PARALLEL STOCHASTIC GRADIENT DESCENT WITH SOUND COMBINERS

Saeed Maleki, Madanlal Musuvathi & Todd Mytkowicz
Microsoft Research
One Microsoft Way
Redmond, Washington
{saemal,madanm,toddm}@microsoft.com

Yufei Ding
North Carolina State University
3511 Ivy Commons Drive
Raleigh, North Carolina
yding8@ncsu.edu

ABSTRACT

Stochastic gradient descent (SGD) is a well-known method for regression and classification tasks. However, it is an inherently sequential algorithm — at each step, the processing of the current example depends on the parameters learned from the previous examples. Prior approaches to parallelizing SGD, such as HOGWILD! and ALLREDUCE, do not honor these dependences across threads and thus can potentially suffer poor convergence rates and/or poor scalability. This paper proposes SYMSGD, a parallel SGD algorithm that retains the sequential semantics of SGD in expectation. Each thread in this approach learns a local model and a *probabilistic model combiner* that allows the local models to be combined to produce the same result as what a sequential SGD would have produced, in expectation. This SYMSGD approach is applicable to any linear learner whose update rule is linear. This paper evaluates SYMSGD's accuracy and performance on 9 datasets on a shared-memory machine shows up-to $13\times$ speedup over our heavily optimized sequential baseline on 16 cores.

1 INTRODUCTION

Stochastic Gradient Descent (SGD) is an effective method for many regression and classification tasks. It is a simple algorithm with few hyper-parameters and its convergence rates are well understood both theoretically and empirically. However, its performance scalability is severely limited by its inherently sequential computation. SGD iteratively processes its input dataset where the computation at each iteration depends on the model parameters learned from the previous iteration.

Current approaches for parallelizing SGD do not honor this inter-step dependence across threads. Each thread learns a local model independently and combine these models in ways that can break sequential behavior. For instance, threads in HOGWILD! Recht et al. (2011) racily update a shared global model without holding any locks. In parameter-server Li et al. (2014a), each thread (or machine) periodically sends its model deltas to a server that applies them to a global model. In ALLREDUCE Agarwal et al. (2014), threads periodically reach a barrier where they compute a weighted-average of the local models. Obviously, these approaches can produce a model that is potentially different from what a sequential SGD would have produced on these examples. Our experiments show that all these algorithms either do not scale or their accuracy on the same number of examples falls short of a sequential baseline.

To address this problem, this paper presents SYMSGD, a parallel SGD algorithm that seeks to retain its sequential semantics. The key idea is for each thread to generate a *sound combiner* that allows the local models to be combined into a model that is the same as the sequential model. This paper describes a method for generating sound combiners for a class of SGD algorithms in which the inter-step dependence is linear in the model parameters. This class includes linear regression, linear regression with L2 regularization, and polynomial regression. While logistic regression is not in this class, our experiments show that linear regression performs equally well in classification tasks as logistic regression for the datasets studied in this paper. Also, this approach works even if the SGD computation is non-linear on the input examples and other parameters such as the learning rate; only the dependence on model parameters has to be linear.

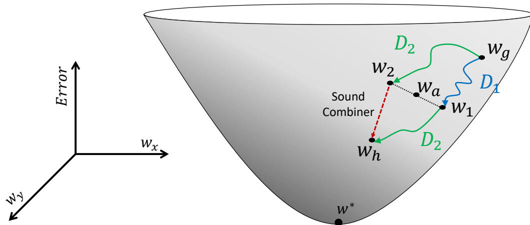


Figure 1: Convex error function for a two-dimensional feature space.

Generating sound combiners can be expensive. SYMSGD uses random projection techniques to reduce this overhead but still retaining sequential semantics in expectation. We call this approach *probabilistically sound combiners*. Even though SYMSGD is expected to produce the same answer as the sequential SGD, controlling the variance introduced by the random projection requires care — a large variance can result in reduced accuracy. This paper describes the factors that affect this variance and explores the ensuing design trade-offs.

The resulting algorithm is fast, scales well on multiple cores, and achieves the same accuracy as sequential SGD on sparse and dense datasets. When compared to our optimized sequential baseline, SYMSGD achieves a speedup of $3.5\times$ to $13\times$ on 16 cores, with the algorithm performing better for denser datasets. Moreover, the cost of computing combiners can be efficiently amortized in a multiclass regression as a single combiner is sufficient for all of the classes. Finally, SYMSGD (like ALLREDUCE) is deterministic, producing the same result for a given dataset, configuration, and random seed. Determinism greatly simplifies the task of debugging and optimizing learning.

2 SOUND AND PROBABILISTIC MODEL COMBINERS

Stochastic gradient descent (SGD) is a robust method for finding the parameters of a model that minimize a given error function. Figure 1 shows an example of a (convex) error function over two dimensions x and y reaching the minimum at parameter w^*. SGD starts from some, not necessarily optimal, parameter w_g (as shown in Figure 1), and repeatedly modifies w by taking a step along the gradient of the error function for a randomly selected example at the current w. The magnitude of the step is called the learning rate and is usually denoted by α. The gradient computed from one example is not necessarily the true gradient at w. Nevertheless, SGD enjoys robust convergence behavior by moving along the "right" direction over a large number of steps. This is shown pictorially in Figure 1, where SGD processes examples in dataset D_1 to reach w_1 from w_g. Subsequently, SGD starts from w_1 and processes a different set D_2 to reach w_h. There is a clear dependence between the processing of D_1 and the processing of D_2 — the latter starts from w_1, which is only determined after processing D_1. Our goal is to parallelize SGD despite this dependence.

State of the art parallelization techniques such as HOGWILD! and ALLREDUCE approach this problem by processing D_1 and D_2 starting from the same model w_g (let us assume that there only two processors for now), and respectively reaching w_1 and w_2. Then, they combine their local models into a global model, but do so in an ad-hoc manner. For instance, ALLREDUCE computes a weighted average of w_1 and w_2, where the per-feature weights are chosen so as to prefer the processor that has larger update for that feature. This weighted average is depicted pictorially as w_a. Similarly, in HOGWILD!, the two processors race to update the global model with their respective local model without any locking. (HOGWILD! performs this udpate after every example, thus the size of D_1 and D_2 is 1.) Both approaches do not necessarily reach w_h, the model that a sequential SGD would have reached on D_1 and D_2. While SGD is algorithmically robust to errors, such ad-hoc combinations can result in slower convergence or poor performance, as we demonstrate in Section 4.

Sound Combiner: The goal of this paper is to *soundly* combine local models. Looking at Figure 1, a sound combiner combines local models w_1 and w_2, respectively generated from datasets D_1 and D_2, into a global model w_h that is guaranteed to be the same as the model achieved by the sequential SGD processing D_1 and then D_2. In effect, a sound combiner allows us to parallelize the sequential computation without changing its semantics.

If we look at the second processor, it starts its computation at w_g, while in a sequential execution it would have started at w_1, the output of the first processor. To obtain sequential semantics, we need to "adjust" its computation from w_g to w_1. To do so, the second processor performs its computation starting from $w_g + \Delta w$, where Δw is an *unknown* symbolic vector. This allows the second processor to both compute a local model (resulting from the concrete part) and a sound combiner (resulting from the symbolic part) that accounts for changes in the initial state. Once both processors are done learning, second processor finds w_h by setting Δw to $w_1 - w_g$ where w_1 is computed by the first processor. This parallelization approach of SGD can be extended to multiple processors where all processor produce a local model and a combiner (except for the first processor) and the local models are combined sequentially using the combiners.

When the update to the model parameters is linear in a SGD computation, then the dependence on the unknown Δw can be concisely represented by a *combiner matrix*, as formally described in Section 3. Many interesting machine learning algorithms, such as linear regression, linear regression with L2 regularization, and polynomial regression already have linear update to the model parameters (but not necessarily linear on the input example).

Probabilistically Sound Combiner: The main problem with generating a sound combiner is that the combiner matrix has as many rows and columns as the total number of features. Thus, it can be effectively generated only for datasets with modest number of features. Most interesting machine learning problems involve learning over tens of thousands to billions of features, for which maintaining a combiner matrix is clearly not feasible.

We solve this problem through dimensionality reduction. Johnson-Lindenstrauss (JL) lemma Johnson & Lindenstrauss (1984) allows us to project a set of vectors from a high-dimensional space to a random low-dimensional space while preserving distances. We use this property to reduce the size of the combiner matrix without losing the fidelity of the computation — our parallel algorithm produces the same result as the sequential SGD in expectation.

Of course, a randomized SGD algorithm that generates the exact result in expectation is only useful if the resulting variance is small enough to maintain accuracy and the rate of convergence. We observe that for the variance to be small, the combiner matrix should have small singular values. Interestingly, the combiner matrix resulting from SGD is dominated by the diagonal entries as the learning rate has to be small for effective learning. We use this property to perform the JL projection only after *s*ubtracting the identity matrix. Also, other factors that control the singular values are the learning rate, number of processors, and the frequency of combining local models. This paper explores this design space and demonstrates the feasibility of efficient parallelization of SGD that retains the convergence properties of sequential SGD while enjoying parallel scalability.

3 PARALLEL SYMSGD ALGORITHM

Consider a training dataset $(X_{n\times f}, y_{n\times 1})$, where f is the number of features, n is the number of examples in the dataset, the ith row of matrix X, X_i, represents the features of the ith example, and y_i is the dependent value (or label) of that example. A linear model seeks to find a

$$w^* = \arg\min_{w\in\mathbb{R}^f} \sum_{i=0}^{n} Q(X_i \cdot w, y_i)$$

that minimizes an error function Q. For linear regression, $Q(X_i \cdot w, y_i) = (X_i \cdot w - y_i)^2$. When (X_i, y_i) is evident from the context, we will simply refer to the error function as $Q_i(w)$.

SGD iteratively finds w^* by updating the current model w with a gradient of $Q_r(w)$ for a randomly selected example r. For the linear regression error function above, this amounts to the update

$$w_i = w_{i-1} - \alpha\nabla Q_r(w_{i-1}) = w_{i-1} - \alpha(X_r \cdot w_{i-1} - y_r)X_r^T \quad (1)$$

Here, α is the *learning rate* that determines the magnitude of the update along the gradient. As it is clear from this equation, w_i is dependent on w_{i-1} which creates a loop-carried dependence and consequently makes parallelization of SGD across iterations using naïve approaches impossible.

The complexity of SGD for each iteration is as follows. Assume that X_r has z non-zeros. Therefore, the computation in Equation 1 requires $O(z)$ amount of time for the inner product computation,

$X_r \cdot w_{i-1}$, and the same $O(z)$ amount of time for scalar-vector multiplication, $\alpha(X_r \cdot w_{i-1} - y_r)X_r^T$. If the updates to the weight vector happen in-place meaning that w_i and w_{i-1} share the same memory location, the computation in Equation 1 takes $O(z)$ amount of time.

3.1 Symbolic Stochastic Gradient Descent

This section explains a new approach to parallelize the SGD algorithm despite its loop-carried dependence. As shown in Figure 1, the basic idea is to start each processor (except the first) on a concrete model w along with a symbolic unknown Δw that captures the fact that the starting model can change based on the output of the previous processor. If the dependence on Δw is linear during an SGD update, which is indeed the case for linear regression, then the symbolic dependence on Δw on the final output can be captured by an appropriate matrix $M_{a \to b}$ that is a function of the input examples $X_a, \ldots, X_b$ processed ($y_a, \ldots, y_b$ do not affect this matrix). Specifically, as Lemma A.1 in the Appendix shows, this *combiner matrix* is given by

$$M_{a \to b} = \prod_{i=b}^{a} (I - \alpha X_i^T \cdot X_i) \tag{2}$$

In effect, the combiner matrix above is the symbolic representation of how a Δw change in the input will affect the output of a processor. $M_{a \to b}$ is referred by M when the inputs are not evident.

The parallel SGD algorithm works as follows (see Figure 1). In the *learning* phase, each processor i starting from w_0, computes both a local model l_i and a combiner matrix M_i. In a subsequent *reduction* phase, each processor in turn computes its true output using

$$w_i = l_i + M_i \cdot (w_{i-1} - w_0) \tag{3}$$

Lemma A.1 ensures that this combination of local models will produce the same output as what these processors would have generated had they run sequentially. We call such combiners *sound*.

One can compute a sound model combiner for other SGD algorithms provided the loop-carried dependence on w is linear. In other words, there should exist a matrix A_i and vector b_i in iteration i such that $w_i = A_i \cdot w_{i-1} + b_i$. Note that A_i and b_i can be nonlinear in terms of input datasets.

3.2 Dimensionality Reduction of a Sound Combiner

The combiner matrix M generate above can be quite large and expensive to compute. The sequential SGD algorithm maintains and updates the weight vector w, and thus requires $O(f)$ space and time, where f is the number of features. In contrast, M is a $f \times f$ matrix and consequently, the space and time complexity of parallel SGD is $O(f^2)$. In practice, this would mean that we would need $O(f)$ processors to see constant speedups, an infeasible proposition particularly for datasets that can have thousands if not millions of features.

SymSGD resolves this issue by projecting M into a smaller space while maintaining its fidelity. This projection is inspired by the Johnson-Lindenstrauss (JL) lemma Johnson & Lindenstrauss (1984) and follows the treatment of Achlioptas Achlioptas (2001).

Lemma 3.1. [1] *Let A be a random $f \times k$ matrix with*

$$a_{ij} = d_{ij}/\sqrt{k}$$

where a_{ij} is the element of A at the ith row and j^{th} column and d_{ij} is independently sampled from a random distribution D with $\mathbf{E}[D] = 0$ and $\mathrm{Var}[D] = 1$. Then

$$\mathbf{E}[A \cdot A^T] = I_{f \times f}$$

The matrix A from Lemma 3.1 projects from $\mathbb{R}^f \to \mathbb{R}^k$ where k can be much smaller than f. This allows us to *approximate* Equation 3 as

$$w_i \approx l_i + M_i \cdot A \cdot A^T (w_{i-1} - w_0) \tag{4}$$

[1] See proof in Appendix A.2.

Lemma 3.1 guarantees that the approximation above is unbiased.

$$\mathbf{E}[l_i + M_i \cdot A \cdot A^T(w_{i-1} - w_0)] = l_i + M_i \cdot \mathbf{E}[A \cdot A^T](w_{i-1} - w_0) = w_i$$

This allows an efficient algorithm that only computes the projected version of the combiner matrix while still producing the same answer as the sequential algorithm in expectation. We call such combiners *probabilistically sound*.

Algorithm 1: SYMSGD learning a local model and a model combiner.

```
1  <vector,matrix,matrix> SymSGD(
2      float α, vector: w₀, X₁..Xₙ,
3      scalar: y₁..yₙ) {
4    vector w = w₀;
5    matrix A = (1/√k) random(D,f,k);
6    matrix M_A = A;
7    for i in (1..n) {
8      w = w - α(Xᵢ·w - yᵢ)Xᵢᵀ;
9      M_A = M_A - α Xᵢ·(XᵢᵀM_A); }
10   return <w,M_A,A>; }
```

Algorithm 2: SYMSGD combining local models using model combiners.

```
1  vector SymSGDCombine(vector w₀,
2      vector w, vector l,
3      matrix M_A, matrix A) {
4    parallel {
5      matrix N_A = M_A - A;
6      w = l+w-w₀+N_A·Aᵀ(w-w₀);
7    }
8    return w; }
```

Algorithm 1 shows the resulting symbolic SGD learner. The `random` function in line 5 returns a $\mathtt{f} \times \mathtt{k}$ matrix with elements chosen independently from the random distribution `D` according to Lemma 3.1. When compared to the sequential SGD, the additional work is the computation of $\mathtt{M_A}$ in Line 9. It is important to note that this algorithm maintains the invariant that $\mathtt{M_A} = \mathtt{M} \cdot \mathtt{A}$ at every step. This projection incurs a space and time overhead of $O(z \times k)$ where z is the number of non-zeros in $\mathtt{X_i}$. This overhead is acceptable for small k. Algorithm 2 combines the resulting probabilistically sound combiners, but additionally employs the optimizations discussed below.

Note that the correctness and performance of SYMSGD do not depend on the sparsity of a dataset and as Section 4 demonstrates, it works for very sparse and completely dense datasets. Also, note that $\mathtt{X_1}, \ldots, \mathtt{X_n}$ may contain a subset of size `f'` of all `f` features. Our implementation of Algorithm 1 takes advantage of this property and allocates and initializes `A` for only the observed features. This optimization is omitted from the pseudo code in Algorithm 1 for the sake of simplicity.

3.3 Controlling the Variance

While the dimensionality reduction discussed above is *expected* to produce the right answer, this is useful only if the variance of the approximation is acceptably small. Computing the variance is involved and is discussed in the associated technical report SymSGDTR. But we discuss the main result that motivates the rest of the paper.

Consider the approximation of $M \cdot \Delta w$ with $v = M \cdot A \cdot A^T \cdot \Delta w$. Let $\mathbb{C}(v)$ be the covariance matrix of v. The trace of the covariance matrix $tr(\mathbb{C}(v))$ is the sum of the variance of individual elements of v. Let $\lambda_i(M)$ by the ith eigenvalue of M and $\sigma_i(M) = \sqrt{\lambda_i(M^T M)}$ the ith singular value of M. Let $\sigma_{max}(M)$ be the maximum singular value of M. Then the following holds SymSGDTR:

$$\frac{\|\Delta w\|_2^2}{k} \sum_i \sigma_i^2(M) \leq tr(\mathbb{C}(v)) \leq \frac{\|\Delta w\|_2^2}{k} (\sum_i \sigma_i^2(M) + \sigma_{max}^2(M))$$

The covariance is small if k, the dimension of the projected space, is large. But increasing k proportionally increases the overhead of the parallel algorithm. Similarly, covariance is small if the projection happens on small Δw. Looking at Equation 4, this means that w_{i-1} should be as close to w_0 as possible, implying that processors should communicate frequently enough such that their models are roughly in sync. Finally, the singular values of M should be as small as possible. The next section describes a crucial optimization that achieves this.

Taking Identity Off: Expanding Equation 2, we see that the combiner matrices are of the form

$$I - \alpha R_1 + \alpha^2 R_2 - \alpha^3 R_3 + \cdots$$

where R_i matrices are formed from the sum of products of $X_j \cdot X_j^T$ matrices. Since α is a small number, this sum is dominated by I. In fact, for a combiner matrix M generated from n examples, $M - I$ has at most n non-zero singular values SymSGDTR. We use these observation to lower the variance of dimensionality reduction by projecting matrix $N = M - I$ instead of M. Appendix A.3 empirically shows the impact of this optimization. Rewriting Equations 3 and 4, we have

$$\begin{aligned} w_i &= l_i + (N_i + I) \cdot (w_{i-1} - w_0) \\ &= l_i + w_{i-1} - w_0 + N_i \cdot (w_{i-1} - w_0) \\ &\approx l_i + w_{i-1} - w_0 + N_i \cdot A \cdot A^T \cdot (w_{i-1} - w_0) \end{aligned} \tag{5}$$

Lemma 3.1 guarantees that the approximation above is unbiased. Algorithm 2 shows the pseudo code for the resulting probabilistically sound combination of local models. The function `SymSGDCombine` is called iteratively to combine the model of the first processor with the local models of the rest. Note that each model combination is executed in parallel (Line 4) by parallelizing the underlying linear algebra operations.

An important factor in controlling the singular values of N is the frequency of model combinations which is a tunable parameter in SYMSGD. As it is shown in Appendix A.3, the fewer the number of examples learned, the smaller the singular values of N and the less variance (error) in Equation 5.

Implementation For the implementation of `SymSGD` function, matrix M and weight vector w are stored next to each other. This enables better utilization of vector units in the processor and improves the performance of our approach significantly. Also, most of datasets are sparse and therefore, `SGD` and `SymSGD` only copy the observed features from `w0` to their learning model `w`. Moreover, for the implementation of matrix `A`, we used Achlioptas (2001) theorem to minimize the overhead of creating A. In this approach, each element of A is independently chosen from $\{\frac{1}{3}, -\frac{1}{3}, 0\}$ with probability $\{\frac{1}{6}, \frac{1}{6}, \frac{2}{3}\}$, respectively.

4 EVALUATION

All experiments described in this section were performed on an Intel Xeon E5-2630 v3 machine clocked at 2.4 GHz with 256 GB of RAM. The machine has two sockets with 8 cores each, allowing us to study the scalability of the algorithms across sockets. We disabled hyper-threading and turbo boost. We also explicitly pinned threads to cores in a compact way which means that thread $i + 1$ was placed as close as possible to thread i. The machine runs Windows 10. All of our implementations were compiled with Intel C/C++ compiler 16.0 and relied heavily on OpenMP primitives for parallelization and MKL for efficient linear algebra computations. And, finally, to measure runtime, we use the average of five independent runs on an otherwise idle machine.

There are several algorithms and implementations that we used for our comparison: Vowpal Wabbit Langford et al. (2007), a widely used public library, Baseline, a fast sequential implementation, HW-Paper, the implementation from Recht et al. (2011), HW-Release, an updated version, HogWild, which runs Baseline in multiple threads without any synchronization, and ALLREDUCE, the implementation from Agarwal et al. (2014). Each of these algorithms have different parameters and settings and we slightly modified to ensure a fair comparison; see Appendix A.4 for more details.

When studying the scalability of a parallel algorithm, it is important to compare the algorithms against an efficient baseline Bailey (1991); McSherry et al. (2015). Otherwise, it is empirically not possible to differentiate between the scalability achieved from the parallelization of the inefficiencies and the scalability inherent in the algorithm. We spent a significant effort to implement a well-tuned sequential algorithm which we call Baseline in our comparisons. Baseline is between 1.97 to 7.62 (3.64 on average) times faster than Vowpal Wabbit and it is used for all speedup graphs in this paper.

Datasets Table 1 describes the datasets used for evaluation. The number of features, training instances, test instances, classes and the sparsity of each dataset is shown in Table 1. We used Vowpal Wabbit with the configurations discussed in Appendix A.4 to measure the maximum accuracies that can be achieved using linear and logistic regression and the result is presented in columns 8 and 9 of Table 1. In the case of aloi dataset, even after 500 passes (the default for our evaluation was 100 passes) the accuracies did not saturate to the maximum possible and we reported that both linear and logistic achieved at least 80% accuracy. The last two columns show the maximum speedup of SYMSGD and HOGWILD! over the baseline.

Benchmarks	#Features	#Training	#Test	#Classes	Sparsity (%)	Logistic	Linear	SYMSGD	HOGWILD!
aloi	128	75463	32537	1000	76.03	> 80	>80	12.94	3.12
mnist8m	784	5668902	2431098	10	74.49	86.697	86.518	7.20	2.65
url	3231961	1677282	718848	2	99.996	99.135	99.144	2.55	4.05
rcv1.multiclass	47236	15564	518571	53	99.86	86.538	86.523	7.43	5.63
epsilon	2000	400000	100000	2	0.00	89.694	89.689	8.05	3.93
sector	55197	6412	3207	105	99.70	85.594	85.75	8.36	4.49
mnist	780	60000	10000	10	80.78	91.91	91.82	13.29	2.62
news20	62061	15935	3993	20	99.87	83.045	83.747	3.55	5.13
rcv1.binary	20242	677399	47236	2	99.84	96.201	96.321	3.50	2.91

Table 1: Datasets used for evaluation with their settings, maximum accuracies using logistic and linear regression and maximum speedup using SYMSGD and HOGWILD!. Red speedup numbers represent the only cases where HOGWILD! performs faster than SYMSGD.

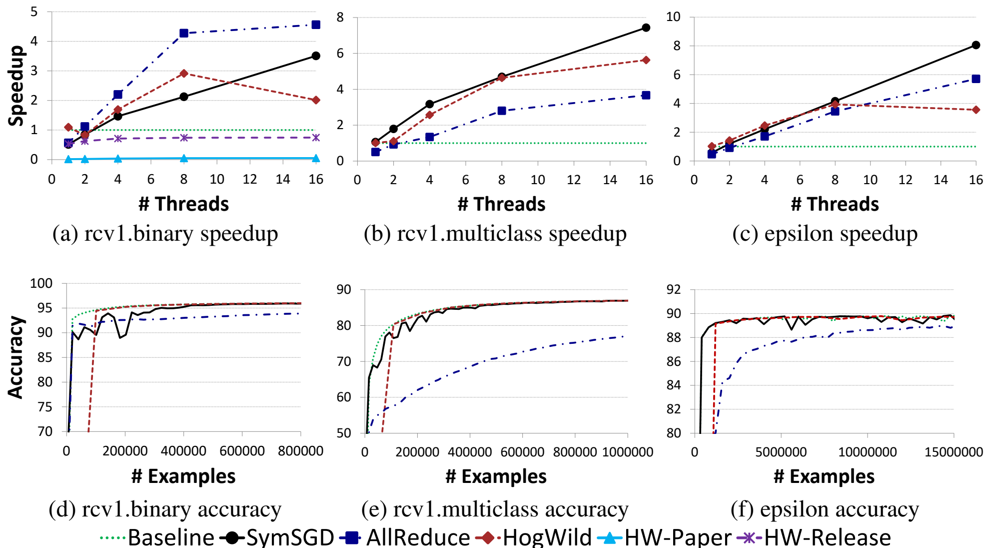


Figure 2: Speedup and accuracy comparison

Parameters Hyper-parameters searching is essential for performance and accuracy. The learning rate, α, for each dataset was selected by searching for a constant value among $\{.5, .05, .005, \dots\}$ where Baseline reached close to maximum accuracy for each benchmark. The parameters for the projection size, k, and the frequency of model combination were searched to pick the best performing configuration. The parameters for ALLREDUCE were similarly searched.

4.1 RESULTS

Figure 2 shows the accuracy and speedup measurements on three benchmarks: rcv1.binary, a sparse binary dataset, rcv1.multiclass, a sparse multiclass dataset, and epsilon, a dense binary dataset. The results for the other six benchmarks are presented in Appendix A.5.

Sparse Binary, rcv1.binary: Figure 2a compares the scalability of all the algorithms studied in this paper. HW-Paper is around six times slower than HW-Release. While this could potentially be a result of us running HW-Release on a Ubuntu VM, our primary aim of this comparison was to ensure that HogWild is a competitive implementation of HOGWILD!. Thus, we remove HW-Paper and HW-Release in our subsequent comparisons.

SYMSGD is half as slow as the Baseline on one thread as it performs lot more computation, but scales to a $3.5\times$ speedup to 16 cores. Note, this represents a roughly $7\times$ strong-scaling speedup with respect to its own performance on one thread. Analysis of the hardware performance counters shows the current limit to SYMSGD's scalability arises from load-imbalance across barrier synchronization, which provides an opportunity for future improvement.

Figure 2d shows the accuracy as a function of the number of examples processed by different algorithms. SYMSGD "stutters" at the beginning, but it too matches the accuracy of Baseline. The initial stuttering happens because the magnitude of the local models on each processor are large during the first set of examples. This directly affect the variance of the combiner matrix approximation. However, as more examples are given to SYMSGD, the magnitude of the local models are smaller and thus SYMSGD better matches the Baseline accuracy. One way to avoid this stuttering is to combine models more frequently (lower variance) or running single threaded for the first few iterations.

HogWild does approach sequential accuracy, however, it does so at the cost of scalablity (i.e., see Figure 2a (a)). Likewise, ALLREDUCE scales slightly better but does so at the cost of accuracy.

Sparse Multiclass, rcv1.multiclass: Figure 2b shows the scalability on rcv1.multiclass. Since this is a multiclass dataset, SYMSGD is competitive with the baseline on one thread as it is able to amortize the combiner matrix computation across all of the classes (M is the same across different classes). Thus, it enjoys much better scalability of $7\times$ when compared to rcv1.binary. HogWild scales similar to SYMSGD up-to 8 threads but suffers when 16 threads across multiple sockets are used. Figure 2e shows that SYMSGD meets the sequential accuracy after an initial stutter. ALLREDUCE suffers from accuracy.

Dense Binary, epsilon: Figure 2c in Appendix A.5 shows that SYMSGD achieves a $7\times$ speedup over the baseline on 16 cores. This represents a $14\times$ strong scaling speedup over SYMSGD on one thread. As HOGWILD! is not designed for dense workloads, its speedup suffers when 16 cores across multiple sockets are used. This shows that SYMSGD scales to both sparse and dense datasets. Similarly, ALLREDUCE suffers from accuracy.

5 Related Work

Most schemes for parallelizing SGD learn local models independently and communicate to update the global model. The algorithms differ in how and how often the update is performed. These choices determine the applicability of the algorithm to shared-memory or distributed systems.

To the best of our knowledge, our approach is the only one that retain the semantics of the sequential SGD algorithm. While some prior work provides theoretical analysis of the convergence rates that justify a specific parallelization, convergence properties of SYMSGD simply follow from the sequential SGD algorithm. On the other hand, SYMSGD is currently restricted to class of SGD computations where the inter-step dependence is linear in the model parameters.

Given a tight coupling of the processing units, Langford et al. Langford et al. (2009) suggest on a round-robin scheme to update the global model allowing for some staleness. However, as the SGD computation per example is usually much smaller when compared to the locking overhead, HOGWILD! Recht et al. (2011) improves on this approach to perform the update in a "racy" manner. While HOGWILD! is theoretically proven to achieve good convergence rates provided the dataset is sparse enough and the processors update the global model fast enough, our experiments show that the generated cache-coherence traffic limits its scalability particularly across multiple sockets. Moreover, as HOGWILD! does not update the model atomically, it potentially loses correlation among more frequent features resulting in loss of accuracy. Lastly, unlike SYMSGD, which works for both sparse and dense datasets, HOGWILD! is expclitly designed for sparse data. Recently, Sallinen et al. (2016) proposed applying lock-free HOGWILD! approach to mini-batch. However, mini-batch converges slower than SGD and also they did not study multi-socket scaling.

Zinkevich et al. Zinkevich et al. (2010) propose a MapReduce-friendly framework for SGD. The basic idea is for each machine/thread to run a sequential SGD on its local data. At the end, the global model is obtained by averaging these local models. Alekh et al. Agarwal et al. (2014) extend this approach by using MPI_AllReduce operation. Additionally, they use the adagrad Duchi et al. (2011) approach for the learning rates at each node and use weighted averaging to combine local models with processors that processed a feature more frequently having a larger weight. Our experiments on our datasets and implementation shows that it does not achieve the sequential accuracy.

Several distributed frameworks for machine learning are based on parameter server Li et al. (2014b;a) where clients perform local learning and periodically send the changes to a central pa-

rameter server that applies the changes. For additional parallelism, the models themselves can be split across multiple servers and clients only contact a subset of the servers to perform their updates.

6 CONCLUSION

With terabytes of memory available on multicore machines today, our current implementation has the capability of learning from large datasets without incurring the communication overheads of a distributed system. That said, we believe the ideas in this paper apply to distributed SGD algorithms and how to pursue in future work.

Many machine learning SGD algorithms require a nonlinear dependence on the parameter models. While SYMSGD does not directly apply to such algorithms, it is an interesting open problem to devise linear approximations (say using Taylor expansion) to these problems and subsequently parallelize with probabilistically sound combiners. This is an interesting study for future work.

REFERENCES

Dimitris Achlioptas. Database-friendly random projections. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pp. 274–281, New York, NY, USA, 2001. ACM. ISBN 1-58113-361-8. doi: 10.1145/375551.375608. URL http://doi.acm.org/10.1145/375551.375608.

Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *Journal of Machine Learning Research*, 15(1):1111–1133, 2014. URL http://dl.acm.org/citation.cfm?id=2638571.

David Bailey. Twelve ways to fool the masses. http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/twelve-ways.pd, 1991.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1953048.2021068.

Hogwild. Hogwild implemenations. http://i.stanford.edu/hazy/victor/Hogwild/, 2016. [accessed Sep 2016].

Intel. Intel math kernel library. https://software.intel.com/en-us/intel-mkl.

William Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in modern analysis and probability (New Haven, Conn., 1982)*, volume 26 of *Contemporary Mathematics*, pp. 189–206. American Mathematical Society, 1984.

John Langford, Lihong Li, and Alex Strehl. Vowpal Wabbit, 2007.

John Langford, Alexander Smola, and Martin Zinkevich. Slow learners are fast. *arXiv preprint arXiv:0911.0491*, 2009.

Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 583–598, Broomfield, CO, October 2014a. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

Mu Li, David G Andersen, Alex J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 27*, pp. 19–27. Curran Associates, Inc., 2014b. URL http://papers.nips.cc/paper/5597-communication-efficient-distributed-machine-learning-with-the-parameter-server.pdf.

Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association. URL https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pp. 693–701, 2011.

S. Sallinen, N. Satish, M. Smelyanskiy, S. S. Sury, and C. R. High performance parallel stochastic gradient descent in shared memory. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 873–882, May 2016. doi: 10.1109/IPDPS.2016.107.

SymSGDTR. Symsgd technical report. https://www.microsoft.com/en-us/research/publication/parallelizing-sgd-symbolically/, 2016. [accessed Sep 2016].

Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pp. 2595–2603, 2010.

A Appendix

A.1 Combiner Matrix

Lemma A.1. *If the SGD algorithm for linear regression processes examples $(X_a, y_a), (X_{a+1}, y_{a+1}), \ldots, (X_b, y_b)$ starting from model w_s to obtain w_b, then its outcome starting on model $w_s + \Delta w$ is given by $w_b + M_{a\to b} \cdot \Delta w$ where the combiner matrix $M_{a\to b}$ is given by*

$$M_{a\to b} = \prod_{i=b}^{a} (I - \alpha X_i^T \cdot X_i)$$

Proof. The proof follows from a simple induction. Starting from w_s, let the models computed by SGD after processing $(X_a, y_a), (X_{a+1}, y_{a+1}), \ldots, (X_b, y_b)$ respectively be w_a, $w_{a+1}, \ldots w_b$. Consider the base case of processing of (X_a, y_a). Starting from $w_s + \Delta w$, SGD computes the model w'_a using Equation 1 (reminder: $w_i = w_{i-1} - \alpha(X_i \cdot w_{i-1} - y_i)X_i^T$):

$$\begin{aligned}
w'_a &= w_s + \Delta w - \alpha(X_a \cdot (w_s + \Delta w) - y_a)X_a^T \\
&= w_s + \Delta w - \alpha(X_a \cdot w_s - y_a)X_a^T - \alpha(X_a \cdot \Delta w)X_a^T \\
&= w_s - \alpha(X_a \cdot w_s - y_a)X_a^T + \Delta w - \alpha(X_a \cdot \Delta w)X_a^T \\
&= w_a + \Delta w - \alpha(X_a \cdot \Delta w)X_a^T && (6) \\
&= w_a + \Delta w - \alpha X_a^T(X_a \cdot \Delta w) && (7) \\
&= w_a + \Delta w - \alpha(X_a^T \cdot X_a) \cdot \Delta w && (8) \\
&= w_a + (I - \alpha X_a^T \cdot X_a) \cdot \Delta w
\end{aligned}$$

Step 6 uses Equation 1, Step 7 uses the fact that $X_a \cdot \Delta w$ is a scalar (allowing it to be rearranged), and Step 8 follows from associativity property of matrix multiplication.

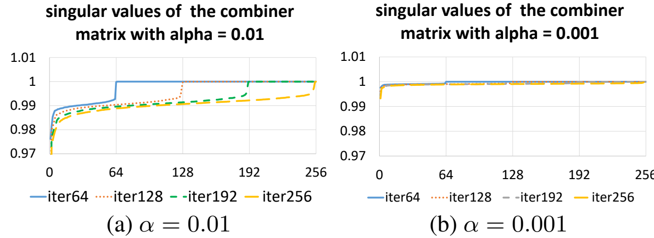
The induction is very similar and follows from replacing Δw with $M_{a\to i-1}\Delta w$ and the property that

$$M_{a\to i} = (I - \alpha X_i^T \cdot X_i) \cdot M_{a\to i-1}$$

□

A.2 Proof of Lemma 3.1

Proof. Let's call $B = A \cdot A^T$. Then b_{ij}, the element of B at row i and column j, is $\sum_s a_{is}a_{js}$. Therefore, $\mathbf{E}[b_{ij}] = \sum_{s=1}^{k} \mathbf{E}[a_{is}a_{js}] = (\frac{1}{\sqrt{k}})^2 \sum_{s=1}^{k} \mathbf{E}[d_{is}d_{js}] = \frac{1}{k}\sum_{s=1}^{k} \mathbf{E}[d_{is}d_{js}]$. For $i \neq j$,



(a) $\alpha = 0.01$ (b) $\alpha = 0.001$

Figure 3: Distribution of singular values of M for rcv1 binary dataset for $\alpha = 0.01$ and $\alpha = 0.001$.

$\mathbf{E}[b_{ij}] = \frac{1}{k}\sum_{s=1}^{k}\mathbf{E}[d_{is}]\mathbf{E}[d_{js}]$ because d_{ij} are chosen independently. Since $\mathbf{E}[D] = 0$ and $d_{is}, d_{js} \in D$, $\mathbf{E}[d_{is}] = \mathbf{E}[d_{js}] = 0$ and consequently, $\mathbf{E}[b_{ij}] = 0$. For $i = j$, $\mathbf{E}[b_{ii}] = \frac{1}{k}\sum_{s}\mathbf{E}[d_{is}d_{is}] = \frac{1}{k}\sum_{s}\mathbf{E}[d_{is}^2]$. Since $\mathbf{E}[D^2] = 1$ and $d_{is} \in D$, $\mathbf{E}[d_{is}^2] = 1$. As a result, $\mathbf{E}[b_{ii}] = \frac{1}{k}\sum_{s=1}^{k}\mathbf{E}[d_{is}^2] = \frac{1}{k}\sum_{s=1}^{k} 1 = 1$. $\square$

A.3 Empirical Evaluating Singular Values of M

Figure 3 empirically demonstrates the benefit of taking identity off. This figure plots the singular values of M for the rcv1.binary dataset (described in Section 4) after processing $64, 128, 192, 256$ examples for two different learning rates. As it can be seen, the singular values are close to 1. However, the singular values of $N = M - I$ are roughly the same as those of M minus 1 and consequently, are small. Finally, the smaller α, the closer the singular values of M are to 1 and the singular values of N are to 0. Also, note that the singular values of M decrease as the numbers of examples increase and therefore, the singular values of N increase. As a result, the more frequent the models are combined, the less variance (and error) is introduced into Equation 5.

A.4 Algorithm Details and Settings

This section provides details of all algorithms we used in this paper. Each algorithm required slight modification to ensure fair comparison.

Vowpal Wabbit: Vowpal Wabbit Langford et al. (2007) is one of the widely used public libraries for machine learning algorithms. We used this application as a baseline for accuracy of different datasets and as a comparison of logistic and linear regression and also an independent validation of the learners without any of our implementation bias. Vowpal Wabbit applies accuracy-improving optimizations such as adaptive and individual learning steps or per feature normalize updates. While all of these optimizations are applicable to SymSGD, we avoided them since the focus of this paper is the running time performance of our learner. The non-default flags that we used are: `--sgd`, `--power_t 0`, `--holdout_off`, `--oaa nc` for multiclass datasets where `nc` is the number of classes, `--loss_function func` where `func` is `squared` or `logistic`. For learning rate, we searched for α, the learning rate, in the set of $\{.1, .5, .01, .05, .001, .005, \dots\}$ and used `--learning_rate` α. We went through dataset 100 times for each dataset (`--passes 100`) and saved the learned model after each pass (`--save_per_pass`). At the end, for linear and logistic regressions, we reported the maximum accuracies achieved among different passes and different learning rates.

Baseline: Baseline uses a mixture of MKL Intel and manually vectorized implementations of linear algebra primitives in order to deliver the fastest performance. Baseline processes up-to 3.20 billion features per second at 6.4 GFLOPS.

Hogwild!: Hogwild! Recht et al. (2011) is a lock-free approach to parallelize SGD where multiple thread apply Equation 1 simultaneously. Although this approach may have race condition when two threads process instances with a shared feature but the authors discuss that this does not hurt the accuracy significantly for sparse datasets. There are multiple implementations of this approach that we studied and evaluated in this section. Below is a description of each:

• **HW-Paper:** This is the implementation used to report the measurements in Recht et al. (2011) which is publicly available Hogwild. This code implements SVM algorithm. Therefore, we modified the update rule to linear regression. The modified code was compiled and run on our Windows machine described above using an Ubuntu VM since the code is configured for Linux systems.

• **HW-Release:** This is an optimized implementation that the authors built after the HOGWILD! paper Recht et al. (2011) was published. Similar to HW-Paper, we changed the update rule accordingly and executed it on the VM.

• **HogWild:** We implemented this version which runs Baseline by multiple threads without any synchronizations. This code runs natively on Windows and enjoys all the optimizations applied to our Baseline such as call to MKL library and manual vectorization of linear algebra primitives.

ALLREDUCE: ALLREDUCE Agarwal et al. (2014) is an approach where each thread makes a copy from the global model and applies the SGD update rule to the local model for certain number of instances. Along with the local model, another vector g is computed which indicates the confidence in an update for the weight of a feature in the local model. After the learning phase, the local weight vectors are averaged based on the confidence vectors from each thread. We implemented this approach similarly using MKL calls and manual vectorization.

A.5 SPEEUPS ON REMAINING BENCHMARKS

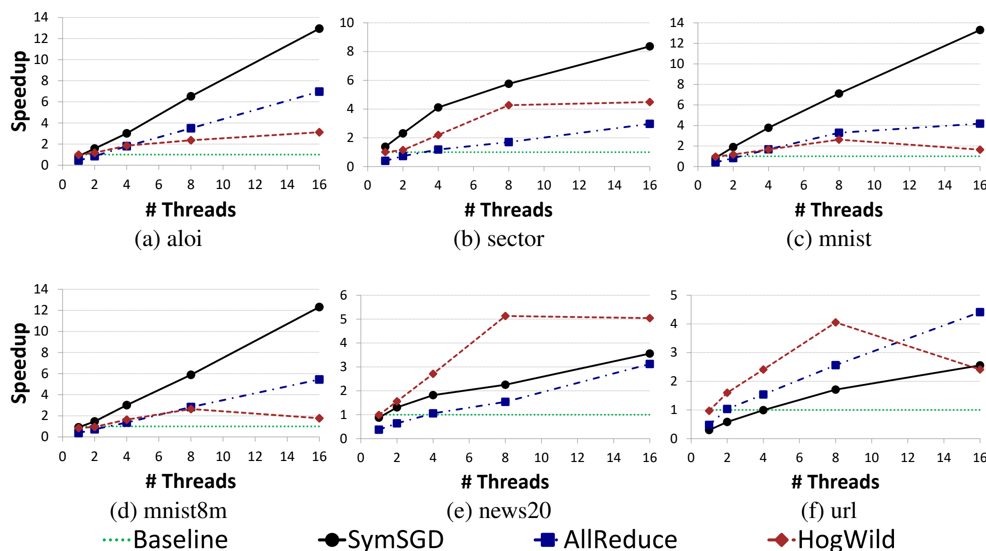


Figure 4: Speedups on remaining benchmarks