High Performance Parallel Stochastic Gradient Descent in Shared Memory

Scott Sallinen[1], Nadathur Satish[2], Mikhail Smelyanskiy[2], Samantika S. Sury[3], Christopher Ré[4]
[1]University of British Columbia; [2]Parallel Computing Lab, [3]Intel Corporation; [4]Stanford University
scotts@ece.ubc.ca, {nadathur.rajagopalan.satish, mikhail.smelyanskiy,
samantika.subramaniam}@intel.com, chrismre@stanford.edu

***Abstract*— Stochastic Gradient Descent (SGD) is a popular optimization method used to train a variety of machine learning models. Most of SGD work to-date has concentrated on improving its statistical efficiency, in terms of rate of convergence to the optimal solution. At the same time, as parallelism of modern CPUs continues to increase through progressively higher core counts, it is imperative to understand the parallel hardware efficiency of SGD, which often comes at odds with its statistical efficiency.**

In this paper, we explore several modern parallelization methods of SGD on a shared memory system, in the context of sparse and convex optimization problems. Specifically, we develop optimized parallel implementations of several SGD algorithms, and show that their parallel efficiency is severely limited by inter-core communication. We propose a new, scalable, communication-avoiding, many-core friendly implementation of SGD, called HogBatch; which exposes parallelism on several levels, minimizes the impact on statistical efficiency, and, as a result significantly outperforms the other methods. On a variety of datasets, HogBatch demonstrates near linear scalability on a system with 14 cores, as well as delivers up to a 20X speedup over previous methods.

I. Introduction

Rapid expansion of applications of machine learning to Big Data in the last decade has resulted in very large datasets, with millions to billions of examples and features. This in turn has given rise to very large optimization problems. For such problems, traditional approaches such as Interior-Point Method or Newton Method, which rely on basic linear algebra routines such as Cholesky, LU or DGEMM, are prohibitive in computational cost. Furthermore, these problems typically do not require solutions to be computed with very high accuracy.

To address these problems, modern machine learning methods employ Stochastic Gradient Decent (SGD) as a method of choice to train machine learning models. Machine learning models are trained using a training data set: this data can be thought of as a $n \times d$ matrix U. Each row (or sample) u in U corresponds to one instance of the training set containing d features (the dimensionality of the data set). The goal of the optimization method is then to find a d-dimensional vector w which minimizes a certain objective function, f, also known as loss function.

While there exists different types of loss functions that arise from different classifiers (e.g., linear, SVM, DNN, etc.), this paper focuses on the widely used binary *Logistic Regression* loss, which arises from classifying the elements of a given set into two groups, and has the form:

$$\sum_{u \in U} \log(1 + exp(-y_u p_u)) \quad (1)$$

where p_u is the dot product of the model w and the matrix sample u, and y_u is the corresponding label.

SGD solves this problem iteratively; each iteration looks at a subset of the training set, computes the gradient of the objective function of the model with respect to the subset of data, and makes an update to the model in the negative direction of the gradient.

SGD, however, is inherently sequential with a dependency across iterations: the model being a descendant of the previous model. This dependency limits parallelism. Several proposed algorithms, such as Hogwild [1], process multiple samples in parallel, neglecting this dependency, in order to offer parallelism. However, extra parallelism in these methods come at the loss of what can be defined as *statistical* and *hardware* efficiency. Specifically, loss of statistical efficiency implies an increased number of iterations (via passes over the data) to converge, while, loss of hardware efficiency implies poor cache reuse and increased amount of inter-core communication.

In this paper we explore design trade-offs for SGD which directly impact statistical and hardware efficiency and propose a new algorithm, based on this study, which addresses inherent weaknesses of existing parallel implementations of SGD. To this end, we make the following contributions, organized as follows:

- We provide a study of the option space for SGD in terms of data pass strategies, choice of privatization and aggregation versus direct shared model updates, and sparsity handling. This is discussed in section III.
- We examine the trade-off between staleness of data (which affects statistical efficiency in the form of convergence) with parallel execution (hardware efficiency), and focus on the metric of overall time-to-convergence to a given loss bound. The properties of staleness are analyzed in section V.
- We propose a new data pass style, HogBatch, that combines the benefits of Hogwild with a Mini-Batch SGD approach. We showcase up to a near linear scaling on multiple cores, with up to a 20x parallel speedup on a CPU with 14 cores (28 threads), even in sparse problems where previous parallelization techniques failed to improve on serial performance. HogBatch can take advantage of the sparsity of data sets, and is the method

of choice for datasets with widely varying sparsity. This method is detailed in sections IV and VI.

For the rest of the paper, we provide a complete evaluation of each method on real world datasets, including comparing performance with parallel scaling, in section VII. We conclude with a comparison to the state of the art framework BidMach [2], offering at least a 6.5x improvement for single model Logistic Regression, in section IX.

II. The Many Faces of SGD

The baseline Stochastic Gradient Descent (SGD) approach can be modified in a variety of ways. First, the exact gradient that needs to be computed depends on the loss function (linear, logistic, hinge loss, least squares, etc). Second, the data access and parallelization strategy can vary, where different approaches such as Hogwild [1] and Mini-Batching perform model updates using varying number of data samples at different times during a pass over the dataset. Third, the choice of how to compute the gradient and fix learning rates can also vary, and in fact techniques such as ADAGRAD [3] can help remove learning rate as a knob required to be tuned. One can also adopt extensions to SGD such as Stochastic Average Gradient (SAG) [4] that uses an average gradient of the dataset based on previous iterations to do model updates.

In this paper, we focus on the analysis and improvement of the parallelization of these algorithms in fundamental ways that affect the hardware efficiency. For instance, while the choice of loss function changes the computation, the access patterns and model updates are very similar. Further, learning rate and adaptive gradient strategies such as ADAGRAD and/or SAG can be added on to existing baseline SGD algorithms, without much impact on the actual algorithm structure, or overall application pattern.

Due to these factors, we focus on improving a relatively simple model for most of the results in this paper: the classical SGD algorithm as used in Logistic Regression. Our novel technique proposed in this paper focuses on reducing the staleness of updates to the model, as well as techniques to help increase hardware efficiency. We believe that our results are representative of other use cases of SGD; in order to help show this, we performed many of our evaluations with the ADAGRAD learning rate as an example, and found comparable efficiency improvements with our approach. We present some of these results in Section VIII.

III. Parallelizing SGD

Semantically, SGD is an inherently sequential algorithm. The update (where $Gradient$ is a function that computes the gradient of the dataset at row (also called sample) u)

$$w_{t+1} = w_t - a_t \; Gradient(w_t, u) \tag{2}$$

has a chain dependency, such that the next w update (t+1) is directly dependent on the previous (t). At first sight, since w is typically a vector of size d, the number of features, there appears to be data parallelism across the features. However, since SGD is typically performed on sparse problems, the sparsity pattern of the input sample row u dictates the number and position of the update to w, with different samples offering different and often small amounts of parallelism. Parallelizing across non-zero features also means that elements of w are written randomly in parallel, which can cause significant inter-core traffic to maintain coherence on cache-coherent architectures. For many sparse datasets, these factors can be quite limiting to scaling.

Algorithm 1: Mini-Batch SGD pseudocode for one datapass.

```
1  for (st = 0; st < num_samples/SIZE; st += SIZE) {
2    #pragma omp parallel for schedule(dynamic)
3    for (index = st; index < SIZE; index++) {
4      // Sparse vector operation.
5      g_tid[TID] += a * Gradient(model, index);
6    }  // (implicit thread barrier)
7
8    #pragma omp parallel for schedule(static)
9    for (f = 0; f < num_features; f++) {
10     for (t = 0; t < NUM_THREADS; t++)
11       model[f] = model[f] - g_tid[t][f];
12   }  // (implicit thread barrier)
13 }
```

When parallelizing across samples, an update may be based on a model w that is not as up to date as it could be. We quantify this as the *staleness*, or number of updates to the global model that happen (by potentially the same or different threads) between the time the model is *read* by a thread to the time that the model update is *written back* by that thread. The golden standard is serial SGD, wherein each update is directly a descendant of the previous and thus has no staleness. These properties will be further discussed in section V.

Interestingly, for sparse datasets if two samples, a followed by b, have non-intersecting non-zero indices, they will then update independent indices of w. Such updates are actually independent; when done in parallel, this maintains sequential convergence properties. This allows an update which is based on a stale model to become less drastic, since the effective indices may be less stale.

A. Mini-Batching

Mini-Batching is a strategy to enable parallelization across samples during SGD updates. A batch of a small number of samples from the data set is taken and a combined gradient is formed for all the samples together. A single update is then performed to the model for the batch. For a batch size of S and a starting sample u, the update follows the equation:

$$w_{t+S} = w_t - a_t \sum_{b=u}^{u+S} Gradient(w_t, b) \tag{3}$$

Our notation above for w_{t+S} indicates that mini-batching internally performs multiple gradient accumulations (denoted by the summation), but only performs a single update at the end of S data items. Note that this breaks the sequential semantics of Equation 2 – each gradient in the batch $b \in S$ is computed based on the gradient at w_t above, while it would have been computed on the basis of w_{t+b} in Equation 2. This has consequences for the staleness of updates, i.e. the model being used to compute the gradient can be quite old depending on S. This affects the *statistical efficiency* of the update and hence convergence. However, it does enable parallelism across the samples of the batch: threads can independently update a private gradient vector for their section of the batch. Upon completion of the batch, threads will work together to apply a reduction and update from all threads into the shared w vector, the model. The parallel algorithm in pseudocode is presented in Algorithm 1 (the vectors are not shown for simplicity).

Algorithm 2: Hogwild SGD pseudocode for one datapass.

```
#pragma omp parallel for schedule(dynamic)
for (index = 0; index < num_samples; index++) {
  // Sparse vector operation.
  model = model - a * Gradient(model, index);
}
```

By parallelizing across samples, Mini-Batching induces two important positive characteristics:

+ **One update per batch size:** There is a reduction in the total amount of work done – the number of updates (writes) to the model w per data pass is now the number of samples divided by the batch size.
+ **Thread independent tasks:** Since threads have their own subset of samples to execute on, they do not need to communicate until the reduction step. Moreover, it is easy to privatize the partial gradient vectors and have one vector per thread. Since these vectors can be accessed irregularly for sparse problems, it is highly advantageous to keep this irregular access to a thread-private structure and avoid cross-core traffic on cache-coherent architectures.

However, Mini-Batching suffers from several drawbacks:

- **Reduction:** As mentioned, all threads need to combine their partial solutions to compute the update to the model vector. This reduction step is added work.
- **Thread Synchronization:** The method has to do two thread synchronizations per batch: the first after all threads complete their local updates to the model, and the second after the reduction step.
- **Updates are stale:** Since the model update is not applied until the end of the batch, updates within the batch become increasingly stale – being based on an out of date model – such that the last update aggregated within the batch is stale by the batch size.

B. Hogwild

Hogwild [1], one of the most popular methods for parallelizing SGD, is a data pass approach for SGD that offers the interesting characteristic that threads do not have to synchronize, and in fact can perform their own asynchronous model updates. In Hogwild, as show in algorithm 2, each thread takes a sample at a time and performs an update to the global model w without any synchronization with other threads. These updates from different threads can potentially overwrite each other, leading to data race conditions. Hogwild works well for very sparse data sets, where many samples are actually near-independent since they write to mostly different indices of the model (e.g. the intersect of non-zeros is generally small).

Hogwild provides two important characteristics:

+ **Thread asynchronicity:** Threads perform independent work and do not have to synchronize. There is no need for any reduction of updates across the threads.
+ **Minimum staleness:** The computation of the gradient at any time is based on the current state of the model visible to the thread at that time. Since each thread directly performs updates to this shared model vector, the values read are only as old as the communication latency between threads, which is quite low as compared to Mini-Batch SGD.

Algorithm 3: HogBatch SGD pseudocode for one datapass.

```
#pragma omp parallel for schedule(dynamic)
for (st = 0; st < num_samples/SIZE; st += SIZE) {
  for (index = st; index < SIZE; index++) {
    // Sparse vector operation.
    g_tid[TID] += a * Gradient(model, index);
  }

  for (f = 0; f < num_features; f++)
    model[f] = model[f] - g_tid[TID][f];
}
```

Hogwild suffers from two problems:

- **Race Conditions:** In this algorithm, updates to the model are somewhat chaotic due to the lock-free design: a thread can be updating the value at a particular index in the model while another thread may be reading/writing from the same location. It is quite possible that parts of the update can be lost, however, the chance of conflict gets smaller as the problem gets more sparse.
- **Inter-core communication:** Although there is no direct communication across threads, all threads issue both read and write requests to a single, common model vector. On cache-coherent architectures, this can cause "ping-ponging" of the vector. Such a phenomenon happens when multiple threads store private copies of the same cache line, and updates from any thread to the cache line invalidates all other copies of that line – the relevant cores may then need to pull in the updated cache line, leading to high cross-core traffic.
  In fact, this problem is quite severe even if the data sets are sparse - there can be false cache line sharing cases where invalidation occurs even if different threads update different indices that happen to fall in the same cache line (typically 64 bytes on x86 architectures). This problem become extreme for small model vectors, dense problems, or when parallelizing to a large amount of threads. In our experiments, core-to-core communication alone could consume up to 60% of the cycles during execution of Hogwild.

IV. Hogwild + Mini-Batching: HogBatching

A previously unexplored topic is the use of both strategies – Hogwild and Mini-Batching together – in order to apply an update. In this work, we propose to combine these approaches as follows: instead of threads working synchronously together within a batch, we can have each Hogwild thread work on their own full batch of samples. In this method, which we call *HogBatching*, a thread would be responsible for handling a batch of samples, continuously aggregating them into a private gradient vector. Once a thread completes its batch, it will apply the update to the shared model vector and move on to its next batch without synchronization between other threads. This method is shown in algorithm 3.

Analyzing this new method, we note it actually takes the positive aspects from each of the previous methods:

+ **Thread asynchronicity:** As in Hogwild, threads have no need for direct inter thread communication, and do not need to synchronize. Threads perform their assigned batches independently, and there is no reduction of updates.

+ **Reduced staleness:** There is some staleness within the batch as in regular Mini-Batching, but other threads may asynchronously update the model vector in the middle of a thread's batch processing. In that case, the thread would get a more current model in the middle of processing a batch. This does not happen in regular Mini-Batch SGD.
+ **One update per batch size:** As in Mini-Batching, the update frequency per pass is again reduced by a factor of the batch size.
+ **Thread independent tasks:** Also as in Mini-Batching, threads have their own independent subset of samples to process. In addition, their work is stored in thread private gradient vectors, which does not need to be shared with other cores and thus is cache friendly.

Weaknesses: Although we addressed the negatives of reduction and synchronization, two other weaknesses inherited from Hogwild and mini-batching need to be addressed: first, the potential for asynchronicity conflicts, and second, inter-core communication. These issues still remain, but are drastically reduced: since threads aggregate model updates to a local model vector and only write out the update once per batch, the potential for conflicts is highly reduced. For the same reason, most accesses to the global vector are now reads – which results in less false sharing and inter-core communication.

When a write is issued on completion of a batch, the write is dense (similar to Mini-Batch SGD), instead of sparse as in Hogwild (many sparse updates have already been aggregated into one dense update during batch processing). Although this of course invalidates the cache, the new cache line contains many updated values instead of potentially only one. Hence most of the cache line would actually be required by the destination core (true sharing as opposed to false sharing), reducing wasteful traffic.

Finally, the issue of staleness within the thread's batch, still remains. However, we address this issue in the next section.

V. Staleness Properties

In table I, we show the staleness factors for each method. In this table, we consider the minimum staleness for the final sample in the batch, and the maximum staleness (which also will be for the final sample in the batch). For asynchronous methods, the minimum would come when a thread applies their update and no other thread updates the model before the next sample is processed. In the case of Hogwild, a sample could be totally up to date (staleness of 0) in the best case, or in the worst case could be out of date by a factor of the number of threads. The worst case scenario for the staleness of an update for Hogwild is that each other thread applies their update to the model in the time between the model is read and the time the model will be written. *Note we simplify (to show relativity between strategies) in the table that processing a single update takes about the same amount of time, and hence we will not encounter scenarios where some threads make multiple updates in the same time as others make a single update.*

For Mini-Batching, the minimum and maximum are the same - the final sample in the batch will always be out of date by the size of the batch, since the model vector is not updated until the batch completes, as all local thread updates are aggregated and written back only at that point.

TABLE I: Staleness Analysis. The number of threads is T, the size of the Mini-Batch is S, and the size of the HogBatch is HS.

Method	Min-Stale (For final update in batch)	Max-Stale	Example: T=8, S=1024, HS=(S/T) [min, max]
Hogwild	0	(T-1)	[0, 7]
Mini-Batch	S	S	[1024, 1024]
HogBatch	HS	(T*HS)	[128, 1024]

HogBatching has some similar properties to each of the previous approaches – although the final sample in the batch could be out of date by the batch size, all other threads could write their updates in the meantime, causing the sample to actually be fairly up to date in the global view. In the worst case, it is similar to regular batching, since the update may be completely out of date due to the asynchronicity.

A. Improving Staleness

There is a unique way to improve the staleness of both batch style algorithms. We note that in line 5 of Mini-Batching (Algorithm 1) and line 5 of HogBatching (Algorithm 3), we calculate the gradient based on a read only version of the model. It is important to keep the model read only during this step, since we aggregate the update and only apply it once per batch. However, note that in lines 11 and 9 respectfully, we simply directly subtract the aggregated results from the model vector. So, before we compute the gradient, we can actually locally subtract g_tid[TID] from the model, and calculate the gradient with this instead – in effect, this is analogous to the thread continuously updating it's local view of the model vector with their own samples, but without actually committing the write back to shared memory.

This improves the staleness within the batch by a factor of the work that each thread does within the batch – although it will not see other thread updates, it will utilize its own local ones. In table II, we show the effect of this. For Mini-Batching, since each thread is responsible for an equal portion of the batch size, the staleness is reduced by that quantity. Although visually this does not seem like much of a difference, the more stale a update is, the more useless that the update is – an update that is only stale by 1 is much more useful than an update stale by 100, and so on. In effect, this modification gets rid of the "tail end", or worst staleness, causing a significant improvement.

In HogBatching, this improvement is magnified – since each thread's batch is locally improved, the cross thread updates seen are also improved, causing the best case scenario of an update to be very recent. Note that having a completely up-to-date model update is possible, although extremely unlikely, due to the definition of asynchronicity. It would require that all threads less one complete an update, and then pause, in which case the final thread is actually performing the equivalent of sequential SGD. Fortunately, the worst case is also equally unlikely, since it requires all other threads to complete their update but but similarly pause before committing it, while the final thread continues.

TABLE II: Improved Staleness Analysis. The number of threads is T, the size of the Mini-Batch is S, and the size of the HogBatch is HS.

Method	Min-Stale (For final update in batch)	Max-Stale	Example: T=8, S=1024, HS=(S/T) [min, max]
Hogwild	0	(T-1)	[0, 7]
Mini-Batch	(S) - (S/T)	(S) - (S/T)	[896, 896]
HogBatch	(HS) - (HS) = 0	(T*HS) - (HS)	[0, 896]

Overall, we find this enhancement to experimentally give up to 30% improvement in time to convergence, especially becoming important for denser problems. In addition, this tweak also showcases a nice algorithmic relationship between Hogwild and HogBatching: In both cases, when T = 1, the algorithms simply perform sequential SGD. In addition, when the batch size HS = 1, HogBatching is equivalent to Hogwild.

VI. Implementation Options with HogBatching

A. Batch Size

One important parameter for HogBatching is the batch size, for statistical efficiency. In tables I and II, we provide an example batch size HS that is equal to the Mini-Batch size divided by the number of threads – this is the basis for the worst case staleness for HogBatching being the same as for regular Mini-Batching. This makes intuitive sense: the number of samples "in flight", or being processed during one logical super-step, becomes equivalent to Mini-Batching if each thread is working on S/T samples.

However, this is not necessarily the best choice for batch size. Since the minimum staleness for HogBatching is much smaller than Mini-Batching, the improved convergence allows for a larger batch size. In practice, we find the optimal batch size can lie anywhere between S/T to the full size S depending on the sparsity of the data set and the value of T.

B. Improvements with Sparsity

For all algorithms (Hogwild, Mini-Batching and HogBatching), a model update is computed based on an aggregation of a few rows of the sparse input data set (as few as one for Hogwild, and equal to the batch size otherwise). Depending on the batch size, it is possible that these aggregates (sums) of the sparse input rows will still be sparse (the number of non-zero indices in the sum of sparse vectors is the size of the set union of the non-zero indexes of each vector). This sparsity in the model update can be exploited to make model updates faster.

However, in order to allow for sparse model updates, we need to keep track of the position of the non-zero indices when aggregating the sparse rows of the input. We tested two different methods of keeping track of non-zero indices: (a) holding a bitmap of indices written during the batch, updating it as each entry is aggregated; followed by performing a bit scan to get non-zero indices; and (b) using a map data structure to accumulating the indices that have been written. It is important to note, however, that both these techniques have overheads; indeed if the density of the aggregated output is large enough, the gains from doing a sparse update may not be enough to justify these overheads. We experimented with these various strategies and found that the extra work of keeping track of non-zero indices was only useful for extremely sparse problems (we specifically only found improvements for the two sparsest of the seven datasets we consider in this work).

C. Hierarchical Parallelism

When enabling SMT threads on the CPU, we realized an opportunity to further improve performance by using a hierarchical parallelism structure. With two threads per core, we have an opportunity to exploit the cache locality between the threads. We begin by noting that in algorithm 3, there are actually two levels of available parallelism – an "outer" parallelism level where different batches are processed asynchronously, and an "inner" level where a thread would perform updates on their local batch, currently as in a serial SGD algorithm.

For runs with SMT, we set the two SMT threads per core to index the same private gradient vector, as it will reside in the same private cache. This reduced the total number of private gradient vectors by half, and furthermore reduced cache pressure, as the effective inter-core communication was reduced as well.

Algorithm 4 shows a generalization of this algorithm, beyond just SMT. We first start as in HogBatching, allowing outer asynchronous parallelism, but have a *group* of threads (rather than a single thread) be responsible for a batch. The work within the group's batch is performed as a small Hogwild problem by the group's *workers*, who index the same group-private gradient vector. Once a group's workers complete their assigned samples, the workers within the group apply the update to the shared model vector. In this algorithm, synchronization occurs only between the workers within the group (and in fact is not necessarily required), and only when the group's workers work together to apply the single group update. In addition, the only shared memory spanning groups is the model, whereas each group would have a private memory location (shared among workers) for the group gradient.

This abstraction to groups and workers is intended to aid the effectiveness past SMT multi-threading and into many core architectures, such as the Intel® Xeon Phi™ coprocessor[1].

VII. Experimental Analysis

A. Experimental Setup

Unless otherwise mentioned, the following was used for the evaluations:

Hardware: We use a single socket Intel® Xeon® E5-2697 v3 Haswell based CPU, with 14 cores (28 threads including Simultaneous Multi-Threading/SMT) running at 2.6 GHz. Our machine has 64 GB RAM and runs Red Hat Enterprise Linux Server release 6.5.

Software: We use custom end-to-end code written in C++ with OpenMP, and compiled with the Intel® C++ Compiler 15.0.2 with all optimizations enabled. We format in single

[1]Intel, Xeon and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

Algorithm 4: Many-Core HogBatch SGD pseudocode, for one datapass.

```
GROUP_START = get_group_start(TID);
// Group parallel asynchronous
for (st = GROUP_START; st < GROUP_COUNT; st++) {
  WORK_START = get_worker_start(st, TID);
  // Worker parallel asynchronous
  for (id = WORK_START; id < WORK_COUNT; id++) {
    // Sparse vector operation.
    g_group += a * Gradient((model + g_group), id);
  }
  sync();  // Sync threads in the group.

  // All threads in the group perform the update.
  for (f = TID; f < num_features; f += T_PER_GROUP)
    model[f] = model[f] - g_group[f];
}
```

TABLE III: Experiment Datasets. The RCV1 datasets are from [5], the others are from the LibSVM resources [6].

Dataset Name	Examples	Features	NNZ	Sparse%	NNZ/Row	Avg/Row
news20.binary	19,996	1,355,191	9,097,916	0.034	1 to 16,423	454.987
RCV1-v2	781,265	276,544	60,534,218	0.028	4 to 1,585	77.482
RCV1-v1-test	677,399	47,236	49,556,258	0.155	4 to 1,224	73.157
real-sim	72,309	20,958	3,709,083	0.245	1 to 3,484	51.295
w8a	64,700	300	753,862	3.884	1 to 114	11.652
connect4	67,557	126	2,837,394	33.333	42 to 42	42.000
covtype	581,012	54	6,940,438	22.121	9 to 12	11.945

precision values (the matrix as well as the labels), although we have also received similar results with double precision. We use the classic SGD update algorithm for Logistic Regression unless otherwise specified.

Datasets: We perform evaluations on a wide range of datasets with varying feature size (which is equivalent to model size), as well as sparsity patterns. The datasets used and their properties are shown in Table III.

Parameters: For each method, we do a combinatorial sweep of the parameters of alpha (learning rate), and batch size, and only present the best result. In addition, when using all cores, we present the better result from enabling or disabling SMT multi-threading, unless presented separately. We adjust the learning rate per iteration for a sample index i to be $alpha/sqrt(i + D)$, where D is the iteration sum from the previous data-passes: $datapass\_number * num\_samples$.

Regularization: We apply L2 regularization for all methods, with the Lambda value set to $1/num\_samples$ of the dataset. We use a sparse optimization to regularization, as described in [7].

Reporting: When reporting time, we discount the time for loading data from disk. We measure the training time of each method until it achieves a chosen *Closeness to "Optimal" Loss* – unless otherwise specified, this is 99.5%. We compute the "optimal" loss using L-BFGS [8], which is a second order method and thus can eventually arrive at a model with a machine precision gradient of zero for our convex problem. We then evaluate the *Closeness to Optimal Loss* as the percentage that the current loss is of this "optimal" loss value, with the function $(2 - |current|/|optimal|) * 100\%$, since the loss decreases to approach the optimal. We do not present the time it takes for L-BFGS, since the goal of this paper is not to compare first order methods to second order ones.

When comparing the performance of different algorithms, we chose time-to-convergence, in terms of closeness to optimal loss, as the standard; this is because each algorithm has its own trade-off between *statistical efficiency*, the number of passes over data for convergence (usually reported in the literature), and *hardware efficiency*, time per datapass. In reality, the effective time to convergence is actually a mixture (product) of these two factors. When possible, the points on figures represent one complete datapass, to help show the difference in time per datapass.

TABLE IV: Speedup (as time to 99.5% convergence) of HogBatch over best alternative solution out of Serial, Mini-Batching, or Hogwild on a 14 core system.

Dataset	Sparse%	Features	Best Alt	vs Best Alt
news20.binary	0.034	1,355,191	Hogwild	0.86x
RCV1-v2	0.028	276,544	Hogwild	1.87x
RCV1-test	0.155	47,236	Hogwild	2.43x
real-sim	0.245	20,958	Hogwild	3.85x
w8a	3.884	300	Hogwild	8.97x
connect4	33.333	126	Mini-Batch	5.81x
covtype	22.121	54	Serial	20.16x

B. Results of our Evaluation

Table IV shows the speedup of HogBatch over the best performing alternative (Serial, Mini-Batching, or Hogwild). As expected, Hogwild was generally the best performing alternative, as it is usually faster than Mini-Batching for sparse data sets due to its asynchronous nature. However, with increasingly dense problems that have a small number of features, Hogwild becomes worse than serial. The figures in 1 show how closely each of these methods approach "optimal solution" overtime, using all 14 cores, on the RCV1-v1 and covtype datasets. We present these two in depth since they have widely different properties: RCV1 is highly sparse and has a relatively large model size, whereas covtype has a very small model size and is slightly dense. In our experiments, we found that other datasets had properties that fell between these extremes.

In RCV1 1a, we see that Hogwild showed a similar convergence behaviour to serial; in fact, it was equivalent in number of data-passes at 8. Mini-Batching *converged* much slower, as was expected, taking twice as many data-passes at 16 – however, it was about twice as fast in time-per-pass compared to Hogwild, leading to a near parity in overall time-to-convergence. HogBatching took the advantages of both – the low overall data-pass count (at 9), with about half in time-per-pass compared to Hogwild. A similar convergence behaviour was observed with our other datasets, with the time-per-pass being improved drastically in some cases: in covtype 1b, Hogwild and Mini-Batching when run with all cores actually took longer to complete a datapass than sequentially with one core, whereas HogBatching scaled near-linearly while maintaining excellent convergence.

On the other hand, table IV shows that HogBatching is not always the best strategy. In the news20 dataset, which has an extremely large number of features and very high sparsity, Hogwild slightly outperforms HogBatching, even when sparse optimizations to HogBatching are applied. The reason for this twofold. First, due to the large model size and sparsity, it is extremely unlikely that samples will suffer from the write conflict or false sharing. This reduces the amount of core-to-core cache communication. Second, with a batching strategy, there is the overhead of applying the local update

to the global model. Although we can limit this overhead by using sparse updates as in Section VI-B, Hogwild does not have this overhead and is hence slightly faster. However, in most other cases, the model size is not as extreme, and hence the false sharing and inter-core communication traffic makes HogBatching superior to Hogwild.

C. Scaling with Cores

Figure 2 shows the scaling of different strategies when we vary the number of cores across two of our datasets. The baseline of 1.0 is the performance of serial SGD.

These figures show several key aspects of each method of parallelization:

Hogwild: For RCV1 2a, this method gets worse before it gets better: when only two cores are used, the method is outperformed by only one core (which is equivalent to the serial method): the overhead of inter-core communication exceeds the benefits of splitting the work across cores. With more cores, the method begins to scale better, but at 14 cores the method is still just under 4x better than serial. In the denser covtype dataset 2b, the behaviour is worse, never breaking a 2x speedup over serial, and performing worse than serial when the entire machine is used.

Mini-Batching: In RCV1 2a, after a sweep of batch sizes, we found that the optimal was a relatively large batch size of 3700. For this dataset, there is sufficient parallelism and the method scales well. Note that the single thread performance of Mini-Batching is worse than serial SGD – this is due to the aggregation into a private vector before application to the model, which is not necessary for a single core solution. This creates an initial overhead; however, this overhead is constant, as it is parallelized when running on more cores. For the covtype dataset 2b, the overheads of the local aggregation and application is actually small due to the small model size, leading to single thread performance that is almost the same as serial SGD. However, the method does not scale well; similar to Hogwild, there is a drop in performance with additional cores. This is, however, for a different reason: with a small optimal batch size of 100, the cost of thread synchronization limited performance. Note that this speedup is in regards to the best time-to-convergence that we can get for Mini-Batching; we found larger batches would lead to overall slowdowns due to slower convergence.

HogBatching: This method had the best scaling in both datasets. In RCV1 2a, similar to Mini-Batching, the method has an overhead due to aggregation, leading to redundant work in the single core case. However, the method scales well with core count, with the performance at 14 cores being about 8X faster than serial SGD, improving to 11X with SMT multi-threading. This is over twice as fast as Hogwild. For the covtype dataset 2b, there is little overhead for single thread runs (as in Mini-Batching), and furthermore the method scales well even with small batch sizes due to lack of synchronization points. Overall, we achieve near-linear scaling, almost 14X with 14 cores, and some super-linear scaling effects due to the cache improvement with optimized SMT multi-threading (20X), as discussed in VI-C.

D. Scaling with Frequency

With the current trends of increasing hardware performance through adding multiple cores, improving core frequency is no longer a focus for architecture design. Hence algorithms that scale well with number of cores rather than with frequency are preferred for current and future hardware.

In Figure 3, we show the performance of each method as we vary number of cores from 1 to 14 and frequency between 1.3GHz and 2.6GHz. We colour pairs of entries with the same colour: one entry having twice the core count of the other while running at half the core frequency. For instance, 2 cores that run at 1.3 GHz are coloured the same as 1 core that runs at 2.6 GHz (Note: 8 to 14 is not exactly double, nor is 14 to adding SMT threads).

The figure shows that Hogwild scales much better with a frequency increase rather than with core count. This is because the frequency of the interconnect between cores is governed by core frequency, which was the key bottleneck in Hogwild. Therefore, a faster inter-core communication on fewer cores results in a overall performance gain, versus a larger amount of cores with a slower inter-core connect.

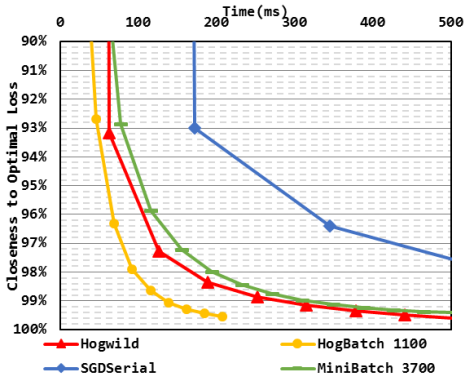
Mini-Batching and HogBatching however, show a different story: in both cases, adding more cores or increasing frequency near equivalently improve performance. This ability to scale to multiple cores is important for hardware efficiency, especially when moving from the multi-core into the many-core domain. However, of these techniques, HogBatch scales better than Mini-Batching.

E. Future Scalability

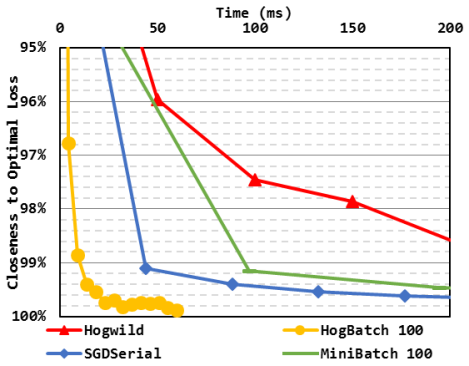
The reported real system measurements show poor hardware efficiency of Hogwild. While Mini-Batch improves hardware efficiency of Hogwild, it loses statistical efficiency with larger batch sizes, which is required to achieve good overall efficiency. HogBatch on the other hand, maintained good statistical and hardware efficiency up to 14 cores.

In order to understand how HogBatch performs on system with larger number of cores, we have simulated HogBatching using Sniper, an execution-driven simulator [9]. We model a single-threaded 2-wide in-order core at 1.8 GHz as well as 2-dimensional mesh, with 2 cores per mesh stop, a 2-cycle hop latency, a link bandwidth of 64bytes/s and MESIF cache coherence protocol. On the RCV1-v1 dataset, we observed 53x scalability in time-per-pass on 64 cores, and projected 90x scalability on 128 cores, indicating the excellent hardware efficiency of HogBatch at larger scale. The trace observed only a minimal loss in statistical convergence per pass compared with serial, at approximately a 25% loss, which was only slightly more than the loss we observe when running all threads on our 14 core machine with this dataset (about 10%). This result indicates HogBatch's ability to sustain high statistical efficiency as well.

This ability to achieve high hardware efficiency, while maintaining statistical efficiency, allows us to consider HogBatching to be the method of choice for future platforms. In addition, we anticipate hierarchical parallelism (as in section VI-C), to continue to improve performance on machines that contain many threads per core.

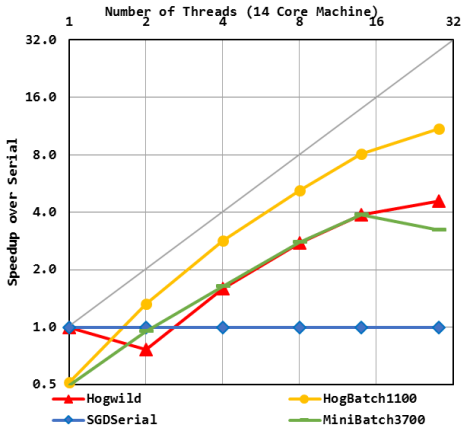


(a) RCV1-v1-test

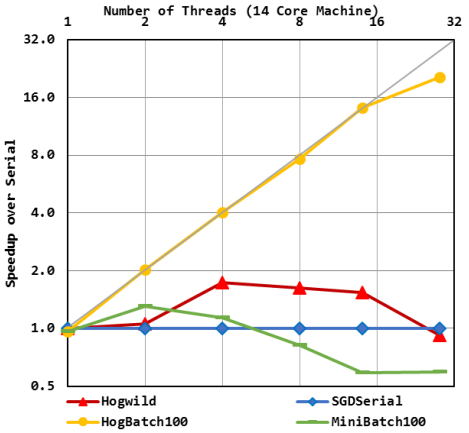


(b) covtype

Fig. 1: Closeness to the optimal solution over time, of each CPU method, using all cores. Each sample point represents a dataset pass. The batch size used follows the method name.



(a) RCV1-v1-test



(b) covtype

Fig. 2: Speedup over serial, in time-to-convergence, while varying the number of cores. SMT multi-threading is disabled up to 14 threads, and for the final result (28) SMT is enabled. The batch size used follows the method name.

		Cores
	SGDSerial	1
Frequency	1.3	4.499
	2.6	2.281

		Cores					
	Hogwild	1	2	4	8	14	14 + HT
Frequency	1.3	5.299	6.751	3.296	1.757	1.243	1.038
	2.6	2.595	2.975	1.436	0.829	0.590	0.500

		Cores					
	MiniBatch	1	2	4	8	14	14 + HT
Frequency	1.3	8.250	4.400	2.586	1.443	1.031	1.336
	2.6	4.608	2.403	1.399	0.820	0.586	0.707

		Cores					
	HogBatch	1	2	4	8	14	14 + HT
Frequency	1.3	8.137	3.257	1.577	0.798	0.533	0.370
	2.6	4.45	1.721	0.800	0.439	0.281	0.208

Fig. 3: Time (in seconds) to 99.5% optimum loss, across core count and variable frequency. Same coloured elements represent approximately a trade-off; a double of core count, or a double of frequency. **Dataset.** RCV1-v1-test.

VIII. Multi Model Regression

In addition to single model regression, which we have focused on up to now, regression can also be extended to produce multiple models from a single input sample dataset. This is useful when there is more than one prediction to be made from the same data, and the dataset has different labels for each prediction to be made. We now deal with a matrix of labels, instead of a vector; the model will similarly also then be a matrix.

Moving into the multi model domain has a few important effects: the first is that parallelism can be applied across the new dimension (being a matrix rather than a vector). Thus, the necessity for creating parallelism across *samples* diminishes, due to the new-found *model* parallelism.

The algorithm exposing this is shown in 5. Second, a strategy that employs batching with sample parallelism (which duplicates the size of the model per thread) is no longer cache friendly: the model state and labels, both of which are dense, are now quite large and may not fit into caches. Each update to a single index for such models now needs to update all the models which pulls in a lot of data and pollutes caches. Due to these reasons, we found Hogwild, which does not use batching, to be the best approach on all of our datasets.

Notably, when allocating the model matrix, it is critical to properly orient the data: the models should be allocated such that the first index for each model are stored consecutively in memory, followed by padding if necessary to align the data (e.g., 103 is padded to 128), and then the second index for

Algorithm 5: Multi-Model Hogwild SGD pseudocode for one datapass.

```
#pragma omp parallel for schedule(dynamic)
for (index = 0; index < num_samples; index++) {
  #pragma simd
  for (m = 0; m < NUM_MODELS; m++) {
    // Sparse indices update of model[m]
    model[m] -= a * Gradient(model[m], index);
} }
```

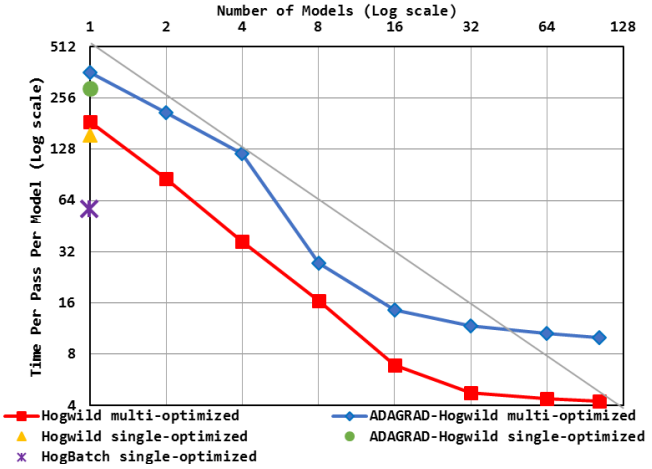


Fig. 4: Time per Pass per Model, scaling with number of models. Also shown at one model is single model optimized Hogwild and HogBatch (Note: Log-Time scale), as well as ADAGRAD scaling. **Dataset.** RCV1-v2.

all models, and so on. This allocation strategy is important: although Hogwild updates the model indices sparsely, it will apply updates to the same index of all models at once. If these are available in consecutive memory locations, cache line utilization can be nearly perfect. This layout also allows the updates to be done using SIMD (Single Instruction Multiple Data, using AVX2 Instruction Set, for instance) operations – the padding will allow these SIMD operations to operate from aligned memory addresses. Furthermore, padding also allows cache loads and stores to not be split – accesses to any index is then guaranteed to be in its own cache line.

Figure 4 shows the result of varying the number of models with RCV1-v2. We go up to 103 models for sake of comparisons with other work that we will describe in Section IX. We report time per datapass per model on the y-axis. Until we hit about 32 models, we get benefits our SIMD-friendly layout and the time per model continues to fall. At about 32 models, we have gotten the most out of the SIMD benefits, and increasing parallelism across this dimension is no longer beneficial.

At this crossover point, we could decide to separate the model training, either across nodes, or across sockets in a multi-socket system. For example, a four socket machine could train 32 models at once per socket, allowing for continued scalability with increasing model count.

IX. Comparison to State-of-the-Art

BidMach [2] is a general purpose machine learning framework that runs on multi-core CPUs as well as GPUs, and has implemented SGD for regression. The framework is open source, and has been shown to compare favourably to other implementations. Performance results for logistic regression performance using BidMach is reported in [10].

TABLE V: Single model comparison using RCV1-v1-test dataset. All methods use the ADAGRAD update, and the same parameters. Single socket CPU was used for all comparisons.

Implementation	Hardware	Time/Pass (ms)
BidMach	TITAN X	723
BidMach	Sandy Bridge	14,190
CPU optimized (Mini-Batch)	Sandy Bridge	289
CPU optimized (Hogwild)	Sandy Bridge	253
CPU optimized (HogBatching)	Sandy Bridge	147
CPU optimized (HogBatching)	Haswell	111

TABLE VI: Multi model comparison using RCV1-v1-test dataset. All methods use the ADAGRAD update, and the same parameters. BidMach uses a size of 5,000. We use one CPU socket for all results, except the last row where we use two. Each implementation trains all 103 models at once, the dual socket run splits the models to train separately on sockets. All CPU optimized runs use Hogwild.

Implementation	Hardware	Models	Time / Pass (ms)
BidMach	TITAN X	103	2,170
BidMach	Sandy Bridge	103	120,720
CPU optimized	Sandy Bridge	103	2,010
CPU optimized	Haswell	103	1,283
CPU optimized	2x Haswell	103	724

We execute the current version of BidMach (1.0.3) on an Intel® Xeon® E5-2680 Sandy Bridge based system with 8 cores at 2.7 GHz. This machine also hosts an NVIDIA Titan X that we use for BidMach runs. We set the parameters for BidMach SGD (e.g. learning rate, regularization) to be the same as those in our code, and we also implemented the ADAGRAD update [3] used in BidMach (we note that the ADAGRAD update is about 3x slower per datapass than the regular SGD update, as shown in figure 4, due to the use of extra state). Since we choose the same parameters, the convergence of our methods was almost identical and we hence only report performance in time per datapass.

In Table V, we compare the results of our single model performance with BidMach. Our single-model HogBatch code is 1-2 orders of magnitude better than the BigMach's CPU version, and is also significantly faster than the GPU results. We note here that (based on communication with the developers) BidMach is not specifically optimized for use in the single model case and is mainly targeted at GPU multi model regression.

Table VI shows the comparison with multi-model regression. The authors of BidMach have also confirmed that the framework's CPU code was not as well optimized as the GPU code (in fact, BidMach's CPU code is in Scala, whereas the GPU code is in native CUDA). With proper attention to vectorization as discussed in section VIII, we are able to be on par with the GPU implementation on one Sandy Bridge CPU, and are slightly faster on our Haswell machine. This result also shows the importance of using a well optimized baseline to compare CPU and GPU platforms.

X. Related work

Hogwild, by F. Niu et al. provides a strong foundation for this paper [1]. Hogwild presents an asynchronous, lock free approach to SGD. Hogwild has been extended in many directions, such as a dual averaging algorithm for non-smooth, non strongly-convex problems as described by J. Duchi et al. [11]. In a very recent paper [12], J. Duchi et al. has also provided related work showing evidence of the effectiveness of asynchronous stochastic optimization schemes, also discussing the need for understanding the underlying hardware – such as avoiding locking. In addition, their work has also independently introduced a related approach to what we describe as HogBatching, as they average updates over a constant 10 samples. However, they did not extend it into the full abstraction (ie. batch size, hierarchical parallelism, etc.) or evaluation that we offer in this paper.

M. Zinkevich et al. offers a distributed solution to parallelize SGD by splitting the workload and having each machine perform SGD on a subset of data before averaging together the results [13]. Similar to their design goal, a new distributed framework Splash by Y. Zhang et al. improves performance in the multi-node domain [14]. They are able to improve convergence with a partitioned dataset with a re-weighting of local data. Finally, Downpour SGD, shown by J. Dean et al., also attempt to improve SGD in distributed systems [15]. Each of these distributed methods work well to scale out for large datasets, as each machine does not need to hold the total amount of data – only a defined portion size. However, the machine that receives the partitioned SGD tasks could itself have a multi-core processor – at this point, a shared memory technique like ours actually becomes harmonious, where one can apply a shared memory solution to speed up the task at the multi-core level, and then perform the strategy to aggregate results at the multi-machine level.

In [16], C. De Sa et al. describe an analysis of Hogwild style method named Buckwild. In their method, one key design choice to improve hardware efficiency is to use lower precision arithmetic, as low as an 8 bit value instead of a 32 bit float. With their techniques, they improve wall-clock time of the solution over a regular Hogwild approach. We believe these same improvements could be applied to HogBatching, as both methods are essentially extensions to Hogwild, but along an orthogonal axis, and we hope to investigate this in the future.

Ce Zhang and C. Ré present DimmWitted, characterizing state of the art SGD on shared memory NUMA machines [17]. Their work is in a similar direction to ours, focusing on understanding the efficiency of SGD: the authors perform a study of the trade-off between statistical and hardware efficiency, which is the focus of this paper as well. However, in DimmWitted the authors focused on access patterns (e.g. matrix row access or column access) and data replication, while we look at the parallelization within datapass strategies, offering analysis of time to convergence.

XI. Conclusion

In this paper we developed several highly optimized implementations of SGD, and analyzed factors which contribute to their statistical and hardware efficiency. We identify inter-core communication as the key impediment to SGD scaling on modern multi-core shared-memory systems. We addressed this challenge with new parallel SGD algorithm, called HogBatch, which demonstrates up to near linear scalability across wide range of sparse datasets, without a noticeable impact on convergence, when compared to serial SGD. HogBatch achieves superior parallel efficiency by localizing a large fraction of updates into thread-private gradient vectors. Localizing and avoiding communication is imperative on modern and future systems, as inter-core communication latency continues to increase with a larger core counts. To this end, we also demonstrated the friendliness of this algorithm to future many-core platforms with large number of cores or threads.

As future work, we intend to explore the use of the HogBatching technique on other machine learning problems, such as stochastic coordinate descent algorithms, where similar trade-offs exist [18], [19], as well as collaborative filtering problems, and non-convex problems.

References

[1] B. Niu, Feng andRecht, C. Re, and S. Wright, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*, pp. 693–701, 2011.

[2] J. Canny and H. Zhao, "Big data analytics with small footprint: Squaring the cloud," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 95–103, ACM, 2013.

[3] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *The Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[4] M. Schmidt, N. L. Roux, and F. Bach, "Minimizing finite sums with the stochastic average gradient," *arXiv preprint arXiv:1309.2388*, 2013.

[5] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *The Journal of Machine Learning Research*, vol. 5, pp. 361–397, 2004.

[6] C.-C. Chang and C.-J. Lin, "Libsvm : a library for support vector machines.," *ACM Transactions on Intelligent Systems and Technology*, pp. 2:27:1–27:27, 2011.

[7] L. Bottou, "Stochastic gradient tricks," pp. 430–445, 2012.

[8] J. Nocedal, "Updating quasi-newton matrices with limited storage," *Mathematics of computation*, vol. 35, no. 151, pp. 773–782, 1980.

[9] W. Heirman, A. Isaev, and I. Hur, "Sniper: Simulation-based instruction-level statistics for optimizing software on future architectures,"

[10] "Bidmach benchmarks." https://github.com/BIDData/BIDMach/wiki/Benchmarks. Accessed: 2015-08-01.

[11] J. Duchi, M. I. Jordan, and B. McMahan, "Estimation, optimization, and parallelism when data is sparse," in *Advances in Neural Information Processing Systems*, pp. 2832–2840, 2013.

[12] J. C. Duchi, S. Chaturapruek, and C. Ré, "Asynchronous stochastic convex optimization," *ArXiv e-prints*, Aug. 2015.

[13] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, pp. 2595–2603, 2010.

[14] Y. Zhang and M. I. Jordan, "Splash: User-friendly programming interface for parallelizing stochastic algorithms," *arXiv preprint arXiv:1506.07552*, 2015.

[15] 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*, pp. 1223–1231, 2012.

[16] C. De Sa, C. Zhang, K. Olukotun, and C. Ré, "Taming the wild: A unified analysis of hogwild!-style algorithms," *arXiv preprint arXiv:1506.06438*, 2015.

[17] C. Zhang and C. Ré, "Dimmwitted: A study of main-memory statistical analytics," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1283–1294, 2014.

[18] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin, "Parallel coordinate descent for l1-regularized loss minimization," in *International Conference on Machine Learning (ICML)*, pp. 321–328, 2011.

[19] C. Scherrer, M. Halappanavar, A. Tewari, and D. Haglin, "Scaling up coordinate descent algorithms for large l1 regularization problems,"