HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent

Feng Niu
leonn@cs.wisc.edu

Benjamin Recht
brecht@cs.wisc.edu

Christopher Ré
chrisre@cs.wisc.edu

Stephen J. Wright
swright@cs.wisc.edu
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

Abstract

Stochastic Gradient Descent (SGD) is a popular algorithm that can achieve state-of-the-art performance on a variety of machine learning tasks. Several researchers have recently proposed schemes to parallelize SGD, but all require performance-destroying memory locking and synchronization. This work aims to show using novel theoretical analysis, algorithms, and implementation that SGD can be implemented *without any locking*. We present an update scheme called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. We show that when the associated optimization problem is *sparse*, meaning most gradient updates only modify small parts of the decision variable, then HOGWILD! achieves a nearly optimal rate of convergence. We demonstrate experimentally that HOGWILD! outperforms alternative schemes that use locking by an order of magnitude.

1 Introduction

With its small memory footprint, robustness against noise, and rapid learning rates, Stochastic Gradient Descent (SGD) has proved to be well suited to data-intensive machine learning tasks [3,5,24]. However, SGD's scalability is limited by its inherently sequential nature; it is difficult to parallelize. Nevertheless, the recent emergence of inexpensive multicore processors and mammoth, web-scale data sets has motivated researchers to develop several clever parallelization schemes for SGD [4, 10, 12, 16, 27]. As many large data sets are currently pre-processed in a MapReduce-like parallel-processing framework, much of the recent work on parallel SGD has focused naturally on MapReduce implementations. MapReduce is a powerful tool developed at Google for extracting information from huge logs (e.g., "find all the urls from a 100TB of Web data") that was designed to ensure fault tolerance and to simplify the maintenance and programming of large clusters of machines [9]. But MapReduce is not ideally suited for online, numerically intensive data analysis. Iterative computation is difficult to express in MapReduce, and the overhead to ensure fault tolerance can result in dismal throughput. Indeed, even Google researchers themselves suggest that other systems, for example Dremel, are more appropriate than MapReduce for data analysis tasks [20].

For some data sets, the sheer size of the data dictates that one use a cluster of machines. However, there are a host of problems in which, after appropriate preprocessing, the data necessary for statistical analysis may consist of a few terabytes or less. For such problems, one can use a single inexpensive work station as opposed to a hundred thousand dollar cluster. Multicore systems have significant performance advantages, including (1) low latency and high throughput shared main memory (a processor in such a system can write and read the shared physical memory at over 12GB/s with latency in the tens of nanoseconds); and (2) high bandwidth off multiple disks (a thousand-dollar RAID

can pump data into main memory at over 1GB/s). In contrast, a typical MapReduce setup will read incoming data at rates less than tens of MB/s due to frequent checkpointing for fault tolerance. The high rates achievable by multicore systems move the bottlenecks in parallel computation to synchronization (or locking) amongst the processors [2, 13]. Thus, to enable scalable data analysis on a multicore machine, any performant solution must minimize the overhead of locking.

In this work, we propose a simple strategy for eliminating the overhead associated with locking: *run SGD in parallel without locks*, a strategy that we call HOGWILD!. In HOGWILD!, processors are allowed equal access to shared memory and are able to update individual components of memory at will. Such a lock-free scheme might appear doomed to fail as processors could overwrite each other's progress. However, when the data access is *sparse*, meaning that individual SGD steps only modify a small part of the decision variable, we show that memory overwrites are rare and that they introduce barely any error into the computation when they do occur. We demonstrate both theoretically and experimentally a near linear speedup with the number of processors on commonly occurring sparse learning problems.

In Section 2, we formalize a notion of sparsity that is sufficient to guarantee such a speedup and provide canonical examples of sparse machine learning problems in classification, collaborative filtering, and graph cuts. Our notion of sparsity allows us to provide theoretical guarantees of linear speedups in Section 4. As a by-product of our analysis, we also derive rates of convergence for algorithms with constant stepsizes. We demonstrate that robust $1/k$ convergence rates are possible with constant stepsize schemes that implement an exponential back-off in the constant over time. This result is interesting in of itself and shows that one need not settle for $1/\sqrt{k}$ rates to ensure robustness in SGD algorithms.

In practice, we find that computational performance of a lock-free procedure exceeds even our theoretical guarantees. We experimentally compare lock-free SGD to several recently proposed methods. We show that all methods that propose memory locking are significantly slower than their respective lock-free counterparts on a variety of machine learning applications.

2 Sparse Separable Cost Functions

Our goal throughout is to minimize a function $f : X \subseteq \mathbb{R}^n \to \mathbb{R}$ of the form

$$f(x) = \sum_{e \in E} f_e(x_e)\,. \tag{1}$$

Here e denotes a small subset of $\{1, \dots, n\}$ and x_e denotes the values of the vector x on the coordinates indexed by e. The key observation that underlies our lock-free approach is that the natural cost functions associated with many machine learning problems of interest are *sparse* in the sense that $|E|$ and n are both very large but each individual f_e acts only on a very small number of components of x. That is, each subvector x_e contains just a few components of x.

The cost function (1) induces a *hypergraph* $G = (V, E)$ whose nodes are the individual components of x. Each subvector x_e induces an edge in the graph $e \in E$ consisting of some subset of nodes. A few examples illustrate this concept.

Sparse SVM. Suppose our goal is to fit a support vector machine to some data pairs $E = \{(z_1, y_1), \dots, (z_{|E|}, y_{|E|})\}$ where $z \in \mathbb{R}^n$ and y is a label for each $(z, y) \in E$.

$$\text{minimize}_x \sum_{\alpha \in E} \max(1 - y_\alpha x^T z_\alpha, 0) + \lambda \|x\|_2^2\,, \tag{2}$$

and we know *a priori* that the examples z_α are very sparse (see for example [14]). To write this cost function in the form of (1), let e_α denote the components which are non-zero in z_α and let d_u denote the number of training examples which are non-zero in component u ($u = 1, 2, \dots, n$). Then we can rewrite (2) as

$$\text{minimize}_x \sum_{\alpha \in E} \left(\max(1 - y_\alpha x^T z_\alpha, 0) + \lambda \sum_{u \in e_\alpha} \frac{x_u^2}{d_u} \right)\,. \tag{3}$$

Each term in the sum (3) depends only on the components of x indexed by the set e_α.

Matrix Completion. In the matrix completion problem, we are provided entries of a low-rank, $n_r \times n_c$ matrix $\boldsymbol{Z}$ from the index set E. Such problems arise in collaborative filtering, Euclidean distance estimation, and clustering [8,17,23]. Our goal is to reconstruct $\boldsymbol{Z}$ from this sparse sampling of data. A popular heuristic recovers the estimate of $\boldsymbol{Z}$ as a product $\boldsymbol{L}\boldsymbol{R}^*$ of factors obtained from the following minimization:

$$\text{minimize}_{(\boldsymbol{L},\boldsymbol{R})} \sum_{(u,v)\in E} (\boldsymbol{L}_u \boldsymbol{R}_v^* - Z_{uv})^2 + \tfrac{\mu}{2}\|\boldsymbol{L}\|_F^2 + \tfrac{\mu}{2}\|\boldsymbol{R}\|_F^2\,, \tag{4}$$

where $\boldsymbol{L}$ is $n_r \times r$, $\boldsymbol{R}$ is $n_c \times r$ and $\boldsymbol{L}_u$ (resp. $\boldsymbol{R}_v$) denotes the uth (resp. vth) row of $\boldsymbol{L}$ (resp. $\boldsymbol{R}$) [17,23,25]. To put this problem in sparse form, i.e., as (1), we write (4) as

$$\text{minimize}_{(\boldsymbol{L},\boldsymbol{R})} \sum_{(u,v)\in E} \left\{ (\boldsymbol{L}_u \boldsymbol{R}_v^* - Z_{uv})^2 + \tfrac{\mu}{2|E_{u-}|}\|\boldsymbol{L}_u\|_F^2 + \tfrac{\mu}{2|E_{-v}|}\|\boldsymbol{R}_v\|_F^2 \right\}$$

where $E_{u-} = \{v \,:\, (u,v)\in E\}$ and $E_{-v} = \{u \,:\, (u,v)\in E\}$.

Graph Cuts. Problems involving minimum cuts in graphs frequently arise in machine learning (see [6] for a comprehensive survey). In such problems, we are given a sparse, nonnegative matrix W which indexes similarity between entities. Our goal is to find a partition of the index set $\{1,\ldots,n\}$ that best conforms to this similarity matrix. Here the graph structure is explicitly determined by the similarity matrix W; arcs correspond to nonzero entries in W. We want to match each string to some list of D entities. Each node is associated with a vector x_i in the D-dimensional simplex $S_D = \{\zeta \in \mathbb{R}^D \,:\, \zeta_v \geq 0 \;\; \sum_{v=1}^D \zeta_v = 1\}$. Here, two-way cuts use $D=2$, but multiway-cuts with tens of thousands of classes also arise in entity resolution problems [18]. For example, we may have a list of n strings, and W_{uv} might index the similarity of each string. Several authors (e.g., [7]) propose to minimize the cost function

$$\text{minimize}_x \sum_{(u,v)\in E} w_{uv}\|x_u - x_v\|_1 \quad \text{subject to} \quad x_v \in S_D \quad \text{for } v = 1,\ldots,n\,. \tag{5}$$

In all three of the preceding examples, the number of components involved in a particular term f_e is a small fraction of the total number of entries. We formalize this notion by defining the following statistics of the hypergraph G:

$$\Omega := \max_{e\in E}|e|,\;\; \Delta := \frac{\max_{1\leq v\leq n}|\{e\in E \,:\, v\in e\}|}{|E|},\;\; \rho := \frac{\max_{e\in E}|\{\hat{e}\in E \,:\, \hat{e}\cap e \neq \emptyset\}|}{|E|}\,. \tag{6}$$

The quantity Ω simply quantifies the size of the hyper edges. ρ determines the maximum fraction of edges that intersect any given edge. Δ determines the maximum fraction of edges that intersect any variable. ρ is a measure of the sparsity of the hypergraph, while Δ measures the node-regularity. For our examples, we can make the following observations about ρ and Δ.

1. **Sparse SVM**. Δ is simply the maximum frequency that any feature appears in an example, while ρ measures how clustered the hypergraph is. If some features are very common across the data set, then ρ will be close to one.
2. **Matrix Completion**. If we assume that the provided examples are sampled uniformly at random and we see more than $n_c \log(n_c)$ of them, then $\Delta \approx \frac{\log(n_r)}{n_r}$ and $\rho \approx \frac{2\log(n_r)}{n_r}$. This follows from a *coupon collector* argument [8].
3. **Graph Cuts**. Δ is the maximum degree divided by $|E|$, and ρ is at most 2Δ.

We now describe a simple protocol that achieves a linear speedup in the number of processors when Ω, Δ, and ρ are relatively small.

3 The HOGWILD! Algorithm

Here we discuss the parallel processing setup. We assume a shared memory model with p processors. The decision variable x is accessible to all processors. Each processor can read x, and can

Algorithm 1 HOGWILD! update for individual processors

```
1: loop
2:     Sample e uniformly at random from E
3:     Read current state x_e and evaluate G_e(x_e)
4:     for v ∈ e do x_v ← x_v − γ G_ev(x_e)
5: end loop
```

contribute an update vector to x. The vector x is stored in shared memory, and we assume that the componentwise addition operation is atomic, that is

$$x_v \leftarrow x_v + a$$

can be performed atomically by any processor for a scalar a and $v \in \{1, \dots, n\}$. This operation does not require a separate locking structure on most modern hardware: such an operation is a single atomic instruction on GPUs and DSPs, and it can be implemented via a compare-and-exchange operation on a general purpose multicore processor like the Intel Nehalem. In contrast, the operation of updating many components at once requires an auxiliary locking structure.

Each processor then follows the procedure in Algorithm 1. Let $G_e(x_e)$ denote a gradient or subgradient of the function f_e multiplied by $|E|$. That is,

$$|E|^{-1} G_e(x_e) \in \partial f_e(x_e).$$

Since it is clear by notation, we often write $G_e(x)$, dropping the notation that identifies the affected indices of x. Note that as a consequence of the uniform random sampling of e from E, we have

$$\mathbb{E}[G_e(x_e)] \in \partial f(x)\,.$$

In Algorithm 1, each processor samples an term $e \in E$ uniformly at random, computes the gradient of f_e at x_e, and then writes

$$x_v \leftarrow x_v - \gamma G_{ev}(x_e), \qquad \text{for each } v \in e. \tag{7}$$

We assume that the stepsize γ is a fixed constant. Note that the processor modifies only the variables indexed by e, leaving all of the components in $\neg e$ (i.e., not in e) alone. Even though the processors have no knowledge as to whether any of the other processors have modified x, we define x_j to be the state of the decision variable x after j updates have been performed[1]. Since two processors can write to x at the same time, we need to be a bit careful with this definition, but we simply break ties at random. Note that x_j is generally updated with a stale gradient, which is based on a value of x read many clock cycles earlier. We use $x_{k(j)}$ to denote the value of the decision variable used to compute the gradient or subgradient that yields the state x_j.

In what follows, we provide conditions under which this asynchronous, incremental gradient algorithm converges. Moreover, we show that if the hypergraph induced by f is isotropic and sparse, then this algorithm converges in nearly the same number of gradient steps as its serial counterpart. Since we are running in parallel and without locks, this means that we get a nearly linear speedup in terms of the number of processors.

4 Fast Rates for Lock-Free Parallelism

To state our theoretical results, we must describe several quantities that important in the analysis of our parallel stochastic gradient descent scheme. We follow the notation and assumptions of Nemirovski *et al* [21]. To simplify the analysis, we will assume that each f_e in (1) is a convex function. We assume Lipschitz continuous differentiability of f with Lipschitz constant L:

$$\|\nabla f(x') - \nabla f(x)\| \le L\|x' - x\|, \;\; \forall\, x', x \in X. \tag{8}$$

We also assume f is strongly convex with modulus c. By this we mean that

$$f(x') \ge f(x) + (x' - x)^T \nabla f(x) + \frac{c}{2}\|x' - x\|^2, \;\; \text{for all } x', x \in X. \tag{9}$$

[1]Our notation overloads subscripts of x. For clarity throughout, subscripts i, j, and k refer to iteration counts, and v and e refer to components or subsets of components.

When f is strongly convex, there exists a unique minimizer $x_\star$ and we denote $f_\star = f(x_\star)$. We additionally assume that there exists a constant M such that

$$\|G_e(x_e)\|_2 \leq M \quad \text{almost surely for all } x \in X\,. \tag{10}$$

We assume throughout that $\gamma c < 1$. (Indeed, when $\gamma c > 1$, even the ordinary gradient descent algorithms will diverge.) Our main results are summarized by the following

Proposition 4.1 *Suppose in Algorithm 1 that the lag between when a gradient is computed and when it is used in step j — namely, $j - k(j)$ — is always less than or equal to τ, and γ is defined to be*

$$\gamma = \frac{\vartheta \epsilon c}{2LM^2\left(1 + 6\rho\tau + 4\tau^2\Omega\Delta^{1/2}\right)}\,. \tag{11}$$

for some $\epsilon > 0$ and $\vartheta \in (0,1)$. Define $D_0 := \|x_0 - x_\star\|^2$ and let k be an integer satisfying

$$k \geq \frac{2LM^2\left(1 + 6\tau\rho + 6\tau^2\Omega\Delta^{1/2}\right)\log(LD_0/\epsilon)}{c^2\vartheta\epsilon}\,. \tag{12}$$

Then after k updates of x, we have $\mathbb{E}[f(x_k) - f_\star] \leq \epsilon$.

A proof of Proposition 4.1 is provided in the full version of this paper [22]. In the case that $\tau = 0$, this reduces to precisely the rate achieved by the serial SGD protocol. A similar rate is achieved if $\tau = o(n^{1/4})$ as ρ and Δ are typically both $o(1/n)$. In our setting, τ is proportional to the number of processors, and hence as long as the number of processors is less $n^{1/4}$, we get nearly the same recursion as in the linear rate.

Note that up to the $\log(1/\epsilon)$ term in (12), our analysis nearly provides a $1/k$ rate of convergence for a constant stepsize SGD scheme, both in the serial and parallel cases. Moreover, note that our rate of convergence is fairly robust to error in the value of c; we pay linearly for our underestimate of the curvature of f. In contrast, Nemirovski *et al* demonstrate that when the stepsize is inversely proportional to the iteration counter, an overestimate of c can result in exponential slow-down [21]!

Robust $1/k$ rates. We note that a $1/k$ can be achieved by a slightly more complicated protocol where the stepsize is slowly decreased after a large number of iterations. Suppose we run Algorithm 1 for a fixed number of gradient updates K with stepsize $\gamma < 1/c$. Then, we wait for the threads to coalesce, reduce γ by a constant factor $\beta \in (0,1)$, and run for $\beta^{-1}K$ iterations. This scheme results in a $1/k$ rate of convergence with the only synchronization overhead occurring at the end of each "round" or "epoch" of iteration. In some sense, this piecewise constant stepsize protocol approximates a $1/k$ diminishing stepsize. The main difference with our approach from previous analysis is that our stepsizes are always less than $1/c$ in contrast to beginning with very large stepsizes. Always working with small stepsizes allows us to avoid the possible exponential slow-downs that occur with standard diminishing stepsize schemes.

5 Related Work

Most schemes for parallelizing stochastic gradient descent are variants of ideas presented in the seminal text by Bertsekas and Tsitsiklis [4]. For instance, in this text, they describe using stale gradient updates computed across many computers in a master-worker setting and describe settings where different processors control access to particular components of the decision variable. They prove global convergence of these approaches, but do not provide rates of convergence (This is one way in which our work extends this prior research). These authors also show that SGD convergence is robust to a variety of models of delay in computation and communication in [26].

Recently, a variety of parallel schemes have been proposed in a variety of contexts. In MapReduce settings, Zinkevich et al proposed running many instances of stochastic gradient descent on different machines and averaging their output [27]. Though the authors claim this method can reduce both the variance of their estimate and the overall bias, we show in our experiments that for the sorts of problems we are concerned with, this method does not outperform a serial scheme.

Schemes involving the averaging of gradients via a distributed protocol have also been proposed by several authors [10, 12]. While these methods do achieve linear speedups, they are difficult

type	data set	size (GB)	ρ	Δ	HOGWILD! time (s)	HOGWILD! train error	HOGWILD! test error	ROUND ROBIN time (s)	ROUND ROBIN train error	ROUND ROBIN test error
SVM	RCV1	0.9	0.44	1.0	9.5	0.297	0.339	61.8	0.297	0.339
MC	Netflix	1.5	2.5e-3	2.3e-3	301.0	0.754	0.928	2569.1	0.754	0.927
	KDD	3.9	3.0e-3	1.8e-3	877.5	19.5	22.6	7139.0	19.5	22.6
	Jumbo	30	2.6e-7	1.4e-7	9453.5	0.031	0.013	**N/A**	**N/A**	**N/A**
Cuts	DBLife	3e-3	8.6e-3	4.3e-3	230.0	10.6	**N/A**	413.5	10.5	**N/A**
	Abdomen	18	9.2e-4	9.2e-4	1181.4	3.99	**N/A**	7467.25	3.99	**N/A**

Figure 1: Comparison of wall clock time across of HOGWILD! and RR. Each algorithm is run for 20 epochs and parallelized over 10 cores.

to implement efficiently on multicore machines as they require massive communication overhead. Distributed averaging of gradients requires message passing between the cores, and the cores need to synchronize frequently in order to compute reasonable gradient averages.

The work most closely related to our own is a round-robin scheme proposed by Langford et al [16]. In this scheme, the processors are ordered and each update the decision variable in order. When the time required to lock memory for writing is dwarfed by the gradient computation time, this method results in a linear speedup, as the errors induced by the lag in the gradients are not too severe. However, we note that in many applications of interest in machine learning, gradient computation time is incredibly fast, and we now demonstrate that in a variety of applications, HOGWILD! outperforms such a round-robin approach by an order of magnitude.

6 Experiments

We ran numerical experiments on a variety of machine learning tasks, and compared against a round-robin approach proposed in [16] and implemented in Vowpal Wabbit [15]. We refer to this approach as RR. To be as fair as possible to prior art, we hand coded RR to be nearly identical to the HOGWILD! approach, with the only difference being the schedule for how the gradients are updated. One notable change in RR from the Vowpal Wabbit software release is that we optimized RR's locking and signaling mechanisms to use spinlocks and busy waits (there is no need for generic signaling to implement round robin). We verified that this optimization results in nearly an order of magnitude increase in wall clock time for all problems that we discuss.

We also compare against a model which we call AIG which can be seen as a middle ground between RR and HOGWILD!. AIG runs a protocol identical to HOGWILD! except that it locks all of the variables in e in before and after the **for** loop on line 4 of Algorithm 1. Our experiments demonstrate that even this fine-grained locking induces undesirable slow-downs.

All of the experiments were coded in C++ are run on an identical configuration: a dual Xeon X650 CPUs (6 cores each x 2 hyperthreading) machine with 24GB of RAM and a software RAID-0 over 7 2TB Seagate Constellation 7200RPM disks. The kernel is Linux 2.6.18-128. We never use more than 2GB of memory. All training data is stored on a seven-disk raid 0. We implemented a custom file scanner to demonstrate the speed of reading data sets of disk into small shared memory. This allows us to read data from the raid at a rate of nearly 1GB/s.

All of the experiments use a constant stepsize γ which is diminished by a factor β at the end of each pass over the training set. We run all experiments for 20 such passes, even though less epochs are often sufficient for convergence. We show results for the largest value of the learning rate γ which converges and we use $\beta = 0.9$ throughout. We note that the results look the same across a large range of (γ, β) pairs and that all three parallelization schemes achieve train and test errors within a few percent of one another. We present experiments on the classes of problems described in Section 2.

Sparse SVM. We tested our sparse SVM implementation on the Reuters RCV1 data set on the binary text classification task CCAT [19]. There are 804,414 examples split into 23,149 training and 781,265 test examples, and there are 47,236 features. We swapped the training set and the test set for our experiments to demonstrate the scalability of the parallel multicore algorithms. In this example,

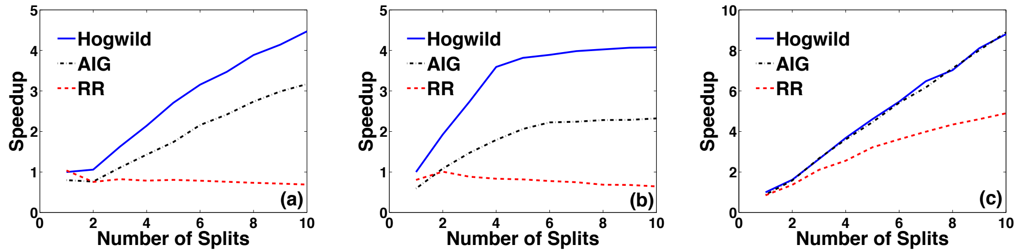

Figure 2: Total CPU time versus number of threads for (a) RCV1, (b) Abdomen, and (c) DBLife.

$\rho = 0.44$ and $\Delta = 1.0$—large values that suggest a bad case for HOGWILD!. Nevertheless, in Figure 2(a), we see that HOGWILD! is able to achieve a factor of 3 speedup with while RR gets worse as more threads are added. Indeed, for fast gradients, RR is worse than a serial implementation.

For this data set, we also implemented the approach in [27] which runs multiple SGD runs in parallel and averages their output. In Figure 3(b), we display at the train error of the ensemble average across parallel threads at the end of each pass over the data. We note that the threads only communicate at the very end of the computation, but we want to demonstrate the effect of parallelization on train error. Each of the parallel threads touches every data example in each pass. Thus, the 10 thread run does 10x more gradient computations than the serial version. Here, the error is the same whether we run in serial or with ten instances. We conclude that on this problem, there is no advantage to running in parallel with this averaging scheme.

Matrix Completion. We ran HOGWILD! on three very large matrix completion problems. The Netflix Prize data set has 17,770 rows, 480,189 columns, and 100,198,805 revealed entries. The KDD Cup 2011 (task 2) data set has 624,961 rows, 1,000,990, columns and 252,800,275 revealed entries. We also synthesized a low-rank matrix with rank 10, 1e7 rows and columns, and 2e9 revealed entries. We refer to this instance as "Jumbo." In this synthetic example, ρ and Δ are both around 1e-7. These values contrast sharply with the real data sets where ρ and Δ are both on the order of 1e-3.

Figure 3(a) shows the speedups for these three data sets using HOGWILD!. Note that the Jumbo and KDD examples do not fit in our allotted memory, but even when reading data off disk, HOGWILD! attains a near linear speedup. The Jumbo problem takes just over two and a half hours to complete. Speedup graphs like in Figure 2 comparing HOGWILD! to AIG and RR on the three matrix completion experiments are provided in the full version of this paper. Similar to the other experiments with quickly computable gradients, RR does not show any improvement over a serial approach. In fact, with 10 threads, RR is 12% slower than serial on KDD Cup and 62% slower on Netflix. We did not allow RR to run to completion on Jumbo because it several hours.

Graph Cuts. Our first cut problem was a standard image segmentation by graph cuts problem popular in computer vision. We computed a two-way cut of the `abdomen` data set [1]. This data set consists of a volumetric scan of a human abdomen, and the goal is to segment the image into organs. The image has $512 \times 512 \times 551$ voxels, and the associated graph is 6-connected with maximum capacity 10. Both ρ and Δ are equal to 9.2e-4 We see that HOGWILD! speeds up the cut problem by more than a factor of 4 with 10 threads, while RR is twice as slow as the serial version.

Our second graph cut problem sought a mulit-way cut to determine entity recognition in a large database of web data. We created a data set of clean entity lists from the DBLife website and of entity mentions from the DBLife Web Crawl [11]. The data set consists of 18,167 entities and 180,110 mentions and similarities given by string similarity. In this problem each stochastic gradient step must compute a Euclidean projection onto a simplex of dimension 18,167. As a result, the individual stochastic gradient steps are quite slow. Nonetheless, the problem is still very sparse with ρ=8.6e-3 and Δ=4.2e-3. Consequently, in Figure 2, we see the that HOGWILD! achieves a ninefold speedup with 10 cores. Since the gradients are slow, RR is able to achieve a parallel speedup for this problem, however the speedup with ten processors is only by a factor of 5. That is, even in this case where the gradient computations are very slow, HOGWILD! outperforms a round-robin scheme.

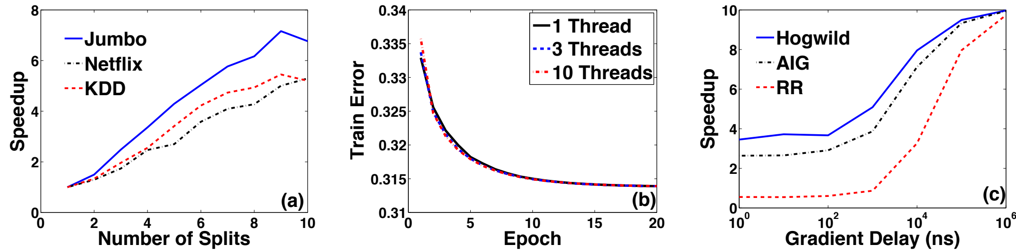

Figure 3: (a) Speedup for the three matrix completion problems with HOGWILD!. In all three cases, massive speedup is achieved via parallelism. (b) The training error at the end of each epoch of SVM training on RCV1 for the averaging algorithm [27]. (c) Speedup achieved over serial method for various levels of delays (measured in nanoseconds).

What if the gradients are slow? As we saw with the DBLIFE data set, the RR method does get a nearly linear speedup when the gradient computation is slow. This raises the question whether RR ever outperforms HOGWILD! for slow gradients. To answer this question, we ran the RCV1 experiment again and introduced an artificial delay at the end of each gradient computation to simulate a slow gradient. In Figure 3(c), we plot the wall clock time required to solve the SVM problem as we vary the delay for both the RR and HOGWILD! approaches.

Notice that HOGWILD! achieves a greater decrease in computation time across the board. The speedups for both methods are the same when the delay is few milliseconds. That is, if a gradient takes longer than one millisecond to compute, RR is on par with HOGWILD! (but not better). At this rate, one is only able to compute about a million stochastic gradients per hour, so the gradient computations must be very labor intensive in order for the RR method to be competitive.

7 Conclusions

Our proposed HOGWILD! algorithm takes advantage of sparsity in machine learning problems to enable near linear speedups on a variety of applications. Empirically, our implementations outperform our theoretical analysis. For instance, ρ is quite large in the RCV1 SVM problem, yet we still obtain significant speedups. Moreover, our algorithms allow parallel speedup even when the gradients are computationally intensive.

Our HOGWILD! schemes can be generalized to problems where some of the variables occur quite frequently as well. We could choose to not update certain variables that would be in particularly high contention. For instance, we might want to add a bias term to our Support Vector Machine, and we could still run a HOGWILD! scheme, updating the bias only every thousand iterations or so.

For future work, it would be of interest to enumerate structures that allow for parallel gradient computations with no collisions at all. That is, It may be possible to bias the SGD iterations to completely avoid memory contention between processors. An investigation into such biased orderings would enable even faster computation of machine learning problems.

Acknowledgements

BR is generously supported by ONR award N00014-11-1-0723 and NSF award CCF-1139953. CR is generously supported by the Air Force Research Laboratory (AFRL) under prime contract no. FA8750-09-C-0181, the NSF CAREER award under IIS-1054009, ONR award N000141210041, and gifts or research awards from Google, LogicBlox, and Johnson Controls, Inc. SJW is generously supported by NSF awards DMS-0914524 and DMS-0906818 and DOE award DE-SC0002283. Any opinions, findings, and conclusion or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of any of the above sponsors including DARPA, AFRL, or the US government.

References

[1] Max-flow problem instances in vision. From `http://vision.csd.uwo.ca/data/maxflow/`.

[2] K. Asanovic and *et al*. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[3] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, 2nd edition, 1999.

[4] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, Belmont, MA, 1997.

[5] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, 2008.

[6] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, 2004.

[7] G. Călinescu, H. Karloff, and Y. Rabani. An improved approximation algorithm for multiway cut. In *Proceedings of the thirtieth annual ACM Symposium on Theory of Computing*, pages 48–52, 1998.

[8] E. Candès and B. Recht. Exact matrix completion via convex optimization. *Foundations of Computational Mathematics*, 9(6):717–772, 2009.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal distributed online prediction using mini-batches. Technical report, Microsoft Research, 2011.

[11] A. Doan. `http://dblife.cs.wisc.edu`.

[12] J. Duchi, A. Agarwal, and M. J. Wainwright. Distributed dual averaging in networks. In *Advances in Neural Information Processing Systems*, 2010.

[13] S. H. Fuller and L. I. Millett, editors. *The Future of Computing Performance: Game Over or Next Level*. Committee on Sustaining Growth in Computing Performance. The National Academies Press, Washington, D.C., 2011.

[14] T. Joachims. Training linear svms in linear time. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*, 2006.

[15] J. Langford. `https://github.com/JohnLangford/vowpal_wabbit/wiki`.

[16] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *Advances in Neural Information Processing Systems*, 2009.

[17] J. Lee, , B. Recht, N. Srebro, R. R. Salakhutdinov, and J. A. Tropp. Practical large-scale optimization for max-norm regularization. In *Advances in Neural Information Processing Systems*, 2010.

[18] T. Lee, Z. Wang, H. Wang, and S. Hwang. Web scale entity resolution using relational evidence. Technical report, Microsoft Research, 2011. Available at `http://research.microsoft.com/apps/pubs/default.aspx?id=145839`.

[19] D. Lewis, Y. Yang, T. Rose, and F. Li. RCV1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5:361–397, 2004.

[20] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proceedings of VLDB*, 2010.

[21] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on Optimization*, 19(4):1574–1609, 2009.

[22] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. Technical report, 2011. `arxiv.org/abs/1106.5730`.

[23] B. Recht, M. Fazel, and P. Parrilo. Guaranteed minimum rank solutions of matrix equations via nuclear norm minimization. *SIAM Review*, 52(3):471–501, 2010.

[24] S. Shalev-Shwartz and N. Srebro. SVM Optimization: Inverse dependence on training set size. In *Proceedings of the 25th Internation Conference on Machine Learning*, 2008.

[25] N. Srebro, J. Rennie, and T. Jaakkola. Maximum margin matrix factorization. In *Advances in Neural Information Processing Systems*, 2004.

[26] J. Tsitsiklis, D. P. Bertsekas, and M. Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on Automatic Control*, 31(9):803–812, 1986.

[27] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. *Advances in Neural Information Processing Systems*, 2010.