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-Topk, a scheme for distributed training with sparse gradients. Ok-Topk 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-Topk 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-Topk achieves similar model accuracy to dense allreduce. Compared with the optimized dense and the state-of-the-art sparse allreduces, Ok-Topk is more scalable and significantly improves training throughput (e.g., 3.29x-12.95x improvement for BERT on 256 GPUs).

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

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

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-Topk, 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-Topk in PyTorch [33] and compare it to four other sparse allreduce approaches. Specifically, Ok-Topk 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. The code of Ok-Top$k$ is available: https://github.com/ShigangLi/Ok-Top$k$

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 $f$ a loss function. During training, it computes the loss in the forward pass for each sample as $f(w_t, x_i, y_i)$, and then a stochastic gradient in the backward pass as

$$ G_f(w_t) = \frac{1}{b} \sum_{i=1}^{b} \nabla f(w_t, x_i, y_i). $$

The model is trained in iterations such that $w_{t+1} = w_t - \alpha G_f(w_t)$, where $\alpha$ 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 scalability 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>
<thead>
<tr>
<th>Algorithms</th>
<th>Bandwidth</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense [12]</td>
<td>$2nP^{-1}\beta$</td>
<td>$2(\log P)\alpha$</td>
</tr>
<tr>
<td>Top$k$A [36, 47]</td>
<td>$2k(P-1)\beta$</td>
<td>$(P+2\log P)\alpha$</td>
</tr>
<tr>
<td>Top$k$DSA [36]</td>
<td>$[4kP^{-1}\beta, (2k+n)P^{-1}\beta]^1$</td>
<td>$(P+2\log P)\alpha$</td>
</tr>
<tr>
<td>gTop$k$ [42]</td>
<td>$4k(\log P)\beta$</td>
<td>$(2\log P)\alpha$</td>
</tr>
<tr>
<td>Gaussian [41]</td>
<td>$2k(P-1)\beta$</td>
<td>$(2\log P)\alpha$</td>
</tr>
<tr>
<td>Ok-Top$k$</td>
<td>$[2kP^{-1}\beta, 6kP^{-1}\beta]^1$</td>
<td>$(2P+2\log P)\alpha$</td>
</tr>
</tbody>
</table>

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 $\alpha$ is the latency and $\beta$ 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 Allgather 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 all-gather is proportional to $P$ [12, 45] and thus not scalable. Top$k$DSA represents the Dynamic Sparse Allreduce 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. gTopk [42] implements the sparse allreduce using a reduction tree followed by a broadcast tree. To solve the fill-in problem, gTopk hierarchically selects top-k values in each level of the reduction tree, which results in 4k log P communication volume. Gaussiank [41] uses the same sparse allreduce algorithm as TopkA with a further optimization for top-k selection. For our Ok-Topk, the communication volume is bounded by 6k. Although Ok-Topk 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-Topk 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, Gaussiank [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 Gaussiank 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-Topk, 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-Topk.

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-Topk, analyze the complexity using the aforementioned latency (\(\alpha\)) - bandwidth (\(\beta\)) 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 Topk(∑P−1i=0 Topk(Gi)), where Gi is the sparse gradient on worker i at training iteration t, the inner Topk operator is the local top-k selection, and the outer Topk 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 0 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 t iterations. We empirically set t = 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)α) is amortized by the reuse in the following t−1 iterations, resulting in only (log P)α/t 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

\[ C_{\text{split and reduce}} = (P - 1)\alpha + 2k \frac{P - 1}{P} \beta \]  

(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.

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 allgatherv is required to collect the consecutive buffer sizes from P workers, which only incurs (log P)α overhead (the bandwidth term can be ignored). Then, each worker uses the collected buffer sizes to generate the communication scheme (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 Pa + 2k \frac{P - 1}{P} β. 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

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

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

$$C_{\text{Ok_sparse_allreduce}} \leq (2P + 2 \log P)\alpha + 6k\frac{P-1}{P}\beta. \quad (3)$$

### 3.1.3 Efficient selection for top-$k$ values

Ok-Topk 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 $G(t), G(t+1), \ldots G(t+t'-1)$ change very slowly. Therefore, we only calculate the accurate thresholds for local and global top-$k$ values after every $t'$ iterations, and then reuse the thresholds in the following $t'-1$ 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.

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-Topk uses a threshold calculated more than 25 iterations ago. We can see the threshold of Ok-Topk is still very close to the accurate threshold (see Section 5.2 for the accuracy verification for top-$k$ selections of Ok-Topk in the scenario of full training). On the contrary, Gaussian 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 Lines 2-4, the local top-$k$ threshold is re-evaluated after every $t'$ iterations. In Lines 5-7, the region boundaries are re-evaluated after every $r$ 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...
\begin{algorithm}
\caption{O(k) sparse allreduce}
1: \textbf{Inputs}: stochastic gradient $G^i_t$ at worker $i$, iteration $t$ ($t>0$), value $k$, space repartition period $\tau$, thresholds re-evaluation period $\tau'$. 
2: if $t$ mod $\tau' == 0$ then 
3: local\_th = th\_re\_evaluate($G^i_t$, $k$) 
4: end if 
5: if $t$ mod $\tau == 0$ then 
6: boundaries = space\_repartition($G^i_t$, local\_th) 
7: end if 
8: reduced\_topk, local\_topk\_indexes = split\_and\_reduce($G^i_t$, local\_th, boundaries) 
9: if $t$ mod $\tau' == 0$ then 
10: all\_reduced\_topk = all\_gather\_v(reduced\_topk) 
11: global\_th = th\_re\_evaluate(all\_reduced\_topk, $k$) 
12: end if 
13: $u_i$, global\_topk\_indexes = balance\_and\_all\_gather\_v(reduced\_topk, global\_th) 
14: indexes = local\_topk\_indexes $\cap$ global\_topk\_indexes 
15: return $u_i$, indexes 
\end{algorithm}

$r'$ iterations. Balance and allgather\_v is conducted in Line 13, which returns a sparse tensor $u_i$, 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-Topk 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 $\frac{1}{P}$ 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.

### 4 Ok-Topk 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-Topk 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 $e_i^t$ 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 $acc_i^t$. Then, in Line 5, $acc_i^t$ is passed to O(k) sparse allreduce (presented in Algorithm 1), which returns the sparse tensor $u_i$ containing global top-k values and the indexes marking the values in $acc_i^t$ which contribute to $u_i$. In Line 6, residuals are updated by setting the values in $acc_i^t$ marked by $indexes$ to zero. In Line 7, $u_i$ 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:

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

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

$$\text{Topk}\left(\frac{1}{P} \sum_{i=0}^{P-1} \text{Topk}\left(aG_i^t(w_t) + e_i^t\right)\right),$$

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
\[ \text{Top}(\frac{1}{p} \sum_{i=0}^{p-1} (aG_t^i(w_i) + e_t^i)). \]

We assume the difference between the update calculated by Ok-Topk 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 \( \xi \) such that, for every iteration \( t \geq 0 \), we have:

\[
\left\| \text{Top}\left(\frac{1}{p} \sum_{i=0}^{p-1} (aG_t^i(w_i) + e_t^i)\right) \right. - \text{Top}\left(\frac{1}{p} \sum_{i=0}^{p-1} \text{Top}(aG_t^i(w_i) + e_t^i)\right) \right\| \leq \xi \| aG_t(w_t) \| 
\]

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 Topk 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.

### Table 2. Neural networks used for evaluation.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Models</th>
<th>Parameters</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image classification</td>
<td>VGG-16 [44]</td>
<td>14,728,266</td>
<td>Cifar-10</td>
</tr>
</tbody>
</table>

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, \( \beta_1 = 0.9, \beta_2 = 0.999 \), 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-Topk 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., **TopkA**, **TopkDSA**, **gTopk**, 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.

We utilize the **topk** function provided by PyTorch [33], which is accelerated on GPU, to realize the top-\( k \) selection in TopkA, TopkDSA, and gTopk, as well as the periodic threshold re-evaluation in Ok-Topk.

### 5.1 Evaluate the empirical value of \( \xi \)

To validate Assumption 1, we present the empirical values of \( \xi \) when training two models until convergence with different densities in Figure 5. For VGG-16 and BERT, the value of \( \xi \) increases quickly in the first few epochs or training iterations, and then turns to be stable. For LSTM, the value of \( \xi \) gradually increases at the beginning and tends to plateau in the second half of the training. For all three models, the value of \( \xi \) 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 \( \xi \) is dampened by both \( P \) and small (i.e., less than 1) learning rates. If \( \xi \)\( P \) (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 \( \xi \) slightly grows in Figure 5b, which is caused by the decreasing of the true gradient norm as 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 \( \xi \).
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 \(\tau' = 32\). Recall that \(\tau'\) 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 \(\tau' = 128\). 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\(kA\) 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 O(k)-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.

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 \(r\) to 64. The repartition overhead is

\[\text{Number of selected values} \begin{array}{c}
0.0e+00 \\
2.5e+5 \\
5.0e+5 \\
7.5e+5 \\
1.0e+6 \\
\end{array} \]

\(\text{Epoch} \begin{array}{c}
0 \\
2500 \\
5000 \\
7500 \\
10000 \\
\end{array} \]

\text{VGG-16 on Cifar-10 running on 16 GPU nodes.}

\(\text{Number of selected values} \begin{array}{c}
0.0e+00 \\
2.0e+00 \\
4.0e+00 \\
6.0e+00 \\
8.0e+00 \\
\end{array} \]

\(\text{Epoch} \begin{array}{c}
0 \\
2500 \\
5000 \\
7500 \\
10000 \\
\end{array} \]

\text{LSTM on AN4 running on 32 GPU nodes.}

\(\text{Number of selected values} \begin{array}{c}
0.0e+00 \\
1.0e+00 \\
2.0e+00 \\
3.0e+00 \\
4.0e+00 \\
\end{array} \]

\(\text{Epoch} \begin{array}{c}
0 \\
50000 \\
100000 \\
150000 \\
200000 \\
\end{array} \]

\text{BERT on Wikipedia on 32 GPU nodes.}

\text{Figure 5. The empirical value of } \xi.\text{ }

\text{Figure 6. Selections for local and global top-}k\text{ values}
We study the training time and model convergence using workers is more than four times larger than the average data size, leading to significant speedup on more GPUs. This trend can be explained by the fact that the load imbalance in 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 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\(k\) and TopkDSA 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 GPUs to 32 GPUs, the communication overhead of Top\(k\) and Gaussian\(k\) almost doubles. This is because allgatherv-based sparse allreduce is not scalable (see the performance

![Figure 7](image1.png)

**Figure 7.** Evaluation for communication balancing in Ok-Top\(k\) using BERT with density = 1.0%.

![Figure 8](image2.png)

**Figure 8.** Weak scaling of training VGG-16 on Cifar-10 with density = 2.0%.
model in Table 1). On 32 GPU nodes, Ok-Topk 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 with density = 2.0% training for 160 epochs. On both 16 and 32 GPUs, Ok-Topk 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, Ok-Topk has a better scalability than the counterparts. On 64 GPUs, Ok-Topk 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, Ok-Topk 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]. How to tune hyperparameters for better accuracy with large batches is not the topic of this work. Surprisingly, on 64 GPUs, Ok-Topk, Gaussiank, TopkA and TopkDSA achieve
Therefore, we focus on pre-training in evaluation various downstream tasks. Pre-training is commonly much smaller than fine-tuning. The model is usually pre-trained on a large dataset and then fine-tuned for the dense allreduce for BERT pre-training. Compared with DenseOvlp, Ok-Topk reduces the total training time on 32 GPUs from 150 hours to 47 hours (more than 3x speedup), and also outperforms Gaussiank 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., Gaussiank, with the highest training throughput among all baselines, and DenseOvlp, a lossless approach). Since the other baselines are inferior to Gaussiank 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 Ok-Topk by comparing it with these two important baselines in Figure 13.

6 Conclusion
Ok-Topk is a novel scheme for distributed deep learning training with sparse gradients. The sparse allreduce of Ok-Topk incurs less than 6k communication volume, which is asymptotically optimal and more scalable than the counterparts. Ok-Topk 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 Ok-Topk 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 Ok-Topk 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


