Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 179

Systems Group, Department of Computer Science, ETH Zurich

Efficient Sparse AllReduce For Scalable Machine Learning

by

Cedric Renggli

Supervised by

Prof. Dr. Torsten Hoefler
Dr. Dan Alistarh

April 2017–September 2017

inf | Informatik
Computer Science

Abstract

We design and implement efficient MPI AllReduce operations when the input data vectors are *sparse*, that is, have a high fraction of neutral elements with respect to a given binary operator. This task, which we call the *sparse AllReduce* problem, strictly generalizes standard MPI collectives, by allowing processes to contribute input data vectors of *heterogeneous sizes*, since neutral elements can be simply ignored. The goal is to design algorithms which minimize the overall communication cost for each node to obtain the result of the AllReduce operation. We present a set of efficient algorithms for varying sparsity, distribution of non-neutral elements in the input vectors, number of processes, and communication size.

We validate our algorithmic results experimentally on a set of large scale machine learning applications, showing that we can leverage sparsity for practical runtime savings, compared to standard MPI Reduction operations. When sparsity is not given naturally, we develop an efficient algorithm for selecting the top k absolute values in expectation of a given vector. This routine, combined with the variants of the sparse AllReduce algorithms, is included into a state-of-the-art deep learning framework. A lightweight generic framework is developed in order to train large scale linear classifiers on various architectures making use of sparse AllReduce calls. We achieve major speedup for training linear classification models on a numerous number of nodes, and see significant reduction on the communication part when training large scaled deep learning models on a supercomputer.

Notation

Variable	Description
P	Number of nodes
N	Dimension
p_i	Node i, $1 \leq i \leq P$
H_i	Set of non-neutral indices at p_i
k_i	Cardinality at each node: $\lvert H_i \rvert$
k	Max number of non-neutral elements: $\max_i k_i$
$\mathcal{K}$	Size of result: $\lvert \cup_{i=1}^{P} H_i \rvert \quad (k \leq \mathcal{K} \leq \min\{N, k \times P\})$
d	Density of non-neutral elements: $\frac{k}{N}$
M	Number of training samples
B	Mini-batch sizes per nodes

Contents

Notation iii

Contents v

1 Introduction and Motivation 1
1.1 Related Work 2
1.2 Contributions 3

2 Scalable Machine Learning 5
2.1 Data-Parallel SGD 5
2.2 Linear Classifiers 8
2.2.1 Logistic Regression 8
2.2.2 Support Vector Machines (SVM) 9
2.2.3 MPI-SGD 9
2.3 Deep Neural Networks 9
2.3.1 Fully Connected Neural Networks 10
2.3.2 Convolutional Neural Networks 10
2.3.3 RNN / LSTM 10
2.3.4 The Microsoft Cognitive Toolkit (CNTK) 11
2.4 Top$K$ SGD 12
2.4.1 Approximate Top$K$ SGD 14

3 The Sparse AllReduce Problem 17
3.1 Formulation 17
3.1.1 Analytical Model 17
3.1.2 Nodes and Elements 18
3.1.3 Generalization 19
3.1.4 Simplification and Assumptions 20
3.2 Resulting Size 20
3.2.1 Uniform Distribution 21

3.3 Algorithms . . . 22
3.3.1 Known Approaches . . . 23
3.3.2 Dense vs. Sparse Representation . . . 26
3.3.3 Static Sparse AllReduce . . . 27
3.3.4 Dynamic Sparse AllReduce . . . 29
3.3.5 Discussion . . . 31

4 Implementation 35
4.1 Sparse AllReduce Library . . . 35
4.1.1 Sparse Vector Representation . . . 35
4.1.2 Auto-switch to Dense Format . . . 37
4.1.3 Sparse Vector Summation . . . 38
4.1.4 Computation and Communication Overlap . . . 42
4.1.5 Non-Blocking Variants . . . 42
4.2 MPI-SGD . . . 43
4.2.1 Linear Models . . . 44
4.2.2 Aggregation Strategies . . . 45
4.3 CNTK Extensions . . . 46
4.3.1 Brainscript Parameters . . . 46
4.3.2 Memory Allocation . . . 46
4.3.3 Approximate Top$K$ Selection . . . 47

5 Experiments 49
5.1 Experimental Setup . . . 49
5.2 Synthetic Data - Micro Benchmarks . . . 50
5.2.1 SSAR_Rec_Dbl vs. SSAR_Split_AlGa . . . 50
5.2.2 Static vs. Dynamic . . . 51
5.2.3 Early Switching DSAR . . . 52
5.2.4 All Algorithms . . . 53
5.2.5 Sparse Summation . . . 54
5.2.6 Lower Bounds . . . 54
5.3 Training of Linear Classifiers . . . 55
5.3.1 Datasets . . . 55
5.3.2 Logistic Regression . . . 55
5.3.3 SVM . . . 56
5.3.4 Sparsity Analysis . . . 57
5.4 Training of Deep Neural Networks . . . 58
5.4.1 Datasets . . . 58
5.4.2 Image Classification . . . 59
5.4.3 Natural Language Understanding . . . 62

6 Discussion 63

Bibliography 65

Chapter 1

Introduction and Motivation

Collective operations play a critical role in parallel systems and are a major component of message passing interface (MPI) standards [16]. Among these, some of the most frequently used are many-to-many reduction operations such as MPI_Allreduce, which logically collects data items (or a vector of items) from each process, applies the reduction operation (e.g. the sum), and returns the result to all participating processes. Many-to-many reductions are central in data-parallel applications such as neural network training [13], where processors compute independently over data batches, and periodically update their local model copies by taking the average over all node updates.

Our work starts from the observation that many of the built-in MPI binary reduction operators, e.g. MPI_SUM, have neutral elements (i.e. zero) for several data types. By ignoring these neutral elements, we obtain a new *sparse* version of the problem, which we call the *sparse AllReduce* problem. Sparse AllReduce generalizes MPI many-to-many reductions as each process is now allowed to contribute vectors of different sizes, since we ignore neutral elements. The intermediate resulting vectors are also heterogeneous in size, because the number of neutral elements can vary after every reduction step. Depending on the sparsity (fraction of neutral elements) of input vectors, the amount of communication and computation can be significantly reduced when comparing to a dense instance, which can lead to performance improvement.

Solving the sparse AllReduce problem efficiently is far from trivial. Even in the dense case, the selection of a reduction structure is often largely based on the input vector sizes and number of participating nodes. For instance, trees are efficient for small data sizes, whereas linear pipelines or rings are preferable for large amounts of data [26]. Moreover, sparse AllReduce operations introduce the additional challenge of *data dependency*, since the size of input vectors depends on the number of neutral elements, and the behav-

ior of the algorithm should adapt to the density of intermediate resulting vectors. This makes the problem significantly more challenging, since the algorithm must fit the data distribution in order to perform efficiently.

There are many practical settings in which the data to be communicated is sparse. An immediate example is computation of large scale graph data, which is known to follow a power-law degree distribution, see e.g. Zhao et al. [45]. Another setting which can benefit from such a primitive is the data-parallel training of machine learning models, widely used to scale training over large datasets and large models [40, 10]. Here, working nodes process batches of examples in parallel and periodically synchronize their model replicas by calculating the average over all the processors updates. Although the models are very large, counting millions of parameters, each of the processors model updates may be sparse naturally, or enforced by postponing the contribution of small valued elements [14, 1]. An efficient implementation of a sparse AllReduce can take advantage of this sparse structure, leading to significant runtime improvements in such scenarios.

1.1 Related Work

The problem of designing efficient algorithms for dense collective operations, such as AllReduce and AllGather, is a classic one in parallel computing. We limit ourselves by covering related work on sparse collectives, and refer the reader to Rabenseifner et al. [36], and Hoefler et al. [26] for complete surveys in the dense case.

The first reference to explicitly consider the sparse Allreduce problem is the system Kylix [45], which considers sparse many-to-many reductions in the context of computation over large scale distributed graph data on community clusters. The same problem is solved naïvely on large scale graph communication packages such as GraphLab [30] and PowerGraph [17]. Kylix proposes a nested, heterogeneous-degree butterfly network architecture for aggregating the data, in which the butterfly degree changes from one layer to the next, hence the name. Values first pass “down” through the network (implementing a ReduceScatter), and then back up through the same nodes (implementing an AllGather). The authors show both, analytically and experimentally (on large graph computation benchmarks), that this solution scales well on power-law distributed data. Machine learning tests with varying in and out vertices after each AllReduce call are omitted in their work. To overcome the fact that the problem is data-dependent, the authors of Kylix propose to run the downward pass twice, once a “configuration” step calculating the overlap of input indices, and the second time to perform the actual ReduceScatter. This configuration pass is needed if the input changes after each iteration, e.g. present in the context of machine learning. Dis-

tributed graph algorithms can highly benefit from this configuration pass, as the in- and output indices remain constant.

There is an extremely rich amount of work studying algorithms and systems for optimizing distributed large scale learning by adapting data-parallel stochastic gradient descent (SGD). One direction of research focuses on updating the model in a non-blocking fashion using an asynchronous architecture. Recent papers by Duchi et al. [15] and Lian et al. [29] give proofs on the convergence for convex and non-convex cases respectively. An orthogonal approach to this and to our work, focuses on compressing communication. Seide et al. [40] propose an 1-Bit SGD version, where every model update value is compressed to its sign. QSGD by Alistarh et al. [3] even goes a step further and proposes a family of communication-efficient adaptations of classical SGD, based on the idea of compressing gradients at each node to an arbitrary number of bits. An alternative idea to data-parallel SGD is to split the model amongst various workers. Dean et al. [13] uses this technique combined with data-parallelism for training deep neural networks.

The work closest to ours is that of Dryden et al. [14], which studies ways to reduce the overheads of communication in data-parallel training of large scale neural networks. They propose a variant of the classical SGD training algorithm in which every node only sends its biggest k update values at each iteration, accumulating the other values locally. This variant renders the updates sparse, and the authors show that the delayed elements do not hurt convergence speed on the MNIST image classification dataset. The speedup achieved compared to MPI_AllReduce is given at a factor of 1.18x. The authors implement a sparse variant of the classical AllReduce algorithm via a pairwise reduce-scatter followed by a ring-based AllGather. The amount of data is kept constant at every stage of their algorithm by re-selecting the top k values and postponing the other received values, contrary to our generic sparse AllReduce formulation.

A second paper suggesting a Top$K$ approach for training deep neural networks is given by Aji et al. [1]. The authors perform neural machine translation on 4 GPUs achieving a speedup of factor 1.22x by mapping the 99% smallest values to zero and then exchange sparse matrices using asynchronous SGD following a parameter-server paradigm. Additionally, Aji et al. conducted an experiment on the MNIST dataset using a fully connected feed forward neural network similar to the one given by Dryden et al. A speedup of a factor 1.49x is achieved by dropping 99% of the gradient.

1.2 Contributions

The main contribution of this thesis lies in the formal definition of the sparse AllReduce problem and the design of communication efficient algorithms

with analytical runtime bounds. Those procedures are verified accurately based on synthetic micro benchmarks. For enforcing sparsity efficiently, an approximate adaptation of Top$K$ SGD, taking k elements in expectation, is proposed.

Numerous experiments are conducted on large scale machine learning applications. For training linear classifiers on various architectures, MPI-SGD, a lightweight generic framework, is developed. The sparse AllReduce algorithm variants are fused into MPI-SGD in order to leverage natural sparsity of the model updates when training large scale binary classification models. We achieve a speedup of up to factor 20x on an infiniband cluster compared to a dense AllReduce version. Finally, the adapted version of Top$K$ SGD combined with our designed sparse AllReduce algorithms is incorporated into a deep learning framework, in order to achieve a speedup of up to a factor 4x on large scaled deep learning models.

All the positive and negative findings are described in the experiment section. The discussion chapter offers an overview over all the results as well as potential topics to further investigate.

Chapter 2

Scalable Machine Learning

In this chapter we give a short but thorough overview over current scalable machine learning topics. Large scale may refer to two, not necessarily exclusive, properties of learning problems:

1. Large number of training samples
2. Large dimension of the models to train.

We give an introduction to the most widely used and theoretically best understood optimization techniques for a large number of samples, some supervised machine learning problems with big models and the frameworks to train those models. Last, but not least, we quickly introduce a variant of data-parallel SGD, Top$K$ SGD, which aims to enforce sparsity in the gradient updates while preserving convergence.

2.1 Data-Parallel SGD

We briefly give an introduction to data-parallel SGD, the most commonly used method for training large scale machine learning problems. For a more detailed overview over optimization methods used in machine learning, we refer to Bottou et al. [5]. His technical report also gives a profound overview on how to scale machine learning to a large number of nodes. However, we formally introduce data-parallel SGD and show how one could scale this method with highlighting potential pitfalls.

Let us have any function of the form:

$$f(\mathbf{w}) = \frac{1}{M} \sum_{i=1}^{M} f_i(\mathbf{w}),$$

with $\mathbf{w} \in \mathbb{R}^N$ representing the model we want to train. In terms of empirical risk minimization, f_i represents the loss given for a single sample, whilst

we try to minimize the loss over the entire training set. We give various examples for such loss functions in the next two chapters. A naïve way to determine the model, which minimizes f, is to apply Gradient Descent (GD). The method iteratively changes the current model parameters by subtracting the gradient of the loss multiplied by some learning rate.

$$\mathbf{w}_{t+1} = \eta_t \nabla f(\mathbf{w}_t)$$

The main problem with this method consists of the fact, that one needs to do a pass over all the training samples in order to calculate the gradient of f and perform one iterative step. A stochastic approach for solving this is to choose one sample j uniformly at random, and than perform the update step solely based on the gradient for this function f_j. This method is called Stochastic Gradient Descent (SGD).

$$\mathbf{w}_{t+1} = \eta_t \nabla f_j(\mathbf{w}_t)$$

Notice that this gradient is unbiased, so even when we introduce some variance, in expectation we move towards a local minimum. Convergence and adaptation of the learning rate η_t has been studied extensively in the stochastic optimization literature. The original paper suggesting SGD by Robbins and Monro [38] states, the learning rate has to satisfy both $\sum_{t=0}^{\infty} \eta_t = \infty$ and $\sum_{t=0}^{\infty} \eta_t^2 < \infty$, in order to ensure convergence.

For reducing the variance at each stochastic gradient, instead of taking one single sample uniformly at random, one can take a mini-batch of size B samples and calculates the stochastic gradient based on those samples.

$$\mathbf{w}_{t+1} = \eta_t \nabla \sum_{j=1}^{B} f_j(\mathbf{w}_t)$$

This gradient is still unbiased. In practice, Gurbuzbalaban et al. [20] noticed, that instead of uniformly sampling those mini-batches, one can simply shuffle the entire dataset, split it into batches of size B and run an update step based on each batch gradient. Such a pass over the entire dataset is commonly called an *epoch*. This leads to mini-batch data-parallel SGD (later only referred as data-parallel SGD or classical SGD) shown in Algorithm 1.

Algorithm 1 Data-Parallel Mini-Batch SGD

1: Initialize $\mathbf{w}$
2: **for** $t = 1, ..., T$ **do**
3: *Randomly shuffle training samples*
4: *Partition all samples into batches:* $\{b_i \mid 1 \leq i \leq \lceil \frac{M}{B \times P} \rceil \}$
5: **for** $i = 1, ..., \lceil \frac{M}{B \times P} \rceil$ **do**
6: *Partition samples* b_i *into batches* $\{b_{ij} \mid 1 \leq j \leq P\}$
7: **for** machine $j = 1, ..., P$ **do in parallel**
8: $\Delta \mathbf{w}_j = \eta_{t_i} \sum_{n \in b_{ij}} \nabla f_n(\mathbf{w})$
9: **end for**
10: $\mathbf{w} \leftarrow \mathbf{w} - \sum_{j=1}^{P} \Delta \mathbf{w}_j$
11: **end for**
12: **end for**

When looking at the algorithm carefully, one notices that the theoretical batch size is not B, but rather $B \times P$. This is of importance for determining the learning rate strategy which ensures convergence. This handcrafted task of choosing those hyperparameters is challenging. Bottou et al. give theoretical and intuitive explanations for this in their technical report [5]. Recent papers show, with the appropriate learning rate strategy (see You et al. [44] and Goyal et al. [19]), one can scale up to huge mini-batches without having any loss in convergence. Those results are newish and have to be verified empirically on various datasets for large scale training.

Another conspicuousness in Algorithm 1 in line 10 is the aggregation of all the model updates, representing gradients scaled by the learning rate on every node, which has to be performed. This operation is implemented usually in two different ways: Either by the parameter-server, or the AllReduce paradigm. When using the parameter-server paradigm, every worker sends its model updates to a designated server, and gets the result from the same node. This is highly non-scalable as there is one single point of communication. In the AllReduce paradigm on the other hand, on which we focus within this work, all workers interchange their model updates by some reduction structure, and thus every node is preferably equally involved in this process. For the sake of completeness we have to mention, when dealing with asynchronous data-parallel SGD, as for now, one can only rely on the parameter-server paradigm.

This gradients aggregation is the main bottleneck when scaling to a big dimension of the model and increasing the number of nodes. For scaling up to huge model dimensions, we notice that we can lower the amount of data communicated when the gradients, and thus the model updates at every node, are sparse. This is the use case, when the sparse AllReduce operation

comes into play. This sparsity can be given naturally, or enforced by some modification of the SGD algorithms we describe in chapter 2.4.

2.2 Linear Classifiers

We give an overview over the two types of linear classifiers used as a showcase for our sparse AllReduce algorithms. Linear classifiers, or linear models for classification, decide upon the class to assign a sample based on a linear combination of the model with the feature vector of each sample. For both examples, we have a set S of $|S| = M$ samples:

$$S = \{(\mathbf{x_j}, y_j) \mid \mathbf{x_j} \in \mathbb{R}^N, y_j \in \{-1, 1\}, 1 \leq j \leq M\}.$$

y_j denotes the value, in this case the class, every sample belongs to, whereas $\mathbf{x_j}$ represents the feature vector of every sample.

2.2.1 Logistic Regression

Logistic Regression is a maximum likelihood method for estimating the model vector $\mathbf{w}$, if one assumes that all observed training samples were generated by a binomial model that depends on the output of the classifier. It was initially developed by David Cox in the late 60s [11]. Following the separability assumption of the function f, we have to define the loss function per sample and its gradient in order to be able to train such a linear classifier.

$$f_j(\mathbf{w}) = \ln\left(1 + \exp\left(-y_j \mathbf{w}^T \mathbf{x_j}\right)\right)$$

'ln' represents the natural logarithm and 'exp' the natural exponential function e^x. The gradient of the function is given by:

$$\nabla f_j(\mathbf{w}) = \frac{-y_j}{1 + \exp\left(y_j \mathbf{w}^T \mathbf{x_j}\right)} \mathbf{x_j}.$$

Based on those two equations, one can simply run the previously described iterative SGD algorithm in order to reach a model which minimizes the empirical risk. Notice that this function f is convex, and therefore it ensures that SGD converges to the global minimum, if choosing appropriate learning rates [6]. A new sample can then be classified based on the trained model vector $\mathbf{w}$ by applying the logit function (hence the name of the regression method) on the linear combination of the model vector and the feature vector:

$$\sigma\left(\mathbf{w}^T \mathbf{x_j}\right) = \frac{1}{1 + \exp\left(-\mathbf{w}^T \mathbf{x_j}\right)}.$$

This logit, also called sigmoid function, returns the probability of belonging to class $+1$. Symmetrically, the probability of belonging to class -1 is $1 - \sigma\left(\mathbf{w}^T \mathbf{x_j}\right)$.

2.2.2 Support Vector Machines (SVM)

Support Vector Machines (SVM) are a second type of linear model classification. SVMs belong to non-probabilistic classification methods. The goal of SVMs is to determine a class-separating hyperplane parametrized by $\mathbf{w}$ in the space of dimension N. The hyperplane should typically be chosen such that the margin between both training classes is maximized. There are some extensions to this if the classes distributions overlap. The exact derivation of the loss function and additional descriptions, such as relations to Logistic Regression, can be found in the book written by Christopher M. Bishop [4, Chapter 7]. Similarly to Logistic Regression we give the definition of the loss function and its gradient derivation.

$$f_j(\mathbf{w}) = \max\left(0, 1 - y_j \mathbf{w}^T \mathbf{x_j}\right)$$

$$\frac{\partial f_j(\mathbf{w})}{\partial w_i} = \begin{cases} -y_j x_{j_i} & \text{, if } y_j \mathbf{w}^T \mathbf{x_j} < 1 \\ 0 & \text{, otherwise} \end{cases}$$

Notice that SVMs are prone for sparse model updates, due to the form of the loss function. A new sample can simply be classified by looking at the sign of the linear combination between feature and model vector.

2.2.3 MPI-SGD

We want to be able to efficiently train our previously described linear models on various architectures including high performance computing environments such as supercomputers. Message passing interface (MPI) is one of the standards used on most of nowadays supercomputing environments to communicate between nodes. There exist some frameworks, such as Apache Spark[1], which enable large scale data processing and training of linear models. Those frameworks are usually not optimized for high performance computing environments and are simply oversized for this rather simple problem of training linear models. Therefore, we chose to develop a lightweight generic framework, which only makes use of any MPI implementation in order to train linear models on any hardware architecture. MPI-SGD is the tool which was finally built for this purpose. We give a feature overview in the implementation chapter as well as a description on how to incorporate different models and gradient aggregation strategies using our generic framework.

2.3 Deep Neural Networks

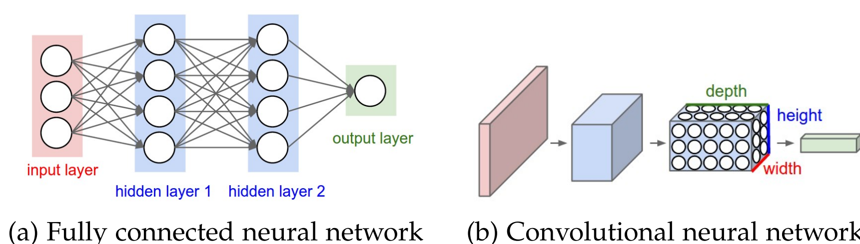
Recent progresses in most of the machine learning and artificial intelligence areas are due to heavy usage of deep neural networks. Those successes

[1] https://spark.apache.org/

affect topics such as image recognition, image understanding, natural language understanding and machine translation to name a few. Giving a profound overview over all types and variants of networks clearly goes beyond the scope of this thesis. Additionally, there is plenty of literature available online or summarized in books such as *Deep Learning* written by Goodfellow et al. [18]. We therefore restrict ourselves on giving an overview over the different types of networks used in the experiment section, as well as a description of the framework utilized for the same purpose.

2.3.1 Fully Connected Neural Networks

Those types of networks are fairly simple but not less powerful. Each unit in the hidden layer is simply connected to every unit in the previous layer. Figure 2.1a illustrates this scheme.



(a) Fully connected neural network (b) Convolutional neural network

Figure 2.1: Various deep neural networks[2]

2.3.2 Convolutional Neural Networks

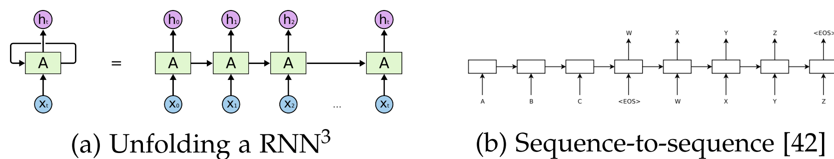
A convolutional neural network, depicted in Figure 2.1b, is mainly used for tasks involving images. Those networks combine fully connected layers with convolutional and max pooling layers. The name indicates already that some layers perform a convolution on sub-patches of the entire input image. Max-pooling layers reduce the size of the convolved input by taking the maximum value of every sub-patch, usually of size 4, over the entire image.

2.3.3 RNN / LSTM

Recurrent neural networks (RNN) are slightly more complicated. Both previously mentioned architectures are feed forward neural networks, that is, every neuron is only connected to neurons on previous layers. This permits efficient gradient computation by using a known technique called *backpropagation*. RNNs on the other hand allow connections between units to form directed cycles. This enables the network to simulate dynamic temporal behavior. Especially for problems involving text and natural language,

[2]Image source: http://cs231n.github.io/convolutional-networks/

this property is powerful as one word alone is in most cases useless if not compared to its context. As some problems arise from long term dependencies when unrolling such RNNs for performing backpropagation, a specialization of RNNs called long short-term memory (LSTM) was invited by Hochreiter et al. in 1997 [24]. Figure 2.2a shows the unfolding of a RNN used for backpropagation. Figure 2.2b illustrates a sequence-to-sequence neural network usually implemented with one or multiple LSTM cells.



(a) Unfolding a RNN[3] (b) Sequence-to-sequence [42]

Figure 2.2: Example usage of recurrent neural networks

2.3.4 The Microsoft Cognitive Toolkit (CNTK)

We need an open source framework for training neural networks using distributed data-parallel SGD. The Microsoft Cognitive Toolkit, former known, and as from now on referred as CNTK in this document, is "a free, easy-to-use, open-source, commercial-grade toolkit that trains deep learning algorithms to learn like the human brain"[4]. A complete documentation is available on the frameworks website and is therefore omitted here. We highlight some basic features, we will later refer to, for the sack of completeness.

Architecture and Parameters

The toolkit offers a generic way for defining any type of network by making use of their customized script language *brainscript*. Scripts for the previously described and other commonly used neural networks are given as examples in the source code repository by Microsoft. Additional to the networks structure, selection of the learning algorithm, aggregation strategy and various hyperparameters are also declared within such brainscript files.

Tensor Handling

The entire model is split into layers according to the network architecture. Every layer holds a tensor representing the current model parameters. This tensor is used for both parts of every iterative learning algorithm. For the forward pass, where the output of the previous units is multiplied by the model tensor before performing activation. And for the backpropagation,

[3] Image source: http://cs231n.github.io/convolutional-networks/
[4] https://www.microsoft.com/en-us/cognitive-toolkit/

where a gradient tensor is calculated using the chainrule and the activation values of the succeeding units. Notice that every gradient tensor is stored non-consecutively with the tensors of the other layers in memory. Any gradient aggregation strategy therefore runs its method for every layer tensor individually, and preferably in a non-blocking way, in order to overlap communication and computation.

Gradient Computation

CNTK enables computation to be executed either on central processing units (CPUs) or graphics processing units (GPUs). The later offers high computation efficiency for easy parallelizable problems such as matrix-vector multiplications. As this is one of the main operations used for calculating the gradients, GPUs have enabled significant speedup when training large scale neural networks. To this goal, CNTK makes use of CUDA by Nvidia as a programming API[5] for executing some code on GPUs. As previously stated, gradient aggregation is performed in a non-blocking fashion due to the tensor handling in CNTK. When computation of the gradients is performed on GPUs, those values have to be copied from device to host memory before running the aggregation across multiple machines. Those memory copy operations are executed using non-blocking CUDA calls in order to ensure further progress in the meantime.

Parallel Training

The deep learning toolkit has already implemented several gradient aggregation strategies. Notably data-parallel SGD, model-averaging SGD [35] and Block-Momentum SGD [9]. Data-parallel SGD comes in three variants: synchronous, asynchronous and 1-Bit SGD [40]. All those distributed variants use MPI for communicating between nodes. We compare our algorithm to the synchronous, full precision data-parallel SGD variant in this work, as the others lack of theoretical convergence guarantees, and synchronous data-parallel SGD is the most often used tool in state-of-the art methods [8, 13]. The code for 1-bit SGD is nevertheless used as a starting point for implementing our SGD variants.

2.4 Top$K$ SGD

We have seen previously, while describing the data-parallel SGD algorithm, one can benefit of a sparse AllReduce algorithm if the model updates are sparse. If this is not given naturally, we can enforce every update to be sparse and have exactly k items. This is doable by selecting the biggest k

[5] https://developer.nvidia.com/cuda-zone

items regarding their absolute values and postpone the contribution of the other elements to the next iterative step. Algorithm 2 shows those modifications highlighted in red to data-parallel SGD.

Algorithm 2 TopK SGD - With Error Correction

1: Initialize $\mathbf{w}$ and $\{\mathbf{r_j} = \mathbf{0} \mid 1 \leq j \leq P\}$
2: **for** $t = 1, ..., T$ **do**
3: *Randomly shuffle training samples*
4: *Partition all samples into batches:* $\{b_i \mid 1 \leq i \leq \lceil \frac{M}{B \times P} \rceil\}$
5: **for** $i = 1, ..., \lceil \frac{M}{B \times P} \rceil$ **do**
6: *Partition samples* b_i *into batches* $\{b_{ij} \mid 1 \leq j \leq P\}$
7: **for** machine $j = 1, ..., P$ **do in parallel**
8: $\widehat{g}_j = \mathbf{r_j} + \sum_{n \in b_{ij}} \nabla f_n(\mathbf{w})$
9: *Select k biggest absolute value of* $\widehat{g}_j$ *being* $|\widehat{g_{j_{[l]}}}|$
10: $\mathbf{r_j} = \mathbf{0}$
11: $\mathbf{r_j}_{[i]} \leftarrow \widehat{g_{j_{[i]}}} \quad \forall i, |\widehat{g_{j_{[i]}}}| < |\widehat{g_{j_{[l]}}}|$
12: $\widehat{g_{j_{[i]}}} \leftarrow 0 \quad \forall i, |\widehat{g_{j_{[i]}}}| < |\widehat{g_{j_{[l]}}}|$
13: $\Delta \mathbf{w}_j = \eta_{t_i} \widehat{g}_j$
14: **end for**
15: $\mathbf{w} \leftarrow \mathbf{w} - \sum_{j=1}^{P} \Delta \mathbf{w}_j$
16: **end for**
17: **end for**

This idea is not entirely new and has been shown to still ensure convergence accuracy and speed for some distinct deep learning problems [1, 14]. Both references only show convergence for small image classification and natural language understanding (NLU) problems empirically, and do not give any theoretical guarantees for maintaining convergence. Having naturally sparse gradients consisting of less than k elements at every node, this method clearly neither affects convergence, nor its speed in comparison to classical SGD. Showing this is trivial, as only zero valued elements are added to the residual and therefore $\widehat{g}_j$ corresponds to the true stochastic gradient at every point in time. We belief, if the gradients are not naturally or at least almost sparse, this method of only incorporating the biggest k absolute values will not hinder convergence, but can compromise its rate. The second claim is shown by giving an example in the experiments section, where convergence for training CNNs with Top$K$ SGD, which are known to have mostly dense updates, is worse than training using classical SGD and all the values. The claim for convergence is not proven yet, but the author beliefs that by adopting recent asynchronous SGD convergence proofs ([12, 15, 29]), one should be able to give convergence guarantees under some assumptions over k.

2.4.1 Approximate Top$K$ SGD

Selecting exactly the biggest k elements in absolute values on a vector of dimension N implies at least a partial sort of the data and $\mathcal{O}(N\log(k))$ runtime complexity. To speed up the algorithm, we can relax the condition of exactness and select approximately the biggest k elements. A natural way to do this would be to follow a randomized algorithm to select the biggest absolute values with a high probability similar to "A Randomized Algorithm for Computing the Median" [32, Chapter 3.4].

There is another path one can follow in order to speed up the algorithm. We select k items in expectation by favoring bigger magnitudes and if possible neglect zero valued entries. We therefore distinct two cases and define the probability p_i of selecting a value v_i at index i. v is an arbitrary vector of dimension N. $\|v\|_1$ and $\|v\|_\infty$ represent the L_1 and L_∞ norm of the vector respectively.

$$p_i = \begin{cases} \frac{k|v_i|}{\|v\|_1} & \text{, if } \|v\|_\infty \leq \frac{\|v\|_1}{k}; \\ \frac{k(|v_i|+\varepsilon)}{(\|v\|_1+N\varepsilon)} & \text{, otherwise,} \end{cases}$$

with:

$$\varepsilon \geq \frac{k\|v\|_\infty - \|v\|_1}{N-k}.$$

Note that ε has to take this form in order to ensure $0 \leq p_i \leq 1, \forall i \in \{1,\ldots,N\}$. Obviously, all p_i have to be positive.

$$\begin{aligned}
\max_i \frac{k(|v_i|+\varepsilon)}{(\|v\|_1+N\varepsilon)} &= \frac{k(\|v\|_\infty+\varepsilon)}{(\|v\|_1+N\varepsilon)} \\
&\leq \frac{k\left(\|v\|_\infty + \frac{k\|v\|_\infty - \|v\|_1}{N-k}\right)}{\|v\|_1 + N\frac{k\|v\|_\infty - \|v\|_1}{N-k}} \\
&= \frac{k\left(\frac{(N-k)\|v\|_\infty + k\|v\|_\infty - \|v\|_1}{N-k}\right)}{\frac{(N-k)\|v\|_1 + Nk\|v\|_\infty - N\|v\|_1}{N-k}} \\
&= \frac{k\left(N\|v\|_\infty - \|v\|_1\right)}{(Nk\|v\|_\infty - k\|v\|_1)} = 1
\end{aligned}$$

Theorem 2.1 *If every index $i \in \{1,\ldots,N\}$ is selected with probability p_i defined above, k items will be selected in expectation.*

Proof We define a Bernoulli random variable X_i with probability p_i for every $i \in \{1,\ldots,N\}$. Additionally, let the random variable $Y = \sum_{i=0}^n X_i$ represent the number of selected indices. We show that $\mathbb{E}[Y] = k$ for the case $\|v\|_\infty > \frac{\|v\|_1}{k}$ (the other case follows a similar, slightly simplified, proof).

$$
\begin{aligned}
\mathbb{E}[Y] &= \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} p_i \\
&= \sum_{i=1}^{n} \frac{k(|v_i| + \varepsilon)}{(\|v\|_1 + N\varepsilon)} = \frac{k}{(\|v\|_1 + N\varepsilon)} \left(\sum_{i=1}^{n} |v_i| + \sum_{i=1}^{n} \varepsilon \right) \\
&= \frac{k}{(\|v\|_1 + N\varepsilon)} (\|v\|_1 + N\varepsilon) = k \qquad \square
\end{aligned}
$$

This modification of the Top$K$ algorithm enables us to speed up the selection of the k biggest items in expectation significantly, as only the calculation of the L_1 and L_∞ norms need to take into account multiple values of the vector. Afterwards, every item can either be selected, or omitted independently of all the other values. This second step is easily parallelizable. Calculating both mentioned norms in a parallel fashion is also feasible in an efficient way, as we describe later in the implementation chapter.

We can even further speed up this algorithm by slightly changing the probability of selecting any index i. If one chooses $p_i = \min\{1, \frac{k|v_i|}{\|v\|_1}\}$, the algorithm would not select exactly k items in expectation anymore, but rather at most k. This modification makes the calculation of the L_∞ norm obsolete and shows to work very well when training text and language related machine learning problems.

Chapter 3

The Sparse AllReduce Problem

In this section, we formally introduce the *sparse AllReduce* problem. The well understood collective operations *AllGather* and *AllReduce* are shown to be specializations. We revisit algorithms used to solve those two problems and come up with variants for the more general formulation. The main difficulty for designing efficient algorithms is the unknown overlap of the non-neutral indices and hence the size of the reduced result, which is highlighted in this chapter. Assuming sparsity fractions and common-case input distributions of the indices allow us to give performance guarantees for the designed algorithms.

3.1 Formulation

This section introduces the analytical model used for runtime analysis, the used notation and describes simplifications of the general formulation to facilitate the analysis.

3.1.1 Analytical Model

We consider a simple $\alpha - \beta$ model which we use to characterize the cost of point-to-point communication. The cost of sending a message of size L is given by $T(L) = \alpha + \beta L$, where both α, the latency of a message transmission, and β, the transfer time per word, are constant. L represents the datum size in words or bytes. This simple model omits the computation of performing a reduction operation, as well as other architecture specific parameters. Even though more complex models such as LogGP [2] have been proposed to address such shortcomings, we stick with this model for simplicity and readability. Giving the exact, or at least proportional computation costs needed in order to reduce two sparse vectors, poses an additional challenge: One needs to know not only the input size (number of non-neutral elements) of both sparse vectors, but also the resulting sparse vector size

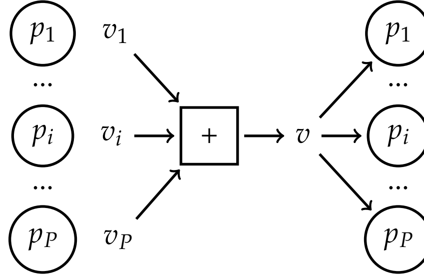


Figure 3.1: The AllReduce Problem with summation

after having performed the reduction operation. Since our focus lays on reducing the overall communication time, we ignore the computation terms for simplicity.

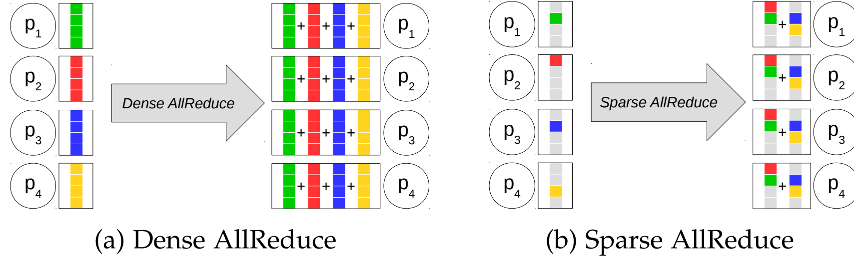
3.1.2 Nodes and Elements

We consider a networked system where a set of P nodes $\mathcal{P} = \{p_1, \ldots, p_P\}$ can communicate via point-to-point links. Initially, each node is assigned a subset of non-zero elements from an universe of size N. Let H_i denote the set of non-neutral elements given at node p_i. We assume that these sets are *sparse* with respect to N, i.e. that $k = \max_i |H_i| \ll N$. We further denote d_i as the density of each set given by $d_i = \frac{|H_i|}{N}$ and define $d = \max_i d_i = \frac{k}{N}$. The goal is to design an efficient communication algorithm to perform a collective operation over all the non-neutral elements at the nodes with respect to the semantics of the given operation. To simplify the exposition, we assume in the following that the collective operation to be performed over all sets is the *sum*, as illustrated in Figure 3.1. In general though, any associative and optionally commutative binary operator could be imaginable. In our setting we can assume both properties, as they are given by *sparse summation*. Notice that the neutral element in this context is the value 0. The 'non-neutral' elements are therefore equivalent to 'non-zero' elements and those two terms can be interchanged. Figure 3.2 illustrates the difference between a commonly known *dense* AllReduce and its sparse variation. In terms of mathematical formula, one defines the result each node has to have after finishing the global reduction as $v \in \mathbb{R}^N$, with

$$v_{[z]} = \sum_{i=1, z \in H_i}^{P} v_{i_{[z]}} \quad \forall z \in \{1, \ldots, N\}.$$

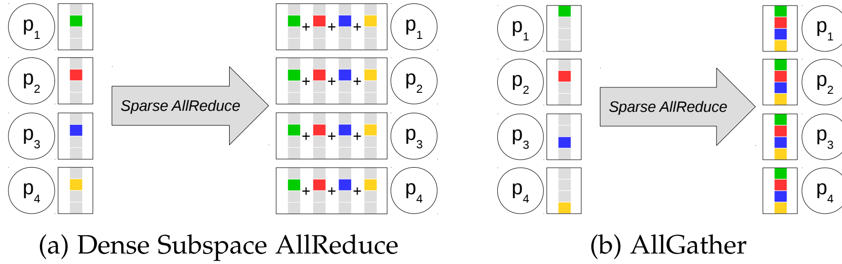
The red part in the formula represents the difference between a sparse vector reduction and, if one omits this term, its dense counterpart.

Given this setting, the goal is to perform an AllReduce operation over the elements present initially at every node, while optimizing for communication



(a) Dense AllReduce (b) Sparse AllReduce

Figure 3.2: Difference between dense and sparse AllReduce. The gray items are neutral elements in the vectors.



(a) Dense Subspace AllReduce (b) AllGather

Figure 3.3: Specializations of sparse AllReduce. The gray items are neutral elements in the vectors.

cost. That is, at the end of the execution, each node should have the correct result locally, i.e. the element-wise sum over the N dimensions, while minimizing the total communication costs, measured in the $\alpha - \beta$ model above. For analyzing the runtime of each algorithm, we assume that every node starts the operation exactly at the same point in time, and the total duration is defined as the interval until the last node receives its last message and, if needed, has computed the reduction operation locally.

3.1.3 Generalization

Notice that the given definition of sparse AllReduce is a generalization of both well known collective operations AllReduce and AllGather simultaneously. Only the sets of non-zero elements initially assigned to each node distinguishes those problems. The AllGather collective is given if non of the sets overlap: $\forall i < j : H_i \cap H_j = \emptyset$. The resulting set of non-zero indices therefore has size $\sum_{i=1}^{P} |H_i|$. On the other hand, the AllReduce problem is given if $H_i = H_j$ for all nodes i and j, that is, the sets fully overlap and therefore the reduced result has $|H_1|$ elements. If $|H_i| = N$ for all i, we refer to this problem as dense AllReduce. If $|H_i| = k < N$, we say that the problem is equivalent to a dense AllReduce on a subspace of dimension k rather than N. Those two specializations are illustrated in Figure 3.3.

3.1.4 Simplification and Assumptions

Let us make some assumptions to facilitate the formulas derived in the subsequent analysis of all the algorithms:

1. Every node contributes exactly k elements: $\forall i : |H_i| = k$
2. P is a valid power of 2 and $P > 4$
3. N is divisible by P.

Discussion

Even though these assumptions simplify the formulas, they do not oversimplify the problem. Ignoring the first assumption and having $k = \max_i |H_i|$, one gets an upper bound on each algorithm. This upper bound only makes sense if we assume approximately an equal number of non-zero elements at every node. Otherwise, one could imagine to design more specific algorithms. If the second assumption does not hold, one can add two additional steps in front and at the end of every algorithm to reduce the number of participating nodes to the nearest lower power of two. Although this might not be optimal (a dissemination approach [23] might be favorable), the cost increases by some constant value and thus, we still get an idea about which algorithm to prefer. If assumption (3) does not hold, each node gets responsible of $\lfloor \frac{N}{P} \rfloor$ items apart of the last one, which is responsible of $N - (P-1)\lfloor \frac{N}{P} \rfloor$ items.

3.2 Resulting Size

We realize, the difficulty of designing any efficient algorithms comes from the fact that we neither know in advance the exact number of items every node contributes, nor the size any intermediate, or the final result will have. This data has to be communicated across the network. Those result sizes are not only dependent on the amount of data contributed by each node, but also alters with different positions of the non-neutral indices. We therefore define the total number of non-zero elements after having performed the overall reduction as

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

We make an additional assumption which holds typically in real world examples by omitting cancellation of indices. Even though theoretically incorrect, we assume no two values at the same position sum up neither in intermediate, nor in the final result, to the neutral element 0. The probability of this scenario to happen goes towards zero when working with floating point values on real world examples. Obviously, in the context of summation,

$$k \leq \mathcal{K} \leq \min\{N, P \times k\}$$

holds, with both extremes representing the problem specializations, AllReduce on a subspace of dimension k and AllGather respectively. If one assumes an underlying probability distribution of the non-zero elements, one can define the expected total number of non-zero elements $\mathbb{E}[\mathcal{K}]$. We therefore make use of N Bernoulli random variables $X_j = 1$, if index $j \in \cup_{i=1}^{P} H_i$, and $X_j = 0$ otherwise, for $1 \leq j \leq N$. The random variable $Y = \sum_{j=1}^{N} X_j$ then represents the total number of non-zero entries after having performed the reduction. So by using the linearity property of the expectation, we get:

$$\mathbb{E}[\mathcal{K}] = \mathbb{E}[Y] = \sum_{j=1}^{N} P\left(j \in \cup_{i=1}^{P} H_i\right).$$

The probability of any index j being an element of a distinct set H_i is given by the underlying distribution. It is true for any distribution that:

$$P\left(j \in \cup_{i=1}^{P} H_i\right) = \sum_{i=1}^{P} P\left(j \in H_i\right) - \sum_{i<k} P\left(j \in (H_i \cap H_k)\right) + \\ \sum_{i<k<l} P\left(j \in (H_i \cap H_k \cap H_l)\right) \ldots + (-1)^{P-1} P\left(j \in \cap_{i=1}^{P} H_i\right).$$

We further know from Union Bound that $P\left(j \in \cup_{i=1}^{P} H_i\right) \leq \sum_{i=1}^{P} P\left(j \in H_i\right)$, which gives us a valuable upper bound on the expected number of non-zero elements $\mathbb{E}[Y] \leq \sum_{j=1}^{N} \sum_{i=1}^{P} P\left(j \in H_i\right)$. This bound is tight if $\forall i < j : H_i \cap H_j = \emptyset$, which is the special case where the problem reduces to an AllGather.

3.2.1 Uniform Distribution

Having derived those formulas, we give more concrete values by assuming a uniform distribution. This use-case gives a worst-case scenario in terms of probabilistic growth of the intermediate results and it is reasonable to make this assumption, if every index is hit with probability higher than 0. For this, let H_i consist of k independent samples drawn from a uniform distribution

$$j \sim \mathcal{U}\left(1, N\right) \quad \forall j \in H_i,$$

therefore $P\left(j \in H_i\right) = \frac{k}{N}$. This is independent of the two indices i and j in the above general formula, so $\mathbb{E}[\mathcal{K}] \leq N \times P \times \frac{k}{N} = P \times k$, which fits the non-probabilistic upper bound given earlier. For the uniform distribution one can give the exact expected number of elements by deriving a closed-form function utilizing the previous equations

$$\mathbb{E}[\mathcal{K}] = f(k, N, P) = N \times \left(\sum_{i=1}^{P} (-1)^{i-1} \binom{P}{i} \left(\frac{k}{N}\right)^{i}\right).$$

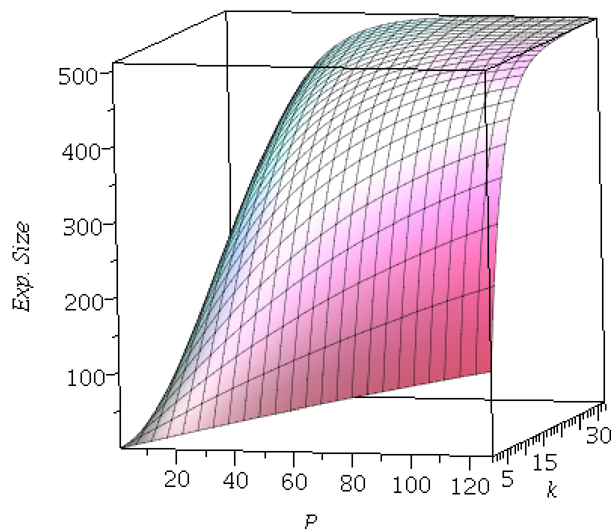
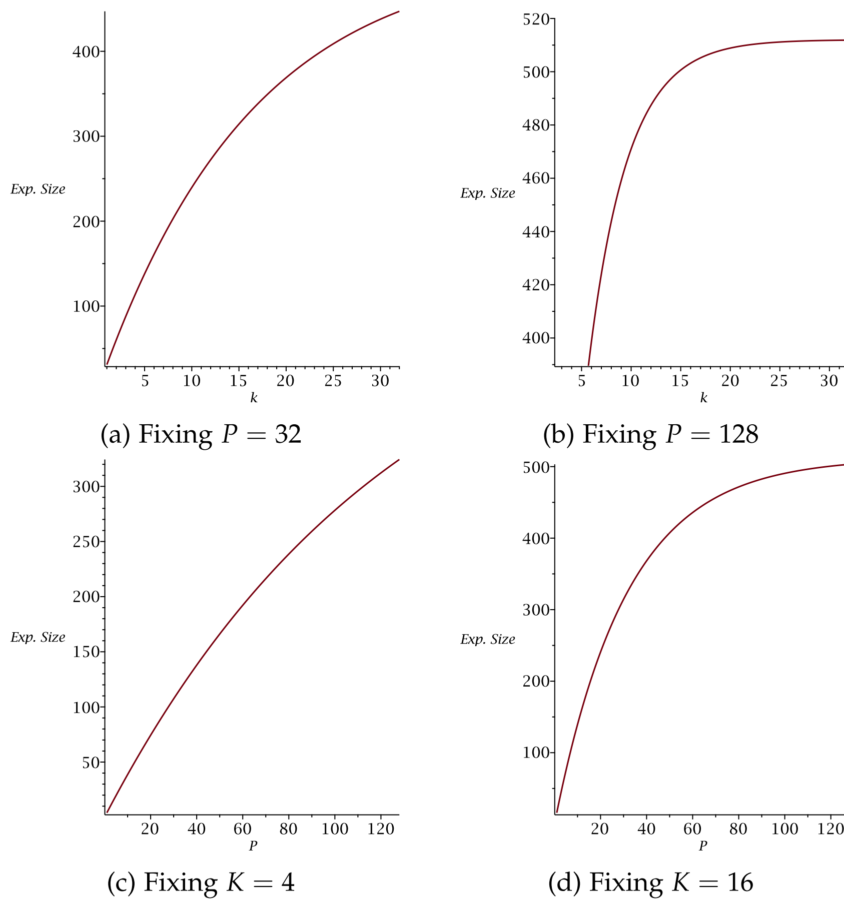


Figure 3.4: Expected size of reduced result assuming a uniform distribution and $N = 512$

We make use of this function f in the next sections to give expected running times assuming this specific distribution. Figure 3.4 illustrates the multiplicative growth dependent on both inputs, the number of nodes P and the number of non-zero entries k at each node. To get a better feeling on how the number of nodes and their input densities affect the expected size of the result, we fix values for P and k additional to N and visualize the resulting graphs in Figure 3.5. One thing to notice is that fixing either parameter does not change the resulting graph much. This implies that the two parameters act in a multiplicative way when calculating a growth factor.

3.3 Algorithms

We focus on designing algorithms for performing the sparse AllReduce collective addition operation, for which we can give a reasonable analysis based on the proposed α-β communication cost model. Notice that none of those algorithms needs to have prior knowledge about the amount of data contributed by each node, or the distribution of the non-zero indices. Nevertheless, based on those two variables and the number of nodes in the network system, one can calculate the amount of non-zero indices after having performed the collective reduction and thus decide upon which algorithm to



(a) Fixing $P = 32$

(b) Fixing $P = 128$

(c) Fixing $K = 4$

(d) Fixing $K = 16$

Figure 3.5: Expected size of reduced result assuming a uniform distribution and $N = 512$

choose in order to achieve least overall time.

3.3.1 Known Approaches

Our problem definition is a generalization of both known collection operations AllGather and AllReduce. We therefore give a quick overview of commonly used algorithms for solving those two specialized problem settings.

AllGather

The AllGather collective is a gather operation in which the data contributed by each node is gathered to all machines instead of only one root node. Assuming each participator contributes exactly k elements, lower bounds in terms of our performance cost model on the AllGather collective are

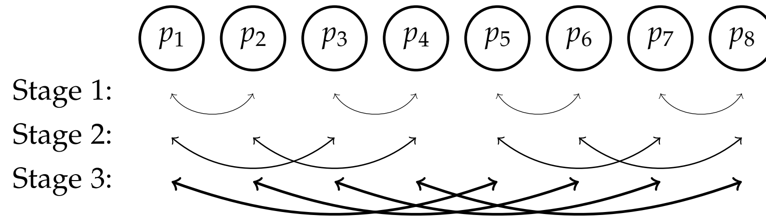


Figure 3.6: AllGather: Recursive Doubling Algorithm - Increasing amount of data in each stage

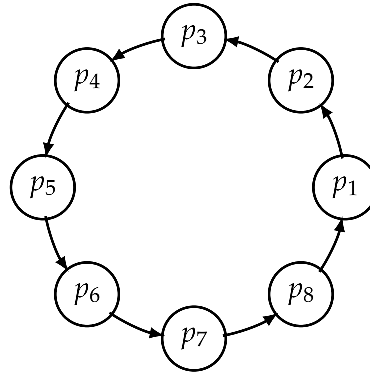


Figure 3.7: AllGather: Ring Algorithm

known [7] to be

$$\log_2(P)\alpha + (P-1)k\beta.$$

Most MPI libraries such as MPICH[1] distinguish between small message sizes (total amount of data < 512 KB) and long messages (> 512 KB) for choosing the appropriate algorithm [43]. For short messages, a recursive doubling algorithm is used. Figure 3.6 illustrates how this algorithm works: In the first round, nodes that are a distance 1 apart exchange their data. In the second round, nodes that are a distance 2 apart exchange their data as well as the data they received in previous rounds. Following this pattern, in the t-th round, nodes that are a distance 2^{t-1} apart exchange all the previously gathered $2^{t-1}k$ data items. Therefore, this algorithm has a total time cost of

$$T_{ag_rec_dbl} = \log_2(P)\alpha + (P-1)k\beta$$

which satisfies the lower bounds on the latency and the bandwidth term. At the same time, for long messages, a ring algorithm seems to perform much better than a recursive doubling algorithm. The ring performs $(P-1)$ steps. In the first step, node i sends its data to the node with rank $i+1$ and

[1] http://www.mpich.org/

receives the data from the node $i - 1$ (with wrap around). In the next steps, the data received in the previous step is sent to the according neighbor. This communication pattern is illustrated in Figure 3.7. The total cost of this algorithm is

$$T_{ag_ring} = (P-1)\alpha + (P-1)k\beta.$$

Even though this implies a higher bandwidth term compared to the recursive doubling algorithm, it is believed that the algorithms performance increase for large data comes from different communication patterns of the two algorithms: In the ring algorithm, nodes only communicate with nearest neighbors, whereas in the recursive doubling case, nodes which are far apart have to exchange a lot of data in the later steps.

AllReduce

The AllReduce collective performs a global reduction operation and distributes the result to all the nodes. It can be seen as a reduce operation to a dedicated root followed by a Broadcast to all the other nodes. Assume each node has a vector of size k. The lower bound in terms of the $\alpha - \beta$ model is [34]

$$\log_2(P)\alpha + 2\frac{P-1}{P}k\beta.$$

For quite small messages (total amount of data < 2 KB), custom operations, or if k is smaller than the number of nodes P, the implementation falls back on a recursive doubling algorithm similar to the one described previously for the AllGather collective. The amount of data communicated at each step remains constant (k items), as the reduction operation is performed after each step. The total time for this algorithm is therefore

$$T_{ar_rec_dbl} = \log_2(P)\alpha + \log_2(P)k\beta.$$

The lower bound for the latency term is reached, but not for the bandwidth part. For a larger amount of data, one wishes to achieve a lower bound on the bandwidth term. Thus Rabenseifner's algorithm is used for that case [36], which solves the AllReduce problem in two steps. A ReduceScatter implemented by a recursive halving algorithm is first applied. Afterwards, the reduced data is gathered from all, to all other nodes by calling a recursive doubling algorithm as described above. Figure 3.8 illustrates how the ReduceScatter operation works. This two step algorithm has a total runtime of

$$T_{ar_rab} = 2\log_2(P)\alpha + 2\frac{(P-1)}{P}k\beta,$$

which reaches the lower bound on the bandwidth term as desired.

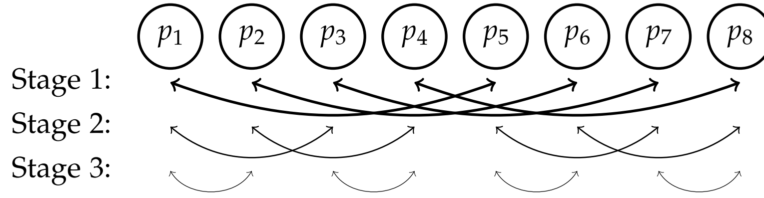


Figure 3.8: AllReduce: Recursive halving for ReduceScattter - Decreasing amount of data in each stage

3.3.2 Dense vs. Sparse Representation

Although we focus on a *sparse* problem formulation, namely $k \ll N$, the size of the resulting vector can be such that the algorithm does not benefit of a sparse representation anymore, or even $\mathcal{K}$ might become equal to N. Let *isize* be the number of bytes needed to represent a value in the space of dimension N. We further define c to be the amount of bytes needed to store a sparse index. Over the entire dimension N, $c \geq \lceil \frac{\log_2(N)}{8} \rceil$ bytes are at least needed to store every index. As the amount of data consists of k sparse elements, this gives a total of $c \times k + k \times isize = (c + isize)k$ bytes to be transmitted compared to $N \times isize$ bytes, if one opts for a dense vector representation. From this fact, going for a sparse representation only helps reducing the communication time if $k \leq \delta = \frac{N \times isizse}{(c+isizse)}$ regarding the size of data transmitted. This formula does not take into account, in practice, applying the reduction operation on dense consecutive items can be significantly faster than working with sparse data. This fact forces the threshold δ to be even smaller, in order to speed up the algorithm. We give some empirical measurements in the experiment section supporting this claim. We assume k to be smaller than this threshold. But this does not necessarily imply $\mathcal{K}$ to be below that value as well. On the contrary, if dealing with a big number of nodes P, $\mathcal{K}$ will most probably lie above this threshold. Reminding ourselves that we usually do not know the exact value of $\mathcal{K}$, we use its upper bound to differentiate between two types of sparse AllReduce subgenres: *static sparse AllReduce* (SSAR) and *dynamic sparse AllReduce* (DSAR). The *static* case assumes that $\mathcal{K}$ remains below δ. For the *dynamic* instance, we know $\mathcal{K} \geq \delta$. We omit both values c and *isize* defined above, in the analysis of the sparse AllReduce algorithms given next. *isize* can simply flow into the constant β. The unit of sent items is no longer words or bytes, rather *isize* bytes. The amount needed for storing a sparse index can be incorporated into the parameter k. Having the formula for running time dependent on k, one could simply plug in the values $k \times (1 + \frac{c}{isize})$ instead of k, in order to get accurate results.

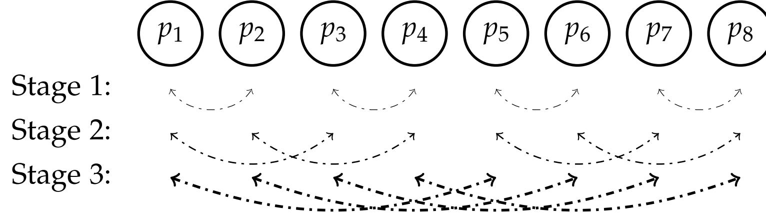


Figure 3.9: Static Sparse AllReduce: Recursive doubling - Increasing amount of sparse data in every stage

3.3.3 Static Sparse AllReduce

The size of the final result remains below the threshold value δ. Clearly it is true that $N \nleq \delta$, this implies that $\mathcal{K} = k \times P < \delta$ must hold. Therefore, even the final result is kept in a sparse format, and we give a lower bound on the bandwidth term any algorithm needs in order to successfully terminate the SSAR problem:

Lemma 3.1 *Any algorithm solving the SSAR problem needs at least a duration of* $\log_2(P)\alpha + (P-1)k\beta$ *if* $\mathcal{K} = k$, *and* $\log_2(P)\alpha + 2\frac{P-1}{P}k\beta$ *if* $\mathcal{K} = k \times P$.

Proof The proof follows directly from the fact that the lower bound is known for both specializations of the sparse AllReduce problem: AllGather if $\mathcal{K} = k \times P$ and subspace dense AllReduce if $\mathcal{K} = k$. □

Inspired by both aforementioned algorithms for solving the AllReduce and AllGather problems, we differ between two algorithms to solve the SSAR problem. One for small messages, and the other one for a larger amount of data. Again, the exact threshold is defined empirically due to the simplified communication cost model and all the problem-specific unknown variables.

SSAR_Rec_Dbl

For small messages, we design a recursive doubling algorithm as visualized in Figure 3.9. After every stage, a sparse vector summation on the received and local data has to be performed. The latency is

$$L(P) = \log_2(P)\alpha$$

for this algorithm, as there are $\log_2(P)$ stages. This part is data-independent. The runtime for this algorithm lies in the range

$$L(P) + \log_2(P)k\beta \leq T_{ssar_rec_dbl} \leq L(P) + (P-1)k\beta.$$

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

Figure 3.10: Static Sparse AllReduce: Uniform splitting of a sparse vector. The light gray items are neutral elements with value 0 and therefore are not sent

constant size. The upper bound is given when the indices do not overlap at all. Therefore, at stage t, the amount of items transmitted is $2^{t-1}k$. Taking the sum, we get

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

Having an underlying uniform distribution of the data, one obtains an expected running time of

$$\mathbb{E}[T_{ssar_rec_dbl}] = L(P) + \beta \sum_{i=1}^{\log_2(P)} f(k, N, 2^{i-1}).$$

Even though the latency term reaches the lower bound, the bandwidth term might get significantly larger than other algorithms can achieve.

SSAR_Split_AlGa

If the overall data is large, we separate the algorithm similarly to Rabenseifner's version into two steps: a *Split* and *sparse AllGather* phase. For the Split part, we uniformly divide the dimension of the space N by P and define each node to be responsible of indices contained in the corresponding

subspace. Figure 3.10 illustrates this act on a sparse vector. We split each sparse vector at its node and directly send the part to the corresponding recipient again in a sparse format. Figure 3.11 depicts this communication pattern. Each node then sums up the data received and builds the result for the distinct part of the entire vector. The data has to be gathered to all other nodes finally. This *sparse AllGather* is executed by a recursive doubling algorithm similar to the previously described variant, with the difference that the summation of two intermediate results is equivalent to a concatenation of both sparse vectors, contrary to the previously described algorithm, where indices might overlap. Giving bounds and expected running times of this algorithm is slightly more involved. The Split part takes

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

Notice that both extremes imply that each node has k items for the sparse AllGather, and thus $\mathcal{K} = k \times P$ is reached. For this second step in the algorithm to be minimal though, every node must have an intermediate result of size $\frac{k}{P}$, as we want the final result to have a size $\mathcal{K} = k$ and the communication to be equally distributed. This is simply provable by contraction and making use of the pigeonhole principle. For every node to have an intermediate result of the desired size, we know that each node has to send at least $\frac{P-1}{P}k$ items to other nodes. Otherwise, if every node has exactly k items, we reach the upper bound for the result size of $\mathcal{K} = k \times P$. So we get

$$\log_2(P)\alpha + \frac{P-1}{P}k\beta \leq T_{sparse_ag} \leq \log_2(P)\alpha + (P-1)k\beta.$$

The latency

$$L(P) = ((P-1) + \log_2(P))\alpha$$

for the entire algorithm is again data-independent. Combining those terms yields to

$$L(P) + 2\frac{P-1}{P}k\beta \leq T_{ssar_split_ag} \leq L(P) + Pk\beta.$$

Not having any knowledge about the distribution hinders us to give better bounds. Assuming this to be uniform, one can state the expected runtime

$$\mathbb{E}[T_{ssar_split_ag}] = L(P) + \left(\frac{P-1}{P}k + (P-1)f\left(\frac{k}{P}, \frac{N}{P}, P\right)\right)\beta.$$

3.3.4 Dynamic Sparse AllReduce

As emphasized previously, the size of the result $\mathcal{K}$ might become bigger than the threshold δ, representing the point after which one should rather switch to a dense representation and replenish with neutral elements (zero in the case of summation).

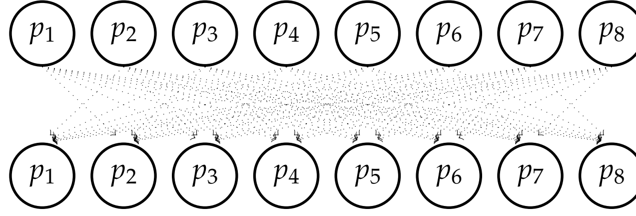


Figure 3.11: Direct sparse send AllToAll

Theorem 3.2 *Any algorithm solving the DSAR problem needs at least a duration of* $\log_2(P)\alpha + N\beta$.

Proof As every node needs to communicate to every other node directly or indirectly, there is at least one node communicating to $\log_2(P)$ other nodes. For the bandwidth term, every node needs to send its k items. As the size of the resulting vector increases such that it eventually has no sparse representation anymore, each node has to receive or send at least $N - k$ items. Taking the sum results in $N\beta$ on the bandwidth term. □

From this observation we further derive the following lemma.

Lemma 3.3 *The bandwidth required by any algorithm solving the DSAR problem is at least* $\frac{1}{2}$ *the minimum bandwidth required by a dense AllReduce for the same problem.*

Proof Notice that the dense AllReduce has a lower bound of $2\frac{P-1}{P}N\beta$ on the bandwidth. From Theorem 3.2 we know that every DSAR algorithm has a minimum bandwidth term of $N\beta$, which is obviously bigger than $\frac{P-1}{P}N\beta$ for any P. □

This lemma tells us, if the resulting size $\mathcal{K}$ is bigger than the threshold δ, one can only hope to get a speedup of factor 2 compared to an optimal dense AllReduce algorithm, when focusing on the bandwidth term.

DSAR_Split_AG

Based on this fact, we adopt the algorithm SSAR_Split_AlGa to force every split to become dense. So, even though the data is received from the other nodes in a sparse format, both, the summation and more important the second step of the algorithm, the *AllGather*, are fulfilled relying on a dense format. There are already highly optimized algorithms to perform this second step with dense data and reaching the lower bounds on both bandwidth and latency terms. We therefore do not further investigate this topic here. The Split part takes again a runtime of

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

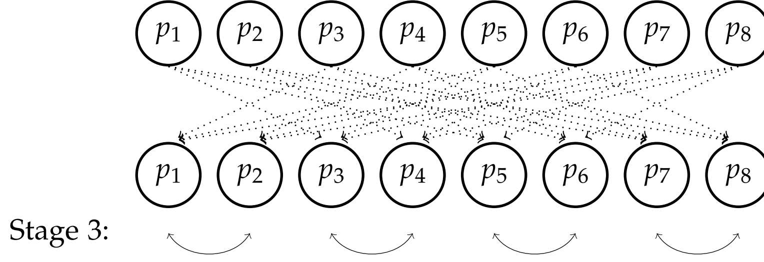


Figure 3.12: Early Switching DSAR - Groupsize 2

As we force every splited results to become dense, there is no data dependency anymore after having performed this first part of the algorithm. Based on the known times needed by a dense AllGather, we derive the running time for our algorithm given both extremes. The latency is given by

$$L(P) = ((P-1) + \log_2(P))\alpha.$$

Combined, we get

$$L(P) + \frac{P-1}{P} N\beta \leq T_{dsar_split_ag} \leq L(P) + \left(k + \frac{P-1}{P} N\right)\beta.$$

Assuming a uniform distribution of the data, one can give an expected runtime, which only affects the Split part

$$\mathbb{E}[T_{dsar_split_ag}] = L(P) + \left(\frac{P-1}{P}(k+N)\right)\beta.$$

Early Switching DSAR

We extend the idea of Dynamic Sparse AllReduce (DSAR) as visualized in Figure 3.12 and 3.13. Intuitively, if one somehow has come to know that the size of every intermediate result on a subset of the nodes $\mathcal{P}$ is larger than the dense threshold δ, one could force the vectors to become dense earlier in the reduction structure. From that point, the last steps of a classical dense ReduceScatter implemented by a recursive halving algorithm can be carried on as depicted in Figure 3.8. We verify empirically in the experiment section, that this idea further lowers the overall runtime at a high number of nodes.

3.3.5 Discussion

One might ask the question: Why should we use a direct AllToAll instead of a recursive halving like algorithm for the Split part in the aforementioned algorithms? Even though we clearly are optimal on the bandwidth term, the one we care most about comes at a price of high latency. Nevertheless, those

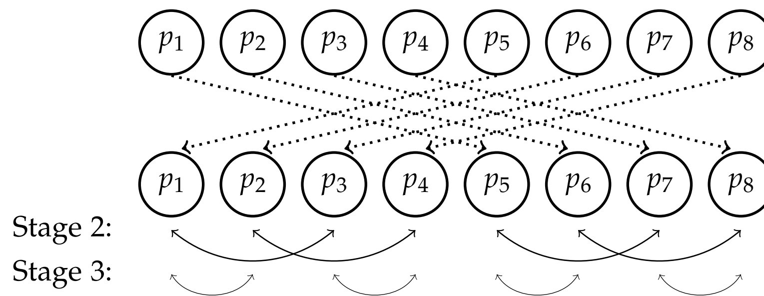


Figure 3.13: Early Switching DSAR - Groupsize 4

costs are in practice less relevant if we work with non-blocking function calls, and thus enable overlapping of computation and communication. We give more implementation related details in the according chapter.

Runtime Summary

Table 3.1 summarizes the results stated so far and gives a comparison between not only all the algorithms, but also the lower bounds and the defined baseline dense AllReduce. We have a verification and empirical determination of the switching points for those algorithms in the experiment chapter based on synthetic micro benchmarks.

	Latency	Bandwidth ($\mathcal{K} = k$)	Bandwidth ($\mathcal{K} = k \times P$)	Exp. Bandwidth ($j \sim \mathcal{U}(1, N)$)
Lower Bound	$\log_2(P)\alpha$	$2\frac{P-1}{P}k\beta$	$(P-1)k\beta$	-
$T_{ssar_rec_dbl}$	$\log_2(P)\alpha$	$\log_2(P)k\beta$	$(P-1)k\beta$	$\left(\sum_{i=1}^{\log_2(P)} f(k, N, 2^{i-1})\right)\beta$
$T_{ssar_split_ag}$	$((P-1) + \log_2(P))\alpha$	$2\frac{P-1}{P}k\beta$	$Pk\beta$	$\left(\frac{P-1}{P}k + (P-1)f\left(\frac{k}{P}, \frac{N}{P}, P\right)\right)\beta$
$T_{dsar_split_ag}$	$((P-1) + \log_2(P))\alpha$	-	$\left(k + \frac{P-1}{P}N\right)\beta$ *	$\left(\frac{P-1}{P}(k+N)\right)\beta$
Dense AllReduce	$2\log_2(P)\alpha$	$2\frac{P-1}{P}N\beta$	$2\frac{P-1}{P}N\beta$	$2\frac{P-1}{P}N\beta$

Table 3.1: Summary of the analytical results. "-" represents unknown values. At (*) we denote an upper bound on the runtime assuming $\mathcal{K} \geq \delta$

Chapter 4

Implementation

In this section we describe the most important parts implemented in the scope of this work. We justify the choice of the previously mentioned algorithms by showing how their code can be written efficiently. We focus mainly on the following three parts contributed in of this thesis:

1. Implementation of the aforementioned sparse AllReduce algorithms in a library
2. Development of a lightweight and generic machine learning framework named MPI-SGD for training large scale linear classifiers (and others) on supercomputers with fusion of the above library
3. Incorporating the suggested Top$K$ SGD variant and the sparse AllReduce library efficiently in the deep learning framework CNTK.

Notice that point two and three are completely independent of each other.

4.1 Sparse AllReduce Library

The aforementioned algorithms are implemented fully in `C++11`. The header-only files form a library with interfaces similar to those defined in the MPI-3 standard. In order to achieve the best possible speedup, a few points have to be considered whilst implementing the algorithms. The library is used in MPI-SGD and therefore fully accessible through the repository of that framework.

4.1.1 Sparse Vector Representation

Even though there are plenty of different representations for sparse matrices, see Bell et al. [33] for a complete overview, currently there does not exist so much related work for sparse vector representations. Notice that we do not want to specifically encode (lossy or lossless) our sparse vectors in this

work. Albeit, this could further reduce the amount of data communicated, it comes at a cost of additional computation for decoding. As a simple representation, we therefore opt for storing every index of the non-neutral elements in addition to those actual values. There are two simple ways for storing all those values in memory:

- Store a vector of all indices followed by a vector of all values
- Store a single vector consisting of index-value pairs.

Regardless of the method, we assume that all the indices are stored in an ordered way. Whilst both variants need the same amount of space, they have their dis- and advantages. We highlight those on the summation of two sparse vectors, as this is the main use case in our applications. When determining the overlap of indices based on two sparse vectors, one would favor the first representation, as the values themselves are not moved to the cache by the prefetcher. This specific operation without directly accessing the values is nevertheless not used in our implementation. There is one operation which clearly makes the second variant preferable: In the second part of the SSAR_Split_AlGa algorithm, we perform an AllGather operation of sparse sub-vectors. Those sub-vectors are again potentially sparse and follow this representation. As the vectors represent distinct, but consecutive subspaces of the entire space, building the sum of two of them simply reduces the problem to a concatenation when using the second variant for the representation. This can be implemented efficiently by calling a memory copy instruction. If one opts to use the first variant, the first vector consisting of all the indices would have to be enlarged, and thus additional memory movement would be needed. This clear advantage leads us to the decision of choosing the single index-value pair vector as a representation for every sparse vector. We omit the analysis of other variants for representations such as linked-lists, which offer faster summation, but have the disadvantage of being stored in a non-consecutive manner in memory. Therefore, one would need to copy all the values before sending them to other nodes, as MPI requires consecutive values in memory, or at least known constant gaps between them.

Obviously, the datatype of the values yields the amount of bits we need for every non-neutral value. Usually, we either work with single (32 bits) or double precision floating point values (64 bits). We stated previously, depending on the dimension N of the space, we need at least $\lceil \log_2(N) \rceil$ bits for storing an index value. Additionally, when using one of the *split* methods, we generate sparse vectors on subspaces of dimension $\frac{N}{P}$ each. This implies that we need at least $\lceil \log_2(\frac{N}{P}) \rceil$ additional bits per non-neutral element for representing their index in the subspace. Although theoretically correct, we want to make use of state of the art `C++11` positive integer datatypes such as `unsigned char` (8 bits), `unsigned short` (16 bits), `unsigned integer` (32 bits) or `unsigned long` (64 bits). The biggest index each one of these types

can store is 256, 65k, 4b (billions) and 1.8×10^{19} respectively. While the last one is clearly oversized for all our problems, having $N > 65$k is almost always the case in all our large scale examples. Hence, we fix the datatype for storing an index to an `unsigned integer` using 32 bits. As we fully implement our code in native `C++11`, we make use of templates in order to define both, the types of each value and index. The example code is given in Listing 4.1.

4.1.2 Auto-switch to Dense Format

We saw that storing sparse vectors in the previously mentioned way introduced an overhead per non-neutral element. Additionally, in the context of sparse AllReduce, we assume the input to be sparse initially. Due to lack of exact knowledge on the amount contributed by each node and their respective distribution of the indices, we cannot tell whether the final, or even intermediate results, might need more space than a fully dense representation. In order to prevent this to happen at every point in our algorithm, we add an extra value in front of every sparse vector. This value called `nofitems` indicates the number of items stored in our stream representing either sparse or dense vectors. We define such a stream to represent a dynamic array of either `s_item`'s if `nofitems`$\neq N$, or a dynamic array of `ValTypes`'s otherwise. Listing 4.1 illustrates this extra value as well as its usage in the given function `countBytes`.

Listing 4.1: Sparse vector stream with function for counting the number of bytes needed

```
1  template<class IdxType, class ValType>
2  struct s_item {
3    IdxType idx;
4    ValType val;
5  };
6
7  struct stream {
8    unsigned nofitems;
9    char items[];        // Will be casted in either (struct s_item*) or (ValType*)
10 };
11
12 template<class IdxType, class ValType>
13 size_t countBytes(const struct stream *s, unsigned dim) {
14   if(s->nofitems == dim) {
15     return sizeof(unsigned) + s->nofitems * sizeof(ValType);
16   }
17   return sizeof(unsigned) + (s->nofitems * (sizeof(IdxType) + sizeof(ValType)));
18 }
```

When allocating such a stream in memory for a vector in dimension N (for the *split* parts, the dimension is $\frac{N}{P}$), we always request `sizeof(unsigned)` $+$ $(N \times$ `sizeof(ValType)`$)$ bytes. It is therefore never possible to store more than $\lfloor \frac{N \times \text{sizeof(ValType)}}{\text{sizeof(ValType)}+\text{sizeof(IdxType)}} \rfloor$ sparse items. We will refer to this number as $\varphi(N)$ in the following sections. This threshold is used to automatically switch the representation from sparse to dense. The library therefore

checks prior to every size-changing operation, whether the format should be adopted.

4.1.3 Sparse Vector Summation

Whilst justifying the way we represent our sparse vectors, we mentioned the operation which is performed within this library: The addition of two such vectors. Notice that the neutral element for this operation is 0, i.e. non-neutral and non-zero elements are equivalently readable. If we look at the summation of two dense vectors of identical size, there are numerous ways to speed up the calculation such as SIMD (single instruction, multiple data) instructions and cache prefetching by carefully iterating over the values. This enables the summation of dense vectors to be extremely efficient and almost impossible to beat in terms of speed if working with the same amount of sparse values. We nevertheless investigate several methods for summing up streams as most efficiently as possible. For this purpose, we differentiate between two scenarios if summing up two vectors u_1 and u_2:

1. u_1 and u_2 do potentially overlap and are elements of $\mathbb{R}^N$
2. u_1 and u_2 are elements of distinct $\mathbb{R}^{\frac{N}{P}}$, hence do not overlap at all.

The dimension of the space the vectors are contained in, is relevant for the complexity specifications. Notice that in the first case, the result $u_1 + u_2$ will again be in $\mathbb{R}^N$, whilst in the second case it will be in $\mathbb{R}^{\frac{2N}{P}}$. For both cases we further have to distinct between four different scenarios:

(a) Both u_1 and u_2 are sparse

(b) Both u_1 and u_2 are dense

(c) u_1 is dense and u_2 sparse

(d) u_1 is sparse and u_2 dense.

Next, there is a description for each one of the eight possibilities. We denote H_1 and H_2 to be the sets containing the indices of non-neutral elements for both vectors u_1 and u_2 in the sparse case. For all the variants of potential overlapping indices (1), we give an example function which could perform the addition. Listing 4.2 represents the function declaration used for this purpose.

Listing 4.2: Potentially overlapping stream summation - Function declaration

```
1 template<class IdxType, class ValType>
2 struct stream * sum_into_stream(struct stream *first_s, struct stream *second_s,
      struct stream *tmpbuf, unsigned dim) {
3   unsigned p1 = 0;
4   unsigned p2 = 0;
5
```

```
6     unsigned len_first = first_s->nofitems;
7     unsigned len_second = second_s->nofitems;
8
9     // Distinction based on len_first and len_second
10  }
```

1.a First, we have to check whether the result might become dense or not. Theoretically, one would therefore need to calculate the size of the union of non-zero indices $|H_1 \cup H_2|$. This is already computational intensive and thus we rely on an upper bound of this result namely $|H_1| + |H_2|$. If this value is bigger than $\varphi(N)$, we force the result to be dense. We also need to initiate a new stream and set all the values based on every index $z \in \{1, \ldots, N\}$. If $z \notin H_1 \cup H_2$, we set it to the neutral element 0. If $z \in H_1 \cap H_2$, it gets the value of $u_{1_z} + u_{2_z}$. Finally if z is only present in H_1 or (exclusively) in H_2, it gets the value at the index of the corresponding vector. If the upper bound on the size remains below $\varphi(N)$, we proceed identically, but with omitting all the indices $z \notin H_1 \cup H_2$. The complexity for both computation and space in the dense case is $\mathcal{O}(N)$ and in the sparse case at most $\mathcal{O}(|H_1| + |H_2|)$. The code for this is given in Listing 4.3.

Listing 4.3: Potentially overlapping stream summation - Both input sparse

```
1     // access first sparse and second sparse
2     struct s_item<IdxType, ValType> *first = (struct s_item<IdxType, ValType> *)
          first_s->items;
3     struct s_item<IdxType, ValType> *second = (struct s_item<IdxType, ValType> *)
          second_s->items;
4
5     if((len_first + len_second) * (sizeof(IdxType) + sizeof(ValType)) >= dim *
          sizeof(ValType)) {
6       // Make dense in temp buf and return that
7
8       tmpbuf->nofitems = dim;
9       ValType * const __restrict__ result = (ValType *)tmpbuf->items;
10
11      #pragma omp simd
12      for(size_t i = 0; i < dim; ++i) {
13        result[i] = 0.0;
14      }
15
16      // Sum sparse vector
17      while(p1 < len_first || p2 < len_second) {
18        if((p1 == len_first) || (p2 != len_second && (second[p2].idx < first[p1].idx
              ))) {
19          result[second[p2].idx] = second[p2].val;
20          p2++;
21        } else if((p2 == len_second) || (first[p1].idx < second[p2].idx)) {
22          result[first[p1].idx] = first[p1].val;
23          p1++;
24        } else {
25          // index of receiver as index of sender must be equal
26          result[first[p1].idx] = first[p1].val + second[p2].val;
27          p1++;
28          p2++;
29        }
30      }
31
32      return tmpbuf;
33    } else {
34      // Result will be sparse
35      int newLen = 0;
36      struct s_item<IdxType, ValType> *result = (struct s_item<IdxType, ValType> *)
            tmpbuf->items;
```

```
37
38      // Sum sparse vector
39      while(p1 < len_first || p2 < len_second) {
40        if((p1 == len_first) || (p2 != len_second && (second[p2].idx < first[p1].idx
            ))) {
41          result[newLen].idx = second[p2].idx;
42          result[newLen].val = second[p2].val;
43          p2++;
44        } else if((p2 == len_second) || (first[p1].idx < second[p2].idx)) {
45          result[newLen].idx = first[p1].idx;
46          result[newLen].val = first[p1].val;
47          p1++;
48        } else {
49          // index of receiver as index of sender must be equal
50          result[newLen].idx = first[p1].idx;
51          result[newLen].val = first[p1].val + second[p2].val;
52          p1++;
53          p2++;
54        }
55        newLen++;
56      }
57
58      tmpbuf->nofitems = newLen;
59      return tmpbuf;
60    }
```

1.b If both vectors are already dense, one can rely on a dense vector summation and does not need to allocate a new stream, but can rather sum the results *inplace* into either u_1 or u_2. The complexity is $\mathcal{O}(N)$. The code is given in Listing 4.4.

Listing 4.4: Potentially overlapping stream summation - Both input dense

```
1    if(len_first == dim && len_second == dim) {
2      // Sum second into first return first
3      ValType *first = (ValType *)first_s->items;
4      const ValType * const __restrict__ second = (const ValType *)second_s->items;
5
6      #pragma omp simd
7      for(size_t i = 0; i < dim; ++i) {
8        first[i] += second[i];
9      }
10     return first_s;
11   }
```

1.c / 1.d Both cases are symmetric. One has to iterate over all the index-value pairs stored in the sparse vector and has to set the value at the corresponding position in the dense vector. No additional memory has to be allocated and the computation complexity is $\mathcal{O}(|H_1|)$ and $\mathcal{O}(|H_2|)$ respectively. The code is given in Listing 4.5.

Listing 4.5: Potentially overlapping stream summation - One input sparse

```
1    if(len_first == dim) {
2      // Sum second into first return first
3      ValType * first = (ValType *)first_s->items;
4      const struct s_item<IdxType, ValType> *second = (const struct s_item<IdxType,
           ValType> *)second_s->items;
5
6      for(size_t i = 0; i < len_second; ++i) {
7        first[second[i].idx] += second[i].val;
8      }
9      return first_s;
```

```
10   }
11
12   if(len_second == dim) {
13     // Sum first into seconnd return second
14     const struct s_item<IdxType, ValType> *first = (const struct s_item<IdxType,
         ValType> *)first_s->items;
15     ValType *second = (ValType *)second_s->items;
16
17     for(size_t i = 0; i < len_first; ++i) {
18       second[first[i].idx] += first[i].val;
19     }
20     return second_s;
21   }
```

For the scenario (2) where there is no overlap, we give similar explanations on how this is implemented efficiently. Notice that the code is much more complicated due to the fact that N might not be properly divisible by P. This introduces several additional checks which makes the code hard to read and is therefore not included into this document.

2.a The result remains sparse as the dimension is doubled and both values satisfy $|H_1| \leq \varphi(\frac{N}{P})$ and $|H_2| \leq \varphi(\frac{N}{P})$. Due to linearity in φ, it holds that $|H_1| + |H_2| \leq 2 \times \varphi(\frac{N}{P}) = \varphi(\frac{2N}{P})$. In order to prevent additional memory allocation and movement at the same time, we need to determine which sparse vector has values with smaller indices (all the indices are either smaller or bigger in one or the other vector). If this is the case for u_1, we can copy all the index-value pairs from u_2 to the end of u_1, increase the number of elements in u_1 by the amount of pairs found in u_2, and return u_1 as the resulting stream. Otherwise, we perform the same procedure simply by interchanging u_1 and u_2. The complexity is either $\mathcal{O}(|H_2|)$ or $\mathcal{O}(|H_1|)$.

2.b This scenario is identical to 2.a. As we do not have any indices though, we need to know which dense vector represents which subspace in oder to concatenate the values in the correct order. The complexity for both cases is $\mathcal{O}(\frac{N}{P})$.

2.c / 2.d Those two cases are slightly more complicated. If we assume that we need the same number of bits to store an index and a value element (`sizeof(IdxType) = sizeof(ValType)`), it implies that $\varphi(N) = \frac{N}{2}$. As a consequence, $\varphi(\frac{2N}{P}) = \frac{N}{P}$. As one of both vectors is dense, it contributes $\frac{N}{P}$ items and the overall resulting size is even higher. Thus, the concatenated result itself should again be stored in a dense format. Only if the vector from the subspace with smaller indices is the dense one, we can set the consecutive $\frac{N}{P}$ values to either 0 or take the value from the second sparse vector. Otherwise, we need to allocate additional space, set all the values coming from the sparse vector as well as performing a memory copy of all the dense values. The overall complexity in space and computation is therefore given by $\mathcal{O}(\frac{2N}{P})$.

4.1.4 Computation and Communication Overlap

As mentioned before, we decided to implement the *Split* part by sending directly all the values each node gets responsible for to its intended destination. This causes the highest imaginable latency costs of $(P-1)\alpha$, where a recursive halving like algorithm could potentially reach a lower bound at $\log_2(P)\alpha$. Nevertheless, such an algorithm splits its stream, exchanges parts of the data with another node, calculates the sum into its stream based on the received data and start over again. This is a purely sequential behavior and there is no room for overlapping computation and communication. Opting for a direct all-to-all method, one can nicely overlap the computation and communication costs. After having split its stream by linearly iterating over it, each node sends the part of the data to every node it is responsible for, using a non-blocking MPI call. As a consequence, every node is able to perform the stream summation on sparse vectors of dimension $\frac{N}{P}$ as soon as the first results are completely received. Sample code for overlapping this communication and computation parts, whilst making use of the previously described inplace sparse vector summation function, is given in Listing 4.6.

Listing 4.6: Overlap computation and communication with sparse inplace summation

```
1  struct stream *tmpbuf = NULL;
2  int pending = worldsize-1;
3  while(pending > 0) {
4    int index;
5    MPI_Status status;
6    MPI_Waitany(worldsize-1, &requests[0], &index, &status); // request should be
         automatically changed to MPI_REQUEST_NULL by Waitany
7    if(index == MPI_UNDEFINED) {
8      printf("Unexpected error!\n");
9      MPI_Abort(MPI_COMM_WORLD, 1);
10   }
11
12   tmpbuf = sum_into_stream<IdxType, ValType>(mytmp1, recvs[index], mytmp2, dim);
13   if(tmpbuf == recvs[index]) {
14     recvs[index] = mytmp1;
15     mytmp1 = tmpbuf;
16   } else if(tmpbuf == mytmp2) {
17     mytmp2 = mytmp1;
18     mytmp1 = tmpbuf;
19   }
20
21   pending--;
22 }
```

4.1.5 Non-Blocking Variants

As described in section 2.3.4, CNTK implements the gradient aggregation of every tensor separately using non-blocking CUDA and MPI calls. As MPI offers the usage of non-blocking collectives such as IAllReduce (the *I* indicates this is a non-blocking call), we need to come up with non-blocking variants of the algorithms in order to incorporate them into CNTK. We make

use of the framework LibNBC[1] for this purpose. LibNBC is a prototypic implementation of a nonblocking interface for MPI collective operations. We follow the documentation given by Hoefler et al. [25]. As there are missing parts for handling our previously described streams for sparse and dense vector representation, slight modifications to the original code have to be performed.

4.2 MPI-SGD

This framework is built entirely in native `C++11` without the need of linking any external libraries neither dynamically, nor statically. The starting point for implementing data-parallel SGD is the code developed by the authors of Hogwild! [37], especially the header-only library given by the Hazy project at Stanford lead by Christopher Re[2]. Parts of this library need to be adopted and thus are completely rewritten. Some files are still being used though, and are therefore properly integrated and referenced in MPI-SGD in a separate folder. We do not give an entire documentation of MPI-SGD in this work, but mainly focus on an overview over some features and highlight the generic way for implementing both, new loss functions and different aggregation strategies. The major feature of MPI-SGD is to run data-parallel SGD on multiple compute nodes communicating via any MPI library. To this end, the following main functionalities are implemented in this framework:

1. Efficient distributed read of any dataset converted in the predefined format using MPI-IO
2. Run data-parallel SGD on multiple compute nodes (communicating using MPI functionalities) simultaneously with multiple threads on every machine (communicating through shared memory)
3. Implementation of two learning rate adaptation strategies
4. Incorporate proper profiling for getting distinct values for the entire running time per epoch, the time spent for computation and communication respectively
5. Train linear models for regression and classification
6. Use various gradient aggregation strategies representing sparse, dense, synchronous and asynchronous variants of AllReduce and parameter-server patterns.

The last two points are implemented in a generic way, such that one could rapidly extend the framework by adding any additional arbitrary model and

[1] `https://htor.inf.ethz.ch/research/nbcoll/libnbc/`
[2] `http://i.stanford.edu/hazy/victor/Hogwild/`

aggregation strategy without having to change any other parts. We give a short overview of both those features in the next sections.

MPI-SGD is available in a Gitlab repository[3] and has to be considered as *prototypic*. That is, the author does not take any responsibilities for possible bugs in the code or not yet fully implemented functionalities. The repository contains various branches as this framework is used to run tests in several other projects.

4.2.1 Linear Models

Every linear model in MPI-SGD is called an *app* and its header file implementation is stored in a folder by the same name. As by the date of delivery, three models are implemented: linear regression, logistic regression and support vector machines. In order to add any additional model, a new class has to be written which implements the three methods shown in Listing 4.7.

Listing 4.7: Example model execution class

```
1  #ifndef _EXAMPLE_MODEL_EXEC_H
2  #define _EXAMPLE_MODEL_EXEC_H
3
4  #include "hazy/vector/dot-inl.h"
5  #include "hazy/vector/operations-inl.h"
6
7  class ExampleModelExec
8  {
9    public:
10     static void CalcModelUpdate(LinearModelSample const * const &samples, size_t *
          current_batch, size_t actual_num_elements_in_batch, LinearModel *model,
         LinearModelParams const &params, unsigned tid) { ... }
11
12     static fp_type SingleLoss(const LinearModelSample &s, LinearModel *m) { ... }
13
14     static fp_type ComputeMetaLoss(const LinearModelSample &s, LinearModelParams
         const &params) { ... }
15 };
16
17 #endif
```

The function `CalcModelUpdate` calculates the gradient based on the loss function and the current weight vector. The result has to be stored into the variable `local_gradients[tid]` in the model object. `SingleLoss` returns the loss for a given sample based on the current model weight vector. `ComputeMetaLoss` on the other hand returns the loss based on the true model parameters. Furthermore, in order to generate the binaries for a new *app*, a single line cpp file has to be created, which calls the `run` method in the `linearmodel_exec` class with the previously mentioned executor class as a template argument. The `main` file has then to be added to the *Makefile* in order to generate all the variations for this new *app*.

[3]http://www.gitlab.com/crenggli/MPI-SGD

4.2.2 Aggregation Strategies

The implemented aggregation strategies are saved in the folder `src/strategy`. As by the date of delivery, the following gradient aggregation strategies are implemented: *AllReduce*, *ParameterServer synchronous*, *ParameterServer asynchronous* and *sparse AllReduce*. The last one is further divided into the three algorithms described earlier in this document by using preprocessor variables. Every strategy has to inherit from the fully virtual class `Executor`. Therefore, all the functions one has to implement are visualized in Listing 4.8.

Listing 4.8: Example aggregation strategy

```
1  #ifndef _EXAMP_STRAT_H
2  #define _EXAMP_STRAT_H
3
4  #include "executor.h"
5
6  template< class Model, class Params, class Sample, class Loader, class Exec>
7  class ExampleStrategy : public Executor<Model, Params, Sample, Loader, Exec>
8  {
9  public:
10   ExampleStrategy() : Executor<Model, Params, Sample, Loader, Exec>() { ... }
11
12   int GetWorkerNumber() { ... }
13
14   void GetModel(hazy::util::Clock &communicate_timer) { ... }
15
16   void SendModelUpdate(hazy::util::Clock &communicate_timer) { ... }
17
18   void PreEpoch(hazy::util::Clock &communicate_timer) { ... }
19
20   void PostEpoch(hazy::util::Clock &communicate_timer) { ... }
21
22   void InitStrategy() { ... }
23 };
24
25 #endif
```

The names of the functions are mostly self-explanatory. `InitStrategy` is called once at the beginning of the execution. `PreEpoch` and `PostEpoch` are called before and after each epoch respectively. The function `GetWorkerNumber` is used if there is one or multiple servers in the parameter-server paradigm. Therefore, each worker has to be assigned an unique id starting at 0, which is handeled by this function. In an epoch, the new model is always received by the worker calling `GetModel`. After having calculated the gradient for a minibatch, the worker (or every worker) calls `SendModelUpdate` to either, send the updates to the server, or perform an AllReduce with all the other nodes. Notice that in the AllReduce case, getting the new model befor calculating the gradient becomes obsolete, so this function can just be left empty. Every new strategy has to be inserted into the Makefile and put into the class `LinearModelExec`, which serves as an executor class for all *apps* and can be found in that coresponding folder.

4.3 CNTK Extensions

We briefly describe the changes we have to perform in CNTK. The code is integrated in a custom branch on a fork from the original CNTK started by a former student of the supervisor[4]. The main functionalities implemented in various branches of this fork concern QSGD [3]. As described previously, we aim to implement Top$K$ SGD into CNTK and as a consequence, we want to aggregate the naturally or enforced sparse gradients by making use of the non-blocking versions of sparse AllReduce described in the prior sections. Similar to the present code introducing QSGD into CNTK, we adopt the base version of 1-bit SGD implemented in the original code to Top$K$ SGD.

4.3.1 Brainscript Parameters

For this goal, we have to introduce additional brainscript parameters. Namely, the number of elements per bucket `numElementsPerBucket`, and `topK` representing the number of absolute top values one whished to select within each bucket. One might notice that selecting the top k elements out of each bucket, is not exactly identical to a selection of the top $k \times number_of_buckets$ elements out of the entire vector. This method nevertheless enables us to speedup the top k selection significantly when working on highly parallelizable GPUs, and does not hinder convergence speed for naturally sparse gradients as we emphasize in the experiement section. Still, as a leftover from the 1-bit SGD implementation, one has to set the number of gradient bits other than a multiple of 32, in order to enable Top$K$ SGD, even though this value does not have any effect on the algorithm.

4.3.2 Memory Allocation

CNTK enables gradient computation to be performed on GPUs. Running backpropagation for determining the gradients on neural networks, the tensors containing the models at every layer have to be stored in device memory. Internode gradient aggregation using MPI on the other hand needs to access the gradient of the models through host memory. As we perform the top k values selection on top of the calculated gradient tensors directly on GPUs, we do not need to copy the initial gradient values back to host memory, but rather can allocate a stream for each layer containing the sparse values. Those streams have to be allocated temporarily on the device and on the host.

[4] https://gitlab.com/demjangrubic/CNTK/tree/ApproxTopK

4.3.3 Approximate Top$K$ Selection

Even though we consider tensors as gradients for every layer, we will refer to the gradient as a vector in the following section. Those tensors are simply reshaped into a column vector before performing the top k selection and internode aggregation. The selection of the top k elements per bucket is executed using GPUs based on the gradient values store in device memory. GPUs are highly efficient when working with a fixed previously known amount of data and having little communication between threads. We want to select the biggest absolute k values in every bucket as fast as possible. This problem requires comparison of data shared between threads, which can become inefficient.

We came up in section 2.4 with an approximate variant of the algorithm called Approximate Top$K$ SGD. This method determines for every value in a vector, based on some probabilities with the L_1 and L_∞ norm of the vector, whether the value should be taken into account for aggregation or has to be added to the residual. In every bucket, we therefore need to efficiently calculate both norms. We assign to every bucket of the vector a distinct block in the computation grid. It is known that when working with a high number of values, it might not be the most efficient way to assign a distinct thread to every item in the bucket. Rather, one might consider assigning multiple items in every bucket to a thread and therefore reduce the number of threads every block gets assigned when calling the CUDA kernel. We tested empirically, when working with bucket sizes of 512, the most efficient code allocates 128 threads per block, and therefore every thread is responsible for 4 items in the vector. For calculating both needed norms we make use of the GPU collective operation *Reduce*. Such collectives are efficiently implemented in the CUB library provided by NVIDIA[5]. Based on the reduced norms and value at each position, every thread can determine whether this value should be taken into account or not. If not, the value can simply be copied into the residual vector in order to ensure convergence. If the value has to be taken, the thread has to save the value and its index into the temporarily allocated stream. For the exact position, each thread has to know how many values are taken by threads with smaller ids though. Luckily, this can again be calculated efficiently using a prefix sum collective function implemented in CUB. Based on this summed up index per taken value, every thread can directly write the index-value pair at the corresponding position in the stream. As we are selecting k values in expectation, we fill up the sparse vector stream with 0 values or truncate to the first k index-value pairs in order to have a fix amount of data for simplicity. This enables every block to know where exactly its k values have to be stored within the entire sparse vector stream allocated for the full gradient vector.

[5] https://nvlabs.github.io/cub/

Chapter 5

Experiments

We validate our algorithms empirically in three different ways:

- We generate synthetic data, with changing dimension N, density d and number of working nodes P. Based on the defined density, k indices out of N are selected uniformly at random, $j \sim \mathcal{U}(1, N)$, at each node. For every selected index, a random value is chosen. On top of those sparse values per nodes, we run our sparse AllReduce algorithms in order to validate both, the correctness and the derived analytical bounds
- We train linear classifiers (Logistic Regression, SVM) on large sized classification datasets using SGD, exploiting natural sparsity by making use of sparse AllReduce on top of the non-zero entries for every model update
- We train deep neural networks by both forcing and exploiting sparsity of gradient updates using Top$K$ SGD.

5.1 Experimental Setup

We conduct all our tests on a CX50 supercomputer CSCS Piz Daint[1]. Each compute node has 12 cores HT-enabled Intel Xeon E5-2690 v3 with 4GB RAM and a NVIDIA Tesla P100 16GB GPU unit, with the latests only used in tests involving CNTK. The internode connection on this supercomputer uses Aries routing and communication ASICs, on a Dragonfly network topology. In all our experiments, we define the MPI_AllReduce implementation on the fully dense vectors to be our baseline. As a MPI implementation we make use of the installed library Cray-MPICH, a specially for this system optimized derivative of MPICH, for fair comparison. The supercomputer

[1] http://www.cscs.ch/computers/piz_daint

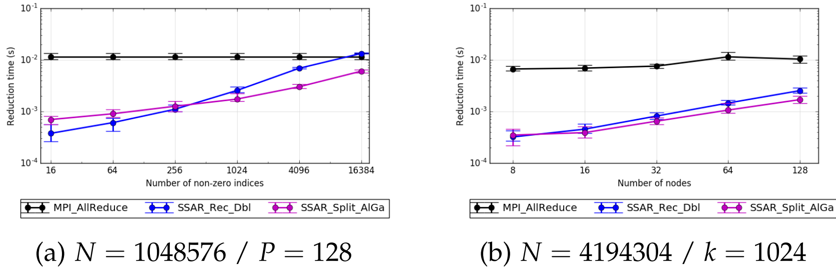
we run all our experiments on is known to have high bandwidth and small latency values. Therefore, we run some tests on an additional infiniband cluster called Greina at CSCS. Greina is a heterogeneous hardware cluster simulating a research or development environment. It has a much higher value in terms of latency and a lower bandwidth parameter compared to Piz Daint. When not stated otherwise, graphs resulting from experiments in the next section where executed on Piz Daint. We do not give, or use exact values for the parameters in the communication model for two reasons. The proposed $\alpha - \beta$ model is known to oversimplify the entire architecture. Additionally, determining exact parameters on a running system is another, self-contained complex topic, we do not investigate within this work. Nevertheless, the analytical model gives us insights on when to use which variant of the algorithms.

5.2 Synthetic Data - Micro Benchmarks

We first focus on generating synthetic data and conducting micro benchmarks. We make use of this to debug and tune the variants of our algorithms. Furthermore, we want to empirically determine switching points between all the algorithms and verify the relative ordering of the derived analytical running times. Bear in mind, that the model is oversimplified and we deal with unknown values not only for the parameters, but also for the exact input at each node. We need to generate the data synthetically following some distribution. One could simulate both extreme cases where $\mathcal{K} = k$ and $\mathcal{K} = k \times P$. This would not give huge insights though, as we know the algorithms which solve those problems efficiently, and are already implemented in MPI libraries. We derive expected runtimes for a uniform distribution of the data and thus stick with this for our experiments. This is also a reasonable choice when having little, or no knowledge about the true distribution of indices in real world examples. Notice that the choice of parameters to generate the data, are within reasonable ranges seen in real world datasets. Most of the graphs are given in a log-log scale for readability reasons. As the resulting execution times are not deterministic, we always conduct five experiments with newly generated data, while running each one for ten times. Based on those 50 resulting runtime values, we state the 25 and 75 percentage quantiles in order to get an idea on the variance of the results.

5.2.1 SSAR_Rec_Dbl vs. SSAR_Split_AlGa

As mentioned previously, we want to verify the relative ordering of the algorithms. First, we start by investigating the Static Sparse AllReduce algorithms. Based on the derived runtimes in terms of latency and bandwidth (see Table 3.1), we see two scenarios where the recursive doubling



(a) $N = 1048576$ / $P = 128$

(b) $N = 4194304$ / $k = 1024$

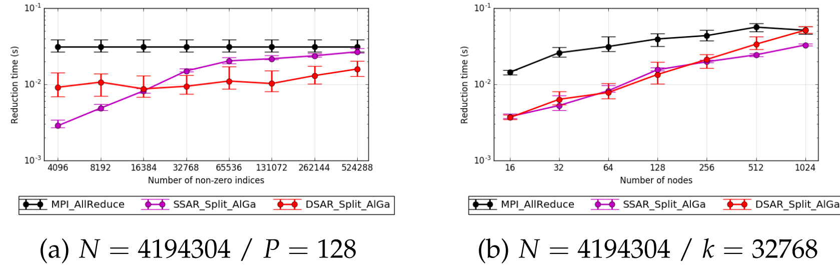
Figure 5.1: SSAR_Rec_Dbl compared to SSAR_Split_AlGa

algorithms SSAR_Rec_Dbl should be faster than SSAR_Split_AlGa . On the one hand, if the amount of data contributed overall is small, latency should dominate. To achieve this, not only k, but also the density has to be low (so high dimension N), especially at a high number of nodes, in order to be significantly faster than MPI_AllReduce. This expected behavior is visible in Figure 5.4. For a small number of non-zero values, the latency clearly dominates and therefore SSAR_Rec_Dbl is favorable over SSAR_Split_AlGa . This impact is less relevant at a higher number of nodes though, as the overall size increases rapidly and we move towards a bandwidth dominated region. Figure 5.1 confirms this claim by showing that the choice between those two algorithms not only depends on the input sizes at each node, but also on the number of nodes itself. Those two graphs correlate with the trend for expected resulting size, formulated for uniform distribution in section 3.2.

On the other hand, if the density is such that at a high number of nodes, the size of the final result reaches the upper bound, we expect to see similar runtimes for both algorithms, if the latency is dominated by the bandwidth. This is visible in Figure 5.4, where at a high number of non-zero indices, the two lines approach each other again, after having been drawn apart. In all the other cases, SSAR_Split_AlGa should dominate over SSAR_Rec_Dbl . Figure 5.1 underlines this assumption. When looking at various graphs, one thing to notice for both algorithms is the fact, that the runtime sometimes remains constant, even the overall amount of data increases, see for instance Figure 5.4a or 5.2a. This behavior is explainable by the automatic switch from a sparse to a dense representation described in the implementation chapter.

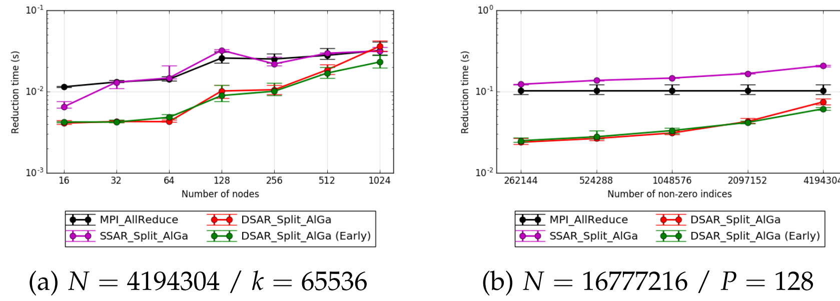
5.2.2 Static vs. Dynamic

We now turn our attention to the previously mentioned threshold δ. Even at a low density, having a big number of nodes can result in an overall reduced vector, for which there is no speedup of storing it in a sparse format compared to a dense representation. Notice that this is both dependent on the initial density at each nodes as well as the number of nodes. Figure 5.2



(a) $N = 4194304$ / $P = 128$ (b) $N = 4194304$ / $k = 32768$

Figure 5.2: SSAR compared to DSAR



(a) $N = 4194304$ / $k = 65536$ (b) $N = 16777216$ / $P = 128$

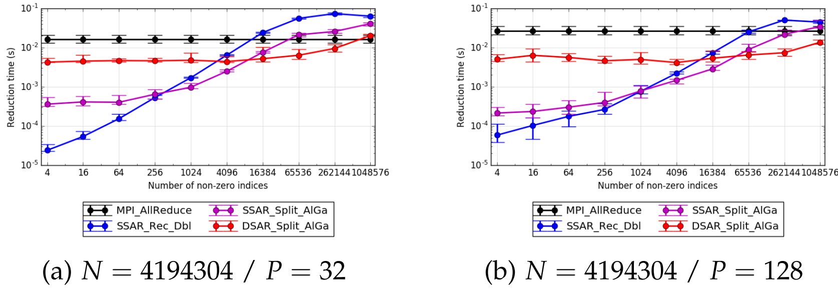
Figure 5.3: Early switching DSAR with groups of size 8

indicates the existence of such a threshold δ. The impact of the number of nodes P and the initial density is visible in those graphs.

DSAR_Split_AG makes use on the MPI specific AllGather implementation for the second part of the algorithm. As mentioned in the previous chapter, most implementations rely on a ring algorithm for large message sizes. Having a high number of nodes though, the additional latency is not negligible anymore. This observation justifies the significant increase in running time for DSAR_Split_AG in Figures 5.2b and 5.3a.

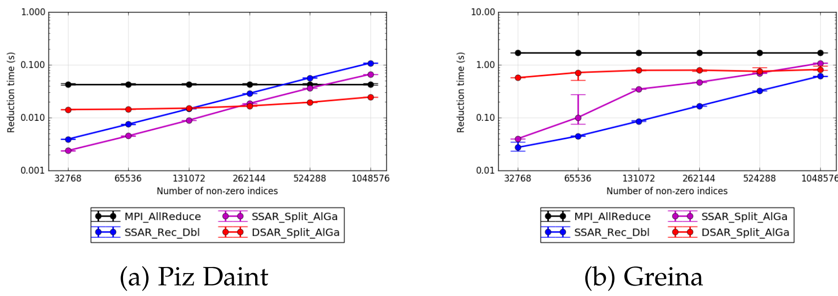
5.2.3 Early Switching DSAR

We test our idea of early switching DSAR for a high number of nodes and high fraction of input densities. We always build groups of 8 nodes, which create dense vectors representing parts of the entire space. The ReduceScatter step is then performed on those subsets of all the nodes in parallel, before running a dense AllGather on all the nodes for interchanging the partial results. Purely empirically, we see the benefit of such a method in Figure 5.3 at high input densities or a huge number of nodes.



(a) $N = 4194304$ / $P = 32$ (b) $N = 4194304$ / $P = 128$

Figure 5.4: All sparse AllReduce algorithms



(a) Piz Daint (b) Greina

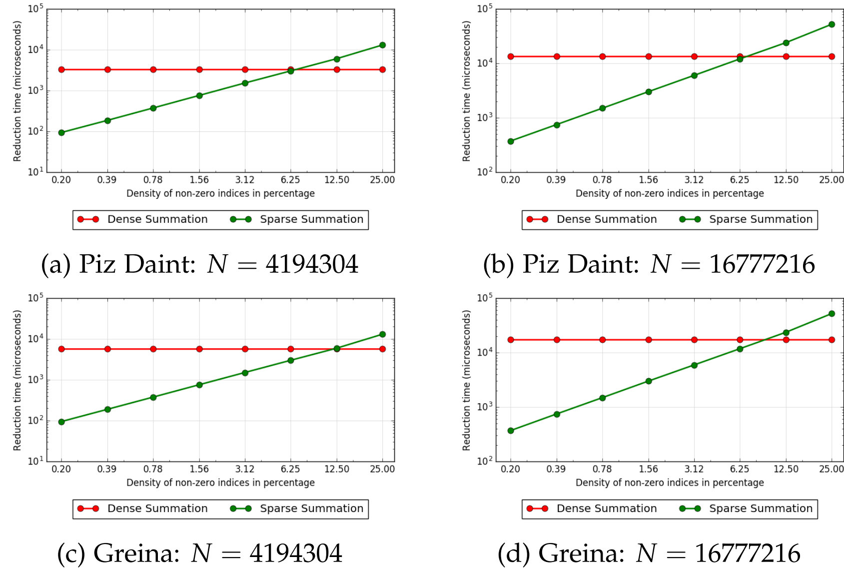
Figure 5.5: System comparison: $N = 16777216$ / $P = 8$

5.2.4 All Algorithms

The theoretical analysis suggests that there are system dependent threshold values, at which one should switch between the variants of the algorithms. This claim is justified empirically in Figure 5.4. The number of parameters given by the problem, its inputs and the system specific parameters hinders us to give constant threshold value. Parametrized closed-form thresholds are imaginable, but are omitted here due to their resulting complexity.

To show the impact of the system on the choice of the algorithm, we run an identical test on both Piz Daint and Greina in Figure 5.5.

Ideally, carefully implemented algorithms automatically switching to dense representations should never result in bigger running times than a dense AllReduce on the entire dimension N. All the graphs underline this claim up to small deviations and enables the use of any of the algorithms for real world examples. Higher runtimes, visible for instance in Figure 5.3b, are assumed to come from the definition of switching point between sparse and dense representations in the code. As pointed out in the implementation chapter, the choice of this threshold should not only rely on the amount of data, as currently done in the code, but rather should take into account the overhead introduced by the sparse summation compared to its dense counterpart.



(a) Piz Daint: $N = 4194304$

(b) Piz Daint: $N = 16777216$

(c) Greina: $N = 4194304$

(d) Greina: $N = 16777216$

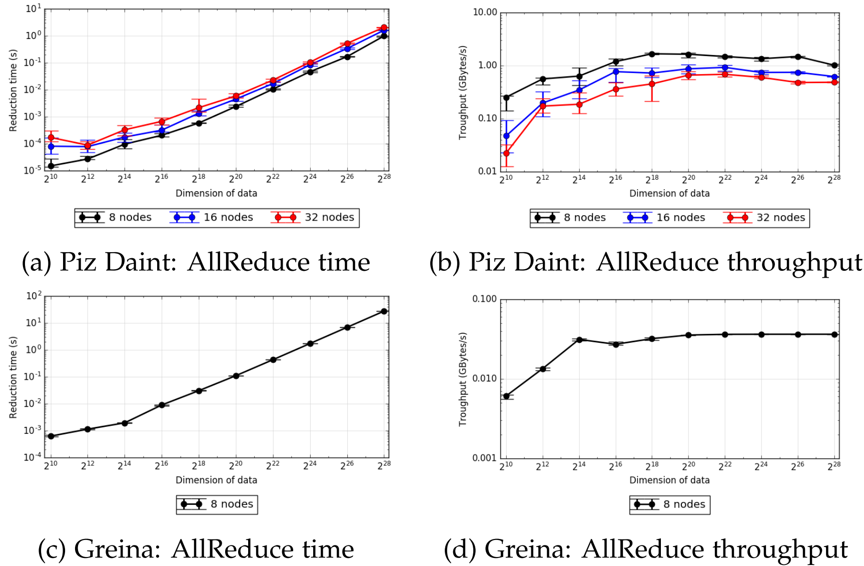
Figure 5.6: Sparse Summation comparison

5.2.5 Sparse Summation

For implementing efficient sparse summation, we compare the resulting operation times with a natural dense vector addition. We run the overlapping version of sparse vector summations for synthetically generated sparse vectors with an alternative number of non-zero indices (uniformly distributed) and an input dimension N. Those results are compared to the dense summation with enabled SIMD optimization and a for the compiler easily optimizable single for-loop. We run the experiments for multiple input dimensions on one node of each system Piz Daint and Greina. From the results in Figure 5.6 we see that at a sparse vector having 6.25% density, the overhead of summing up such sparse vectors makes the operation as expensive as summing up the vectors in a dense representation. Based on those empirical values, one can try to tune the threshold for automatically switch to a dense representation in order to balance the impact of both, this operation and the overall amount of data added by switching to a dense representation.

5.2.6 Lower Bounds

We give lower bounds for MPI_AllReduce calls with an increasing number of items. Every item needs 4 bytes of representation. Based on the amount of bytes needed and the duration it takes, one can derive the throughput, the AllReduce operation achieves on both machines Piz Daint and Greina. Figure 5.7 shows the lower bound one can achieve on such systems having those numbers of sparse input values and fully overlapping indices. Addi-



(a) Piz Daint: AllReduce time

(b) Piz Daint: AllReduce throughput

(c) Greina: AllReduce time

(d) Greina: AllReduce throughput

Figure 5.7: Sparse Summation comparison

tionally, one sees the throughput each system achieves for various numbers of nodes. Notice that the throughput saturates once we find ourselves in a bandwidth dominated region.

5.3 Training of Linear Classifiers

We run our sparse AllReduce algorithms for training linear classifiers using MPI-SGD, setting the batch size to $1000 \times P$. As convergence is not affected by omitting zero valued elements, we do not focus on tuning the other hyperparameters such as learning rate or momentum. We exploit natural sparsity in the model updates exchanged between nodes in order to achieve a speedup of factors up to 3.5x on 32 and 128 nodes compared to dense AllReduce on the supercomputer Piz Daint. On the infiniband cluster Greina, we get a speedup of factor 20x on 8 nodes.

5.3.1 Datasets

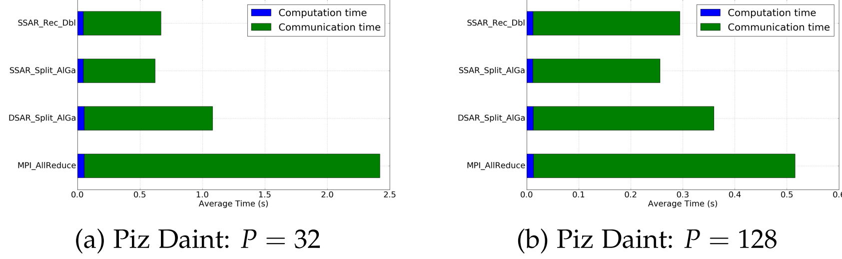
We run all our tests on two large scale binary classification datasets, given in Table 5.1.

5.3.2 Logistic Regression

Figure 5.8 and 5.9 give results for training a logistic regression model on both datasets executed on Piz Daint. The relative difference between 32 and 128 nodes for the URL dataset is as expected, based on the synthetic

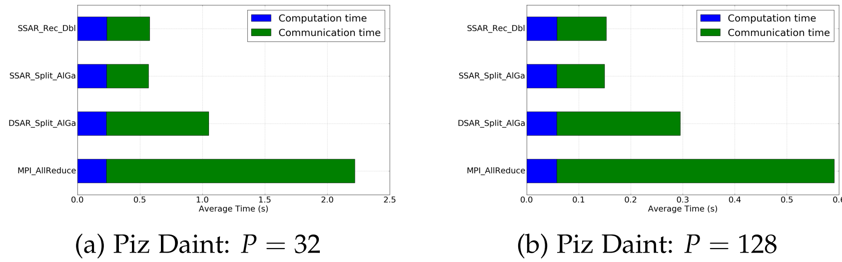
Name	Classes	Number of samples	Dimension
URL [31]	2	2 396 130	3 231 961
Webspam[2]	2	350 000	16 609 143

Table 5.1: Datasets for training Linear Classifiers



(a) Piz Daint: $P = 32$ (b) Piz Daint: $P = 128$

Figure 5.8: Average logistic regression epoch time for the dataset 'URL'



(a) Piz Daint: $P = 32$ (b) Piz Daint: $P = 128$

Figure 5.9: Average logistic regression epoch time for the dataset 'Webspam'

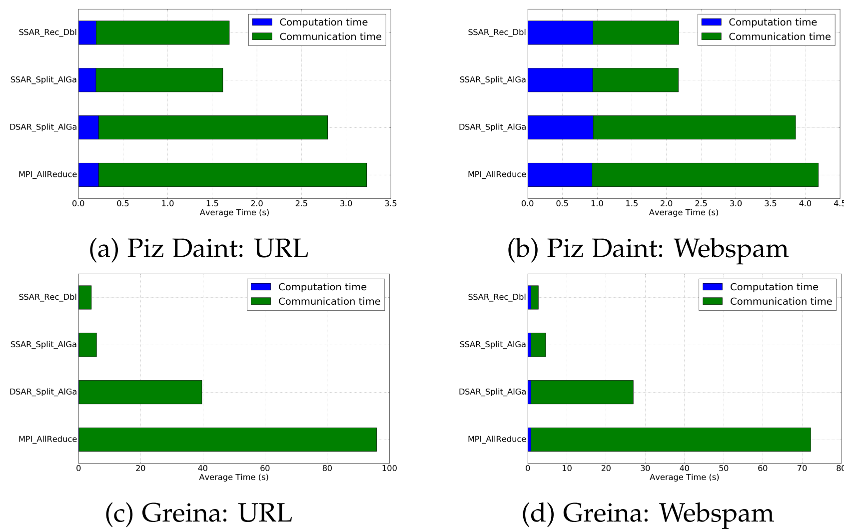
experiments. For Webspam, the number of nodes does not affect the relative difference much. The reason for this is treated in section 5.3.4.

Figure 5.10 illustrates the impact of the underlying network architecture. For the same problem and number of nodes, we achieve a speedup of factor 2x on Piz Daint, whereas the speedup on the infiniband cluster Greina reaches nearly 20x. Similar to the findings based on the synthetic micro benchmarks, the choice of the best algorithm in terms of speed can vary on different hardware architectures.

5.3.3 SVM

We conduct the same experiments for training support vector machines. The only difference to logistic regression lies in the gradient computation, which does not affect the communication time. Figure 5.11 supports this claim, as we get similar results in Figure 5.9a and 5.8a.

[2] https://www.cc.gatech.edu/projects/doi/WebbSpamCorpus.html



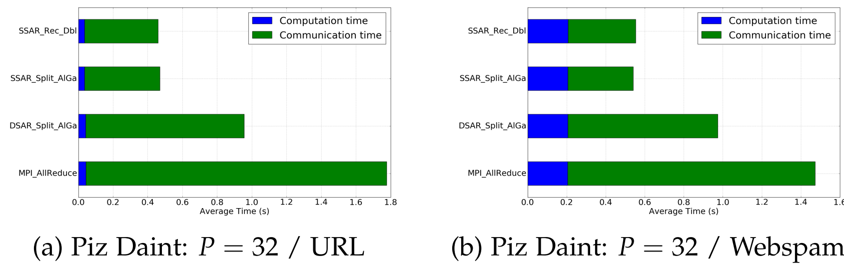
(a) Piz Daint: URL

(b) Piz Daint: Webspam

(c) Greina: URL

(d) Greina: Webspam

Figure 5.10: Average logistic regression epoch time compared on 8 nodes between Piz Daint and Greina



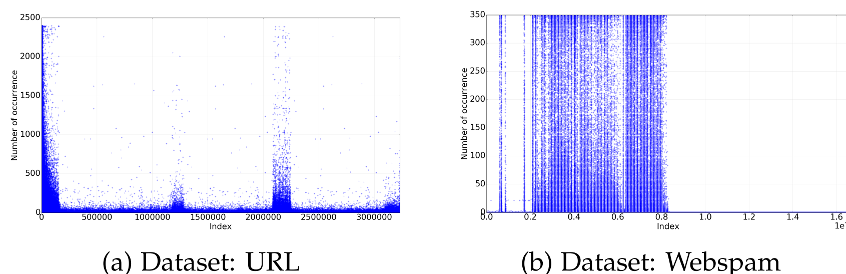
(a) Piz Daint: $P = 32$ / URL

(b) Piz Daint: $P = 32$ / Webspam

Figure 5.11: Average SVM epoch time for the dataset 'Webspam'

5.3.4 Sparsity Analysis

Figures 5.8 and 5.9 suggest that there is a difference in the distribution of the non-zero indices between the two datasets when running logistic regression. We perform an empirical analysis by plotting the number of occurrences in a single epoch (run over the entire dataset) of each index. From Figure 5.12, one can deduct that the indices affected at every iteration in the webspam dataset are not well distributed over the entire space. Therefore, the resulting size stops growing after having a certain number of participating nodes. This explains the missing difference between the runtime ratios in the above graphs.



(a) Dataset: URL (b) Dataset: Webspam

Figure 5.12: Sparsity analysis of non-zero indices contributed by each iteration

5.4 Training of Deep Neural Networks

We train various deep neural networks using CNTK and the described variation of approximate Top$K$ SGD in section 2.4. $[NUM]$ in the labels $Top[NUM]$ represents the expected number of elements selected within each bucket of 512 elements. Theoretically, this implies k being equal to $[NUM] * \frac{N}{512}$. This algorithm makes use of naturally sparse model updates if present, or forces them to be at a desired density level. We run the experiments relying on the non-blocking version of SSAR_Rec_Dbl on all the layers with dimension $N \geq 10000$. All the experiments are conducted with default parameters provided in the example scripts by CNTK. We follow a weak-scaling strategy when increasing the number of nodes. That is, we choose B to be the maximal number of samples a GPU can store in memory, and thus have a mini-batch size of $P \times B$. For the convergence tests, we illustrate the evolution of the training error per epoch or time, as well as the top-1 accuracy on the test samples, if available. In this second graph, one would remark overfitting on the model parameters, if this happens to occur. We run all the experiments on the CSCS supercomputer Piz Daint. As long runs are not only time consuming, but also costly on this cluster, we run full convergence tests only on small datasets.

5.4.1 Datasets

We run all our tests for training deep neural networks using the datasets given in Table 5.2. The top three entries are used for image classification, whereas the last one contains natural language understanding (NLU) content. The number of samples for the ATIS corpus is given in sentences (s) and words (w).

Name	Classes	Number of samples	Dimension
ImageNet 2012 [39]	1000	1,2 million	256x256x3
MNIST [28]	10	60 000	28x28
CIFAR-10 [27]	10	60 000	32x32x3
ATIS [22]	128	4 978 s / 56 590 w	-

Table 5.2: Datasets for training deep neural networks

5.4.2 Image Classification

Convergence of Top$K$ SGD

We start by showing that a Top$K$ algorithm, compared to a classical SGD, can lead to a lower convergence rate. For this, we train the ResNet110 neural network [21] using the CIFAR-10 dataset. The graphs in Figure 5.13 point out two observations:

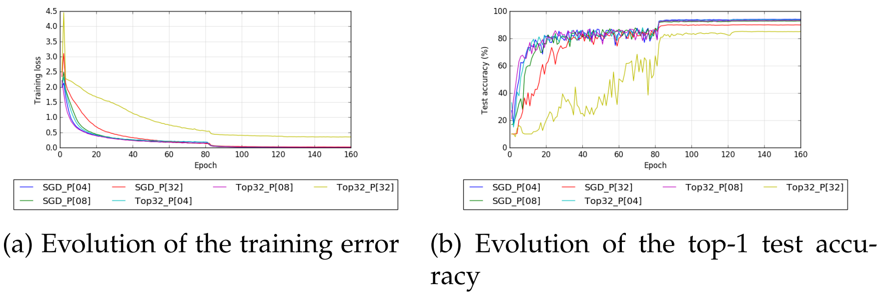
1. The convergence of SGD running on 32 nodes is slower than on 8 nodes
2. Selecting Top32 (32 elements per bucket of size 512) results in lower convergence rate on 32 nodes, but not on 8 nodes.

(1) comes from the fact, that we do not tune any learning rate, even though we have bigger mini batch sizes due to weak scaling. (2) underlies the claim in section 2.4, stating that the convergence speed might be affected when postponing the contribution of smaller values and not having naturally sparse model updates. Nevertheless, this seems to happen above a certain number of nodes only. We can think of two possible explanations for this:

1. The variance of each stochastic gradient at a higher number of nodes is lower due to weak scaling, which can significantly affect the convergence properties of Top$K$ SGD
2. The truncation function on top of the gradients is non-linear.

For the second point we realize that taking the biggest k absolute values before and after summing up the partial model updates on every node is not necessarily identical. This fact could become more significant if the model update is distributed over a larger number of nodes.

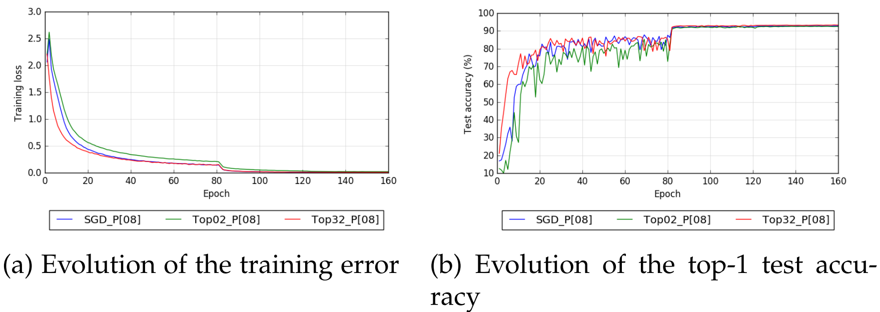
Based on those results, we want to determine empirically how small k can be, in order to still ensure convergence on a small number of nodes. Apparently, selecting the top 2 out of 512 items per gradient at every layer, is sufficient for this network and dataset to ensure convergence rate. This is visible in Figure 5.14.



(a) Evolution of the training error

(b) Evolution of the top-1 test accuracy

Figure 5.13: Convergence of ResNet110 with CIFAR-10: Comparision between classical SGD and Top$K$ on 4, 8 and 32 nodes



(a) Evolution of the training error

(b) Evolution of the top-1 test accuracy

Figure 5.14: Convergence of ResNet110 with CIFAR-10 using 8 nodes

Speedup

The above tests are based on a ResNet architecture, which consists of many small dimensioned layers. It is therefore not well suited for reducing the runtime by enforcing sparsity. The Top02 run for instance, was executed in 32 minutes and 20 seconds, whereas the classical SGD pass took 34 minutes and 20 seconds. This represents only a speedup of a factor 1.06x.

What could we theoretically achieve on a bigger dataset and possibly better suited large scale network? For answering this question, we run 200 iterations of training VGG19 [41] on the large dataset ImageNet. The computation and communication costs are roughly equal for this setting. This implies that we can at most achieve a speedup of a factor 2x by lowering the amount of data communicated. We notice in Figure 5.15 a significant speedup of factor 1.5x arising from lowering the communication time as expected. Due to cost and time constraints on the cluster, a full convergence test with this setting has not been performed during this work.

Similar to both references suggesting the Top$K$ algorithm [1, 14], we run an image classification task using the dataset MNIST on a fully connected neural network. We design the network to consist of two hidden fully con-

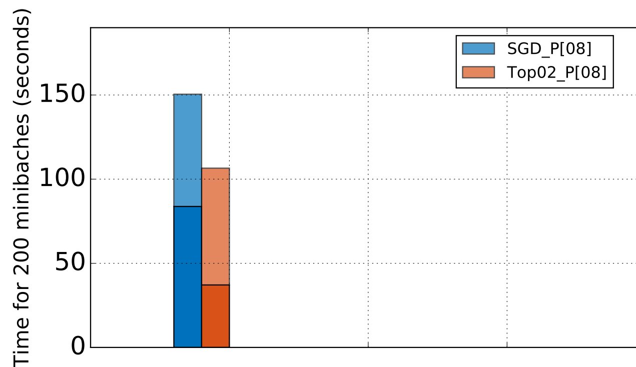
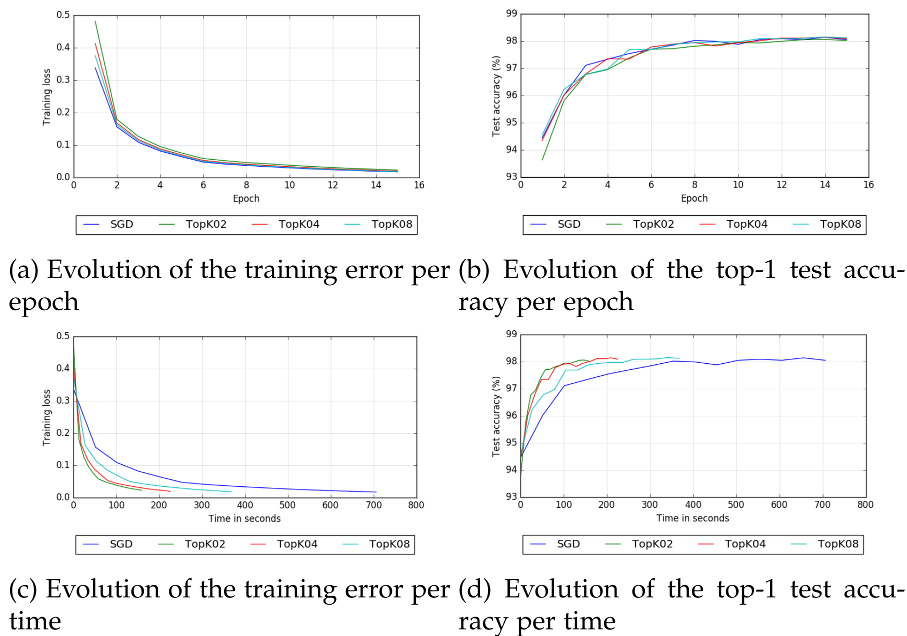


Figure 5.15: Speedup training ImageNet on VGG19. The top part represents the computation time, whilst the lower part indicates the communication time



(a) Evolution of the training error per epoch

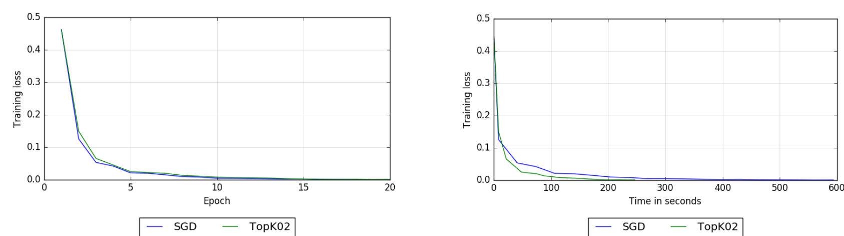
(b) Evolution of the top-1 test accuracy per epoch

(c) Evolution of the training error per time

(d) Evolution of the top-1 test accuracy per time

Figure 5.16: Results of Fully Connected Neural Network with MNIST on 8 nodes

nected layers of dimension 4096. Figure 5.16 indicates the time needed and its accuracy reached. We achieve a speedup of 4.6x without loss of test or training accuracy.



(a) Evolution of the training error per epoch

(b) Evolution of the training error per time

Figure 5.17: Results of LSTM with ATIS on 8 nodes

5.4.3 Natural Language Understanding

As a last empirical verification for the communication cost reducing algorithmic idea, we conduct a single experiment on a natural language understanding problem. We run a small dataset test on the ATIS corpus using an encoder-decoder network consisting of two LSTM cells each. Due to missing large scale datasets, we are only able to run this experiment on 8 nodes. We show a speedup of 2.5x in Figure 5.17. It remains to test whether the same loss of accuracy at a high number of nodes is visible for those types of problems. If not, for instance if the model updates are naturally sparse, this network architecture, in combination with this type of problem, is ideally suited for applying Top$K$ SGD with sparse AllReduce and hence reducing the communication time needed at a high number of nodes.

Chapter 6

Discussion

In this work, we give a formal definition, design efficient algorithms for the sparse AllReduce problem and show that it can be regarded as a generalization of the AllReduce and AllGather collective operations simultaneously. Especially under a uniform distribution assumption, the intermediate and resulting reduced vector sizes are not only dependent on the input at each node, but also rely on the number of involved processes. This fact makes it infeasible to achieve high speedup in terms of reduced communication cost, when dealing with moderate sparsity and a large number of nodes. Nonetheless, large scaled machine learning models with natural sparsity can benefit of a sparse AllReduce to reduce communication costs significantly on slow network architectures. Enforcing sparsity on machine learning problems, by applying a Top$K$ algorithm, only seems to maintain convergence speed at a low number of nodes and small mini-batch sizes. For 8 nodes, we can achieve a speedup up to a factor of 4x combining all the methods proposed in this thesis. The results suggest that the sparse AllReduce is perfectly suited for training very large models on a reasonably small number of nodes. On the downside, the method lacks of performance increase at a higher number of workers due to both, the nature of the problem, and the loss in convergence when forcing the updates to be sparse.

Some potential topics to further investigate include the analysis of other data distributions and the sparsity evaluation of various deep learning problems, especially those involving text or language content. This work entirely focuses on data-parallel SGD as a main use-case for sparse AllReduce. As pointed out in the related work section, other non machine learning fields, such as distributed graph algorithms, can further benefit of those sparse structures. One might also envisage to make use of the same algorithms for solving efficiently other optimization methods such as distributed coordinate descent or model-parallel SGD.

Bibliography

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

[2] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. Loggp: incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM, 1995.

[3] Dan Alistarh, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Randomized quantization for communication-optimal stochastic gradient descent. *arXiv preprint arXiv:1610.02132*, 2016.

[4] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[5] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

[6] Sébastien Bubeck et al. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning*, 8(3-4):231–357, 2015.

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

[8] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

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

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

[11] David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 215–242, 1958.

[12] Christopher De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. Taming the wild: A unified analysis of hogwild. *Style Algorithms. In NIPS*, 2015.

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

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

[15] John C Duchi, Sorathan Chaturapruek, and Christopher Ré. Asynchronous stochastic convex optimization. *arXiv preprint arXiv:1508.00882*, 2015.

[16] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, Sep. 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. http://www.deeplearningbook.org.

[19] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[20] Mert Gürbüzbalaban, Asu Ozdaglar, and Pablo Parrilo. Why random reshuffling beats stochastic gradient descent. *arXiv preprint arXiv:1510.08560*, 2015.

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

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

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

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

[25] Torsten Hoefler and Andrew Lumsdaine. Design, implementation, and usage of libnbc. *Open Systems Lab, Indiana University, Tech. Rep*, 8, 2006.

[26] Torsten Hoefler and Dmitry Moor. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Supercomputing Frontiers and Innovations*, 1(2):58–75, 2014.

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

[28] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

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

[30] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

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

[32] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge university press, 2017.

[33] Bell Nathan and Garland Michael. Efficient sparse matrix-vector multiplication on cuda. Technical report, NVIDIA Technical Report NVR-2008-004, 2008.

[34] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[35] Daniel Povey, Xiaohui Zhang, and Sanjeev Khudanpur. Parallel training of deep neural networks with natural gradient and parameter averaging. *arXiv preprint*, 2014.

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

[37] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[38] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

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

[40] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

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

[42] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

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

[44] Yang You, Igor Gitman, and Boris Ginsburg. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888*, 2017.

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

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Efficient Sparse AllReduce For Scalable Machine Learning

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Renggli	Cédric

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
Zurich, 30. September 2017	C. Renggli

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.