Communication-Efficient Distributed Stochastic Gradient Descent with Butterfly Mixing

Huasha Zhao
John F. Canny

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-96
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-96.html

May 11, 2012

Copyright © 2012, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Communication-Efficient Distributed Stochastic Gradient Descent with Butterfly Mixing

by Huasha Zhao

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John F. Canny
Research Advisor

(Date)

* * * * * * *

Professor Pieter Abbeel
Second Reader

(Date)

Communication-Efficient Distributed Stochastic Gradient Descent with Butterfly Mixing

Huasha Zhao
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Research Adviser: Professor John F. Canny

May 11, 2012

Abstract

Stochastic gradient descent is a widely used method to find locally-optimal models in machine learning and data mining. However, it is naturally a sequential algorithm, and parallelization involves severe compromises because the cost of synchronizing across a cluster is much larger than the time required to compute an optimal-sized gradient step. Here we explore butterfly mixing, where gradient steps are interleaved with the k stages of a butterfly network on 2^k nodes. Udp based butterfly mix steps should be extremely fast and failure-tolerant, and convergence is almost as fast as a full mix (AllReduce) on every step.

1 Introduction

We are entering the era of "big data"; exabytes of information are generated over the Internet on a daily basis [15]. People leverage on machine learning and data mining algorithms to explore these data and make sense of this explosive amount of information. As modern societies rely more and more heavily on these interpreted information from big data sets to make operation decisions, such as traffic optimization and advertisement placement, there is a pressing need to design efficient learning algorithms which can understand large-scale data in a fast and accurate manner.

Stochastic gradient descent is a popular algorithm which is suitable for a variety of data-driven learning tasks [8]. Stochastic gradient is simple, widely applicable and has proved to achieve reasonably high performance on large-scale machine learning problems [3]. However, the scalability of the algorithm is limited by its inherently sequential nature. The bottleneck of solving large-scale learning problems with stochastic gradient, like many other sequential algorithms, is oftentimes communication overhead instead of computation because synchronizations across machines are required before each learning step. A simple example illustrates the dilemma. At current commodity computational capacity, it takes 1ms to process every 100kBytes of data which is a typical size of optimal-sized gradient step. In contrast, the communication process in which gradients from different machines are exchanged takes up to 40ms for a mere 4 node cluster in a recent cluster implementation of MPI AllReduce [11].

Recognizing this problem, some prior works attempt to resolve this parallelization dilemma. Zinkevich et al. [19] proposes a simple algorithm in which multiple gradient descents run in parallels and their outputs are averaged in the end. However, the algorithm fails to reduce the variance of the gradient estimation, and therefore the final minimizer, and has been shown that in many problems there is no advantage to running this averaging method without communication [14]. Niu et al. [14] consider a lock-free approach to parallelizing stochastic gradient descent, but their focus is on multi-core settings instead of large clusters. MapReduce [5] has become very popular for distributed data processing, and Chu et al. [4] describes a general framework to run machine learning algorithms with this popular platform. Unfortunately, the MapReduce abstraction is ill-suited for iterative algorithms which are commonly used to solve machine learning problems. Researchers have also explored performance gain through developing new parallelization frameworks [18, 17, 13]; they are aiming at creating a resource-efficient programming environment that is friendly to parallel computing operations, and our system can be built on top of them.

More closely related to our work is that of Agarwal et al.[1] who generalizes MPI [9] AllReduce to a more data friendly Hadoop-compatible AllReduce communication framework. AllReduce computes the average of vectors sitting on each individual node and broadcast this average value with the help of a tree structure on communication nodes. The abstraction of AllReduce is naturally applicable to parallelize iterative optimization algorithms, and in general, any algorithms with the form of statistical query [10] can fit into this framework. However, each AllReduce operation takes up to $2k$ steps to reduce the vectors and distribute the final result for a cluster with 2^k worker nodes; the communication cost is unreasonably high and tends to dominate application execution times.

In this paper, we design and build a new abstraction called butterfly mixing that enables efficient parallelization of a variety of iterative learning algorithms in cluster settings. As a starting point we evaluate our algorithm on stochastic gradient method. Butterfly mixing interleaves communication with computation within a balanced butterfly reduce structure. Unlike AllReduce, learning parameter update proceeds as long as local agreements on average gradient are reached. As a result, communication cost is saved by a factor of $2k$, while convergence performance are not affected. That is to say, butterfly mixing remains all benefit of AllReduce, and at the same time, significantly improve its communication efficiency.

The benefit of butterfly mixing comes from two sources: first, variance of gradient estimation is significantly reduced by aggregating information before each update; second, local information can be propagated to the whole network within a small number of steps because of our balanced reduce structure, so that we can achieve the variance reduction through only local agreements at minimal cost. It is also worth noting that butterfly mixing is not specifically designed for stochastic gradient; it is applicable to any statistical query algorithms [10].

This paper is organized as follows. We begin with an overview of stochastic gradient methods in Section 2. Then, in Section 3 we discuss the main algorithms and performance evaluations are explained in Section 4. Finally, Section 5 concludes our work.

2 Training Models with Stochastic Gradient Descent

Stochastic gradient descent is a widely used method to find locally-optimal models in machine learning and data mining. Consider the problem of minimizing a loss function L written as a sum of differentiable functions $L^i : \mathbb{R}^d \mapsto \mathbb{R}$,

$$L(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} L^i(\mathbf{w}; \mathbf{x}^i, y^i), \tag{1}$$

This formulation arises from a variety of machine learning tasks. In practical settings, $\mathbf{w}$ is a $d-$dimensional weight vector to be estimated. There are in total n training examples, and $\mathbf{x}^i$ is the feature vector of the i^{th} example with the same dimension d, and y^i is the label or value associated with it. In most cases, the loss function L^i is homogeneous with respect to all examples.

Stochastic gradient can be used to minimize the loss function iteratively according to the formula,

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \gamma \hat{\nabla} L(\mathbf{w}(t)), \tag{2}$$

where γ is a sufficiently small step size and $\mathbf{w}(t)$ is iterative value of $\mathbf{w}$ which we also call the position of the system at time t; $\hat{\nabla} L(\mathbf{w}(t))$ is the noisy estimation of the true gradient at the current position.

The gradient is estimated by partial average of the loss function,

$$\hat{\nabla} L(\mathbf{w}(t)) = \frac{1}{m} \sum_{i=k_0}^{k_0+m-1} \nabla L^i(\mathbf{w}; \mathbf{x}^i, y^i) \tag{3}$$

where k_0 is a starting point which can be the index of the next unvisited data point. This update algorithm is also called mini-batch algorithm, and m is called the batch size of the algorithm. The larger m is, the more accurate gradient is estimated, and the more computation required by each individual update. This is another motivation of using parallelization, because clusters of machines can compute and aggregate the gradient collectively, so that the variance of the gradient step should be significantly reduced.

In most cases, preconditioning is necessary to ensure the fast convergence of the gradient method. The preconditioned stochastic gradient step takes the following form,

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \gamma H(t) \hat{\nabla} L(\mathbf{w}(t)), \tag{4}$$

where $H(t)$ is called preconditioner, which attempts to reduce the condition number of the system and, as a result, to increase the convergence rate [16]. Inverse Hessian would be a desirable preconditioner when dimension of the problem d is small, however, it can easily run into memory issues for a typical “big data” problem with moderately large dimensions.

A simple preconditioner, called diagonal preconditioning or Jacobi preconditioning, whose diagonal entries are identical to those of full preconditioner and all the other entries zero, provides similar convergence improvements [6]. Empirically, we use inverse feature frequencies as our diagonal preconditioner for the experiments of the paper.

2.1 Logistic Regression

Logistic regression model [8] is among the most successful classification algorithms, and is widely used for predicting the outcome of a categorical variable. We discuss in detail how stochastic gradient can be applied to solve logistic regression.

Logistic regression models the posterior probability of two classes via the inner product of weight vector $\mathbf{w}$ and feature vector $\mathbf{x}^i$,

$$Pr(y^i = 1|\mathbf{x}^i) = \frac{e^{\mathbf{w}^T \mathbf{x}^i}}{1 + e^{\mathbf{w}^T \mathbf{x}^i}} \tag{5}$$

$$Pr(y^i = 0|\mathbf{x}^i) = \frac{1}{1 + e^{\mathbf{w}^T \mathbf{x}^i}}. \tag{6}$$

Loss function is defined as the negative log-likelihood, which can be written as,

$$L(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} L^i(\mathbf{w}; \mathbf{x}^i, y^i) \tag{7}$$

$$\begin{aligned} L^i(\mathbf{w}; \mathbf{x}^i, y^i) &= -\frac{1}{n}\sum_{i=1}^{n} \left\{ \, y^i \log Pr(y^i = 1|\mathbf{x}^i) + (1-y^i)\log Pr(y^i = 0|\mathbf{x}^i) \right\} \\ &= \frac{1}{n}\sum_{i=1}^{n} \left\{ -y^i \mathbf{w}^T\mathbf{x}^i + \log(1 + e^{\mathbf{w}^T\mathbf{x}^i}) \right\} \end{aligned} \tag{8}$$

Finally, the gradient step to find optimal weight $\mathbf{w}$ in logistic regression model can be computed according to Equation (3-4) and

$$\nabla L^i(\mathbf{w}; \mathbf{x}^i, y^i) = \mathbf{x}^i(y^i - \frac{e^{\mathbf{w}^T\mathbf{x}^i}}{1 + e^{\mathbf{w}^T\mathbf{x}^i}}). \tag{9}$$

3 Butterfly Mixing Algorithm

In this section, we present our butterfly mixing algorithm, and briefly discuss its convergence and implementation concerns.

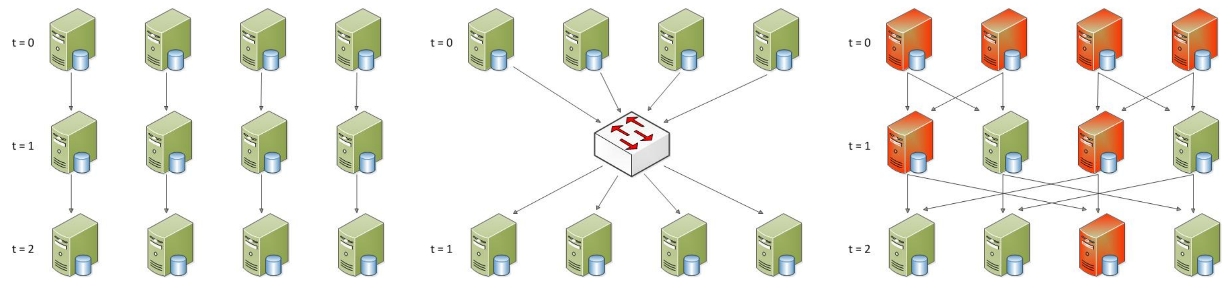
3.1 Proposed Algorithm

Our main algorithm consists of two components, i.e. butterfly reduce and asynchronous mixing of stochastic gradient step updates. Butterfly reduce guarantees local gradient estimates are propagated to the network in a balanced and efficient pattern, and asynchronous updates make sure each mixing contains most up-to-date gradient information.

Butterfly Reduce

Let there be $N = 2^k$ nodes in the cluster. A simple protocol to distribute local gradient information to the entire network is to first use a tree-based algorithm to compute the average in a single task, and then broadcast the average to each individual node, as in Vowpal Wabbit [11] implementation of AllReduce [1].

However, there also exists alternative algorithms in which execution time is faster. In butterfly reduce, the tree algorithm can be modified to avoid additional broadcast steps to save communication. The idea is to perform multiple averages concurrently, with each average producing a value in a different task [7]. The resulting butterfly communication structure is illustrated in Figure 1c. All N nodes execute the same average algorithm so that all N partial averages are in motion simultaneously. After k steps, the average is replicated on every single node. As highlighted in Figure 1c with $N = 4$ nodes, at $t = 2$ step, each node already contains gradient information from all the other nodes.



(a) Sequential update (b) AllReduce (c) Butterfly mixing

Figure 1: Different parallelization schemes for $N = 4$ nodes; t represents iteration time. (a) updates weight vector sequentially within single nodes, and averages weights from different nodes at the end of the task; this scheme is not desirable because the final estimate has high variance. (b) synchronizes weight vectors before performing any gradient descent; the synchronization process is formed with the help of a tree structure in AllReduce to reduce latency. However, it suffers from high communication cost (c) gets around the above limitations by mixing computation with communication.

Asynchronous Mixing

Butterfly mixing interleaves the above average operation with iterative stochastic gradient updates. Denote $\mathbf{w}^k(t)$ as the weight vector available on node k at time t, and $\mathbf{g}^k(t)$ the gradient evaluated at the current position. We now formally present the model of asynchronous updates of weight vector $\mathbf{w}$. Let S^{kj} be the set of times that weight vector is received by node j from node k. In our algorithm, S is determined by butterfly reduction structure. For instance, S^{kk} includes all time ticks up to the end of the algorithm for all $k \in \{1, 2, \ldots, N\}$, because each node "sends" gradient update to its own at each iterative step. As another example, according to butterfly structure, $S^{12} = \{1, 3, 5, \ldots\}$ for $N = 4$, which is an arithmetic sequence with common difference $k = 2$. The full reduce algorithm is presented in Algorithm 1.

Algorithm 1 Butterfly reduce algorithm that aggregate weight vectors in a balanced pattern

```
function ButterflyReduce($\mathbf{W}, k, t, N$)
    $i \leftarrow mod(t, \log N)$
    $j \leftarrow k + 2^{i-1}$
    if $j > 2^i \times \lceil \frac{k-0.5}{2} \rceil$ then
        $j \leftarrow j - 2^i$
    end if
    return $mean(\mathbf{w}^k, \mathbf{w}^j)$
end function
```

Butterfly mixing is initialized with zero weight at the beginning. At time t, each node updates its position according to messages $\mathbf{w}^j(t), \{j | t \in S^{ij}\}$ it receives, and incorporates new training examples coming in to compute its current gradient and new position. Specifically, $x^k(t)$ is updated according to the formula,

$$\mathbf{w}^k(t+1) = \frac{1}{2}\sum_{\{j|t\in S^{ij}\}} \mathbf{w}^j(t) + \gamma^k(t)\mathbf{g}^k(t+1), \tag{10}$$

where set $\{j|t \in S^{ij}\}$ is of cardinality 2 for all t, so that the expectation of stochastic process $\mathbf{w}^k(t)$ remains stable.

Algorithm 2 Distributed stochastic gradient descent with butterfly mixing

```
Require: Data split across $N$ cluster nodes
  $\mathbf{w} = \mathbf{0}, t = 0$, $H$ = inverse feature frequencies
  repeat
    for all nodes k parallel do
      for $j = 0 \to \lfloor \frac{n}{m} \rfloor - 1$ do
        $\mathbf{w}^k \leftarrow \textsc{ButterflyReduce}(\mathbf{W}, k, t, N)$
        $\mathbf{g}^k \leftarrow \frac{1}{m}\sum_{i=jm}^{jm+m-1} \nabla L^i(\mathbf{w^k}; \mathbf{x}^i, y^i)$ according to Equation (3) and (9)
        $\mathbf{w^k} \leftarrow \mathbf{w^k} - \gamma_t H \mathbf{g^k}$
        $t \leftarrow t + 1$
      end for
    end for
  until $p$ pass over data
```

We present detailed butterfly mixing in Algorithm 2, where $\mathbf{W}$ is an aggregation of $\mathbf{w}^k, \forall k \in \{1, 2, \ldots, N\}$. Notice that the distributed iterative update model does not guarantee the agreement on the average of weight vector $\mathbf{w}$ across nodes at any time t. However, as we will show later, the final average of $\mathbf{w}^k$ does converge in a reasonably small number of iterations. An intuitive explanation would be that butterfly reduce accelerates the convergence of $\mathbf{w}^k(t)$ to a small neighborhood of the optimal through efficient aggregation of gradient steps across the network, while timely update of asynchronous mixing provides refined gradient direction by introducing new training examples at each mixing, which further improves convergence rate.

Comparisons between different reduction and mixing schemes are illustrated in Figure 1. Sequential update provides the most resource-efficient option for parallelization of stochastic gradient steps. However, this method is not desirable because it fails to reduce the variance of gradient estimates and has no advantage over single node sequential algorithm in terms of convergence rate [14]. AllReduce synchronizes weight vectors before performing each gradient descent, and it suffers from high reduce latency and communication cost, which lengthens the overall convergence time. Our algorithm gets around the above limitation by mixing computation with communication creatively.

3.2 Theoretical Analysis

We briefly present the convergence analysis of our algorithm in this subsection. A full fledged proof and analysis could be found in Sec. 7.8 of [2] or our future technical report. The proof consists of two major components. We first find a single vector $\mathbf{z}(t)$ to keep track of all vectors $\mathbf{w}^1(t), \mathbf{w}^2(t), \ldots, \mathbf{w}^N(t)$, simultaneously and analyze its convergence; then we show that $\mathbf{w}^k(t)$ is actually converge to $\mathbf{z}(t)$ at a certain rate. The overall convergence performance is a mixture of the above two.

Weight vector $\mathbf{w}^k(t)$ is defined recursively in Equation (10). It will be useful for the analysis if we explicitly expand $\mathbf{w}^k(t)$ in terms of gradient estimates $\mathbf{g}^j(t), \forall j \in \{1, 2, \ldots, N\}, 0 < \tau < t$, that is,

$$\mathbf{w}^k(t) = \sum_{\tau=1}^{t-1} \sum_{j=1}^{N} \Phi^{kj}(t, \tau)\gamma^j(\tau)\mathbf{g}^j(\tau). \tag{11}$$

It turns out that the limit of coefficient scalar $\Phi^{kj}(t, \tau), \forall k, j$ exists as t tends to infinity. And there exist a constant A and a communication protocol dependent $\rho \in (0, 1]$, such that,

$$|\Phi^{kj}(t, \tau) - \Phi^j(\tau)| \leq A\rho^{t-\tau}, \forall t > \tau > 0. \tag{12}$$

It is not difficult to see that the more frequent the communication is, the smaller the ρ is. It is also natural to define $\mathbf{z}(t)$ that summarizes all $\mathbf{w}^k(t), \forall k \in \{1, 2, \ldots, N\}$ using the limit of $\Phi^{kj}(t, \tau)$,

$$\mathbf{z}(t) = \sum_{\tau=1}^{t-1} \sum_{j=1}^{N} \Phi^j(\tau)\gamma^j(\tau)\mathbf{g}^j(\tau), \tag{13}$$

$\mathbf{z}(t)$ can also be expressed in a recursive way to apply Lipschiz properties,

$$\mathbf{z}(t+1) = \mathbf{z}(t) + \sum_{j=1}^{N} \Phi^k(t)\gamma^j(t)\mathbf{g}^j(t), \tag{14}$$

Under Lipschitz continuity assumptions on loss function L and some bounded gradient conditions, we have

$$||\mathbf{z}(t) - \mathbf{w}^k(t)||_2 \leq A \sum_{\tau=1}^{t-1} \frac{1}{\tau} \rho^{t-\tau} b(\tau), \tag{15}$$

$$L(\mathbf{z}(t+1)) \leq L(\mathbf{z}(t)) - \frac{1}{t} G(t) + C \sum_{\tau=1}^{t} \rho^{t-\tau} \frac{b^2(\tau)}{\tau^2}, \tag{16}$$

where $b(t) = \sum_{k=1}^{N} ||\mathbf{g}^k(t)||_2$ and $G(t) = -\sum_{k=1}^{N} \Phi^k(t)||\mathbf{g}^k(t)||_2^2$, for $t \geq 1$. This concludes the convergence of the algorithm. And as we can see, both convergence depends on synchronization protocol dependent rate ρ.

3.3 Implementation Issues

We propose to implement our butterfly mixing abstraction with udp datagram protocol. Udp enjoys several key advantages such as low latency, light overhead and high flexibility. It is not a reliable communication protocol in general, but this is not a big concern in our application settings. Intra-rack communications are collision-free and packet loss can be detected by implementing some simple parity check. More attractively, it is not even necessary to recover the missing packets because occasional loss of weight vectors will not affect the overall performance of butterfly mixing. This fault-tolerance feature of our algorithm further reduces the overall communication costs.

Our system should also be compatible with Hadoop, because of its widely adoption for distributed data processing tasks. In addition, by building butterfly mixing on top of Hadoop, we can also take advantage of its speculative execution and job scheduling to further optimize our system.

4 Experiments

We evaluate the performance of our algorithm on Reuters RCV1 dataset [12]. The dataset consists of a collection of approximately 800,000 news articles, each of which is assigned to one or more categories. We are focusing on training binary classifier for CCAT (Commerce) category with logistic regression model. The input of the classifier is an article bag-of-words feature vector with tf-idf values, and the output is a binary variable indicating whether an individual article belongs to CCAT or not. Articles are randomly shuffled and split into approximately 760,000 training examples and 40,000 test examples. Training examples are further evenly portioned to N cluster nodes.

4.1 Convergence

Experiments are performed on the above RCV1 dataset with $N = 64$ and $k = 6$. According to [6], learning rates $\gamma(t)$ are chosen to be inversely proportional to $\sqrt{t}$, where t is iteration time. We find batch size m plays a conflicting role in determining overall convergence time. The smaller m is, the fewer training examples are required to reach certain loss. However, it is also desirable to use large m, so that fewer mixing/reduce steps are needed for the same amount of data usage for both butterfly mixing and AllReduce.

The impact of batch size on convergence rate is illustrated in Figure 2. Cross validation loss is plotted against total number of training examples visited across $N = 64$ nodes. Training examples are reused after one pass through the data on individual node. Interestingly, we can observe the saturation effect when batch size m reaches 1000: it takes approximately the same amount of data to achieve $loss = 4000$ for $m \leq m_s = 1000$, while the convergence performance severely deteriorates when m is beyond m_s. Therefore, considering communication overhead, the optimal batch size of gradient step should be $m = m_s$. We use $m_s = 1000$ in the following experiments in the paper.

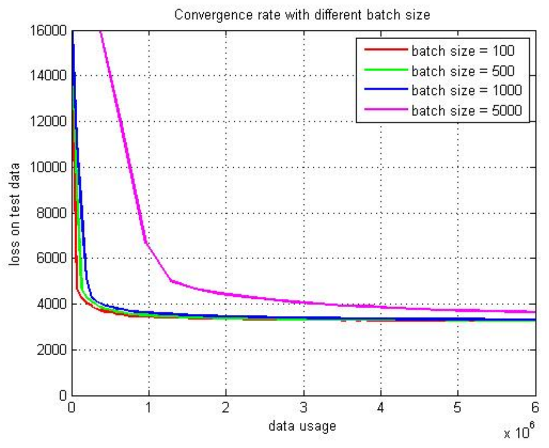
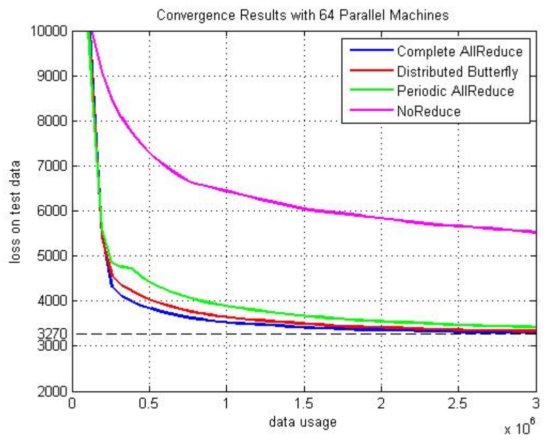
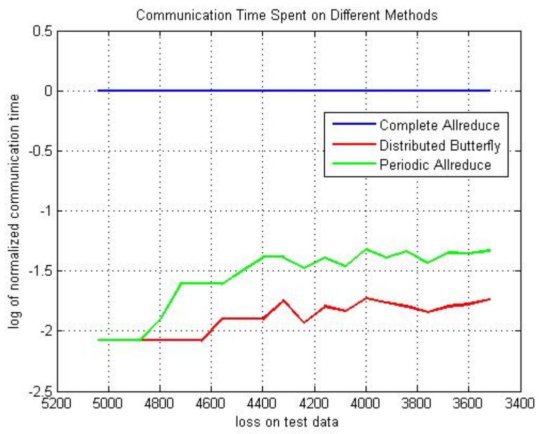


Figure 2: Impact of batch size on convergence rate.

We compare three other algorithms with our butterfly mixing in terms of convergence rate and communication cost. NoReduce is a sequential update algorithm running separately on distributed cluster nodes; it requires no communication except a final average of weight vectors at the very end of the algorithm. AllReduce synchronizes positions across all nodes before every gradient step. Pe-



(a) Convergence



(b) Comm time taken at certain loss

Figure 3: Comparisons of convergence performance and communication cost.

riodic AllReduce performs weight synchronization every k steps, so that the number communication step required is the same with butterfly mixing.

Convergence performance is presented in Figure 3a. Cross validation loss is compared across different algorithms. Step sizes are tuned individually for each algorithm, so that each algorithm performs its best and converges to the optimal. The horizontal dash line: $loss = 3270$ shows the minimized loss that all algorithms converge at. We can clearly see from Figure 3a that our butterfly mixing outperforms both NoReduce and Periodic AllReduce and the gap between AllReduce and butterfly mixing becomes insignificant after one pass through the data. In this particular RCV1 task, NoReduce achieves $loss = 3900$ after 64 times the data required for AllReduce to converge, and slowly moves to the optimal after that.

Figure 3b further illustrates the advantage of butterfly mixing. It shows the number of communication steps (y-axis) required to achieve certain loss (x-axis). AllReduce takes k communication steps for a single gradient update, while both periodic AllReduce and butterfly mixing require only one. x-axis is reversed because a smaller loss indicates better fitting of the model. Communication steps for all algorithms are normalized by that of AllReduce, and logarithmized for better visualization. As shown in Figure 3b, butterfly mixing always takes less time to achieve targeted loss, which shows its superiority over the other two algorithms.

4.2 System Performance

Execution time per iteration should be determined by three main processes, i.e. file IO, network communication and computation, as illustrated in Figure 4. The actual overall system performance also closely depends on the number of iteration steps taken to achieve convergence. We compare performance of our system with Hadoop/MapReduce [5] and Vowpal Wabbit [11], a recent cluster based implementation of AllReduce.

File IO should be the bottleneck for Hadoop/MapReduce; it is widely known that its binary SequenceFile IO is unbearably slow, and we benchmark the system to find that binary reading throughput is as low as 1mB/sec. Another inefficiency is introduced by MapReduce abstraction because of its unfriendly nature to iterative algorithms; the system has to restart after every iteration, which requires additional overhead.

Communication overhead is further compared between Vowpal Wabbit and our system. We benchmark communication performance with the unit of ms/100kB, where 100kB is approximately the size of data used for a single gradient step with optimal batch size. The cost should consist of driver overhead, switching latency and transmission delay. The last factor should be quite reliable because bandwidth over an otherwise empty link should be predictable.

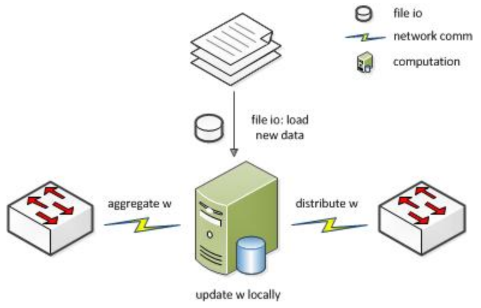


Figure 4: Performance breakdown. Execution time per update includes file IO, network communication and computation; overall execution time is multiples of number of gradient steps taken until convergence.

We test communication time per iteration of Vowpal Wabbit on a mere 4 node cluster. Execution times spent on communication are recorded for 800,000 iteration steps and are averaged to find the per iteration network IO time. Performance of our system is estimated by measuring the end-to-end response time between a pair of udp server and client; the payload is a data file with similar size of the sparse weight vector. Computation throughputs are also projected for both system.

System	File IO	Network comm.	Computation	Convergence	Bottleneck
Hadoop/MapReduce	Slow	-	-	Slow	SequenceFile IO, Hadoop overhead
VW/AllReduce	Fast	40ms/100kB	1ms/100kB	Moderate	AllReduce comm. cost and delay
Our system	Fast	2ms/100kB	1ms/100kB	Fast	-

Table 1: Comparisons of system performance

System performance and its breakdowns for all three systems mentioned above are summarized in Table 1. As we can see, time spent on communication $40ms/100kB$ is significantly higher than computation in Vowpal Wabbit. Our system strikes a better balance between communication and computation, and enjoys superior overall performance in terms of convergence.

5 Conclusion

In this paper, we present a parallelized stochastic gradient algorithm with butterfly mixing, and experiments show that its fast convergence and high resource-efficiency are desirable for large scale parallelized implementation in cluster computing settings. Butterfly mixing is also suitable for parallelization of many other iterative methods. We propose to further investigate butterfly mixing through extensive tests on a real system which will be implemented with udp communication protocol and compatible with Hadoop.

Acknowledgments

I would like to thank Prof. Abbeel for his insightful comments on this paper.

References

[1] A. Agarwal, O. Chapelle, M. Dudik, and J. Langford. A reliable effective terascale linear learning system. *Arxiv preprint arXiv:1110.4198*, 2011.

[2] D.P. Bertsekas and J.N. Tsitsiklis. Parallel and distributed computation. 1989.

[3] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. *Advances in neural information processing systems*, 20:161–168, 2008.

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

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

[6] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2010.

[7] I. Foster. *Designing and building parallel programs*, volume 95. Addison-Wesley Reading, MA, 1995.

[8] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning, second edition.* Springer Series in Statistics, 2009.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface, seconde édition.* the MIT Press, 1999.

[10] M. Kearns. Efficient noise-tolerant learning from statistical queries. *Journal of the ACM (JACM)*, 45(6):983–1006, 1998.

[11] J. Langford, L. Li, and A. Strehl. Vowpal wabbit online learning project. Technical report, Technical report, http://hunch. net, 2007.

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

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J.M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI), Catalina Island, California*, 2010.

[14] F. Niu, B. Recht, C. Ré, and S.J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in Neural Information Processing Systems*, 2011.

[15] Eric Schmidt. http://techcrunch.com/2010/08/04/schmidt-data. *TechCrunch*, 2010.

[16] J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. 1994.

[17] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P.K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14. USENIX Association, 2008.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. 2012.

[19] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. *Advances in Neural Information Processing Systems*, 23(23):1–9, 2010.