Near-Optimal Sparse Allreduce for Distributed Deep Learning

Shigang Li
shigang.li@inf.ethz.ch
Department of Computer Science, ETH Zurich
Switzerland

Torsten Hoefler
htor@inf.ethz.ch
Department of Computer Science, ETH Zurich
Switzerland

Abstract

Communication overhead is one of the major obstacles to train large deep learning models at scale. Gradient sparsification is a promising technique to reduce the communication volume. However, it is very challenging to obtain real performance improvement because of (1) the difficulty of achieving an scalable and efficient sparse *allreduce* algorithm and (2) the sparsification overhead. This paper proposes Ok-Top$k$, a scheme for distributed training with sparse gradients. Ok-Top$k$ integrates a novel sparse allreduce algorithm (less than $6k$ communication volume which is asymptotically optimal) with the decentralized parallel Stochastic Gradient Descent (SGD) optimizer, and its convergence is proved. To reduce the sparsification overhead, Ok-Top$k$ efficiently selects the top-k gradient values according to an estimated threshold. Evaluations are conducted on the Piz Daint supercomputer with neural network models from different deep learning domains. Empirical results show that Ok-Top$k$ achieves similar model accuracy to dense allreduce. Compared with the optimized dense and the state-of-the-art sparse allreduces, Ok-Top$k$ is more scalable and significantly improves training throughput (e.g., 3.29x-12.95x improvement for BERT on 256 GPUs).

CCS Concepts: • **Theory of computation** → **Parallel algorithms**; • **Computing methodologies** → *Neural networks.*

Keywords: distributed deep learning, allreduce, gradient sparsification, data parallelism

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9204-4/22/04...$15.00
https://doi.org/10.1145/3503221.3508399

1 Introduction

Training deep learning models is a major workload on large-scale computing systems. While such training may be parallelized in many ways [7], the dominant and simplest form is data parallelism. In data-parallel training, the model is replicated across different compute nodes. After the computation of local gradient on each process is finished, the distributed gradients are accumulated across all processes, usually using an *allreduce* [12] operation. However, not all dimensions are equally important and the communication of the distributed gradients can be sparsified significantly, introducing up to 99.9% zero values without significant loss of accuracy. Only the nonzero values of the distributed gradients are accumulated across all processes. See [22] for an overview of gradient and other sparsification approaches in deep learning.

However, sparse reductions suffer from scalability issues. Specifically, the communication volume of the existing sparse reduction algorithms grows with the number of processes P. Taking the *allgather*-based sparse reduction [36, 41, 47] as an example, its communication volume is proportional to P, which eventually surpasses the dense *allreduce* as P increases. Other more complex algorithms [36] suffer from significant fill-in during the reduction, which also leads to a quick increase of the data volume as P grows, and may degrade to dense representations on the fly. For example, let us assume the model has 1 million weights and it is 99% sparse at each node—thus, each node contributes its 10,000 largest gradient values and their indexes to the calculation. Let us now assume that the computation is distributed across 128 data-parallel nodes and the reduction uses a dissemination algorithm [20, 28] with 7 stages. In stage one, each process communicates its 10,000 values to be summed up. Each process now enters the next stage with up to 20,000 values. Those again are summed up leading to up to 40,000 values in stage 3 (if the value indexes do not overlap). The number of values grows exponentially until the algorithm converges after 7 stages with 640,000 values (nearly dense!). Even with overlapping indexes, the fill-in will quickly diminish the benefits of gradient sparsity in practice and lead to large and suboptimal communication volumes [36].

We show how to solve or significantly alleviate the scalability issues for large allreduce operations, leading to an asymptotically optimal O(k) sparse reduction algorithm. Our intuitive and effective scheme, called Ok-Top$k$ is easy to implement and can be extended with several features to improve its performance: (1) explicit sparsity load balancing can distribute the communication and computation more evenly, leading to higher performance; (2) a shifted schedule and bucketing during the reduction phase avoids local hot-spots; and (3) an efficient selection scheme for top-k values avoids costly sorting of values leading to a significant speedup.

We implement Ok-Top$k$ in PyTorch [33] and compare it to four other sparse allreduce approaches. Specifically, Ok-Top$k$ enables:

- a novel sparse allreduce incurring less than $6k$ (asymptotically optimal) communication volume which is more scalable than the existing algorithms,
- a parallel SGD optimizer using the proposed sparse allreduce with high training speed and convergence guarantee,
- an efficient and accurate top-k values prediction by regarding the gradient values (along the time dimension) as a slowly changing stochastic process.

We study the parallel scalability and the convergence of different neural networks, including image classification (VGG-16 [44] on Cifar-10), speech recognition (LSTM [21] on AN4), and natural language processing (BERT [13] on Wikipedia), on the Piz Daint supercomputer with a Cray Aries HPC network. Compared with the state-of-the-art approaches, Ok-Top$k$ achieves the fastest time-to-solution (i.e., reaching the target accuracy/score using the shortest time for full training), and significantly improves training throughput (e.g., 3.29x-12.95x improvement for BERT on 256 GPUs). We expect speedups to be even bigger in cloud environments with commodity networks.

2 Background and Related Work

Mini-batch stochastic gradient descent (SGD) [9] is the mainstream method to train deep neural networks. Let b be the mini-batch size, w_t the neural network weights at iteration t, (x_i, y_i) a sample in a mini-batch, and ℓ a loss function. During training, it computes the loss in the forward pass for each sample as $\ell(w_t, x_i, y_i)$, and then a stochastic gradient in the backward pass as

$$G_t(w_t) = \frac{1}{b}\sum_{i=0}^{b} \nabla \ell(w_t, x_i, y_i).$$

The model is trained in iterations such that $w_{t+1} = w_t - \alpha G_t(w_t)$, where α is the learning rate.

To scale up the training process to parallel machines, *data parallelism* [18, 25, 26, 38, 52, 53] is the common method, in which the mini-batch is partitioned among P workers and each worker maintains a copy of the entire model. Gradient accumulation across P workers is often implemented using a standard dense *allreduce* [12], leading to about $2n$ communication volume where n is the number of gradient components (equal to the number of model parameters). However, recent deep learning models [10, 13, 34, 35] scale rapidly from millions to billions of parameters, and the proportionally increasing overhead of dense allreduce becomes the main bottleneck in data-parallel training.

Gradient sparsification [2, 4, 11, 16, 19, 36, 41–43, 50] is a key approach to lower the communication volume. By top-k selection, i.e., only selecting the largest (in terms of the absolute value) k of n components, the gradient becomes very sparse (commonly around 99%). Sparse gradients are accumulated across P workers using a sparse allreduce. Then, the accumulated sparse gradient is used in the Stochastic Gradient Descent (SGD) optimizer to update the model parameters, which is called Top$k$ SGD. The convergence of Top$k$ SGD has been theoretically and empirically proved [4, 36, 41]. However, the parallel scalablity of the existing sparse allreduce algorithms is limited, which makes it very difficult to obtain real performance improvement, especially on the machines (e.g., supercomputers) with high-performance interconnected networks [5, 17, 37, 40].

Table 1. Communication overhead of dense and sparse allreduces (n is the number of gradient components and $n \gg k$)

Algorithms	Bandwidth	Latency
Dense [12]	$2n\frac{P-1}{P}\beta$	$2(\log P)\alpha$
Top$k$A [36, 47]	$2k(P-1)\beta$	$(\log P)\alpha$
Top$k$DSA [36]	$[4k\frac{P-1}{P}\beta, (2k+n)\frac{P-1}{P}\beta]^1$	$(P + 2\log P)\alpha$
gTop$k$ [42]	$4k(\log P)\beta$	$2(\log P)\alpha$
Gaussian$k$ [41]	$2k(P-1)\beta$	$2(\log P)\alpha$
Ok-Top$k$	$[2k\frac{P-1}{P}\beta, 6k\frac{P-1}{P}\beta]^1$	$(2P + 2\log P)\alpha$

^{1} Intervals.

Table 1 summarizes the existing dense and sparse allreduce approaches. We assume all sparse approaches use the coordinate (*COO*) format to store the sparse gradient, which consumes $2k$ storage, i.e., k values plus k indexes. There are other sparse formats (see [22] for an overview), but format selection for a given sparsity is not the topic of this work. To model the communication overhead, we assume bidirectional and direct point-to-point communication between the compute nodes, and use the classic latency-bandwidth cost model. The cost of sending a message of size L is $\alpha + \beta L$, where α is the latency and β is the transfer time per word.

For **dense** allreduce, Rabenseifner's algorithm [12] reaches the lower bound [45] on the bandwidth term (i.e., about $2n$). **Top$k$A** represents the *A*llgather based approach [36, 42], in which each worker gathers the sparse gradients across P workers, and then the sparse reduction is conducted locally. Although Top$k$A is easy to realize and does not suffer from the fill-in problem, the bandwidth overhead of allgather is proportional to P [12, 45] and thus not scalable.

Top*k*DSA represents the *D*ynamic *S*parse *A*llreduce used in SparCML [36], which consists of reduce-scatter and allgather (motivated by Rabenseifner's algorithm) on the sparse gradients. In the best case of that the indexes of top-k values are fully overlapped across P workers and the top-k values are uniformly distributed in the gradient space, Top$k$DSA only incurs about $4k$ communication volume. However, the best case is almost never encountered in the real world of distributed training, and Top$k$DSA usually suffers from the fill-in problem and switches to a dense allgather before sparsity cannot bring benefit, which incurs about $2k+n$ communication volume. **gTop*k*** [42] implements the sparse allreduce using a reduction tree followed by a broadcast tree. To solve the fill-in problem, gTop$k$ hierarchically selects top-k values in each level of the reduction tree, which results in $4k \log P$ communication volume. **Gaussian*k*** [41] uses the same sparse allreduce algorithm as Top$k$A with a further optimization for top-k selection. For our **O*k*-Top*k***, the communication volume is bounded by $6k$. Although Ok-Top$k$ has a little higher latency term than the others, we target large-scale models and thus the bandwidth term dominates. Since the bandwidth term of Ok-Top$k$ is only related to k, our algorithm is more efficient and scalable than all others. See Section 5.4 for experimental results.

Gradient quantization [3, 8, 14, 23, 30, 48], which reduces the precision and uses a smaller number of bits to represent gradient values, is another technique to reduce the communication volume. Note that this method is orthogonal to gradient sparsification. A combination of sparsification and quantization is studied in SparCML [36].

Another issue is the sparsification overhead. Although gradient sparsification significantly reduces the local message size, top-k selection on many-core architectures, such as GPU, is not efficient. A native method is to sort all values and then select the top-k components. Asymptotically optimal comparison-based sorting algorithms, such as merge sort and heap sort, have $O(n \log n)$ complexity, but not friendly to GPU. Bitonic sort is friendly to GPU but requires $O(n \log^2 n)$ comparisons. The quickselect [29] based top-k selection has an average complexity of $O(n)$ but again not GPU-friendly. Bitonic top-k [39] is a GPU-friendly algorithm with complexity $O(n \log^2 k)$, but still not good enough for large k. To lower the overhead of sparsification, Gaussian$k$ [41] approximates the gradient values distribution to a Gaussian distribution with the same mean and standard deviation, and then estimates a threshold using the percent-point function and selects the values above the threshold. The top-k selection in Gaussian$k$ is GPU-friendly with complexity $O(n)$, but it usually underestimates the value of k because of the difference between Gaussian and the real distributions (see Section 3.1.3). Adjusting the threshold adaptively (e.g., lower the threshold for an underestimated k) [41] is difficult to be accurate. In Ok-Top$k$, we use a different method for the top-k selection. We observe that the distribution of gradient values changes slowly during training. Therefore, we periodically calculate the accurate threshold and reuse it in the following iterations within a period. Empirical results show that this threshold reuse strategy achieves both accuracy (see Section 5.2) and efficiency (see Section 5.4) when selecting local and global top-k values in Ok-Top$k$.

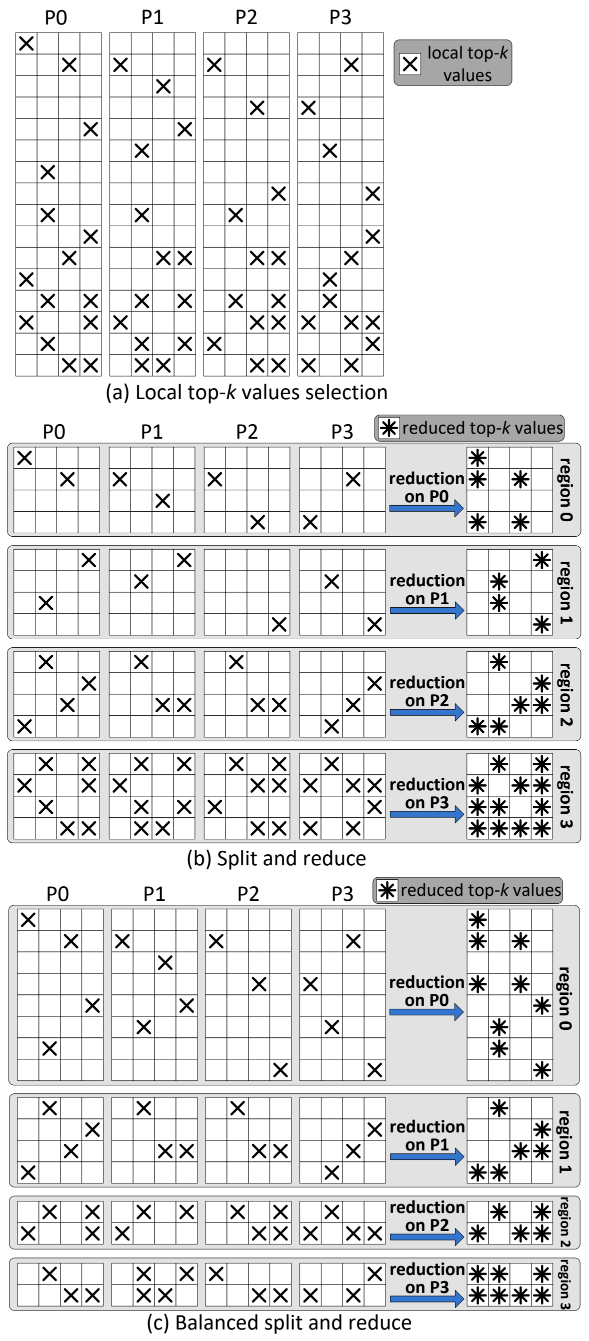


Figure 1. Gradient space split with balanced top-k values and reduction on subspace.

3 O(k) Sparse Allreduce

In this section, we will present the sparse allreduce algorithm of Ok-Top$k$, analyze the complexity using the aforementioned latency (α) - bandwidth (β) cost model, and prove its optimality. We use *COO* format to store the sparse gradient. Since the algorithm incurs less than $6k$ communication volume, we call it *O(k) sparse allreduce.*

3.1 Sparse allreduce algorithm

O(k) sparse allreduce mainly includes two phases: (1) *split and reduce*, and (2) *balance and allgatherv.* During the two phases, we propose to use an efficient top-k selection strategy to select (what we call) *local* top-k values and *global* top-k values, respectively. Specifically, the semantic of O(k) sparse allreduce is defined by Top$k(\sum_{i=0}^{P-1} \text{Top}k(G_t^i))$, where G_t^i is the sparse gradient on worker i at training iteration t, the inner Top$k$ operator is the local top-k selection, and the outer Top$k$ operator is the global top-k selection.

3.1.1 Split and reduce. Figure 1 presents the *split and reduce* phase. Suppose we have 4 workers and each worker has a 2D gradient of size 16x4. In Figure 1(a), each worker selects the local top-k values to sparsify the gradient. How to efficiently select the top-k values will be discussed in Section 3.1.3. Then, a straightforward *split and reduce* for the sparse gradients is presented in Figure 1(b), in which the 2D space of the sparse gradient is evenly partitioned into P regions and worker i is responsible for the reduction on region i. Each worker i receives sparse regions from the other workers and then conducts the reduction locally. However, this simple partitioning method may lead to severe load imbalance, since the top-k values may not be evenly distributed among the regions. In an extreme case, all local top-k values will be in region 0 of each worker, then worker 0 has to receive a total of $2(P-1)k$ elements (i.e., values and indexes) while the other workers receive zero elements.

Without loss of generality, we can make a more balanced partition (as shown in Figure 1(c)) based on our observations for deep learning tasks: the coordinates distribution of the local top-k values of the gradient is approximately consistent among the workers at the coarse-grained (e.g., region-wise) level, and changes slowly during training. To achieve a balanced partition, each worker calculates the local boundaries of the P regions by balancing the local top-k values. Then, a consensus is made among P workers by globally averaging the P-dimensional boundary vectors, which requires an allreduce with message size of P elements. The boundaries are recalculated after every τ iterations. We empirically set $\tau = 64$ to get a performance benefit from periodic space repartition as shown in Section 5.3. Note that the small overhead of allreduce (i.e., $(\log P)\alpha$) is amortized by the reuse in the following τ-1 iterations, resulting in only $(\log P)\alpha/\tau$ overhead per iteration. Therefore, the overhead of boundary recalculation can be ignored. After making a balanced split, each worker approximately receives $2k/P$ elements from any of the other $P-1$ workers. Therefore, the overhead is

$$C_{split\_and\_reduce} = (P-1)\alpha + 2k\frac{P-1}{P}\beta \tag{1}$$

We further make two optimizations for *split and reduce*, including destination rotation and bucketing. As shown in Figure 2(a), a native communication pattern is that all workers send data to worker i at step i, which may lead to endpoint congestion [49]. To avoid these hot-spots, we rotate the destinations of each worker as shown in Figure 2(b). Furthermore, to utilize the network parallelism, we bucketize the communications. The messages with in a bucket are sent out simultaneously using non-blocking point-to-point communication functions. Communications in the current bucket can be overlapped with the computation (i.e., local reduction) of the previous bucket.

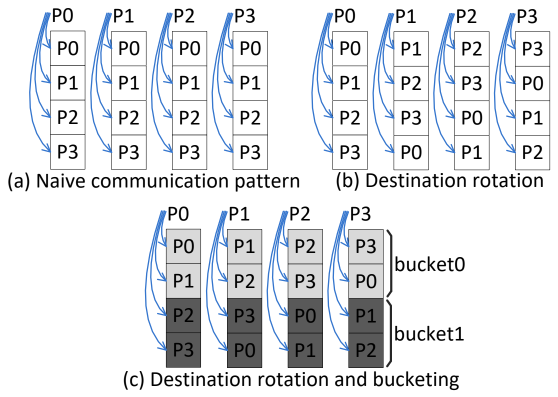

P0
P1
P2
P3
(a) Naive communication pattern
(b) Destination rotation
bucket0
bucket1
(c) Destination rotation and bucketing

Figure 2. Top-k values reduction with endpoint congestion avoidance and network parallelism exploitation.

3.1.2 Balance and allgatherv. Figure 3 presents the phase of *balance and allgatherv.* First, each worker selects the global top-k values from the reduced top-k values in the region that the worker is in charge of. Note that the global top-k selection only happens locally according to an estimated threshold (will be discussed in detail in Section 3.1.3). Next, each worker packages the selected global top-k values (and the corresponding indexes) into a consecutive buffer. Similar to the local top-k values, the global top-k values may also not be evenly distributed among the workers, causing load imbalance. In an extreme case, all global top-k values will be in one worker. The classic recursive doubling based allgatherv [45] would incur $2k \log P$ communication volume, namely, total $\log P$ steps with each step causing $2k$ traffic.

To bound the communication volume by $4k$, we conduct a data balancing step after packaging. Before balancing, an allgather is required to collect the consecutive buffer sizes from P workers, which only incurs $(\log P)\alpha$ overhead (the bandwidth term can be ignored). Then, each worker uses the collected buffer sizes to generate the communication scheme

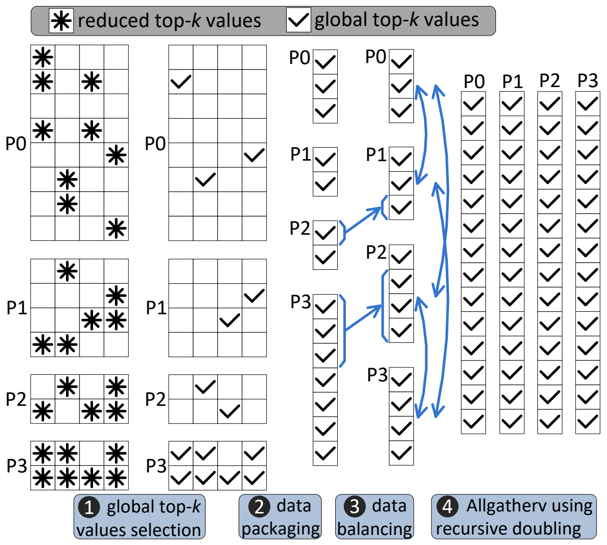


Figure 3. Data balancing and allgatherv for global top-k values.

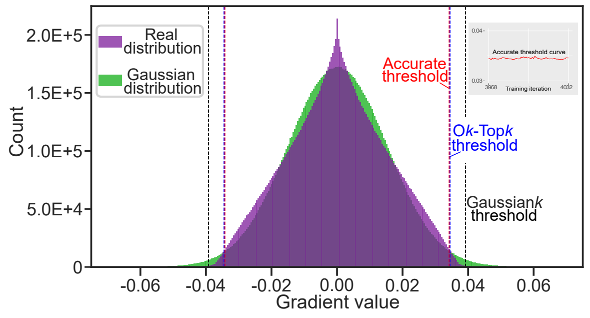
(i.e., which data chunk should be sent to which worker) for data balancing. We use point-to-point communication (blue arrows in the step of *data balancing*) to realize the scheme. The overhead of data balancing is bounded by the extreme case of all global top-k values locate in one worker, where data balancing costs $P\alpha + 2k\frac{P-1}{P}\beta$. Data balancing in any other case has less cost than the extreme case. At last, an allgatherv using recursive doubling on the balanced data has the overhead of $(\log P)\alpha + 2k\frac{P-1}{P}\beta$. Therefore, the overhead of *balance and allgatherv* is

$$C_{balance\_and\_allgatherv} \leq (P + 2\log P)\alpha + 4k\frac{P-1}{P}\beta. \quad (2)$$

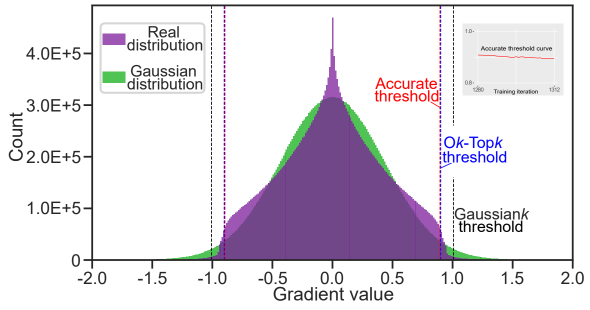
By adding the costs of the two phases, the total overhead of O(k) sparse allreduce is

$$C_{Ok\_sparse\_allreduce} \leq (2P + 2\log P)\alpha + 6k\frac{P-1}{P}\beta \quad (3)$$

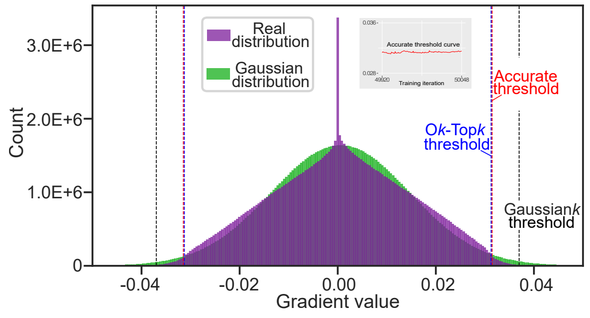
3.1.3 Efficient selection for top-k values. Ok-Top$k$ relies on estimated thresholds to approximately select the local and global top-k values. The key idea is to regard the gradient values (along the time dimension) as a slowly changing stochastic process $G(t)$. Specifically, the statistics (such as top-k thresholds) of $G(t), G(t+1), ...G(t+\tau'-1)$ change very slowly. Therefore, we only calculate the accurate thresholds for local and global top-k values after every τ' iterations, and then reuse the thresholds in the following τ'-1 iterations. The accurate threshold can be obtained by sorting the gradient values and returning the k-th largest value. Top-k selection according to a threshold only requires n comparisons and is quite efficient on GPU. The overhead of accurate threshold calculation is amortized by the reuse.



(a) VGG-16 [44] on Cifar-10 with density = 1.0%.



(b) LSTM [21] on AN4 [1] with density = 2.0%.



(c) BERT [13] on Wikipedia [13] with density = 1.0%.

Figure 4. Gradient value distribution and local top-k threshold predictions. Density is defined as k/n.

We validate our claim by the empirical results from different deep learning tasks presented in Figure 4. The gradient value distribution shown in Figure 4 is for a selected iteration where Ok-Top$k$ uses a threshold calculated more than 25 iterations ago. We can see the threshold of Ok-Top$k$ is still very close to the accurate threshold (see Section 5.2 for the accuracy verification for top-k selections of Ok-Top$k$ in the scenario of full training). On the contrary, Gaussian$k$ severely underestimates the value of k by predicting a larger threshold, especially after the first few training epochs. This is because as the training progresses, the gradient values are getting closer to zero. Gaussian distribution, with the same mean and standard deviation as the real distribution, usually has a longer tail than the real distribution (see Figure 4).

3.1.4 Pseudocode of O(k) sparse allreduce. We present the pseudocode of O(k) sparse allreduce in Algorithm 1. In

Algorithm 1 O(k) sparse allreduce

```
1: Inputs: stochastic gradient $G_t^i$ at worker $i$, iteration $t$ ($t$>0), value $k$, space repartition period $\tau$, thresholds re-evaluation period $\tau'$.
2: if ($t$-1)  mod $\tau'$ == 0 then
3:     $local\_th$ = $th\_re\_evaluate$($G_t^i$, $k$)
4: end if
5: if ($t$-1)  mod $\tau$ == 0 then
6:     $boundaries$ = $space\_repartition$($G_t^i$, $local\_th$)
7: end if
8: $reduced\_topk^i$, $local\_topk\_indexes$ = $split\_and\_reduce$($G_t^i$, $local\_th$, $boundaries$)
9: if ($t$-1)  mod $\tau'$ == 0 then
10:     $all\_reduced\_topk$ = $allgatherv$($reduced\_topk^i$)
11:     $global\_th$ = $th\_re\_evaluate$($all\_reduced\_topk$, $k$)
12: end if
13: $u_t$,  $global\_topk\_indexes$ = $balance\_and\_allgatherv$($reduced\_topk^i$, $global\_th$)
14: $indexes$ = $local\_topk\_indexes \cap global\_topk\_indexes$
15: return $u_t$, $indexes$
```

Lines 2-4, the local top-k threshold is re-evaluated after every τ' iterations. In Lines 5-7, the region boundaries are re-evaluated after every τ iterations. *Split and reduce* is conducted in Line 8, which returns the reduced local top-k values in region i and the indexes of local top-k values. In Lines 9-12, the global top-k threshold is re-evaluated after every τ' iterations. *Balance and allgatherv* is conducted in Line 13, which returns a sparse tensor u_t with global top-k values and the indexes of global top-k values. Line 14 calculates the intersection of the indexes of local top-k values and the indexes of global top-k values. This intersection (will be used in Ok-Top$k$ SGD in Section 4) covers the indexes of local values which eventually contribute to the global top-k values.

3.2 Lower bound for communication volume

Theorem 3.1. *For sparse gradients stored in COO format, O(k) sparse allreduce incurs at least* $2k\frac{P-1}{P}$ *communication volume.*

Proof. For O(k) sparse allreduce, each worker eventually obtains the global top-k values. Assume that all workers receive less than $k\frac{P-1}{P}$ values from the others, which means that each worker already has more than $\frac{k}{P}$ of the global top-k values locally. By adding up the number of global top-k values in each worker, we obtain more than k global top-k values, which is impossible. Therefore, each worker has to receive at least $k\frac{P-1}{P}$ values. Considering the corresponding k indexes, the lower bound is $2k\frac{P-1}{P}$.

□

The lower bound in Theorem 3.1 is achieved by O(k) sparse allreduce in the following special case: All local top-k values of worker i are in region i that worker i is in charge of, so that the reduction across P workers is no longer required. Furthermore, the global top-k values are uniformly distributed among P workers, namely each worker has exactly $\frac{k}{P}$ of the global top-k values. Then, an allgather to obtain the global top-k values (plus k indexes) incurs $2k\frac{P-1}{P}$ communication volume. Therefore, the lower bound is tight. Since O(k) sparse allreduce incurs at most $6k\frac{P-1}{P}$ communication volume (see Equation 3), it is asymptotically optimal.

Algorithm 2 Ok-Top$k$ SGD

```
1: Inputs: stochastic gradient $G^i(\cdot)$ at worker $i$, value $k$, learning rate $\alpha$.
2: Initialize $\epsilon_0^i = 0$, $G_0^i = 0$
3: for $t = 1$ to $T$ do
4:     $acc_t^i = \epsilon_{t-1}^i + \alpha G_{t-1}^i(w_{t-1})$        ▷ Accumulate residuals
5:     $u_t$, $indexes$ = $Ok\_sparse\_allreduce$($acc_t^i$, $t$, $k$)
6:     $\epsilon_t^i = acc_t^i - acc_t^i(indexes)$        ▷ Update residuals
7:     $w_t = w_{t-1} - \frac{1}{P}u_t$        ▷ Apply the model update
8: end for
```

4 Ok-Top$k$ SGD Algorithm

We discuss how O(k) sparse allreduce works with the SGD optimizer for distributed training in this section. An algorithmic description of Ok-Top$k$ SGD in presented in Algorithm 2. The key point of the algorithm is to accumulate the residuals (i.e., the gradient values not contributing to the global top-k values) locally, which may be chosen by the top-k selection in the future training iterations to make contribution. Empirical results of existing work [4, 36, 42] show that residual accumulation benefits the convergence. We use ϵ_t^i to represent the residuals maintained by worker i at iteration t. In Line 4 of Algorithm 2, residuals are accumulated with the newly generated gradient to obtain the accumulator acc_t^i. Then, in Line 5, acc_t^i is passed to O(k) sparse allreduce (presented in Algorithm 1), which returns the sparse tensor u_t containing global top-k values and the *indexes* marking the values in acc_t^i which contribute to u_t. In Line 6, residuals are updated by setting the values in acc_t^i marked by *indexes* to zero. In Line 7, u_t is applied to update the model parameters.

4.1 Convergence proof

Unless otherwise stated, $\|\cdot\|$ denotes the 2-norm.

Theorem 4.1. *Consider the Ok-Topk SGD algorithm when minimizing a smooth, non-convex objective function f. Then there exists a learning rate schedule $(\alpha_t)_{t=1,T}$ such that the following holds:*

$$\min_{t\in\{1,\ldots,T\}} \mathbb{E}[\|\nabla f(w_t)\|^2] \overset{T\to\infty}{\to} 0 \tag{4}$$

Proof. The update to w_{t+1} in Ok-Top$k$ SGD is

$$\mathrm{Top}k\left(\frac{1}{P}\sum_{i=0}^{P-1}\mathrm{Top}k\left(\alpha G_t^i(w_t)+\epsilon_t^i\right)\right),$$

while top-k components of the sum of updates across all P workers, i.e., the true global top-k values intended to be applied, is

$$\mathrm{Top}k\left(\frac{1}{P}\sum_{i=0}^{P-1}\left(\alpha G_t^i(w_t)+\epsilon_t^i\right)\right).$$

We assume the difference between the update calculated by Ok-Top$k$ SGD and the true global top-k values is bounded by the norm of the true gradient $G_t(w_t)=\frac{1}{P}\sum_{i=0}^{P-1}G_t^i(w_t)$. Then, we make the following assumption:

Assumption 1. *There exists a (small) constant ξ such that, for every iteration $t \geq 0$, we have:*

$$\left\| \mathrm{Top}k\left(\frac{1}{P}\sum_{i=0}^{P-1}\left(\alpha G_t^i(w_t)+\epsilon_t^i\right)\right) - \mathrm{Top}k\left(\frac{1}{P}\sum_{i=0}^{P-1}\mathrm{Top}k\left(\alpha G_t^i(w_t)+\epsilon_t^i\right)\right)\right\| \leq \xi\|\alpha G_t(w_t)\| \tag{5}$$

We validate Assumption 1 by the empirical results on different deep learning tasks in Section 5.1. Then, we utilize the convergence proof process for Top$k$ SGD in the non-convex case, presented in the work of [4], to prove Theorem 4.1. □

Regarding Theorem 4.1, we have the following discussion. First, since we analyze non-convex objectives, a weaker notion of convergence than in the convex case (where we can prove convergence to a global minimum) is settled. Specifically, for a given sequence of learning rates, we prove that the algorithm will converge to a stationary point of negligible gradient. Second, like most theoretical results, it does not provide a precise set for the hyperparameters, except the indication of diminishing learning rates.

5 Evaluations

We conduct our experiments on the CSCS Piz Daint supercomputer. Each Cray XC50 compute node contains an Intel Xeon E5-2690 CPU, and one NVIDIA P100 GPU with 16 GB global memory. We utilize the GPU for acceleration in all following experiments. The compute nodes of Piz Daint are connected by a Cray Aries interconnect network in a Dragonfly topology.

We use three neural networks from different deep learning domains summarized in Table 2 for evaluation. For VGG-16, we use SGD optimizer with initial learning rate of 0.1; for LSTM, we use SGD optimizer with initial learning rate of 1e-3; for BERT, we use Adam [24] optimizer with initial learning rate of 2e-4, β_1 =0.9, β_2 =0.999, weight decay of 0.01, and linear decay of the learning rate. For BERT, sparse allreduce is conducted on the gradients and Adam optimizer is applied afterwards. We compare the performance of Ok-Top$k$ with the parallel SGD schemes using the dense and sparse allreduce algorithms listed in Table 1, which covers the state-of-the-art. For a fair comparison, all schemes are implemented in PyTorch [33] with *mpi4py* as the communication library, which is built against Cray-MPICH 7.7.16. Commonly, the gradients of network layers locate in non-contiguous buffers. We use **Dense** to denote a single dense allreduce on a long message aggregated from the gradients of all neural network layers. Furthermore, we use **DenseOvlp** to denote dense allreduce with the optimization of communication and computation overlap. For DenseOvlp, the gradients are grouped into buckets and the message aggregation is conducted within each bucket; once the aggregated message in a bucket is ready, a dense allreduce is fired. The sparse allreduce counterparts (i.e., **Top*k*A**, **Top*k*DSA**, **gTop*k***, and **Gaussian*k***) are already discussed in Section 2. In all following experiments, we define *density* as k/n, where n is the number of components in the gradient.

Table 2. Neural networks used for evaluation.

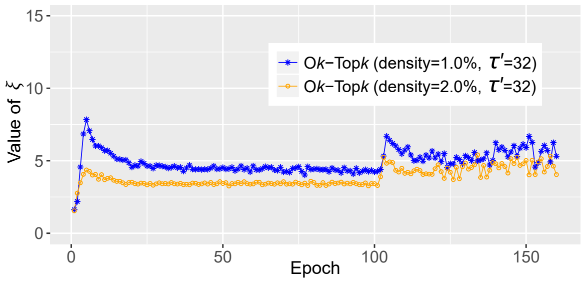
Tasks	Models	Parameters	Dataset
Image classification	VGG-16 [44]	14,728,266	Cifar-10
Speech recognition	LSTM [21]	27,569,568	AN4 [1]
Language processing	BERT [13]	133,547,324	Wikipedia [13]

We utilize the *topk* function provided by PyTorch [33], which is accelerated on GPU, to realize the top-k selection in Top$k$A, Top$k$DSA, and gTop$k$, as well as the periodic threshold re-evaluation in Ok-Top$k$.

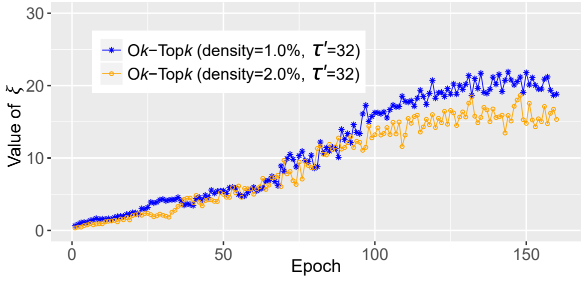
5.1 Evaluate the empirical value of ξ

To validate Assumption 1, we present the empirical values of ξ when training two models until convergence with different densities in Figure 5. For VGG-16 and BERT, the value of ξ increases quickly in the first few epochs or training iterations, and then turns to be stable. For LSTM, the value of ξ gradually increases at the beginning and tends to plateau in the second half of the training. For all three models, the value of ξ with a higher density is generally smaller than that with a lower density, especially in the stable intervals. This can be explained trivially by the reason that, the higher the density the higher probability that the results of sparse and dense allreduces get closer.

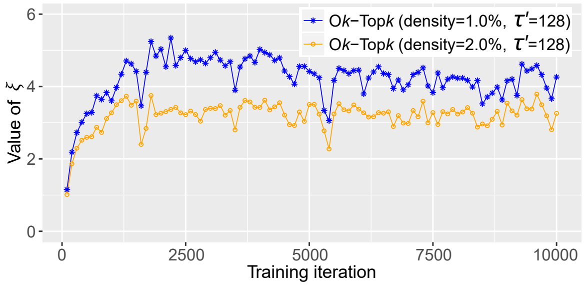
As shown in Equation 14 of [4], the effect of ξ is dampened by both P and small (i.e., less than 1) learning rates. If ξ<P (satisfied by all three models in Figure 5) or not too larger than P, we consider it has no significant effect on the convergence. Although ξ slightly grows in Figure 5b, which is caused by the decreasing of the true gradient norm as



(a) VGG-16 on Cifar-10 running on 16 GPU nodes.

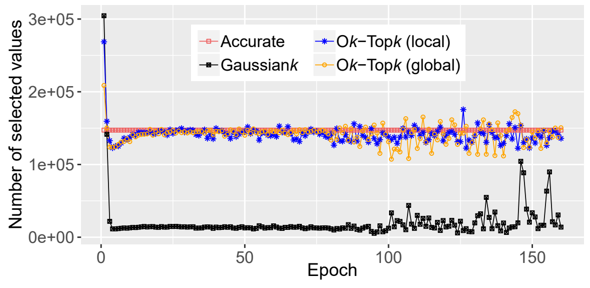


(b) LSTM on AN4 running on 32 GPU nodes.

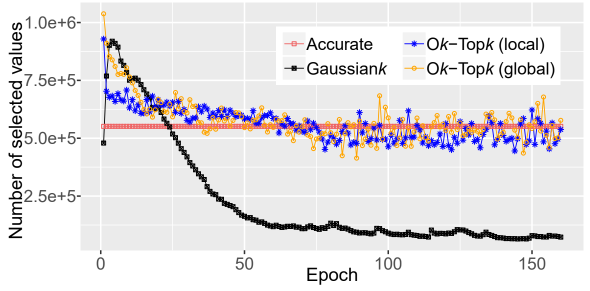


(c) BERT on Wikipedia on 32 GPU nodes.

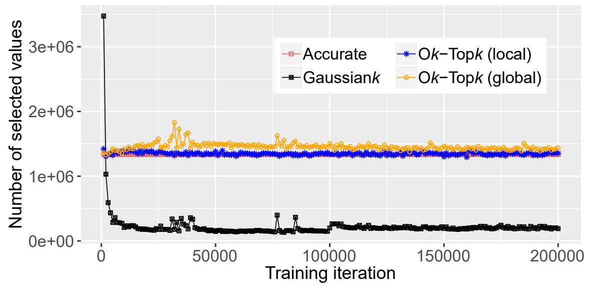
Figure 5. The empirical value of ξ.



(a) VGG-16 on Cifar-10 on 16 GPUs (density=1.0%, τ'=32).



(b) LSTM on AN4 on 32 GPUs (density=2.0%, τ'=32).



(c) BERT on Wikipedia on 32 GPUs (density=1.0%, τ'=128).

Figure 6. Selections for local and global top-k values

the model converges, a small learning rate (e.g., 0.001) can restrain its effect. Overall, Assumption 1 empirically holds with relatively low, stable or slowly growing values of ξ.

5.2 Top-k values selection

We will verify the accuracy of the top-k selection strategy used by Ok-Top$k$ on different neural network models. For VGG-16 and LSTM, the models are trained for 160 epochs until convergence with τ'=32. Recall that τ' is the period of thresholds re-evaluation. For BERT, the model is trained for 200,000 iterations (more than 20 hours on 32 GPU nodes) with τ'=128. The numbers of local and global top-k values selected by Ok-Top$k$ during training are monitored. We also record the values of k predicted by Gaussian$k$ for comparison. The results are reported in Figure 6. We can see that the numbers of both local and global top-k values selected by Ok-Top$k$ are very close to the accurate number for a given density, except that Ok-Top$k$ overestimates the value of k in the early epochs of VGG-16 and LSTM. For both local top-k and global top-k on three models, the average deviation from the accurate number is below 11%. For example, the average deviation for local top-k selection on BERT is only 1.4%. These results demonstrate the accuracy of the threshold reuse strategy adopted by Ok-Top$k$. On the contrary, Gaussian$k$ overestimates the value of k in the first few epochs and then severely underestimate k (an order of magnitude lower than the accurate number) in the following epochs. This can be explained by the difference between Gaussian and the real distributions, as discussed in Section 3.1.3.

As a comparison, we also count the density of the output buffer (i.e., the accumulated gradient) for Top$k$DSA (Top$k$A has the same density), which expands to 13.2% and 34.5% on average for VGG-16 (local density = 1.0%, on 16 GPUs) and LSTM (local density = 2.0%, on 32 GPUs), respectively. These statistics show the effect of the fill-in issue for Top$k$DSA.

5.3 Optimizations for load balancing in Ok-Top$k$

To evaluate the effectiveness of the load balancing optimizations of O(k) sparse allreduce, we train BERT for 8,192 iterations and report the average values.

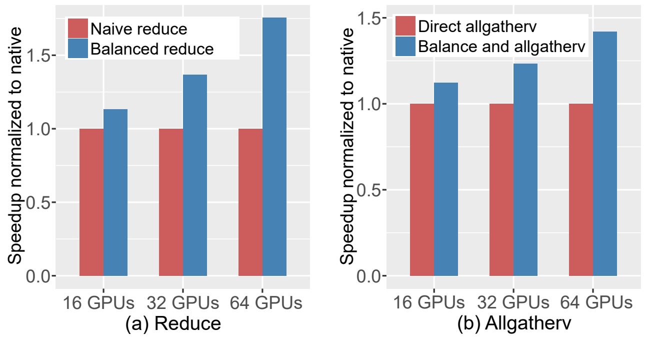


Figure 7. Evaluation for communication balancing in Ok-Top$k$ using BERT with density = 1.0%.

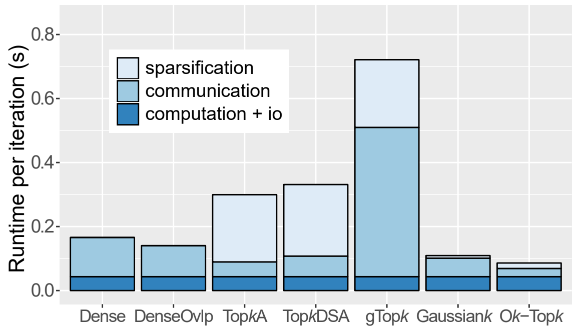
First, we evaluate the periodic space repartition strategy (discussed in Section 3.1.1) for load balancing in the phase of *split and reduce*. The results are presented in Figure 7(a). Recall that we set the period τ to 64. The repartition overhead is counted and averaged in the runtime of the balanced reduce. In the naive reduce, the gradient space is partitioned into equal-sized regions, regardless of the coordinate distribution of the local top-k values. The balanced reduce achieves 1.13x to 1.75x speedup over the naive one, with a trend of more significant speedup on more GPUs. This trend can be explained by that the load imbalance in the naive reduce incurs up to $2(P-1)k$ communication volume (proportional to P). While the balanced reduce incurs less than $2k$ communication volume, which is more scalable.

Next, we evaluate the data balancing strategy (discussed in Section 3.1.2) in the phase of *balance and allgatherv*. Although data balancing helps to bound the bandwidth overhead of allgatherv, there is no need to conduct it if the data is roughly balanced already. Empirically, we choose to conduct data balancing before allgatherv if the max data size among P workers is more than four times larger than the average data size, and otherwise use an allgatherv directly. Figure 7(b) presents the results for the iterations where data balancing is triggered. Data balancing and allgatherv achieve 1.12x to 1.43x speedup over the direct allgatherv. For similar reasons as *split and reduce*, more speedup is achieved on more GPUs.

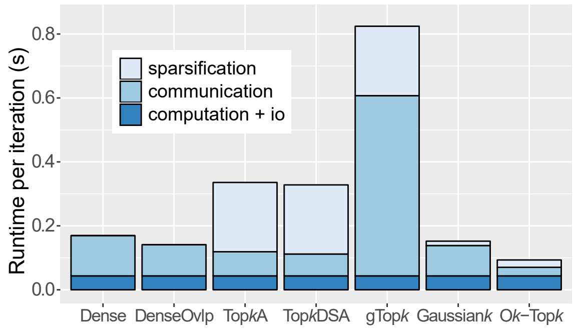
5.4 Case studies on training time and model convergence

We study the training time and model convergence using real-world applications listed in Table 2. For training time per iteration, we report the average value of full training. To better understand the results, we make a further breakdown of the training time, including sparsification (i.e., top-k selection from the gradient), communication (i.e., dense or sparse allreduces), and computation (i.e., forward and backward passes) plus I/O (i.e., sampling from dataset).

As discussed in Section 5.2, Gaussian$k$ usually underestimates the value of k, which makes the actual density



(a) Running on 16 GPU nodes with global batch size = 256.

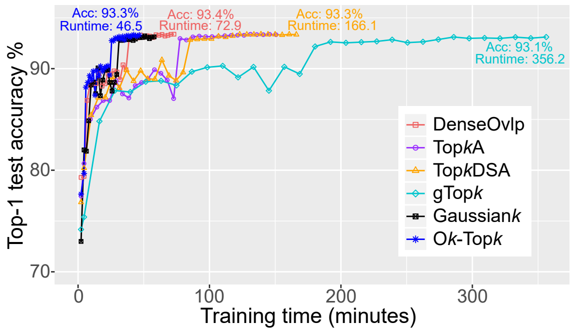


(b) Running on 32 GPU nodes with global batch size = 512.

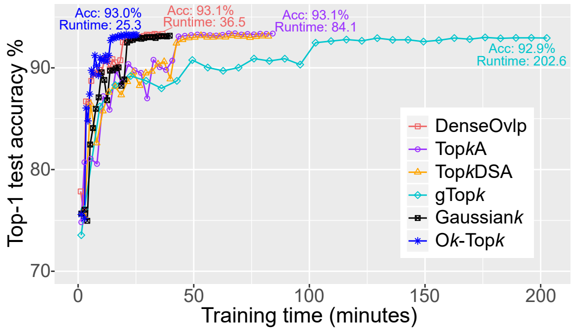
Figure 8. Weak scaling of training VGG-16 on Cifar-10 with density = 2.0%.

far below the setting. Both empirical and theoretical results [4, 36, 42] show that a very low density would jeopardize the convergence. To make a fair comparison between the counterparts for both training time and model accuracy, we gradually scale the predicted threshold of Gaussian$k$ until the number of selected values is more than $3k/4$. The threshold adjustment is also suggested by [41], although it is difficult to be accurate. The threshold adjustment may slightly increase the sparsification overhead of Gaussian$k$, but compared with the other overheads it can be ignored (see the following results).

5.4.1 Image classification. Figure 8 presents the results of weak scaling for training VGG-16 on Cifar-10. DenseOvlp outperforms Dense by enabling communication and computation overlap. Although Top$k$A and Top$k$DSA have lower communication overhead than DenseOvlp, they have a high overhead for sparsification, which makes the benefit of lower communication disappear. Note that the communication overhead of gTop$k$ seems much higher than the others; this is because the overhead of hierarchical top-k selections in the reduction-tree (with $\log P$ steps) is also counted in the communication overhead. Among all sparse allreduce schemes, Gaussian$k$ has the lowest sparsification overhead. Ok-Top$k$ has the lowest communication overhead; by using the threshold reuse strategy, Ok-Top$k$ only has a slightly higher sparsification overhead than Gaussian$k$. When scaling from 16

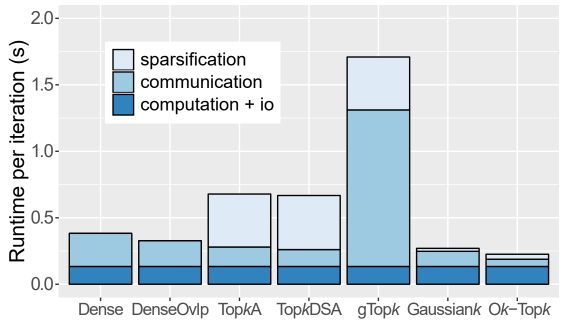


(a) Running on 16 GPU nodes with global batch size = 256.

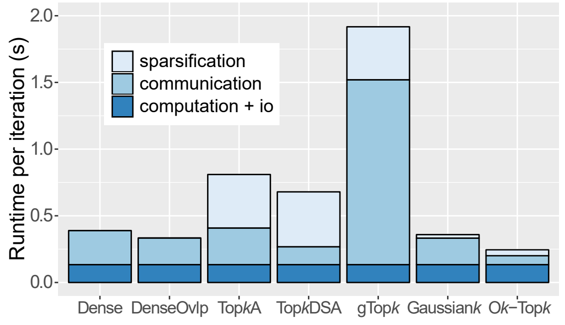


(b) Running on 32 GPU nodes with global batch size = 512.

Figure 9. Top-1 test accuracy for VGG-16 on Cifar-10 with density = 2.0% training for 160 epochs.

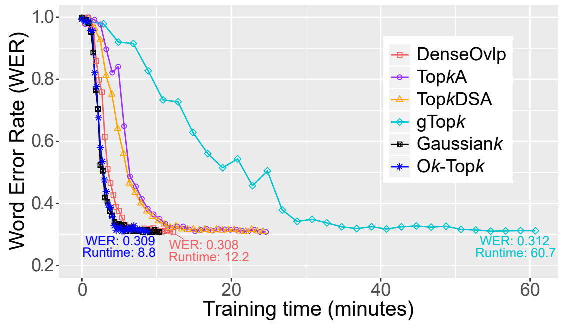


(a) Running on 32 GPU nodes with global batch size = 64.

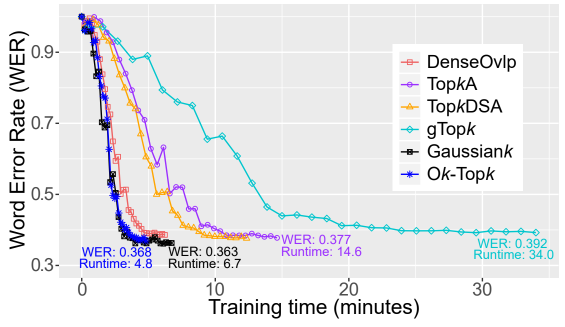


(b) Running on 64 GPU nodes with global batch size = 128.

Figure 10. Weak scaling of training LSTM on AN4 with density = 2.0%.



(a) Running on 32 GPU nodes with global batch size = 64.



(b) Running on 64 GPU nodes with global batch size = 128.

Figure 11. WER for LSTM on AN4 with density = 2.0% training for 160 epochs.

GPUs to 32 GPUs, the communication overhead of Top*k*A and Gaussian*k* almost doubles. This is because allgather-based sparse allreduce is not scalable (see the performance model in Table 1). On 32 GPU nodes, O*k*-Top*k* outperforms the other schemes by 1.51x-8.83x for the total training time.

Figure 9 presents the Top-1 test accuracy as a function of runtime by training VGG-16 on Cifar-10 for 160 epochs. On both 16 and 32 GPUs, the accuracy achieved by O*k*-Top*k* is very close to dense allreduce. We did not do any hyperparameter optimization except simply diminishing the learning rate. The accuracy results are consistent with these reported in machine learning community [6, 41]. On both 16 and 32 GPUs, O*k*-Top*k* achieves the fastest time-to-solution.

5.4.2 Speech recognition. Figure 10 presents the results of weak scaling for training LSTM on AN4. Similar to the results on VGG-16, O*k*-Top*k* has a better scalability than the counterparts. On 64 GPUs, O*k*-Top*k* outperforms the other schemes by 1.34x-7.71x for the total training time.

Figure 11 presents the test Word Error Rate (WER, the smaller the better) as a function of runtime by training for 160 epochs. On 32 GPUs, O*k*-Top*k* is 1.39x faster than DenseOvlp, and achieves 0.309 WER, which is very close to DenseOvlp (0.308). On 64 GPUs, all schemes achieve higher WERs than these on 32 GPUs. This is because the model accuracy is compromised by using a larger global batch size, which is also observed in many other deep learning tasks [7, 51, 53].

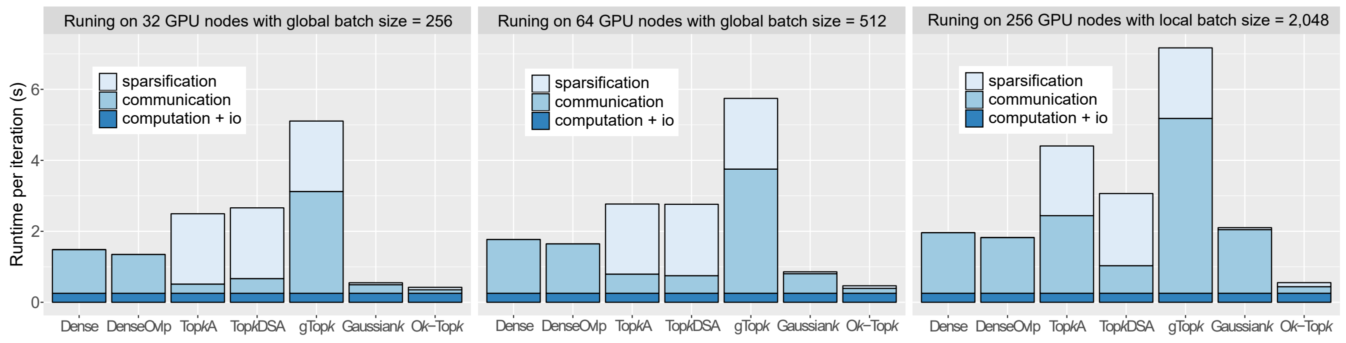


Figure 12. Weak scaling (from 32 to 256 GPU nodes) of training BERT on Wikipedia with density = 1.0%.

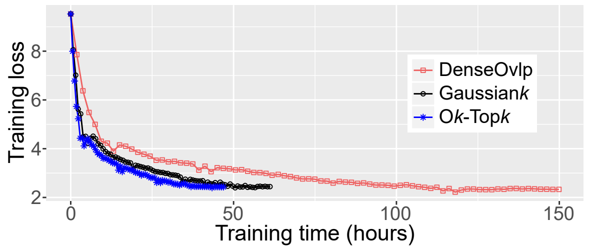


Figure 13. BERT pre-training on 32 GPU nodes for 400,000 iterations with global batch size = 256 and density = 1.0%.

How to tune hyperparameters for better accuracy with large batches is not the topic of this work. Surprisingly, on 64 GPUs, O*k*-Top*k*, Gaussian*k*, Top*k*A and Top*k*DSA achieve lower WERs than DenseOvlp, which may be caused by the noise introduced by the sparsification. Overall, on both 32 and 64 GPUs, O*k*-Top*k* achieves the fastest time-to-solution.

5.4.3 Natural language processing. BERT [13] is a popular language model based on Transformer [46]. The model is usually pre-trained on a large dataset and then fine-tuned for various downstream tasks. Pre-training is commonly much more expensive (years on a single GPU) than fine-tuning. Therefore, we focus on pre-training in the evaluation.

Figure 12 presents the results of weak scaling for pre-training BERT. When scaling to 256 GPUs, the communication overhead of Top*k*A and Gaussian*k* is even higher than the dense allreduce, which again demonstrates that the allgather-based sparse allreduce is not scalable. Top*k*DSA exhibits better scalablity than the allgather-based sparse allreduce, but its communication overhead also significantly increases, since the more workers, the more severe the fill-in problem [36]. On 256 GPUs, O*k*-Top*k* outperforms all counterparts by 3.29x-12.95x. Using 32 nodes as the baseline, O*k*-Top*k* achieves 76.3% parallel efficiency on 256 GPUs in weak scaling, which demonstrates a strong scalabilty of O*k*-Top*k*.

In Figure 13, we report the training loss by pre-training BERT from scratch on the Wikipedia dataset (containing 114.5 million sequences with a max length of 128) for 400,000 iterations. Eventually, the training loss of O*k*-Top*k* decreases to 2.43, which is very close to DenseOvlp (2.33). These results show that O*k*-Top*k* has a similar convergence rate as the dense allreduce for BERT pre-training. Compared with DenseOvlp, O*k*-Top*k* reduces the total training time on 32 GPUs from 150 hours to 47 hours (more than 3x speedup), and also outperforms Gaussian*k* by 1.30x. Since pre-training BERT is very costly (energy- and time-consuming), in Figure 13 we only present the results for two important baselines (i.e., Gaussian*k*, with the highest training throughput among all baselines, and DenseOvlp, a lossless approach). Since the other baselines are inferior to Gaussian*k* and DenseOvlp in terms of training throughput and not better than DenseOvlp in terms of convergence rate, it is sufficient to show the advantage of O*k*-Top*k* by comparing it with these two important baselines in Figure 13.

6 Conclusion

O*k*-Top*k* is a novel scheme for distributed deep learning training with sparse gradients. The sparse allreduce of O*k*-Top*k* incurs less than 6*k* communication volume, which is asymptotically optimal and more scalable than the counterparts. O*k*-Top*k* enables an efficient and accurate top-*k* values prediction by utilizing the temporal locality of gradient value statistics. Empirical results for data-parallel training of real-world deep learning models on the Piz Daint supercomputer show that O*k*-Top*k* significantly improves the training throughput while guaranteeing the model accuracy. The throughput improvement would be more significant on commodity clusters with low-bandwidth network. We foresee that our scheme will play an important role in scalable distributed training for large-scale models with low communication overhead. In future work, we aim to further utilize O*k*-Top*k* to reduce the communication overhead in distributed training with a hybrid data and pipeline parallelism [15, 27, 31, 32].

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880, EPiGRAM-HS, No. 801039, and MAELSTROM, No. 955513). We also thank the Swiss National Supercomputing Center for providing the computing resources and technical support.

References

[1] Alejandro Acero and Richard M Stern. 1990. Environmental robustness in automatic speech recognition. In *International Conference on Acoustics, Speech, and Signal Processing*. IEEE, 849–852.

[2] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).

[3] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. *Advances in Neural Information Processing Systems* 30 (2017), 1709–1720.

[4] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Sarit Khirirat, Nikola Konstantinov, and Cédric Renggli. 2018. The convergence of sparsified gradient methods. *arXiv preprint arXiv:1809.10505* (2018).

[5] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. 2012. Cray XC series network. *Cray Inc., White Paper WP-Aries01-1112* (2012).

[6] Babajide O Ayinde and Jacek M Zurada. 2018. Building efficient convnets using redundant feature pruning. *arXiv preprint arXiv:1802.07653* (2018).

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

[8] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. 2018. signSGD: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*. PMLR, 560–569.

[9] Léon Bottou, Frank E Curtis, and Jorge Nocedal. 2018. Optimization methods for large-scale machine learning. *SIAM Rev.* 60, 2 (2018).

[10] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[11] Yi Cai, Yujun Lin, Lixue Xia, Xiaoming Chen, Song Han, Yu Wang, and Huazhong Yang. 2018. Long live time: improving lifetime for training-in-memory engines by structured gradient sparsification. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.

[12] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. 2007. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience* 19, 13 (2007), 1749–1783.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

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

[15] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[16] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 676–691.

[17] Denis Foley and John Danskin. 2017. Ultra-performance Pascal GPU and NVLink Interconnect. *IEEE Micro* 37, 2 (2017), 7–17.

[18] 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 preprint arXiv:1706.02677* (2017).

[19] Pengchao Han, Shiqiang Wang, and Kin K Leung. 2020. Adaptive gradient sparsification for efficient federated learning: An online learning approach. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 300–310.

[20] Debra Hensgen, Raphael Finkel, and Udi Manber. 1988. Two algorithms for barrier synchronization. *International Journal of Parallel Programming* 17, 1 (1988), 1–17.

[21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[22] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554* (2021).

[23] Samuel Horváth, Dmitry Kovalev, Konstantin Mishchenko, Sebastian Stich, and Peter Richtárik. 2019. Stochastic distributed learning with gradient quantization and variance reduction. *arXiv preprint arXiv:1904.05115* (2019).

[24] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[25] Shigang Li, Tal Ben-Nun, Salvatore Di Girolamo, Dan Alistarh, and Torsten Hoefler. 2020. Taming unbalanced training workloads in deep learning with partial collective operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[26] Shigang Li, Tal Ben-Nun, Giorgi Nadiradze, Salvatore Di Girolamo, Nikoli Dryden, Dan Alistarh, and Torsten Hoefler. 2020. Breaking (global) barriers in parallel stochastic optimization with wait-avoiding group averaging. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1725–1739.

[27] Shigang Li and Torsten Hoefler. 2021. Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines. *arXiv preprint arXiv:2107.06925* (2021).

[28] Shigang Li, Torsten Hoefler, and Marc Snir. 2013. NUMA-aware shared-memory collective communication for MPI. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. 85–96.

[29] Hosam M Mahmoud, Reza Modarres, and Robert T Smythe. 1995. Analysis of quickselect: An algorithm for order statistics. *RAIRO-Theoretical Informatics and Applications-Informatique Théorique et Applications* 29, 4 (1995), 255–276.

[30] Giorgi Nadiradze, Amirmojtaba Sabour, Peter Davies, Shigang Li, and Dan Alistarh. 2021. Asynchronous decentralized SGD with quantized and local updates. *Advances in Neural Information Processing Systems* 34 (2021).

[31] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[32] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, 7937–7947.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style,

high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[34] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[35] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 33. 4780–4789.

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

[37] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. 2020. An In-Depth Analysis of the Slingshot Interconnect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC20)*.

[38] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[39] Anil Shanbhag, Holger Pirk, and Samuel Madden. 2018. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*. 1557–1570.

[40] Tom Shanley. 2003. *InfiniBand network architecture*. Addison-Wesley Professional.

[41] Shaohuai Shi, Xiaowen Chu, Ka Chun Cheung, and Simon See. 2019. Understanding top-k sparsification in distributed deep learning. *arXiv preprint arXiv:1911.08772* (2019).

[42] Shaohuai Shi, Qiang Wang, Kaiyong Zhao, Zhenheng Tang, Yuxin Wang, Xiang Huang, and Xiaowen Chu. 2019. A distributed synchronous SGD algorithm with global top-k sparsification for low bandwidth networks. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2238–2247.

[43] Shaohuai Shi, Kaiyong Zhao, Qiang Wang, Zhenheng Tang, and Xiaowen Chu. 2019. A Convergence Analysis of Distributed SGD with Communication-Efficient Gradient Sparsification.. In *IJCAI*. 3411–3417.

[44] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[45] 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.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[47] Linnan Wang, Wei Wu, Junyu Zhang, Hang Liu, George Bosilca, Maurice Herlihy, and Rodrigo Fonseca. 2020. FFT-based Gradient Sparsification for the Distributed Training of Deep Neural Networks. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 113–124.

[48] 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 *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1508–1518.

[49] Ke Wu, Dezun Dong, Cunlu Li, Shan Huang, and Yi Dai. 2019. Network congestion avoidance through packet-chaining reservation. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[50] Hang Xu, Kelly Kostopoulou, Aritra Dutta, Xin Li, Alexandros Ntoulas, and Panos Kalnis. 2021. DeepReduce: A Sparse-tensor Communication Framework for Federated Deep Learning. *Advances in Neural Information Processing Systems* 34 (2021).

[51] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large-batch training for LSTM and beyond. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[52] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962* (2019).

[53] 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*. 1–10.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code for our Ok-Top$k$ and the benchmarks used in the evaluation. It supports the results in Section 5. To validate or reproduce the results, build this artifact and check the results returned by running benchmarks.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Ok-Top$k$
- **Compilation:** Python 3.8
- **Model:** VGG, LSTM, BERT
- **Data set:** Cifar-10, AN4, Wikipedia
- **Run-time environment:** torch, mpi4py, apex
- **Hardware:** GPU clusters
- **Execution:** srun or mpirun
- **Metrics:** execution time, top-1 test accuracy, WER, training loss
- **Output:** txt files
- **How much disk space required (approximately)?:** 100 GB
- **How much time is needed to prepare workflow (approximately)?:** Except for preparing the Wikipedia dataset, it takes about 30 minutes. Preprocessing the Wikipedia dataset takes several hours.
- **How much time is needed to complete experiments (approximately)?:** It takes about 5 hours without pre-training BERT. BERT pre-training takes more than 100 hours.
- **Archived?:** Yes. https://doi.org/10.5281/zenodo.5808267

A.3 Description

A.3.1 How to access.
The artifact can be downloaded from https://github.com/Shigangli/Ok-Topk

The artifact can also be downloaded using the DOI link https://doi.org/10.5281/zenodo.5808267

A.3.2 Hardware dependencies.
GPU clusters

A.3.3 Software dependencies.
To run the experiments, Python 3.8 is required. Python packages, including `torch, mpi4py, apex, simplejson, tensorboard, tensorboardX, ujson, tqdm, h5py, coloredlogs, psutil, torchaudio, torchvision, numba, librosa, python-Levenshtein`, and `warpctc-pytorch`, are required.

A.4 Installation

1. Download the artifact and extract it in your personal `$WORK` directory.

2. Setup Python environment and install the dependent packages.

```
> conda create -name py38_oktopk python=3.8
> conda activate py38_oktopk

> pip3 install pip==20.2.4
> pip install -r requirements.txt
> MPICC="cc -shared" pip install -no-binary=mpi4py
mpi4py
> git clone https://github.com/NVIDIA/apex
> cd apex
> pip install -v -disable-pip-version-check -no-cache-dir
-global-option="-cpp_ext" -global-option="-cuda_ext" ./
```

3. Download and preprocess data sets.

a) Cifar-10

```
> cd $WORK/Ok-Topk/VGG/vgg_data
> wget https://www.cs.toronto.edu/~kriz/
cifar-10-python.tar.gz
> tar -zxvf cifar-10-python.tar.gz
```

b) AN4

```
> cd $WORK/Ok-Topk/LSTM/audio_data
> wget www.dropbox.com/s/l5w4up20u5pfjxf/an4.zip
> unzip an4.zip
```

c) Wikipedia

```
> cd $WORK/Ok-Topk/BERT/bert/bert_data
```

Prepare the dataset according to the `README` file in this directory.

A.5 Experiment workflow

We run experiments on GPU clusters with `SLURM` job scheduler. In the job scripts of the artifact, we use `srun` to launch multiple processes among the computation nodes. But one can also use `mpirun` to launch multiple processes if there is no `SLURM` job scheduler on your machine. Once the jobs are finished, the results of training speed, test accuracy and training loss values for each algorithm will be output into `txt` files.

A.6 Evaluation and expected result

1. To run VGG jobs

```
> cd $WORK/Ok-Topk/VGG
> ./sbatch_vgg_jobs.sh
```

2. To run LSTM jobs

```
> cd $WORK/Ok-Topk/LSTM
> ./sbatch_lstm_jobs.sh
```

3. To run BERT jobs

```
> cd $WORK/Ok-Topk/BERT/bert
> ./sbatch_bert_jobs.sh
```

4. Check the output results

It takes about 5 hours for all jobs to be finished, which depends on the busyness of the job queue. To make sure all jobs have been finished, checking the status by:

```
> squeue -u username
```

If you see no job is running, then all jobs are finished.

```
> cd $WORK/Ok-Topk/VGG
```

Check the 6 output `txt` files for the 6 algorithms (i.e., `dense, gaussiank, gtopk, oktopk, topkA`, and `topkDSA`), respectively, which contain the training speed and top-1 test accuracy for VGG.

```
> cd $WORK/Ok-Topk/LSTM
```

Check the 6 output `txt` files for the 6 algorithms, respectively, which contain the training speed and WER values for LSTM.

```
> cd $WORK/Ok-Topk/BERT/bert
```

Check the 6 output `txt` files for the 6 algorithms, respectively, which contain the training speed and training loss values for BERT.

Users are expected to reproduce the results in this paper. Different software or hardware versions may lead to slightly variant results compared with the numbers reported in the paper, but it should not affect the general trends claimed in the paper, namely Ok-Top$k$ achieves higher training throughput and faster time-to-solution, and is more scalable than the other algorithms.

A.7 Experiment customization

Users can modify the variables in the job scripts to customize the experiments. For example, modify *density* to change the density of the gradient; modify *max-epochs* to change the number of training epochs; modify *nworkers* and *nodes* to change the number of processes used for training.

A.8 Notes

None.

A.9 Methodology

Submission, reviewing and badging methodology:
https://www.acm.org/publications/policies/artifactreview-badging
http://cTuning.org/ae/submission-20201122.html
http://cTuning.org/ae/reviewing-20201122.html