Accelerating Distributed Deep Learning Training with Gradient Compression

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

Cho-Jui Hsieh*
University of California, Los Angeles
chohsieh@cs.ucla.edu

ABSTRACT

Data parallelism has become a dominant method to scale Deep Neural Network (DNN) training across multiple nodes. Since synchronizing a large number of gradients of the local model can be a bottleneck for large-scale distributed training, compressing communication data 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 transmitting message size (0.1% of the gradient size) of each node and still achieve correct accuracy and the same convergence speed. However, the literature on compressing deep networks focuses almost exclusively on achieving good theoretical compression rate, while the efficiency of RGC in real distributed implementation has been less investigated. In this paper, we develop an RGC-based system that is able to reduce the end-to-end training time on real-world multi-GPU systems. Our proposed design called RedSync, which introduces a set of optimizations to reduce communication bandwidth requirement while introducing limited overhead. We evaluate the performance of RedSync on two different multiple GPU platforms, including 128 GPUs of a supercomputer and an 8-GPU server. Our test cases include image classification tasks on Cifar10 and ImageNet, and language modeling tasks on Penn Treebank and Wiki2 datasets. For DNNs featured with high communication to computation ratio, which have long been considered with poor scalability, RedSync brings significant performance improvements.

KEYWORDS

Deep Learning System, Data Parallel, Gradient Compression

1 INTRODUCTION

For training large-scale deep neural networks (DNNs) on multiple computing nodes, data parallelism has emerged as the most popular choice due to its simplicity and effectiveness [7, 19]. However, the limited communication bandwidth of the interconnected network has become the bottleneck limiting data parallel performance. First, models of DNNs, which already contain tens to hundreds of layers and totaling 10-20 million parameters today, continue to grow bigger [18]. Therefore, the requirement of fast synchronizing model parameter updates among all computing nodes poses a greater challenge. Second, the development of DNN-customized training accelerators has shifted the bottleneck of training from computing

hardware towards communication across nodes. Meanwhile, the evolution of the interconnected network is not as fast as computing hardware, and this trend still continues. As shown in Figure1, in the past decade, the most powerful GPU has been over 40 times faster, comparing the latest Tesla V100 GPU with G8800 GPU. However, the network bandwidth of switches in clusters is only 5 times faster, comparing InfiniBand HDR with InfiniBand QDR. Third, high-quality network fabric like InfiniBand is too expensive to be available for every data center. Publicly available cloud computing resources are still connected by the low-level network. For example, Amazon EC2 instances now provide a maximum bandwidth of 25 Gbps, far less than 96 Gbps of InfiniBand EDR 4-Link speed.
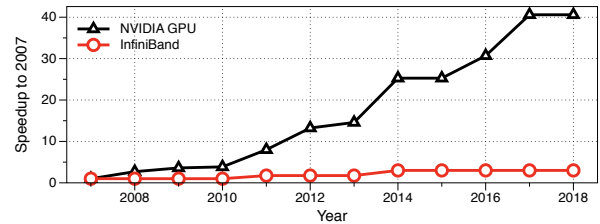

Figure 1: Evolution Speed of Infiniband and GPU.

Many recent studies focused on reducing the communication cost between nodes by reducing the size of the gradients to be transmitted. One line of work [22, 28] propose to quantize the gradients to low-precision values. Considering compression ratio (ratio of compressed gradients size to their original size) achieved by quantization is limited, another line of research orthogonal to quantization is to sparsify communication gradients and restrict weight-updates to a small subset of parameters. Residual Gradient Compression (RGC) method [1, 6, 13, 21, 26] is currently the most promising sparsification method to achieve good compression ratio while ensuring no loss of training accuracy. It transmits only a small subset of gradients and maintains the remaining gradients locally as residuals to be added to gradients of the next iteration. The first RGC implementation is proposed by Strom[26] and uses a threshold-based method to only send gradients larger than a predefined constant threshold for fully-connected layers. Considering a predefined threshold is hard to be chosen appropriately, Aji et. al.[1] improve the robustness of RGC by selecting top 1% gradients to communicate according to their magnitude. Because these two implementations are tuned for some specific network structures, applying them to other DNNs will lead to accuracy loss as indicated in AdaComp[6]. Based on their work, after introducing some key modifications, the latest RGC variant called DGC[13] is able to

achieve a 0.1% compression ratio on local gradients while ensuring almost no loss of model accuracy on a variety of DNN structures.

Despite good model accuracy achieved with simulation experiments, no recent studies have discussed the potential performance gain after integrating the latest RGC methods to a real distributed training system, especially to the multi-GPU systems equipped with high-quality network infrastructures. The challenges of applying RGC to distributed GPU systems come from two aspects. First, there is no efficient compression algorithm of RGC method designed for massive parallel processors such as GPU. According to our experimental results of Section 2.1, selecting top-0.1% elements with the state-of-the-art GPU-based top-k algorithm are so expensive that the overhead of compression is much higher than the benefits of network bandwidth reduction. Second, the synchronization scheme of sparse data structures generated by the RGC method has not been well studied. It is not easy to be supported with existing efficient communication libraries, such as Message Passing Interface (MPI), which are designed for dense data structures.

To increase scalability and efficiency of DNN training, we propose a systematic distributed design called RedSync (short for **Red**uction of **Sync**hronization Bandwidth), which combines RGC-based sparsification and quantization techniques together to compress transmitting gradient size of each node to its 0.1%. Our contributions are listed as follows:

- In terms of the algorithm level design, we propose a quantization technique called Alternating Signs Quantization (ASQ) to further compress the size of transmitting data sparsified by RGC to its half. In addition, we adapt the-state-of-the-art algorithmic improvements of RGC to a distributed situation. Applying the proposed algorithmic improvements, RGC+ASQ improves the efficiency of RGC-only in tasks of training a set of state-of-the-art DNN models with no accuracy loss.

- In terms of the system level design, we remove two main obstacles for efficient RGC deployment. A set of parallel-friendly top-0.1% selection algorithms are proposed to support the sparsification process. They are orders of magnitude faster than the state-of-the-art top-k selection method on GPU. Considering the distribution characteristics of communication data, we apply Allgather operation using MPI for a sparse synchronization scheme. A cost model is derived to analyze both communication cost and calculation overhead. Based on it, we pointed out potential performance gain and the bottleneck of current RGC method.

- This is the first work, as far as we know, to evaluate the performance of RGC method on supercomputer scale. On 128 GPUs, RedSync provides significant performance improvements for communication-intensive networks, like VGG, AlexNet, and some LSTMs.

2 DESIGN OF REDSYNC

Algorithm 1 presents the workflow used in RedSync. We denote a DNN model as $f(\mathbf{w})$, where $\mathbf{w}$ is the vector of parameters. We assume a system has N workers. Each worker, say the k-th worker, holds a local dataset χ_k^t at iteration t with size b and a local copy of the global weight $\mathbf{w}$. Synchronous SGD method is

---

Algorithm 1 RedSync Workflow

Input: node id k; the number of node N
Input: training dataset χ; mini batch size b per node
Input: initial model $\mathbf{w} = \mathbf{w}[0], \ldots, \mathbf{w}[\#layer]$; compression ratio D
  $V^k \leftarrow 0$
  for $t = 0, 1, \ldots max\_iter$ **do**
    sample b elements as χ_k^t
    $G^k \leftarrow \nabla f(\chi_k^t; \mathbf{w})$: forward and backward propagation
    for $j = \#layer, \#layer - 1, \ldots, 0$ **do**
      $V_j^k += G_j^k$
      Masks $\leftarrow$ select (V_j^k, D)
      $G_j^k \leftarrow$ SparseAllreduce(compress(quantize($V_j^k \odot$ Masks)))
      $V_j^k \leftarrow V_j^k \odot (1 - $Masks$)$
    end for
    $\mathbf{w} \leftarrow$ SGD($\mathbf{w}$, decompress(G^k))
  end for

---

adopted in RedSync. At each iteration, node k computes the gradient G^k using local data and we represent G_j^k as gradients of layer j. Each node also maintains a residual V^k, which is initialized as 0 and used to accumulate untransmitted gradient from previous iterations. After added with latest gradient, a subset of residuals is selected as the *communication-set*. The select operation in Algorithm 1 chooses important elements as communication-set based on magnitude. Masks is a 0/1 matrix, in which 1 indicating that the element at the corresponding position is selected as communication-set. After quantize operation, the quantized communication-set is compressed into sparse data structures for communication. Those selected elements are synchronized among all the nodes using SparseAllreduce operations. Synchronous SGD implemented with Allreduce, rather than Parameter Server[12], has been widely adopted in state-of-the-art large-scale DNN training tasks[9][29] on HPC platforms. Remaining elements outside the communication-set are assigned as new residuals of the next iteration. Figure 2 presents the design overview of RedSync according to the Algorithm 1. In the following, we details our contribution in design of select, quantize, SparseAllreduce and decompress operations to make this workflow work efficient in practice.
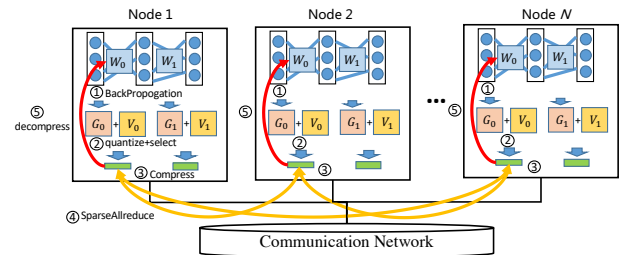

Figure 2: Overview of RedSync Design.

2.1 Parallel-Friendly Residual Sparsification

The efficiency of communication-set selection operation to sparsify the gradients is critical to the overall performance of the system.

There is still a lack of efficient implementation methods on parallel architectures such as GPUs, which is a major obstacle to the deployment of RGC methods. Recent RGC works [13, 21] suggest selecting top 0.1% elements from residuals of each layer as the communication-set. It is well-known that, by applying *Quickselect* algorithm [10], the time complexity of a top-k selection on a list of n elements using a single-core CPU is $O(n)$. However, the top-0.1% selection is not trivial to be implemented on massively parallel architectures, such as GPUs. One of the most efficient top-k selection methods designed for GPU can be implemented based on *radixSelect* algorithm [2], which determines each bit of the k-th largest element by scan and scatter. The *scan* [23] and scatter operations are not suitable for parallel processing and are extremely time-consuming. As shown in Figure 3, the computation time for top-0.1% with radixSelect on a Titan X GPU sometimes is even slightly higher than the time for synchronizing these parameters through a 28 Gbps network. To overcome the problem of slow radixSelect operation on GPU, we propose two communication-set selection algorithms called *trimmed top-k selection* and *threshold binary search selection*, which are more efficient on GPUs.

Trimmed top-0.1% selection. We notice that the distribution of residuals is usually similar to a normal distribution, we can use statistical features to remove most of the smaller elements and apply radixSelect operation on a relatively small subset. 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 value, for example, $0.8 \times (max - mean) + mean$. Operation count_nonzero gets the number of elements whose absolute values are greater than the threshold. If the number is smaller than k (the number of top-0.1% elements), 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. Operation mean, max and count_nonzero can all be efficiently implemented with a single reduction operation. nonzero_indices is a typical *stream compaction* problem, which uses just one scan operation as its backbone [24].

Threshold binary search selection. For a network layer with a large number of parameters, even using radixSelect operation on a small number of gradient elements is still a very time consuming operation. In order to completely avoid using radixSelect operation on GPU, we propose a method to select approximate top-0.1% elements as communication-set. Instead of identifying the kth (top 0.1%th) largest element, we search for a threshold to make it between the kth to $2k$th largest element, and then select elements larger than the threshold as communication-set. In this case, at least 0.1% largest elements are included in the communication-set, so the convergence rate of the algorithm will not be affected. As shown in Algorithm 3, we use a binary search algorithm to find such a threshold. To avoid over-searching, the algorithm will automatically terminate when the difference between the left and right borders is less than the small value ϵ.

For layers with large sizes, such as the first fully-connected layer in VGG16 and softmax layer in LSTM, the time for count_nonzero operation is still not negligible. We further improve the efficiency of the selection algorithm by reducing the number of 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 of search is empirically set to 5, and the selection algorithm introduces only one nonzero_count overhead on average. Such a method is called sampled Threshold binary search selection.

Algorithm 2 Trimmed Top-0.1% Selection

Input: tensor to be compressed X
Input: number of elements remained k
Output: $< indices, values >$
1: $mean \leftarrow \text{mean}(\text{abs}(X)); max \leftarrow \text{max}(\text{abs}(X))$
2: $\epsilon \leftarrow 0.2; ratio \leftarrow (1 - \epsilon)$
3: $nnz = \text{count\_nonzero}(\text{abs}(X) > threshold)$
4: **while** $nnz < k$ **do**
5:    $threshold \leftarrow mean + ratio \times (max - mean)$
6:    $nnz = \text{count\_nonzero}(\text{abs}(X) > threshold)$
7:    $ratio = ratio - \epsilon$
8: **end while**
9: $indices \leftarrow \text{nonzero\_indices}(\text{abs}(X) > threshold))$
10: $values \leftarrow X[indice]$

Algorithm 3 Top-0.1% using Threshold Binary Search Selection

Input: tensor to be compressed X
Input: number of elements remained k
Input: Termination condition parameter ϵ
Output: $< indices, values >$
1: $mean \leftarrow \text{mean}(\text{abs}(X)); max \leftarrow \text{max}(\text{abs}(X))$
2: $l \leftarrow 0.0; r \leftarrow 1.0; threshold = 0.0$
3: **while** $r - l > \epsilon$ **do**
4:    $ratio = l + (r - l)/2$
5:    $threshold \leftarrow mean + ratio \times (max - mean)$
6:    $nnz = \text{count\_nonzero}(\text{abs}(X) > threshold)$
7:    **if** $nnz > k$ and $2k > nnz$ **then**
8:       break
9:    **else if** $nnz < k/2$ **then**
10:       $r = ratio$
11:    **else**
12:       $l = ratio$
13:    **end if**
14: **end while**
15: $indices \leftarrow \text{nonzero\_indices}(\text{abs}(X) > threshold))$
16: $values \leftarrow X[indices]$

In Figure 3, we compared the time cost of different selection approaches applied to data of different sizes. Test data is generated from a standard uniform distribution. Allreduce indicates the time cost to synchronize messages using the Allreduce operation, with a peak network bandwidth of 28 Gbps. Performance is measured as total time cost for 100 times independent operations. Compared with directly performing radixSelect, both proposed methods significantly reduce the selection operation time for data of large size. For top-0.1% selection on 64MB elements, trimmed top-0.1% and sampled threshold binary search selection are 38.13 and 16.17 $\times$

faster than radixSelect. In practice, a hybrid compression strategy is adopted: For smaller parameter sets such as biases and batch norm layers, we do not compress residuals or directly use radixSelect to select top-0.1% significant elements. Trimmed top-0.1% selection is suitable for parameters of middle size layers, like convolutional layers, because it can ensure the compression ratio to be exactly 0.1% and introduce no extra communication bandwidth requirements. Threshold binary search based selection is suitable for large size layers, like hidden layers and softmax layers in LSTMs, for which the compression cost is more critical to be optimized than the communication cost.
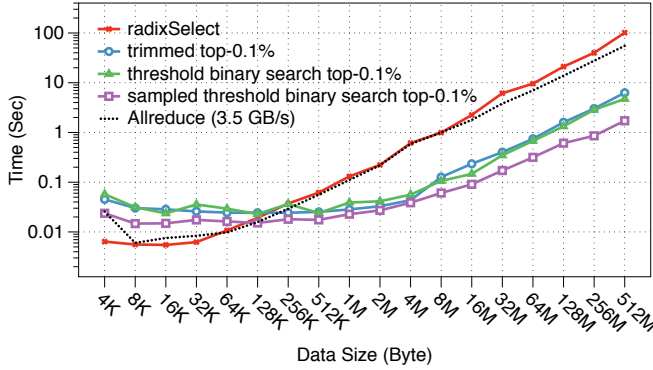

Figure 3: Performance of four communication-set selection algorithms vs Data sizes.

2.2 Quantization of Sparsified Residuals

The sparsified communication-set to be transmitted includes k index elements and k value elements. As indicated in Strom[26], by setting all value elements of the same sign to their average, the communication bandwidth requirement of value elements can be eliminated by transmitting only one average element instead of k elements. As there are both positive and negative elements in the communication-set, an extra data structure is used to record the sign of each element and extra efforts are required for quantization and de-quantization.

To eliminate this overhead, we design a quantization approach called *Alternating Signs Quantization (ASQ)* to further reduce 1/2 of the bandwidth requirement. In two adjacent training iterations, ASQ alternately quantizes the maximum 0.1% elements and the minimum 0.1% elements as communication-set instead of quantifying the maximum 0.1% elements with the largest absolute value. In other words, if we select the largest k elements (all positive numbers) of this layer as the communication-set at current iteration, we will choose smallest k elements (all negative numbers) as the communication-set for the next iteration. Since top-0.1% and bottom-0.1% elements are all of the same sign, no communication bandwidth requires to transmit extra sign information.

As shown in experimental results, the quantized RedSync using ASQ approach is able to guarantee no accuracy loss. As elements of residuals are designed to be delayed updated hundreds of steps, updating the elements of the same sign one more step later does not have signification impact on the direction of the gradient update.

ASQ introduces no quantization overhead and can be implemented efficiently by slightly modifying our parallel-friendly top-0.1% approaches. Compared with Strom's quantization method, ASQ approach is more memory-efficient and overhead-reduced. Their method uses an extra bitmap to record the sign of each value element, which is also required to be transmitted. Instead of once scan of entire data as ASQ, they have to separate positive and negative elements in the communication-set, and quantize them individually. In addition, it is worth noting that *sampled threshold binary search selection* cannot be used with quantization. We also do not quantify the output layer of the DNN, in order to distinguish the correct classification information.

2.3 Sparse Synchronization and Decompression

Synchronization of dense gradient structures in traditional distributed DNN systems can be simply implemented with an Allreduce operation, which has been well-studied on multiple-GPU systems[4]. However, the design of a sparse Allreduce in a distributed setting is not as simple because each worker may contribute different non-zero indices from its own communication-set. According to our observation, there are very few overlapping indices of the communication-set distribution of different nodes. For example, training VGG16 on Cifar10 dataset using 16 GPUs with a compression ratio as 0.1% for each node, the averaged compression ratio of synchronized residuals of all nodes is 1.55%. In this case, it is inefficient to use a sparse Allreduce [20, 30] for synchronization. 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 compressed message representing communication-set of each node should include the information of indices and values of elements in communication-set. When using threshold binary search selection, the length of each node's message is different. As a result, the packaged message should also include an initial element, which indicates the length of the compressed elements. 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.

After finishing the Allgather operation, each node collects N compressed communication-sets from all the other nodes. We add the compressed communication-sets to the corresponding weights in the local model after scaling with the 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 on GPU.

To analyze the potential performance gain of sparse synchronization, we adopt a performance model which is widely-used by [5, 16, 27] to analyze the cost in terms of latency and bandwidth used. 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. Generally, the node's 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 of the current layer. D is the compression ratio. If we use *threshold binary search* for communication-set selection, D here

should be the average compression ratio of all nodes. In the case of reduction operations, we assume that γ_2 is the computational cost for performing the reduction operation for a message of size M, and γ_1 is the cost to decompress the collected sparse message of size M.

Suppose that we use recursive doubling for Allgather and Rabenseifner's algorithm mentioned in [27] for Allreduce communication. The cost of quantized sparse and dense synchronization is illustrated Equation (1) and Equation (2), respectively. The derivations are as follows:

The left part of Figure 4 illustrates how sparse Allgather works by recursive doubling method. In the first step, nodes that are a distance 1 apart exchange their compressed data, the size of which is $M \times D$. In the second step, nodes that are a distance 2 apart exchange their own compressed data as well as the data they received in the previous step, which is $2M \times D$ 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, which is $4M \times D$ in total. 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 $M \times D$ in the first step, $2M \times D$ in the second step, and so forth, up to $2^{lg(p)-1}M \times D$ in the last step. Therefore, The time for message transfer taken by this algorithm is $T_{transfer} = lg(p)\alpha + (p-1)M \times D\beta$. After including decompressing overhead γ and communication-set selection overhead T_{select}, the time for all-gather based synchronization should be $T_{Allgather} = T_{select} + lg(p)\alpha + (p-1)M \times D\beta + p\gamma_1$.

As shown in the right part of Figure 4, the Rabenseifner's algorithm is adopted for Allreduce operation. It does a reduce-scatter followed by an Allgather. Reduce-scatter is a variant of reducing in which the results, instead of being stored at the root, are scattered among all p nodes. We use a recursive halving algorithm, which is analogous to the recursive doubling algorithm used for Allgather but in a reverse way. 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 receive the data needed by all processes in its own half and performs the reduction operation 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. After reduce-scatter, Allgather phase will have the the same bandwidth and latency requirements. Therefore, the communication time taken by this algorithm is $T_{transfer} = 2lg(p)\alpha + 2\frac{p-1}{p}M\beta$. The time taken by Allreduce is the sum of the times taken by reduce-scatter, Allgather and reduction operations. The total time should be $T_{dense\_Allreduce} = 2lg(p)\alpha + 2\frac{p-1}{p}M\beta + \frac{p-1}{p}\gamma_2$.

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

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

As implicated by the performance model, two important conclusions about the current RGC algorithm can be drawn. First, **the compression rate for the model is not equal to the compression rate for communication bandwidth.** The bandwidth term
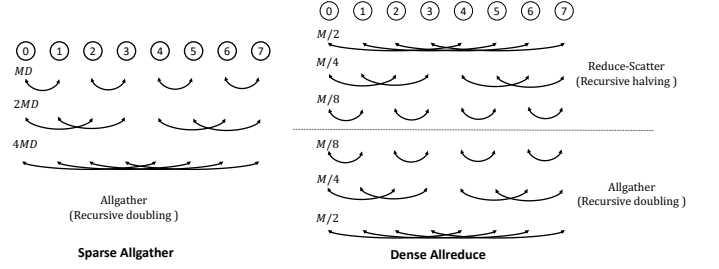

Figure 4: Communication pattern of sparse synchronization with Allgather and dense synchronization with Allreduce.

of sparse synchronization is $(p-1)DM\beta$, which is proportional to the number of nodes p. Even if the compression ratio 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%. Second, **the overhead of decompression rather than communication will be a bottleneck when scaling RGC method to larger scale.** The last term $p\gamma_1$ in Equation 1 indicates that the overhead of decompression in sparse Allreduce also increases linearly with the number of nodes p. However, in Equation 2, reduction overhead of dense Allreduce almost does not increase with number of nodes.

2.4 Overlapping Communication and Computation

It is necessary to improve data parallel efficiency by overlapping communication with computation through pipelining gradient Allreduce operations and backpropagation calculations. 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 RGC methods, the local clipping technique [13] is used to perform gradient clipping by a new threshold ($N^{-1/2}$ of the original one) locally before adding the current gradients to previous residuals. The difference is that traditional data parallel does clipping after communication of all layers is completed, while the RGC algorithm needs to do clipping before communication. In this case, we need to wait for the completion of the entire back-propagation to get gradients of all layers. And then we do clipping on gradients and then perform compression for communication. Local clipping introduces synchronization between computing and communication and thus eliminates the overlapping of communication and computation.

As shown in Figure 5, RedSync has abandoned gradient clipping for CNNs, which seldom have gradient exploration problem in order to explore the potential overlapping. As for RNNs, gradients are achieved after backpropagation of all time steps using Back Propagation Through Time (BPTT). When backpropagation of the last time step is completed, gradients of all layers are used to conduct local gradient clipping. In this case, the communication time can only overlap with the compression (selection) operation.

2.5 Other techniques

RedSync supports a set of algorithmic improvements proposed by DGC[13] to avoid convergence problem. For momentum SGD and

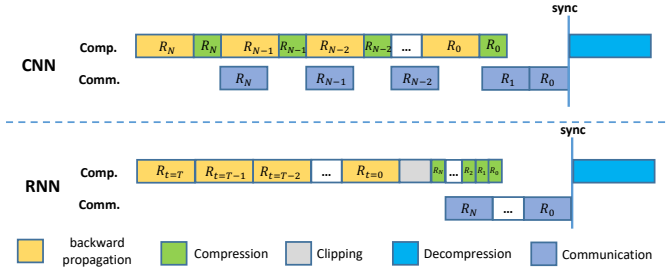Jiarui Fang, Haohuan Fu and Guangwen Yang and Cho-Jui Hsieh

Figure 5: CNNs and RNNs adopt different schemes to overlap communication with computation.

Nesterov momentum SGD optimizers, the momentum masking and momentum correction schemes of DGC are implemented by simply modifying workflow of Algorithm 1. A warm-up training, by exponentially decreasing the compression ratio in the first few epochs, is generally adopted to accelerate convergence. For example, it is recommended to decrease the compression ratio in the warm-up period as follows: 25%, 6.25%, 1.5625%, 0.4%, 0.1%. However, according to our performance model of communication, such an approach is inefficient on a large scale. For example, synchronization with a compression ratio as 1.5625% requires 100% bandwidth of dense Allreduce for quantized RedSync on 64 GPUs. Instead of adopting a high-compression-ratio RGC method of warm-up training, we use original SGD optimizer synchronized by dense Allreduce in first few epochs if necessary.

3 EXPERIMENTAL RESULTS

3.1 Setups

We tested the accuracy, speed of convergence and scalability of RedSync on two different multi-GPU systems, including a world's top GPU supercomputer (Piz Daint) and a multi-GPU server (Muradin).

Muradin is a server with eight GPUs in the same node. It is equipped with one Intel(R) Xeon(R) CPU E5-2640 v4 and 8 TITAN Vs, which is connected to the CPU through PCI-E 3.0.

Piz Daint is a GPU supercomputer. Each node of it includes two Intel Xeon E5-2690v3 CPUs and one NVIDIA Tesla P100 GPUs. In total, there are 5320 nodes connected by Aries interconnect with Dragonfly topology.

We used pytorch v0.4 to conduct DNN training on a single GPU. In terms of the communication library, an MPI wrapper upon pytorch called horovod [25], is used to provide collective communication operations. The CUDA version is 9.1 on Muradin and 8.0 on Piz Daint. Horovod was compiled with OpenMPI v3.1 with cuda-aware supported on both systems.

Two major types of mainstream deep learning applications are used in the experiments. For **Image Classification** tasks, we studied ResNet44 and VGG16 on Cifar10[11], AlexNet, VGG16 and ResNet-50 on ImageNet[8]. For all CNNs, we used Nesterov's momentum SGD as the optimizer. RGC methods used the same learning rate strategies as SGD. The warm-up technique was applied to the first 5 epochs of ResNet50 and VGG16 for both SGD and RGC. For **Language Modeling** tasks, we picked two datasets for evaluation.

The Penn Treebank corpus (PTB) dataset [14] consists of 923,000 training, 73,000 validation and 82,000 test words. 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 [15]. It consists 2,088,628 training, 217,646 and 245,569 test words. We adopted a 2-layer LSTM language model architecture with 1500 hidden units per layer [17] to evaluate both datasets. We tied 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. For RedSync, we used trimmed top-0.1% selection for convolutional layers larger than 128 KB and used threshold binary search top-0.1% selection for hidden layers and the softmax layer of LSTM.

3.2 Evaluation of Accuracy and Convergence Speed

Table 1: Accuracy Results for Various DNNs.

Dataset	DNN	Size	Gflops	Accuracy		
				SGD	RGC	RGC+ASQ
Cifar10	ResNet44	2.65	0.20	7.48%	**7.17%**	7.87%
	VGG16	59	0.31	8.31%	8.45%	**8.13%**
ImageNet	AlexNet	233	0.72	**44.73%**	44.91%	44.80%
	ResNet50	103	8.22	24.07%	23.98%	**23.85%**
	VGG16	528	15.5	29.5%	**29.1%**	29.3%
PTB	LSTM	204	2.52	75.86	75.14	**74.69**
Wiki2	LSTM	344	2.52	88.23	88.01	**87.84**

As shown in Table 1, we examined the accuracy of both RGC and our proposed quantization version RGC+ASQ on Muradin. The results of RGC and RGC+ASQ are compared with a classical data parallel implementation using SGD. Size indicates the model size in Megabyte. GFlop shows Giga Floating-Point Operations required for a forward pass using a single input sample. Accuracy of CNNs was measured as top-1 validation errors, and accuracy of LSTMs is measured as perplexity on validating dataset. Results on Cifar10 were achieved using 4 nodes (batch-size[1] = 64). Results on ImageNet were achieved using 6 nodes (batch-size = 32). Results of LSTM were achieved using 4 nodes (batch-size = 5).

As implicated by Table 1, the accuracy of the models obtained by the RGC and RGC+ASQ methods using RedSync is similar to that obtained by the SGD method using classical data parallel, with a difference of no more than 1%. In only one case (ImageNet-AlexNet), SGD achieves the best accuracy results, and in the other six cases, RGC and RGC+ASQ are even slightly better than SGD. In three cases (Cifar10-VGG16, ImageNet-ResNet50, Wiki2-LSTM), RGC+ASQ is slightly better than RGC, which indicates that the ASQ quantization method proposed in this chapter is very reliable and has no impact on training accuracy.

Figure 6 shows the speed of convergence of RGC and RGC+ASQ implemented with RedSync on three typical test cases compared with SGD implemented with the classical data parallel. The left figure shows top-1 validation accuracy vs the number of epochs of training VGG16 on Cifar10 (batch-size = 64). The center figure

---

[1]The batch-size here is for a single node.

Table 2: Accuracy of RGC and SGD Methods Under Different Batch Size.

	Batch Size	128	256	512	1024	2048
ResNet44	SGD	7.09%	7.48%	8.18%	**10.02%**	10.84%
	RGC	**6.40%**	**7.17%**	**7.47%**	10.13%	10.87%
	RGC+ASQ	7.06%	7.87%	7.62%	11.86%	**10.83%**
VGG16	SGD	7.74%	8.31%	**9.06%**	**9.49%**	10.09%
	RGC	**7.43%**	8.45%	9.31%	9.90%	11.12%
	RGC+ASQ	8.17%	**8.13%**	9.09%	9.97%	**9.81%**

shows top-1 validation accuracy vs the number of epochs of training ResNet50 on ImageNet (batch-size = 32). The right figure shows Perplexity vs the number of epochs of training LSTM on PTB (batch-size = 5).

As shown in Table 2, we also tested the sensitivity of the RGC and RGC+ASQ methods to large training data batch size. when increasing the batch size to 2048, RedSync got no loss of accuracy compared to the original SGD.

3.3 Evaluation of Scalability

To evaluate the scalability of RedSync on different scales, we compare RGC and RGC+ASQ implemented by RedSync with SGD implemented by the data parallel scheme provided by horovod. The performance was measured by averaging training time of 1000 iterations. Figure 7 shows the performance of RedSync on Muradin with six test cases. Figure 9 illustrates the scalability of RedSync on Piz Daint with four test cases. In order to find time-consuming parts, we also illustrate the cost of different parts in RedSync in Figure 8 when scaling it to 128 GPUs on Piz Daint. Our observations are summarized as follows.

Our proposed parallel-friendly selection algorithms for gradient sparsification are critical for improving the overall performance of RGC-based system. In Figure 7, we added a naive RGC implementation, which uses radixSelect to select top 0.1% elements as communication-set rather than our proposed methods. Since the compression time is too long, the performance of the naive RGC is even much slower than the original data parallel SGD version. After adopting our top-0.1% selection algorithms, the RGC and RGC+ASQ systems are now able to run faster than the SGD version.

Our proposed RGC+ASQ method is better than RGC-only method in most cases. RGC+ASQ always achieves better performance than RGC for CNNs. However, for LSTM training on a small scale, RGC+ASQ achieves worse performance than ASQ. The variance of communication and computational overhead accounts for such a phenomenon. CNN adopts trimmed top-0.1% as the communication-set selection method and its quantized version (RGC+ASQ) has a similar computation cost. As shown in Figure 8, no significant difference of *selection* cost between RCG and RGC+ASQ in CNN training. Therefore, the reducing of communication cost by ASQ improves the system's overall performance. As for selection algorithm of LSTMs, RGC uses sampled threshold binary search selection as communication-set selection, but RGC+ASQ uses threshold binary search. As we mentioned, the sampled selection is much faster. Therefore, on small-scale, RGC has better scalability than RGC+ASQ due to less selection overhead. When scaling to more

than 16 GPUs, benefit from the reduction of communication bandwidth compensates for the cost of the communication-set selection.

RedSync is suitable for training DNNs of high communication to computation ratio. In the past, these DNNs have been considered as not suitable for classical data parallel. As shown in the Figure 9, for VGG16, AlexNet, and LSTM, although the performance of RedSync using RGC and RGC+ASQ on a single GPU are not as good as the data parallel due to compression and decompression overhead, RedSync can achieve significant speedup on more than 2 GPUs. An exception is ResNet50. in most of the cases, RedSync brings no performance gain for it both on Piz Daint and Muradin. As implicated in Table 1, the ratio of computation to the communication of ResNet50 is the highest in the DNNs we investigated. On a large scale, most of the time during ResNet50 training with RedSync is wasted on the decompression phase, as shown in Figure 8, which overshadows the benefit of communication bandwidth reduction.

RedSync is suitable for the medium parallel scale. As shown in the right part of Figure 9, for the AlexNet network, RedSync is strong (brings the most significant performance improvement) on 4-16 nodes, bringing around 3x speedup. For the ResNet50 network, RedSync is strong on 4-16 nodes, bringing around 1.2x speedup. For the VGG16 network, RedSync is strong on 4-64 nodes, bringing around 2x speedup. For LSTM networks, RedSync is strong on 2-16 nodes, bringing around 3x speedup. Since both communication bandwidth requirements and decompression overhead increase linearly with the number of GPUs in use, RedSync is relatively weak on a larger scale. This phenomenon just confirms the communication performance model proposed in Section 2.3.

4 RELATED WORKS

As shown in Table 3, the related works to reduce communication cost in data parallel training of DNNs can be divided into two categories: quantification (Qunat.) and sparsification (Spars.). The compression ratio (Ratio in the table) is for a single node. Impl. in the table Indicates whether the method has been implemented in a real distributed environment.

Table 3: Related Works.

Name	Spars.	Quant.	1/Ratio	Sync.	Impl.	Scale
1-bit SGD[22]	×	√	32x	PS	√	40 GPU
QSGD[3]	×	√	4-6.8x	PS	√	16 GPU
Strom[26]	√	√	800x	PS	√	80 GPU
AdaComp[6]	√	×	40-200x	PS	×	-
TernGrad[28]	×	√	16x	PS	×	-
DGC[13]	√	×	1000x	AllRed.	×	-
SparCML[20]	√	√	256x	AllRed.	√	128 GPU

In terms of synchronization scheme (Sync.), DGC and SparCML use Allreduce and the other methods adopt Parameter Server (PS). Allreduce is more suitable for HPC platforms, while PS can hardly benefit from efficient Allreduce routines designed for HPC network hardware. For Allreduce systems, according to analysis in Section 2.3, it should be distinguished from the compression ratio of communication bandwidth. Similar for PS systems, the local compression ratio only reflects the bandwidth reduction of pushing gradients of
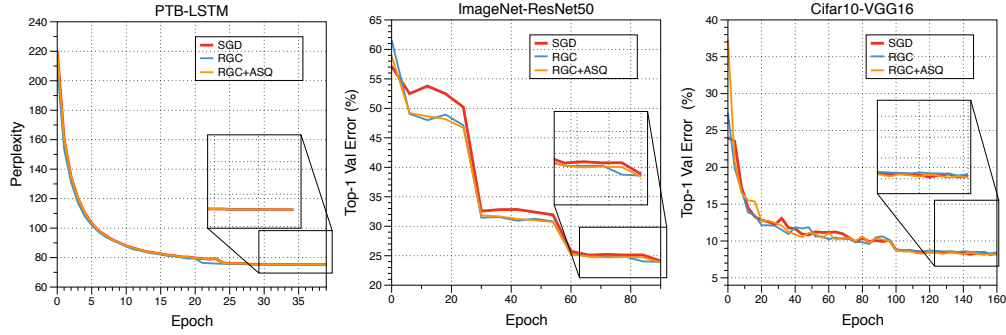
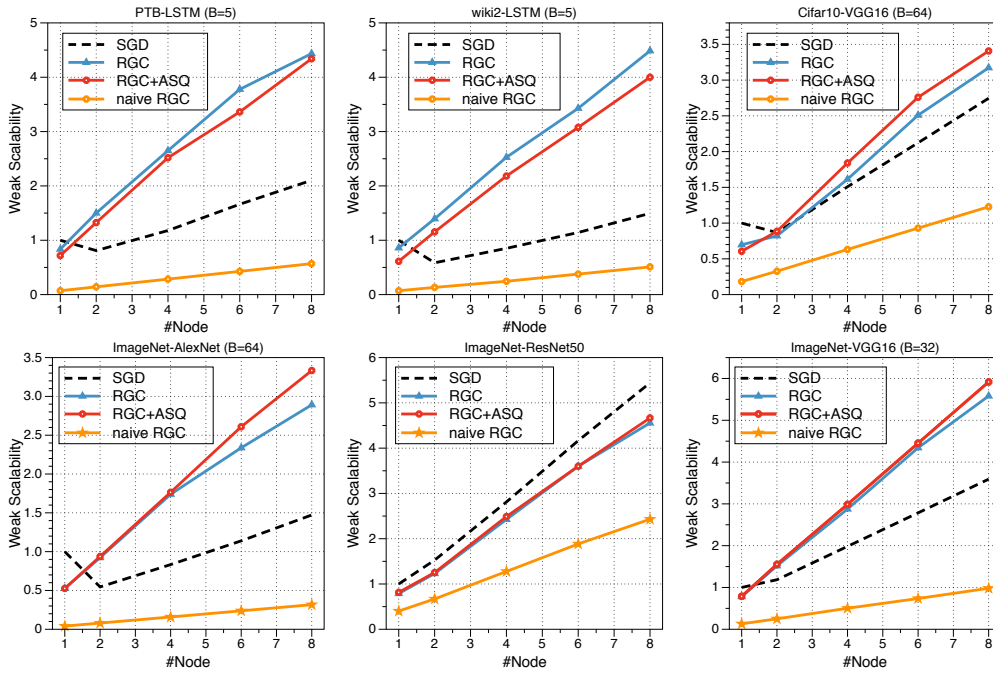Figure 6: Speed of Convergence using RedSync compared with classical data parallel implementation.

Figure 7: Scalability of RedSync for CNNs and RNNs training using Muradin.

works to the servers, while the bandwidth requirement of pulling gradients from the servers to works also increases linearly with the number of nodes. RedSync is the best in terms of scale compared with the other distributed implementations. SparCML also scales DNN training to 128 GPUs, but it adopts a "fast randomized top-k algorithm", which is not equal to the top-k selection. Its communication-set misses some important gradient elements and the system will suffer from convergence problem. Both Storm and SparCML combine quantization technique with sparsification, but in a less efficient way than our proposed ASQ method. As we mentioned, Storm's method is less efficient than ASQ, while Spar-CML quantizes the value elements of communication-set using 4-bit precision.

In terms of accuracy, DGC, TernGrad, and AdaComp presented comprehensive evaluation results on classical CNNs and RNNs. The other methods are either tested on some specified DNN models or have relative large accuracy loss compared with SGD. RedSync has been comprehensively evaluated using most of the test cases used in DGC. It is unfair to compare the absolute speedup achieved by each proposed method, because the achievable benefit is dependent on multiple factors of experimental settings, including the type of DNN model, the bandwidth of network hardware and the speed of computing hardware. The Aries interconnect of Piz Daint used in our experiments is the most high-quality one compared with others, which means RedSync is able to achieve more performance gain on low-quality network hardware.

5 CONCLUSION

This paper proposes an innovative data parallel DNN training design called RedSync that compressing transmitting data by gradient
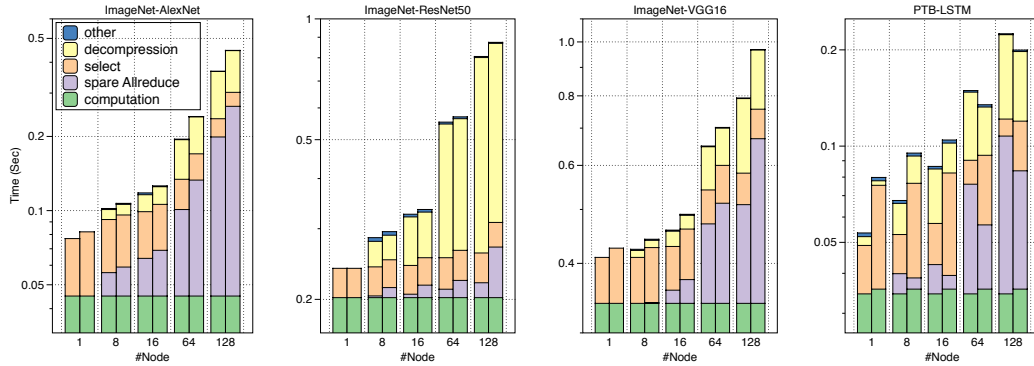
Figure 8: Decomposition of time cost of RedSync on Piz Daint. For each two-column group, the left one is for RGC+ASQ and the right one is for RGC. Y-axis is presented as 10 average iteration time.

sparsification and quantization. Residual Gradient Compression (RGC) method is used for sparsification. RedSync solved two major obstacles to implement RGC on multi-GPU systems: the high overhead of communication-set selection on GPU and lack of support for collective communication scheme for sparse data structures. Based on RGC, Alternating Signs Quantization (ASQ) method is first proposed, which further reduces half of the communication bandwidth requirement and introduces no extra overhead. The performance and accuracy of RedSync are evaluated on two typical GPU platforms, including a supercomputer system and a multi-GPU server. For AlexNet, VGG16, and LSTM, RedSync brings significant speedup for large-scale DNN training.

REFERENCES

[1] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).
[2] Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. 2012. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)* 17 (2012), 4–2.
[3] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*. 1709–1720.
[4] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. 2017. 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*. ACM, 193–205.
[5] Mike Barnett, Lance Shuler, Robert van De Geijn, Satya Gupta, David G Payne, and Jerrell Watts. 1994. Interprocessor collective communication library (InterCom). In *Proceedings of IEEE Scalable High Performance Computing Conference*. IEEE, 357–364.
[6] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2018. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *Thirty-Second AAAI Conference on Artificial Intelligence*.
[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.
[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 248–255.
[9] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: training imagenet in 1 hour. *arXiv:1706.02677* (2017).
[10] Charles AR Hoare. 1961. Find (algorithm 65). *Commun. ACM* 4, 7 (1961), 321–322.
[11] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. *Technical report, University of Toronto* (2009).

[12] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server.. In *OSDI*, Vol. 14. 583–598.
[13] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv preprint arXiv:1712.01887* (2017).
[14] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics* 19, 2 (1993), 313–330.
[15] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).
[16] Prasenjit Mitra, David Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. 1995. Fast collective communication libraries, please. In *Proceedings of the Intel Supercomputing UsersâĂŹ Group Meeting*, Vol. 1995.
[17] Ofir Press and Lior Wolf. 2016. Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859* (2016).
[18] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2018. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548* (2018).
[19] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*. 693–701.
[20] Cèdric Renggli, Dan Alistarh, and Torsten Hoefler. 2018. SparCML: High-Performance Sparse Communication for Machine Learning. *arXiv preprint arXiv:1802.08021* (2018).
[21] Felix Sattler, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. 2018. Sparse binary compression: Towards distributed deep learning with minimal communication. *arXiv preprint arXiv:1805.08768* (2018).
[22] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *15th Annual Conference of the International Speech Communication Association*.
[23] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. 2007. Scan primitives for GPU computing. In *Graphics hardware*, Vol. 2007. 97–106.
[24] Shubhabrata Sengupta, Aaron E Lefohn, and John D Owens. 2006. A work-efficient step-efficient prefix sum algorithm. In *Workshop on edge computing using new commodity architectures*. 26–27.
[25] A Sergeev and MD Balso. 2017. Meet Horovod: UberâĂŹs Open Source Distributed Deep Learning Framework for TensorFlow. *Uber Engineering Blog* (2017).
[26] Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*.
[27] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.
[28] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in Neural Information Processing Systems*. 1508–1518.
[29] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 1.
[30] Huasha Zhao and John Canny. 2014. Kylix: A sparse allreduce for commodity clusters. In *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE,
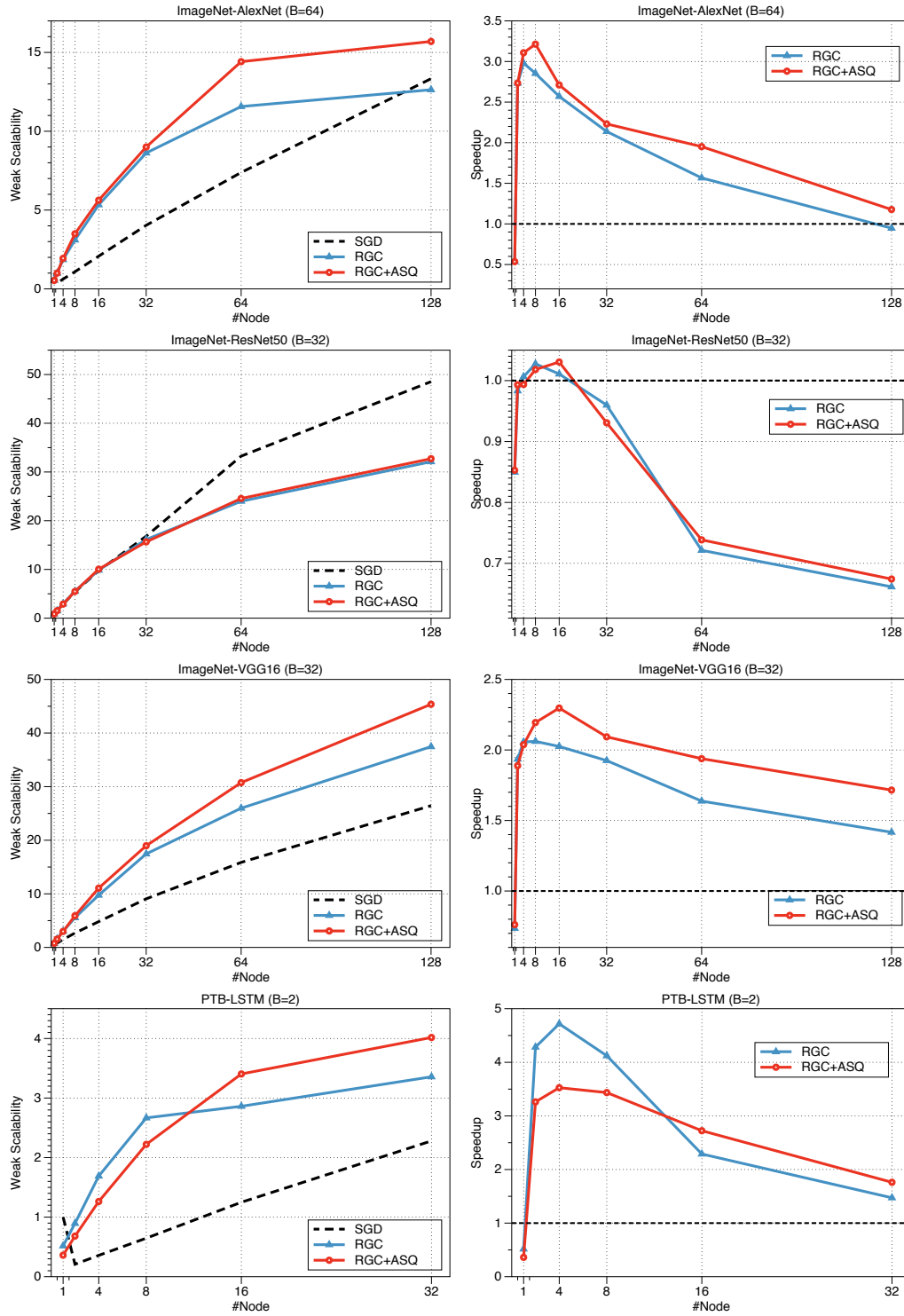
Figure 9: Left: Scalability of RedSync for four DNNs on Piz Daint. Right: Speedup of RedSync compared with a classical data parallel implementation.