Efficient Sparse Collective Communication and its application to Accelerate Distributed Deep Learning

Jiawei Fei*
NUDT
KAUST

Chen-Yu Ho*
KAUST

Atal Narayan Sahu
KAUST

Marco Canini
KAUST

Amedeo Sapio
Intel

ABSTRACT

Efficient collective communication is crucial to parallel-computing applications such as distributed training of large-scale recommendation systems and natural language processing models. Existing collective communication libraries focus on optimizing operations for dense inputs, resulting in transmissions of many zeros when inputs are sparse. This counters current trends that see increasing data sparsity in large models.

We propose OmniReduce, an efficient streaming aggregation system that exploits sparsity to maximize effective bandwidth use by sending only non-zero data blocks. We demonstrate that this idea is beneficial and accelerates distributed training by up to 8.2×. Even at 100 Gbps, OmniReduce delivers 1.4–2.9× better performance for network-bottlenecked DNNs.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Computing methodologies** → **Machine learning**.

ACM Reference Format:
Jiawei Fei, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient Sparse Collective Communication and its application to Accelerate Distributed Deep Learning. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3452296.3472904

1 INTRODUCTION

Collective communication routines (or simply, collectives) are a core building block of parallel-computing applications. Collectives are commonly used to combine data among multiple processes performing operations in parallel. Achieving high-performance collective communication is paramount in virtually every scenario where an unfavorable computation to communication ratio restricts the ability to efficiently scale the workload.

One such scenario – also the focus of this paper – is distributed deep learning (DDL), which is now in widespread use to reduce the

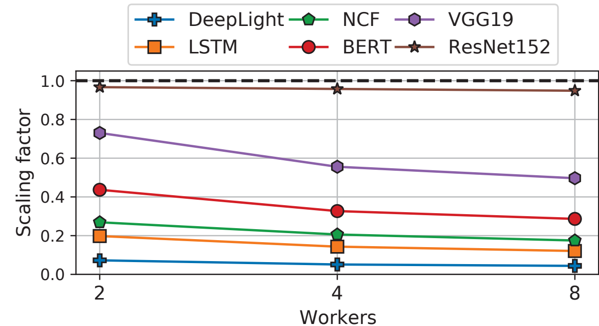


Figure 1: Scalability of six DDL workloads (cf. Table 1) as the number of workers increases in 10Gbps network. The y-axis shows the scaling factor (sf) defined as in [69]: $\frac{T_N}{NT}$ where T is the single GPU throughput and T_N is the measured throughput for a cluster with N workers. Linear scalability requires $sf = 1$ for any N. The experimental setup is in §6.

training time of large deep neural networks (DNNs) by parallelizing training over a large number of workers with GPUs or other AI accelerators (e.g., TPUs). The most common DDL approach is data-parallel training via stochastic gradient descent (SGD) [33]. Distributed SGD is a parallel, iterative workload with two steps: (1) every worker trains a local copy of the model by processing in parallel a different subset (mini-batch) of the training data; (2) all workers combine the results of their computation (i.e., the local gradient) to produce an average gradient that is applied to the model, prior to the next iteration.[1]

When distributing SGD on many workers, one can either keep the per-iteration total mini-batch constant (strong scaling) or linearly increase the mini-batch size with the number of workers (weak scaling). In the former case, the computation time decreases while the per-worker gradient size stays constant; therefore, training quickly becomes communication-bound. In the latter case, the computation to communication ratio (ideally) remains constant. However, in reality, communication time increases with the number of workers due to scaling overheads [35, 50, 64]. Moreover, a large mini-batch size can degrade training quality [31]. To enable better scaling, we aim to *decrease communication overheads by optimizing collective communication.* Figure 1 shows that these overheads are substantial in many DNN workloads, especially for large models where there exists a significant gap between the measured performance and ideal linear scaling.

*Equal contribution.

[1]Other DDL approaches are model-parallel, pipeline-parallel, and asynchronous data-parallel. These are not as common and out of scope.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM '21, August 23–28, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8383-7/21/08…$15.00
https://doi.org/10.1145/3452296.3472904

Model	Task	Dataset	Batch size	Dense weights	Embedding weights	Gradient sparsity	OmniReduce comm.
DeepLight [12]	Click-through Rate Prediction	Criteo 1TB [45]	2^{11}	1.8 MB	2.26 GB	99.73% (50 epochs)	**16 MB (0.7%)**
LSTM [29]	Language Modeling	GBW [8]	128	74 MB	1.52 GB	94.50% (50 epochs)	**90 MB (5.5%)**
NCF [22]	Recommendation	ML-20mx4x16 [19]	2^{20}	0.4 MB	679 MB	84.6% (30 epochs)	**280 MB (41%)**
BERT [13]	Question Answering	SQuAD [53]	4	1 GB	284 MB	9.31% (1 epoch *)	**1.13 GB (88%)**
VGG19 [61]	Image Classification	ImageNet-1K [56]	64	548 MB	–	32.0% (1 epoch *)	**547 MB (100%)**
ResNet152 [21]	Image Classification	ImageNet-1K [56]	64	230 MB	–	21.6% (1 epoch *)	**230 MB (100%)**

Table 1: Characteristics of benchmark DNN workloads. The table separates model size as dense and embedding weights (which are the weights in embedding layers of a DNN). The gradient has the same size as the model and the table lists its sparsity averaged over a longitudinal analysis of several epochs (* refers to a pre-trained model). The last column details the average per-worker communication by using OmniReduce shown as volume (and % of the otherwise dense communication).

Moreover, the size of new DNN models is increasing at a faster pace than hardware compute capacity [24]. Therefore efficient communication is becoming even more crucial. For instance, in a short span of 2.5 years, model size has grown by over $1,000\times$ from $\sim$ 100M weights in 2018 for ELMo [51] or BERT [13] to $\sim$ 100B for OpenAI GPT-3 [6] (May 2020). In contrast, the current best-in-class NVIDIA A100 GPU (May 2020) is advertised [47] as up to $10\times$ and $20\times$ faster for floating-point and mixed-precision calculations, respectively, than its V100 predecessor, released in December 2017.

Indeed, the fact that communication is a major performance bottleneck in DDL is well-known [32], and many works [10, 35, 39, 44, 58, 66] proposed various optimizations to achieve high-bandwidth collective communication specialized for DDL. Besides, a recent body of work, primarily within the ML community, developed gradient compression methods [1, 2, 42, 63, 67] to reduce communication time by sending a smaller amount of data, albeit at the cost of reduced training quality due to the lossy nature of compression.

However, these works have failed to observe that, along with the fast-paced increase in model size, gradient sparsity (i.e., the proportion of zero elements in the gradient vector) follows a similar trend. Table 1 shows that gradient sparsity exceeds 94% for the two largest DNN workloads in our study. Sparse gradient vectors (i.e., with sparsity above 50%) are typical for DNNs with a large proportion of embedding weights.[2] This characteristic spans a broad range of deep learning tasks.

Most existing collective libraries – including DDL-specialized ones like NCCL [48] and Gloo [16] – have no native support for sparse data. These libraries assume dense input data and make inefficient use of precious network bandwidth to transmit large volumes of zeros. Generally, this is also a limitation for gradient compression methods because their implementations first gather the sparse data into a dense-like format (which has overheads) before invoking a collective routine [68] (§2).

Our key innovation is the design of efficient collective operations for sparse data. We present OmniReduce, a streaming aggregation system designed to maximize the efficient use of bandwidth and serve as a drop-in replacement for the traditional collective libraries. OmniReduce exploits the sparsity of input data to reduce the amount of communication. As shown in the last column of Table 1, OmniReduce moves up to two orders of magnitude less data by leveraging an aggregator component that determines the non-zero data at each worker in a streaming look-ahead fashion. OmniReduce splits input data into blocks where a block is either a split of contiguous values within an input vector in a dense format or a list of key-value pairs representing non-zero values. OmniReduce achieves high performance through fine-grained parallelization across blocks and pipelining to saturate network bandwidth. OmniReduce leverages fine-grained control of the network to design a self-clocked, bandwidth-optimal protocol. The block-oriented approach, fine-grained parallelism, and built-in flow control allow us to implement the aggregator in-network using modern programmable switching ASICs. In addition, OmniReduce supports both DPDK and RDMA, and where available, exploits GPU-direct RDMA (GDR) to improve the performance.

OmniReduce achieves the following goals:

• **High performance and scalability.** Algorithmically, computational and space complexity do not depend on the number of nodes, while aggregation latency is masked with pipelining. This allows OmniReduce to scale better than previous approaches fundamentally.

• **Data-format universality.** The acceleration is proportional to the sparsity of input data. At the same time, OmniReduce does not require data to be sparse to provide benefits. In the limit, when data is dense, OmniReduce is comparable to bandwidth-optimal dense AllReduce.

• **Flexibility.** OmniReduce's streaming aggregation algorithms admit a variety of instantiations. Sparse input data can be in a block-based dense or sparse (key-value) format without requiring a new API. The aggregator component can run on dedicated server resources (cheaper than worker nodes equipped with GPUs), can run co-located on worker nodes, or with the aid of network switches, as an in-network aggregation component similarly to Mellanox SHARP [44], SwitchML [58] or ATP [38].

To the best of our knowledge, OmniReduce is the first system that realizes all of the above goals at once. SparcML [55] is a collective library for sparse data; however, it requires very high sparsity to achieve performance benefits over dense AllReduce (their results, which we confirm (§6) show benefits when sparsity $>$ 94%). Parallax [34] is a parameter-server architecture specialized for sparse data but requires runtime profiling. Unlike OmniReduce, both of these approaches require input data in the sparse format. We believe that OmniReduce is a general approach and could benefit other applications like data-parallel analytics and sparse matrix multiplication.

We make the following contributions:

• We present the design (§3) and implementation (§5) of OmniReduce, an efficient streaming aggregation system for sparse-native

[2]Embedding layers are used to process high-dimensional and typically sparse data. Typically, updates to embedding weights are sparse as only a few embedding vectors from a huge dictionary are used in one batch, and only these vectors have non-zero gradients in the batch.

collective communication. Via performance modeling, we demonstrate the theoretical advantages of OmniReduce over standard approaches.

- We introduce block gradient sparsification (§4), a gradient compression technique that works by sampling gradients' blocks of contiguous elements. We prove convergence and demonstrate empirically that our block-based sparsification techniques can sparsify data to obtain training speedup with negligible degradation in model performance.
- We quantify the performance benefits of OmniReduce using six popular DNN workloads (§6). In end-to-end settings, OmniReduce speeds up training throughput by up to 8.2× at 10 Gbps and 2.9× at 100 Gbps compared to standard ring AllReduce. We also use benchmarks to compare to state-of-the-art solutions in both TCP/IP and RDMA networks and show that OmniReduce outperforms them by 3.5–16×. OmniReduce is also effective for large-DNN distributed training jobs with multi-GPU servers.

This work does not raise any ethical issues. OmniReduce is available at https://github.com/sands-lab/omnireduce.

2 BACKGROUND

Collective communication routines. The Message Passing Interface (MPI) [17] standard defines a set of communication protocols for point-to-point and collective routines. In DDL, three collectives are typically used:

- *Broadcast* distributes data from one process to all other processes. This is often used to sync model state among workers, e.g., when reading from a model checkpoint.
- *AllReduce* combines data collected from all processes into a global result by a chosen operator (e.g., sum, min, max). AllReduce is the most frequently used collective operation in DDL workloads to aggregate gradients by summation.
- *AllGather* collects data from all processes and stores the collected data on all processes. AllGather is useful when the reduction operation is not an associative, point-wise operation. Some gradient compressors use it [68].

We refer to the datatype of collectives' input and output data as a *tensor* (i.e., a multi-dimensional matrix). Let n be the input size, and c_v be the number of bytes needed to represent a non-zero input value; m is the number of non-zero values.

Tensor data format. The elements of a *dense tensor* are consecutively stored like an array in memory. It is often beneficial or sometimes necessary (due to insufficient memory capacity) to use specialized data structures to store *sparse tensors*. For example, coordinate lists (COO) store a list of non-zero values and a list of the corresponding indices. Dictionary of keys (DOK) stores a dictionary that maps indices to non-zero elements. Although some ML toolkits support sparse tensors (typically in COO format), state-of-the-art collective libraries like NCCL and Gloo operate only with dense tensors even though the underlying data may be sparse.

2.1 Related work

Efficient sparse collectives. A strawman solution to perform collective operations with sparse tensors is to collect the values and indices separately [65]. Further, one can compress the indices using a bitmask [60] or Bloom filters [37], and the values by run-length encoding [23] or quantization [14]. PyTorch implements such a strawman – AllGather-based sparse AllReduce (AGsparse) – that invokes AllGather twice to collect the values and indices of a sparse tensor and makes a local reduction at every process [52]. Because AllGather needs to allocate an intermediate buffer with the size proportional to the number of processes, AGsparse increments the memory footprint despite sparse data. Further, AGsparse has poor scalability (analyzed in §3.4) as it implicitly assumes no overlap of non-zero indices and is viable only when $m \leq \rho = \frac{nc_v}{c_i+c_v}$, where c_i is the number of bytes needed to store an index (i.e., sparsity above 50% assuming $c_v = c_i$).

Kylix [70] is also an AllGather-based sparse AllReduce method, which uses a Butterfly network. However, Kylix performs multiple passes, and its design makes a particular assumption on the data distribution.

SparCML [55] is a set of collectives for arbitrary sparse input data designed for DDL. SparCML uses a latency-bandwidth cost model to characterize different cases and trade-offs between small vs. large messages, and decide whether the output remains sparse or becomes dense (adapting to $m > \rho$), delineating two scenarios: static and dynamic sparse AllReduce (SSAR or DSAR, respectively). Data representation in SSAR is always in the sparse format. When the amount of data is small, latency dominates the bandwidth term; thus, a latency-optimal recursive doubling algorithm is used. With large data, SSAR_Split_allgather is a two-phase algorithm that optimizes AGsparse by (1) splitting the input into N partitions, one per process, each processed via an AGsparse-like approach to gather data at each designated process and (2) a gathering phase that uses a concatenating AllGather to collect reduced sparse data at all processes. In DASR, DSAR_Split_allgather starts with sparse representation and switches to dense representation during the reduction operation once the condition $m > \rho$ is detected.

In AGsparse and SparCML, communication and reduction occur separately and serially. Instead, OmniReduce performs communication and reduction in parallel by streaming data via the aggregator. OmniReduce thus can make full use of network bandwidth, while bandwidth is wasted when conducting local reductions in AGsparse and SparCML. OmniReduce supports dense inputs without the format conversion overheads paid by AGsparse and SparCML (§6.1). OmniReduce can reduce communication volume by adopting a block-based format because it does not need to transfer indices.

Parallax [34] devises a hybrid DDL system that has a runtime sparsity monitor and uses a cost model to partition the model weights between a parameter server (PS) architecture for sparse data and traditional AllReduce for dense data. OmniReduce neither requires prior knowledge nor introduces runtime profiling.

All the above works do not use streaming methods for sparse data AllReduce because the indices of non-zero elements across all workers are unknown before the AllReduce operation; thus, the result can only be broadcast after all the key-value pairs are received and reduced. This problem makes current sparse collective works unable to take full advantage of the inbound and outbound bandwidth at workers. OmniReduce solves this problem by using aggregators to coordinate between workers. Besides, no prior work evaluates performance with a 100 Gbps network nor utilizes RDMA fully (Parallax uses RDMA only with dense data). This is mainly

because the memory access operation for each key-value pair is not suitable for RDMA. OmniReduce uses a block-wise method to solve this problem, which ensures full use of the bandwidth while taking sparsity into account.

Gradient compression. Orthogonal to efficient sparse collective communication, a recent body of work proposes to reduce the amount of communication via gradient compression. As there is a vast literature on the topic, we refer to a recent survey [68] for a comprehensive discussion. And while techniques abound [1, 2, 42, 63, 67], we distinguish two main approaches: *sparsification* – which sends a subset of elements – and *quantization* – which reduces the per-element bit-width. Gradient compression is typically lossy and, as a result, can impact the resulted model quality; however, the drop in accuracy is usually small, and one can regulate the compression level to navigate the trade-off. These techniques and OmniReduce are complementary: on the one hand, gradient compression helps to sparsify data in a principled manner; on the other hand, OmniReduce accelerates collective communication of sparse data (allowing for a less aggressive compression level for a given communication budget). We defer discussing other related work to §8.

3 OMNIREDUCE DESIGN

To minimize AllReduce latency, the core idea of OmniReduce is to partition an input tensor G into blocks of tensor elements and transmit only non-zero blocks (i.e., blocks with at least one non-zero value). OmniReduce consists of *worker* and *aggregator* components. The aggregator coordinates workers, instructing them on which block to send next. For scalability, the aggregator executes over one or more nodes; in the latter case, each node owns a disjoint shard of blocks. Each aggregator node has a pool of slots, and each slot aggregates a block-sized set of tensor elements. Workers are responsible for detecting and sending non-zero blocks.

Depending on the application, the block format is either dense (i.e., a contiguous subset of G) or sparse (i.e., a list of key-value pairs). We first consider the dense format and then generalize it to the sparse format.

OmniReduce fundamentally improves AllReduce performance thanks to the following two design principles:

• **Fine-grained parallelism and data pipelining.** With each block being independent of any other block, aggregation can be easily parallelized. This enables tightly coupled workers to stream data as a form of a latency-masking pipeline to saturate the aggregator's processing rate, which also serves as a flow control function, yielding a self-clocked protocol similar to other streaming aggregation approaches [44, 58].

• **Coordinated aggregation.** Coordination is key to sending only the non-zero data. The aggregator globally determines the positions of non-zero values among workers in a look-ahead fashion based on the next position metadata efficiently available at the workers (which communicate it to the aggregator). This component differentiates OmniReduce from any related work.

3.1 Basic solution

For simplicity, we introduce the more straightforward scenario of a lossless network with guaranteed packet delivery. This matches the environment of our RDMA implementation. Figure 2 illustrates this scenario with an example. Due to space limit, we discuss packet loss recovery in Appendix A.

Algorithm 1: OmniReduce block aggregation

1 **At Worker:**
2 $p.next, next \leftarrow$ index of first non-zero block past block 0
3 $p.block \leftarrow 0$
4 $p.wid \leftarrow$ worker ID
5 $p.data \leftarrow G[0 : bs]$
6 **send** p to *agg*
7 **repeat upon receive** *p(data, block, next, wid)*
8 $G[p.block : p.block + bs] \leftarrow p.data$
9 **if** $p.next = next$ **then**
10 $p.data \leftarrow G[next : next + bs]$
11 $p.block \leftarrow next$
12 $p.next, next \leftarrow$ next non-zero block index **or else** ∞
13 $p.wid \leftarrow$ worker ID
14 **send** p to *agg*
15 **until** $p.next = \infty$
16 **At Aggregator:**
17 $slot[bs] := \{0\}$
18 $next[N] := \{-\infty\}$ // *per-worker next non-zero block index*
19 **forever upon receive** *p(data, block, next, wid)*
20 $slot \leftarrow slot + p.data$ // *reduction operation*
21 $next[p.wid] \leftarrow p.next$
22 **if** $p.block < \min(next)$ **then**
23 $p.data \leftarrow slot$
24 $p.next \leftarrow \min(next)$
25 $slot[bs] := \{0\}$
26 **if** $\min(next) = \infty$ **then** $next[N] := \{-\infty\}$
27 **send** p to all workers

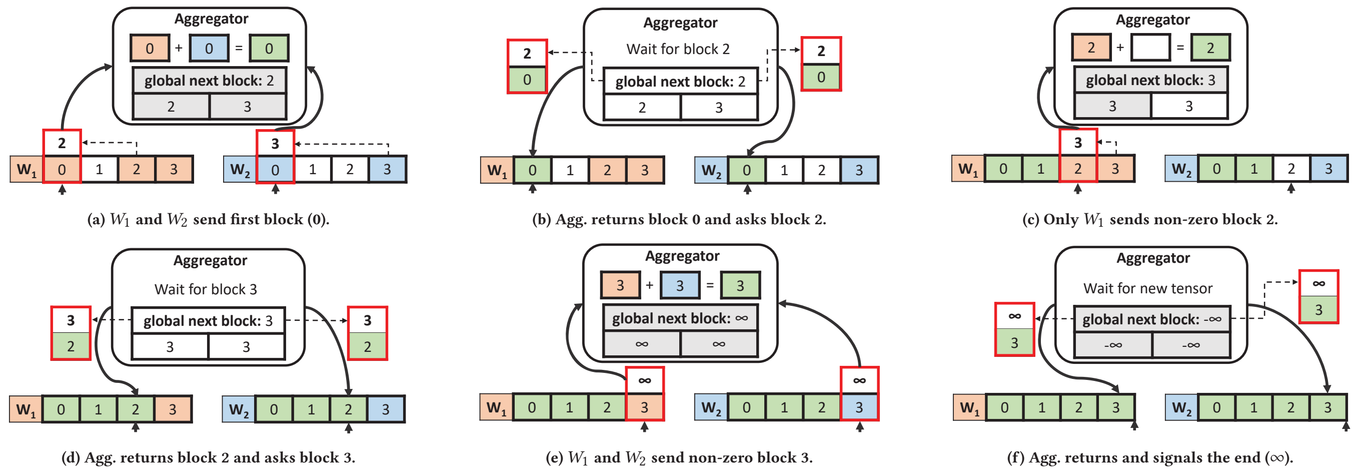
Algorithm 1 illustrates the basic OmniReduce algorithm for dense tensors. A dense tensor consists of a list of values partitioned into blocks. Every block has a size of bs. For ease of description and without loss of generality, we assume that the tensor size is a multiple of bs, and the pool size is 1 (i.e., the aggregator has a single slot). We assume the reduction operation is sum (+). Other commutative reduction operations are analogous.

Worker: Every worker initially sets *next* as the offset of the next non-zero block after the first block and records it locally. The worker then sends a packet p containing the first block and *next* (Figure 2a).

Then, each worker enters a loop where it awaits the aggregator's response. Upon receiving a packet, the worker obtains: (1) the aggregated block data ($p.data$) along with its respective number ($p.block$), and (2) the next block ($p.next$) requested by the aggregator (Figure 2b).

The worker stores the aggregated block data into the local tensor G; then, the worker checks whether its next non-zero block corresponds to the aggregator's request. If so, the worker updates *next* with the subsequent non-zero block and sends the requested block to the aggregator (W_1 in Figure 2c). Otherwise the worker awaits a further packet (W_2 in Figure 2c). The loop repeats (Figure 2c-2e) and ends once the aggregator signals that reduction is complete by requesting ∞ as the *next* block (Figure 2f).

Aggregator: The aggregator does not only aggregate blocks but also keeps track of each worker's next non-zero block. This state is updated whenever a worker sends a packet and enables the aggregator to know the global next non-zero block number. This information is piggybacked into a packet that the aggregator multicasts to the workers with the aggregated data once it determines that a slot is complete. To determine so, the aggregator compares



(a) W_1 and W_2 send first block (0).

(b) Agg. returns block 0 and asks block 2.

(c) Only W_1 sends non-zero block 2.

(d) Agg. returns block 2 and asks block 3.

(e) W_1 and W_2 send non-zero block 3.

(f) Agg. returns and signals the end (∞).

Figure 2: After every worker sends its first block, the aggregator maintains a view of the global next block necessary for aggregation. Workers only transmit non-zero blocks when the block number matches the requested global next block and inform the aggregator of their next non-zero block. Cumulatively, this ensures that aggregation completes once all non-zero block are transmitted; zero blocks are not transmitted.

Legends: ■ and ■ are non-zero blocks; ■ are aggregated blocks; □ are zero blocks.

the packet's block number ($p.block$) with the minimum of next non-zero blocks across all workers, $\min(next)$. Note that $next$ is already updated to reflect $p.next$. If $p.block$ is less than $\min(next)$, then the current packet is the last one for this slot. The aggregator then crafts a response packet for $p.block$ with the aggregated data in $slot$, resets the state ($next$ and $slot$), and multicasts that packet to the workers.

3.1.1 Fine-grained parallelism. It is easy to observe that the above aggregation logic, while tightly coupling workers at a particular slot, can be parallelized across slots. In the limit, each slot is an independent unit of aggregation. In practice, available network bandwidth limits the number of slots that can be addressed in parallel before the aggregator responds.

OmniReduce exploits this kind of fine-grained parallelism to achieve a form of pipelining that improves performance. The aggregator maintains a *pool of slots* addressable by indices (carried in each packet). Workers run Algorithm 1 for S independent aggregation streams (or threads), each of which addresses a separate slot while proceeding at the same rate. As packets are serialized over the network, this architecture can be viewed as pipeline-based processing of one slot per time unit. The available network bandwidth dictates the pipeline depth that is necessary to avoid processing stalls.

3.1.2 How sparsity affects performance. Since workers only send non-zero blocks, a crucial performance factor is the tensor's block sparsity, which is the proportion of all-zero blocks in the tensor. In turn, block sparsity is determined by not only the tensor itself but also the block size. In general, a smaller block size increases block sparsity, but it also decreases bandwidth utilization efficiency due to packets carrying a smaller payload. Focusing on this, we propose the Block Fusion method (§3.2). We analyze the effects of block size on the performance of OmniReduce in the evaluation section. Therein we empirically find (§6.4) that a block size of 256 elements (32-bit floating point values) is the best choice in our setting.

Another factor influencing the performance of OmniReduce in practice might be the cost of finding the next non-zero block. Nevertheless, we find that checking all values in one block has a negligible overhead when this operation is done in parallel on the GPU, as we implement it (§5).

3.2 Block fusion

In the basic solution, each packet contains one block; therefore the payload is equal to $bs \cdot c_v$ plus metadata (next). Having bigger blocks increase bandwidth utilization but also leads to a lower block sparsity because a larger block is more likely to contain at least one non-zero element.

To balance the trade-off between block sparsity and bandwidth usage, we propose a method called Block Fusion, which packs multiple blocks into a single packet and uses one slot to process them in batch. As with the basic solution, a slot remains the minimum unit of aggregation and has the capacity to aggregate all values in a single packet. Thus, if w is the number of blocks fused in a packet to maximize its payload, a slot aggregates $w \cdot bs$ values at once.

The main technical challenge is how to choose which blocks to fuse in a packet. While it may seem straightforward to have each worker fuse its next w non-zero blocks, note that streaming aggregation is most effective when same-offset blocks are sent at the same time among workers. Thus, we must ensure that blocks are mapped at consistent locations into the packet (and, in turn, the slot) based on block offsets. To do so, a gradient tensor in a worker is arranged as a two-dimensional matrix of blocks. For clarity, consider the example shown in Figure 3. Then, we map each block at a determined location into the packet based on its column index and we include in each packet a next non-zero block index for each fused block. The next block offset is found by scanning over the rows for each column independently. This scheme ensures that two blocks at the same column index cannot be fused into the same packet. As a result, the same logic of the basic solution

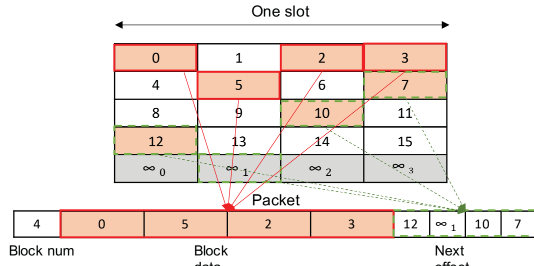


Figure 3: Tensor data at one worker viewed in a two-dimensional layout and a packet in the Block Fusion method with $w = 4$ packed blocks. Blocks in orange are non-zero.

(Algorithm 1) remains unchanged and this scheme does not require transmitting any additional information.

When the aggregator receives a packet, it checks the completion for each block using the same condition as the basic solution (line 22 in Algorithm 1). The aggregator multicasts aggregated data in a slot to the workers after all blocks in the slot are completely aggregated. Workers, upon receiving a result packet, check whether any of the next blocks requested by the aggregator are non-zero. As long as one requested block is non-zero, each worker fuses the requested blocks into a packet and sends it to the aggregator. The reduction is complete when the aggregator requests ∞ as the next block for all the w blocks in the slot, for every slot.

The worker does not send non-zero blocks. The packet includes a *block num* field to denote how many blocks it includes. The aggregator identifies the column index of each included non-zero block based on the values for the next block index.[3]

3.3 Extension to sparse block format

OmniReduce's block aggregation approach generalizes to sparse tensors (e.g., in COO format). We briefly discuss this extension, the algorithm of which is in Algorithm 3. For ease of presentation, we do not consider stream parallelism or packet loss recovery. In this case, the input tensor is a K, V pair, where K is a list of keys (or indices) and V is a list of the corresponding values. The worker sends a packet with a block of bs key-value pairs along with the $nextkey$ to indicate the key of the next non-zero value. The aggregator keeps track of $nextkey$ for every worker, attaching the minimum next key it needs to receive from any worker when sending back a result packet. Only when a worker receives a $p.nextkey$ matching its next non-zero value will it send another block to the aggregator. The aggregator internally uses a hashtable or a similar keyed-memory abstraction to carry out aggregation based on key-value pairs.

While we present the above approach for completeness, we do not investigate its practical realization and leave it as future work. We note that our real-world applications only use dense tensors, and format conversion entails non-negligible overheads (§6.1). As this approach only transmits non-zero values, it could be more advantageous than the dense block format when a block has more than $\frac{bs \cdot c_v}{c_i + c_v}$ zero values within it. We observe that, the dense block format maintains high sparsity for a range of block sizes (§6.4).

[3]Column index i is determined as $i = \frac{next}{bs} \mod w$. We use w distinct values of ∞, each for a different column. The value ∞_i maps to i.

3.4 Performance analysis

We analyze the theoretical benefits of OmniReduce following the modeling approach of Patarasuk et al. [49]. We use a performance model to compare OmniReduce versus ring AllReduce, which is bandwidth optimal [49] and versus AGsparse AllReduce. As the primary interest is the dominating communication time, our analysis ignores the unitary local reduction time in the model below since pipelining could mask much of this latency term.

Ring AllReduce is a widely-adopted AllReduce algorithm and is the default algorithm for Gloo and NCCL. Consider N workers and that each worker has full-duplex network bandwidth B; the time to perform a ring-based AllReduce operation of S elements is: $T_{ring} = 2(N-1)(\alpha + \frac{S}{NB})$.

Where α is the one-way network latency between workers (assumed to be uniform).

AGsparse AllReduce is a commonly used method to reduce sparse format data (key and value pairs). It consists of two steps: (1) AllGather keys and values, and (2) local reduction. Let $D \in [0, 1]$ be the density of elements at each worker; the number of input elements to AllGather is $2DS$ (i.e., DS keys and DS values). An AllGather operation only performs the first phase of the AllReduce operation, halving its time for input with $2DSN$ elements. Thus, the AGsparse AllReduce time is: $T_{AGsparse} = (N-1)(\alpha + \frac{2DS}{B})$.

OmniReduce achieves bandwidth-optimality when the aggregator bandwidth matches the combined worker bandwidth NB and only non-zero elements are transmitted. This best-case scenario is analyzed here, which implies that block density is the same as the element density D. Note that the number of aggregator nodes used is not relevant because fine-grained parallelism enables ideal linear scaling through sharding. The aggregator receives a total of DS elements ($\frac{DS}{N}$ from each worker).

As the data transmission and aggregation at the aggregator is pipelined, the latency of intermediate packets is masked. Thus, the OmniReduce time is: $T_{OmniReduce} = \alpha + \frac{DS}{B}$

Speedup. To ease comparison, we distinguish two cases: (1) very sparse data and (2) sparse-to-dense data.

Very sparse data: in this case, D is very small and the latency term α dominates the bandwidth term. OmniReduce is expected to be better than both ring AllReduce and AGsparse AllReduce because OmniReduce's performance does not depend on the number of workers N.

Sparse-to-dense data: as the data volume is larger in this case, we can ignore the latency α. We calculate the theoretical speedup factor of OmniReduce relative to other approaches as follows:

	$SU_{\text{vs. ring}}$	$SU_{\text{vs. AGsparse}}$
$\frac{T_{\text{other}}}{T_{\text{OmniReduce}}}$	$\frac{2(N-1)}{ND}$	$2(N-1)$

The performance benefit of OmniReduce is two-fold. First, OmniReduce is much more scalable, and both speedup factors grow with the number of workers because OmniReduce's time does not depend on the number of workers. This speedup is fundamental and exists even with a dense input ($D = 1$). Second, in contrast to ring AllReduce, OmniReduce only sends non-zero elements, which reduces the time proportionally to $\frac{1}{D}$.

Further, we observe that OmniReduce remains advantageous even in a co-location setting where the aggregator service is sharded and co-located across N workers (each of which thus has $\frac{B}{2}$ bandwidth). In this case, the benefit over ring AllReduce overall diminishes by a factor of 2 and $SU_{\text{vs. ring}} = 1$ when $D = 1$.

4 BLOCK-BASED SPARSIFICATION

Given the performance benefits of OmniReduce for sparse gradients by sending only non-zero blocks, one can think of using OmniReduce with block-based gradient sparsification techniques when gradients are not sparse. While many element-wise sparsification techniques exist in the literature, e.g., Random-k [62], Top-k [3, 42], and threshold [15, 63], no block-based sparsification technique exists.

Hence, as a natural extension to the existing element-wise sparsification techniques, we devise and experiment with the following block-based sparsification schemes:

- *Block Random-k*: Randomly sample k blocks.
- *Block Top-k*: Select Top-k blocks according to the block gradient norm (ℓ_2 norm of the gradient values in the block).
- *Block Top-k Ratio*: Select Top-k blocks according to the block update-ratio norm, where update-ratio for a parameter is the ratio of its gradient value to parameter value.
- *Block threshold*: Select blocks with the block gradient norm higher than a given threshold.

While a new theoretical analysis for block-based sparsification is out of scope, we show that *Block Random-k* and *Block Top-k* are δ-compressors [30], and hence *converge according to the Error-Feedback theory* [62, 71].

LEMMA. *Let b denote the total number of blocks. Both Block Random-k and Block Top-k are δ-compressors with $\delta = \frac{k}{b}$.*

PROOF. *The proof is in Appendix C.*

Using this lemma, Theorem 1 in [71] gives the convergence result for compressed distributed SGD with error-feedback for any δ-compressor. Our empirical results confirm that block-based gradient compression converges (§6.2).

5 IMPLEMENTATION

We implement OmniReduce using C++11 and CUDA, optimizing communication for both TCP/IP and RDMA networks. We also make use of GPU-direct RDMA (GDR) where available. Furthermore, OmniReduce is integrated with PyTorch's DistributedDataParallel (DDP) package (torch.distributed), which transparently performs distributed data-parallel training. The worker component is written in $\sim 4,000$ lines of codes (LoC), while the aggregator is $\sim 1,500$ LoCs in C++.

DPDK. For clusters without RDMA support, we use Intel DPDK for kernel bypass and communicate using UDP packets. This implementation includes our packet-loss recovery method (Appendix A). To reach full bandwidth utilization, we use DPDK flow director to scale packet processing to 4 CPU cores on both workers and aggregators. The number of outstanding packets processed by each worker is set to 256 (64 packets per core).

RDMA. OmniReduce uses RDMA RoCE v2 in Reliable Connected (RC) mode, which guarantees at-most-once, in order, and without corruption delivery. We use sender/receiver buffers for data exchange between workers and aggregators. OmniReduce uses RDMA SEND/RECV operations to exchange buffer memory addresses and RDMA WRITE_WITH_IMM for data transfer. OmniReduce metadata is encoded as 32-bit immediate values consisting of data type (2 bits), AllReduce opcode (2 bits), slot id (12 bits) and the number of blocks (16 bits) in this message. The block data and the offsets of next non-zero blocks are taken as the payload of the RDMA message. An aggregation slot works at the granularity of an RDMA message and not of a single packet; the logic is unchanged.

Multi-GPU servers. When there are multiple GPUs per server, OmniReduce performs a two-layer hierarchical aggregation. We use NCCL for intra-server multi-GPU reduction and broadcast in the first layer and use OmniReduce for inter-server communication.

Additional implementation details are in Appendix B.

6 EVALUATION

We evaluate OmniReduce's performance and compare it to both dense and sparse state-of-the-art collective libraries in both 10 Gbps and 100 Gbps networks. We also evaluate the benefits of the Block Fusion method and analyze the influence of different factors like block size, sparsity, and non-zero block overlap among workers in §6.4. For the DPDK-based OmniReduce, we also analyze the influence of packet loss rates in Appendix D. Our experiments rely on several microbenchmarks as well as six end-to-end training workloads with 256 as the default block size.

Testbeds. Our experiments mainly target two testbeds for the 10 and 100 Gbps cases, which consist of 24 machines in total. Eight machines are equipped with dual 8-core Intel Xeon Silver 4108 CPU at 1.80 GHz and have no GPUs; these are connected at both 10 and 100 Gbps and serve as aggregators. Other machines are workers: (1) In the 10 Gbps testbed, there are eight machines, each equipped with 1 NVIDIA P100 GPU, dual 10-core CPU Intel Xeon E5-2630 v4 at 2.20 GHz, 128 GB of RAM, and 10 GbE Intel NIC. (2) In the 100 Gbps testbed, there are eight machines, each equipped with 2 NVIDIA V100 GPUs (only one supports GDR), dual 4-core CPU Intel Xeon Silver 4112 CPU at 2.60 GHz, 128 GB of RAM, and 100 GbE Mellanox ConnectX-5 NIC. CPU frequency scaling is disabled for all the machines. The machines run Ubuntu 18.04 (Linux 4.15.0), CUDA 10.1 (where applicable), PyTorch 1.8.0a0 and NCCL 2.4.8.

For multi-GPU experiments, another testbed comprising of 6 multi-GPU servers and 6 CPU servers is used. Each machine is equipped with 64-core Intel Xeon Gold 5218 CPU at 2.30 GHz. Each GPU machine has 8 NVIDIA V100 GPUs, which are connected with NVLink. These machines are networked at 100 Gbps.

Microbenchmark setup. For microbenchmarks, we use AllReduce completion time as the performance metric. We collect measurements at each worker for 200 iterations after 10 warm-ups. Sparse tensors are generated randomly at each iteration. As the baseline, we use the industry-standard ring AllReduce algorithm implemented in NCCL and run it with TCP/IP at 10 Gbps, RDMA and GDR at 100 Gbps.

Training workloads. We use six real-world models for the end-to-end experiments, including two image classification models, two

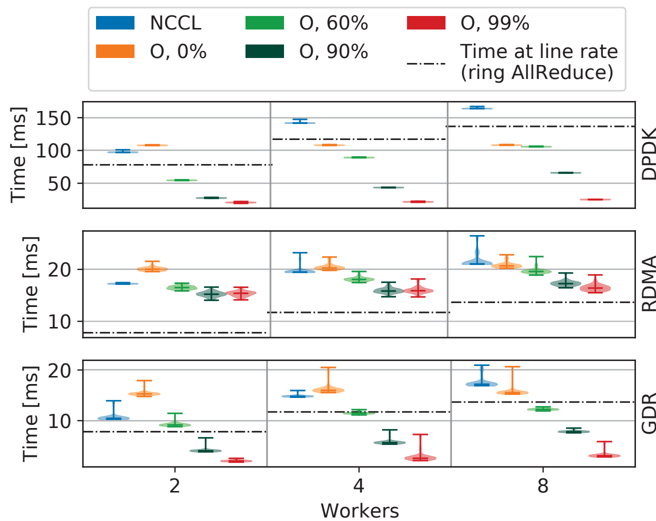


Figure 4: Time to complete AllReduce on 100 MB tensors. Dashed lines show the optimal ring AllReduce time [49] based on the maximum goodput under line-rate bandwidth. $O, s\%$ denotes OmniReduce on tensors with $s\%$ sparsity. The block size is 256. OmniReduce runs at 10 Gbps with DPDK; at 100 Gbps with RDMA and GDR.

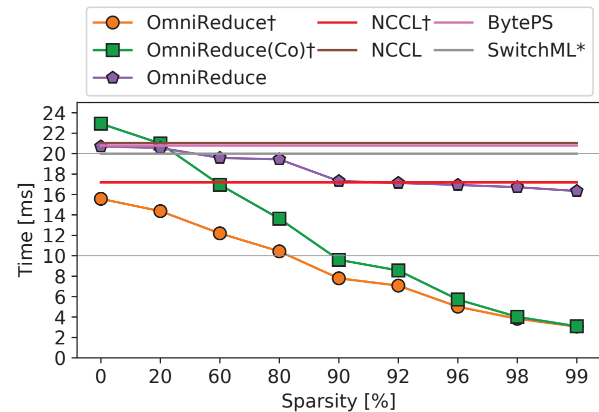


Figure 5: Comparison of OmniReduce and other dense AllReduce methods with RDMA or GDR (denoted with †) support in 100 Gbps network. OmniReduce(Co) refers to the colocated version.

NLP models, and two recommendation models. Table 1 shows details of the models, datasets, and batch sizes. We observed that *training throughput* (the number of training samples processed per unit of time) stabilizes after the first 100 iterations. Thus, we exclude them and report performance for the subsequent 200 iterations. We do not include training accuracy results because OmniReduce merely skips zero blocks and does not affect accuracy; we do report accuracy for block-based compression results. The end-to-end experiments are done in both 10 Gbps and 100 Gbps networks (the latter case only for RDMA and GDR).

6.1 Microbenchmarks

6.1.1 Comparison with dense AllReduce. We first compare OmniReduce with NCCL on the most commonly used collective operation for DDL: dense AllReduce. We devise this micro-benchmark atop PyTorch by generating input tensors on the GPU and invoking PyTorch's all_reduce API from the torch.distributed package, with the communication backend being OmniReduce or NCCL.

We tested tensor sizes from 100 MB to 1,000 MB, and observe that tensor size has a low impact on the throughput. Therefore, we only report results for 100 MB tensors. Moreover, to analyze tensor sparsity's influence on performance, we generate tensors with different sparsity s from 0% to 99%. All tensors are generated randomly, and so non-zero blocks randomly overlap among workers. We analyze the effects of overlap in §6.4.2.

Figure 4 shows the results as we vary the number of workers from 2 to 8 in three configurations. The results show that OmniReduce achieves up to 6.3× and 5.5× speedup over NCCL at 99% sparsity in 10 and 100 Gbps networks, respectively. With 60% sparsity or more, OmniReduce always outperforms NCCL. When data is dense, OmniReduce with two workers is slower than NCCL. Note that in this case, the aggregation is not necessary because full-duplex communication is the ideal strategy. However, we attribute this to two factors: (1) The block size of 256 causes inefficient use of the network bandwidth. As we increase the block size to 1024, OmniReduce performance is close to NCCL for dense data with two workers; (2) OmniReduce adds metadata (e.g., *next*) within each packet, which is pure overhead when data is dense. The performance of RDMA-based OmniReduce no longer improves significantly when sparsity is higher than 90%, This is because the memory copy between GPU and host memory becomes the bottleneck at 100 Gbps. GDR reduces this PCIe traffic, and so OmniReduce can benefit from higher sparsity. Overall these performance gains confirm the previous theoretical insights (§3.4) and the observation that non-zero block overlap influences performance (sensitivity analysis in §6.4).

As expected, OmniReduce exhibits higher scalability than NCCL. As the number of workers increases, OmniReduce with dense data ($s = 0\%$) maintains a constant AllReduce time, whereas that of NCCL increases. However, when $s > 0\%$, OmniReduce's performance is affected by the number of workers, especially while $s < 90\%$. Our performance model did not capture this behavior, and we discuss this apparent gap: The model assumed a uniform block sparsity across workers; however, the input tensors are generated randomly within each worker in this experiment. Thus, workers are likely to hold non-zero blocks distributed at different parts of the tensor. As long as one worker holds a non-zero block at a certain index, one round-trip time is needed, meaning that non-zero block overlap conditions can influence OmniReduce's performance. In particular, the block sparsity decreases with a higher worker count. Nevertheless, the performance speedup increases with the number of workers.

We also compare OmniReduce with other state-of-the-art dense AllReduce methods: BytePS [7] and SwitchML [58]. Figure 5 shows the performance comparison with 8 workers at 100 Gbps as we vary sparsity. Specifically, we use a server-based implementation of SwitchML denoted as SwitchML*. BytePS and SwitchML* support RDMA but do not run with GDR. The results show that BytePS performs very closely to NCCL. SwitchML* has better performance

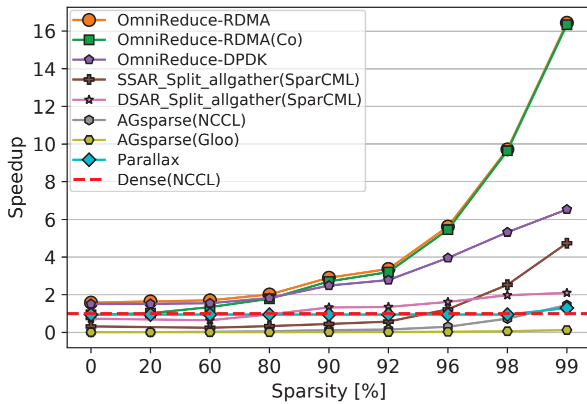


Figure 6: Comparison of OmniReduce and other sparse AllReduce methods as sparsity varies in 10 Gbps network.

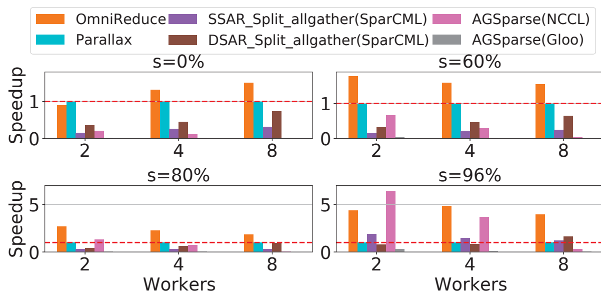


Figure 7: Scalability of OmniReduce and other sparse AllReduce methods as workers and sparsity vary.

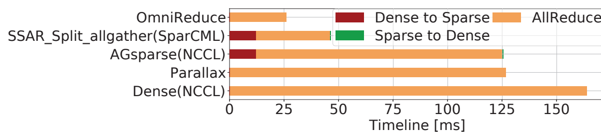


Figure 8: Breakdown of AllReduce execution (including format conversion time) with $s = 99\%$.

for dense tensors due to its streaming aggregation protocol. RDMA-based OmniReduce outperforms SwitchML* when the sparsity is higher than 60%. When using GDR, OmniReduce in dedicated mode is better than NCCL at any sparsity level, while colocated OmniReduce outperforms NCCL only when the sparsity is more than 60%. For dense tensors, colocated OmniReduce is worse than NCCL, because it does not make full use of network bandwidth due to the limited number of CPU cores in our testbed.

6.1.2 Comparison with other sparse AllReduce methods. We focus on three sparse AllReduce approaches:
1) AGsparse, which PyTorch implemented for sparse format (key-value pairs) tensors atop Gloo's AllGather operation. We also implement AGsparse atop NCCL.
2) Two SparCML [55] methods – SSAR_Split_allgather and DSAR-_Split_allgather – that dominate performance for all SparCML methods in our experiments.
3) Parallax [34], which uses a parameter server (PS) to aggregate

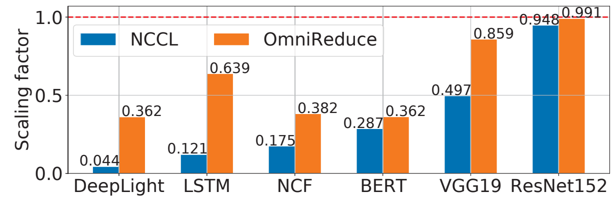


Figure 9: Scaling factor comparison of OmniReduce and NCCL in 10 Gbps network. Results for 8 workers; 2 and 4 workers are similar. See Figure 1 for the definition of scaling factor.

sparse format tensors and NCCL AllReduce operations to aggregate dense format tensors.

We compare the performance of OmniReduce with all these sparse AllReduce methods using a 100 MB tensor, with sparsity varying from 0% to 99%. The non-zero blocks randomly overlap among workers. We exclude the format conversion overheads (for now), i.e., we use dense format for Omnireduce and the baseline (dense AllReduce using NCCL) while we use sparse format (key-value pairs) for AGsparse and SparCML. We mimic the Parallax runtime profiler by an ideal oracle: For each tensor, we separately measure the sparse format performance with both the PS and the dense format performance with AllReduce, then cherry-pick the better one as Parallax's performance.

To fairly compare with SparCML, we use the benchmark provided in the SparCML release [54] and we restrict it to the 10 Gbps network since SparCML was prototyped and evaluated with 1 Gbps (at 100 Gbps it has no sensible speedup even at high sparsity).

Figure 6 presents the performance of OmniReduce, AGsparse, SparCML and Parallax normalized to the baseline in an 8-worker setting. OmniReduce-RDMA(Co) denotes the case where the aggregator processes are colocated with the workers. In dedicated mode, OmniReduce outperforms all other approaches at any sparsity. In colocated mode, OmniReduce does not hurt performance even for dense data and achieves up to 16× speedup. In fact, when the sparsity is more than 80%, colocated OmniReduce matches the performance of dedicated mode because at high sparsity and at 10 Gbps, 4 CPU cores for colocated mode are sufficient to make good use of network bandwidth to transfer non-zero-blocks.

Compared to the baseline, both DPDK- and RDMA-based OmniReduce achieve at least 1.5× speedup and up to 6.3× and 16× speedup at $s = 99\%$, respectively. SparCML, AGsparse (NCCL), and Parallax are only beneficial when the tensor sparsity is higher than 90%, 98%, and 99%, respectively.

Figure 7 further shows the speedup for four sparsity levels as we vary the number of workers from 2 to 8.[4] Following our theoretical insights (§3.4), we expect OmniReduce to have the best scalability and AGsparse to have the worst scalability. OmniReduce is only affected by the number of workers N to the extent that N influences global sparsity, whereas AGsparse scales poorly with the number of workers (the speedup actually decreases). For dense tensors ($s = 0\%$), the speedup of OmniReduce increases with more workers. When the sparsity is higher, the speedup of OmniReduce tends to diminish as workers increase. This is because the non-zero blocks

[4] Parallax is the same as NCCL; the PS is only effective at 99% sparsity.

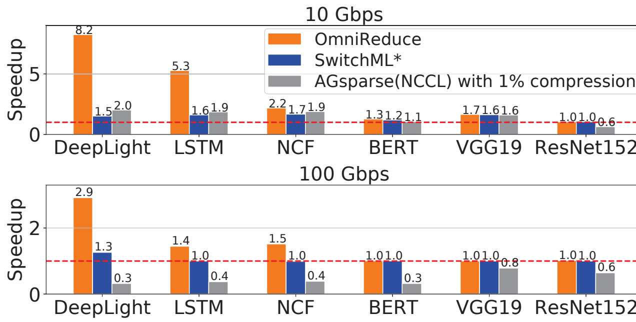


Figure 10: Training performance speedup for 6 DNNs normalized to dense AllReduce (NCCL).

Overlap	DeepLight	LSTM	NCF	BERT	VGG19	ResNet152	sBERT
None	59.49%	18.10%	27.48%	0.60%	0.03%	0.01%	83.15%
2	11.94%	4.58%	17.78%	0.11%	0.02%	0.01%	12.81%
3	5.61%	1.98%	13.10%	0.04%	0.01%	0.00%	2.63%
4	3.40%	1.11%	10.29%	0.02%	0.01%	0.00%	0.78%
5	2.36%	0.71%	8.52%	0.01%	0.02%	0.00%	0.31%
6	1.85%	0.50%	7.60%	0.01%	0.06%	0.01%	0.14%
7	1.73%	0.40%	7.39%	0.01%	1.05%	0.01%	0.07%
All	13.62%	72.61%	7.85%	99.20%	98.79%	99.96%	0.11%

Table 2: Breakdown of OmniReduce communication (8 workers) by the number of workers that overlap non-zero blocks. sBERT denotes BERT with 1% Block Top-k compression.

in every worker do not completely overlap, which overall results in lower global sparsity. We study this effect in §6.4.

Table 2 breaks down the proportion of communication by the extent of block overlap among workers. For example, among the 280 MB or 41% non-zero blocks (c.f. last column of Table 1) that OmniReduce sends during the training of NCF, 7.85% of the blocks fully overlap while 27.48% of them are from one worker only. The observation also reflects that, while NCF has relatively high gradient sparsity, the non-zero blocks do not overlap ideally, making the performance gain smaller than expected.

Amongst other sparse methods, SparCML has better scalability than AGsparse, especially the DSAR_Split_allgather method, whose speedup always increases with more workers. This method's benefit comes from the automatic switch between sparse representation and dense representation. According to the results in [55], this scalability trend saturates at 16 workers, and the speedup then decreases for higher worker counts. Nevertheless, OmniReduce outperforms all SparCML methods at any sparsity with 2 to 8 workers.

6.1.3 Format conversion cost. The experiments above use either a dense or sparse format input matching each method's assumption. In practice, our DNNs use dense tensors, and format conversion is required for AGsparse and SparCML. Figure 8 illustrates the total AllReduce time when including format conversion costs. These overheads increase with lower sparsity. In this scenario, OmniReduce's advantages are even more apparent.

6.2 End-to-end training

We demonstrate that OmniReduce increases scalability and accelerates training for real-world DNNs (Table 1).

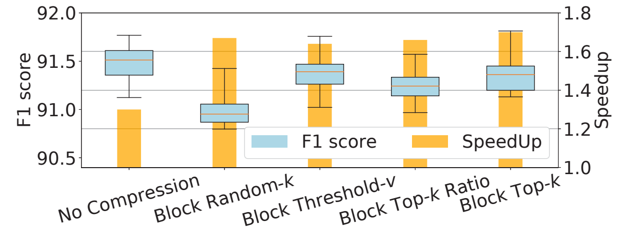


Figure 11: BERT training accuracy and speedup with OmniReduce with different blck-based compression methods.

6.2.1 Scalability. As discussed, inefficient collective communication in DDL results in poor scalability. Figure 9 shows that OmniReduce improves the scalability in every DNN benchmark, whereas the scaling factors for NCCL decrease with more workers. OmniReduce outperforms NCCL in these workloads and achieves a substantial scalability improvement, especially with large DNNs: 8.2× for DeepLight and 5.3× for LSTM.

6.2.2 Training speedup. To tease out the contribution of sparsity versus streaming aggregation in OmniReduce's performance improvement, we also compare OmniReduce performance with SwitchML*, which only supports streaming aggregation. Figure 10 shows the training speedup of OmniReduce, SwitchML* and AGsparse (NCCL) relative to dense AllReduce in an 8-worker setup. OmniReduce accelerates training by up to 8.2× compared to NCCL at 10 Gbps. For certain DNNs, e.g., ResNet152, there is no speedup. This is expected because not every DNN is network-bound [69]. However, OmniReduce does not decrease performance in this case. Moreover, the speedup is understandably more significant for DNNs with high gradient sparsity. For models with low sparsity like BERT, VGG19, and ResNet152, OmniReduce performance is the same as SwitchML* because only streaming aggregation contributes to the speedup in this case. For models with high sparsity like DeepLight, LSTM and NCF, OmniReduce performance is much better than SwitchML*. The performance gain comes from avoiding zero blocks transmission in OmniReduce. OmniReduce also outperforms AGsparse (NCCL) and provides benefits even at 100 Gbps. We now elaborate on these results.

Since AGsparse methods are beneficial only at high sparsity (§6.1), we apply gradient compression at 1% (s = 99%) before invoking AGsparse AllReduce. To focus on collective communication performance, we do not consider compression overheads (even though they may be prohibitive in practice [40, 68]). The results show that AGsparse (NCCL) achieves a lower speedup than OmniReduce at 10 Gbps and is not effective at 100 Gbps; this is due to the format conversion overheads that become the main performance bottleneck. SparCML and Parallax do not integrate with PyTorch, thus we do not use them in these experiments.

At 100 Gbps, OmniReduce provides benefits in the range 1.4× to 2.9× for half of the workloads and precisely for the DNNs with a large proportion of embedding weights that yield gradients with more than 84% sparsity (Table 1). BERT also has large embedding weights, but they only account for a minor part (∼ 20%) of the model. We next show how OmniReduce with block-based gradient compression accelerates the BERT workload.

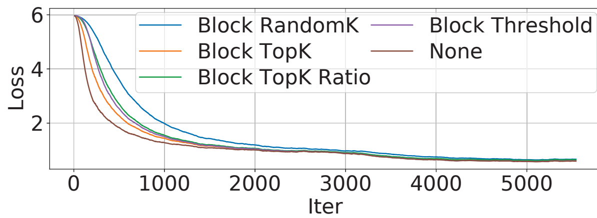


Figure 12: BERT median training (finetuning) loss of 10 runs. Data points are applied EMA smoothing with $\alpha = 0.5$.

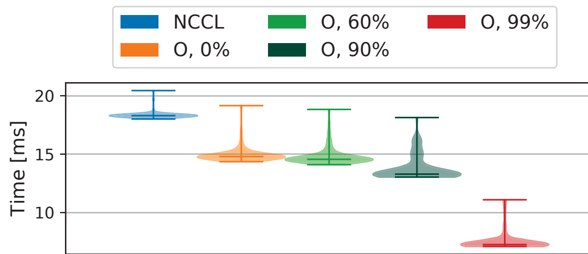


Figure 13: Time to complete AllReduce on 100 MB tensors in the multi-GPU scenario.

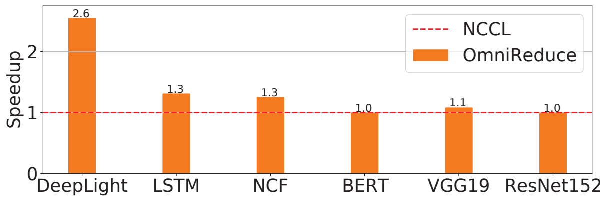


Figure 14: Training performance speedup to NCCL in the multi-GPU scenario.

6.2.3 OmniReduce speedup with block-based compression. We apply all 4 block-based compression methods introduced in §4 to speedup the BERT workload, which consists of a large model (1.2 GB), but its gradient sparsity is only ~ 9%. We use 0.1664 as threshold (which results in ~ 1% compression ratio) and otherwise apply $k = 1\%$ compression ratio. While evaluating the performance speedup relative to NCCL, we also track the model accuracy (F1 score). We repeat the experiments ten times and plot ranges with quartiles. We fine-tune BERT for the question answering task on the Stanford Question Answering Dataset [53].

Figure 11 shows these results for an 8-worker setup at 10 Gbps. OmniReduce now accelerates training by ~ 1.7×. The training loss change shown in Figure 12 reveals that block-based compression methods can preserve convergence for BERT. Compression affects accuracy slightly (at most a 1-point drop in F1 score), highlighting the trade-off between speedup and accuracy, which depends on the compression level.

6.3 Multi-GPU and multi-node scenario

Lastly, we evaluate OmniReduce in a multi-GPU and multi-node testbed consisting of 6 × 8-GPU servers as workers and 6 CPU servers as aggregators. We run both microbenchmarks and end-to-end training experiments.

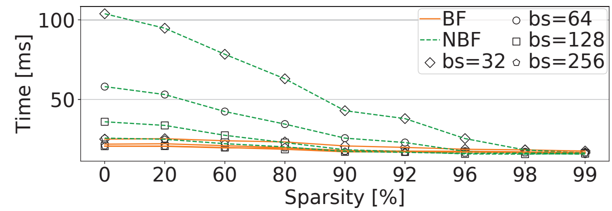


Figure 15: Influence of block size (*bs*) and sparsity. *BF* and *NBF* refer to OmniReduce w/ or w/o Block Fusion, respectively.

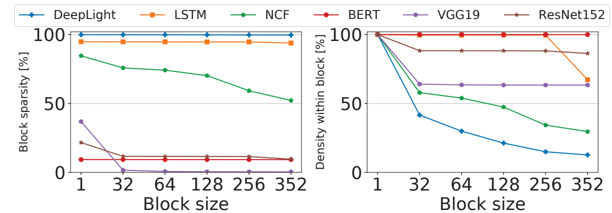


Figure 16: Block sparsity and density within block of gradients for DNN models.

Figure 13 shows the microbenchmark results as we vary the tensor sparsity. We follow a setup similar to §6.1. The results show that even in the multi-GPU and multi-node environment, OmniReduce always outperforms NCCL and achieves up to 2.5× speedup over NCCL at 99% sparsity.

Figure 14 shows the end-to-end training speedup of OmniReduce relative to NCCL in the multi-GPU and multi-node setup. For models with high sparsity like DeepLight, LSTM and NCF, OmniReduce has a speedup ranging 1.3× to 2.6×. Even for the models with low sparsity, OmniReduce does not negatively affect performance.

6.4 Sensitivity analysis

6.4.1 Block size. Figure 15 shows how block sparsity influences the performance of OmniReduce w/ and w/o Block Fusion for various choices of the block size. Without Block Fusion, OmniReduce is very sensitive to block size, especially for data with low sparsity. This is because a larger block size is better at amortizing the per-packet metadata overheads. The results demonstrate that the Block Fusion method improves the performance stability for OmniReduce.

Figure 16 shows the effective sparsity as a function of block size for various DNNs. A block size of one identifies the real gradient sparsity. Models with large embedding layers can maintain large block sparsity at packet-size blocks. Notably, the density of non-zero values within each block does not decrease too drastically in many cases. Given these characteristics of block sparsity in relation to performance and density within a block, we choose block size 256 as the default for our setting.

6.4.2 Overlap of non-zero blocks. Two extremes exist: (1) all non-zero blocks overlap at every worker, and (2) no non-zero block overlaps among N workers. Dense AllReduce (on the non-zero blocks only) and AGsparse ideally address these extremes, whereas OmniReduce – while capable of handling the entire spectrum – is best suited for when data is sparse and block overlap somewhat.

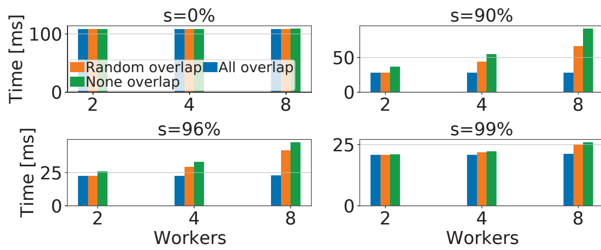


Figure 17: Effect of non-zero element overlap among workers on the OmniReduce performance.

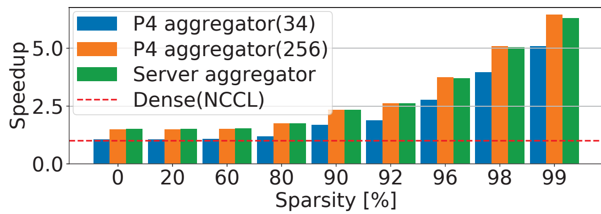


Figure 18: OmniReduce in-network P4 aggregator compared to server-based aggregator for two block sizes (34 and 256).

Figure 17 shows the AllReduce time for the extremes as well as with an amount of overlap generated at random. It is noteworthy that at both no sparsity ($s = 0\%$) or very high sparsity ($s > 95\%$), the impact of overlap is small or none. This is because the total number of elements of a dense tensor is equal to S in all cases, while NDS is close to DS if the tensor is very sparse. Recall that we denote D as the average data density and S the tensor size. However, when $s \in [60\%, 90\%]$, the "all overlap" performance is significantly better than the other cases.

7 EXTENSIONS

In-network aggregation. SwitchML [58], Mellanox SHARP [44] and ATP [38] recently demonstrated the feasibility of streaming collective aggregation protocols where the aggregation takes place within network switches. OmniReduce lends itself to these advancements. In particular, because the time and space complexity of the OmniReduce aggregator is low and the aggregation function is the arithmetic sum, we demonstrate that the aggregator can run on suitable network switches.

We implement Algorithm 2 in P4 [5] and offload it to a Barefoot Tofino switch. Figure 18 shows that with this offload, OmniReduce is slightly faster than with the server-based aggregator. This implementation inherits some of the limitations described by Sapio et al. [58] in terms of numeric representation and slot size. However, SHARPv2 [44] demonstrated that 100 Gbps line-rate aggregation of floating-point values is within reach for current technology and there is recent work exploring in-network sparse reductions [59].

Generalized collective operations. We observe that our algorithms generalize to three collective operations: AllReduce, AllGather, and Broadcast. In fact, AllGather can be viewed as a sparse AllReduce with no block overlap. Broadcast is a more straightforward case in which there is no block overlap, and the tensor size of $N-1$ workers is 0. In these cases, the aggregator realizes both a multicast function and flow control mechanism to coordinate collective communication. By not sending zero blocks, OmniReduce improves the efficiency for these collectives.

Numeric reproducibility and non-commutative operations. Due to the numeric representation of floating-point values, sum is not generally a commutative operator. OmniReduce can support numeric reproducibility and non-commutative operators by enforcing a serial order of slot updates. At the cost of a larger pool of slots, one can modify our algorithms so that every slot is writable by one worker at a time, in a pre-defined sequence, while pipelining slot updates for efficiency. For example, in an N worker group, worker 1 is $N-1$ blocks ahead of worker N, worker 2 is $N-2$ blocks ahead, and so on. The overhead for doing so is that slot aggregation latency increases with $O(\log_2 N)$; throughput, however, is unaffected. Signaling information to synchronize progress can be piggybacked by data packets to lower overheads.

8 OTHER RELATED WORK

Efficient communication in DDL. Several efforts optimize DDL communication ranging from designing high-performance PS software [43] and transfer scheduler [20, 25, 50], to improving collective communication in heterogeneous networks fabrics [10, 28] and within multi-GPU servers [66], to developing in-network reduction systems [35, 39, 44, 57, 58], to customizing network congestion protocols and architecture [18]. OmniReduce leverages data sparsity to optimize communication and is complementary to these efforts.

Accelerating DDL. Orthogonal to our work, various works propose efficient distributed optimization algorithms [4, 36, 41, 72]. Besides data parallelism, other parallelization strategies include model parallelism [9, 11], and hybrids of model and data parallelism [26, 27]. Going one step further, pipeline parallelism [46] processes multiple batches simultaneously, with individual layers either having model or data parallelism. OmniReduce speeds up the data parallel aspect of these works.

9 CONCLUSION

We leverage sparsity in distributed deep learning to accelerate training for six real-world DNNs by up to 8.2×. OmniReduce is a generic collective communication primitive aiming especially at efficiently aggregating sparse data. We proposed streaming aggregation algorithms that outperform previous approaches, surpassing them by 3.5–16×. Our approach runs efficiently on a server, yet its modest computational complexity affords it to run on programmable switch ASICs. OmniReduce has already spurred adoption for large scale training workloads at Meituan.

Acknowledgments

We are grateful to Arvind Krishnamurthy, Jacob Nelson and Dan R. K. Ports for their helpful suggestions. We are thankful to Meituan for granting us access to a multi-GPU server testbed. We thank our shepherd, Kate Lin, and the anonymous reviewers for their helpful feedback. This publication is based upon work supported by the King Abdullah University of Science and Technology (KAUST) Office of Sponsored Research (OSR) under Award No. OSR-CRG2020-4382. For computer time, this research used the resources of the Supercomputing Laboratory at KAUST. The work of Jiawei Fei at KAUST is supported by a sponsorship from China Scholarship Council (CSC). This work was partially supported by a gift in kind from Huawei.

REFERENCES

[1] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. In *EMNLP-IJCNLP*.

[2] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *NeurIPS*.

[3] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. 2018. The Convergence of Sparsified Gradient Methods. In *NeurIPS*.

[4] Debraj Basu, Deepesh Data, Can Karakus, and Suhas Diggavi. 2019. Qsparse-local-SGD: Distributed SGD with Quantization, Sparsification, and Local Computations. In *NeurIPS*.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014).

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[7] bytedance/byteps. 2019. A High Performance and Generic Framework for Distributed DNN Training. https://github.com/bytedance/byteps.

[8] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2013. One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling. arXiv:1312.3005 [cs.CL]

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

[10] Minsik Cho, Ulrich Finkler, David S. Kung, and Hillery C. Hunter. 2019. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In *MLSys*.

[11] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. 2012. Large Scale Distributed Deep Networks. In *NeurIPS*.

[12] Wei Deng, Junwei Pan, Tian Zhou, Deguang Kong, Aaron Flores, and Guang Lin. 2020. DeepLight: Deep Lightweight Feature Interactions for Accelerating CTR Predictions in Ad Serving. arXiv:2002.06987 [cs.LG]

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[14] Nikoli Dryden, Sam Ade Jacobs, Tim Moon, and Brian Van Essen. 2016. Communication Quantization for Data-Parallel Training of Deep Neural Networks. In *MLHPC*.

[15] Aritra Dutta, El Houcine Bergou, Ahmed M. Abdelmoniem, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Panos Kalnis. 2020. On the Discrepancy between the Theoretical Analysis and Practical Implementations of Compressed Communication for Distributed Deep Learning. In *AAAI*.

[16] Facebook. 2021. Gloo. https://github.com/facebookincubator/gloo.

[17] Message Passing Interface Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. University of Tennessee.

[18] Nadeen Gebara, Paolo Costa, and Manya Ghobadi. 2021. In-network Aggregation for Shared Machine Learning Clusters. In *MLSys*.

[19] F Maxwell Harper and Joseph A Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4 (Dec. 2015).

[20] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. 2019. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *MLSys*.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

[22] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *WWW*.

[23] Michael Hofmann and Gudula Rünger. 2008. MPI Reduction Operations for Sparse Floating-point Data. In *EuroPVM/MPI*.

[24] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*.

[25] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based Parameter Propagation for Distributed DNN Training. In *MLSys*.

[26] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *ICML*.

[27] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *MLSys*.

[28] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*.

[29] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the Limits of Language Modeling. arXiv:1602.02410 [cs.CL]

[30] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. 2019. Error Feedback Fixes SignSGD and other Gradient Compression Schemes. In *ICML*.

[31] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *ICLR*.

[32] Janis Keuper and Franz-Josef Pfreundt. 2016. Distributed Training of Deep Neural Networks: Theoretical and Practical Limits of Parallel Scalability. In *MLHPC*.

[33] J. Kiefer and J. Wolfowitz. 1952. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics* 23, 3 (1952), 462–466.

[34] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *EuroSys*.

[35] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. 2020. An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives. In *ISCA*.

[36] Anastasia Koloskova, Sebastian U Stich, and Martin Jaggi. 2019. Decentralized Stochastic Optimization and Gossip Algorithms with Compressed Communication. In *ICML*.

[37] Kelly Kostopoulou, Hang Xu, Aritra Dutta, Xin Li, Alexandros Ntoulas, and Panos Kalnis. 2021. DeepReduce: A Sparse-tensor Communication Framework for Distributed Deep Learning. arXiv:2102.03112 [cs.LG]

[38] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *NSDI*.

[39] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *ISCA*.

[40] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. 2018. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *Micro*.

[41] Tao Lin, Sebastian U Stich, Kumar Kshitij Patel, and Martin Jaggi. 2019. Don't Use Large Mini-batches, Use Local SGD. In *ICLR*.

[42] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *ICLR*.

[43] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2018. PHub: Rack-Scale Parameter Server for Distributed Deep Neural Network Training. In *SoCC*.

[44] Mellanox. 2021. Scalable Hierarchical Aggregation and Reduction Protocol (SHARP). https://www.mellanox.com/products/sharp.

[45] Microsoft. 2015. Criteo's 1TB Click Prediction Dataset. https://docs.microsoft.com/en-us/archive/blogs/machinelearning/now-available-on-azure-ml-criteos-1tb-click-prediction-dataset.

[46] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *SOSP*.

[47] NVIDIA. 2021. Ampere Architecture In-Depth. https://devblogs.nvidia.com/nvidia-ampere-architecture-in-depth/.

[48] NVIDIA. 2021. Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.

[49] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009).

[50] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP*.

[51] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. arXiv:1802.05365 [cs.CL]

[52] pytorch/pytorch. 2019. Support sparse gradients in DistributedDataParallel. https://github.com/pytorch/pytorch/pull/22037.

[53] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know What You Don't Know: Unanswerable Questions for SQuAD. arXiv:1806.03822 [cs.CL]

[54] Cedric Renggli. 2019. SparCML. https://gitlab.com/rengglic/SparCML.

[55] Cedric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. 2019. SparCML: High-Performance Sparse Communication for Machine Learning. In *SC*.

[56] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge.

International Journal of Computer Vision 115, 3 (2015).
[57] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *HotNets*.
[58] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *NSDI*.
[59] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: Flexible In-Network Allreduce. arXiv:2106.15565 [cs.DC]
[60] Hongzhang Shan, Samuel Williams, and Calvin W. Johnson. 2018. Improving MPI Reduction Performance for Manycore Architectures with OpenMP and Data Compression. In *PMBS*.
[61] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*.
[62] Sebastian U. Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified SGD with Memory. In *NeurIPS*.
[63] Nikko Strom. 2015. Scalable Distributed DNN Training Using Commodity GPU Cloud Computing. In *ISCA*.
[64] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.* 19, 1 (Feb. 2005).
[65] Jesper Larsson Träff. 2010. Transparent Neutral Element Elimination in MPI Reduction Operations. In *EuroMPI*.
[66] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. 2020. Blink: Fast and Generic Collectives for Distributed ML. In *MLSys*.
[67] 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 *NeurIPS*.
[68] Hang Xu, Chen-Yu Ho, Ahmed M. Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. 2021. GRACE: A Compressed Communication Framework for Distributed Machine Learning. In *ICDCS*.
[69] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. 2020. Is Network the Bottleneck of Distributed Training?. In *NetAI*.
[70] Huasha Zhao and John Canny. 2014. Kylix: A Sparse Allreduce for Commodity Clusters. In *ICPP*.
[71] Shuai Zheng, Ziyue Huang, and James Kwok. 2019. Communication-Efficient Distributed Blockwise Momentum SGD with Error-Feedback. In *NeurIPS*.
[72] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. 2017. Asynchronous Stochastic Gradient Descent with Delay Compensation. In *ICML*.

Appendices are supporting material that has not been peer-reviewed.

A PACKET LOSS RECOVERY

We now extend our design to support packet retransmission to account for lossy network environments. First, we revisit Algorithm 1 and see how it would fail in the presence of packet loss. A packet loss in the upward path from worker to aggregator prevents the aggregator from completing block aggregation. Whereas, the loss of one of the result packets sent to the workers on the downward path (aggregator to worker) not only keeps a worker from obtaining the aggregated block, but may also stop the worker from sending the next block, and halt the entire aggregation.

To tolerate packet loss, we include acknowledgment packets and use a timer mechanism to detect losses. Further, the aggregator keeps two versions of its per-slot state. The revised algorithm is listed in Algorithm 2. Note that this description includes the pool of S slots, one per stream used by independent worker threads.

Every time the worker receives a packet, it responds to the aggregator for the requested block. However, when the aggregator requests a block that the worker would not send (a zero block), the worker only sends an ack. packet with no payload. The worker associates a timer to every transmitted packet; if the timer fires, the worker assumes packet loss and retransmits it. The aggregator has a *count* of aggregated packets; a result packet is sent only once the *count* reaches the number of workers N. To avoid (incorrectly) aggregating duplicate transmissions, the aggregator maintains a boolean vector *seen* that tracks which worker's packet has been processed.

Put together, the approach above ensures that single-sided timers are sufficient to recover from packet loss, regardless of whether a loss occurs on the upward or downward path. However, the aggregator must be able to retransmit a dropped result packet to worker i even after a different worker j has already sent its next non-zero block addressing the same slot. This requires two versions of each *slot* that are used in alternate phases. When the worker receives the resulting packet from the aggregator, it changes the *slot* version by flipping $p.ver$ before sending the next block to the aggregator. Each version of a *slot* gets reused only when it is certain that all workers have received the aggregated result in that *slot*. This happens when all workers have sent their blocks to the other version of that *slot*, signaling that all workers have moved forward.

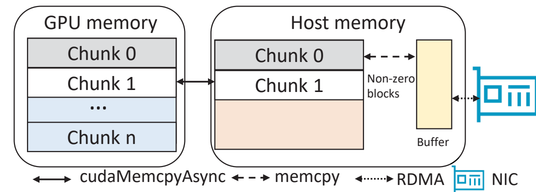


Figure 19: Workflow of chunk prefetch optimization for systems without GDR support.

B IMPLEMENTATION DETAILS

In OmniReduce, only non-zero blocks will be copied to the transmission buffer. Nevertheless, copying blocks from GPU memory to host memory is still the bottleneck when the network bandwidth is close to the PCIe bandwidth (128Gbps for PCIe gen3). Because OmniReduce only copies one block (<1KB) at a time, and we find small data copy between GPU and host inefficient, We propose and implement two solutions respectively for systems with and without GDR support as follows:

GPU-direct RDMA. GDR only supports GPUs connected to the same PCIe switch as RDMA NIC. For machines that support GDR, we do not use buffers. Instead, we send non-zero blocks directly from GPU to aggregators with the support of GDR. OmniReduce has larger benefits with GDR as it reduces the PCIe traffic.

Chunk prefetch. Figure 19 shows our solution when machines do not support GDR. Firstly, we copy all the data (including zero and non-zero blocks) from GPU to host in chunk (4MB) asynchronously (*cudaMemcpyAsync*). At the same time, worker threads check the completion of the copy event (*cudaEventSynchronize*) and will copy non-zero block from the completed chunk to the communication buffer. Once blocks are received from aggregators, they will be written to the chunk in host memory, and this chunk will be copied to the GPU memory (*cudaMemcpyAsync*) if all blocks in it are aggregated completely. With this solution, the memory copy operation between GPU and host is almost completely overlapped with the communication.

Algorithm 2: Block aggregation w/ loss recovery

```
1  At Worker:
2    p.next, next ← first non-zero block past block 0
3    p.block, p.ver ← 0
4    p.stream ← stream/thread ID s
5    p.wid ← worker ID
6    p.data ← G_s[0 : bs]
7    send p to agg; start_timer(p)
8    repeat upon receive p(data, ver, block, next, stream, wid)
9      cancel_timer(p)
10     G_s[p.block : p.block + bs] ← p.data
11     p.ver ← (p.ver + 1)%2
12     if p.next = next then
13       p.block ← next
14       p.data ← G_s[next : next + bs]
15       p.next, next ←next non-zero block or else ∞
16       p.wid ← worker ID
17       send p to agg; start_timer(p)
18     else
19       p.next ← next
20       p.data ← {0}                        // empty packet payload
21       send p to agg; start_timer(p)
22   until p.next = ∞
23   upon timeout for p                      // timeout handler
24     send p to agg; start_timer(p)

25 At Aggregator:
26   for s in 0 . . . S − 1 do               // pool initialization, 2-way versioned
27     slot_s[2] := {0}
28     seen_s[2, N], count_s[2] := {0}
29     min_next_s := ∞
30   forever upon receive p(data, ver, block, next, stream, wid)
31     s ← p.stream                          // reference p's slot
32     if seen_s[p.ver, p.wid] = 0 then
33       seen_s[p.ver, p.wid] ← 1
34       seen_s[(p.ver + 1)%2, p.wid] ← 0
35       count_s[p.ver] ← (count_s[p.ver] + 1)%N
36       if count_s[p.ver] = 1 then
37         slot_s[p.ver] ← p.data
38         min_next_s ← p.next
39       else
40         slot_s[p.ver] ← slot_s[p.ver] + p.data
41         min_next_s ← min(min_next_s, p.next)
42       if count_s[p.ver] = 0 then
43         p.data ← slot_s[p.ver]
44         p.next ← min_next_s
45         send p to all workers
46     else
47       if count_s[p.ver] = 0 then
48         p.data ← slot_s[p.ver]
49         send p to p.wid
```

Algorithm 3: Extension to sparse format

```
1  At Worker:
2    nextkey_idx := bs
3    p.nextkey := K[nextkey_idx]
4    p.keys ← K[0 : bs]
5    p.values ← V[0 : bs]
6    p.wid ← Worker ID
7    send p to agg
8    repeat upon receive p(keys, values, nextkey, wid)
9      update K, V according to p.keys, p.values
10     if p.nextkey ≥ K[nextkey_idx] then
11       p.keys ← K[nextkey_idx : nextkey_idx + bs]
12       p.values ← V[nextkey_idx : nextkey_idx + bs]
13       p.nextkey ← K[nextkey_idx + bs]
14       nextkey_idx ← nextkey_idx + bs
15       p.wid ← Worker ID
16       send p to agg
17     end
18   until update K, V is complete
19 At Aggregator:
20   nextkey[N] := {−∞}
21   sent := 0
22   forever upon receive p(keys, values, nextkey, wid)
23     nextkey[p.wid] ← p.nextkey
24     send_up_to ← min(nextkey)
25     update K, V accordingly
26     if send_up_to > sent then
27       p.keys ← keys from sent to send_up_to in K
28       p.values ← values from sent to send_up_to in V
29       p.nextkey ← send_up_to
30       sent ← send_up_to
31       send p to all workers
32     end
```

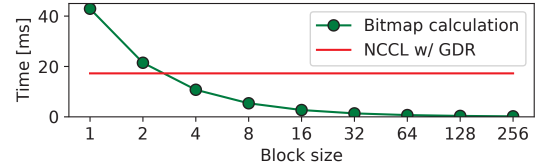


Figure 20: Time cost comparison between bitmap calculation and NCCL w/ GDR AllReduce on 100MB float tensor. The block size refers to the number of floats in one block. The GPU we use is V100.

B.1 Bitmap calculation

To determine non-zero blocks, we use GPU to calculate a bitmap (one bit per block). This function runs whenever a part of the gradient is ready for aggregation. Figure 20 shows that small (< 4) block sizes degrade the bitmap calculation performance greatly. We only use block sizes greater than 16 in OmniReduce as we find that bitmap calculation overheads are negligible in this case. Our OmniReduce implementation currently only supports AllReduce for GPU data as we use the GPU to efficiently calculate the bitmap.

C PROOF OF CONVERGENCE

We first start with the definition of a δ-compressor, then prove that *Block Random-k*, and *Block Top-k* are δ-compressors, and then finally relate it to the convergence result for δ-compressors.

Definition. ***(δ-compressor) [30]*** *A probabilistic operator $C : \mathbb{R}^d \to \mathbb{R}^d$ is called a δ-approximate compressor for $\delta \in (0, 1]$ if*

$$\mathbb{E}\|x - C(x)\|_2^2 \leq (1 - \delta)\|x\|_2^2 \quad \forall x \in \mathbb{R}^d.$$

Lemma. *Let b denote the total number of blocks. Both block-rand$_k$ (Block Random-k) and block-top$_k$ (Block Top-k) are δ-compressors with $\delta = \frac{k}{b}$.*

Proof. *Let for any $x \in \mathbb{R}^d$, $x[i] \in \mathbb{R}^{\lceil d/b \rceil}$ denote the ith block. Then, $x = [x[1]^\top, x[2]^\top, \ldots, x[b]^\top]^T$. Also, let $\Omega_k = \binom{[b]}{k}$ denote the set of all k element subsets of $[b]$.*
Block Random-k: We have,

$$\begin{aligned}\mathbb{E}\|x - \textit{block-rand}_k(x)\|_2^2 &= \frac{1}{|\Omega_k|}\sum_{\omega\in\Omega_k}\sum_{i=1}^{b}\|x_i\|_2^2 \cdot \mathbb{I}\{i \notin \omega\} \\ &= \sum_{i=1}^{b}\|x_i\|_2^2 \sum_{\omega\in\Omega_k}\frac{\mathbb{I}\{i\notin\omega\}}{\Omega_k} \\ &= (1 - \frac{k}{b})\|x\|_2^2,\end{aligned}$$

which implies that block-rand$_k$ is a δ-compressor with $\delta = \frac{k}{b}$.

Block Top-k: Let $S_{block\text{-}top_k}(x)$ denote the set of Top-k blocks corresponding to a given x. Then,

$$\begin{aligned}\|x - \textit{block-top}_k(x)\|_2^2 &= \|x\|_2^2 + \|\textit{block-top}_k(x)\|_2^2 \\ &\quad - 2\langle x, \textit{block-top}_k(x)\rangle \\ &= \|x\|_2^2 - \|\textit{block-top}_k(x)\|_2^2 \\ &= \|x\|_2^2 - \sum_{i\in S_{block\text{-}top_k}(x)}\|x[i]\|_2^2 \\ &\leq \|x\|_2^2 - \frac{k}{b}\|x\|_2^2 \\ &= (1-\frac{k}{b})\|x\|_2^2,\end{aligned}$$

where the inequality follows from the fact

$$\frac{\sum_{i\in S_{block\text{-}top_k}(x)}\|x[i]\|_2^2}{k} \geq \frac{\sum_{i=1}^{b}\|x[i]\|_2^2}{b}.$$

The above implies that block-top$_k$ is a δ-compressor with $\delta = \frac{k}{b}$.

Using the above lemma, Theorem 1 in [71] gives us the convergence result for compressed distributed SGD with error-feedback for an arbitrary δ-compressor.

D LOSS RECOVERY PERFORMANCE

We show how different packet loss rates (between 0.01% and 1%) affect DPDK-based OmniReduce. In our setup, we do not need to handle packet loss in RDMA network as we use Reliable Connected (RC) mode. Since no packet loss actually occurs in our experiments, we emulate packet loss assuming uniform probability at a given loss rate.

Figure 21 shows the difference between AllReduce time with no loss minus AllReduce time with a given loss rate. We compare against Gloo and NCCL while using TCP as transport protocol in order to see a reaction to packet drops. The results show that OmniReduce's packet retransmission is effective in every sparsity level and loss rate. However, with a high loss rate (1%), the performance of Gloo and NCCL-TCP drops sharply. We attribute this to TCP congestion control.

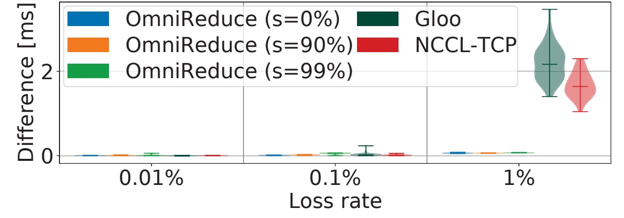


Figure 21: Performance drop of AllReduce time due to packet loss and recovery. No packet loss is the baseline.

E ARTIFACT APPENDIX

Abstract

The main artifact accompanying this paper is the implementation of OmniReduce, along with scripts and models to benchmark its performance.

Scope

The artifact allows to validate the results of this paper with regard to the performance of OmniReduce both with microbenchmarks and in the end-to-end training setting.

Contents

- omnireduce: the code implementing the worker and aggregator components of OmniReduce.
- docs: documentation, including instructions for reproducing the experiments.
- benchmark: a benchmark to perform AllReduce on tensors with different sparsity to test the performance of OmniReduce and NCCL.
- models: the DNN models used in end-to-end training.
- notebooks: Jupyter notebooks to process and plot results.

Hosting

The artifact is available on GitHub at https://www.github.com/sands-lab/omnireduce.

Requirements

Our experiments require 8 CPU servers used as aggregators and 8 GPU servers used as workers. Each GPU server requires one GPU but can have more. Hardware support for DPDK, RDMA and GDR is required to use those features. All machines should be interconnected by a full-bisection network fabric.