SparCML: High-Performance Sparse Communication for Machine Learning

Cedric Renggli
ETH Zurich

Saleh Ashkboos
IST Austria

Mehdi Aghagolzadeh
Microsoft

Dan Alistarh
IST Austria

Torsten Hoefler
ETH Zurich

ABSTRACT

Applying machine learning techniques to the quickly growing data in science and industry requires highly-scalable algorithms. Large datasets are most commonly processed "data parallel" distributed across many nodes. Each node's contribution to the overall gradient is summed using a global allreduce. This allreduce is the single communication and thus scalability bottleneck for most machine learning workloads. We observe that frequently, many gradient values are (close to) zero, leading to sparse of sparsifyable communications. To exploit this insight, we analyze, design, and implement a set of communication-efficient protocols for sparse input data, in conjunction with efficient machine learning algorithms which can leverage these primitives. Our communication protocols generalize standard collective operations, by allowing processes to contribute *arbitrary* sparse input data vectors. Our generic communication library, SparCML[1], extends MPI to support additional features, such as non-blocking (asynchronous) operations and low-precision data representations. As such, SparCML and its techniques will form the basis of future highly-scalable machine learning frameworks.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Computing methodologies** → **Distributed algorithms**; *Supervised learning*.

KEYWORDS

Sparse AllReduce, Sparse Input Vectors, Sparse AllGather

ACM Reference Format:
Cedric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. 2019. SparCML: High-Performance Sparse Communication for Machine Learning. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3295500.3356222

[1]Stands for Sparse Communication layer for Machine Learning, to be read as *sparse ML*.

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356222

1 INTRODUCTION AND MOTIVATION

Machine learning workloads are quickly becoming more demanding. The size of the trained models and with it the computation required for training grow with the current exponential growth of data availability. While small models used to train in minutes on laptops, and image recognition networks such as AlexNet required days on a GPU system, newer models such as BERT [17] would take more than one year to train on a single GPU [31]. Similarly, the sizes of the networks grow quickly from a handful of parameters for simple regression tasks to more than 200 MB for Alexnet to up to 340 million parameters, i.e., 11 GB with 32 bit precision, for the largest BERT network.

The arguably standard distribution strategy in machine learning is *data parallelism*, in which nodes partition the dataset, and maintain consistent copies of the set of model parameters computing a global sum, either with allreduce, or through a coordinator node, called a parameter server [35]. Here, we consider only allreduce due to the obvious scaling limitations of a parameter server. While it's relatively simple to scale the number of execution nodes to the thousands, the biggest bottleneck is the allreduce of the gradient values at each step. The size of this reduction is equivalent to the model size itself and it is not reduced when more nodes are used. When scaling to large numbers of nodes, the full parameter set, commonly hundreds of megabytes, must be summed globally every few microseconds.

Given the large impact of communication, significant effort has been invested into identifying scalable solutions. Virtually all major frameworks optimize for efficient communication [1, 13, 33, 44, 57], while GPU vendors are developing specific communication layers for this goal [40]. The research community proposed several communication-reduction techniques, such as *quantization* [4, 44, 45], *asynchronous communication* [59], *structured sparsification* [2, 18, 46], or *large batch methods* [20, 56]. However, scaling machine learning applications remains a complex process, requiring non-trivial insights. In our work, we focus on a largely unexplored aspect of scaling: *how to exploit sparsity in the global summation itself*. Our techniques build on established HPC techniques such as MPI and extend it for generic support or arbitrary sparse reductions.

Conceptual Contribution. We propose SparCML, a scalable, general communication library for machine learning. SparCML starts from the idea that, to reduce communication and synchronization cost, we can exploit *sparsity* and *relaxed consistency* in machine learning applications. In particular, individual nodes can compute with a partially *inconsistent* view of the parameters. The immediate system implication, which we exploit in SparCML, is that the updates which nodes wish to communicate are either *naturally*

sparse [51], or can be *sparsified in a principled manner*, without loss of convergence [2, 4, 18, 46].

Technical Contribution. Our thesis is that *exploiting sparsity and compression* should be standard when scaling machine learning applications. Surprisingly, support for efficient sparse communication or compression is currently neither available in standard communication libraries such as MPI [19], nor in specialized machine-learning communication libraries [40]. One possible reason is the fact that designing and implementing general sparse collective operations is non-trivial, as sparsity adds a new dimension to the already complex system trade-offs arising when implementing collective operations efficiently at scale [48].

We take on this challenge in SparCML. Our implementation is efficient both in theory and in practice: for some workload parameters, it can be shown to be within constant factors of optimal in terms of bandwidth and latency cost. At the same time, our implementation achieves order-of-magnitude speedups versus highly optimized *dense collective* implementations, or over naive sparse implementations, both in synthetic tests and in real application scenarios. SparCML has several additional features, such as efficient support for *reduced-precision collectives* and for *non-blocking* operations. For example, we can perform sparse reductions for gradient exchange at 4 bits of precision per coordinate, overlapping computation and communication.

Targets. Our main target applications are two large-scale distributed machine learning tasks: training of state-of-the-art deep neural networks and large-scale regularized classification tasks. Our target systems are multi-node computing clusters. We study two scenarios: the first is *supercomputing*, where nodes are connected by a high-powered, extremely well optimized network. The second scenario is *datacenters*, where the network is *relatively* slower, such as InfiniBand or Gigabit Ethernet.

Challenges. The main algorithmic contribution behind our layer is a set of techniques for implementing collective communication operations, such as allreduce sum, over a large number of nodes having input vectors that are *sparse*. The principal difficulty for designing and analyzing such algorithms lies in the unknown overlap of non-zero indices, and hence the size of the reduced result. We provide an adaptive set of techniques which can systematically handle all cases and their trade-offs. These algorithmic insights are backed by careful optimizations and additional system features. An additional challenge from the machine learning side comes with avoiding additional hyperparameter tuning in order to leverage sparsity—in our experiments, we find that this is possible, with a few notable exceptions.

Experimental Results. We validate SparCML on a range of benchmarks: 1) synthetic instances aimed to validate our analysis, 2) academic benchmark datasets and models, and 3) large-scale deployments for image classification and automated speech recognition (ASR). Synthetic benchmarks show that SparCML can bring order-of-magnitude speedups with respect to highly-optimized dense implementations, with limited overhead in the dense case. We incorporate SparCML into two machine learning frameworks: CNTK (developed by Microsoft) and MPI-OPT (developed by us). In the supercomputing deployment, SparCML can reduce end-to-end convergence time of a state-of-the-art network for natural language understanding by 6×. Further, it completes a large-scale URL classification task 31× faster than its Cray MPI-based variant, which however does not exploit sparsity. The speedups are more significant on less performant cloud networks.

For large-scale workloads, we investigated training CNNs on the ImageNet dataset [43], and training the LSTM networks powering the ASR component of a popular personal assistant. In the first scenario, we found that SparCML was able to reduce the end-to-end training time for wide residual networks [58] on ImageNet by $\simeq$ 2× on 64 GPUs, with relatively negligible accuracy loss (< 0.5% Top5 validation), and no additional hyperparameter tuning. However, gains were negligible when applied to the standard ResNet50 benchmark [20, 23], which has fewer parameters and is therefore less amenable to training via sparse gradients. In the ASR task, SparCML reduced the training time for a state-of-the-art LSTM model on 128 GPUs by almost 10× (from 14 days to 1.78 days), without significant accuracy loss. Our conclusion is that SparCML can yield non-trivial speedups on a variety of machine learning applications, and that existing frameworks can significantly leverage sparsity and relaxed consistency guarantees.

2 PRELIMINARIES

Notation. Throughout this paper, we use the following notation for input parameters:

Variable	Description
P	Number of nodes
N	Problem dimension
p_i	Node i, $1 \leq i \leq P$
H_i	Set of non-zero indices which p_i wishes to communicate
k	Max number of non-zero (nnz) elements: $\max_i \lvert H_i \rvert$
$\mathcal{K}$	Total nnz in global sum: $\lvert \cup_{i=1}^{P} H_i \rvert$
d	Density of non-zero elements: $\frac{k}{N}$

2.1 Data Parallelism and Communication Costs

Data-parallelism is a standard distribution strategy for machine learning algorithms [1, 57]: P computing nodes share a large dataset and each maintains its own copy of the model $\vec{x}_t$. Model copies are kept in sync across nodes by exchanging the model updates computed locally between nodes, either via global averaging of updates, or through a central coordinator [35]. Specifically, in Stochastic Gradient Descent (SGD), each node i has access to (part of) the dataset, and, in each *iteration*, it processes a randomly chosen set of samples (a *mini-batch*), and computes a model update (gradient) $\nabla F_i(\vec{x}_t)$ locally. Nodes then globally *sum* these updates, and apply them locally, resulting in the following standard SGD iteration

$$\vec{x}_{t+1} = \vec{x}_t - \eta \sum_{i=1}^{P} \nabla F_i(\vec{x}_t),$$

where $\vec{x}_t$ is the value of the model at time t, η is the learning rate, and ∇F_i is the *stochastic* gradient of the current model with respect to the set of samples processed at node i.[2] Since gradient updates are averaged globally at the end of every iteration, all nodes have a consistent version of the model. The trade-off is between the parallelism due to the fact that we are processing P times more samples per iteration given P nodes, and the additional communication

[2]For simplicity, the reader may think of the model $\vec{x}_t$ as a large array of parameters, and of the gradients $\nabla F_i(\vec{x}_t)$ as array of entry-wise updates.

Algorithm 1 SparCML Quantized TopK SGD at a node i.

Input: Stochastic Gradient $\nabla F_i(\cdot)$ at node i
Input: value K, learning rate α
Initialize $v_0 = \epsilon_0^i = \vec{0}$
for each step $t \geq 1$ **do**
  $acc_t^i \leftarrow \epsilon_{t-1}^i + \alpha \nabla F_i(v_{t-1})$ {accumulate error into a locally generated gradient}
  $\epsilon_t^i \leftarrow acc_t^i - \text{TopK}(acc_t^i)$ {update the error}
  $g_t^i \leftarrow \text{allreduce}(Q(\text{TopK}(acc_t^i)), \text{SUM})$ { sum (sparse) contribution from all nodes }
  $v_t^i \leftarrow v_{t-1}^i - g_t^i$ { apply the update }
end for

cost due to the sum reduction, necessary to maintain a consistent model. To reduce this overhead, several communication reduction techniques have been proposed.

2.2 Communication-Reduction Techniques

Structured Sparsification. Recent work proposes the following communication-reduced SGD variant which we call Top-k SGD [2, 18]: each node communicates only the k largest (by magnitude) components of its gradient vector $\nabla F(\vec{x}_t)$, instead of all values in the traditional method. Usually, k is fixed to represent some percentage of the components, which can be even lower than 1% [37]. This forces gradient sparsity at each node, although the chosen components may vary across nodes. The value of the components which are not chosen is *accumulated*, and added to the gradient vector of the next iteration. A precise description of this procedure can be obtained by following Algorithm 1, where the quantization function Q should be taken to be the identity.

Quantization. An orthogonal approach for reducing the communication cost of machine learning algorithms has been to *quantize* their updates, lowering the number of bits used to represent each value, e.g. [4, 16, 44, 52]. Mathematically, the resulting iteration can be represented as:

$$\vec{x}_{t+1} = \vec{x}_t - \eta \sum_{i=1}^{P} Q(\nabla F_i(\vec{x}_t)),$$

where $Q : \mathbb{R}^N \rightarrow \mathbb{R}^N$ is an element-wise quantization operator which reduces the precision of the gradients' data representation. Quantization techniques can also be shown to preserve convergence, as long as the quantization noise is zero-mean, but may slow down convergence due to added variance [4].

3 COMMUNICATION-REDUCTION: A CRITICAL VIEW

We now examine communication-reduction techniques in the context of large-scale deployments characteristic to super-computing or large-scale cloud computing.

Structured Sparsification. On the positive side, sparsification has been proven to preserve convergence even for non-convex objectives [5], and have been empirically shown, in the context of neural network training, to be able to allow nodes to send even less than 1% of their local gradient update, without losing convergence [37].

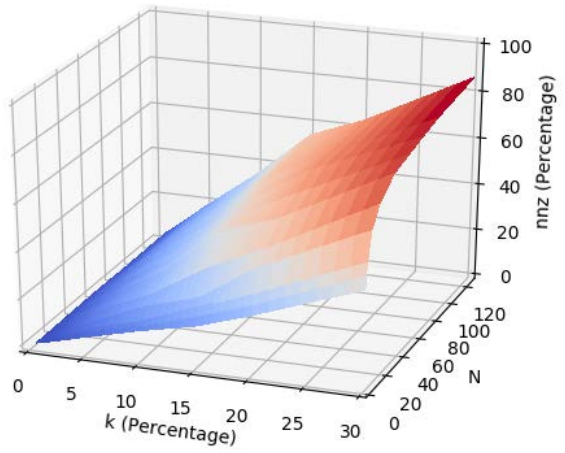


Figure 1: The density (in percentage) of the reduced result versus number of nodes N and per-node density k (in percentage) for the ResNet20 model trained on the CIFAR-10 dataset. The snapshot is taken at training epoch 5. Results are consistent across training stages (1st, 5th, and last epoch) and models (ResNet, DenseNet).

Unfortunately however, reaching high sparsity levels (above 99%) requires extremely careful tuning of momentum and learning rate hyperparameters, which is error-prone and time consuming. Employing *lower* sparsity levels–say, $5-10\%$ per node, which tends to be more stable with respect to hyper-parameter tuning–can negate the benefits of compression: in this case, reducing across a large number of nodes cans cause the reduced vector to become *dense*, at which point communication is again a bottleneck. We illustrate this issue in Figure 1, where we plot the density of the reduced gradient versus the number of nodes and the sparsity level at each node, on a standard CNN/dataset combination.

Quantization. These techniques do not have the issue that communication-compression is dependent of the node count: their compression rate is fixed and independent of the number of nodes. However, both theory and experiments suggest that quantization can only yield a limited amount of compression ($4-8\times$) before the added variance affects end accuracy [4, 22].

4 COMMUNICATION REDUCTION IN SPARCML

In sum, the previous section suggests that neither sparsification nor quantization is ideal in isolation, when considered at high node counts. In this context, the SparCML framework allows the user to leverage both quantization and sparsification methods. We provide efficient implementations of structured sparse methods (TopK SGD) via sparse collective operations, with non-blocking semantics, as well as an implementation of state-of-the-art quantization methods [4]. Importantly, the implementation of the reduction method natively supports both of these techniques to reduce communication and latency constraints.

Sparse Quantized Reduction. We now provide a high-level algorithmic description of the data-parallel SGD variant implemented in SparCML. Please see Algorithm 1 for pseudocode. Each node i maintains the residual error ϵ^i locally, which accumulates gradient components which did not get applied. Upon each step t, this gets added to the newly generated gradient, to obtain the accumulator acc_t^i. This accumulator is then truncated to obtain the value g_t^i to be sent by node i, and the new value of the error ϵ^i is generated.

The allreduce call sums all the truncated gradients in a sparsity-aware fashion. In particular, since the sum may become *dense*, SparCML may quantize the resulting dense vector at an intermediate stage of the reduction, in order to reduce the bandwidth overhead, using stochastic QSGD quantization [4]. We note that, even though sparsification and stochastic quantization have been introduced independently in the literature (see references [2, 18, 37] and [4, 44], respectively), we are the first to employ these two techniques in conjunction, and to prove that the resulting method provably converges.

Convergence Proof. The following result formally states the convergence guarantees of SparCML, under standard analytic assumptions on the objective function. The argument (provided in the appendix) is based on the convergence proof of TopK SGD [5]; the main source of novelty is the addition of stochastic quantization.

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

$$\min_{t\in\{1,2,\ldots,T\}} \mathbb{E}\left[\|\nabla f(x_t)\|^2\right] \overset{T\to\infty}{\to} 0.$$

Discussion. The above statement is quite general in that it covers a large class of non-convex objectives. However, it only proves *ergodic convergence* to a stationary point of expected zero gradient. This is weaker than proving convergence to a global minimum, but is in line with state-of-the-art results for this problem setting even without quantization, e.g. [36]. A second limitation shared by most theoretical results is that it does not provide a precise set of hyperparameters for practical deployments, beyond the indication that learning rates should be diminishing.

5 SUPPORTING SPARSITY IN SparCML

5.1 Data Representation: Sparse Streams

We now describe the data types used to store sparse and dense vectors, which we call *sparse streams*. Sparse streams allow for efficient computation and communication of the data. Our implementation is in `C++11`, and we follow this standard in our description. For simplicity, we focus on the case where the binary operation executed upon two or multiple streams is *summation*, but the same discussion would apply for other component-wise operations.

Initially, we assume that each node is assigned a subset of non-zero elements from a universe of size N. Let H_i denote the set of non-zero elements given at node p_i. We assume that these sets are *sparse* with respect to N, i.e., that $k = \max_i |H_i| \ll N$. We further denote by d_i the density of each set given by $d_i = \frac{|H_i|}{N}$ and define $d = \max_i d_i = \frac{k}{N}$.

We define the total number of non-zero elements after having performed the reduction as

$$\mathcal{K} = |\cup_{i=1}^{P} H_i|.$$

For simplicity, we ignore cancellation of indices during the summation and therefore get $k \le \mathcal{K} \le \min\{N, P\times k\}$.

Vector Representations. We start from the standard sparse representation, storing a sparse vector as a sequence of non-zero indices, together with the actual scalar values of each dimension. The stream is stored in an array of consecutive index-value pairs. The datatype of the values yields the number of bits needed for every non-zero value. We either work with single or double precision floating point values. We discuss lower precision support in Section 7.

Switching to a Dense Format. Although we are interested in *sparse* problems, the size and non-zero index distribution of the input vectors can be such that the algorithm may not benefit from the sparse representation after some intermediate point in the summation process: as the density of the intermediate result vector approaches the universe size N, the sparse representation becomes wasteful.

We can model the benefits of sparsity as follows: Let *isize* be the number of bytes needed to represent a non-zero input value and *nnz* the number of non-zero elements. We further define $c \ge \left\lceil \frac{\log_2(N)}{8} \right\rceil$ to be the number of bytes needed to store an index. Thus, the sparse format will transmit $nnz(c + isize)$ bytes while the dense format transmits $N \times isize$ bytes. Our sparse representation only reduces the communication volume if $nnz \le \delta = \frac{N\times isize}{(c+isize)}$. Yet, this volume estimation does not capture the fact that summing sparse vectors is computationally more expensive than summing dense vectors. Thus, in practice, δ should be even smaller, to reflect this trade-off.

It is safe to assume that the initial $nnz = k$ is smaller than this threshold. However, as the summation advances and number of nonzero elements *nnz* in the vector grows, the condition $nnz \le \delta$ may be violated. Especially for large node counts P, $\mathcal{K}$ is almost certainly larger than δ. To address this dynamic fill-in, we add an extra value to the beginning of each vector that indicates whether the vector is dense or sparse. In fact, when allocating memory for vectors of dimension N, we request $N \times isize$ bytes. It is therefore never possible to store more than δ sparse items. This threshold is used to automatically switch the representation.

Efficient Summation. The key operation is summing up two vectors u_1 and u_2, which could be either sparse or dense. To implement this operation efficiently, we distinguish two cases: (1) u_1 and u_2's indices draw from any position between 1 and N, and can potentially overlap and (2) u_1 and u_2's index sets are disjoint. The latter which arises if we partition the problem by dimension in which case we can implement the sum as simple concatenation.

If input indices can overlap, we distinguish the following cases depending on whether inputs are sparse or dense. Denote by H_1 and H_2 the sets containing the sparse indices of non-zero elements for the vectors u_1 and u_2, respectively.

If indices are overlapping, and both vectors are sparse, we first check whether the result might become dense. Theoretically, one needs to calculate the size of the union of non-zero indices $|H_1 \cup H_2|$. This is costly, and thus we only upper bound this result by $|H_1|+|H_2|$. The tightness of this upper bound will depend on the underlying

sparsity distribution, on which we make no prior assumptions. If this value is bigger than δ, we switch to a dense representation. If one of the inputs is dense, whereas the other is sparse, we iterate over all the index-value pairs stored in the sparse vector and set the value at the corresponding position in the dense vector. Finally, if both vectors are already dense, we simply perform a (vectorized) dense vector summation in either u_1 or u_2, and do not allocate a new stream.

5.2 Efficient Collectives on Sparse Streams

We now proceed to define collective operations over a set of sparse vectors located at the nodes. We focus on allgather and allreduce as defined by the MPI specification [21]. We support arbitrary coordinate-wise associative reduction operations for which a neutral-element can be defined. (By *neutral* we mean that the element which does not change the result of the underlying operation, e.g., 0 for the sum operation.)

Analytical Model. We assume bidirectional, direct point-to-point communication between the nodes, and consider the classic Latency-Bandwidth (α-β) cost model: the cost of sending a message of size L is $T(L) = \alpha + \beta L$, where both α, the latency of a message transmission, and β, the transfer time per word, are constant. L represents the message size in words. When sending sparse items, let β_s be the transfer time per sparse index-value pair and $\beta_d < \beta_s$ the time per word.

Given this setting, the goal is to perform a collective operation over the elements present initially at every node. That is, each node should obtain the correct result locally, i.e., the element-wise sum over the N dimensions in the allreduce case, while minimizing the total communication costs, measured in the α-β model.

Assumptions. For simplicity, we will assume that each node initially has k elements: $\forall i : |H_i| = k$; P is a power of 2, $P > 4$; and N is divisible by P. We discuss these assumptions and relax them in the supplementary material.

5.3 Communication Algorithms

We proceed with the composition of sparse streams into sparse collective communication algorithms. For this, we modify two existing dense algorithms to efficiently work with sparse streams. In an allreduce operation, each node i has a vector $x_i \in \mathbb{R}^N$ and the operation computes the element-wise sum $x = \sum_{i=0}^{N} x_i$ of all distributed vectors such that a copy of x is available at each node after the operation.

Allreduce can be implemented in many ways, for example, the nodes could collaborate to compute the result at a single node (reduce) followed by a broadcast or each node sends its x_i to all nodes (allgather). If the vector x_i is dense (the traditional case), the algorithms are well understood and implemented in MPI libraries [29]. However, we now explore the case where x_i is sparse such that not all elements need to be sent. We do not assume global information, i.e., none of our algorithms requires knowledge about the amount of data contributed by each node, nor about the distribution of non-zero indices.

Yet, we require the user to have some rough idea about $\mathcal{K}$, the final size of the result. This is often easily observable and we will differentiate two types of instances: In *static sparse allreduce* (SSAR), $\mathcal{K}$ remains below δ, such that we will never switch to a dense representation. Conversely, in *dynamic sparse allreduce* (DSAR) instances, where $\mathcal{K} \geq \delta$, we will start with a sparse and switch to a dense representation at some point during the collective operation.

If we assume that the number of non-zero indices is identical on all nodes, we can distinguish two extreme cases: (1) none of the zero elements overlap, i.e., $H_i \cap H_j = \emptyset \; \forall i, j$ and (2) all elements overlap fully, i.e., $H_i = H_j \; \forall i, j$. The first case is the case of maximum fill-in, at the end, x will have kP non-zero elements. If it would be known that no elements overlap, then the sparse allreduce could be implemented efficienlty with a simple allgather operation because no computation is necessary. Similarly, the second case is equivalent to a dense allreduce of size k. Any other possible distribution of non-zero indices lies in between these two extremes. We can now bound the communication time from above in case (1) as $\log_2(P)\alpha + (P-1)k\beta_d$ [10] and from below in case (2) as $\log_2(P)\alpha + 2\frac{P-1}{P}k\beta_d$ [10]. We note that the latter communication bound is only valid for negligible computational cost.

Lemma 5.1. *The time T for sparse allreduce is bounded by $T \geq \log_2(P)\alpha + (P-1)k\beta_d$ if $\mathcal{K} = kP$, and $T \geq \log_2(P)\alpha + 2\frac{P-1}{P}k\beta_d$ assuming that $\mathcal{K} = k$ and computation for reduction is perfectly parallelized.*

In practice, allreduce implementations switch between different implementations depending on the message size and the number of processes [48]. We distinguish between two cases: small messages and large messages on a moderate number of processes.

5.3.1 The Small Data Case. When the overall reduced data is small, latency dominates the bandwidth term. In this case, we adopt a *recursive doubling* technique: in the first round, nodes that are a distance 1 apart exchange their data and perform a local sparse stream reduction. In the second round, nodes that are a distance 2 apart exchange their reduced data. Following this pattern, in the t-th round, nodes that are a distance 2^{t-1} apart exchange all the previously reduced $2^{t-1}k$ data items. This behavior is illustrated in Figure 2. The recursive doubling technique can also be used for solving *dense* allreduce and allgather problems [29].

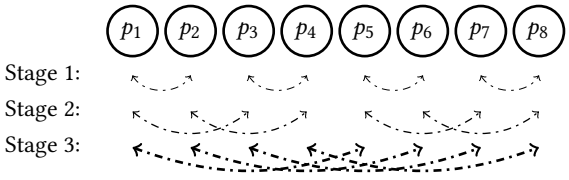


Figure 2: Static Sparse allreduce: Recursive doubling - Increasing amount of sparse data in every stage

The resulting latency for the SSAR_Recursive_double algorithm is $L_1(P) = \log_2(P)\alpha$, as there are $\log_2(P)$ stages. This is latency-optimal and data-independent. The bandwidth term varies with the sparsity pattern between the extremes discussed above:

$$L_1(P) + \log_2(P)k\beta_s \leq T_{ssar\_rec\_dbl} \leq L_1(P) + (P-1)k\beta_s.$$

The lower bound is reached when the k indices fully overlap. Therefore, at every stage, k items need to be transmitted as the intermediate results maintain constant size. The upper bound is given when

the indices do not overlap at all. Therefore, at stage t, the number of items transmitted is $2^{t-1}k$. Taking the sum, we get

$$\sum_{i=1}^{\log_2(P)} 2^{i-1}k = k\frac{2^{\log_2(P)}-1}{2-1} = k(P-1).$$

5.3.2 The Large Data Case. When the data is large, dense allreduce implementations make use of Rabenseifner's algorithm [42], which has two steps: The first is a Reduce-Scatter step, that partitions the result vector across nodes, assigning a partition to each node. This is implemented by a *recursive halving* technique [42]. In the second step, the reduced answers are gathered to all other nodes by calling a recursive doubling algorithm as described above. This two step algorithm has a total runtime of

$$T_{ar\_rab} = 2\log_2(P)\alpha + 2\frac{(P-1)}{P}k\beta_s,$$

reaching the lower bound on the bandwidth term and off by a factor 2 on the latency term.

Our sparse allreduce for large data is inspired by this dense algorithm. It split the execution into two steps: a *split* phase and a *sparse allgather* phase. In the *split* phase, we uniformly split the space dimension N into P partitions and assign to each node the indices contained in the corresponding partition. We split each sparse vector at its node and directly send each subrange of indices to the corresponding recipient in a *sparse* format. This direct communication comes at a higher latency cost, which we mitigate by using non-blocking send and receive calls. Each node then reduces the data it received and builds the result for its partition. In the second phase, the data has to be gathered to all other nodes with a simple (concatenating) sparse allgather.

Obtaining runtime bounds for SSAR_Split_allgather is slightly more involved. The *split* part takes time

$$(P-1)\alpha + 0\beta_s \leq T_{split} \leq (P-1)\alpha + k\beta_s.$$

Notice that both extremes imply that each node has k items for the sparse allgather and thus $\mathcal{K} = kP$. For this second step in the algorithm to be optimal, every node must have an intermediate result of size $\frac{k}{P}$, as we want the final result to have a size $\mathcal{K} = k$ and the communication to be equally distributed.

For every node to have an intermediate result of the desired size, we know that each node has to send at least $\frac{P-1}{P}k$ items to other nodes. Otherwise, if every node has exactly k items, we reach the upper bound for the result size of $\mathcal{K} = k \times P$. So we get

$$L_1(P) + \frac{P-1}{P}k\beta_s \leq T_{sparse\_ag} \leq L_1(P) + (P-1)k\beta_s.$$

The algorithm latency is again data-independent:

$$L_2(P) = (P-1)\alpha + L_1(P).$$

Combining these terms yields

$$L_2(P) + 2\frac{P-1}{P}k\beta_s \leq T_{ssar\_split\_ag} \leq L_2(P) + Pk\beta_s.$$

5.3.3 The Dynamic Case: Switching to Dense. ***Analysis.*** The discussion so far focused on the case where maintaining a sparse representation is efficient. However, as we gather data, the size of the result $\mathcal{K}$ might become larger than the sparsity-efficient threshold δ, in which case we switch to a dense representation. We call this the *dynamic* version of the problem (DSAR). The first result regarding this case is negative: the bandwidth savings due to sparsity are limited to a constant improvement relative to the dense case.

Assume the final size of the reduction result $\mathcal{K}$ is larger than the threshold δ, where a sparse representation is efficient. Let $\mathcal{K} \geq \delta = \kappa N$ be the size of the final reduction, which is too large to allow for a sparse representation (e.g. $\delta \log N \geq 1$). The algorithm will therefore switch to a dense representation at some point during the reduction operation. Additionally, we want to avoid unnecessary computation and, following [10], we assume equally distributed optimal computation among the nodes during the reduction process. We further know that every node has to send k elements to at least one other node and receive at least the other $\delta - k$ items of the dense final result. Following Chan et al. [10] we can prove the following claim:

Lemma 5.2. *Any algorithm solving the DSAR problem needs at least time* $\log_2(P)\alpha + \delta\beta_d$*, where the lower bound on the bandwidth required is at least* $\frac{1}{2}\kappa$ *that of any bandwidth-optimal fully dense allreduce algorithm, with* $\kappa = \frac{\delta}{N}$.

Proof. The optimal latency term is identical to the fully dense allreduce lower bound given by Chan et al. [10]. The fully dense allreduce with $k = N$ has a lower bound of $2\frac{P-1}{P}N\beta_d$ on the bandwidth, if computation is equally distributed. Based on the previous assumptions, the DSAR problem has a minimum bandwidth term of $\delta\beta_d$, which yields the $\frac{1}{2}\kappa$ factor as a lower bound. □

Algorithm. Based on these insights, our solution for DSAR adapts the previous two-stage algorithm to exploit the fact that every reduced split will become dense. DSAR_Split_allgather hence receives the data in a sparse format from all the other nodes in the first phase, then switches the representation and performs a dense allgather in the second stage. Here, we can leverage existing implementations, which are highly optimized to perform this second step with dense data. Based on the known times needed by those algorithms, which are obviously independent of the input density, we derive the running time for our algorithm given both extremes. The latency is again $L_2(P)$. Combined, we get

$$L_2(P) + \frac{P-1}{P}N\beta_d \leq T_{dsar\_split\_ag}$$

and

$$T_{dsar\_split\_ag} \leq L_2(P) + k\beta_s + \frac{P-1}{P}N\beta_d.$$

Another interesting observation following Lemma 5.2 is the fact, that by exploiting sparsity alone, and if the end-result is not efficiently storable in a sparse format compared to the dense representation, the achievable speedup of a sparse allreduce is at most $\frac{2}{\kappa}\times$ (with $\kappa = 0.5$, this yield a max speedup of 4×) compared to a fully dense algorithm. Other representation reduction techniques are needed in order to achieve higher speedups.

6 SUPPORTING LOW-PRECISION COMMUNICATION

In the previous sections, we showed that the bandwidth cost in the dynamic case is lower bounded by a constant fraction of the

bandwidth cost of a dense reduction, and we provided experimental evidence in the corresponding section that this case is in fact likely in large-scale deployments.

Therefore, to further reduce bandwidth cost, SparCML supports lower-precision data representation for the outputs (2, 4, and 8 bits per entry), using stochastic quantization as defined in the QSGD scheme [4]. This quantization scheme provably preserves the convergence of the SGD algorithm. Due to space constraints, we only present an outline of the scheme and its implementation.

In brief, to implement QSGD quantization, each (dense) stream is split into *buckets* of size B (in the order of 1024 consecutive entries) and each bucket is quantized independently and stochastically to the given number of quantization levels. Thus, each bucket corresponds to B *low-precision data items*, e.g., 4-bit integers, packed to reduce space and a full-precision *scaling factor*, which is used to provide a scale to all the entries in the bucket. We focus on low-precision to reduce the bandwidth cost of the *dense* case. In practice, we employ the low-precision data representation only in the second part of the DSAR_Split_allgather algorithm, where the data becomes dense. This allows us to reduce the bandwidth cost of this last step by a constant corresponding to the quantization.

7 ARTIFACT AND ADDITIONAL FEATURES

Interface and Code. The SparCML library provides a similar interface to that of standard MPI calls, with the caveat that the data representation is assumed to be a sparse stream. Given this, the changes needed to port MPI-enabled code to exploit sparsity through SparCML are minor. The library implementation consists of around 2,000 lines of native `C++11` (This does not include infrastructure such as benchmarks and tests which raises the line count by an order of magnitude). Adding SparCML to CNTK required changing around 100 lines of code.

Non-Blocking Operations. We also implement the previous algorithms in a *nonblocking way*, similar as specified for nonblocking collectives in MPI-3 [27, 28]. Specifically, we allow a thread to trigger a collective operation, such as allreduce, in a nonblocking way. This enables the thread to proceed with local computations while the operation is performed in the background [26].

MPI-OPT. MPI-OPT is a framework we developed from scratch to run distributed optimization algorithms such as SGD. It is written in native `C++11`, and can link external libraries such as SparCML and MPI for communication. MPI-OPT implements parallel stochastic optimization algorithms, like gradient and coordinate descent, on multiple compute nodes communicating via any MPI library, with low overhead. It implements efficient distributed partitioning of any dataset converted in the predefined format using MPI-IO, data-parallel optimization on multiple compute nodes, with efficient multi-threading inside each node, parametrized learning rate adaptation strategies, as well as customizations to use SparCML as the communication layer between nodes allowing for sparse, dense, synchronous, and asynchronous aggregation.

The Microsoft Cognitive Toolkit (CNTK). For large-scale neural network training, we modify CNTK [57] v2.0 to use SparCML as its communication layer. CNTK is a computational platform optimized for deep learning. The general principle behind CNTK is that neural network operations are described by a directed computation graph, in which leaf nodes represent input values or network parameters, and internal nodes represent matrix operations on their children. CNTK supports and implements most popular neural network architectures. To train such networks, CNTK implements stochastic gradient descent (SGD) with automatic differentiation. The CNTK baseline supports parallelization across multiple GPUs and servers, with efficient MPI-based communication.

8 EXPERIMENTS

Setup. We now validate SparCML on real world applications and synthetic experiments. Open code and experimental logs are available in a public repository [3]. Our experiments target two scenarios: supercomputing and cloud computing. For the first setting, we execute on the CSCS Piz Daint supercomputer [9], with Cray XC50 nodes, each of which has a 12 cores HT-enabled Intel Xeon E5-2690 v3 CPU with 4GB RAM and an NVIDIA Tesla P100 16GB GPU. Piz Daint is currently the most powerful supercomputer in Europe and has a high-performance Cray Aries interconnect with a Dragonfly network topology. We use multiple nodes using relatively older NVIDIA K80 GPUs connected through Gigabit Ethernet to simulate a standard cloud deployment, but ensuring no background traffic. We perform additional tests on a distinct cluster called Greina, with CX50 nodes and an InfiniBand FDR or Gigabit Ethernet interconnect and on a production-grade GPU cluster, described in the corresponding section.

In all our experiments, the baseline will be the MPI allreduce implementation on the fully dense vectors. In general we make use of the default Open MPI installation. On Piz Daint, we compare against the custom Cray-MPICH installation, highly optimized by Cray. Since our problems usually have dimension $N > 65$K, we fix the datatype for storing an index to an `unsigned int`.

8.1 Micro-Benchmarks

We begin by validating our theoretical analysis on synthetic data, on the Piz Daint and Greina (GigE) clusters. We vary the data dimension N and the data density d as well as the number of nodes P. Based on the defined density, k indices out of N are selected uniformly at random at each node and are assigned a random value. We run our sparse allreduce algorithms in order to validate both correctness and the relative ordering of the derived analytical bounds. The choice of parameters is realistic (we pick data dimensions corresponding to common layer sizes in neural networks).

For readability, graphs are in a log-log scale. As execution times are non-deterministic, we conduct five experiments with newly generated data, while running each one for ten times. Based on those 50 resulting runtime values, we state the 25 and 75 percentage quantiles. Results showing reduction times versus node count and density are given in Figure 3.

Following the theoretical analysis, we expect the variant SSAR_Recursive_double to perform best for a small amount of data, when latency dominates over the bandwidth term. At higher node count P, data becomes larger, which leads to less improvement of the algorithm SSAR_Recursive_double at the same number of non-zero entries over the other variants. Furthermore, the algorithm SSAR_Split_allgather dominates over the DSAR_Split_allgather

[3] Code and experimental logs: http://gitlab.com/rengglic/sparcml

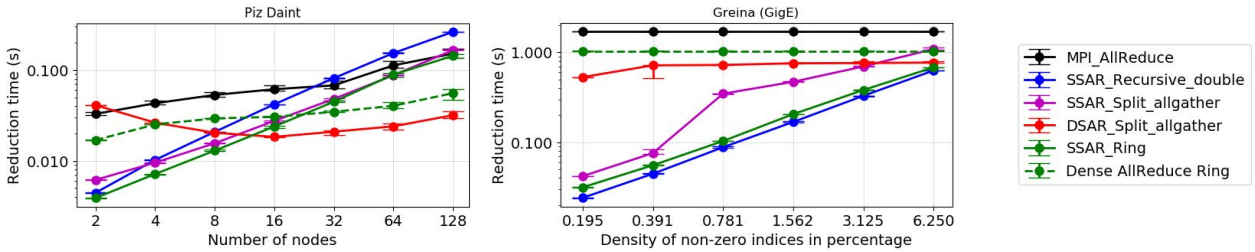


Figure 3: Reduction time versus number of nodes on Daint ($N = 16$M and $d = 0.781\%$), and reduction time versus data density on Greina ($N = 16$M and $P = 8$) for various algorithms.

variant as long as the number of non-zero indices is relatively low compared to the overall reduced size. Both facts are visible in Figure 3, where we notice that the difference between SSAR_Recursive_double and SSAR_Split_allgather is larger, when increasing the number of nodes compared to increasing density. To show the impact of the network on performance, we did run identical tests on both Piz Daint and Greina (GigE) as in Figure 3 on the right. The relative ordering remains comparable requiring less overall reduction time on high performance networks.

Additionally, the experiments in Figure 3 also compare our approaches against a ring-based MPI dense allreduce and its sparse counterpart. We note that, on a fast network and relatively small number of nodes, the ring-based algorithm is faster then any all other algorithms, but does not give any speedup at high number of nodes even at low density. As expected, DSAR_Split_allgather offers improvement even at a relative large number of nodes, but only up to a constant factor.

Name	# Classes	# of samples	Dimension
URL [38]	2	2 396 130	3 231 961
Webspam [51]	2	350 000	16 609 143
CIFAR-10 [34]	10	60 000	32x32x3
ImageNet-1K [43]	1000	1.3M	224x224x3
ATIS [24]	128	4 978 *s* / 56 590 *w*	-
Hansards [41]	-	948K *s* / 15 657K *w*	-

Table 1: Real World Application Datasets. *s* stands for sentences (or pairs) and *w* for words.

8.2 Large-Scale Classification

We use MPI-OPT to train linear classifiers (Logistic Regression, SVM) on large-scale classification datasets using SGD and stochastic coordinate descent (SCD). The goal is to examine the runtime improvements by just exploiting the sparsity inherently present in the datasets and algorithms. More precisely, in these experiments, we do *not* sparsify or quantize the gradient updates, but exploit the fact that data and hence gradients tend to be sparse for these tasks. The datasets are specified in Table 1. We examine the standard URL and Webspam high-dimensional binary classification datasets.

For SGD, the samples have high sparsity since the features are trigrams: while many such combinations exist, an item, e.g., a sentence, can only have a very limited set of them present. This is extremely common in text-based datasets. Since we are executing a task having a linear dependency between model and feature vector, this implies that the *gradients themselves* will be sparse. Since communication is lossless, convergence is preserved and we only report speedup of the communication and overall training time. We run SGD with large batches ($1,000 \times P$) for various combinations. The achieved speed of MPI-OPT with the best sparse reduction algorithm is reported in Table 2. (Communication speedup is reported in brackets.)

Additionally, we run MPI-OPT's SCD implementation, which follows the distributed random block coordinate descent algorithm of [53]. We focus on optimizing directly the primal problem in order to showcase the usage of SparCML on other algorithms, ignoring the fact that more sophisticated algorithms solving the dual problem might exist [32]. We run the optimization on the logistic regression loss function for the URL dataset distributed on 8 nodes of Piz Daint to achieve identical convergence compared to SGD. Every node contributes 100 coordinates after every iteration. As the values calculated by each node lie in different slices of the entire model vector, we compare the runtime of an sparse allgather from SparCML to its dense counterpart. MPI-OPT with a dense allgather has an average epoch time of 49 seconds, with 24 seconds dedicated to communication. The sparse allgather executes a dataset pass (epoch) in 26 seconds on average, with 4.5 seconds spent in the communication layer. This implies an overall speedup of factor **1.8**×, due to a **5.3**× speedup in communication time.

Comparison with Apache Spark. As an exercise, we also compare MPI-OPT with Apache Spark v1.6, which is officially supported by CSCS [8]. Comparison is performed on the same datasets; Spark uses its own communication layer which does not exploit sparsity.

On Piz Daint, using 8 nodes, MPI-OPT with SparCML reduces the time to convergence on the URL dataset by 63×. This is largely due to the reduction in communication time, which we measure to be of 185×. Concretely, the average epoch time is reduced from 378 seconds, with 319 seconds spent for communication, to an average of 6 seconds per epoch, whereof 1.7 seconds represent the communication time. Compared to Spark, MPI-OPT with the standard Cray-optimized *dense* allreduce has a 31× speedup to convergence, due to a 43× speedup in communication time. An epoch is executed in 13 seconds on average, with 8.6 seconds spent on communication. We further investigated these speedups on an 8-node research cluster with a Gigabit Ethernet interconnect. Using MPI-OPT, the average training time per epoch drops from 1,274 seconds (Spark) to 14 seconds (86×). On the communication part, the time per epoch drops from 1,042 seconds to 6 seconds. The communication time and overall speedup of a *dense* allreduce over Spark's communication layer are both of factor 12×.

System	Dataset	Model	# of nodes	Baseline Time (s)	Algorithm	Algo. Time (s)	Speedup
Piz Daint	Webspam	LR	32	24.0 (21.6)	SSAR_Recursive_double	6.8 (3.5)	**3.53 (6.17)**
		SVM	32	16.2 (14.2)		6.5 (4.4)	**2.49 (3.23)**
Piz Daint	URL	LR	32	26.4 (25.8)	SSAR_Recursive_double	7.5 (7.0)	**3.52 (3.69)**
		SVM	32	19.8 (19.3)		5.6 (5.3)	**3.54 (3.64)**
Piz Daint	Webspam	LR	8	46.7 (37.9)	SSAR_Split_allgather	25.6 (15.8)	**1.82 (2.40)**
	URL	LR	8	37.7 (35.3)		20.9 (15.0)	**1.80 (2.35)**
Greina (IB)	Webspam	LR	8	65.2 (46.7)	SSAR_Split_allgather	36.3 (19.0)	**1.80 (2.46)**
	URL	LR	8	81.4 (44.7)		61.1 (24.9)	**1.33 (1.80)**
Greina (GigE)	Webspam	LR	8	768.0 (759.5)	SSAR_Split_allgather	37.9 (29.5)	**20.26 (25.75)**
	URL	LR	8	1045.0 (1004.6)		80.26 (42.2)	**12.65 (23.81)**

Table 2: Distributed optimization using MPI-OPT. The times are averages for a full dataset pass, with the communication part in brackets. Speedup versus dense MPI is shown end-to-end, with communication speedup in brackets.

Discussion. The Spark comparison should be taken with a grain of salt, since Spark implements additional non-trivial features, notably *fault-tolerance*. However, we believe our results show conclusively that sparsity support can provide significant savings in this large-scale classification scenario, where sparsity is naturally present.

8.3 Training Deep Neural Networks

In this section, we examine the applicability of SparCML for distributed training of deep neural networks in CNTK, on academic datasets. (We present results on a larger tasks in the next section.) To exploit sparsity, we implement the Top-k SGD algorithm [2, 18, 46] with low-precision support. The resulting protocol is provided in Algorithm 1. We execute three types of tasks: *image classification* on the CIFAR-10 dataset, *natural language understanding* on the ATIS corpus and *machine translation* on the Hansards dataset. (See Table 1 for details.) For vision, we train the ResNet-110 architecture [23]. For natural language understanding and machine translation we use an encoder-decoder network consisting of two LSTM [25] cells each. We use the default hyper-parameters for single-GPU 32-bit full accuracy convergence in all our experiments, as provided in the open-source CNTK 2.0 repository [39]. For completeness, these parameters are provided in the Supplementary Material. For CIFAR-10 we select $k = 8$ and 16 entries from every bucket of 512 consecutive elements ($\sim 3\%$ density), and stochastically quantize the values to 4-bit precision. For ATIS we select $k = 2$, and $k = 4$ for Hansards, entries out of each bucket of 512 ($\sim 0.4\%$ and $\sim 0.8\%$ density), using no additional quantization strategy. Top-k selection and quantization are implemented using optimized GPU kernels, and communication is done layer-wise using non-blocking calls; this ensures that the impact on overall computation is minimal ($< 1\%$).

To illustrate the bandwidth reduction, we note that the LSTM model we use for ATIS has approximately 20M parameters, which total approximately 80 MB in full precision, which would need to be transmitted every upon every minibatch. By contrast, the compressed gradient received by every node in SparCML totals less than 0.5 MB.

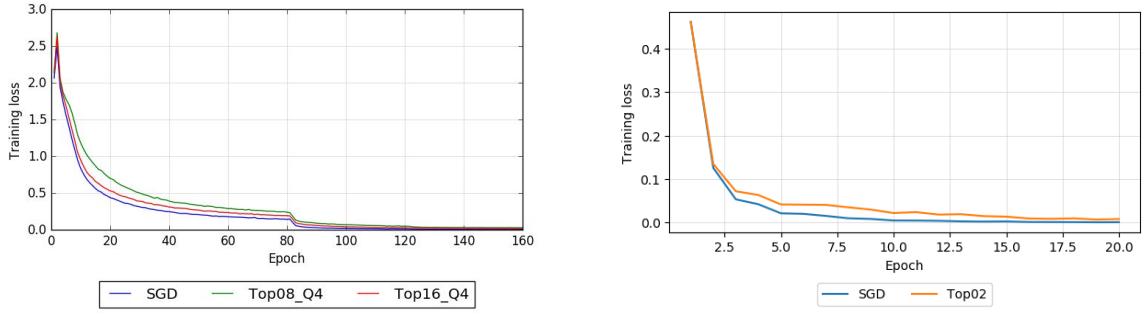
Accuracy & Speedup. The key metric we track is the *accuracy* of the converged models. For this, we note that on image classification (CIFAR-10), the model is able to recover virtually the same accuracy, both in terms of training and test error versus the number of epochs. Specifically, the end accuracy matches that of the full-precision baseline when selecting $k = 16$ out of every 512 elements, and for $k = 8/512$ the accuracy is 1% *above* the 32-bit variant as visible in Figure 4a. For both ATIS and Hansards tasks, training and test metrics (losses and BLEU scores) are within 1% of the full-precision baselines, as shown for ATIS in Figure 4b.

Examining end-to-end training speedup, on the CIFAR-10 task we achieve an overall speedup of factor 1.12× to full convergence with 8 nodes on Piz Daint versus the full-precision baseline. Training ATIS for 20 epochs, and Hansard for 5200 iterations (as standard), we are able to reduce the overall training time on Piz Daint by a factor 5.99× for ATIS, and 1.5× for Hansard respectively. The variance in these speedup numbers is explained by the varying ratios of communication and computation of the models: for the models we employ on CIFAR-10 and Hansards, computation dominates communication, whereas this ratio is inverted for ATIS, in which case reducing communication has a much larger impact on end-to-end training time.

8.4 Large Workload Experiments

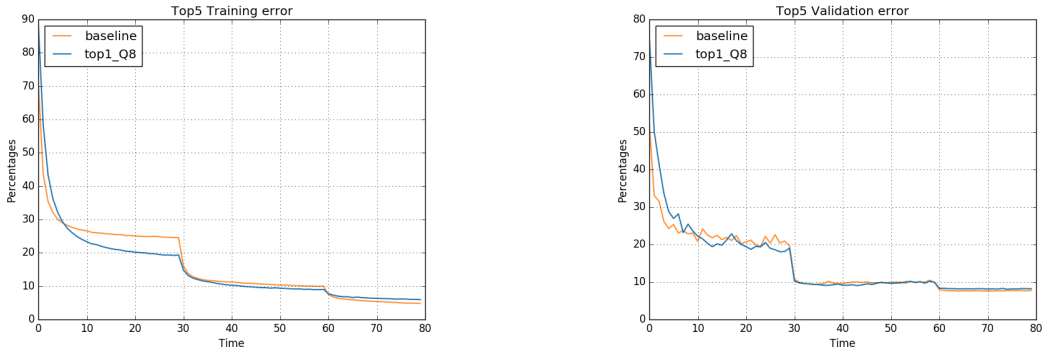
ImageNet Experiments. Our next experiment considers the applicability of our method in the context of large-scale image classification, in particular by training ResNet architectures on the standard ImageNet ILSVRC dataset, with 1000 target classes [43] (also known as ImageNet-1K). This experiment was executed on CSCS Piz Daint, using 64 compute nodes, each with a P100 GPU.

Our first target model is the classic ResNet50 architecture [23], totalling approximately 25 million parameters across 50 layers. This model scales well when using the baseline Cray MPI implementation, as it benefits from the relatively low number of parameters, and from the fact that per-layer gradient transmission can be overlapped via non-blocking calls. For this model, our results in terms of scaling are negative: the runtime improvements due to layer sparsification to 99% sparsity are of $\simeq 6\%$ (1950 seconds per epoch versus 2071 for the Cray MPI baseline). Our profiling revealed that this negative result is due to several factors, in particular that: (1) For this parameter setting, gradients become dense during aggregation, which limits our speedup. We found that enforcing higher sparsity levels hurts model convergence, even if we implemented techniques such as momentum correction and warm-up training [37] to alleviate this issue; (2) The overhead of sparsification and densification during TopK is non-negligible relative to the transmission cost of ResNet50 layers; (3) Our implementation does not benefit from the



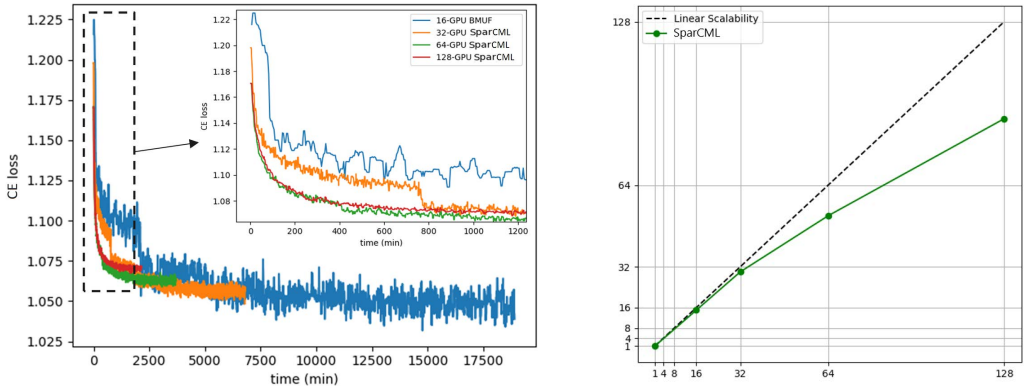
(a) Training Accuracy for ResNet-110 Model on CIFAR-10. (b) Training Accuracy LSTM Model on ATIS Dataset.

Figure 4: Train Accuracy for Sparsified (and Quantized) Versions Vs. Full Dense SGD.



(a) Top5 Training Error. (b) Top5 Validation Error.

Figure 5: Train and Validation Error for 4× Wide ResNet for the baseline (orange) versus TopK Quantized SGD implemented in SparCML.



(a) Accuracy versus training time numbers for 6 training passes over the entire dataset, recording training error (CE loss). Vali-dation results (word-error-rates) are discussed in the text.

(b) SparCML Scalability as a function of number of GPUs

Figure 6: Production Workload Speech Experiments.

additional parameter tuning of the proprietary Cray implementation. While items (2) and (3) can be addressed with additional implementation effort, item (1) strongly suggests that sparsification is not a universal solution for scaling to the very large node counts achieved by previous work for this type of CNN architecture [20, 56]: since we cannot scale the sparsity linearly with the number of nodes without hurting model convergence, gradients naturally will become dense at high node count, which limits the benefits of our method in this scenario.

Our second target model class for this task is *wide* residual models [58]. These models are variants of ResNet architectures, where the only difference is that the number of *channels* in each block is multiplied by a constant factor. It has been empirically found that shallow variants of wide models can achieve similar or better levels of accuracy as considerably deeper architectures [58], and that they are less sensitive to hyperparametrization, and in particular to large-batch training [12]. Due to their increased capacity, wide residual networks are popular when transferring to more complex tasks, such as ImageNet-10K and 22K [15]. In particular, we focus on training the 4xResNet18 and 4xResNet34 models (which have $4x$ the channels of their regular variants) on ImageNet-1K using TopK SGD, with $K = 1/512$, that is, on average only the top 0.2% of parameter values are transmitted by each node. Each P100 GPU can only process four images in a batch, leading to a global batch size of 512 images. We emphasize that we employ standard hyperparameter values for training these networks–besides theglobal batch size of 512 which is higer due to parallelization; in particular, our learning rate schedules are identical to the single-GPU case, and we perform no adjustments for sparsity, such as warmup or momentum correction.

Convergence results for 4xResNet18 are presented in Figure 5, for both training and validation accuracy. We notice that the final accuracy of the models differs by less than 0.9% in terms of top-1 accuracy, and less than 0.5% in terms of top-5 accuracy. At the same time, SparCML provides a speedup of $\simeq 2\times$ versus the Cray MPI baseline. Upon examination, this speedup is due almost entirely to the reduced aggregation time on the gradients on the last fully-connected layer of the network, which totals more than 2M parameters on this wide variant. The results are similar for 4xResNet34: the speedup is of approximately $\simeq 1.85\times$ versus the Cray MPI baseline, with accuracy difference of 0.8% in terms of top-1 accuracy, and less than 0.4% in terms of top-5 accuracy versus the fully-dense baseline. We note the faster loss reduction of TopK in the earlier stages of training, whereas the improvement saturates and inverts at the end of training. In sum, we conclude that gradient sparsity can indeed provide non-trivial speedups for wide residual networks, at the cost of a relatively minor decrease in accuracy, with no additional hyperparameter tuning.

Automated Speech Recognition. The final test of our framework is on a state-of-the-art acoustic model for automated speech recognition (ASR), powering a popular digital personal assistant inside Microsoft. The model we train is a state-of-the-art LSTM network with attention. The model has more than 60 million parameters, 2.4 million of which reside in the attention layer. We employ Top-k SGD for the training of the attention layer, starting from a pre-trained LSTM network. The dataset consists of approximately 30,000 hours (3.5 years) of annotated speech. Our cluster deployment consists of 32 server nodes, each with four NVIDIA V100 GPUs, totalling 128 GPUs. Servers have an InfiniBand interconnect, and aggregation inside each node is performed via NVIDIA NVLink with NCCL [40].

The baseline we compare against is training on 4 nodes, 16 GPUs in total, without sparsity or quantization, but employing a carefully-tuned instance of block-momentum SGD (BMUF) [11]. *Higher node counts for this full-precision variant led to negative scalability and, in some cases, divergence.* We note that this baseline already performs non-trivial communication reduction, since it communicates updates less frequently between nodes with respect to standard minibatch SGD. (Standard minibatch SGD is infeasible on our setup due to the large model size and node count.)

We execute six passes over the entire dataset and register the time to complete the experiment and the final accuracy. The 16 GPU BMUF baseline takes approximately 14 days to complete. This variant increases the batch size linearly with the number of nodes (weak scaling). We compare against our version of Top-k SGD with SparCML, in which gradients are split into groups of 512 consecutive coordinates, out of which we select the 4 largest ones, which we transmit from each group, saving the rest locally. We aim to leverage the fact that, in this production setting, most updates will occur in the parameters of the attention layer. When executing this variant, we tuned the initial learning rate, and the batch size; in particular, we keep a fixed global batch size of 512 samples, which is the same as for sequential training (strong scaling).

Figure 6a presents the results in error-versus-time format, where error is measured by standard cross-entropy (CE) loss, using our implementation, for 32, 64, and 128 GPUs. We highlight the fact that the sparse implementation is able to reach similar accuracy to the full-precision baseline in a fraction of the time: at 32 nodes (128 GPUs), we are able to reduce training time to < 1.8 days. Figure 6b illustrates the good scalability of the method. To further test accuracy, we also performed testing in terms of word-error-rate (WER) for the converged models, on validation sets. We found that the models trained with SparCML incur error rates that are less than 1% higher than full-precision (but unscalable) training and can sometimes *improve* accuracy by up to 1%. This trade-off is very advantageous for this application scenario, as it enables much faster model iteration times.

Hyperparameter Tuning. One important question regards the need for additional hyperparameter tuning when using the Quantized TopK algorithm. We note that, although we enforced sparse gradients, we have recovered accuracy under standard hyperparameter values even under high sparsity levels, in most cases. There are two notable exceptions: ResNet50 training, where high sparsity combined with large batch sizes induced significant accuracy loss, and the ASR experiment, where we have maintained a small global batch size to preserve convergence. These results suggest a non-trivial interaction between sparse gradients, batch size, and convergence, which we aim to investigate further in future work.

9 RELATED WORK

There has recently been a tremendous surge of interest in distributed machine learning [1, 13, 57]; see Ben-Nun and Hoefler [7] for a survey. In the following we focus on closely related techniques.

Reduced Communication Techniques. Seide et al. [44] was among the first to propose quantization to reduce the bandwidth and latency costs of training deep networks. More recently, Alistarh et al. [4] introduced a theoretically-justified distributed SGD variant called Quantized SGD (QSGD), which allows the user to trade off compression and convergence rate. We implement QSGD as a default quantization method. Dryden et al. [18] and Aji and Heafield [2] considered an alternative approach to communication reduction for data-parallel SGD, *sparsifying* the gradient updates

by only applying the top-k components, taken at at every node, in every iteration, for k corresponding to $< 1\%$ of the update size. Since then, other references [37, 46] explored this space, showing that extremely high gradient sparsity ($< 0.1\%$) can be supported by convolutional and recurrent networks with preserved accuracy, although maintaining accuracy requires hyperparameter tuning.

Our paper complements this line of the work by 1) considering stochastic quantization and sparsification in conjunction, and proving that the resulting technique still provably converges and is practically useful; 2) providing highly efficient sparsity and quantization support, with consistent runtime gains in large-scale settings, both for supercomputing and cloud computing scenarios.

Lossless Methods. Factorization is a lossless compression technique [14, 54] that is effective in deep neural networks with large fully-connected layers, but less applicable in networks with large convolutional layers, which are quite common [23, 47]. A second lossless method is executing *extremely large batches*, thus hiding the cost of communication behind larger computation [3, 20, 55, 56]. The compression methods in SparCML are orthogonal to this direction, as they aim to reduce bandwidth cost given a fixed batch size; as we have observed experimentally, sparsification can be applied with little additional tuning at a fixed batch size. However, when distributing training to large node counts, batches become large, and the aggregated gradients become dense, as we usually cannot scale sparsity up linearly with the node count. Thus, large-batch sparse-gradient hyperparameter tuning would become necessary in such cases, which we leave for future work. s We note that SparCML already implements several optimizations which are common in the large-batch setting, such as merging gradients for adjoining layers ("tensor fusion"), or non-blocking operations [55].

Communication Frameworks. Several frameworks have been proposed for reducing communication cost of distributed machine learning. One popular example is NVIDIA's NCCL framework [40], which significantly reduces communication cost when the nodes are NVIDIA GPUs and the proprietary NVLINK interconnect is available, which is not the case in multi-node settings, such as supercomputing. Further, NCCL currently only implements a very restricted set of reduction operations. In addition, there is a non-trivial number of frameworks customized to specific application scenarios, such as the Livermore Big Artificial Neural Network Toolkit (LBANN) [50] or S-Caffe [6]. While very efficient in specific instances, these frameworks do not usually leverage reduced-communication techniques, or sparsity.

Sparse Reduction. Hofmann and Rünger [30] propose a simple and effective runlength encoding approach for sparse reductions. We significantly extend this approach in the current work, including the observation that data might become dense during the reduction process and that an efficient and flexible data representation must be provided in this case. Träff [49] proposes a general approach for implementing sparsity in MPI by ignoring neutral elements in MPI reductions. Our sparse allreduce implementation could be seen as a special case of this general approach, where we precisely specify the reduction algorithms, and carefully analyze the performance bounds for small and large message scenarios. In addition, SparCML makes several additional contributions which are specific to machine learning applications, such as efficient low-precision support and integration with machine learning frameworks.

Kylix [60] considers sparse many-to-many reductions in the context of computation over large scale distributed graph data on community clusters. However, Kylix assumes knowledge of the data distribution and performs multiple passes over the reduction, which make it not applicable to our scenario. Dryden et al. [18] implement a sparse variant of the classical allreduce algorithm via a pairwise reduce-scatter followed by a ring-based allgather. The amount of data is kept constant at every stage of their algorithm by re-selecting the top k values and postponing the other received values. We note that this ability to preserve a local residual is specific to Top-k SGD and that our framework is more general. In terms of performance, their implementation will provide similar results to our SSAR_Split_allgather algorithm.

10 CONCLUSIONS AND FURTHER WORK

We have described and analyzed SparCML, a high-performance communication framework that allows the user to leverage sparse and low-precision communication in the context of machine learning algorithms. SparCML integrates easily into existing computational frameworks and can provide order-of-magnitude speedups in several real-world applications. In future work, we aim to further investigate other distributed machine learning applications which can benefit from sparsity, and to further investigate the interaction between sparsity and other parallelization approaches, such as large-batch training. We believe that the simple but effective sparsity schemes we described can play a significant role in reducing communication cost in future machine learning systems.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 805223), and from a Swiss National Supercomputing Centre Small Development Project (code d94). Torsten Hoefler was supported by the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880). We also thank Tal Ben-Nun, Salvatore Di Girolamo and the CSCS support team for their help with running experiments on Piz Daint.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

[2] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. *arXiv preprint arXiv:1704.05021* (2017).

[3] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. 2017. Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325* (2017).

[4] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Randomized Quantization for Communication-Efficient Stochastic Gradient Descent. In *Proceedings of NIPS 2017*.

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

[6] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. 2017. S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters. In *Proceedings of the 22nd ACM*

SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 193–205.

[7] T. Ben-Nun and T. Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR* abs/1802.09941 (Feb. 2018).

[8] CSCS Swiss National Supercomputing Centre. 2018. Apache Spark on the CSCS Cluster. https://user.cscs.ch/scientific_computing/supported_applications/spark/.

[9] CSCS Swiss National Supercomputing Centre. 2018. The CSCS Piz Daint Supercomputer. http://www.cscs.ch/computers/piz_daint.

[10] 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.

[11] Kai Chen and Qiang Huo. 2016. Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 5880–5884.

[12] Lingjiao Chen, Hongyi Wang, Jinman Zhao, Dimitris Papailiopoulos, and Paraschos Koutris. 2018. The effect of network width on the performance of large-batch training. In *Advances in Neural Information Processing Systems*. 9302–9309.

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[14] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *OSDI*, Vol. 14. 571–582.

[15] Valeriu Codreanu, Damian Podareanu, and Vikram Saletore. 2017. Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train. *arXiv preprint arXiv:1711.04291* (2017).

[16] Christopher De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. 2015. Taming the Wild: A Unified Analysis of Hogwild. *Style Algorithms. In NIPS* (2015).

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[18] Nikoli Dryden, Sam Ade Jacobs, Tim Moon, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*. IEEE Press, 1–8.

[19] Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0.

[20] 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).

[21] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk. 2014. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press.

[22] Demjan Grubic, Leo Tam, Dan Alistarh, and Ce Zhang. 2018. Synchronous Multi-GPU Training for Deep Learning with Low-Precision Communications: An Empirical Study. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.*, Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose (Eds.). OpenProceedings.org, 145–156. https://doi.org/10.5441/002/edbt.2018.14

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[24] Charles T Hemphill, John J Godfrey, George R Doddington, et al. 1990. The ATIS spoken language systems pilot corpus. In *Proceedings of the DARPA speech and natural language workshop*. 96–101.

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

[26] T. Hoefler and A. Lumsdaine. 2008. Message Progression in Parallel Computing - To Thread or not to Thread?. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society.

[27] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Implementation and performance analysis of non-blocking collective operations for MPI. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*. IEEE, 1–10.

[28] T. Hoefler, A. Lumsdaine, and W. Rehm. 2007. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM.

[29] T. Hoefler and D. Moor. 2014. Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations. *Journal of Supercomputing Frontiers and Innovations* 1, 2 (Oct. 2014), 58–75.

[30] Michael Hofmann and Gudula Rünger. 2008. MPI reduction operations for sparse floating-point data. *Lecture Notes in Computer Science* 5205 (2008), 94.

[31] Rani Horev. [n. d.]. BERT Explained: State of the art language model for NLP. https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270. visited, Mar. 2019.

[32] Martin Jaggi, Virginia Smith, Martin Takác, Jonathan Terhorst, Sanjay Krishnan, Thomas Hofmann, and Michael I Jordan. 2014. Communication-efficient distributed dual coordinate ascent. In *Advances in neural information processing systems*. 3068–3076.

[33] Nikhil Ketkar. 2017. Introduction to PyTorch. In *Deep Learning with Python*. Springer, 195–208.

[34] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).

[35] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server.. In *OSDI*, Vol. 1. 3.

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

[37] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv preprint arXiv:1712.01887* (2017).

[38] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. 2009. Identifying suspicious URLs: an application of large-scale online learning. In *Proceedings of the 26th annual international conference on machine learning*. ACM, 681–688.

[39] Microsoft. [n. d.]. CNTK Examples. https://github.com/microsoft/CNTK/tree/987b22a8350211cb4c44278951857af1289c3666/Examples. visited, July. 2019, commit 7882cf0.

[40] NVIDIA. 2016. The NVIDIA Collective Communications Library (NCCL). *https://developer.nvidia.com/nccl* (2016).

[41] Natural Language Group of the USC Information Sciences Institute. 2017. Aligned Hansards of the 36th Parliament of Canada. https://www.isi.edu/natural-language/download/hansard/

[42] Rolf Rabenseifner. 2004. Optimization of collective reduction operations. In *International Conference on Computational Science*. Springer, 1–9.

[43] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[44] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit Stochastic Gradient Descent and its Application to Data-parallel Distributed Training of Speech DNNs. In *Fifteenth Annual Conference of the International Speech Communication Association*.

[45] Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*.

[46] Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. 2017. meProp: Sparsified Back Propagation for Accelerated Deep Learning with Reduced Overfitting. *arXiv preprint arXiv:1706.06197* (2017).

[47] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning.. In *AAAI*. 4278–4284.

[48] Rajeev Thakur and William D Gropp. 2003. Improving the performance of collective operations in MPICH. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 257–267.

[49] Jesper Träff. 2010. Transparent neutral element elimination in MPI reduction operations. *Recent Advances in the Message Passing Interface* (2010), 275–284.

[50] Brian Van Essen, Hyojin Kim, Roger Pearce, Kofi Boakye, and Barry Chen. 2015. LBANN: Livermore big artificial neural network HPC toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 5.

[51] Steve Webb, James Caverlee, and Calton Pu. 2006. Introducing the Webb Spam Corpus: Using Email Spam to Identify Web Spam Automatically.. In *CEAS*.

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

[53] Stephen J Wright. 2015. Coordinate descent algorithms. *Mathematical Programming* 151, 1 (2015), 3–34.

[54] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* 1, 2 (2015), 49–67.

[55] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. 2019. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. *arXiv preprint arXiv:1903.12650* (2019).

[56] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling SGD Batch Size to 32K for ImageNet Training. *arXiv preprint arXiv:1708.03888* (2017).

[57] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. 2014. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112* (2014).

[58] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide residual networks. *arXiv preprint arXiv:1605.07146* (2016).

[59] Jian Zhang, Ioannis Mitliagkas, and Christopher Ré. 2017. YellowFin and the Art of Momentum Tuning. *arXiv preprint arXiv:1706.03471* (2017).

[60] Huasha Zhao and John Canny. 2014. Kylix: A sparse allreduce for commodity clusters. In *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE, 273–282.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Datasets and Tasks: We validate SparCML on real world applications and synthetic experiments. The real world applications are: - Training of linear classifiers (Logistic Regression, SVM) on standard URL and Webspam high-dimensional binary classification datasets - Training of ResNet-110 for image classicifation on CIFAR-10 - Training of a two cell LSTM for natural language understanding (on ATIS dataset) and machine translation (Hansards dataset) - Training of state-of-the-art LSTM network with attention representing an acoustic model for automated speech recognition (ASR). The proprietary dataset consists of approximately 30,000 hours (3.5 years) of annotated speech.

— Code: Complete code and experimental logs are available in a public repository and further described in the paper.

— Hardware, OS, and compiler details: We execute our experiment on the CSCS Piz Daint supercomputer, with Cray XC50 nodes, each of which has a 12 cores HT-enabled Intel Xeon E5-2690 v3 CPU with 4GB RAM and an NVIDIA Tesla P100 16GB GPU. Piz Daint has a high-performance Cray Aries interconnect with a Dragonfly network topology. We use multiple nodes using relatively older NVIDIA K80 GPUs connected through Gigabit Ethernet to simulate a standard cloud deployment, but ensuring no background traffic. The code is compiled using GCC 6.2.0 provided by Cray.

We perform additional tests on a cluster called Greina, with CX50 nodes and an InfiniBand FDR or Gigabit Ethernet interconnect and on a production-grade GPU cluster consisting of 32 server nodes, each with four NVIDIA V100 GPUs, totalling 128 GPUs. Aggregation inside each node of this specific cluster is performed via NVIDIA NCCL. We make use of the compiler GCC 5.4.0.

In all our experiments, the baseline is the default MPI allreduce implementation on the fully dense vectors. We make use of the default Open MPI 4.0 installation in general. On Piz Daint, we compare against the custom Cray-MPICH installation, highly optimized by Cray.

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: Some author-created data artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

```
Public repository with non-proprietary artifacts
↪ regarding code and generated logs available:
↪ https://gitlab.com/rengglic/SparCML
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Paper Modifications: See Artifact Description and details in the paper.