top-k selection* to identify communication-set. Although its performance is slightly worse than the binary-search-based method, it can guarantee the length of compressed residual on all nodes be the same, thus reducing the overhead of data transmission at large scale. When the parameter size is larger 4MB, we use sampled threshold binary search selection with an interval of 5.

5.6 OVERLAPPING COMMUNICATION AND COMPUTATION

It is necessary to improve system efficiency by overlapping communication with computation. In original SGD data parallelism, each node computes partial weight gradients for its mini-batch in the back-propagation step in each layer and aggregates these partial gradients across all nodes using an allreduce operation. These aggregated weight gradients are used to update the weights and only required right before the forward propagation step for that layer in the next iteration. As a result, the communication of allreduce operation for this layer can overlap with back propagation calculation of previous layers. Before updating aggregated gradients to weights, gradient clipping is usually adopted to avoid gradient explosion. It rescales all of the gradients when the sum of their norms exceeds a threshold. For gradient residual compression methods, because no explicit aggregated gradients can be achieved to do clipping, a technique called local clipping (Lin et al. (2017)) is adopted. It performs gradient clipping by a new threshold ($N^{-1/2}$ of original) locally before adding the current gradient to previous residual. In this case, we need to wait for the completion of the entire backpropagation to get gradients of all layers. Consequently, it is impossible to hide communication with back prorogation computation in each layer.

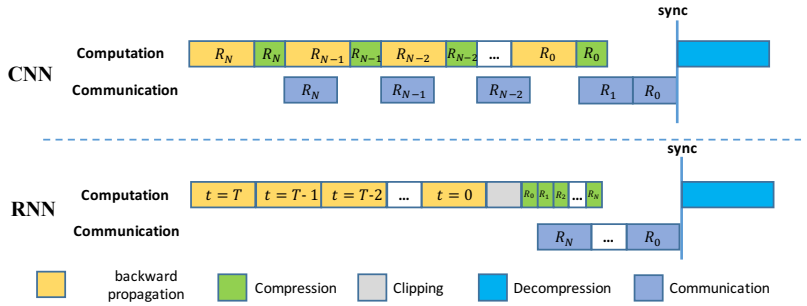

Figure 4: Two different schemes to overlap communication with computation for CNNs and RNNs.

As shown in Figure 4, in the RedSync, we adopt two different communication schemes for CNNs and RNNs. We have abandoned gradient clipping for CNNs, which seldom have gradient exploration problem for the deep networks. We initialize communication for this layer's transmitting residuals after compression finished. The backpropagation calculation of this layer can overlap with the communication of the next layer. As for RNNs, gradients are achieved after backpropagation of all time steps using Back Propagation Through Time (BPTT). When backpropagation of the last layer is completed, we use the gradients of all layers to conduct local gradient clipping. In this case, the communication time can only overlap with the compression calculation. Because even with the original data parallel approach, the computation and communication overlap for each layer can only be made at the last time step. The calculation of each time step is small and the parameters are large, so the original data parallelism cannot fully hide the communication.

5.7 Correctness for Momentum SGD and Warm-up Training

We integrate the momentum masking and momentum correction schemes as proposed in Lin et al. (2017) for momentum SGD and Nesterov momentum SGD optimizers in RedSync. The final version of RGC method adopted by RedSync is illustrated in Appendix A. A warm-up training, by exponentially decreasing the density of the residual in communication-set in first few epochs, is generally adopted to accelerate convergence in the first few iterations. For example, it is recommended to decrease the density of residual in the warm-up period as follows: 25%, 6.25%, 1.5625%, 0.4%, 0.1%. However, we find it could be inefficient for large-scale. As analyzed in the previous section, even synchronization of compressed residual with a density as 1.5625% requires 100% bandwidth of dense allreduce for quantized RedSync on 64 GPUs. Instead of adopting high-density RGC method of warm-up training, we use original SGD optimizer synchronized by allreduce in first few epochs if necessary.

6 Experimental Results

6.1 System and Software Setups

We test our implementation on two different multi-GPU systems, including a world's top GPU supercomputer and multi-GPU server.

Muradin : This a server with eight GPUs in the same node. It is equipped with a Intel(R) Xeon(R) CPU E5-2640 v4. Eight TITAN Vs are connect to CPU through PCI-E 3.0.

Piz Daint : It is located at Swiss National Supercomputing Centre. Currently, it ranks 6th on June 2018's Top500 list. The LINPACK performance is 19.6 petaflops. Each node includes two Intel Xeon E5-2690v3 CPUs (2.6GHz, 64GB RAM, 24 cores) and one NVIDIA Tesla P100 GPUs (16GB). In total, there are 5320 nodes connected by Aries interconnect with Dragonfly topology.

We use pytorch v4.0 [2] to conduct basic DNN training operations. For communication library, horovod [3] an MPI wrapper upon pytorch, is used to provide allgather and allreduce operations. The CUDA version is 9.1 on Muradin and 8.0 on Piz Daint. Horovod is compiled with OpenMPI v3.1 [4] with cuda-aware supported on both systems. For muradin, horovod is supported by NCCL v2.1 [5] for collective communication.

In Figure 5, we show device-memory-to-device-memory communication bandwidth among GPU nodes on above two multiple GPU systems. Communication bandwidth is obtained by measuring allreduce operations for data distributed inside the GPU's device memory. It is calculated from $\frac{S}{t} \times (2\frac{n-1}{n})$, where S is data size on each node, n is the number of node, t is measured allreduce time. One hundred times asynchronous allreduce operations are performed to obtain an average allreduce time. On Piz Daint, allreduce bandwidth reaches its peak around 1.5 GB/s and is less affected by the number of GPUs. For 8 GPUs on Muradin, the peak allreduce bandwidth is around 3.5 GB/s.

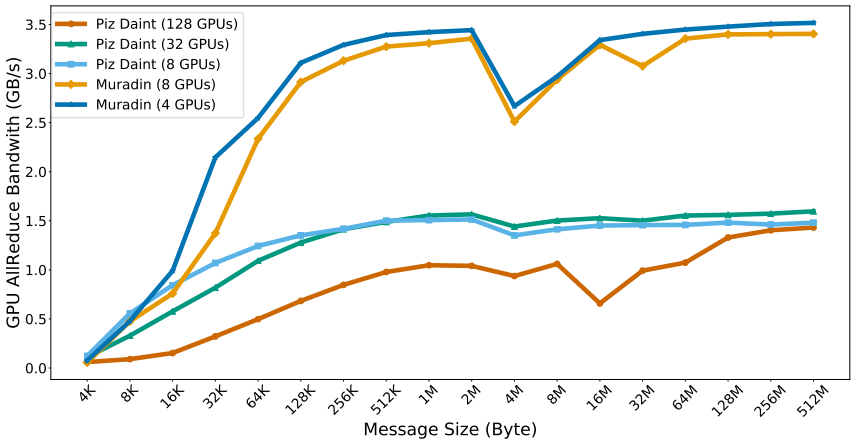

Figure 5: Communication bandwidth between GPU device memory.

6.2 Network structures and Datasets

We tested our performance on two major types of mainstream deep learning applications.

Image Classification : We studied ResNet-44 and VGG16 on Cifar10, AlexNet, VGG16 and ResNet-50 on ImageNet. Cifar10 consists of 50,000 training images and 10,000 validation images in 10 classes Krizhevsky & Hinton (2009), while ImageNet contains over 1 million training images and 50,000 validation images in 1000 classes Deng et al. (2009).

Language Modeling We picked two datasets for evaluation. The Penn Treebank corpus (PTB) dataset consists of 923,000 training, 73,000 validation and 82,000 test words (Marcus et al. (1993)). The WikiText language modeling dataset is a collection of over 100 million tokens extracted from the set of verified Good and Featured articles on Wikipedia (Merity et al. (2016)). It consists 2,088,628 training, 217,646 and 245,569 test words. We adopt a 2-layer LSTM language model architecture with 1500 hidden units per layer (Press & Wolf (2016)) to evaluate both datasets. We do not tie the weights of encoder and decoder and use vanilla SGD with gradient clipping. Learning rate decays when no improvement has been made in validation loss.

6.3 Evaluation of Accuracy

We examine the convergence of RedSycn on three different datasets mentioned before. The compression density for all layers is set as 0.01% when adopting *trimmed top-k* as selection method, and

[2] https://pytorch.org/

[3] https://github.com/uber/horovod

[4] https://www.open-mpi.org/software/ompi/v3.1/

[5] https://developer.nvidia.com/nccl

set between 0.01% and 0.01% for *threshold-based binary search*. For the Cifar10 dataset, we use two CNNs, i.e. ResNet44 and VGG16, as test cases. Both DNNs are tested on 4 GPUs, and the total mini-batch size is 256. On the ImageNet test set, we tested AlexNet, ResNet50, and VGG16. For all CNNs, we use Nesterov SGD with momentum as an optimizer. On the PTB dataset, we examine the perplexity of the 2-layer LSTM. We use the same hyperparameters as the SGD for the RGC method. A warm-up technique is applied to the first 5 epochs of ResNet50 and VGG16.

Figure 6 shows the error curve of SGD, RGC and quantized RGC on two types of CNNs and one LSTM. All of them shows similar convergence speedups for RGC and its quantization version. We tested the performance of the RCG method in the big batch case on Cifar10. As shown in Table 2, when increasing the batch size to 2K, RedSync got no loss of accuracy compared to the original SGD.

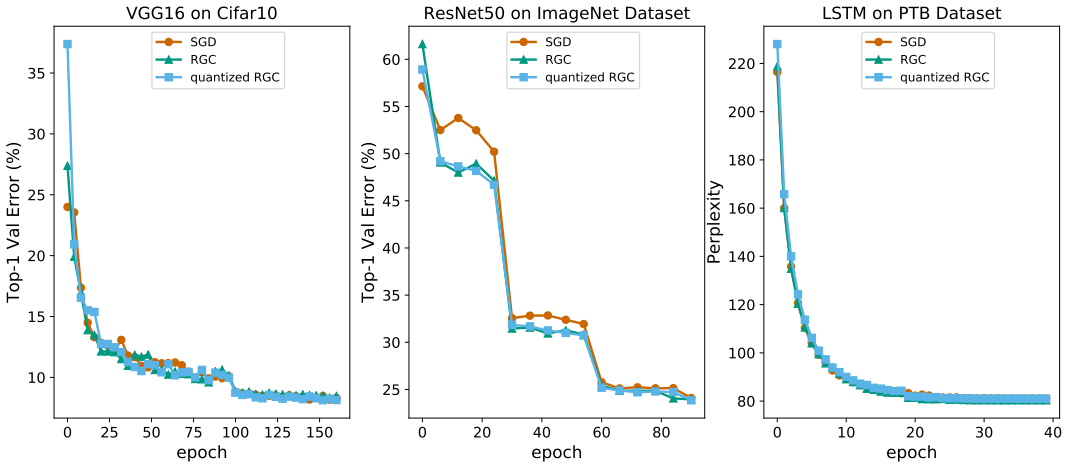

Figure 6: Left : top-1 validation accuracy vs number of epochs of training VGG16 on Cifar10 (4 GPUs, total batch size = 256). Center : top-1 validation accuracy vs number of epochs of training ResNet50 on ImageNet (8 GPUs, total batch size = 256). Right : Perplexity vs number of epochs of training LSTM on PTB (4 GPUs, total batch size = 20).

Table 1: Accuracy results for various networks. Model Size is the size of learnable parameters, as well as gradients and residuals. Compt. Amount shows Floating-Point Operations (Flops) required for a forward pass of a single data sample. Test errors of CNNs are measured as top-1 validation error and LSTM is measured as perplexity on validating dataset. Results on Cifar10 are measured using 4 nodes with batch-size as 64 for each node. Results on ImageNet are measured using 6 nodes with batch-size as 32 for each node. Results of LSTM are measured using 4 nodes with batch-size as 5 for each node.

		model size (MB)	Compt. Amount (GFlop)	SGD	RGC	RGC+ quant
Cifar10	ResNet44	2.65	0.20	7.48%	7.17%	7.87%
	VGG16	58.91	0.31	8.31%	8.45%	8.13%
ImageNet	AlexNet	233	0.72	44.73%	44.53%	44.30%
	ResNet50	103	8.22	24.07%	23.98%	23.85%
	VGG16	528	15.5	TBD	TBD	TBD
PTB	LSTM	264	2.52	81.07	81.01	80.69
Wiki2	LSTM	543	2.52	88.23	88.01	87.84

6.4 Evaluation of Performance

We compared the performance of RedSync with a baseline data parallel implementation provided by horovod. Performance of two version RedSync is illustrated here. *RGC* uses the original RGC methods, transmitting all of the values of compressed residual. *Quantized-RGC* combines quantization to compressed residual values and only transmit a quantized value. The speedup is calculated

Table 2: Test error of RCG and SGD methods under different batch sizes.

	Batch Size	128	256	512	1024	2048
ResNet44	SGD	7.09	7.48	8.18	**10.02**	16.8
	RGC	**6.40**	**7.17**	**7.471**	10.13	10.87
	quant RGC	7.06	7.87	7.62	11.86	**10.83**
VGG16	SGD	7.74	8.31	**9.06**	**9.49**	10.09
	thd RGC	**7.43**	8.45	9.31	9.90	11.12
	quant RGC	8.17	**8.13**	9.09	9.97	**9.81**

by comparing the average time of distributed training with the average time training on a single GPU node.

Figure 7 shows performance of RedSync on Piz Daint. RedSync is able to accelerate data parallel training on three networks, i.e. VGG16, AlexNet, and LSTM. Training VGG16 on ImageNet dataset using 128 GPUs, *RGC* and *Quantized-RGC* achieve 1.42× and 1.71× speedup compared with the baseline version. Training AlexNet on ImageNet dataset using 128 GPUs, Despite the slightly worse performance (0.94× of baseline version) of the *RGC* method, *Quantized-RGC* achieve 1.17× speedup compared with the baseline version. It is worth noting that RedSync achieves even greater performance gains on a smaller scale. For AlexNet, *Quantized-RGC* is 1.68×, 2.31× and 1.68 × faster than baseline version on 16, 32, 64 GPU nodes, respectively. Training 2-layer LSTM on PTB dataset using 32 GPUs, *RGC* and *Quantized-RGC* achieve 1.47× and 1.76× speedup compared with the baseline version. The acceleration of small scale is more obvious, *RGC* achieves 4.28×, 4.72× and 4.12× on 2, 4, 8 GPU nodes, respectively. For the above three DNNs, although performance of RedSync on a single GPU is not as good as baseline version due to compression and decompression overhead, RedSync can achieve acceleration with more than 2 GPUs, due to the greatly reduced synchronization traffic. However, there is no performance gain with RedSync for ResNet50. Although there is no obvious performance disadvantage when scaling it to 32 GPU scale, performance of *Quantized-RGC* is only 0.66 × of baseline version when using 128 GPU nodes. In ResNet50, the parameters of each layer of ResNet50 are relatively small and can easily be hidden by calculation.

When we scaled RedSync on Piz Daint, we found that the speedup curve is a concave curve shape. Such phenomenon verifies our communication cost model. Communication bandwidth requirement and decompression overhead both grow linearly with the number of GPU in use. Figure 10 illustrates time decomposition of different parts, during the process of scaling RedSync to 128 nodes on Piz Daint. *mask* is the time cost for Momentum Correction and Momentum Factor Masking. *pack* is the time cost to package indices and values of each layer's compressed residuals into a single message. *unpack* is the time cost to decompress collected messages and add to the weights of each layer. In term of ResNet50 training on ImageNet, on large scale, most of the time in RedSync is wasted on *unpack* part ,i.e. 69% and 67% for *RGC* and *Quantized-RGC* on 128 GPUs. ResNet50 is characterized by high computation to communication ratio. As shown in Table 1, the ratio of computation of a single sample to model size is 0.079 for ResNet50, 0.029 for VGG16 and 0.003 for AlexNet, 0.0095. Therefore, the traditional data parallel scheme can largely hide communication overhead with computation. Additionally, due to the small size of each layer, GPU memory bandwidth resources cannot be fully utilized when decompressing.

Quantized-RGC always achieves better performance than *RGC* for CNNs. For LSTM training on small scale, *Quantized-RGC* achieves worse performance than *RGC*. This is due to the balance of the selection operation and communication time. *Trimmed top-k* is adopted as the communication-set selection method for most of convolutional layers, since they have a relative small number of model parameter. Quantized *trimmed top-k* has similar computation cost as *trimmed top-k*, thus no extra overhead is introduced in *selection* part. The reduction in synchronization traffic by quantization can be directly transferred into an absolute increase in system performance. Parameter size of layers in LSTM is relatively larger, using *sampled threshold binary search* has better performance than using *threshold binary search* to identify communication-set. Because threshold sharing is not compatible with quantization, so we need to perform binary search process for the threshold at every iteration and introduce more selection cost. Therefore, on small-scale, *RGC* has better performance than *Quantized-RGC* due to less selection overhead. When scaling to more than 16 GPUs, benefit from the reduction of transmission traffic compensates for the time consuming of the selection algorithm.

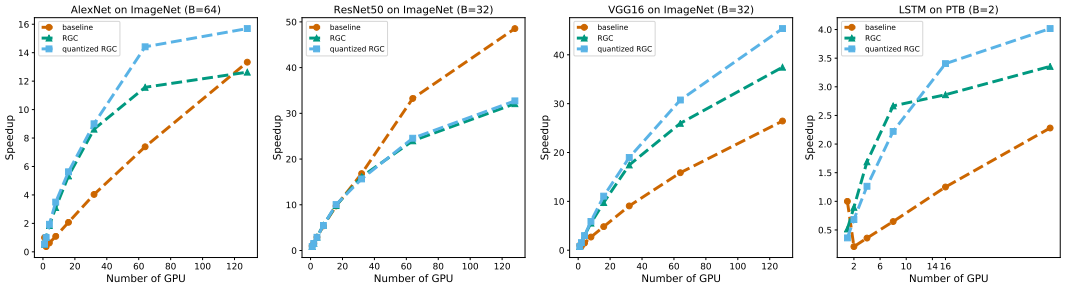

Figure 7: Scalability of RedSync for CNNs with ImageNet and LSTM with PTB on Piz Daint.

Figure 8 and Figure 9 show performance of RedSync on Muradin. Muradin is a single server node having higher bandwidth network connection. We still observe performance improvement for data parallel training of VGG16, AlexNet, and 2-layer LSTM. Training VGG16 on ImageNet dataset, *RGC* and *Quantized-RGC* achieve 1.55× and 1.64× speedup compared with baseline version using 8 GPUs. Training AlexNet on ImageNet dataset, *RGC* and *Quantized-RGC* achieve 1.96× and 2.26× speedup compared with baseline version using 8 GPUs. As on Piz Daint, RedSync has no performance improvement on ResNet on Muradin. Training ResNet on ImageNet dataset, *RGC* and *Quantized-RGC* achieve 0.83× and 0.85× speedup compared with baseline version using 8 GPUs. RedSync also works for communication-intensive CNNs on small datasets. Training VGG16 on Cifar10 dataset, *RGC* and *Quantized-RGC* achieve 1.16× and 1.24× speedup compared with baseline version using 8 GPUs. RedSync is suitable for training process of LSTM, featuring high communication-to-computation ratio. Training 2-layer LSTM on PTB dataset, *RGC* and *Quantized-RGC* achieve 2.11× and 2.06× speedup compared with baseline version using 8 GPUs. Training 2-layer LSTM on Wiki2 dataset, *RGC* and *Quantized-RGC* achieve 2.11× and 2.06× speedup compared with baseline version using 8 GPUs.

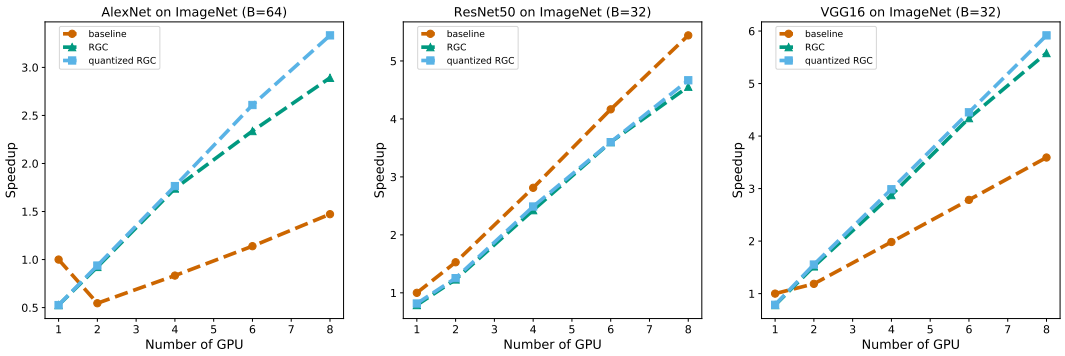

Figure 8: Scalability of RedSync for CNNs training on ImageNet using Muradin.

7 Conclusion

In this paper, we proposed a distributed implementation called RedSync to accelerate data parallel DNN training by utilizing Residual Gradient Compression (RGC). We solved two major obstacles to the application of RGC in multi-GPU systems : high overhead of compression and lack of support for collective communication implementation for sparse data structures. We design two parallel-friendly compression methods, which are *trimmed top-k* and *threshold-based binary search*, to select and compress important elements to be transmitted on GPU. A synchronization scheme based on highly-optimized allgather operation is adopted to exchange compressed sparse data structures across different nodes. We tested the performance of RedSync on two GPU platforms connected through a high-quantity network, including a supercomputer system and a multi-GPU server. For AlexNet, VGG16, and LSTM, we observe significant speedup for large-scale DNN training. For ResNet with high computation to communication ratio, RedSync show no performance improvement. We analyze this situation and give a reasonable explanation.

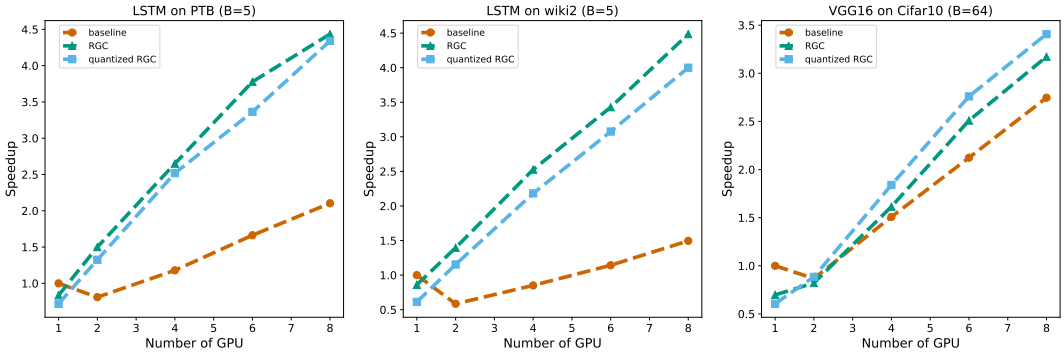

Figure 9: Scalability of RedSync for LSTM on PTB and Wiki2 datasets. Scalability of RedSync for LSTM VGG16 on Muradin.

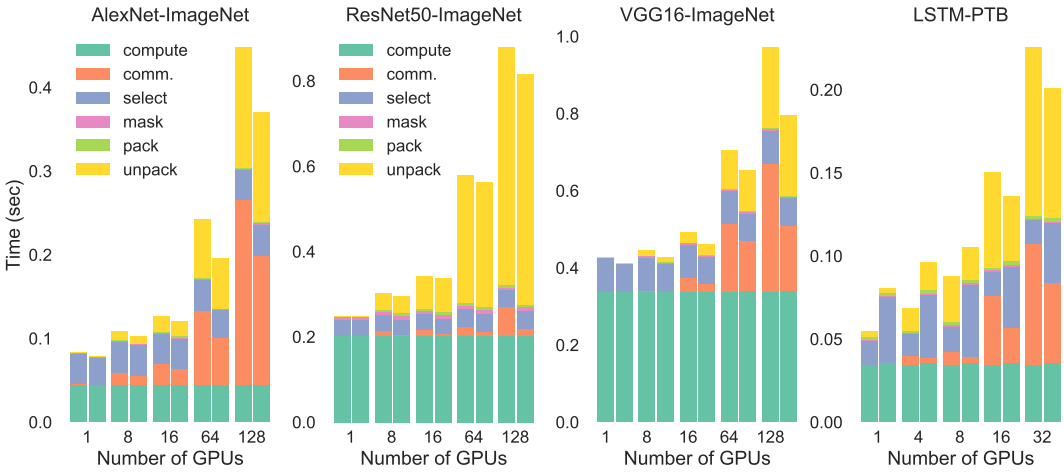

Figure 10: The proportion of time of each part of RedSync on Piz Daint. Time is the average time to measure 10 iterations. For each position, the left column illustrates time decomposition for RGC and right column illustrates time decomposition for quantized RGC.

REFERENCES

Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)*, 17:4–2, 2012.

Dan Alistarh, Demjan Grubic, Jerry Liu, Ryota Tomioka, and Milan Vojnovic. Communication-efficient stochastic gradient descent, with applications to neural networks. 2017.

Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 193–205. ACM, 2017.

Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. *arXiv preprint arXiv:1712.02679*, 2017.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pp. 1223–1231, 2012.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255. IEEE, 2009.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pp. 1223–1231, 2013.

Charles AR Hoare. Find (algorithm 65). *Communications of the ACM*, 4(7):321–322, 1961.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pp. 583–598, 2014.

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Ofir Press and Lior Wolf. Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859*, 2016.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

Felix Sattler, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Sparse binary compression: Towards distributed deep learning with minimal communication. *arXiv preprint arXiv:1805.08768*, 2018.

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.

Shubhabrata Sengupta, Aaron E Lefohn, and John D Owens. A work-efficient step-efficient prefix sum algorithm. In *Workshop on edge computing using new commodity architectures*, pp. 26–27, 2006.

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

Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19 (1):49–66, 2005.

Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in Neural Information Processing Systems*, pp. 1508–1518, 2017.

Yang You, Zhao Zhang, C Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. *CoRR, abs/1709.05011*, 2017.

A RGC ALGORITHM USED IN REDSYNC

We show the complete RGC method used in RedSync in Algorithm 4. For layers, whose parameter size are less than a *compress_thd*, we perform original data parallel SGD to update model parameters. For layers which are required to be compressed, besides vanilla SGD, the algorithm supports both momentum SGD and Nesterov momentum SGD as an optimizer. We overlap communication with computation using nonblocking collective operations. async_allgather and async_allreduce are able to progress independently of any computation or other communication. We need to designate a destination memory address to store the result after the allgather operation, while allreduce can be performed inline. The communication-set selection method is described in Algorithm 5. We use two parameter size thresholds (i.e., $thsd1$ and $thsd2$) to determine when to use the three selection methods. Selection method *topk trimmed_topk*, *threshold_binary_search_topk* return the indices of top-k elements according to residuals' absolute values and their values in original residuals. Selection method *topk_quant trimmed_topk_quant*, *threshold_binary_search_topk_quant* return the indices of top-k elements according to residuals' original values and their values in original residuals. We do not describe the situation of using gradient clipping. In that case, we need to put the compression part after the gradient synchronization.

B COST MODEL FOR SPARSE AND DENSE SYNCHRONIZATIONS

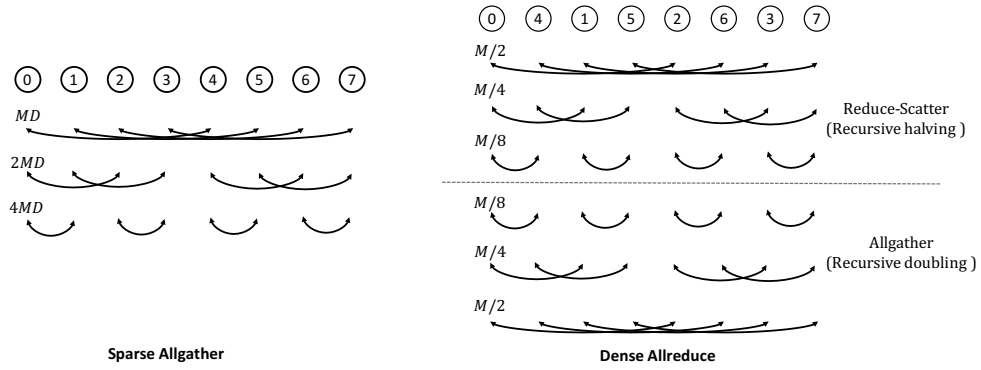

Figure 11: Communication pattern of sparse synchronization with allgather and dense synchronization with allreduce.

The left part of Figure 11 illustrates how sparse allgather works by recursive doubling method. We assume the density of compressed residual on all of the node is the same as D. If we use *threshold binary search* for communication-set selection, D here should be the average density of all nodes for a good approximation. In the first step, nodes that are a distance 1 apart exchange their compressed residual, the size of which is MD. In the second step, nodes that are a distance 2 apart exchange their own data as well as the data they received in the previous step, which is $2MD$ in total. In the third step, nodes that are a distance 4 apart exchange their own data as well the data they received in the previous two steps. In this way, for a power-of-two number of processes, all processes get all the data in lg$p$ steps. The amount of data exchanged by each node is MD in the first step, $2MD$ in the second step, and so forth, up to $2^{lg(p)-1}MD$ in the last step. Therefore, The time for message transfer taken by this algorithm is $T_{transfer} = lg(p)\alpha + (p-1)MD\beta$. After including decompressing overhead γ for collected p different compressed residuals and communication selection overhead T_{select}, the time for all-gather based synchronization should be $T_{transfer} = T_{select} + lg(p)\alpha + (p-1)MD\beta + p\gamma_1$

As shown in the right part of Figure 11, the Rabenseifners algorithm is adopted for allreduce operation on messages. It does a reduce-scatter followed by an allgather. Reduce-scatter is a variant of reduce in which the result, instead of being stored at the root, is scattered among all p nodes. rations, we use a recursive halving algorithm, which is analogous to the recursive doubling algorithm used for allgather. In the first step, each node exchanges data with a node that is a distance $p/2$ away: Each process sends the data needed by all processes in the other half, which is of size $M/2$. They also receives the data needed by all processes in its own half, and performs the reduction operation

Algorithm 4 Residual Gradient Compression on each node

Input: node id k, the number of node N
Input: dataset χ, mini batch size b per node
Input: initial learnable parameters $w = w[0], ..., w[\#layer]$
Input: D : density ratio , *compress_thd* : a threshold for the size of a parameter to determine whether to compress residuals.
Input: momentum parameter : *momentum*, weight decay parameter : *weight_decay*

$V^k \leftarrow 0, U^k \leftarrow 0$
for t = 0,1, ... max_iter **do**
  sample b elements as x_k from χ
  for $j = \#layer, \#layer - 1, ..., 0$ **do**
    obtain layer j's name as $name$
    calculate $G_j^k \leftarrow \nabla_j f(x_k; w)$ with forward and backward propagation on x_k
    if size(w_j) > *compress_thd* **then**
      if *weight_decay* not 0 **then**
        G_j^k+=*weight_decay* $\cdot\ w_j$
      end if
      if *momentum* != 0 **then**
        U_j^k = *momentum* $\cdot\ U_j^k$ + $\nabla_j f(x_k; w_j)$
        V_j^k = V_j^k + U_j^k
        if *use_nestrov* **then**
          V_j^k += $\nabla_j f(x_k; w)$
        end if
      else
        V_j^k += $\nabla_j f(x_k; w)$
      end if
      compress_idx, *compress_val* ← selection(V_j^k, D)
      $Masks \leftarrow 1$; $Masks[compress\_idx] \leftarrow 0$
      $V_j^k \leftarrow V_j^k \cdot Masks$
      $U_j^k \leftarrow U_j^k \cdot Masks$
      if *use_quantization* **then**
        handle[j] ← async_allgather(src = concate(len(*compress_idx*), *compress_idx*, mean(*compress_val*)), dst = *msg[name]*)
      else
        handle[j] ← async_allgather(src = concate(len(*compress_idx*), *compress_idx*, *compress_val*), dst = *msg[name]*)
      end if
    else
      handle[j] ← async_allreduce(G_j^k)
    end if
  end for
  for $j = \#layer, \#layer - 1, ..., 0$ **do**
    synchronize(handle[j])
    if size(w_j) > compress_thd **then**
      $G_j^k \leftarrow 0$
      offset ← 0
      for $j = 0, 1, ..., \#GPU$ **do**
        size ← $compressed\_msg$[offset]
        if *use_quantization* **then**
          $G_j^k[msg[name]$[offset:offset+size]] += $msg[name]$[offset+size]
          offset += size+1
        else
          $G_j^k[msg[name]$[offset:offset+size]] += $msg[name]$[offset+size:offset+2×size]
          offset += 2×size
        end if
      end for
      $w_j \leftarrow leaning\_rate \times G_j^k$
    else
      w_j ← SGD(G_j^k, w_j) #update weight with original SGD optimizer
    end if
  end for
end for

Algorithm 5 Communication-set Selection

Input: *thsd1*, *thsd2*
Input: V residuals to be compressed
Output: *compressed_idx*, *compressed_val*

$size \leftarrow$ size(V)
interval $\leftarrow 0$
flag $\leftarrow$ true
density D
if $size < thsd1$ **then**
  if use_quantization **then**
    if flag **then**
      compressed_idx, *compressed_val* $\leftarrow$ *topk_quant*(V, D)
    else
      compressed_idx, *compressed_val* $\leftarrow$ *lowk_quant*(V, D)
    end if
    flag = !flag
  else
    compressed_idx, *compressed_val* $\leftarrow$ *topk*($|V|$, D)
  end if
else if $size < thsd2$ **then**
  if use_quantization **then**
    if flag **then**
      compressed_idx, *compressed_val* $\leftarrow$ *trimmed_topk_quant*(V, D)
    else
      compressed_idx, *compressed_val* $\leftarrow$ *trimmed_lowk_quant*(V, D)
    end if
    flag = !flag
  else
    compressed_idx, *compressed_val* $\leftarrow$ *trimmed_topk*($|V|$, D)
  end if
else
  if use_quantization **then**
    if flag **then**
      compressed_idx, *compressed_val* $\leftarrow$ *threshold_binary_search_topk_quant*(V, D)
    else
      compressed_idx, *compressed_val* $\leftarrow$ *threshold_binary_search_lowk_quant*(V, D)
    end if
    flag = !flag
  else
    if interval $\%$ 5 == 0 **then**
      compressed_idx, *compressed_val*, threshold $\leftarrow$ *threshold_binary_search_topk*($|V|$, D)
      interval = 0
    else
      compressed_idx, *compressed_val* $\leftarrow$ *threshold_filter*($|V|$, threshold)
      interval++
    end if
  end if
end if

on the received data. In the second step, each process exchanges data with a process that is a distance $p/4$ away. This procedure continues recursively, halving the data communicated at each step, for a total of lg$p$ steps. Therefore, the communication time taken by this algorithm is $T_{transfer}$ = $2lg(p)\alpha + 2\frac{p-1}{p}M\beta$. Considering the time for reduction of intermediate data, the total time should be $T_{transfer}$ = $2lg(p)\alpha + 2\frac{p-1}{p}M\beta + \frac{p-1}{p}M\gamma_2$.