Linear Models with Many Cores and CPUs: A Stochastic Atomic Update Scheme

Edward Raff
Laboratory for Physical Sciences
edraff@lps.umd.edu
Booz Allen Hamilton
raff_edward@bah.com

Jared Sylvester
Laboratory for Physical Sciences
jared@lps.umd.edu
Booz Allen Hamilton
sylvester_jared@bah.com

***Abstract*—Linear models are fast to train, apply, and still state of the art for sparse and high dimensional problems. Their computational efficiency makes them difficult to parallelize, with the standard multi-core approaches often diverging after more than 8 cores are added. We propose a Stochastic Atomic Update Scheme (SAUS) for training linear models on many core machines. It is simple to implement, reduces the number of divergent cases, and obtains greater speedups by being able to effectively use an 80-core server.**

I. Introduction

The focus of our work is to train linear models with multiple CPU cores. Given a dataset $\{x_1, \ldots, x_n\}$ of n datums, where each $x_i \in \mathbb{R}^D$ has an associated label y_i, we want to train a simple linear model with respect to some loss function $\ell(\cdot, \cdot)$ and regularization penalty $\Omega(\cdot)$. The general form for such linear models is given in Equation 1, where λ is the regularization term.

$$\arg\min_{w} \frac{1}{n} \sum_{i=1}^{n} \ell(w^T x_i, y_i) + \lambda \cdot \Omega(w) \tag{1}$$

This form covers a wide class of problems, such as logistic and linear regression, and useful regularizers such as L_1 and L_2. Such models are also effective and so considerable effort has been spent creating highly optimized algorithms and packages — such as the seminal LIBLINEAR [5]. But as hardware has progressed, single-core clock speeds are plateauing and new CPUs are instead being developed with more cores. Existing training algorithms were not optimized with such multi-core environments in mind.

Significant many-core resources are becoming available to a wider audience every year, and continuing to scale linear models to larger problems means we must develop efficient multi-core solvers. For example, services like Amazon AWS and Google's Compute Cloud make high end virtual machines available with up to 128 vCPUs.[1] Unfortunately, it is not easy to utilize such resources when seeking to train "bread-and-butter" linear models such as Logistic Regression. Parallelized implementations of such models are often algorithm specific, and require understanding the limits of individual methods and the particular hardware in use. Worse, the algorithms in use today can — counter-intuitively — be slowed down by the addition of more cores. The performance characteristics are often sensitive to the sparsity of the dataset, leading to performance regression and sometimes model divergence. These issues make practical use of these parallel methods difficult for non-expert users and impede adoption. We tackle all of these problems: increasing scalability, increasing reliability, and reducing execution time reversals.

A parallel linear learning method should have two primary goals: *Model Quality* (the algorithm produces the same final result, or one that performs nearly as well) and *Training Efficiency* (the algorithm should scale as well as possible with increased resources). At a minimum, training time when using P CPUs should be less than or equal to using $P-1$ CPUs. In this work we present a new stochastic method of sharing information between local and global weight vectors, which we term the Stochastic Atomic Update Scheme (SAUS). SAUS improves upon both model quality and training efficiency, especially when we consider the many-core scenario. Applying SAUS to the Stochastic Dual Coordinate Ascent (SDCA) algorithm, we show it is competitive with prior work when $P \leq 8$, and has superior accuracy and speed when $P > 8$. SAUS scales up to 80 cores without significant performance regressions, and it obtains solutions with the same accuracy as the serial SDCA algorithm. While these issues are not perfectly solved, we believe SAUS represents the first approach that can be easily used by practitioners on almost any system they would have access to today.

The rest of our paper is organized as follows. First we will review related work in parallel linear models in section II. We will introduce our contribution of a stochastic approach to information sharing between local and global weight vectors in section III, which underpins our new SAUS algorithm. We will show experimental results in section IV that show it obtains better speedup and efficiency as P increases. In section V we will then analyze SAUS's convergence properties under prior frameworks, and perform analysis to show why SAUS obtains superior speedups. Finally we will conclude in section VI.

II. Related Work

The most common method for producing parallel linear models is the "Wild" approach, where updates are done asynchronously without any locking. This was introduced in the now

[1]e.g., see https://aws.amazon.com/ec2/instance-types/ for Amazon's x1e.32xlarge instance

seminal "Hogwild!" paper, which first proposed and proved general convergence rates for stochastic gradient decent in this manner [13]. This asynchronous strategy has become the dominant method in producing parallel linear models for a single-machine system.

Many works have applied the Wild approach to specific algorithms to create specialized convergence proofs and implementations [7, 9, 16, 21]. In most of these a statement about the convergence rate as a function of the sparsity and update frequency is developed. These provide theoretical backing that also explains why models often diverge after many cores are added, or when the feature set is dense.

In our work, we will be producing a parallel version of the Stochastic Dual Coordinate Assent (SDCA) algorithm, which covers a wide set of possible linear models. Our new SAUS scheme does not require the SDCA algorithm. We choose SDCA because it does not require the specification of a learning rate η or of any decay rate. This makes it faster in practical use, as the parameter search can be restricted to simply the regularization penalty λ. Hsieh et al. [7] developed the PASSCoDe framework for parallel implementations of the SDCA algorithm [14]. In their work they use SDCA as the base algorithm for parallelization, exploring a Wild, Atomic, and Lock-based implementations. Using a single 10-core CPU, they found the Wild algorithm to provide the best overall efficiency. The Atomic approach, which performs atomic updates of the weight vector by each thread, varied between being marginally slower to significantly slower, depending on the sparsity of the dataset. Their work will be our primary comparison point as it represents the Wild approach applied specifically to the same SDCA algorithm we consider.

While the wild update scheme has found popularity, it has difficulty converging on multi-socket systems with multiple CPUs available, each of which may have multiple cores. Zhang and Hsieh [19] observed this on a dual-socket sever with the PASSCoDe approach, and developed PASSCoDe-fix to counteract this problem. This approach performs a convergence check and line search at the end of each iteration to ensure that the solution does not diverge, resulting in a "Semi-Asynchronous" algorithm. Updates are performed wildly during each epoch, synchronized, a line search is performed, and then the cycle repeats. While PASSCoDe-fix was effective in fixing the divergence issue, it further complicates the implementation which increases compute time. In addition, is was tested only up to 20 cores, and is specific to the SDCA algorithm. Since SAUS can be applied to SDCA, PASSCoDe-fix can be used in combination with SAUS, and so we will focus on the base SDCA solvers to which PASSCoDe-fix might be applied. Our SAUS approach, while applied in this paper to SDCA, is not algorithm specific. SAUS also reduces compute time, and is evaluated on a machine with four times as many cores.

Zhang et al. [20] proposed Hogwild++, which uses multiple local weight vectors with regularly shared updates to avoid performance regressions with SGD on multi-CPU systems. However, it needs a new hyper-parameter to be tuned (which depends on hardware and data) in addition to a step size η and decay rate γ. Tunning of these additional hyper-parameters consumes valuable CPU time, reducing the effective speedup in practice and making it a poor comparison choice for our work. Our approach instead uses a stochastic update policy and introduces no new tunable hyper-parameters, and is applicable to step-size free algorithms like SDCA so that the regularization term λ is the only hyper-parameter.

III. Stochastic Atomic Updates

We now introduce the design of our Stochastic Atomic Update Scheme (SAUS). The goal of SAUS is to improve scalability by reducing communication.

In the Atomic scheme, communication occurs via the Compare and Swap (CAS) hardware instruction. This instruction provides a safe way for a contenting thread to "swap" a known old value with a desired new value, returning a boolean indicating success. For each parameter that needs updating, the CAS is run in a loop until success, which results in $\mathcal{O}(P)$ communication cost for P competing threads.

The wild approach attempts to side-step this cost by performing unsafe (a.k.a, "wild") updates of the shared weight vector. But its performance still suffers from hidden communication costs. Within a single CPU, the MESIF protocol is used to ensure cache-coherence between cores transparently [15]. Between multiple CPUs, the QPI bus serves a similar purpose, but with a latency two-orders of magnitude slower [10]. Both of these processes run at the hardware level and cause communication overhead in the Wild approach, which is why it too has reduced scaling as the number of cores and CPUs increases.

Algorithm 1 Stochastic Atomic Update Scheme

```
1:  procedure Update(Thread context $p$, weight vector $\boldsymbol{w}$, update $\boldsymbol{z}$, Probability multiplier $\varrho$)
2:      rand $\sim \mathcal{U}(0,1)$
3:      $\boldsymbol{w}^p \leftarrow$ weight vector local to thread $p$.
4:      for each non-zero value $z_i \in \boldsymbol{z}$ do
5:          $w_i^p \leftarrow w_i^p + z_i$
6:          if rand $\leq \varrho/|P|$ then
7:              $w_i \leftarrow w_i + w_i^p$                    ▷ atomic update
8:              $w_i^p \leftarrow 0$
9:  procedure Accumulate(All thread contexts $P$)
10:     for $p \in [0, |P|-1]$ do                         ▷ in parallel
11:         for $i \in [pD/|P|, (p+1)D/|P|]$ do
12:             $w_i \leftarrow w_i + w_i^p$               ▷ non-atomic, but safe
```

To alleviate this communication overhead problem, we propose to stochastically update the weight vector. If we have P total threads performing work, we will give each thread a $1/P$ chance to do an Atomic update of the shared weight vector $\boldsymbol{w}$. If the thread fails this Bernoulli trial, we store the update into a *thread local* weight vector $\boldsymbol{w}^p$. This local vector will not generally be seen by other threads, and so updates to this vector can be done using standard "Wild" updates. Because no other thread will be accessing it, there are no thread safety issues. This local weight vector is a critical component of our SAUS approach. It allows each thread to maintain local information, which will be periodically shared with the other threads. Updating the shared weight vector $\boldsymbol{w}$ stochastically then reduces the communication overhead.

This sharing of information is done in two primary stages. We refer to the first as the *update stage*. When a thread wins the Bernoulli trial to perform an atomic update of the main weight vector $\boldsymbol{w}$, the update will be done using the accumulated values in $\boldsymbol{w}^p$ but only for the values that have just been updated locally. This means the stochastic update will not fully transfer all information that is contained in the local weight vector. This can be seen as a method that, after several trials, transfers information from frequently used features from the working threads to the global state.

Algorithm 2 Generic Learning Framework for SAUS

```
Require: Loss function $\ell(\cdot,\cdot)$
 1: $\boldsymbol{w} \leftarrow 0$, $\varrho \leftarrow 1$
 2: prevLoss $\leftarrow \infty$, lowestLoss $\leftarrow \infty$
 3: for each Epoch do
 4:     Thread safe accumulator epochLoss $\leftarrow 0$
        ▷ Below loop done in parallel with $P$ threads
 5:     for Datum $\boldsymbol{x}$ with label $y$ do
 6:         dot $\leftarrow (\boldsymbol{w} + \boldsymbol{w}^p)^T \boldsymbol{x}$
 7:         epochLoss $\leftarrow$ epochLoss $+ \ell(\text{dot}, y)$
 8:         Get gradient direction $\beta$
 9:         UPDATE($p$, $\boldsymbol{w}$, $\beta\boldsymbol{x}$, $\varrho$)
10:     ACCUMULATE($P$)
11:     if epochLoss $> 1.01 \cdot$ prevLoss then
12:         $\varrho \leftarrow \min(4 \cdot \varrho, |P|)$
13:     if epochLoss $<$ lowestLoss then
14:         $\varrho \leftarrow \max(\varrho - 1, 1)$
15:     lowestLoss $\leftarrow \min(\text{lowestLoss}, \text{epochLoss})$
16:     prevLoss $\leftarrow$ epochLoss
```

The second stage we refer to as the *accumulate stage*. It occurs at the end of each epoch, and runs through all local vectors $\boldsymbol{w}^p$, accumulating them into the shared weight vector $\boldsymbol{w}$. Doing so ensures the global vector is completely up-to-date at a regular frequency. This consolidation can be done without atomic updates by temporarily allowing each thread to access all local vectors $\boldsymbol{w}^{1\cdots P}$, and each thread reading a disjoint range of values to update into the shared $\boldsymbol{w}$. These accumulate and update steps are defined more explicitly in Algorithm 1.

We now describe how our SAUS approach can be applied to any generic learning process. The UPDATE procedure includes an argument for a probability multiplier ϱ. This is included to cover the rare case when the delay caused by stochastic updates is too large, and causes divergence. By increasing the probability ϱ based on the training loss, we cheaply detect and adjust for this situation. This in effect becomes an adaptive mechanism to ensure convergence. In the worst cases, $\varrho \Rightarrow P$, in which case we degrade back to fully-atomic updates. In any other case, we are still gaining a communication efficiency over the classic approach. We emphasize this adaptive scheme was only needed for one of our datasets to converge, but is used in all of our experiments.

The details of the ϱ updates and method is given in Algorithm 2, with ACCUMULATE being called at the end of each epoch, and UPDATE with each gradient. The only other addition of our approach is in computing the dot product between the weight vector $\boldsymbol{w}$ and the current feature vector $\boldsymbol{x}$. We instead have each thread include its own local weight vector $\boldsymbol{w}^p$ in a read-only fashion when computing dot products. Doing so ensures that each thread is locally correct with regard to all the gradient updates performed in that thread. Additional information is then communicated solely through $\boldsymbol{w}$, as done via the UPDATE and ACCUMULATE procedures.

A. *Special Bias Term Handing*

As the last part of our SAUS approach, we note an optional special-case handling for the bias term. We use this optional handling in all of our tests. A special case for the bias term is desirable as for most of our datasets, the bias term will be the only feature present for every datum x_i. This means the bias term b will be missing considerably more updates to its value than any other feature at any given time.

To remedy this situation, we can insist that the bias term be the only feature updated at every iteration. However, naively performing atomic updates of the bias term would cause increased contention.

Since our implementation is in Java (as detailed more in the next section), we make use of the DoubleAdder [8] to implement the bias term for our model. This choice was done primarily to improve convergence on the URL dataset, and can optionally be replaced with a standard atomic double variable. All of our results are shown with the use of the DoubleAdder for the SAUS algorithm. We do not use the DoubleAdder for the Atomic or Wild approaches because it is not a part of those algorithms, and would not be in their spirit (all atomic or wild updates, respectively).

The DoubleAdder reduces the contention for updating the bias term. It works by keeping an array of atomic doubles, the sum of which represents one double value of interest. When a thread reads the DoubleAdder's value, it atomically reads and sums all values in the array. This requires no contention. When writing updates, each thread indexes into a location in the array of atomic doubles based on its identity, and attempts to update the value atomically. When too many CAS operations fail (indicating contention) the array is doubled in size. This doubling occurs until contention stops, which is guaranteed once the size is greater than or equal to that of the number of processors P.

We emphasize that despite the bias term always being updated, it may not degrade to creating the full array of P double values. Each thread will be performing work with a variable number of non-zero values that they must store locally and have workloads that have inconsistent timing. This means while each thread will updated the bias b, not all threads will be updating b at the same time.

The DoubleAdder thus represents a trade off: it reduces update contention at a cost of increased memory usage, read time, and cache inefficiency. For this reason it is not practical to make all coefficients DoubleAdder objects, and our preliminary testing showed this to be slower than the standard single threaded implementation. Even if each DoubleAdder was under no contention, storing the coefficients as an array of DoubleAdders means the memory cost doubles compared to the array of atomic doubles due to each DoubleAdder getting the object header overhead for the object itself, as well as the array object each DoubleAdder contains. This in turn causes numerous cache misses due to memory reference indirections.

The use of the DoubleAdder for the bias term is a strategic choice, based on the fact that the bias term is always present and thus always updated. This makes the bias term necessarily the highest contention coefficient, and thus appropriate to make the trade-off of the DoubleAdder for that specific component.

IV. Evaluation and Experiments

To evaluate our new SAUS for parallel training of linear models, we will apply them to the SDCA algorithm for training Elastic-Net regularized logistic regression. We choose Logistic Regression because of its widespread use, and use Elastic-Net regularization because of its robustness to high-dimensional settings, a common case where logistic regression still provides state-of-the-art performance [11]. This means $\ell(s, y) = \log(1 + \exp(-y_i \cdot s))$ and $\Omega(w) = \frac{1}{4}||w||_2^2 + \frac{1}{2}||w||_1$. For all experiments we set $\lambda = n^{-1}$ for reproducibility and simplicity. In extended testing we have found that our method continues to work for various values of λ.

In our experiments we will look at results for eight different datasets detailed in Table I. While most prior work focuses on only two or three (sparse) corpora, we test our approach on a wide array of datasets. This includes small datasets (e.g. a9a) and dense (e.g. Epsilon) ones, as well as massive and sparse datasets like KDD 2012. Testing on this wide array helps build confidence that our approach is generally applicable and needs less fine-tuning, meaning it could be used by more novice users. All datasets were downloaded from the LIBSVM site [2]. Since not every dataset we evaluate has a standard testing set, we will use a random 90/10 split for training and testing. The same split will be used for each approach for every value of P tested.

TABLE I: Summary of datasets used in experiments, with the number of rows N, the number of features D, and the average number of non-zero features per row $\overline{D}$. The last column shows the average percent of non-zero values in the corpus.

Dataset	N	D	$\overline{D}$	density (%)
Epsilon	400,000	2,000	2,000.0	100.0
a9a	22,696	123	13.9	11.8
Webspam trigrams	350,000	16,609,143	3,727.7	0.0224
Criteo Kaggle 2014	45,840,617	1,000,000	39.0	0.00390
Avazu	40,428,967	1,000,000	15.0	0.00150
URL Combined	2,396,130	3,231,961	115.6	0.00358
KDD 2010	19,264,097	29,890,095	29.4	0.0000984
KDD 2012	149,639,105	54,686,452	11.0	0.0000201

There are two general qualities we want to evaluate each parallel training approach for: quality of solution and parallel speedup. Solution quality can be measured by looking at changes in accuracy with respect to the standard SDCA algorithm trained with a single core. When a model trained with multiple-cores has $\geq 2\%$ drop in accuracy compared to the standard SDCA algorithm, we consider it a model failure. Parallel speedup is a more concrete property. It is defined as the runtime required for the single-threaded SDCA algorithm divided by the runtime for the parallel approach under question.

To compare the SAUS, Wild, and Atomic approaches to training a parallel model via SDCA, we implement all three approaches in Java using JSAT [12]. Comparing with pre-existing C/C++ implementations of single-threaded SDCA, we found no significant performance degradations. Implementing all algorithms in the same framework and language allows us to meaningfully compare speedup results. To test in the many-core regime, all experiments were done on a single machine with 4 Intel XEON E7-8870v4 CPUs @ 2.10 GHz, 2 TB of RAM, and 40 TB of SSD storage. Hyper-threading was disabled for all tests. This machines allows us to test with up to 80 total CPU cores across 4 processors. For tests with $P \leq 80$, we pin the process to use the minimum number of CPUs, ensuring that no extraneous cross-CPU communication occurs over the QPI interface. We will use this many-core machine and the aforementioned metrics to evaluate three approaches: our new SAUS algorithm, the Wild-PASSCoDe, and the Atomic-PASSCoDe [7].

A. Model Quality

We first look at the model quality results. For any parallel training approach to be used in practice it needs to reliably produce models of quality comparable to that of the serial approach. We plot the final accuracy after convergence as a function of the number of cores used for the SAUS, Wild, and Atomic approaches. This is shown in Figure 1, where the solid black line indicates the accuracy of the serial SDCA algorithm. The x-axis is shared across figures and is log spaced, showing performance for $P \in \{2, 4, 8, 16, 32, 64, 80\}$ cores.

The SAUS, Wild, and Atomic approaches all converge to models with nearly identical accuracy on three datasets: Avazu, webspam, and KDD12. However, as we have discussed, the approaches have performance that is both hardware and dataset sensitive. We see failures for the Atomic and Wild approaches on datasets with many different properties.

On the epsilon dataset, which is 100% dense, we see the Atomic approach begin to fail after 32 cores are in use, and dramatically diverge to random guessing when all 80 cores are in use. The Wild approach also begins to degrade on this dataset after 64 cores. The a9a dataset is also interesting as our smallest corpus. Due to its limited size, we would not expect any approach to obtain significant speedups. Yet we see the Wild approach begin to fail after just 4 cores are in use, with the Atomic approach also becoming unstable after 32 cores are in play.

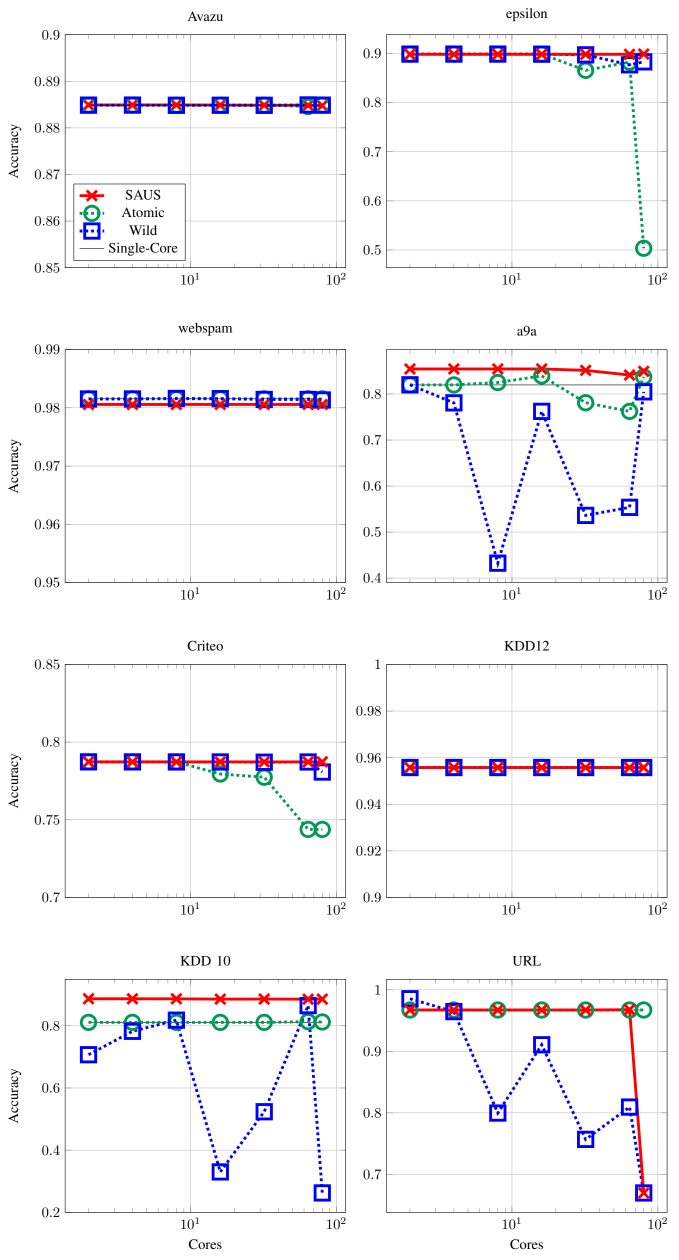

Fig. 1: Accuracy of final model (y-axis, scaled differently for each figure) as a function of the number of CPU cores used (x-axis, scale shared for each figure).

The divergence issues of the Atomic and Wild approaches are not limited to the datasets which are difficult for parallelism such as epsilon (due to its density) and a9a (due to its diminutive size). Both Atomic and Wild also suffer quality failures on large and sparse datasets, which is the ideal scenario for training parallel linear models. On Criteo, we see the Atomic approach degrading after 16 cores enter use. On KDD 2010, we see the Wild approach failing after just 2 cores enter use.

The only case where we see a model quality failure for SAUS is with the URL dataset, where it fails at the extreme of 80 cores. This is also the most challenging dataset for all approaches, likely due to a mix of features that are frequently non-zero and features that are rarely used — a fact hidden from view by the overall sparsity of the corpus. The Wild approach has greater model divergence on this corpus than SAUS, failing after just 8 cores. While the Atomic approach manages to have no failures on this corpus, we will see in subsection IV-B that it achieves this by degrading down to single-threaded performance, rendering it no faster than the serial approach. While not a failure by our accuracy definition, it is a failure in parallel speedup.

Overall, SAUS has only *one* model failure at 80 cores on the most challenging dataset, URL. The Wild approach has 16 failures, and the Atomic approach seven. SAUS's superior model quality can also be measured with a Wilcoxon signed ranks test [17]. The Wilcoxon test is preferable to the more common t-test and Friedman test [1, 4], and we use the table provided in [18] for the p-values. This is a pairwise test done across each dataset for each number of CPU cores tested. For both SAUS vs Atomic updates and SAUS vs Wild updates, we obtain $p < 0.001$ — a statistically significant result.

A Potential Regularizing Effect of SAUS: In all cases we used the same regularization parameter λ for each dataset, and used the same random split for the training and testing sets. Despite this, we notice that SAUS consistently has a slightly improved accuracy on a9a and KDD 2010, and slightly degraded accuracy on Webspam. We suspect that the SAUS update scheme introduces an implicit regularization effect. Similarly, Hsieh et al. [7] showed that the Wild update approach to SDCA, when solving Equation 1 with $\Omega(w) = ||w||_2^2$, actually finds the solution to a perturbed regularizer $\widetilde{\Omega}(w) = \Omega(w + \epsilon)$. This would explain the slight accuracy difference, despite all implementations sharing the same code and usually obtaining results that are identical to that of the serial SDCA algorithm. This also occurs, albeit inconsistently, with both the Atomic and Wild approaches on a9a, KDD 2010, and URL datasets.

B. *Parallel Speedup*

We have now demonstrated that SAUS produces, to a statistically significant degree, the highest quality model solutions across several datasets for any number of cores $P \in [2, 80]$. The next important question is how much faster the SAUS approach is compared to the serial case, and compared to the pre-existing Wild and Atomic approaches. Similar to the previous section, we plot in Figure 2 the speedup for each dataset as a function of the number of cores P used.

The epsilon and a9a datasets are worst-case scenarios for parallel linear models. The a9a dataset is simply too small for there to be much potential speedup. Our SAUS approach achieved the highest speedup at a factor of 3x when 8 cores were used, but all three approaches degrade to single-core performance when 80 cores are used. On epsilon, the Wild approach performed best — reaching a 7x speedup, where our SAUS approach could only obtain 3x. The Atomic approach has a misleading "speedup" at 80 cores because the model diverged to random guessing causing early termination. While SAUS dose not perform well in these cases, they are also

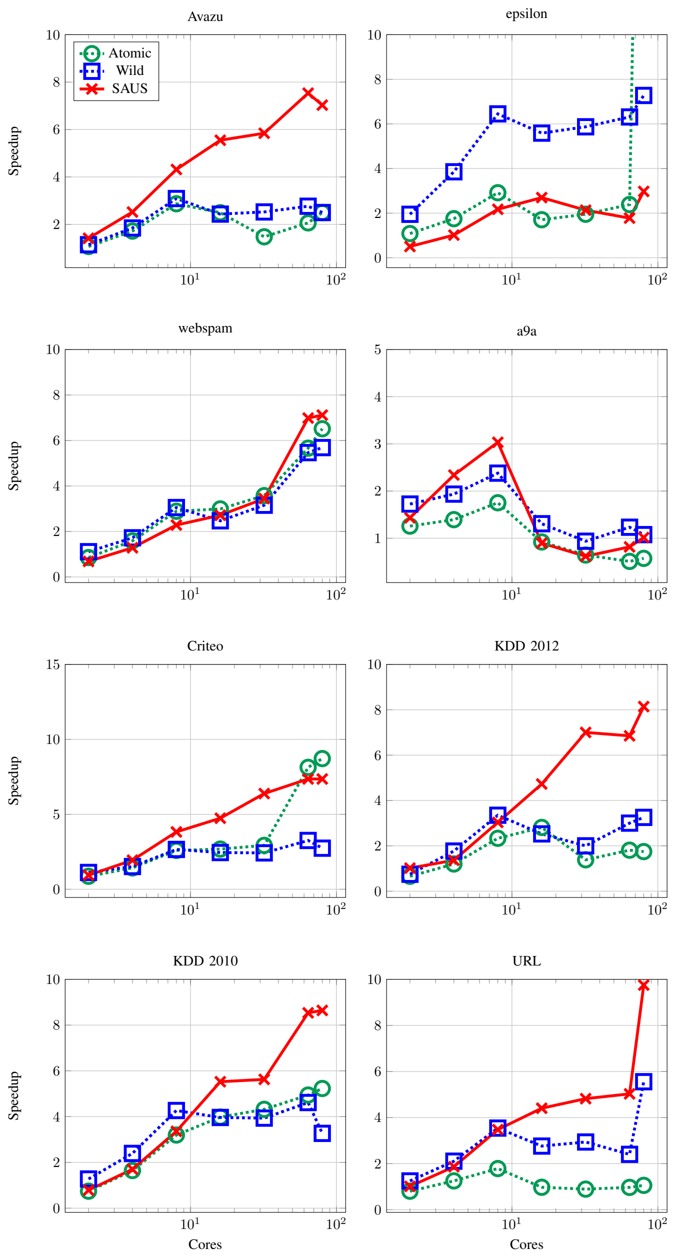

Fig. 2: Speedup of each approach as a function of the number of CPU cores used.

uniquely ill-suited for parallel computation. More important is that, as discussed in subsection IV-A, SAUS is the only method that has zero model quality failures on these two datasets while also still obtaining a speedup.

For all other datasets, when $P \in [2, 8]$ cores, we tend to see the SAUS and Wild approaches having competitive performance. On some datasets like webspam and KDD 2010, the Wild approach has slightly better speedups. On others like Criteo and Avazu, the SAUS approach performs slightly better.

This begins to change significantly as we move past 8 cores and enter the many-core scenarios. On most datasets, we see Wild's performance plateau in terms of speedup. The only dataset where the Wild approach continues to see a speed improvement is on the webspam corpus, reaching a final speedup of 5.7x at 80-cores, but still being out performed by the SAUS approach which achieves 7.1x. The Atomic approach has a similar plateauing, but usually has worse performance due to the overhead of constant atomic communication. The Atomic approach has an additional exception on the Criteo dataset, where its speedup "improves" after 32 cores are in use. However this is a false speedup, and as can be seen in Figure 1 this is actually caused by a degradation in the model quality leading to early convergence.

We emphasize that the plateau for the Wild approach at $P = 8$ *is not* because of inter-CPU communication (i.e., the QPI buss). Each CPU in our system has 20 cores and processes are pinned to the minimum number of needed CPUs. If QPI was the culprit, then we should have continued to see improvement at $P = 16$, but this does not happen. It appears passing 8 cores is when the communication overhead of the MESIF protocol between cores starts to dominate.

As mentioned in the previous section, the URL dataset is the most challenging due to its unusual distribution of non-zero values. The SAUS approach fails in terms of model quality & accuracy only once 80 cores are used, causing the false speedup spike. (The Wild approach shows the same behavior when $P = 80$.) While the Atomic approach has converged to a model with the same accuracy as the serial SDCA algorithm, its runtime performance has also degraded to that of the serial algorithm once 16 cores were in use. Because the Wild approach also has model quality failures after 4-cores are used, the SAUS approach is the only algorithm that achieved both a speedup and high quality solution when $P > 4$, topping out at a 5.0x speedup at 64 cores. While this isn't dramatic in terms of an ideal linear speedup with P, the SAUS approach is the only method to obtain any speedup *and* an accurate solution, making it the only realistically usable solution for $P \in [8, 64]$ cores.

When we look at the other datasets, we also see that the SAUS approach does not tend to plateau as more cores are added. Indeed it achieves the highest 80-core speedups on criteo, KDD 2010, KDD 2012, webspam, and avazu. The specific results on the maximum speedup achieved without any model accuracy failures are presented in Table II, where X indicates the speedup over the serial SDCA algorithm, and P indicates the number of processors used to achieve that speedup.

Here it becomes clear that SAUS approach is the only method to consistently benefit from using more than 8 cores. Being able to effectively use all available cores is important for more than just efficiency grounds: it frees the practitioner to use as many computational resources as they have on a problem without worry. Even when the Wild and Atomic approaches occasionally use more than 8 cores, they still don't achieve the same speedups that SAUS does. The only exception is the epsilon dataset, where the Wild approach obtains a considerably better speedup than the SAUS or Atomic methods, but this dataset is dense, making it an exceptional case. Looking at the median results across datasets, SAUS is able to scale to 10x as many CPU cores and is 2.4x times faster than the Wild

TABLE II: The best speedup results X for SAUS, Wild, and Atomic updates, and the number of cores P used to achieve that speedup. Best results shown in **bold**.

	SAUS		Wild		Atomic	
Dataset	X	P	X	P	X	P
Epsilon	2.97	**80**	**7.27**	**80**	2.92	8
a9a	**3.03**	**8**	1.94	2	1.75	**8**
Webspam	**7.12**	**80**	5.69	**80**	6.51	**80**
Criteo	**7.36**	**80**	3.25	64	2.59	8
Avazu	**7.53**	**64**	3.09	8	2.87	8
URL	**5.03**	**64**	2.11	4	1.79	8
KDD 2010	**8.64**	**80**	4.27	8	5.23	**80**
KDD 2012	**8.14**	**80**	1.77	4	2.81	16
Median	**7.45**	**80**	3.17	8	2.87	8

approach and 2.6x faster than the Atomic version.

V. Discussion

Given these results we have shown that the SAUS approach is better able to use the large number of cores available on modern systems. We now discuss why SAUS is able to achieve these improvements over prior approaches. First, our SAUS algorithm can also be shown to converge with SDCA using the same framework developed in [7]. From that mathematical perspective there is no obvious reason why SAUS should perform better. Instead we find it informative to also look at the communication costs. In examining the nature of our stochastic updates we can show that the SAUS approach has naturally limited communication overhead. Using this result, we then show analytically that its performance is independent of the number of cores used.

A. SAUS Convergence

We first expound upon how SAUS can be seen to converge to the solution from the same mathematical perspective given in Hsieh et al. [7]. Their work, like others, require simplifying assumptions on the communication in order to reason about in an analytic manner. By showing how SAUS fits in and stretches the assumptions of their framework, we obtain better understanding about where improvements in modeling could be obtained or where the theory may be overly conservative in its assumptions.

Regarding the convergence properties of SAUS, we remind the reader of the probability multiplier ϱ that adaptively increases as necessary to ensure convergence. As $\varrho \Rightarrow P$, SAUS approaches the original PASSCoDe-Atomic behavior. In the extreme, $\varrho = P$ and SAUS degrades into the standard PASSCoDe-Atomic algorithm, and thus converges under the exact same logic that was presented in Hsieh et al. [7]. This is the simplest case in which we converge in the same framework, but is not informative to a deeper understanding.

Irrespective of the existence of ϱ, we can analyze the convergence of SAUS for SDCA using the same assumptions and proof provided for PASSCoDe-Atomic. Hsieh et al. define the current "accurate" weight vector at time step j as $w^j = \sum_{i=1}^{n} \boldsymbol{\alpha}_i^j \boldsymbol{x}_i$, where $\boldsymbol{\alpha}^j$ is the vector of coefficients for each dataum. Simultaneously, $\hat{\boldsymbol{w}}^j$ represents the current model weight vector that exists in memory. Because not all threads will have written their updates to $\hat{\boldsymbol{w}}^j$, its value will be inaccurate. For their proof, they assume that all updates before the $(j-\tau)$-th iteration have been written into $\hat{\boldsymbol{w}}^j$. Their convergence proof then relies on the assumption that

$$\left(6\tau(\tau+1)^2 eM\right)/\sqrt{n} \leq 1 \tag{2}$$

where M is a data dependent constant defined in [7].

We note critically that no bound, expectation, or intuition on the value of τ is given, it is simply assumed that it will be small enough to satisfy the bound. Given that we frequently see PASSCoDe-Atomic (and Wild) diverge in our testing, we know this assumption does not always hold in practice. Most all convergence proofs we are aware of rely on similar assumptions that updates will be "fast enough" in some sense [9, 16, 21], which makes the proofs dependent on hardware and dataset combinations. Given that no statements are given about the delay τ, we could argue that SAUS converges under the same assumptions at this point. This would be reasonable given that SAUS converges to a high quality model more frequently than PASSCoDe-Atomic in our tests. Nevertheless, we will attempt to provide further insights into the expected value of τ with SAUS.

For SAUS, we can use the global shared weight vector $\boldsymbol{w}$ for the same purpose as $\hat{\boldsymbol{w}}^j$. The sum of weight vectors in thread local contexts $\tilde{\boldsymbol{w}} = \sum_{\forall p \in P} \boldsymbol{w}^p$ then represents all $(j-\tau)$ updates that have not yet been written to the global weight vector. We can then say τ will be distributed similarly to the rate at which non-zero coefficients from $\tilde{\boldsymbol{w}}$ are transfered into $\hat{\boldsymbol{w}}^j$. Because we expect only one thread to be performing an update of $\hat{\boldsymbol{w}}^j$ at a time (see subsection V-B), we can assume τ will not be any larger than this value in expectation. If this assumption where not true, we would see $\varrho \Rightarrow P$, at which point we converge again under the same trivial assumptions we discussed at the beginning of this section.

This allows us to infer a worst case value for τ. If f_d is the frequency at which the d'th feature was non-zero in the dataset, $f_M = \arg\min_{d \in [1,D]} f_d$, and each thread has an $1/|P|$ chance of transferring its information for its local $\boldsymbol{w}^p$ into $\hat{\boldsymbol{w}}^j$, we expect $\tau \lesssim n|P|/f_M$. This is essentially the pessimistic assumption that τ follows the average update rate of the most infrequently seen feature and is updated for all contexts.

A more complicated analysis could be done to take into account that we are working with an L_1 penalty on the regularizer which will induce sparsity in the weight coefficients. As such τ could be tightened by replacing f_M with the frequency of the least frequent feature d that its expected to have a non-zero coefficient in $\hat{\boldsymbol{w}}_d^j$.

In practical use, the worst case $\tau \approx n|P|/f_M$ can't hold all the time. f_M can only have an impact on τ when the feature has occurred once, and we are waiting for a 2nd occurrence to move the information from $\tilde{\boldsymbol{w}}_M$ into $\hat{\boldsymbol{w}}_M^j$.

Toward our understanding of the convergence framework, it would appear that our worst case analysis would general place τ on the order of $\mathcal{O}(n)$, which would imply the bound (2) is likely to fail. Because we see better convergence of SAUS than PASSCoDe, this would seem to imply that the bound is too pessimistic in practice. Our intuition is that the analysis fails to capture that the value of τ will vary of the execution of the program, and that not all features are equally informative to the solution. Beyond the scope of our work, an improved framework might take into account multiple τ for each feature and consider the average value of each coordinate-wise τ to better capture the range of convergence.

Ultimately, our SAUS approach has only one convergence failure in our testing, compared to seven for PASSCoDe-Atomic. As such we do not believe the theoretical frameworks fully capture the intricacies of convergence on modern hardware when many cores, CPUs, caches, and other hardware intricacies begin to interact.

B. Expected Number of Threads per Update

We first show that we will expect only a constant number of threads to be attempting an update at a given time. This is under the light assumption that all processors P are operating in a synchronous manner with each thread simultaneously performing updates at the same rate.

Under this assumption, we can view the choice of performing a stochastic update as a Binomial distribution (3), for n trials with probability of success p.

$$B(n,p) = \binom{n}{k} p^k \left(1-p\right)^{n-k} \tag{3}$$

The mean of this is $\mathbb{E}[B(n,p)] = np$. In our scenario, the n trials corresponds to the P processors, and the probability of success is defined as P^{-1}. Thus we can see that the expected number of threads performing an update is $\mathbb{E}[B(P, P^{-1})] = PP^{-1} = 1$.

A stronger statement can be made with Chernoff's upper-tail bound [3]. Given random variables Z_i such that $Z_i \in [0,1]$, we have the bound $\mathrm{P}\left(\sum_{i=1}^{n} Z_i \geq (1+\epsilon)\mu\right) \leq \exp\left(-\mu\left((1+\epsilon)\log(1+\epsilon) - \epsilon\right)\right)$. In our application, the expected sum $\mu = 1$ and $Z \sim B(P, P^{-1})$. Thus right-hand side simplifies to $\leq \exp\left(-(1+\epsilon)\log(1+\epsilon) + \epsilon\right)$. This gives us a probabilistic bound on the number of threads attempting to perform a simultaneous update. Choosing a value of $\epsilon = 4$ tells us that there is a less than 1.7% chance of five threads performing a simultaneous update, *regardless of the number of processors P in the system.*

C. Analytic Efficiency

We use this result of an expected constant number of threads per update to derive the *efficiency* of the SAUS algorithm. The efficiency $E \in (0,1]$ describes its use of parallel resources, and is defined as the speedup of the algorithm divided by the number of processors P used. When $E = 1$, we obtain linear speedup with the number of cores P. As the parallel algorithm becomes slower E will trend toward 0. By describing efficiency via the explicit costs and communication overheads in each algorithm, we can derive analytic equations describing the efficiency of each approach. This is done using the same framework and notation introduced by Grama et al. [6] in their Isoefficiency work, which describes the costs of parallel algorithms in big-O notation (which we will assume implicitly in this section for legibility). In particular, Grama et al. show that the efficiency can be described as $E = 1/(1 + T_o/T_1)$, where T_o describes the *total overhead* of the parallel algorithm, and T_1 is the work for the single-threaded algorithm. If T_p is the total work done per thread in the parallel case, the three are connected via $T_p = (T_1 + T_o)/P$.

All of our models have the same $T_1 = IN\overline{D}$, where I is the number of epochs performed. For the Atomic approach, the communication overhead T_o for P threads performing an update is $\mathcal{O}(P)$ due to the CAS loop. This cost occurs on every update of every epoch. Thus the overhead T_o for the Atomic approach is $T_o^{\text{Atomic}} = PIN\overline{D}$. In the case of the Wild approach, there is no explicit communication costs. This would imply that $T_o^{\text{wild}} = 0$, but this ignores the communication that occurs via the hardware. For this reason we don't try to characterize its behavior, as it will be a function of the specific hardware instantiation. Based on the results we see in section IV, it would be reasonable to argue that the Wild approach has the same asymptotic T_o term as the Atomic approach, but has differing constants. We draw this conclusion by noting that the Wild and Atomic speedup curves tend to exhibit the same shape, but offset from one-another.

In the case of SAUS, we have just shown that the expected number of threads performing an update at a time is 1, and with a high probability will be a small constant number. This disappears in the big-O notation, and so the majority of communication overhead compared to the serial algorithm comes from the Accumulation step. The accumulation step is run without communication on disjoint subsets of the weight vector w, and so would contribute $\mathcal{O}(DI)$ to the overhead term (adding each feature once per epoch). Thus we can say that $T_o^{\text{SAUS}} = DI$.

Now we can compute the efficiency via $E = \frac{1}{1+T_o/T_1}$ [6]. This gives us:

$$E_{\text{Atomic}} = \frac{1}{1 + \frac{T_o^{\text{Atomic}}}{T_1}} = \frac{1}{1 + \frac{PIN\overline{D}}{IN\overline{D}}} = \frac{1}{1+P} \tag{4}$$

$$E_{\text{SAUS}} = \frac{1}{1 + \frac{T_o^{\text{SAUS}}}{T_1}} = \frac{1}{1 + \frac{ID}{IN\overline{D}}} = \frac{1}{1 + \frac{D}{\overline{D}}\frac{1}{N}} \tag{5}$$

For the Atomic case in (4), we see that the efficiency is a function dependent on the number of processors P in the system. As we keep the problem size constant, we expect efficiency to decrease with more cores.

Our new SAUS approach reveals an efficiency in (5) that is independent of the number of cores P. Instead it depends on the inverse sparsity divided by the problem size. The inverse sparsity comes from the Accumulate step which has low constant overhead, and we can see SAUS's largest speedups come from sparse datasets like KDD 2012. The $1/N$ term also benefits SAUS, as we are interested in parallel training specifically when N is large and thus would take too long with a single core.

Overall, our results back this theoretical analysis. Equation 5 tells us that the efficiency of our method should be independent of the number of processors P, which explains why we see superior speedup compared to the prior approaches as more cores are added. This analysis assumes that the ϱ term in SAUS equals one, which does not happen for the URL dataset as more cores are added. This is a limitation of our current analysis, but the addition has the benefit of degrading back to the Atomic approach for difficult datasets. Being able to describe when ϱ changes and incorporating it into this efficiency analysis is an important direction for future work.

VI. Conclusion

In this paper we have presented SAUS, a new approach to training linear models in parallel that stochastically updates the shared weight vector using safe atomic updates. SAUS provides better model stability compared to both the prior Wild and Atomic updating approaches, while also scaling better as the number of CPU cores increases past 8 and up to 80. This allows SAUS to work effectively on the wide array of massively parallel systems available today, without requiring significant user expertise on the potential pitfalls of parallel linear model training. Further, we have provided theoretical analysis to explain why SAUS provides better performance in the many-core scenario.

References

[1] A. Benavoli, G. Corani, and F. Mangili. Should We Really Use Post-Hoc Tests Based on Mean-Ranks? *Journal of Machine Learning Research*, 17(5):1–10, 2016.

[2] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3), Apr. 2011. ISSN: 21576904.

[3] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations. *The Annals of Mathematical Statistics*, 23(4):493–507, 1952. ISSN: 00034851.

[4] J. Demšar. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research*, 7:1–30, Dec. 2006. ISSN: 1532-4435.

[5] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A Library for Large Linear Classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

[6] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel Distrib. Technol.*, 1(3):12–21, Aug. 1993. ISSN: 1063-6552.

[7] C.-J. Hsieh, H.-F. Yu, and I. S. Dhillon. PASSCoDe: Parallel Asynchronous Stochastic Dual Co-ordinate Descent. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 2370–2379. JMLR.org, 2015.

[8] D. Lea. DoubleAdder, 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/DoubleAdder.html.

[9] R. Leblond, F. Pedregosa, and S. Lacoste-Julien. ASAGA: Asynchronous Parallel SAGA. In A. Singh and J. Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 46–54, Fort Lauderdale, FL, USA. PMLR, 2017.

[10] D. Molka, D. Hackenberg, R. Schone, and W. E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *2015 44th International Conference on Parallel Processing*, pages 739–748. IEEE, Sept. 2015. ISBN: 978-1-4673-7587-0.

[11] A. Y. Ng. Feature selection, L1 vs. L2 regularization, and rotational invariance. *Twenty-first international conference on Machine learning - ICML '04*:78, 2004.

[12] E. Raff. JSAT: Java Statistical Analysis Tool, a Library for Machine Learning. *Journal of Machine Learning Research*, 18(23):1–5, 2017.

[13] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[14] S. Shalev-Shwartz and T. Zhang. Stochastic Dual Coordinate Ascent Methods for Regularized Loss Minimization. *Journal ofMachine Learning Research*, 14:567–599, Sept. 2012.

[15] M. E. Thomadakis. The architecture of the Nehalem processor and Nehalem-EP SMP platforms. *Resource*, 3(2), 2011.

[16] K. Tran, S. Hosseini, L. Xiao, T. Finley, and M. Bilenko. Scaling Up Stochastic Dual Coordinate Ascent. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '15*, pages 1185–1194, New York, New York, USA. ACM Press, Aug. 2015. ISBN: 9781450336642.

[17] F. Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80, Dec. 1945. ISSN: 00994987.

[18] J. H. Zar. *Biostatistical Analysis*. Prentice Hall, 1999. ISBN: 9780130815422.

[19] H. Zhang and C.-j. Hsieh. Fixing the Convergence Problems in Parallel Asynchronous Dual Coordinate Descent. In *IEEE International Conference on Data Mining (ICDM)*, 2016.

[20] H. Zhang, C.-J. Hsieh, and V. Akella. HogWild++: A New Mechanism for Decentralized Asynchronous Stochastic Gradient Descent. In *IEEE International Conference on Data Mining (ICDM)*. ICDM, 2016.

[21] S.-Y. Zhao and W.-J. Li. Fast Asynchronous Parallel Stochastic Gradient Descent: A Lock-free Approach with Convergence Guarantee. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2379–2385. AAAI Press, 2016.