Butterfly Mixing: Accelerating Incremental-Update Algorithms on Clusters

Huasha Zhao* John Canny†

Abstract

Incremental model-update strategies are widely used in machine learning and data mining. By "incremental update" we refer to models that are updated many times using small subsets of the training data. Two well-known examples are stochastic gradient and MCMC. Both provide fast sequential performance and have generated many of the best-performing methods for particular problems (logistic regression, SVM, LDA etc.). But these methods are difficult to adapt to parallel or cluster settings because of the overhead of distributing model updates through the network. Updates can be locally batched to reduce communication overhead, but convergence typically suffers as the batch size increases. In this paper we introduce and analyze *butterfly mixing*, an approach which *interleaves* communication with computation. We evaluate butterfly mixing on stochastic gradient algorithms for logistic regression and SVM, on two datasets. Results show that butterfly mix steps are fast and failure-tolerant, and overall we achieved a 3.3x speed-up over full mix (AllReduce) on an Amazon EC2 cluster.

1 Introduction

The availability of massive data sources creates tremendous opportunities for business and the sciences. But harnessing this potential is challenging. Most machine learning methods involve some iteration to infer the best model, and many can be formulated directly as optimization problems. But traditional (batch) gradient-based optimization methods are too slow to run many passes over large datasets. Much faster convergence is typically obtained using stochastic gradient descent [3] - continuous model updates on relatively small subsets of the data. MCMC methods also follow this paradigm with updates ideally performed after each sample.

*Electrical Engineering and Computer Science Department, University of California, Berkeley. Email: hzhao@eecs.berkeley.edu

†Computer Science Division, University of California, Berkeley. Email: jfc@cs.berkeley.edu

Hence we consider "incremental update" strategies in this paper. Both stochastic gradient and MCMC methods provide good sequential performance and have generated many of the best-performing methods for particular problems (logistic regression, SVM, LDA etc.). But these methods are difficult to adapt to parallel or cluster settings because of the overhead of distributing frequent model updates through the network. Updates can be locally batched to reduce communication overhead, but convergence typically suffers as the batch size increases - see Figure 3 for an example. In this paper we introduce and analyze *butterfly mixing*, an approach which *interleaves* communication with computation.

Butterfly mixing uses a butterfly network on 2^k nodes, and executes one *constant time* communication step (a butterfly shuffle) for each computation step. Butterfly mixing fully distributes model updates after k steps, but data from smaller subsets of nodes travels with lower latency. Convergence of butterfly mixing is intermediate between full model synchronization (an AllReduce operation) on every cycle which has k times the communication cost, and full AllReduce every k compute cycles which has the same communication cost as butterfly mixing. We show through simulation and experiment that butterfly mixing comes close to offering *the best of both worlds*: i.e. low communication cost similar to periodic AllReduce but convergence which is closer to AllReduce on every cycle. Our implementation uses a hardware accelerated (through Intel MKL) matrix library written in the Scala language to achieve state-of-the-art performance on each node.

Stochastic gradient descent is a simple, widely applicable algorithm that has proved to achieve reasonably high performance on large-scale learning problems [6]. There has been a lot of research recently on stochastic gradient [9, 6] improving gradient approximations, update schedules, and some work on distributed implementations [22, 13]. However, stochastic gradient is most efficient with small batches i.e. it is a "mostly sequential" method.

When batch updates are parallelized, the batch size is multiplied by the number of parallel nodes. Local updates must then be combined by an additive or average reduction, and then redistributed to the hosts. This process is commonly known as AllReduce. AllReduce can be implemented using peer communication during gradient updates, and this has been used to improve the performance of gradient algorithms in the Map-Reduce framework[1]. However, on all but the smallest, fastest networks, AllReduce is much more expensive than a gradient update. As we will show later, AllReduce is an order of magtitude slower then an optimal-sized gradient step given both communication and computation capacity at gigabytes scale (Gbps and Gflops).

AllReduce is commonly implemented with either tree [1] or butterfly [14] topologies, as shown in Figure 1a and 1b. The tree topology uses the lowest overall bandwidth, but effectively maximizes latency since the delay is set by the slowest path in the tree. It also has no fault-tolerance. A butterfly network can be used to compute an AllReduce with half the worst-case latency. Faults in the basic butterfly network still affect the outputs, but on only a subset of the nodes. Simple recovery strategies (failover to the sibling just averaged with) can produce complete recovery since every value is computed at two nodes. Butterfly networks involve higher bandwidth, but this is not normally a problem in switched networks in individual racks. For larger clusters, a hybrid approach can use butterfly communication within each rack and then multi-node communication between racks as limited by the available bandwidth. In any case, the same strategy of interleaving communication and computation can be used.

Whatever AllReduce strategy is used, typical AllReduce communication times are significantly higher than the time to perform model updates for optimal-sized blocks of data. In the example presented in Figure 3, block sizes larger than 16000 (on 16 nodes that means 1000 samples per node) lead to significant increases in convergence time. Processing a 1k block of data takes less than 15 msec per node, but communicating model updates through AllReduce takes about 50 msecs (Figure 4a) or 200msecs (Figure 4b) - these numbers were for butterfly AllReduce, and the tree AllReduce times are much higher. As Figure 4 shows, we can achieve much lower (effectively constant) communication costs by using either butterfly reduce steps or doing a full butterfly AllReduce every k steps. The latter approach however delays global model updates by a factor of k, effectively increasing block size by k which is far from optimal. With butterfly mixing communication time is still significant, but we can use a batch size which is closer to optimal and achieve better overall running time for a given loss.

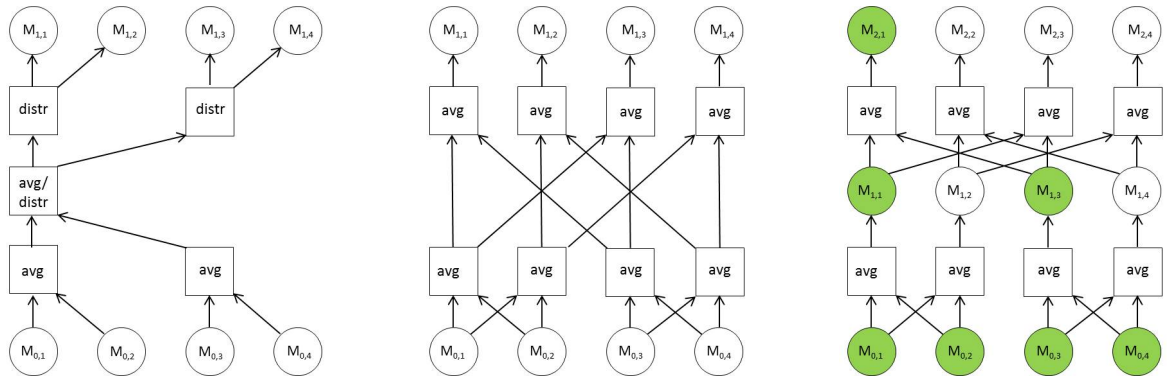
1.1 Contributions In this paper, we introduce butterfly mixing and validate its performance on two stochastic gradient algorithms on two datasets. The method should provide similar gains for MCMC methods, but that is the subject of future work. It should also be of value for other optimization methods that admit incremental optimization of a model. An MPI version of butterfly mixing was implemented with high performance libraries (Intel MKL) in the Scala language. We evaluated the system for training a topic classifier on RCV1 and a sentiment predictor on a proprietary twitter dataset featuring 170 million automatically labelled tweets. We tested the system on a 64-node Amazon EC2 cluster. Experiments show that butterfly mix steps is fast and failure-tolerant, and overall achieves a 3.3x speed-up over AllReduce.

2 Related Works

In this section, we briefly survey some of the previous works on scalable machine learning and how it informed the goals for this paper.

In recognition of the communication and computation trade-off in stochastic gradient method, Zinkevich et al. [22] propose a simple algorithm in which multiple gradient descents run in parallels and their outputs are averaged in the end. It has also been shown that in many problems there is no advantage to running this averaging method without communication [13]. Niu et al. [13] consider a lock-free approach to parallelize stochastic gradient descent, but their focus is on single multi-core processors where access/communication times are minimal. Similar to us, Agarwal et al. [2] describe and analyse a gradient-based optimization algorithm based on delayed stochastic gradient information. However, they consider an arbitrary fixed network ignoring the costs of routing messages, and leaving out the efficiency gains of routing dynamically with a butterfly.

Hadoop MapReduce [8] is the most popular platform for distributed data processing, and Chu et al. [7] describes a general framework to run machine learning algorithms on top of it. However, Hadoop has often been criticised for using "disk for communication", and practical running times for machine learning algorithms are typically much higher than



(a) Tree AllReduce (b) Butterfly AllReduce (c) Butterfly mixing

Figure 1: Different parallelization schemes for $N = 4$ nodes. Each node (circle) M_{ij} performs model update on new data at time i for node j. (a) and (b) synchronize model before every gradient step, with tree-based and butterfly AllReduce respectively. They suffer from overwhelmingly high communication overhead. (c) reduces synchronization overhead without losing convergence performance by mixing model update with communication.

algorithms using the network for communication. In recognition of this, Spark [19, 21] was developed to allow in-memory models as RDDs. However, Spark was optimized for relatively infrequent "batch" operations on these datasets. RDDs are immutable and each iteration involves creation of a new RDD with a distributed dataflow scheduled by the Mesos system. We believe this is a much longer operation than an AllReduce step.

While one can use Hadoop and Spark for basic data management, efficient Stochastic gradient implementation requires custom communication code, which can still be done on MapReduce jobs. That was the approach used for Hadoop-compatible AllReduce [1]. These authors added a communication primitive to the map phase that allows map tasks to perform multiple AllReduce steps throughout the map phase. This approach has the same problems with AllReduce addressed by our work, and we believe would benefit from replacing the AllReduce step with a butterfly mix step. While we have used MPI for butterfly mixing, similar gains should be possible in systems that support more flexible message routing such as Hyracks [5] and DraydLINQ [20], since these systems can implement batterfly mixing directly.

3 Training Models with Stochastic Gradient Descent

Stochastic gradient is typically used to minimize a loss function L written as a sum of differentiable functions over data instances $L : \mathbb{R}^d \mapsto \mathbb{R}$,

$$L(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{w}; \mathbf{x}_i, y_i), \tag{3.1}$$

where $\mathbf{w}$ is a $d-$dimensional weight vector to be estimated, and $\mathbf{x}_i \in \mathbb{R}^d$, $y_i \in \{+1, -1\}$ are the feature vector and the label of the i^{th} example respectively.

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

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \gamma(t)H(t)\hat{\nabla}L(\mathbf{w}(t)). \tag{3.2}$$

In the formula above, $\gamma(t)$ is a time varying step size, and $H(t)$ is called preconditioner, which attempts to reduce the condition number of the system and, as a result, to ensure the fast convergence of the gradient method [18]. In this paper, we use $\gamma(t) = \frac{\gamma_0}{\sqrt{t}}$ and Jacob preconditioner to be the inverse feature frequencies [9]. This simple weighting scheme shows very good convergence on most problems.

In stochastic gradient method, $\nabla L(\mathbf{w}(t))$ is estimated by partial average of the empirical loss function,

$$\hat{\nabla}L(\mathbf{w}(t)) = \frac{1}{|\mathcal{T}_j|}\sum_{i\in\mathcal{T}_j} \nabla L(\mathbf{w}; \mathbf{x}_i, y_i) \tag{3.3}$$

where $\mathcal{T}_j$ is the j^{th} "batch", and $|\mathcal{T}_j|$ is called batch size. Stochastic gradient converges fastest with batch size 1, but in practice batch size can be increased until there is significant interaction between sample updates. This will occur when there is a lot of overlap between the (sparse) model coefficients being changed by different sample updates.

3.1 Logistic Regression Logistic regression model [10] is among the most widely used supervised learning models.

The loss function is defined as the negative log-

likelihood of the model, which can be written as,

(3.4)
$$L(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\left\{-y_i\mathbf{w}^T\mathbf{x}_i + \log(1+e^{\mathbf{w}^T\mathbf{x}_i})\right\}$$

The gradient step to find optimal weight $\mathbf{w}$ in logistic regression model can be computed according to Equation 3.3 and

$$\nabla L(\mathbf{w};\mathbf{x}_i,y_i) = \mathbf{x}_i(y_i - \frac{e^{\mathbf{w}^T\mathbf{x}_i}}{1+e^{\mathbf{w}^T\mathbf{x}_i}}). \tag{3.5}$$

3.2 Support Vector Machine (SVM) SVM is perhaps the most popular algorithm for training classification models. The goal of linear SVM is to find the minimal value of the following optimization problem,

$$\min_{\mathbf{w},b}\frac{\lambda}{2}||\mathbf{w}||^2 + \sum_{i=1}^{n}\max\{0, 1-y_i\mathbf{w}^T\mathbf{x}_i\} \tag{3.6}$$

Shalev-Shwartz et al. [17] have shown that the primal problem with hinge loss can be solved by first updating $\mathbf{w}(t)$ using the following sub-gradient,

$$\nabla L(\mathbf{w};\mathbf{x}_i,y_i) = \lambda\mathbf{w} - y_i\mathbf{x}_i, \tag{3.7}$$

and then scaling $\mathbf{w}(t)$ by a factor of $\min\{1, \frac{1}{\sqrt{\lambda}||\mathbf{w}(t)||}\}$. While many algorithms for SVM have been developed, the fastest in recent years have used stochastic gradient methods [6].

4 Butterfly Mixing Algorithm

In this section, we present our butterfly mixing algorithm, and briefly study its convergence behaviour.

4.1 Butterfly Network In a butterfly network, every node computes some function of its own value and one other node and outputs to itself and one other node. The neighbors at one level of the network are all neighbors in a particular dimension of a hypercube over the nodes. For our purposes, the node operation will always be an average. At the end of the process every output node holds the average of the inputs. The latency is k for 2^k nodes which is best possible for point-to-point communication. Butterfly network is failure tolerant: A fault or delay at one step only affects the subtree above it.

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.

4.2 Butterfly Mixing Butterfly mixing interleaves the above average operation in butterfly network 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 present in detail the model of butterfly 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(W, k, t, N)
    i ← mod(t, log N)
    j ← k + 2^{i-1}
    if j > 2^i × ⌈(k−0.5)/2⌉ then
        j ← j − 2^i
    end if
    return mean(w^k, w^j)
end function
```

Butterfly mixing is initialized with zero at the beginning. At time t, each node updates its weight vector 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,

(4.8)
$$\mathbf{w}^k(t+1) = \frac{1}{2}\sum_{\{j|t\in S^{ij}\}}\mathbf{w}^j(t) + \gamma^k(t)H(t)\mathbf{g}^k(t+1),$$

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.

The detailed butterfly mixing is presented in Algorithm 2, where $\mathbf{W}$ is an aggregation of $\mathbf{w}^k, \forall k \in$

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**
        $\mathbf{w}^k \leftarrow \text{ButterflyReduce}(\mathbf{W}, k, t, N)$
        $\mathbf{g}^k \leftarrow \frac{1}{m} \sum_{i=jm}^{jm+m-1} \nabla L^i(\mathbf{w^k}; \mathbf{x}^i, y^i)$
        $\mathbf{w^k} \leftarrow \mathbf{w^k} - \gamma_t H \mathbf{g^k}$
        $t \leftarrow t + 1$
      end for
    end for
  until p pass over data

$\{1, 2, \ldots, N\}$. 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 reduction accelerates the convergence of $\mathbf{w}^k(t)$ to a small neighbourhood 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.

Comparisons between different reduction and mixing schemes are illustrated in Figure 1. The communication latency for the butterfly mixing network is the same as performing a tree AllReduce every $2k$ steps, and a butterfly AllReduce every k updates. Butterfly mixing guarantees that data are fully mixed after k steps, but because the mixing is continuous the average over smaller sub-cubes of data is available at lower latencies. We would therefore expect the butterfly mix network convergence rate to be somewhere between a network with AllReduce on every step, and periodic AllReduce every k steps. As we will see, performance is in fact closer to Allreduce on every step in terms of convergence, while the communication cost is the same as AllReduce every k steps.

4.3 Convergence Results We briefly present the convergence analysis of our algorithm in this subsection. A similar proof and analysis could be found in Sec. 7 of [4]. 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.

In Section 4.2, vector $\mathbf{w}^k(t)$ is defined recursively in Equation 4.8. 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,

$$\mathbf{w}^k(t) = \sum_{\tau=1}^{t-1} \sum_{j=1}^{N} \Phi^{kj}(t, \tau) \gamma^j(\tau) \mathbf{g}^j(\tau). \tag{4.9}$$

It can be shown that the limit of coefficient scalar $\Phi^{kj}(t, \tau), \forall k$ converges to $\Phi^j(\tau)$ linearly with rate ρ, as follows,

$$|\Phi^{kj}(t, \tau) - \Phi^j(\tau)| \leq A \rho^{t-\tau}, \forall t > \tau > 0. \tag{4.10}$$

On the other hand, it is natural to define $\mathbf{z}(t)$ that summarizes all $\mathbf{w}^k(t), \forall k \in \{1, 2, \ldots, N\}$ using the limit of $\Phi^{kj}(t, \tau)$,

$$\mathbf{z}(t) = \sum_{\tau=1}^{t-1} \sum_{j=1}^{N} \Phi^j(\tau) \gamma^j(\tau) \mathbf{g}^j(\tau), \tag{4.11}$$

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

$$\mathbf{z}(t+1) = \mathbf{z}(t) + \sum_{j=1}^{N} \Phi^k(t) \gamma^j(t) \mathbf{g}^j(t), \tag{4.12}$$

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

$$||\mathbf{z}(t) - \mathbf{w}^k(t)||_2 \leq A \sum_{\tau=1}^{t-1} \frac{1}{\tau} \rho^{t-\tau} b(\tau), \tag{4.13}$$

(4.14)

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

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.

5 System Implementation

Butterfly mixing is a rather general design pattern that can be implemented on top of many systems. We start with a simple MPI version written on top of the BIDMat matrix toolkit in the Scala language: `http://bid.berkeley.edu/BIDMat/index.php/`. BIDMat inherits the REPL (command interpreter) from Scala but also runs on a JVM and so can

be used in cluster toolkits like Hadoop and Spark. Its uses native code linkage to high-performance libraries including Intel MKL and HDF5 for file IO. Our system can be configured for training widely used logistic regression model and SVM. And, it can be easily extendible and suitable to any cumulative update method, and particular any gradient-based optimization algorithms.

5.1 Communication Module We build our MPI communication framework on top of MPJ Express [16], which is an open-source implementation of Java bindings for the MPI standard. The performance of MPJ Express has been completely studied in [15] and shown to be close to a more widely used C/C++ MPI interface, Open MPI [11]. Furthermore, MPJ Express provides seamless integration with to butterfly mixing developing environment.

Four communication patterns, including butterfly mixing, have been implemented and detailed below. We also enforce synchronization using mpi barrier at the end of each communication step.

- **NoReduce:** there was no communication between nodes in the cluster.
- **Complete AllReduce:** an MPI AllReduce was performed on every step, which is equivalent to a sequential batch SGD where batch size was single-node batch size times number of nodes. AllReduce is implemented with the MPI standard API, `public void Allreduce(...)`. We use the butterfly implementation of AllReduce in MPJ Express package with `EXOTIC_ALLREDUCE` tag turned on.
- **Butterfly mixing:** a Butterfly mix step occurred on each round, where each node send and receive updated weight $\mathbf{w}$ to/from its pair node. This procedure is implemented using the API `public Status Sendrecv(...)`.
- **Periodic AllReduce:** A complete butterfly AllReduce was performed every $log_2 N$ steps. Periodic AllReduce has the same communication cost as butterfly mixing.

6 Experiments

We chose to train a standard logistic regression model on a widely used sparse training set, Reuters RCV1 [12] and a linear SVM on a filtered subset of about 6 months of data from Twitter.

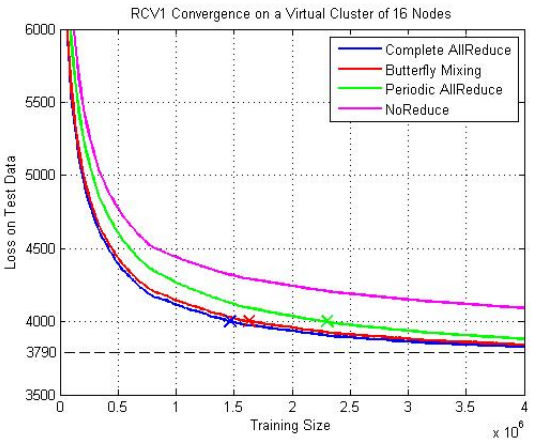


Figure 2: Convergence Performance with Different Communication Schemes.

6.1 Description of Datasets The RCV1 news is standard data set for algorithm benchmarking. It 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 of dimension 50k. 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.

To further evaluate the performance of butterfly mixing, we build a tweets sentiment classifier using SVM with 250k uni-gram features. We collected a filtered stream of about 6 months of tweets which contain emoticons such as:

- Positive sentiment: ":-)", ":D", ";)" etc.
- Negative sentiment: ":-(", "=(", ";(" etc.

There are about 170 million unique tweets in this dataset. Since emoticons have known positive or negative sentiment, they are used as training labels.

6.2 Simulation Study We first study the convergence rates between different parallization schemes given training sizes. Experiments are performed on RCV1 dataset on a virtual cluster with $N = 16$ nodes. We measured the logistic loss as a function of the number of examples processed. As is shown in Figure 2, in terms of loss at a fixed amount of data, the butterfly mix loss is indeed closer to complete AllReduce than periodic AllReduce. More importantly, butter-

fly mixing is even closer to complete AllReduce when the graph is sliced at a fixed loss value. As marked in Figure 2, at $loss = 4000$, butterfly mixing consumes almost as much data as complete AllReduce, while periodic AllReduce needs 60% more. Similar ratios occur at other loss values.

This result is very encouraging. On this relatively small cluster, we have been able to approach the convergence rate of complete AllReduce using only $1/log_2 N$ as much communications. We also reduced the time to reach a given loss value by roughly 30% relative to periodic AllReduce by simply reordering the gradient and mixing step.

6.3 System Performance on Real Cluster We tested our system on both Berkeley CITRIS cluster for high-performance computing and Amazon EC2 cluster. The CITRIS cluster is made up of 33 IBM Dataplex server nodes. Each node has two Intel Xeon quad core processors (Nehelam's) at 2.7 Ghz with about 24GB of RAM. All the nodes are connected with QDR Infiniband interconnect. For the EC2 cluster, we use M3 Extra Large instances with 15G memory and 13 EC2 compute units.

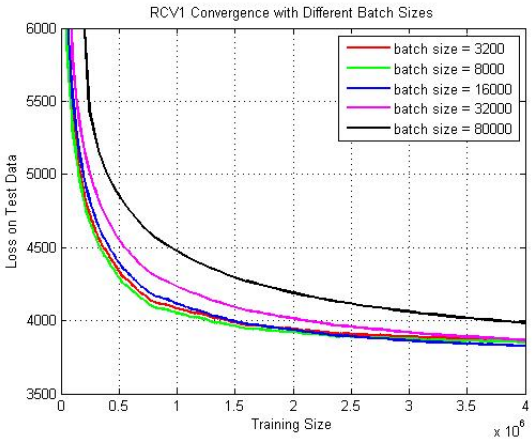


Figure 3: Impact of Aggregate Batch Sizes on Convergence Performance.

We evaluated the performance of the system in terms of "time to reach targeted loss" T_{loss}. We can break down T_{loss} into the following two parts,

$$T_{loss} = \alpha^m \frac{S_{loss}}{N} + \beta^m(N)\frac{S_{loss}(S_{batch})}{S_{batch}} \tag{6.15}$$

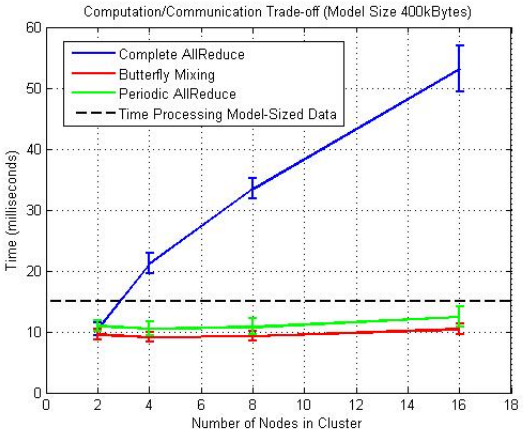
where S_{batch} is the aggregate batch size across N nodes, and S_{loss} is the training size required to reach the target loss value. The first part of the formula measures computation time, where α^m is a model specific parameter that quantifies the time of processing unit size of training data for model m; and the second part is communication time, in which β^m is also model relevant and a function of cluster size N for Complete AllReduce. Note above that parallelization will provide benefit only when α^m is comparable to, if not much bigger than β^m.

It is important to notice that the training size required for convergence S_{loss} is a function of S_{batch}. As illustrated in Figure 3, the smaller S_{batch} is, the fewer training examples are required to reach certain loss. Interestingly, we can observe the saturation effect when S_{batch} reaches 16000: further decrease on S_{batch} does not improve the convergence performance. Actually, S_{batch} is a key parameter that determines the overall system performance. Larger batch size will indicate less communication time, however, it will also worsen the convergence performance in terms of S_{loss} which will eventually lift up T_{loss}.

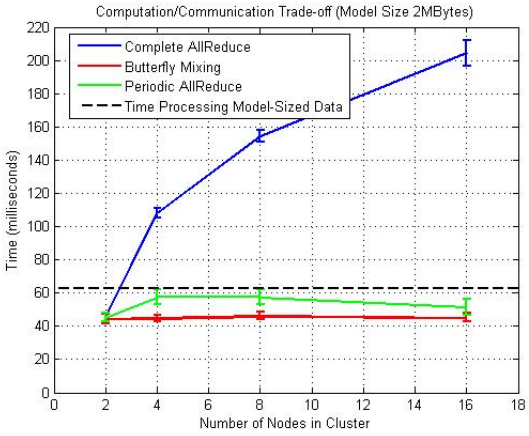
To further understand the trade-off between communication and computation, we present in Figure 4 the benchmark of wall clock times taken for computing/communication per gradient step. Computation time should break down to CPU time for gradient step and disk time, while communication time should be a function of model sizes (model dimension $\times$ size of Double) as well as the size of cluster. We report the computation time for processing model-sized data, this number can provide some insight on CPU/network trade-off, because optimal batch size should be multiples of model size, and "mini-batch" size processed by individual node in one round should be comparable to model size after paralellization. Results are shown in Figure 4, as we can see, communication time per butterfly mixing is stable across different sizes of cluster N and comparable to that of computation for both models. At the same time, communication time is much worse for AllReduce, where it increases logarithmically with N.

We test the algorithm performance with different communication schemes on the CITRIS cluster. S_{batch} are tuned to minimize T_{loss} for different schemes according to 6.15. Loss on test data is plotted against wall clock time for both RCV1 and Twitter datasets in Figure 5. This results are very promising. Butterfly mixing have been able to achieve 2.5x and 2x speed-up in comparison with complete AllReduce (at $loss = 30000$ and 4000) on Twitter and RCV1 datasets respectively. We have also saved 30% time to reach a given loss value relative to periodic AllReduce.

Finally, we explore the running time as a function of the number of nodes. We changed the number of



(a) Model Size: 400kBytes



(b) Model Size: 2Mbyes

Figure 4: Communication Overhead per Gradient Step. Dash lines show the times it takes to process model-sized training data on a single node.

nodes from 2 to 64 and computed the speedup ratio relative to the run with single node [1]. Speedup ratio is defined as the ratio between T_{loss}'s of different cluster setups. S_{batch} for each cluster is optimized individually to balance the communication trade-off as in 6.15. Results on Twitter dataset are reported in Figure 6. Butterfly mixing scales well on both clusters, providing a 5x speedup on the 16-node CITRIS cluster and 11.5x speedup on the 64-node EC2 cluster. In contrast, complete AllReduce performs badly with merely 3.5x gain on the 64-node cluster, and this verifies the argument earlier that iterative algorithms are in general difficult to adapt to parallel. In fact, AllReduce spend most time to communicate model updates across nodes, while butterfly mixing successfully mitigates this communication dilemma and achieves a 3.3x (out of 6x theoretically) performance gain over AllReduce on the 64-node cluster.

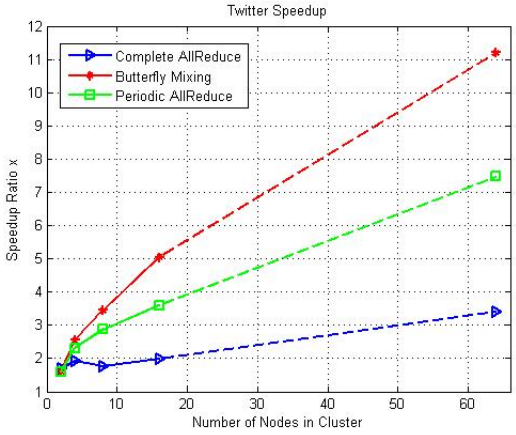


Figure 6: Speedup Ratio Relative to Single Processor.

7 Conclusions and Future Work

In this paper, we described butterfly mixing as an approach to improve the cluster performance with incremental update algorithms such as stochastic gradient and MCMC. Experiments were done with stochastic gradient implementations of logistic regression and SVM. Work remains to quantify the performance of butterfly mixing on different algorithms especially MCMC and on clusters with dynamic node scheduling. We will also explore the uses of butterfly mixing in conjunction with more advanced gradient estimation methods (e.g. LBFGS) which may admit larger optimal block sizes and reduce the communication "pressure".

[1] We run the experiments for cluster size 2-16 on the CITRIS cluster, and cluster size 64 on Amazon EC2.

References

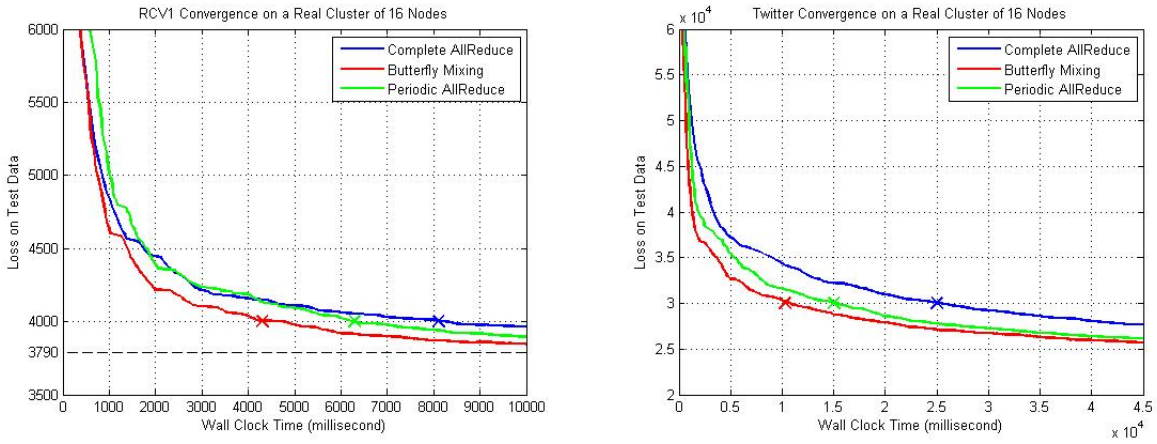
[1] A. Agarwal, O. Chapelle, M. Dudik, and J. Langford. A reliable effective terascale linear learning system. *Arxiv preprint arXiv:1110.4198*, 2011.

[2] A. Agarwal and J.C. Duchi. Distributed delayed stochastic optimization. *arXiv preprint arXiv:1104.5525*, 2011.

[3] D.P. Bertsekas. Nonlinear programming. 1999.

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

[5] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Data*



(a) RCV1

(b) Twitter

Figure 5: Convergence Performance including Communication Overhead on a 16-node Cluster.

Engineering (ICDE), 2011 IEEE 27th International Conference on, pages 1151–1162. IEEE, 2011.

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

[7] 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.

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

[9] 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.

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

[11] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[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] 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.

[14] P. Patarasuk and X. Yuan. Bandwidth efficient all-reduce operation on tree topologies. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[15] A. Shafi, B. Carpenter, M. Baker, and A. Hussain. A comparative study of java and c performance in two large-scale parallel applications. *Concurrency and Computation: Practice and Experience*, 21(15):1882–1906, 2009.

[16] Aamir Shafi, Bryan Carpenter, and Mark Baker. Nested parallelism for multi-core hpc systems using java. *J. Parallel Distrib. Comput.*, 69(6):532–545, 2009.

[17] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal estimated sub-gradient solver for svm. In *Proceedings of the 24th international conference on Machine learning*, pages 807–814. ACM, 2007.

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

[19] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[20] 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.

[21] 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.

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