Asynchronous Parallel Stochastic Gradient Descent

A Numeric Core for Scalable Distributed Machine Learning Algorithms

Janis Keuper and Franz-Josef Pfreundt
Fraunhofer ITWM
Competence Center High Performance Computing
Kaiserslautern, Germany
{janis.keuper | franz-josef.pfreundt}@itwm.fhg.de

ABSTRACT

The implementation of a vast majority of machine learning (ML) algorithms boils down to solving a numerical optimization problem. In this context, Stochastic Gradient Descent (SGD) methods have long proven to provide good results, both in terms of convergence and accuracy. Recently, several parallelization approaches have been proposed in order to scale SGD to solve very large ML problems. At their core, most of these approaches are following a MapReduce scheme.
This paper presents a novel parallel updating algorithm for SGD, which utilizes the asynchronous single-sided communication paradigm. Compared to existing methods, Asynchronous Parallel Stochastic Gradient Descent (ASGD) provides faster convergence, at linear scalability and stable accuracy.

Categories and Subject Descriptors

I.2.6 [**Computing Methodologies**]: Artificial Intelligence—*learning*; G.1.6 [**Mathematics of Computing**]: Numerical Analysis—*optimization*

General Terms

Performance, Algorithms

Keywords

high performance computing, stochastic gradient descent, optimization

1. INTRODUCTION

The enduring success of Big Data applications, which typically includes the mining, analysis and inference of very large datasets, is leading to a change in paradigm for machine learning research objectives [4]. With plenty data at

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MLHPC2015, November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-4006-9/15/11 ...$15.00
DOI: http://dx.doi.org/10.1145/2834892.2834893

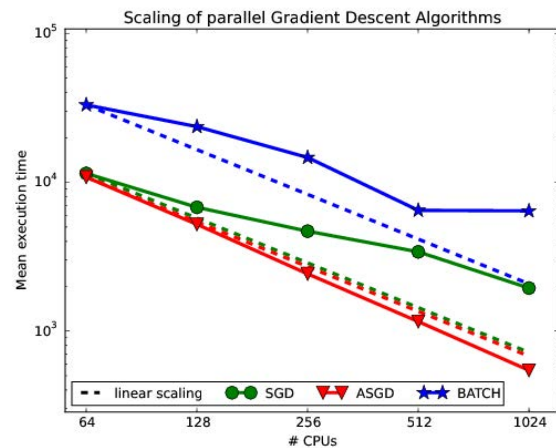

Figure 1: Evaluation of the scaling properties of different parallel gradient descent algorithms for machine learning applications on distributed memory sytems. Results show a K-Means clustering with k=10 on a 10-dimensional target space, represented by ∼1TB of training samples. Our novel ASGD method is not only the fastest algorithm in this test, it also shows better than linear scaling performance. Outperforming the SGD parallelization by [20] and the MapReduce based BATCH [5] optimization, which both suffer from communication overheads.

hand, the traditional challenge of inferring generalizing models from small sets of available training samples moves out of focus. Instead, the availability of resources like CPU time, memory size or network bandwidth has become the dominating limiting factor for large scale machine learning algorithms.
In this context, algorithms which guarantee useful results even in the case of an early termination are of special interest. With limited (CPU) time, fast and stable convergence is of high practical value, especially when the computation can be stopped at any time and continued some time later when more resources are available.
Parallelization of machine learning (ML) methods has been a rising topic for some time (refer to [1] for a comprehensive overview). However, until the introduction of the MapReduce pattern, research was mainly focused on shared memory systems. This changed with the presentation of a generic

MapReduce strategy for ML algorithms in [5], which showed that most of the existing ML techniques could easily be transformed to fit the MapReduce scheme.
After a short period of rather enthusiastic porting of algorithms to this framework, concerns started to grow if following the MapReduce ansatz truly provides a solid solution for large scale ML. It turns out, that MapReduce's easy parallelization comes at the cost of poor scalability [16]. The main reason for this undesired behavior resides deep down within the numerical properties most machine learning algorithms have in common: an optimization problem. In this context, MapReduce works very well for the implementation of so called batch-solver approaches, which were also used in the MapReduce framework of [5]. However, batch-solvers have to run over the entire dataset to compute a single iteration step. Hence, their scalability with respect to the data size is obviously poor.
Even long before parallelization had become a topic, most ML implementations avoided the known drawbacks of batch-solvers by usage of alternative online optimization methods. Most notably, Stochastic Gradient Descent (SGD) methods have long proven to provide good results for ML optimization problems [16]. However, due to its inherent sequential nature, SGD is hard to parallelize and even harder to scale [16]. Especially when communication latencies are causing dependency locks, which is typical for parallelization tasks on distributed memory systems [20].
The aim of this paper is to propose a novel, lock-free parallelization method for the computation of stochastic gradient optimization for large scale machine learning algorithms on cluster environments.

1.1 Related Work

Recently, several approaches [6][20][17][16][13] towards an effective parallelization of the SGD optimization have been proposed. A detailed overview and in-depth analysis of their application to machine learning can be found in [20].
In this section, we focus on a brief discussion of four related publications, which provided the essentials for our approach:

- A theoretical framework for the analysis of SGD parallelization performance has been presented in [20]. The same paper also introduced a novel approach (called SimuParallelSGD), which avoids communication and any locking mechanisms up to a single and final MapReduce step. To the best of our knowledge, SimuParallelSGD is currently the best performing algorithm concerning cluster based parallelized SGD. A detailed discussion of this method is given in section 2.3.

- In [17], a so-called mini-BATCH update scheme has been introduced. It was shown that replacing the strict online updating mechanism of SGD with small accumulations of gradient steps can significantly improve the convergence speed and robustness (also see section 2.4).

- A widely noticed approach for a "lock-free" parallelization of SGD on shared memory systems has been introduced in [16]. The basic idea of this method is to explicitly ignore potential data races and to write updates directly into the memory of other processes. Given a minimum level of sparsity, [16] showed that possible data races will neither harm the convergence nor the accuracy of a parallel SGD. Even more, without any locking overhead, [16] sets the current performance standard for shared memory systems.

- A distributed version of [16] has been presented in [14], showing cross-host CPU to CPU and GPU to GPU gradient updates over Ethernet connections.

- In [8], the concept of a Partitioned Global Address Space programming framework (called GASPI) has been introduced. This provides an asynchronous, single-sided communication and parallelization scheme for cluster environments (further details in section 3.1). We build our asynchronous communication on the basis of this framework.

1.2 Asynchronous SGD

The basic idea of our proposed method is to port the "lock-free" shared memory approach from [16] to distributed memory systems. This is far from trivial, mostly because communication latencies in such systems will inevitably cause expensive dependency locks if the communication is performed in common two-sided protocols (such as MPI message passing or MapReduce). This is also the motivation for SimuParallelSGD [20] to avoid communication during the optimization: locking costs are usually much higher than the information gain induced by the communication.
We overcome this dilemma by the application of the asynchronous, single-sided communication model provided by [8]: individual processes send mini-BATCH [17] updates completely uninformed of the recipients status whenever they are ready to do so. On the recipient side, available updates are included in the local computation as available. In this scheme, no process ever waits for any communication to be sent or received. Hence, communication is literally "free" (in terms of latency).
Of course, such a communication scheme will cause data races and race conditions: updates might be (partially) overwritten before they are used or even might be contra productive because the sender state is way behind the state of the recipient.
We resolve these problems by two strategies: first, we obey the sparsity requirements introduced by [16]. This can be achieved by sending only partial updates to a few random recipients. Second, we introduce a Parzen-window function, selecting only those updates for local descent which are likely to improve the local state. Figure 2 gives a schematic overview of the ASGD algorithm's asynchronous communication scheme. The remainder of this paper is organized as follows: first, we briefly review gradient descent methods in section 2 and discuss further aspects of the previously mentioned related approaches in more detail. Section 3 gives a quick overview of the asynchronous communication concept and its implementation. The actual details of the ASGD algorithm are introduced in section 4, followed by a theoretical analysis and an extensive experimental evaluation in section 5.

2. GRADIENT DESCENT OPTIMIZATION

From a strongly simplified perspective, machine learning tasks are usually about the inference of generalized models from a given dataset $X = \{x_0, \ldots, x_m\}$ with $x_i \in \mathbb{R}^n$, which in case of supervised learning is also assigned with semantic labels $Y = \{y_0, \ldots, y_m\}, y_i \in \mathbb{R}$.
During the learning process, the quality of a model is evaluated by use of so-called loss-functions, which measure how well the current model represents the given data. We write $x_j(w)$ or $(x_j, y_j)(w)$ to indicate the loss of a data point for the current parameter set w of the model function. We will also refer to w as the state of the model. The actual learning is then the process of minimizing the loss over all samples. This is usually implemented via a gradient descent over the partial derivative of the loss function in the parameter space of w.

2.1 Batch Optimization

The numerically easiest way to solve most gradient descent optimization problems is the so-called batch optimization. A state w_t at time t is updated by the mean gradient generated by ALL samples of the available dataset. Algorithm 1 gives an overview of the BATCH optimization scheme. A MapReduce parallelization for many BATCH optimized machine learning algorithms has been introduced by [5].

Algorithm 1 BATCH optimization with samples $X = \{x_0, \ldots, x_m\}$, iterations T, steps size ϵ and states w

1: **for all** $t = 0 \ldots T$ **do**
2: **Init** $w_{t+1} = 0$
3: **update** $w_{t+1} = w_t - \epsilon \sum_{(X_j \in X)} \partial_w x_j(w_t)$
4: $w_{t+1} = w_{t+1}/|X|$

2.2 Stochastic Gradient Descent

In order to overcome the drawbacks of full batch optimization, many online updating methods have been proposed. One of the most prominent is SGD. Although some properties of Stochastic Gradient Descent approaches might prevent their successful application to some optimization domains, they are well established in the machine learning community [2]. Following the notation in [20], SGD can be formalized in pseudo code as outlined in algorithm 2. The advantage in terms of computational cost with respect to the number of data samples is eminent: compared to batch updates of quadratic complexity, SGD updates come at linearly growing iteration costs. At least for ML-applications, SGD error rates even outperform batch algorithms in many cases [2].

Algorithm 2 SGD with samples $X = \{x_0, \ldots, x_m\}$, iterations T, steps size ϵ and states w

Require: $\epsilon > 0$
1: **for all** $t = 0 \ldots T$ **do**
2: **draw** $j \in \{1 \ldots m\}$ uniformly at random
3: **update** $w_{t+1} \leftarrow w_t - \epsilon \partial_w x_j(w_t)$
4: **return** w_T

Since the actual update step in line 3 of algorithm 2 plays a crucial role deriving our approach, we are simplifying the notation in this step and denote the partial derivative of the loss-function for the remainder of this paper in terms of an update step Δ:

$$\Delta_j(w_t) := \partial_w x_j(w_t). \quad (1)$$

2.3 Parallel SGD

The current "state of the art" approach towards a parallel SGD algorithm for shared memory systems has been presented in [20]. The main objective in their ansatz is to avoid communication between working threads, thus preventing dependency locks. After a coordinated initialization step, all workers operate independently until convergence (or early termination). The theoretical analysis in [20] unveiled the surprising fact that a single aggregation of the distributed results after termination is sufficient in order to guarantee good convergence and error rates.
Given a learning rate (i.e. step size) ϵ and the number of threads n, this formalizes as shown in algorithm 3.

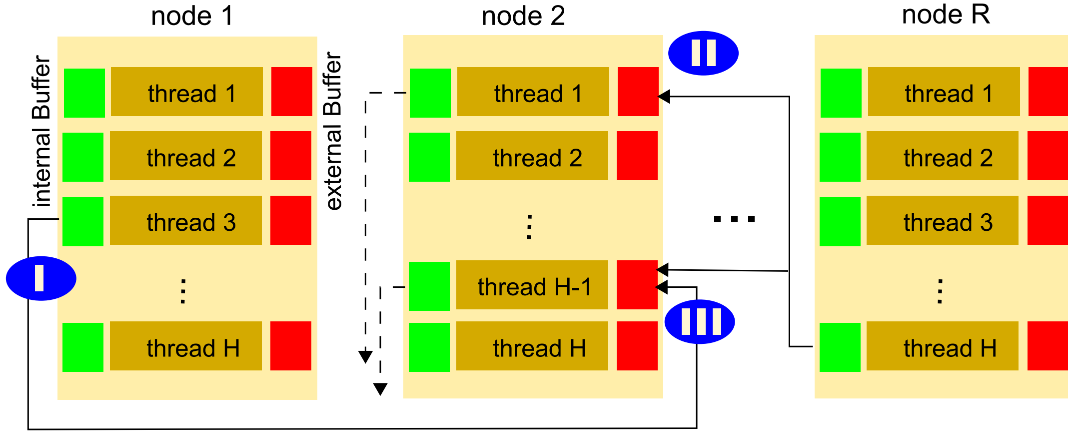

Figure 2: Overview of the asynchronous update communication used in ASGD. Given a cluster environment of R nodes with H threads each, the blue markers indicate different stages and scenarios of the communication mode. I: Thread 3 of node 1 finished the computation of of its local mini-batch update. The external buffer is empty. Hence it executes the update locally and sends the resulting state to a few random recipients. II: Thread 1 of node 2 receives an update. When its local mini-batch update is ready, it will use the external buffer to correct its local update and then follow I. III: Shows a potential data race: two external updates might overlap in the external buffer of thread $H-1$ of node 2. Resolving data races is discussed in section 4.4.

Algorithm 3 SimuParallelSGD with samples $X = \{x_0, \ldots, x_m\}$, iterations T, steps size ϵ, number of threads n and states w

Require: $\epsilon > 0, n > 1$
1: **define** $H = \lfloor \frac{m}{n} \rfloor$
2: randomly **partition** X, giving H samples to each node
3: **for all** $i \in \{1, \ldots, n\}$ **parallel do**
4: randomly **shuffle** samples on node i
5: **init** $w_0^i = 0$
6: **for all** $t = 0 \ldots T$ **do**
7: get the tth sample on the ith node and compute
8: **update** $w_{t+1}^i \leftarrow w_t^i - \epsilon \Delta_t (w_t^i)$
9: **aggregate** $v = \frac{1}{n} \sum_{i=1}^{n} w_t^i$
10: **return** v

2.4 Mini-Batch SGD

The mini-batch modification introduced by [17] tries to unite the advantages of online SGD with the stability of BATCH methods. It follows the SGD scheme, but instead of updating after each single data sample, it aggregates several samples into a small batch. This mini batch is then used to perform the online update. It can be implemented as shown in algorithm 4.

Algorithm 4 Mini-Batch SGD with samples $X = \{x_0, \ldots, x_m\}$, iterations T, steps size ϵ, number of threads n and mini-batch size b

Require: $\epsilon > 0$
1: **for all** $t = 0 \ldots T$ **do**
2: **draw** mini-batch $M \leftarrow b$ samples from X
3: **Init**$\boldsymbol{\Delta} w_t = 0$
4: **for all** $x \in M$ **do**
5: **aggregate update** $\boldsymbol{\Delta} w \leftarrow \partial_w x_j(w_t)$
6: **update** $w_{t+1} \leftarrow w_t - \epsilon \boldsymbol{\Delta} w_t$
7: **return** w_T

3. ASYNCHRONOUS COMMUNICATION

Figure 3 shows the basic principle of the asynchronous communication model compared to the more commonly applied two-sided synchronous message passing scheme. An

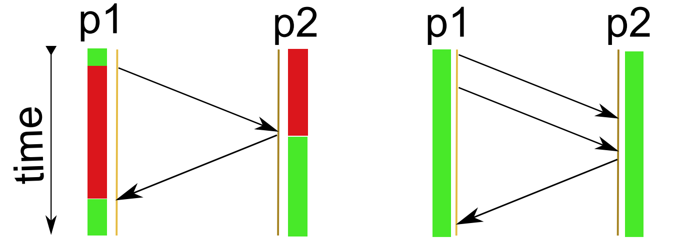

Figure 3: Single-sided asynchronous communication model (right) compared to a typical synchronous model (left). The red areas indicate dependency locks of the processes $p1, p2$, waiting for data or acknowledgements. The asynchronous model is lock-free, but comes at the price that processes never know if and when and in what order messages reach the receiver. Hence, a process can only be informed about past states of a remote computation, never about the current status.

overview of the properties, theoretical implications and pitfalls of parallelization by asynchronous communication can be found in [18]. For the scope of this paper, we rely on the fact that single-sided communication can be used to design lock-free parallel algorithms. This can be achieved by design patterns propagating an early communication of data into work-queues of remote processes. Keeping these busy at all times. If successful, communication virtually becomes "free" in terms of latency.

However, adopting sequential algorithms to fit such a pattern is far from trivial. This is mostly because of the inevitable loss of information on the current state of the sender

and receiver.

3.1 GASPI Specification

The Global Address Space Programming Interface (GASPI) [8] uses one-sided RDMA driven communication with remote completion to provide a scalable, flexible and failure tolerant parallelization framework. GASPI favors an asynchronous communication model over two-sided bulk communication schemes. The open-source implementation GPI 2.0[1] provides a C++ interface of the GASPI specification.
Benchmarks for various applications[2] show that the GASPI communication schemes can outperform MPI based implementations [7] [11] for many applications.

4. THE ASGD ALGORITHM

The concept of the ASGD algorithm, as described in section 1.2 and figure 2 is formalized and implemented on the basis of the SGD parallelization presented in [20]. In fact, the asynchronous communication is just added to the existing approach. This is based on the assumption that communication (if performed correctly) can only improve the gradient descent - especially when it is "free". If the communication interval is set to infinity, ASGD will become SimuParallelSGD.

Parameters

ASGD takes several parameters which can have a strong influence on the convergence speed and quality (see experiments for details on the impact):
T defines the size of the data partition for each thread, ϵ sets the gradient step size (which needs to be fixed following the theoretic constraints shown in [20]), **b** sets the size of the mini-batch aggregation, and **I** gives the number of SGD iterations for each thread. Practically, this also equals the number of data points touched by each thread.

Initialization

The initialization step is straight forward and analog to SimuParallelSGD [20] : the data is split into working packages of size T and distributed to the worker threads. A control thread generates initial, problem dependent values for w_0 and communicates w_0 to all workers. From that point on, all workers run independently, following the asynchronous communication scheme shown in figure 2.
It should be noted, that w_0 also could be initialized with the preliminary results of a previously early terminated optimization run.

Updating

The online gradient descent update step is the key leverage point of the ASGD algorithm. The local state w_t^i of thread i at iteration t is updated by an externally modified step $\overline{\Delta_t(w_{t+1}^i)}$, which not only depends on the local $\Delta_t(w_{t+1}^i)$ but also on a possible communicated state $w_{t'}^j$ from an unknown iteration t' at some random thread j:

$$\overline{\Delta_t(w_{t+1}^i)} = w_t^i - \frac{1}{2}\left(w_t^i + w_{t'}^j\right) + \Delta_t(w_{t+1}^i) \qquad (2)$$

For the usage of N external buffers per thread, we generalize equation (2) to:

$$\overline{\Delta_t(w_{t+1}^i)} = w_t^i - \tfrac{1}{|N|+1}\left(\sum_{n=1}^N \left(w_{t'}^n\right) + w_t^i\right) + \Delta_t(w_{t+1}^i),$$

$$\text{where } |N| := \sum_{n=0}^N \lambda(w_{t'}^n), \quad \lambda(w_{t'}^n) = \begin{cases} 1 & \text{if } \|w_{t'}^n\|_2 > 0 \\ 0 & \text{otherwise} \end{cases} \qquad (3)$$

with N incoming messages. Figure 4 gives a schematic overview of the update process.

4.1 Parzen-Window Optimization

As discussed in 1.2 and shown in figure 2, the asynchronous communication scheme is prone to cause data races and other conditions during the update. Hence, we introduce a Parzen-window like function $\delta(i,j)$ to avoid "bad" update conditions. The data races are discussed in section 4.4.

$$\delta(i,j) := \begin{cases} 1 & \text{if } \|(w_t^i - \epsilon\Delta w_t^i) - w_{t'}^j\|^2 < \|w_t^i - w_{t'}^j\|^2 \\ 0 & \text{otherwise} \end{cases}, \qquad (4)$$

We consider an update to be "bad", if the external state $w_{t'}^j$ would direct the update away from the projected solution, rather than towards it. Figure 4 shows the evaluation of $\delta(i,j)$, which is then plugged into the update functions of ASGD in order to exclude undesirable external states from the computation. Hence, equation (2) turns into

$$\overline{\Delta_t(w_{t+1}^i)} = \left[w_t^i - \frac{1}{2}\left(w_t^i + w_{t'}^j\right)\right]\delta(i,j) + \Delta_t(w_{t+1}^i) \qquad (5)$$

and equation (3) to

$$\begin{aligned} \overline{\Delta_t(w_{t+1}^i)} = \quad & w_t^i - 1/\left(\sum_{n=1}^N \left(\delta(i,n)\right) + 1\right) \\ & \cdot \left(\sum_{n=1}^N \left(\delta(i,n) w_{t'}^n\right) + w_t^i\right) \\ & + \Delta_t(w_{t+1}^i) \end{aligned} \qquad (6)$$

Computational Costs

Obviously, the evaluation of $\delta(i,j)$ comes at some computational cost. Since $\delta(i,j)$ has to be evaluated for each received message, the "free" communication is actually not so free after all. However, the costs are very low and can be reduced to the computation of the distance between two states, which can be achieved linearly in the dimensionality of the parameter-space of w and the mini-batch size: $O(\frac{1}{b}|w|)$. Experiments in section 5 show that the impact of the communication costs are neglectable.
In practice, the communication frequency $\frac{1}{b}$ is mostly constrained by the network bandwidth between the compute nodes, which is briefly discussed in section 4.5.

4.2 Mini-Batch Extension

We further alter the update of our ASGD by extending it with the mini-batch approach introduced in section 2.4. The motivation for this is twofold: first, we would like to benefit from the advantages of mini-batch updates shown in [17]. Also, the sparse nature of the asynchronous communication forces us to accumulate updates anyway. Otherwise, the external states could only affect single SGD iteration steps. Because the communication frequency is practically bound by the node interconnection bandwidth, the size of the mini-batch b is used to control the impact of external states.
We write $\mathbf{\Delta}_M$ in order to differentiate mini-batch steps from

[1]Download available at http://www.gpi-site.com/gpi2/
[2]Further benchmarks available at http://www.gpi-site.com/gpi2/benchmarks/

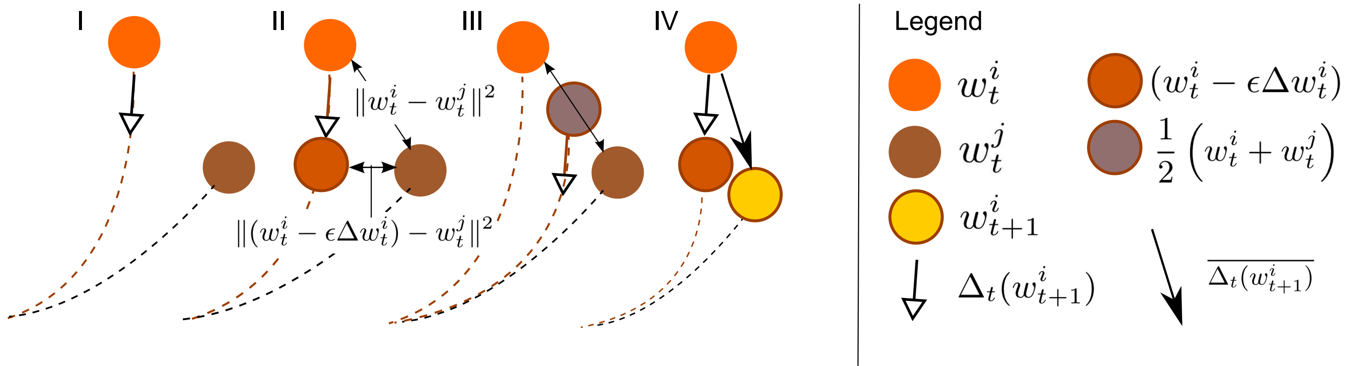

Figure 4: ASGD updating. This figure visualizes the update algorithm of a process with state w_t^i, its local mini-batch update $\Delta_t(w_{t+1}^i)$ and received external state w_t^j for a simplified 1-dimensional optimization problem. The dotted lines indicate a projection of the expected descent path to an (local) optimum. I: Initial setting: $\mathbf{\Delta}_M(w_{t+1}^i)$ is computed and w_t^j is in the external buffer. II: Parzen-window masking of w_t^j. Only if the condition of equation (4) is met, w_t^j will contribute to the local update. III: Computing $\overline{\mathbf{\Delta}_M(w_{t+1}^i)}$. IV: Updating $w_{t+1}^i \leftarrow w_t^i - \epsilon\overline{\mathbf{\Delta}_M(w_{t+1}^i)}$.

single sample steps Δ_t of sample x_t:

$$\overline{\mathbf{\Delta}_M(w_{t+1}^i)} = \left[w_t^i - \frac{1}{2}\left(w_t^i + w_t^j\right)\right]\delta(i,j) + \mathbf{\Delta}_M(w_{t+1}^i) \quad (7)$$

Note, that the step size ϵ is not independent of b and should be adjusted accordingly.

4.3 The final ASGD Update Algorithm

Reassembling our extension into SGD, we yield the final ASGD algorithm. With mini-batch size b, number of iterations T and learning rate ϵ the update can be implemented like this: At termination, all nodes $w^i, i \in \{1, \dots, n\}$ hold

Algorithm 5 ASGD ($X = \{x_0, \dots, x_m\}, T, \epsilon, w_0, b$)

Require: $\epsilon > 0, n > 1$
1: **define** $H = \lfloor \frac{m}{n} \rfloor$
2: randomly **partition** X, giving H samples to each node
3: **for all** $i \in \{1, \dots, n\}$ **parallel do**
4: randomly **shuffle** samples on node i
5: **init** $w_0^i = 0$
6: **for all** $t = 0 \dots T$ **do**
7: **draw** mini-batch $M \leftarrow b$ samples from X
8: **update** $w_{t+1}^i \leftarrow w_t^i - \epsilon\overline{\mathbf{\Delta}_M(w_{t+1}^i)}$
9: **send** w_{t+1}^i to random node $\neq i$
10: **return** w_I^1

small local variations of the global result. As shown in algorithm 5, one can simply return one of these local models (namely w_I^1) as global result.

4.4 Data races and sparsity

Potential data races during the asynchronous external update come in two forms: First, the complete negligence of an update state w^j because it has been completely overwritten by a second state w^h. Since ASGD communication is de-facto optional, a lost message might slow down the convergence by a margin, but is completely harmless otherwise. The second case is more complicated: a partially overwritten message, i.e. w^i reads an update from w^j while this is overwritten by the update from w^h.

We address this data race issue based on the findings in [16]. There, it has been shown that the error which is induced by such data races during an SGD update is linearly bound in the number of conflicting variables and tends to underestimate the gradient projection. [16] also showed that for sparse problems, where the probability of conflicts is reduced, data race errors are negligible. For non sparse problems, [16] showed that sparsity can be induced by partial updating. We apply this approach to ASGD updates, leaving the choice of the partitioning to the application, e.g. for K-Means we partition along the individual cluster centers of the states. Additionally, the asynchronous communication model causes further sparsity in time, as processes read external updates with shifted delays. This further decreases the probability of races.

4.5 Communication load balancing

We previously discussed that the choice of the communication frequency $\frac{1}{b}$ has a significant impact on the convergence speed. Theoretically, more communication should be beneficial. However, due to the limited bandwidth, the practical limit is expected to be far from $b = 1$.

The choice of an optimal b strongly depends on the data (in terms of dimensionality) and the computing environment: interconnection bandwidth, number of nodes, CPUs per node, NUMA layout and so on. Hence, b is a parameter which needs to be determined experimentally.

For most of the experiments shown in section 5, we found $500 \leq b \leq 2000$ to be quite stable.

5. EXPERIMENTS

We evaluate the performance of our proposed method in terms of convergence speed, scalability and error rates of the learning objective function using the K-Means Clustering algorithm. The motivation to choose this algorithm for evaluation is twofold: First, K-Means is probably one of the simplest machine learning algorithms known in the literature (refer to [9] for a comprehensive overview). This leaves little room for algorithmic optimization other than the choice of

the numerical optimization method. Second, it is also one of the most popular[3] unsupervised learning algorithms with a wide range of applications and a large practical impact.

5.1 K-Means Clustering

K-Means is an unsupervised learning algorithm, which tries to find the underlying cluster structure of an unlabeled vectorized dataset. Given a set of m n-dimensional points $X = \{x_i\}, i = 1, \ldots, m$, which is to be clustered into a set of k clusters, $w = \{w_k\}, k = 1, \ldots, k$. The K-Means algorithm finds a partition such that the squared error between the empirical mean of a cluster and the points in the cluster is minimized.
It should be noted, that finding the global minimum of the squared error over all k clusters $E(w)$ is proven to be a NP-HARD problem [9]. Hence, all optimization methods investigated in this paper are only approximations of an optimal solution. However, it has been shown [12], that K-Means finds local optima which are very likely to be in close proximity to the global minimum if the assumed structure of k clusters is actually present in the given data.

Gradient Descent Optimization

Following the notation given in [3], K-Means is formalized as minimization problem of the quantization error $E(w)$:

$$E(w) = \sum_i \frac{1}{2}(x_i - w_{s_i(w)})^2, \tag{8}$$

where $w = \{w_k\}$ is the target set of k prototypes for given m examples $\{x_i\}$ and $s_i(w)$ returns the index of the closest prototype to the sample x_i. The gradient descent of the quantization error $E(w)$ is then derived as $\Delta(w) = \frac{\partial E(w)}{\partial w}$. For the usage with the previously defined gradient descent algorithms, this can be reformulated to the following update functions with step size ϵ. Algorithms 1 and 5 use a batch update scheme. Where the size $m' = m$ for the original BATCH algorithm and $m' << m$ for our ASGD:

$$\Delta(w_k) = \frac{1}{m'} \sum_i \begin{cases} x_i - w_k & \text{if } k = s_i(w) \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

The SGD (algorithm 3) uses an online update:

$$\Delta(w_k) = \begin{cases} x_i - w_k & \text{if } k = s_i(w) \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Implementation

We applied all three previously introduced gradient descent methods to K-Means clustering: the batch optimization with MapReduce [5] (algorithm 1), the parallel SGD [20] (algorithm 3) and our proposed ASGD (see algorithm 5) method. We used the C++ interface of GPI 2.0 [8] and the C++11 standard library threads for local parallelization.
To assure a fair comparison, all methods share the same data IO and distribution methods, as well as an optimized MapReduce method, which uses a tree structured communication model to avoid transmission bottlenecks.

5.2 Cluster Setup

The experiments were conducted on a Linux cluster with a BeeGFS[4] parallel file system. Each compute node is equipped with dual Intel Xeon E5-2670, totaling to 16 CPUs per node, 32 GB RAM and interconnected with FDR Infiniband.
If not noted otherwise, we used a standard of 64 nodes to compute the experimental results (which totals to 1024 CPUs).

5.3 Data

We use two different types of datasets for the experimental evaluation and comparison of the three investigated algorithms: a synthetically generated collection of datasets and data from an image classification application.

Synthetic Data Sets

The need to use synthetic datasets for evaluation arises from several rather profound reasons: (I) the optimal solution is usually unknown for real data, (II) only a few very large datasets are publicly available, and, (III) we even need a collection of datasets with varying parameters such as dimensionality n, size m and number of clusters k in order to evaluate the scalability.
The generation of the data follows a simple heuristic: given n, m and k we randomly sample k cluster centers and then randomly draw m samples. Each sample is randomly drawn from a distribution which is uniquely generated for the individual centers. Possible cluster overlaps are controlled by additional minimum cluster distance and cluster variance parameters. The detailed properties of the datasets are given in the context of the experiments.

Image Classification

Image classification is a common task in the field of computer vision. Roughly speaking, the goal is to automatically detect and classify the content of images into a given set of categories like persons, cars, airplanes, bikes, furniture and so on. A common approach is to extract low level image features and then to generate a "Codebook" of universal image parts, the so-called Bag of Features [15]. Objects are then described as statistical model of these parts. The key step towards the generation of the "Codebook" is a clustering of the image feature space.
In our case, large numbers of $d = 128$ dimensional HOG features [19] were extracted from a collection of images and clustered to form "Codebooks" with $k = 100, \ldots, 1000$ entries.

5.4 Evaluation

Due to the non-deterministic nature of stochastic methods and the fact that the investigated K-Means algorithms might get stuck in local minima, we apply a 10-fold evaluation of all experiments. If not noted otherwise, plots show the mean results. Since the variance is usually magnitudes lower than the plotted scale, we neglect the display of variance bars in the plots for the sake of readability. If needed, we report significant differences in the variance statistics separately. To simplify the notation, we will denote the SimuParallelSGD [20] algorithm by SGD, the MapReduce baseline method [5] by BATCH and our algorithm by ASGD. For better comparability, we give the number of iterations I as global sum over all samples that have been touched by the respective

[3]The original paper [10] has been cited several thousand times.

[4]see www.beegfs.com for details

algorithm. Hence, $I_{BATCH} := T \cdot |X|$, $I_{SGD} := T \cdot |CPUs|$ and $I_{ASGD} := T \cdot b \cdot |CPUs|$.
Given runtimes are computed for optimization only, neglecting the initial data transfer to the nodes, which is the same for all methods. Errors reported for the synthetic datasets are computed as follows: We use the "ground-truth" cluster centers from the data generation step to measure their distance to the centers returned by the investigated algorithms. It is obvious that this measure has no absolute value. It is only useful to compare the relative differences in the convergence of the algorithms. Also, it can not be expected that a method will be able to reach a zero error result. This is simply because there is no absolute truth for overlapping clusters which can be obtained from the generation process without actually solving the exact NP-HARD clustering problem. Hence, the "ground-truth" is most likely also biased in some way.

5.5 Experimental Results

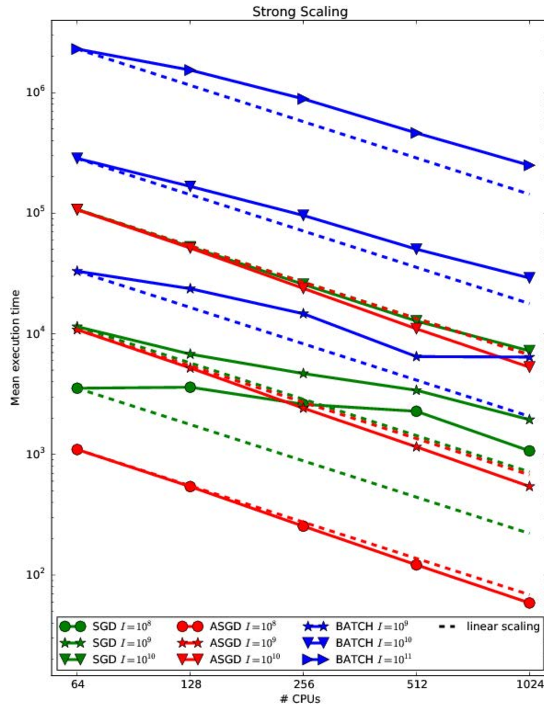

Figure 5: Results of a strong scaling experiment on the synthetic dataset with k=10, d=10 and ∼1TB data samples for different numbers of iterations I. The related error rates are shown in figure 9.

Scaling

We evaluate the runtime and scaling properties of our proposed algorithm in a series of experiments on synthetic and real datasets (see section 5.3). First, we test a strong scaling scenario, where the size of the input data (in k, d and number of samples) and the global number of iterations are constant for each experiment, while the number of CPUs is increased. Independent of the number of iterations and CPUs, ASGD

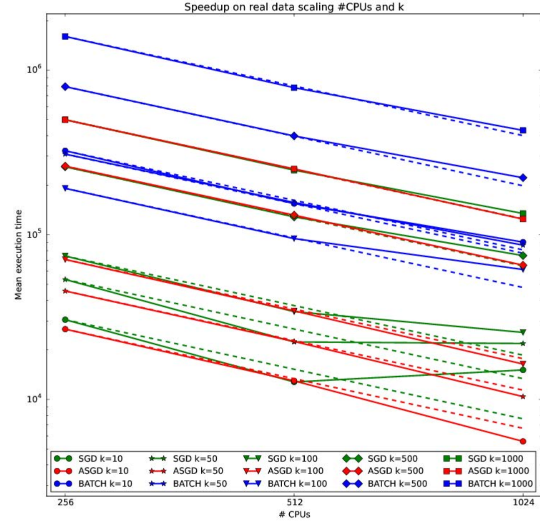

Figure 6: Strong scaling of real data. Results for with $I = 10^{10}$ and $k = 10..1000$ on the image classification dataset.

is always the fastest method, both for synthetic (see figures 6, 9) and real data (figure 6). Notably, it shows (slightly) better than linear scaling properties. The SGD and BATCH methods suffer from a communication overhead which drives them well beyond linear scaling (which is projected by the dotted lines in the graphs). For SGD, this effect is dominant for smaller numbers of iterations[5] and softens proportionally with the increasing number of iterations. This is due to the fact that the communication cost is independent of the number of iterations.
The second experiment investigates scaling in the number of target clusters k, given constant I, d, number of CPUs and data size. Figure 7 shows that all methods scale better than $O(\log k)$. While ASGD is faster than the other methods, its scaling properties are slightly worse. This is due to the fact that the necessary sparseness of the asynchronous updates (see section 4.4) is increasing with k.

Convergence Speed

Convergence (in terms of iterations and time) is an important factor in large scale machine learning, where the early termination properties of algorithms have a huge practical impact. Figure 8 shows the superior convergence properties of ASGD. While it finally converges to similar error rates, it reaches a fixed error rate with less iterations than SGD or

[5]Note: as shown in figure 9, a smaller number of iterations is actually sufficient to solve the given problem.

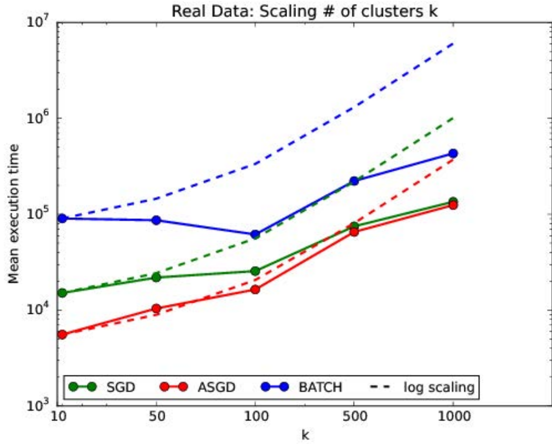

Figure 7: Scaling the number of clusters k on real data. Results for the same experiment as in figure 6. Note: here, the dotted lines project a logarithmic scaling of the runtime in the number of clusters.

BATCH. As shown in figure 8, this early convergence property can result in speedups up to one order of magnitude.

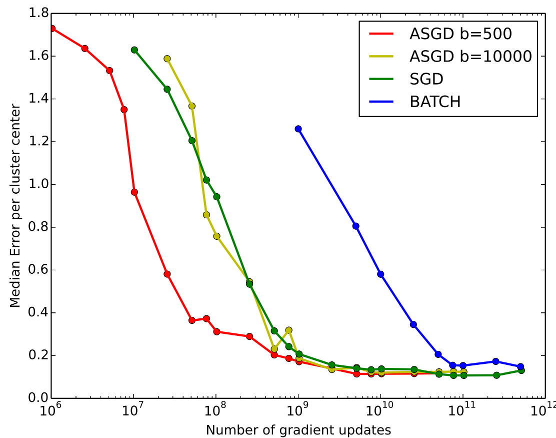

Figure 8: Convergence speed of different gradient descent methods used to solve K-Means clustering with $k = 100$ and $b = 500, 10000$ on a 10-dimensional target space parallelized over 1024 CPUs on a cluster. Given the right communication frequency, our novel ASGD method outperforms communication free SGD [20] and MapReduce based BATCH [5] optimization by the order of magnitudes. For low communication rates, ASGD converges towards [20].

Optimization Error after Convergence

The optimization error after full convergence[6] for the strong scaling experiment (see section 5.5) is shown in figure 9. While ASGD outperforms BATCH, it has no significant difference in the mean error rates compared to SGD. However, figure 9 also shows, that it tends to be more stable in terms of the variance of the non-deterministic K-Means results.

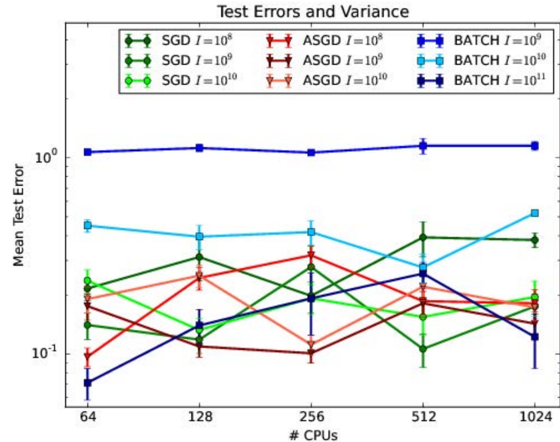

Figure 9: Error rates and their variance of the strong scaling experiment on synthetic data shown in figure 5.

Communication Frequency

Theoretically, more communication should lead to better results of the ASGD algorithm, as long as the node interconnection provides enough bandwidth. Figure 10 shows this effect: as long as the bandwidth suits the the communication load, the overhead of an ASGD update is marginal compared to the SGD baseline. However, the overhead increases to over 30% when the bandwidth is exceeded. As indicated by the date in figure 10, we chose $b = 500$ for all of our experiments. However, as noted in section 4.5, an optimal choice for b has to be found for each hardware configuration separately. The number of messages exchanged during the strong scaling experiments is shown in figure 11. While the number of messages sent by each CPU stays close to constant, the number of received messages is decreasing. Notably, the impact on the asynchronous communication remains stable, because the number of "good" messages is also close to constant. Figure 8 shows the impact of the communication frequencies of $\frac{1}{b}$ on the convergence properties of ASGD. If the frequency is set to lower values, the convergence moves towards the original SimuParallelSGD behavior.

Impact of the Asynchronous Communication

ASGD differs in two major aspects from SimuParallelSGD: asynchronous communication and mini-batch updates. In order to verify that the single-sided communication is the dominating factor of ASGD's properties, we simply turned

[6]Full convergence is here defined as the state where the error rate is not improving after several iterations.

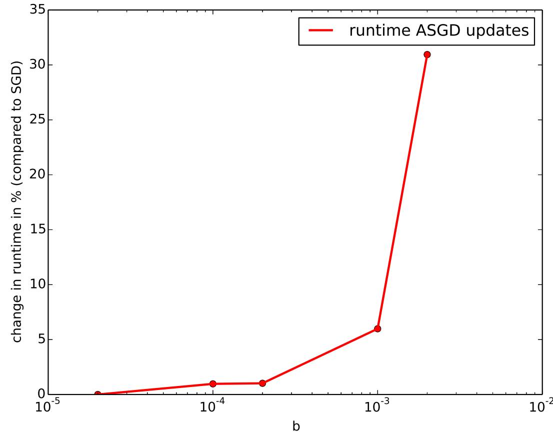

Figure 10: Communication cost of ASGD. The cost of higher communication frequencies $\frac{1}{b}$ in ASGD updates compared to communication free SGD updates.

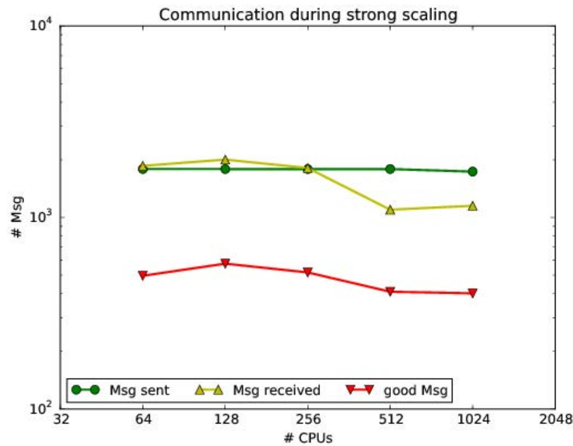

Figure 11: Asynchronous communication rates during strong scaling experiment (see figure 5). This figure shows the average number of messages sent or received by a single CPU over all iterations. "Good" messages are defined as those, which were selected by the Parzen-window function, contributing to the local update.

off the communication (silent mode) during the optimization. Figures 12 and 13 show, that our communication model is indeed driving the early convergence feature, both in terms of iterations and time needed to reach a given error level.

6. CONCLUSIONS

We presented a novel approach towards an effective parallelization of stochastic gradient descent optimization on distributed memory systems. Our experiments show, that the asynchronous communication scheme can be applied successfully to SGD optimizations of machine learning algorithms, providing superior scalability and convergence compared to previous methods.
Especially the early convergence property of ASGD should be of high practical value to many applications in large scale machine learning.

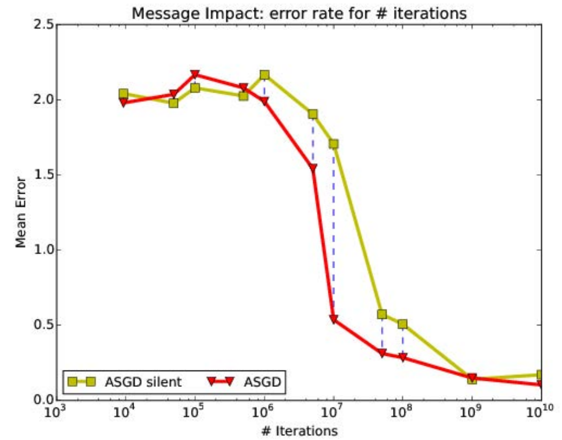

Figure 12: Convergence speed of ASGD optimization (synthetic dataset, $k = 10, d = 10$) with and without asynchronous communication (silent).

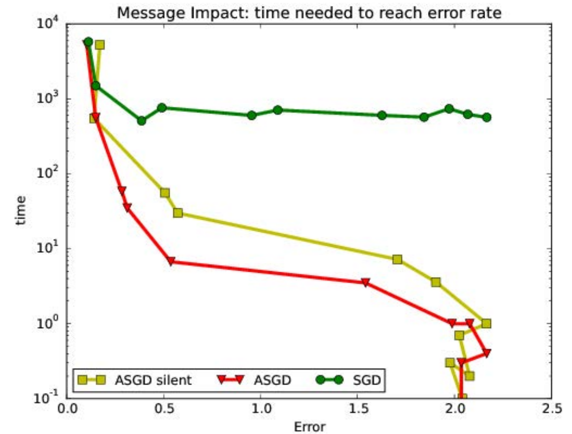

Figure 13: Early convergence properties of ASGD without communication (silent) compared to ASGD and SGD.

7. REFERENCES

[1] K. Bhaduri, K. Das, K. Liu, and H. Kargupta. Distributed data mining bibliography. In *http://www.csee.umbc.edu/ hillol/DDMBIB/*.

[2] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[3] L. Bottou and Y. Bengio. Convergence properties of the k-means algorithms. In *Advances in Neural Information Processing Systems 7,[NIPS Conference, Denver, Colorado, USA, 1994]*, pages 585–592, 1994.

[4] L. Bottou and O. Bousquet. The tradeoffs of

large-scale learning. In *Neural Information Processing Systems 20*, pages 161–168. MIT Press, 2008.

[5] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

[6] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[7] D. Grunewald. Bqcd with gpi: A case study. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 388–394. IEEE, 2012.

[8] D. Grünewald and C. Simmendinger. The gaspi api specification and its implementation gpi 2.0. In *7th International Conference on PGAS Programming Models*, volume 243, 2013.

[9] A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[10] S. Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[11] R. Machado, C. Lojewski, S. Abreu, and F.-J. Pfreundt. Unbalanced tree search on a manycore system using the gpi programming model. *Computer Science-Research and Development*, 26(3-4):229–236, 2011.

[12] M. Meila. The uniqueness of a good optimum for k-means. In *Proc. 23rd Internat. Conf. Machine Learning*, pages 625–632, 2006.

[13] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 265–272, 2011.

[14] C. Noel and S. Osindero. Dogwild!–distributed hogwild for cpu & gpu. 2014.

[15] E. Nowak, F. Jurie, and B. Triggs. Sampling strategies for bag-of-features image classification. In *Computer Vision–ECCV 2006*, pages 490–503. Springer, 2006.

[16] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[17] D. Sculley. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pages 1177–1178. ACM, 2010.

[18] H. Shan, B. Austin, N. J. Wright, E. Strohmaier, J. Shalf, and K. Yelick. Accelerating applications at scale using one-sided communication.

[19] Q. Zhu, M.-C. Yeh, K.-T. Cheng, and S. Avidan. Fast human detection using a cascade of histograms of oriented gradients. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 1491–1498. IEEE, 2006.

[20] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 2595–2603, 2010.