# Large-Scale Distributed Non-negative Sparse Coding and Sparse Dictionary Learning

Vikas Sindhwani and Amol Ghoting
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
{vsindhw,aghoting}@us.ibm.com

## ABSTRACT

We consider the problem of building compact, unsupervised representations of large, high-dimensional, non-negative data using sparse coding and dictionary learning schemes, with an emphasis on executing the algorithm in a Map-Reduce environment. The proposed algorithms may be seen as parallel optimization procedures for constructing sparse non-negative factorizations of large, sparse matrices. Our approach alternates between a parallel sparse coding phase implemented using greedy or convex ($l_1$) regularized risk minimization procedures, and a sequential dictionary learning phase where we solve a set of $l_0$ optimization problems exactly. These two-fold sparsity constraints lead to better statistical performance on text analysis tasks and at the same time make it possible to implement each iteration in a single Map-Reduce job. We detail our implementations and optimizations that lead to the ability to factor matrices with more than 100 million rows and billions of non-zero entries in just a few hours on a small commodity cluster.

## Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

## General Terms

Algorithms, Performance, Experimentation

## 1. INTRODUCTION

Let us assume that we are given a collection of $N$ data points or signals in a high-dimensional space $\mathbb{R}^D$: $\mathbf{x}_i \in \mathbb{R}^D, 1 \leq i \leq N$. Let $\mathbf{h}_j \in \mathbb{R}^D, 1 \leq j \leq K$ denote a dictionary of $K$ "atoms" which we collect as rows of the matrix $\mathbf{H} = (\mathbf{h}_1 \ldots \mathbf{h}_K)^T \in \mathbb{R}^{K \times D}$. Given a suitable dictionary $\mathbf{H}$, the goal of *sparse coding* is to represent datapoints approximately as *sparse* linear combinations of atoms in the dictionary, i.e., $\mathbf{x}_i \approx \sum_{j=1}^{K} w_{ij}\mathbf{h}_j = \mathbf{H}^T\mathbf{w}_i$, where the coefficient vector $\mathbf{w}_i \in \mathbb{R}^K$ typically only has a few non-zero elements. Of course, the effectiveness of sparse coding in lending a high-quality compact representation to the entire dataset depends on whether the data distribution concentrates around lower-dimensional subspaces, and crucially, on whether the choice of the dictionary appropriately captures this structure.

Spread over several decades, the quest for an appropriate dictionary that is well-suited to a class of signals (e.g. natural images) has generated a vast body of work in Signal Processing and Computational Harmonic Analysis, encapsulating the development of Fourier, DCT, Wavelets and other bases, and matured into the establishment of impactful coding standards. Broadly speaking, in these efforts, dictionary design centers around pre-specified mathematical functions that are justified by optimality proofs and error bounds for the class of signals they attempt to model.

An alternative to the aforementioned approach to modeling is *learning a dictionary* [9] from the data itself. Led by the emergence of the web, we are increasingly encountering new sources of data that more often than not are completely unstructured, very high-dimensional, and expectedly voluminous. Dictionary learning now becomes appealing, as large web-based repositories essentially provide unrestricted samples of natural signals, thereby allowing one to bypass the limitations of analytical formulations that are applicable to restricted signal classes. However, to make dictionary learning feasible, we are confronted with the formidable challenge of scaling associated algorithms to web-scale datasets.

Fortunately, the growing ubiquity of multi-core and cluster systems potentially offers a solution. To make parallel systems more accessible to machine learning researchers and practitioners, recent years have seen the development and adoption of the Map-Reduce (MR) programming model [7]. Map-Reduce programs, when deployed using engines like Hadoop, can be executed on large clusters in a fault tolerant fashion. Higher-level declarative [11] and imperative [10] programming frameworks to ease the implementation of learning algorithms on top of Map-Reduce have also been developed. This paper delves into the problem of carefully designing and implementing large-scale sparse coding and dictionary learning methods that afford efficient parallelizations on distributed computing environments like Hadoop.

We focus on text-like domains that generate very high-dimensional, non-negative, and sparse data matrices. Our framework may also be viewed as providing sparse Non-negative Matrix Factorizations (NMF) [14, 12, 16]. Sparse NMFs and dictionary learning on document collections has recently emerged [13] as an appealing alternative to popular Bayesian topic models such as the Latent Dirichlet Alloca-

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 and/or a fee.
*KDD'12,* August 12–16, 2012, Beijing, China.
Copyright 2012 ACM 978-1-4503-1462-6 /12/08 ...$15.00.

tion [4]. In very recent work, [1] present the first theoretical results, under certain assumptions, on provably correct NMF algorithms. For various extensions and applications of NMFs, see [6].

Our contributions may be highlighted as follows: (1) We describe highly efficient parallel algorithms for sparse coding and dictionary learning. These algorithms are implemented using multithreading on a multi-core system and the Map-Reduce programming model on a cluster-system. A hallmark of the method is that it is possible to implement each iteration of the optimization algorithm using a single Map-Reduce job. We note several recent efforts [18, 15, 17] on scaling up related algorithms. (2) Our algorithms alternate between a distributed sparse coding phase and a sequential in-memory dictionary learning phase. For sparse coding, we organize non-negative variants of both Orthogonal Matching Pursuit (OMP) and Lasso [9, 5] around very efficient and light-weight updates. Our work lends a better understanding of the strengths and weaknesses of greedy versus convex methods in the context of parallel sparse non-negative matrix factorizations, a natural comparison that we have not seen reported elsewhere in the literature. (3) In contrast to existing Sparse NMF methods, we also impose hard sparsity constraints on the dictionary atoms. We observe that the related subproblem in coordinate descent can be solved exactly. For textual problems, dictionary sparsity imposes the natural *statistical* prior that meaningful topics have support on only a small, distinctive set of words. As a complimentary *computational* benefit, these dictionary sparsity constraints also enable fast in-memory updates of dictionary atoms. (4) We present empirical results on both small and large problems on multi-core and cluster systems. We show that datasets with more than 6 billion non-zero entries - amongst the largest demonstrations of such techniques - can be factored in a matter of hours on a moderately sized cluster, exhibiting near linear scaling with respect to data size, number of cores, and dictionary size.

## 2. FORMULATION

We are given $N$ datapoints, $\{\mathbf{x}_i\}_{i=1}^N, \mathbf{x}_i \in \mathbb{R}^D$, and $\mathbf{H} = (\mathbf{h}_1 \dots \mathbf{h}_K)^T \in \mathbb{R}^{K\times D}$ denotes the dictionary matrix comprising of $K$ atoms which are optimization variables in the dictionary learning problem. We denote the associated sparse coding variables by $\{\mathbf{w}_i\}_{i=1}^N, \mathbf{w}_i \in \mathbb{R}^K$ and collect them as rows of the matrix $\mathbf{W} = (\mathbf{w}_1 \dots \mathbf{w}_N)^T$. We will use $w_{ij}$ to denote the $j^{th}$ element of $\mathbf{w}_i$. We pose the following class of joint optimization problems over $\mathbf{W}$ and $\mathbf{H}$,

$$\arg\min_{\mathbf{W},\mathbf{H}} \frac{1}{N}\sum_{i=1}^{N} \|\mathbf{x}_i - \mathbf{H}^T\mathbf{w}_i\|_2^2 + \lambda\|\mathbf{H} - \mathbf{H}^0\|_{fro}^2 \qquad (1)$$

subject to the following constraints:

$$\begin{aligned} \mathbf{w}_i &\geq 0 && \text{non-negative coding} && (2)\\ \|\mathbf{w}_i\|_p &\leq \gamma && \text{sparse coding} && (3)\\ \mathbf{h}_j &\geq 0 && \text{non-negative dictionary} && (4)\\ \|\mathbf{h}_j\|_q &\leq \nu && \text{sparse dictionary} && (5)\\ \|\mathbf{h}_j\|_2^2 &= 1 && \text{uniqueness} && (6) \end{aligned}$$

where $i$ runs from 1 to $N$, $j$ runs from 1 to $K$, $p = \{0, 1\}$, $q = 0$, and $\lambda, \gamma, \nu$ are real-valued parameters.

The first term in the objective function above, Eqn 1, measures the squared reconstruction error. It is known to be non-convex in $\mathbf{W}$ and $\mathbf{H}$. The second term is a regularizer that enforces the learnt dictionary to be close, in Frobenius norm, to a prior $\mathbf{H}_0$, which may be available e.g., if models are being learnt periodically over time (if $\mathbf{H}^0$ is not available, $\lambda$ may be set to 0). Equations 2,4 impose non-negativity constraints. The constraint on $\|\mathbf{w}\|_p$ in Eqn 3 implements sparse coding by imposing a bound $\gamma$ on a sparsity inducing norm ball: $p = 0$ casts this constraint in terms of the $l_0$ pseudo-norm, i.e., the number of non-zeros in $\mathbf{w}_i$ should not exceed $\gamma$, while $p = 1$ uses the $l_1$ ball of radius $\gamma$ which is a convex set. Equation 5 imposes similar sparsity constraints on the dictionary elements. We mainly restrict our implementation to hard sparsity constraints, i.e., $q = 0$, and bound the number of non-zeros per atom exactly by $\nu$. As we outline in section 4, the associated rank-one subproblem can actually be solved exactly despite the intractability of working with $l_0$ norm in general. This allows us to apriori control the memory requirements for maintaining $\mathbf{H}$ as a sparse matrix and turn its updates across iterations into very cheap sequential operations. Our approach therefore enforces *double sparsity* in both the factors. Finally, for vanishing $\lambda$ and sufficiently large values of $\gamma, \nu$, it is easy to see that the reconstruction error is invariant under the transformation $\mathbf{H}^T\mathbf{w}_i = \mathbf{H}^T\mathbf{Q}\mathbf{Q}^{-1}\mathbf{w}_i$ for any invertible $K \times K$ scaling matrix $\mathbf{Q}$, making the minimization ill-posed. In particular, for clustering applications, it is common to assign $\mathbf{x}_i$ to the cluster $\arg\max_j w_{ij}$, which can be arbitrarily changed by rescaling. We therefore require dictionary elements to have unit $l_2$ norm; other constraint sets which are convex can also be instead used [16].

Our optimization strategy is cyclic Block Coordinate Descent (BCD) [3]. Each iteration of the algorithm cycles over the variables $\mathbf{W}$ and $\mathbf{h}_1 \dots \mathbf{h}_K$, optimizing a single variable at a time while holding others fixed. It is easy to see that since the objective function in Eqn. 1 is separable in $\{\mathbf{w}_i\}_{i=1}^N$, i.e. rows of $\mathbf{W}$, they can be optimized in parallel. We call this the Parallel Sparse Coding phase of the algorithm:

```
PARALLEL FOR  $i = 1 \dots N$ :
    $\mathbf{w}_i = \arg\min_{\mathbf{w}\geq 0, \|\mathbf{w}\|_p \leq \gamma} \|\mathbf{x}_i - \mathbf{H}^T\mathbf{w}\|_2^2$    (7)
```

In the next section, we outline non-negative variations of Orthogonal Matching Pursuit and Lasso to solve these subproblems for $p = 0$ and $p = 1$ respectively.

In the Dictionary Learning phase of BCD, we cycle sequentially over $\mathbf{h}_1 \dots \mathbf{h}_K$, i.e., rows of $\mathbf{H}$, keeping $\mathbf{W}$ fixed and solve the resulting subproblems. Due to hard sparsity constraints, as well as appropriate aggregations from the sparse coding phase, we can guarantee that dictionary variables can be conveniently held in memory as a sparse matrix $\mathbf{H}$, and manipulated very fast with efficient and exact sequential updates as outlined in section 4. We declare convergence if the overall mean reconstruction error fails to improve significantly relative to its previous value.

In the most general non-convex optimization setting, limit points of a BCD process may not be stationary points of the objective function. Even for a stationary point, apriori theoretical guarantees concerning model quality typically cannot be established. In practice though, such models nonetheless often yield good solutions that turn out to provide valuable empirical insight. We next describe the details of our approach.

## 3. NON-NEGATIVE SPARSE CODING

Without non-negativity constraints, for a fixed $\mathbf{H}$, Eqn 7 is the classic least-squares sparse regression model, for which greedy orthogonal matching pursuit (OMP) and Lasso are two very well-studied and widely successful algorithmic frameworks [9, 5], with intriguingly similar theoretical guarantees concerning correctness.

Since the dictionary $\mathbf{H}$ is also varying in our setting, the contrast between greedy and convex methods becomes particularly interesting for the following reasons. In each iteration, OMP greedily selects the atom that best helps in reducing the current reconstruction residual. All the atoms are then reoptimized. Lasso, on the other hand, requires numerical optimization procedures, e.g. FISTA [2], that can handle composite objectives with a smooth and a non-differentiable component. Due to its simplicity, OMP tends to be faster than Lasso. The per iteration complexity of OMP involves greedy selection of an atom from upto $K$ candidates, and solving a least squares system for refitting. For Lasso, it involves taking the gradient of the objective and adapting the line search stepsize. OMP, by design, converges in no more than $\gamma$ iterations while Lasso (using FISTA) converges to an $\epsilon$-optimal solution in $O(\frac{1}{\sqrt{\epsilon}})$ iterations. One advantage of OMP is that it imposes a predictable sparsity structure on $\mathbf{W}$ and hence we know exact memory requirements upfront. However, in the context of the full BCD algorithm, OMP-based sparse coding is not guaranteed to converge (using currently available theoretical results) without making assumptions on the dictionary at every iteration. For sparse coding with Lasso (assuming $q = 1$ or $\nu = D$, convex uniqueness constraints and exact minimization), well-known convergence results [3] imply that the full BCD algorithm will show monotonic decrease and converge to a stationary point. In practice though, OMP may still offer consistent descent to a good solution. We also point out that across BCD iterations, Lasso can utilize warm starts from previously found solutions, while OMP essentially needs to start from scratch. In a distributed setting, it is furthermore worthwhile to investigate if OMP and Lasso offer different load balancing characteristics.

These differences motivate us to study both OMP and Lasso for sparse coding and dictionary learning, and benchmark them in a parallel environment. We next derive very efficient implementations of non-negative versions of these algorithms. For simplicity in the exposition below, we assume that the data points are normalized to unit $l_2$ norm, i.e., $\|\mathbf{x}_i\|_2 = 1$.

### 3.1 Non-negative OMP (NOMP)

We are interested in solving $\arg\min_{\mathbf{w}\geq 0, \|\mathbf{w}\|_0 \leq \gamma} R(\mathbf{w})$ where $R(\mathbf{w}) = \|\mathbf{x} - \mathbf{H}^T\mathbf{w}\|_2^2$ denotes the reconstruction error. We will express both Non-negative OMP (NOMP), and also Non-negative Lasso(NLASSO) below, compactly in terms of the following,

$$\mathbf{s} = \mathbf{H}\mathbf{x} \quad \text{and} \quad \mathbf{S} = \mathbf{H}\mathbf{H}^T \tag{8}$$

where $\mathbf{s}$ may be interpreted as the vector of cosine similarities between signals and dictionary atoms (since both dictionary elements and data points are unit normalized) and $\mathbf{S}$ is the (small) $K \times K$ matrix of inter-atom similarities. For example note that the objective function can be written as,

$$\begin{aligned} R(\mathbf{w}) &= \|\mathbf{x} - \mathbf{H}^T\mathbf{w}\|^2 = \|\mathbf{x}\|^2 - 2\mathbf{w}^T\mathbf{s} + \mathbf{w}^T\mathbf{S}\mathbf{w} \\ &= 1 - \mathbf{w}^T(\mathbf{s} + \mathbf{u}) \quad \text{where } \mathbf{u} = \mathbf{s} - \mathbf{S}\mathbf{w} \end{aligned} \tag{9}$$

Let $\mathcal{A} \subseteq \{1 \ldots K\}$ denote the support of a candidate solution $\mathbf{w}$, i.e., $\mathcal{A} = \{i : w_i > 0\}$. NOMP starts with $\mathbf{w} = 0$, $\mathcal{A} = \{\}$ and builds the solution by greedily adding variables that help reduce the current residual, $\mathbf{r} = \mathbf{x} - \mathbf{H}^T\mathbf{w} = \mathbf{x} - \sum_{i\in\mathcal{A}} \mathbf{h}_i w_i$. We use $\mathcal{A}^c$ to denote the complement of $\mathcal{A}$. Let $j \in \mathcal{A}^c$ be a candidate variable for inclusion. Its quality is measured how much, acting alone, it can help reduce the current residual. Since $\|\mathbf{h}_j\|_2 = 1$ we can easily see the following: $\arg\min_{w_j\geq 0} \|\mathbf{r} - \mathbf{h}_j w_j\|^2 = \arg\min_{w_j\geq 0}(w_j - (\mathbf{r}^T\mathbf{h}_j))^2$, and hence the minimizing value of $w_j$ is $w_j = \max(\mathbf{r}^T\mathbf{h}_j, 0)$. Furthermore, note that $\mathbf{r}^T\mathbf{h}_j = \mathbf{x}^T\mathbf{h}_j - (\mathbf{H}\mathbf{h}_j)^T\mathbf{w} = u_j$, where $\mathbf{u}$ was defined in Eqn 9. Therefore the OMP selection criteria, quite simply, evaluates to $(u_j - u_{j+})^2$ for a candidate variable $j$, where we use the notation $u_{j+} = \max(u_j, 0)$. Equivalently, the reduction in residual norm offered is $u_{j+}^2 - 2u_j u_{j+}$ which we can maximize over. As in unconstrained OMP, we now digest the chosen variable into $\mathbf{A}$ and then reoptimize over it with cylic coordinate descent. Given inputs $\mathbf{s}, \mathbf{S}, \gamma$ and convergence parameters $\epsilon, \tau$, the entire algorithm, can be organized around very simple and efficient update rules as follows,

```
Initialize $\mathcal{A} = \{\}, \mathbf{w} = \mathbf{0}_k, \mathbf{u} = \mathbf{s}$, reconstruction error $R = 1$.
while $|\mathcal{A}| < \gamma$
    ◇ Variable selection: $j^\star = \arg\max_{j\in\mathcal{A}^c} \left(u_{j+}^2 - 2u_j u_{+j}\right)$
    ◇ If $\left(u_{j^\star+}^2 - 2u_{j^\star} u_{+j^\star}\right) \leq \epsilon$ return, else continue below.
    ◇ $\mathcal{A} = \mathcal{A} \cup \{j^\star\}$. Refit over $\mathcal{A}$ by cyclic coordinate descent:
        – Cycle over $j \in \mathcal{A}$, reoptimizing $w_j$ as follows:
            * $\delta_j = w_j$, $w_j = \max(u_j + w_j, 0)$, $\delta_j = w_j - \delta_j$
            * $\mathbf{u} = \mathbf{u} - \mathbf{S}_j \delta_j$
        – $R^{old} = R$, $R = 1 - \sum_{j\in\mathcal{A}} w_j\,(s_j + u_j)$
        – Converge if $(R - R^{old}) < \tau R^{old}$ else run another cycle
          of coordinate descent over $j \in \mathcal{A}$.
```

The coordinate descent procedure for refitting is guaranteed to converge using standard BCD convergence results [3]. Given inputs $\mathbf{x}$ and $\mathbf{H}$, NOMP offers monotonic decrease in the reconstruction error of $\mathbf{x}$. In the dictionary learning setting, however, what we cannot guarantee is global decrease with respect to the previously found dictionary. Nonetheless, as we report in section 6, in hundreds of experiments on real world data, we always saw stable descent behavior.

### 3.2 Non-negative Lasso (NLASSO)

We rewrite the NLASSO problem, Eqn 7 with $p = 1$, equivalently as:

$$\arg\min_{\mathbf{w}\in\mathbb{R}^K} R(\mathbf{w}) + \delta(\mathbf{w}) \tag{10}$$

which is a composite objective comprising of a smooth term $R(\mathbf{w})$ and a non-smooth indicator function for the simplex defined as $\delta(\mathbf{w}) = 0$ if $\mathbf{w} \geq 0, \|\mathbf{w}\|_1 \leq \gamma$ and $\infty$ otherwise. We use an accelerated proximal method, FISTA [2], to handle such an objective function. In proximal methods, the idea is to linearize the smooth component, $R$, around the current iterate (say $\mathbf{w}^t$), and minimize

$$\min_{\mathbf{w}\in\mathbb{R}^K} R(\mathbf{w}^t) + \nabla R(\mathbf{w}^t)(\mathbf{w} - \mathbf{w}^t) + \frac{L}{2}\|\mathbf{w} - \mathbf{w}^t\|_2^2 + \delta(\mathbf{w})$$

where $\nabla R(\mathbf{w}) = \mathbf{s} - \mathbf{S}\mathbf{w}$ (recall the notation from Eqn. 8). The third term above, called proximal term, keeps the update in a neighborhood of the current iterate $\mathbf{w}^t$ where $R$ is close to its linear approximation. $L > 0$ is a parameter, which should essentially be an upper bound on the Lipschitz constant of $\nabla R$ and is typically set with a linesearch. We can rewrite this problem as, $\mathbf{w}^\star = \min_{\mathbf{w}\in\mathbb{R}^K} \frac{1}{2}\|\mathbf{w}-\mathbf{u}\|_2^2 + \delta(\mathbf{w})$ where $\mathbf{u} = \mathbf{w}^t - \frac{1}{L}\nabla R(\mathbf{w}^t)$. The above minimization functional (mapping $\mathbf{u}$ to $\mathbf{w}^\star$) is also called the *proximal operator*. For our purposes, it reduces to the projection operator to the convex set $\mathbf{w} \geq 0, \|\mathbf{w}\|_1 \leq \gamma$. An efficient linear time algorithm for such a projection onto the simplex is given in [8]. The rest of the FISTA details remain unchanged; we point the reader to [2] for more details. It is easy to see that we can also equivalently solve the more familiar penalized form of Lasso: $\arg\min_{\mathbf{w}} \left(R(\mathbf{w}) + \gamma' \sum_j w_j\right) + \delta'(\mathbf{w})$ for some one-to-one correspondence between $\gamma'$ and $\gamma$ and with $\delta'$ simply being an indicator function for the non-negative orthant $\mathbf{w} \geq 0$. In this case, the associated prox operator simply returns $(\mathbf{w}^t - \frac{1}{L}\nabla R(\mathbf{w}^t) - \frac{\gamma'}{L})_+$. Furthermore, due to our dictionary and data normalization, Lasso screening tests proposed very recently [19] can be immediately adapted to the non-negative case and applied to discard elements of $\mathbf{w}$ upfront that are guaranteed to be zero-valued.

## 4. SPARSE DICTIONARY LEARNING

We now discuss the dictionary learning phase where we update $\mathbf{H}$. Our strategy for this phase is reminiscent of the Hierarchical Alternating Least Squares (HALS) algorithm [6] which solves rank-one minimization problems to sequentially update both the rows of $\mathbf{H}$ as well as the columns of $\mathbf{W}$ (while we perform parallel row updates for $\mathbf{W}$ by invoking the sparse solvers developed in Section 3).

Let us denote the data matrix as $\mathbf{X} = (\mathbf{x}_1 \dots \mathbf{x}_N)^T \in \mathbb{R}^{N\times D}$ and let $\mathbf{v}_k$ be the $k^{th}$ column of $\mathbf{W}$, i.e., $\mathbf{W} = (\mathbf{v}_1 \dots \mathbf{v}_K)$. Let $\mathbf{R}_k = \left(\mathbf{X} - \sum_{i:i\neq k} \mathbf{v}_i \mathbf{h}_i^T\right)$ be the residual matrix excluding the $k^{th}$ atom. We denote the constraint set for learning dictionaries as

$$\mathcal{C} = \{\mathbf{h} \in \mathbb{R}^D \text{ such that } \mathbf{h} \geq 0, \|\mathbf{h}\|_0 \leq \nu, \|\mathbf{h}\|_2 = 1\}$$

The BCD subproblem of optimizing $\mathbf{h}_k$ keeping other atoms fixed becomes a rank-one matrix approximation problem,

$$\mathbf{h}_k = \arg\min_{\mathbf{h}\in\mathcal{C}} \frac{1}{N}\left\|\mathbf{R}_k - \mathbf{v}_k\mathbf{h}^T\right\|_{fro}^2 + \lambda\|\mathbf{h} - \mathbf{h}_k^0\|_2^2 \qquad (11)$$

where $\|\mathbf{A}\|_{fro}$ denotes the Frobenius norm of matrix $\mathbf{A}$. It is easy to see that this reduces to a projection problem, $\mathbf{h}_k = \arg\min_{\mathbf{h}\in\mathcal{C}} \|\mathbf{h} - \mathbf{q}_k\|_2^2$, where the vector $\mathbf{q}_k$ is given by,

$$\mathbf{q}_k = \frac{\frac{1}{N}\mathbf{R}_k^T\mathbf{v}_k + \lambda\mathbf{h}_0}{\frac{1}{N}\|\mathbf{v}_k\|_2^2 + \lambda} \qquad (12)$$

Note from the numerator that the residual term and the prior compete for contributing to $\mathbf{q}_k$.

**Efficient computation of Residual matrix times a vector:** Since $\mathbf{R}_k$ is dense, we want to avoid computing it explicitly and instead use the following,

$$\mathbf{R}_k^T\mathbf{v}_k = \mathbf{X}^T\mathbf{v}_k - \sum_{i\neq k}\mathbf{h}_i(\mathbf{v}_i^T\mathbf{v}_k) \qquad (13)$$

The second term, $\sum_{i\neq k}\mathbf{h}_i(\mathbf{v}_i^T\mathbf{v}_k)$, can be expressed compactly and efficiently as follows: Let $\mathbf{G} = \mathbf{W}^T\mathbf{W}$, which is a small $K \times K$ matrix, and let $\mathbf{g} = \mathbf{G}_k$, its $k^{th}$ column. Set $\mathbf{g}_k = 0$. Then its easy to see that $\mathbf{H}^T\mathbf{g} = \sum_{i\neq k}\mathbf{h}_i(\mathbf{v}_i^T\mathbf{v}_k)$. Therefore, we have,

$$\mathbf{q}_k = \frac{\frac{1}{N}\mathbf{R}^T\mathbf{v}_k + \lambda\mathbf{h}_0}{\frac{1}{N}\|\mathbf{v}_k\|_2^2 + \lambda} = \frac{\frac{1}{N}\left(\mathbf{X}^T\mathbf{v}_k - \mathbf{H}^T\mathbf{g}\right) + \lambda\mathbf{h}_0}{\frac{1}{N}\mathbf{G}_{kk} + \lambda} \qquad (14)$$

**Projection to the set $\mathcal{C}$ of Unit Norm Non-negative Sparse Vectors:** We now consider the following projection problem: $\mathbf{h}^\star = \arg\min_{\mathbf{h}\in\mathcal{C}} \|\mathbf{h} - \mathbf{q}\|_2^2$. First, let $\mathbf{h}$ be any $\nu$-sparse vector in $\mathbb{R}^D$ and let $\mathcal{I}(\mathbf{h})$ be its support. Let $i_1 \dots i_D$ be a permutation of the integer set $1 \leq i \leq D$ such that $q_{i_1}, \dots q_{i_D}$ is in sorted order and define $\mathcal{I}^\star = \{i_1, \dots, i_s\}$ where $s$ is the largest integer such that $s \leq \nu$ and $q_{i_1} > \dots > q_{i_s} > 0$. Now its easy to see that,

$$\arg\min_{\mathbf{h}\in\mathcal{C}} \|\mathbf{h} - \mathbf{q}\|_2^2 = \arg\max_{\mathbf{h}\in\mathcal{C}} \sum_{i\in\mathcal{I}(\mathbf{h})} h_i q_i \leq \arg\max_{\mathbf{h}\geq 0, \|\mathbf{h}\|_2=1} \sum_{i\in\mathcal{I}^\star} h_i q_i$$

Therefore, the exact solution to this problem given by,

$$\mathbf{h}^\star_{\mathcal{I}^\star} = \arg\min_{\mathbf{h}\in\mathbb{R}^{|\mathcal{I}^\star|}, \|\mathbf{h}\|_2=1} \|\mathbf{h} - \mathbf{q}_{I^\star}\|^2 = \frac{\mathbf{q}_{I^\star}}{\|\mathbf{q}_{I^\star}\|}, \; \mathbf{h}^\star_{\mathcal{I}^{\star c}} = 0 \qquad (15)$$

where $\mathcal{I}^{\star c}$ denotes the complement of $\mathcal{I}^\star$. Note that if $\mathcal{I}^\star$ turns out to be empty while running dictionary learning phase in BCD, $\mathbf{h}^\star_{\mathcal{I}^\star}$ is undefined, and in this case, we do not update the previous value of that variable. However, we never observed this to happen in practice.

Hence, all we need to solve the projection problem *exactly* is to find upto the top-$\nu$ non-negative values of $\mathbf{q}_k$ and normalize them to unit norm. We also note that in the dictionary learning phase we require the summary statistics: $\mathbf{G} = \mathbf{W}^T\mathbf{W} \in \mathbb{R}^{K\times K}$ and columns of the matrix $\mathbf{X}^T\mathbf{W} \in \mathbb{R}^{D\times K}$.

## 5. IMPLEMENTATION DETAILS

We briefly recap the developments so far and provide an outline for this section. Our parallelization strategy hinges on two observations: (a) the objective function is separable in the sparse coding variables (rows of $\mathbf{W}$) which therefore can be optimized in an embarrassingly parallel fashion, and (b) by enforcing hard sparsity on the dictionary elements, we turn $\mathbf{H}$ into an object that can be manipulated very efficiently in memory. Our implementation is carefully organized around efficient matrix computations and one-pass aggregation of summary statistics. We begin by describing our data structures (summarized in Table 1) and then outline our multicore implementation which requires that the data matrix should fit in memory. We relax this assumption in our cluster implementation sketched in Figure 1 which runs each iteration of the algorithm by making a single pass over the data (i.e., one MapReduce job). Note that our approach generalizes to other large-scale matrix factorization problems where separability, sparsity and in-memory computation can be similarly exploited.

**Efficient Matrix Computations:** Table 1 records various matrices, their storage schemes, computation strategy and whether they are materialized in memory in our single-node multicore and Hadoop-based cluster implementation. We use compressed row storage for holding the read-only static sparse matrix $\mathbf{X}$. For $\mathbf{H}$, and $\mathbf{W}$ if it is materialized, we use a dynamic sparse matrix data structure which essentially is an array of pointers to the non-zero indices and values for each row, allowing sparsity structure of rows to change across the iterations. Note that sparsity "compresses" the bigger dimension ($D$) for $\mathbf{H}$ as opposed to the smaller dimension ($K$) for $\mathbf{W}$, making it reasonable to manipulate $\mathbf{H}$ in-memory. Also note that the maximum memory requirement for $\mathbf{W}$ can be bounded by $O(N\gamma)$ for NOMP, while NLASSO does not offer similar predictability beforehand.

**Table 1: Matrices: Storage and Materialization**

| Matrix | Dims | #Bytes | Storage | Multi-Core | Cluster |
|---|---|---|---|---|---|
| $\mathbf{X}$ | $N \times D$ | $16n_X$ | CRS | Y | N |
| $\mathbf{W}$ | $N \times K$ | $16N\gamma$ | Dyn. Sparse | Y/N | N |
| $\mathbf{H}$ | $K \times D$ | $16K\nu$ | Dyn. Sparse | Y | Y |
| $\mathbf{W}^T\mathbf{W}$ | $K \times K$ | $4K(K+1)$ | Dense Symm. | Y | Y |
| $\mathbf{HH}^T$ | $K \times K$ | $4K(K+1)$ | Dense Symm. | Y | Y |
| $\mathbf{X}^T\mathbf{W}$ | $D \times K$ | $8DK$ | Dense | N/Y | N |
| $\mathbf{HX}^T$ | $K \times N$ | $8NK$ | Dense | N | N |

Recall that in the parallel sparse coding phase, for NOMP and NLASSO, we can carry out computations using $\mathbf{s} = \mathbf{Hx}$ and $\mathbf{S} = \mathbf{HH}^T$ (see Eqn. 8). The matrix $\mathbf{HX}^T$ need not be materialized as its columns – which are the $\mathbf{s}$ vectors – can be computed on the fly, used, and then immediately discarded. The matrix $\mathbf{S}$ is a dense symmetric matrix whose lower half is stored in the standard lower-packed storage format, and computed by $K$ sparse matrix-vector products: $\mathbf{Hh}_1 \ldots \mathbf{Hh}_K$. Alternatively, matrix-vector products against $\mathbf{S}$ may be computed in time $O(K\nu)$ implicitly as $\mathbf{Sv} = \mathbf{H}(\mathbf{H}^T\mathbf{v})$. Likewise, note that dictionary learning phase requires $\mathbf{G} = \mathbf{W}^T\mathbf{W}$ and $\mathbf{X}^T\mathbf{W}$. It is profitable to express these matrices as the aggregation of rank-one matrices: $\mathbf{W}^T\mathbf{W} = \sum_{i=1}^N \mathbf{w}_i\mathbf{w}_i^T$ and $\mathbf{X}^T\mathbf{W} = \sum_{i=1}^N \mathbf{x}_i\mathbf{w}_i^T$. These matrices are the summary statistics we need for dictionary learning. Hence, a single pass over the data during the parallel sparse coding phase enables us to perform these aggregations, and prepare these matrices for the dictionary learning phase. Furthermore, in the aggregation above, we exploit the sparsity of $\mathbf{x}_i$ and $\mathbf{w}_i$ by updating only the non-zero cells. Finally, in the projection step of the dictionary learning phase, Eqn 15, we use priority queues for finding top $\nu$ non-negative elements in time $O(D\log(\nu))$.

**Single-Node In-memory Multicore Implementation:** We implemented parallel sparse coding using multithreading to utilize parallelism on a single multicore machine. The data matrix $\mathbf{X}$ is held in memory in its entirety. It is divided into as many blocks as the number of threads, and for each block, a single thread runs NOMP or NLASSO sequentially on the datapoints in that block. Also held in memory are the $K \times K$ dense symmetric matrices, $\mathbf{S} = \mathbf{HH}^T$ and $\mathbf{G} = \mathbf{W}^T\mathbf{W}$ which we assume are small. Assuming $\nu = O(K)$, $\mathbf{H}$ also similarly requires $O(K^2)$ storage and is held in memory. To maximize the data sizes that our implementation can accommodate, we considered two different execution plans: (a) **Plan 1**: The first plan does not materialize the matrix $\mathbf{W}$, but explicitly maintains the dense $D \times K$ matrix $\mathbf{X}^T\mathbf{W}$. As each invocation of NOMP or NLASSO completes, the associated sparse coding vector $\mathbf{w}_i$ is used to update the summary statistics matrices, $\mathbf{W}^T\mathbf{W}$ and $\mathbf{X}^T\mathbf{W}$, and then discarded, since everything needed for dictionary learning is contained in these summary statistics. When $DK \ll N\gamma$, this leaves more room to accommodate larger datasizes in memory. However, not materializing $\mathbf{W}$ means that NLASSO cannot exploit warm-starts from the sparse coding solutions found with respect to the previous dictionary. (b) **Plan 2**: In an alternate plan, we materialize $\mathbf{W}$ instead which consumes lesser memory if $N\gamma \ll DK$. We then serve the columns of $\mathbf{X}^T\mathbf{W}$, i.e., the vectors $\mathbf{X}^T\mathbf{v}_k$ in Eqn 14, by performing a sparse matrix-vector product on the fly. However, this requires extracting a column from $\mathbf{W}$ which is stored in a row-major dynamic sparse format. To make column extraction efficient, we utilize the fact that the indices of non-zero entries for each row are held in sorted order, and the associated value arrays conform to this order. Hence, we can keep a record of where the $i^{th}$ column was found for each row, and simply advance this pointer when the $(i+1)^{th}$ column is required. Thus, all columns can be served efficiently with one pass over $\mathbf{W}$. Two benefits of this plan are: (i) the matrix-vector product of $\mathbf{X}^T$ against the columns of $\mathbf{W}$ can be parallelized and (ii) NLASSO can use warm-starts. Thus, in this plan, both sparse coding and dictionary learning phases benefit from parallelism. To concretize the preceding discussion with some numbers: on a multicore machine with 16-GB RAM, accounting for the matrix storage given in Table 1, both plans can handle around 10 million documents assuming sparsity of 100 words, $\gamma = 5, \nu = 1000$, and 1000 topics need to be learnt. If 15000 topics need to learnt, the first plan can only handle about 2 million documents while the second plan can still manage 9 million by not materializing the dense $D \times K$ matrix $\mathbf{X}^T\mathbf{W}$.

Together, these considerations outlined above, lead to a highly efficient single-node implementation that forms the basis for the cluster version described next.

**Cluster Implementation:** Unlike the multi-core setting, our cluster implementation does not make the assumption that the data matrix $\mathbf{X}$ fits in main memory. Rather, $\mathbf{X}$ is stored in a distributed file system and our implementation makes one or more passes over it in parallel. Likewise, only small blocks of the large dense matrix $\mathbf{X}^T\mathbf{W}$ are ever materialized at a time and written to disk in parallel in every iteration. We continue making the assumption that the small matrices $\mathbf{W}^T\mathbf{W}$, $\mathbf{HH}^T$ and the highly sparse matrix $\mathbf{H}$, can be held in-memory on both the control and worker nodes. While there are many choices for the framework one could use to parallelize algorithms, we work with the Map-Reduce paradigm, which fits well with the embarrassingly parallel nature of the sparse coding phase.

The Map-Reduce (MR) programming model was designed to simplify the processing of large files on a parallel system through user-defined Map and Reduce primitives. A MR job consists of two phases: a *Map* phase and a *Reduce* phase. During the Map phase, the user-defined Map primitive transforms the input data into (key, value) pairs in parallel. These pairs are stored and then sorted by the system so as to accumulate all values for each key. During the Reduce phase, the user-defined Reduce primitive is invoked on each unique key with a list of all the values for that key; usually, this phase is used to perform aggregations. Finally, the results are output in the form of (key, value) pairs. Each key can be processed in parallel during the Reduce phase.

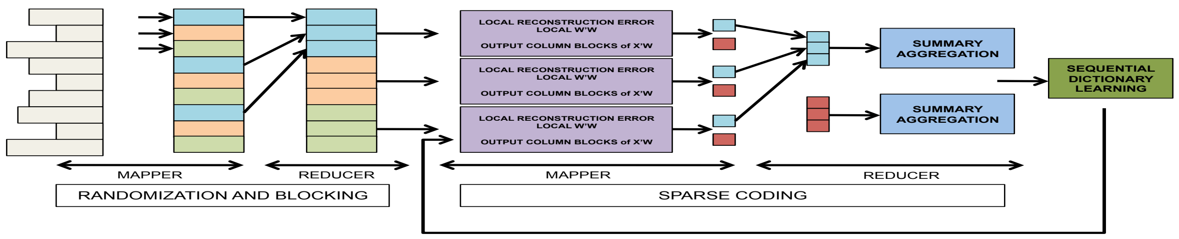

LOCAL RECONSTRUCTION ERROR
LOCAL W'W
OUTPUT COLUMN BLOCKS of X'W
SUMMARY AGGREGATION
SEQUENTIAL DICTIONARY LEARNING
MAPPER
REDUCER
RANDOMIZATION AND BLOCKING
SPARSE CODING

**Figure 1: MapReduce implementation**

Hadoop (`http://hadoop.apache.org/`), an open-source implementation of the MR programming model, has emerged as a vastly popular platform for parallelization in industry and academia. A user can perform parallel computations by submitting one or more MR jobs to Hadoop. One of the key advantages of Hadoop is that it is capable of running on large commodity clusters and recovering from both data as well as compute node failures.

Hadoop's implementation of the MR programming model targets executions where the input, intermediate (shuffle), and output datasets do not fit in aggregate main memory. Each MR job has to scan the input and intermediate datasets, which is time consuming. Furthermore, each job adds significantly to execution time in the form of startup costs. For these reasons, implementations that require the fewest number of MR jobs are ideal. Our cluster implementation below is driven by this goal.

Figure 1 presents the overall flow for the MR implementation of our algorithms. The execution proceeds in two phases, the preprocessing phase and the learning phase. The preprocessing phase is a one-time step whose objective is to re-organize the data for better parallel execution of the BCD algorithm. The learning phase is iterative and is coordinated by a control node that first spawns NOMP or NLASSO MR jobs on the worker nodes, then runs sequential dictionary learning and finally monitors the global mean reconstruction error to decide on convergence.

The preprocessing step consists of randomization and blocking. The goal of this step is to transform the data set into a set of "blocks" such that each block is a row-wise partition of the input matrix and all the blocks are roughly of the same size and sparsity, implying roughly equal units of work. We block the data set for the following reasons. First, during the learning phase, if one were to provide the input matrix to the Mappers one row at a time, majority of the time spent would go into performing I/O and not actual computation. To amortize the cost of performing I/O, we want to process a block of rows at a time. Second, ensuring these blocks are roughly of the same size and sparsity allows for a load balanced execution. Finally, blocking the input matrix amortizes the cost of writing out blocks of the $\mathbf{X}^T\mathbf{W}$ matrix during the learning phase by allowing one to partially aggregate this matrix for each input block, directly in memory. During the pre-processing step, the Mapper assigns each row to a random block and emits the block id as a key and row as a value. The Reducer then assimilates each block and writes it out to the file system. At the end of the pre-processing phase, we expect to have row-wise blocks of the input matrix that are roughly of same size and sparsity.

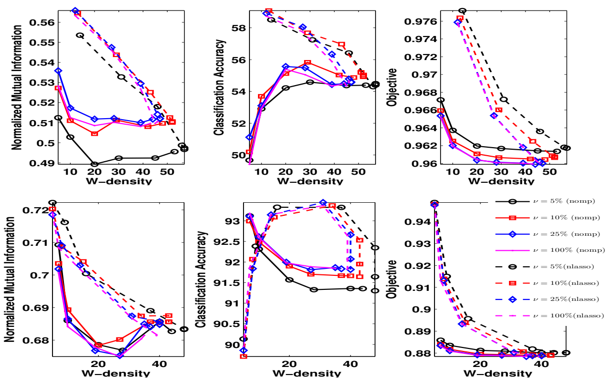
The subsequent learning phase proceeds iteratively with each iteration consisting of a parallel sparse coding phase and a sequential dictionary learning phase. The parallel sparse coding phase is implemented using *a single Map-Reduce job*. To explain its steps, we need the following notation: let $\mathbf{X}^r$ denote the $r^{th}$ block of the $\mathbf{X}$ and $R$ be the number of rows that were assigned to a block by the preprocessing phase. We refer to $R$ as the *row blocking parameter*. A Mapper for the parallel sparse coding phase accepts $\mathbf{X}^r$ and runs NOMP/NLASSO to obtain the associated sparse coding matrix $\mathbf{W}^r$ which can be materialized in memory since the blocks are small. We use the notation $\mathbf{W}^{r,i}$ to denote its $i^{th}$ column. For large $D, K$, the local summary statistics matrix $\mathbf{X}^{rT}\mathbf{W}^r$ cannot be materialized and written out in one go. Rather, the Mapper computes a set of $D \times C$ matrices, where $C \le K$: $\mathbf{Q}^{r,j} = X_r^T \left(\mathbf{W}^{r,(j-1)*C+1}, \dots \mathbf{W}^{r,j*C}\right)$ matrix for $j = 1 \dots \lceil \frac{K}{C} \rceil$. $C$ is the *column blocking parameter* of our implementation. The Mapper emits the block id $j$ as a key and column block $\mathbf{Q}^{r,j}$ as value. The Reducer then sums up these blocks over the row-blocks: $\mathbf{Q}^j = \sum_r \mathbf{Q}^{r,j}$ which gives the $j^{th}$ block of $C$ columns of the global summary statistics matrix $\mathbf{X}^T\mathbf{W}$. This mechanism to write out column blocks of the local summary statistics matrices (and not all columns) has the following benefits: First, we can constrain the size of the aggregations that need to take place on the Reducers. Second, we get parallelism by having each Reducer perform a complete aggregation for a portion of the columns. In addition to producing the summary statistics matrix, the MR job also writes out the aggregated reconstruction error and $\mathbf{W}^T\mathbf{W}$ matrix. The output of the MR job is delivered to the sequential dictionary learning phase, at the end of which we check for convergence and proceed to the next iteration if necessary. Note that each mapper can writeout its associated $\mathbf{W}^r$ as well to help jumpstart NLASSO in the next iteration.

## 6. EMPIRICAL STUDIES

We begin with small scale experiments on two classic document collections, the 20 newsgroups collection ($N = 19228$, $D = 18607$ and $K = 20$) and TDT news corpus ($N = 9394$, $D = 19528$ and $K = 30$), that come with manually generated labels with respect to which quality measurements may be made. We then report scalability experiments for our multicore and Hadoop implementations on much larger datasets (New York times and an extended Pubmed Corpora) where extrinsic evaluation cannot be done due to the high cost of labeling.

**Statistical Performance on Small Datasets:** We explore the effect of double sparsity on the statistical performance of the non-negative sparse coding. We varied dictionary sparsity, covering 5% to 100% of the full data dimensionality. The coding sparsity was varied from 0.05 to 1.0 interpreted as the fraction of the full dictionary size $K$ that NOMP is allowed to use for coding. For NLASSO, the same fractions were used literally as the $\gamma$ values. Empirically, in this range, NLASSO gave similar degrees of sparsity as NOMP. For each choice of $\gamma, \nu$, we ran the learning algorithms 10 times with different initialization and averaged the following: (a) the clustering quality in terms of normalized mutual information (NMI) obtained by assigning cluster $\arg\max_j w_{ij}$ to document $i$, (b) the classification accuracy returned by running linear logistic regression on 5% labeled samples after unsupervised sparse coding, to emulate a semi-supervised learning scenario, and (c) the final objective values attained at convergence. Each choice of $\gamma$ was associated with the mean sparsity of the final $\mathbf{W}$ across different random initializations.

**Figure 2: Results on 20NG (top), TDT2 (bottom)**



In Figure 2, we plot NMI, Classification accuracy of Logistic Regression and Objective values as a function of percentage of non-zeros in $\mathbf{W}$ for different degrees of percentage of non-zeros in $\mathbf{H}$, for both NOMP and NLASSO. We make the following observations: (1) Our Sparse NMF (both NOMP and NLASSO) demonstrates substantially better clustering and classification performance than NMF which corresponds to the rightmost point on the curves for $\nu = 100\%$. Furthermore, we never observed an increase in objective function with NOMP in these experiments; all runs smoothly converged. (2) Interestingly, we find that on both datasets, NOMP gives substantially better reconstruction error at similar sparsity levels than NLASSO. However, the induced sparse representation does not translate into better clustering performance or classification accuracy. This might seem counter-intuitive at first, but can be reasoned as follows. The NOMP hypothesis space is larger since it does not restrict the magnitude of the weights as NLASSO does. This leads to better mean reconstruction error rates at similar sparsity levels. Indeed, reconstruction error alone does not reflect model utility for these tasks since we know that SVD can do even better without non-negativity and sparsity constraints. The presence of additional weight shrinkage in NLASSO appears to act as an effective regularizer that leads to better statistical performance. (3) We see that the $\nu = 10\%$ curve performs as well or better than the $\nu = 100\%$ curve. This implies a 10-fold storage reduction for $\mathbf{H}$ strongly supporting our choice to maintain it in memory as a light-weight sparse matrix. (4) In Table 2, as a sanity check, we compare clustering performance against popular alternatives. The parameters are $\gamma = 0.05$ (to be interpreted as fraction of K for NOMP) and $\nu = 0.1D$. We see that Sparse NMF with NLASSO performs quite competitively.

**Figure 3: Multicore Performance: NYTimes data**

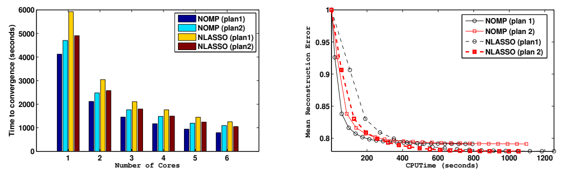


**Table 2: Clustering Quality**

| Algorithm | 20NG | TDT2 |
|---|---|---|
| pLSA | 51.5 (2.8) | 60.3 (1.6) |
| LDA | 49.4 (1.3) | 66.7 (1.5) |
| K-means | 41.4 (14.4) | 70.6 (2.3) |
| NMF | 51.0 (1.1) | 68.1 (2.1) |
| SparseNMF (NLASSO) | 55.4 (2.4) | 72.2(2.5) |
| SparseNMF (NOMP) | 51.3 (2.0) | 70.9 (2.2) |

**Multicore Performance:** We next consider computational performance on a commodity hexacore machine with 4GB RAM. For these experiments, we work with the popular NY times corpus (available from UCI Machine Learning Repository) comprising of $N = 299752$ documents indexed over a vocabulary of $D = 102660$ words, with about 70-million non-zero entries in the document-term matrix. We set $\nu = 1000$, $\gamma = 5$ for NOMP and 5 for NLASSO and ran our algorithms for $K = 100$ while increasing the number of cores from 1 to 6. The total time to convergence for both the execution plans is shown in Figure 3 (left). We make the following observations. The use of 6-cores offers the following speedups over a single core: 5.2x for NOMP (plan 1), 4.3x for NOMP (plan 2), 4.7x for both NLASSO (plan 1) and NLASSO (plan 2). Thus, we see near-linear speedups with increasing number of cores. NOMP prefers plan 1 because it cannot utilize warm starts and because dictionary learning is extremely fast (about 0.5 seconds for updating 100 topics per iteration) since it only needs to perform a top-$\nu$ operation followed by simple normalization step. NLASSO, on the other hand, prefers plan 2 since warm starts allow it to converge faster. As shown in Figure 3 (right), its iterations start to take lesser and lesser time. The dictionary learning phase for plan 2 takes about 35 seconds in each iteration for 1 core, which reduces to about 13 seconds with 6 cores, due to parallel matrix-vector products as described in section 5.

**Hadoop Scaleout:** These experiments were performed on IBM's Nadal cluster, which has 48 cores distributed across 12 nodes with 4 GB of RAM per core. All the nodes were running RHEL 5 (kernel 2.6.18-164.el5) with IBM Java v1.6.0 and Hadoop 0.20.2. We measured execution time of the Hadoop implementation on an extended PubMed dataset (also available at UCI repository) containing 106 million documents, 141K words, with more than 6 Billion non-zeros in the document term matrix, as we varied various algorithm and hardware parameters. For these experiments, the row blocking parameter $R$ = 100K rows, the column blocking parameter $C$ = 10 columns, $\gamma$ = 5 and $\nu$ = 1000, unless otherwise noted.

**(a) Varying number of cores:** Figures 4 and 5 present execution time for an iteration of NOMP, an iteration of NLASSO, and the blocking phase that is common to both algorithms, for different dataset sizes, as we vary the number of cores from 16 to 48. Speedup for the blocking phase (that is common to both algorithms) varies from 1.53x for 8.2M rows to 2.81x for 106M rows, the maximum possible speedup here being 3x (as we only vary the number of cores from 16 to 48). For 8.2M rows, the speedup is roughly half that of the maximum possible speedup as the data set is small and the system is under utilized. For 106M rows, all cores take part in I/O and the blocking procedure, improving scalability. The blocking phase is thus very efficient and is expected to continue to scale well to larger datasets.

As for the performance of an iteration of NOMP, the speedup is roughly 1.85x for 8.2M rows and 2.7x for 106M rows when $K = 100$ and 2.1x for 8.2M rows and 3.54x for 106M rows when $K = 200$. As for the performance of an iteration of NLASSO, the speedup is roughly 1.81x for 8.2M rows and 2.75x for 106M rows when $K = 100$ and 1.86x for 8.2M rows and 3.02x for 106M rows when $K = 200$. Keeping $K$ fixed, speedup improves with increasing number of rows as the execution changes from being I/O bound and under utilizing the system to being compute bound. Furthermore, keeping the number of rows fixed, speedup improves as we increase $K$ as there is more computation on offer to better utilize the available parallelism. We observe super linear speedup in some cases likely due to improvements in caching behavior as we increase the number of cores. Given that the execution is compute-bound on large datasets, we expect the implementation to continue to scale on larger number of cores provided the dataset is sufficiently large to make the execution compute-bound.

**Figure 4: Varying Number of Cores - 100 Topics**

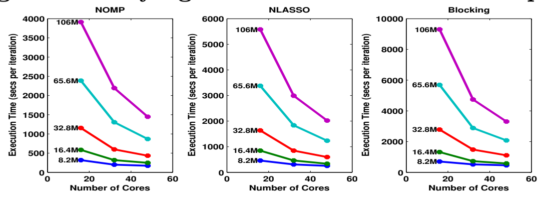
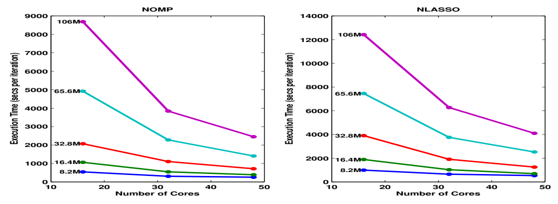
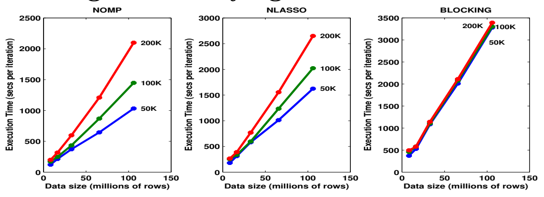


**Figure 5: Varying Number of Cores - 200 Topics**



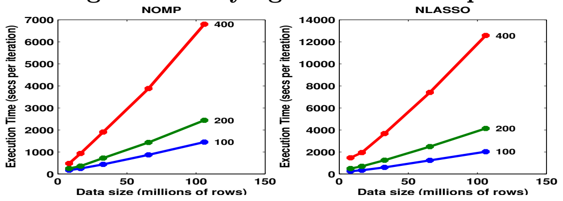
**(b) Varying row block size:** Figure 6 presents execution times for an iteration of NOMP, an iteration of NLASSO, and the blocking phase that is common to both algorithms, for different values of the row blocking parameter $R$, as we vary the number of rows in the dataset, keeping number of cores fixed at 48. The performance of the blocking phase improves marginally (from 496 to 373 for 8.2M rows and from 3392 to 3382 for 106M rows) as we increase the size of each block due to the fact that the system needs to sort a fewer keys with fewer blocks. As the performance of the blocking phase is not adversely affected by increasing data sizes, we expect the blocking phase to scale well to larger datasets. However, the performance of an iteration of NOMP and NLASSO varies significantly with varying $R$. For NOMP, seconds per iteration varies from 200 to 122 for 8.2M rows and from 2099 to 1032 for 106M rows. For NLASSO, seconds per iterations varies from 266 to 180 for 8.2M rows and from 2650 to 1624 for 106M rows. $R = 200\text{K}$ can improve performance by nearly 2x when compared to $R = 50\text{K}$. Given that performance of the blocking phase is relatively unaffected by the parameter $R$, and that the performance of the learning phase is heavily dependent on the value of $R$, picking the largest possible value for $R$ that allows for an in-memory execution will likely provide the best performance.

**Figure 6: Varying Row Block Size**



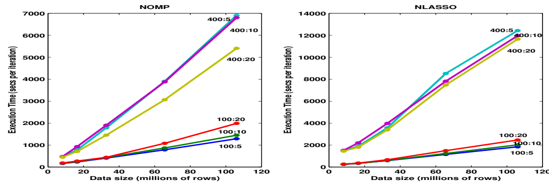
**(c) Varying number of topics:** Figure 7 presents execution times for an iteration of NOMP, an iteration of NLASSO, for different number of topics $K$, as we vary the data size, keeping number of cores fixed at 48. For NOMP, at 8.2M rows, execution time increases by 2.76x, and at 106M rows, execution time increases by 4.69x, when going from $K = 100$ to 400. The execution under utilizes the system at 8.2M rows and hence execution time grows sub-linearly with $K$. However, at 106M rows, execution time grows near-linearly as the execution is compute bound. Overall, execution time for NOMP grows near-linearly with $K$. For NLASSO, at 8.2M rows, execution time increases by 5.81x, and at 106M rows, execution time increases by 6.22x, when going from $K = 100$ to 400. Thus, execution time for NLASSO grows super-linearly with $K$. We can explain these observations as follows. The per iteration cost of NOMP is linear in $K$, involving search over $K$ atoms, and the total number of iterations is atmost $\gamma$. For NLASSO, the gradient computation involves the dense matrix-vector product $\mathbf{Sw}$ whose complexity depends on the sparsity of $\mathbf{w}$, but for dense iterates is $O(K^2)$. For large $K$, linear scaling behavior can be recovered for NLASSO by not materializing $\mathbf{S}$, but rather computing $\mathbf{Sw}$ using two *sparse* matrix-vector products $\mathbf{H}(\mathbf{H}^T\mathbf{w})$ which has $O(K\nu)$ cost. Convergence results from [2] show that NLASSO, implemented using FISTA, will take about $O(\sqrt{\frac{1}{\epsilon}})$ iterations to return an $\epsilon$ optimal solution, independent of $K$.

**Figure 7: Varying Number of Topics**



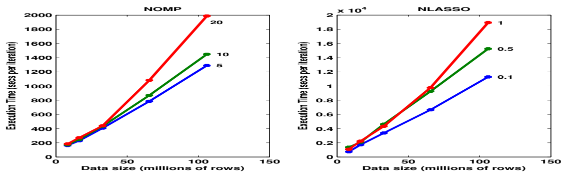
**(d) Varying column block size:** Figure 8 presents execution times for an iteration of NOMP, an iteration of NLASSO, for different number of topics $K$ (100,400) and column blocking parameter $C$ (5,10,20), as we vary the data size, keeping number of cores fixed at 48. For both NOMP and NLASSO, we observe an interesting trend. When $K = 100$, $C = 5$ results in the least execution time as it allows one to leverage a larger number of cores in the Reduce phase of the MR job. However, $K = 400$, $C = 5$ results in the largest execution time as it results in an excessively larger number of Reduce tasks, while $C = 20$ results in the least execution time. These indicate that one must carefully select the parameter $C$ based on the cluster configuration and desired number of topics.

**Figure 8: Column Block Size and Topics**



**(e) Varying coding sparsity:** Figure 9 presents execution times for an iteration of NOMP, an iteration of NLASSO, for $K = 400$, as we vary the data size, keeping number of cores fixed at 48. For NOMP, varying $\gamma$ from 5 to 20 increases execution time by roughly 1.82x for 106M rows. For NLASSO, varying $\gamma$ from 0.1 to 1 increases execution time by roughly 1.67x for 106M rows. Thus, for both algorithms, execution time grows sub-linearly with coding sparsity. The per iteration cost of NLASSO is independent of $\gamma$ and as outlined in (c), it has linear dependence on $\gamma$. While the refitting step of NOMP has superlinear dependence on $\gamma$, its values are such a small fraction of $K$ that this dependence is "washed out".

**Figure 9: Varying Coding Sparsity**



**Comparison to related work:** On the extended Pubmed corpus with over 106-million rows and more than 6 billion non-zero entries, with $K = 100$ and parameters reported in this section, our fastest algorithms running on 12 nodes (with 48 cores) are expected to converge and complete execution in less than 10 hours. To the best of our knowledge, this is the largest demonstration of sparse coding/dictionary learning algorithms reported to date. Comparing this performance with other related published work has several caveats due to major differences in convergence properties of algorithms benchmarked, actual cluster configuration, characteristics of datasets used, as well as innumerable other lower-level implementation details. In particular, we are not aware of any publically available Hadoop implementation of Sparse NMF and related dictionary learning algorithms to compare against. Nonetheless, to put our performance into perspective, we collect some published results in this section. [15] report running the standard NMF [14] algorithms on a proprietary dataset with around 4.38-billion non-zeros with $K = 10$, on a Hadoop cluster with unspecified configuration. They report 7-hours per iteration each requiring multiple Map-Reduce jobs, which appears to be significantly less efficient than our implementation. [18] parallelize LDA and report about 4.1 hours on 8.2 million Pubmed documents, with $K = 2000$, on a non-dedicated Hadoop cluster with 50 nodes. [17] provide a different MPI-based parallel LDA implementation and report 10-hours for the same dataset on 1024 processors.

**Acknowledgments:** We thank Prabhanjan Kambadur for several insightful conversations. This research is continuing through participation in the Social Media in Strategic Communication (SMISC) program sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under Agreement Number W911NF-12-C-0028.

## 7. REFERENCES

[1] S. Arora, R. Ge, R. Kannan, and A. Moitra. Computing a nonnegative matrix factorization – provably. In *STOC*, 2012.

[2] A. Beck and M. Teboulle. Gradient-based algorithms with applications to signal recovery problems. *Convex Opt. in Sig. Proc. Comm., Camb. Univ. Press*, 2010.

[3] D. Bertsekas. *Non-linear Programming*. Athena Scientific, 1999.

[4] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *JMLR*, 3:993–1022, 2003.

[5] Peter Buhlmann and Sara Van De Geer. *Statistics for High-Dimensional Data*. Springer, 2011.

[6] A. Cichocki, R. Zdunek, A. H. Phan, and S. Amari. *Nonneg. Matrix and Tensor Fact.* Wiley, 2009.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[8] J. Duchi, S. S. Schwartz, Y. Singer, and T. Chandra. Efficient projections onto the l1-ball for learning in high dimensions. In *ICML*, 2008.

[9] M. Elad. *Sparse and Redundant Representations.* Springer, 2010.

[10] A. Ghoting, P. Kambadur, E. Pednault, and R. Kannan. Nimble: a toolkit for the implementation of parallel data mining and machine learning algorithms on mapreduce. In *KDD*, 2011.

[11] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Declarative machine learning on mapreduce. In *ICDE*, 2011.

[12] P. Hoyer. Non-negative matrix factorization with sparseness constraints. *JMLR*, 5:1457–1469, 2004.

[13] R. Jenatton, J. Mairal, G. Obozinski, and F. Bach. Proximal methods for sparse hierarchical dictionary learning. In *ICML*, 2010.

[14] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 1999.

[15] C. Liu, H. Yang, J. Fan, L. He, and Y. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW*, 2010.

[16] J. Mairal, F. Bach, J. Ponce, and G. Sapiro. Online learning for matrix factorization and sparse coding. *JMLR*, 11:19–60, 2010.

[17] D. Newman, A. Ascuncion, P. Smyth, and M. Welling. Distributed algorithms for topic models. In *JMLR*, 2010.

[18] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *VLDB*, 2010.

[19] Z. J. Xiang, H. Xu, and P. J. Ramadge. Learning sparse representations of high-dimensional data on large-scale dictionaries. In *NIPS*, 2011.