A High-Performance Parallel Algorithm for Nonnegative Matrix Factorization

Ramakrishnan Kannan
Georgia Tech
rkannan@gatech.edu

Grey Ballard
Sandia National Laboratories
gmballa@sandia.gov

Haesun Park
Georgia Tech
hpark@cc.gatech.edu

Abstract

Non-negative matrix factorization (NMF) is the problem of determining two non-negative low rank factors **W** and **H**, for the given input matrix **A**, such that $\mathbf{A} \approx \mathbf{WH}$. NMF is a useful tool for many applications in different domains such as topic modeling in text mining, background separation in video analysis, and community detection in social networks. Despite its popularity in the data mining community, there is a lack of efficient distributed algorithms to solve the problem for big data sets.

We propose a high-performance distributed-memory parallel algorithm that computes the factorization by iteratively solving alternating non-negative least squares (NLS) subproblems for **W** and **H**. It maintains the data and factor matrices in memory (distributed across processors), uses MPI for interprocessor communication, and, in the dense case, provably minimizes communication costs (under mild assumptions). As opposed to previous implementations, our algorithm is also flexible: (1) it performs well for both dense and sparse matrices, and (2) it allows the user to choose any one of the multiple algorithms for solving the updates to low rank factors **W** and **H** within the alternating iterations. We demonstrate the scalability of our algorithm and compare it with baseline implementations, showing significant performance improvements.

1. Introduction

Non-negative Matrix Factorization (NMF) is the problem of finding two low rank factors $\mathbf{W} \in \mathbb{R}_+^{m\times k}$ and $\mathbf{H} \in \mathbb{R}_+^{k\times n}$ for a given input matrix $\mathbf{A} \in \mathbb{R}_+^{m\times n}$, such that $\mathbf{A} \approx \mathbf{WH}$. Here, $\mathbb{R}_+^{m\times n}$ denotes the set of $m \times n$ matrices with non-negative real values. Formally, the NMF problem [24] can be defined as

$$\min_{\mathbf{W}\geqslant 0, \mathbf{H}\geqslant 0} \|\mathbf{A} - \mathbf{WH}\|_F, \tag{1}$$

where $\|\mathbf{X}\|_F = (\sum_{ij} x_{ij}^2)^{1/2}$ is the Frobenius norm.

NMF is widely used in data mining and machine learning as a dimension reduction and factor analysis method. It is a natural fit for many real world problems as the non-negativity is inherent in many representations of real-world data and the resulting low rank factors are expected to have natural interpretation. The applications of NMF range from text mining [22], computer vision [11], and bioinformatics [13] to blind source separation [3], unsupervised clustering [16, 17] and many other areas. In the typical case, $k \ll \min(m, n)$; for problems today, m and n can be on the order of millions or more, and k is on the order of tens or hundreds.

There is a vast literature on algorithms for NMF and their convergence properties [15]. The commonly adopted NMF algorithms are – (i) Multiplicative Update (MU) [24] (ii) Hierarchical Alternative Least Squares (HALS) [3, 10] (iii) NMF using Block Principal Pivoting (ANLS-BPP) [14], and (iv) Stochastic Gradient Descent (SGD) Updates [8]. Most of the algorithms in NMF literature are based on alternately optimizing each of the low rank factors **W** and **H** while keeping the other fixed, in which case each subproblem is a constrained convex optimization problem. Subproblems can then be solved using standard optimization techniques such as projected gradient or interior point; a detailed survey for solving such problems can be found in [15, 26]. In this paper, our implementation uses a fast active-set based method called Block Principal Pivoting (BPP) [14], but the parallel algorithm proposed in this paper can be easily extended for other algorithms such as MU and HALS.

Recently with the advent of large scale internet data and interest in Big Data, researchers have started studying scalability of many foundational machine learning algorithms. To illustrate the dimension of matrices commonly used in the machine learning community, we present a few examples. Nowadays the adjacency matrix of a billion-node social network is common. In the matrix representation of a video data, every frame contains three matrices for each RGB color, which is reshaped into a column. Thus in the case of a 4K video, every frame will take approximately 27 million rows (4096 row pixels x 2196 column pixels x 3 colors). Similarly, the popular representation of documents in text mining is a bag-of-words matrix, where the rows are the dictionary and the columns are the documents (e.g., webpages). Each entry A_{ij} in the bag-of-words matrix is generally the frequency count of the word i in the document j. Typically with the explosion of the new terms in social media, the number of words spans to millions.

To handle such high dimensional matrices, it is important to study low rank approximation methods in a data-distributed environment. For example, in many large scale scenarios, data samples are collected and stored over many general purpose computers, as the set of samples is too large to store on a single machine. In this case, the computation must also be distributed across processors. Local computation is preferred as local access of data is much faster than remote access due to the costs of interprocessor communication. However, for low rank approximation algorithms, like MU, HALS, and BPP, some communication is necessary.

The simplest way to organize these distributed computations on large data sets is through a MapReduce framework like Hadoop, but this simplicity comes at the expense of performance. In particular, most MapReduce frameworks require data to be read from and written to disk at every iteration, and they involve communication-intensive global, input-data shuffles across machines.

PPoPP'16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851152

In this work, we present a much more efficient algorithm and implementation using tools from the field of High-Performance Computing (HPC). We maintain data in memory (distributed across processors), take advantage of optimized libraries for local computational routines, and use the Message Passing Interface (MPI) standard to organize interprocessor communication. The current trend for high-performance computers is that available parallelism (and therefore aggregate computational rate) is increasing much more quickly than improvements in network bandwidth and latency. This trend implies that the relative cost of communication (compared to computation) is increasing.

To address this challenge, we analyze algorithms in terms of both their computation and communication costs. The two major tasks of the NMF algorithm are (a) performing matrix multiplications and (b) solving many Non-negative Least Squares (NLS) subproblems. In this paper, we use a carefully chosen data distribution in order to use a communication-optimal algorithm for each of the matrix multiplications, while at the same time exploiting the parallelism in the NLS problems. In particular, our proposed algorithm ensures that after the input data is initially read into memory, it is *never* communicated; we communicate only the factor matrices and other smaller temporary matrices among the p processors that participate in the distributed computation. Furthermore, we prove that in the case of dense input and under the assumption that $k \leqslant \sqrt{mn/p}$, our proposed algorithm *minimizes* bandwidth cost (the amount of data communicated between processors) to within a constant factor of the lower bound. We also reduce latency costs (the number of times processors communicate with each other) by utilizing MPI collective communication operations, along with temporary local memory space, performing $O(\log p)$ messages per iteration, the minimum achievable for aggregating global data.

Fairbanks et al. [5] present a parallel NMF algorithm designed for multicore machines. To demonstrate the importance of minimizing communication, we consider this approach to parallelizing an alternating NMF algorithm in distributed memory. While this naive algorithm exploits the natural parallelism available within the alternating iterations (the fact that rows of $\mathbf{W}$ and columns of $\mathbf{H}$ can be computed independently), it performs more communication than necessary to set up the independent problems. We compare the performance of this algorithm with our proposed approach to demonstrate the importance of designing algorithms to minimize communication; that is, simply parallelizing the computation is not sufficient for satisfactory performance and parallel scalability.

The main contribution of this work is a new, high-performance parallel algorithm for non-negative matrix factorization. The algorithm is flexible, as it is designed for both sparse and dense input matrices and can leverage many different algorithms for determining the non-negative low rank factors $\mathbf{W}$ and $\mathbf{H}$. The algorithm is also efficient, maintaining data in memory, using MPI collectives for interprocessor communication, and using efficient libraries for local computation. Furthermore, we provide a theoretical communication cost analysis to show that our algorithm reduces communication relative to the naive approach, and in the case of dense input, that it provably minimizes communication. We show with performance experiments that the algorithm outperforms the naive approach by significant factors, and that it scales well for up to 100s of processors on both synthetic and real-world data.

2. Preliminaries

2.1 Notation

Table 1 summarizes the notation we use throughout this paper. We use *upper case* letters for matrices and *lower case* letters for vectors. We use both subscripts and superscripts for sub-blocks of matrices. For example, $\mathbf{A}_i$ is the ith row block of matrix $\mathbf{A}$, and $\mathbf{A}^i$ is the ith column block. Likewise, $\mathbf{a}_i$ is the ith row of $\mathbf{A}$, and $\mathbf{a}^i$ is the ith column. We use m and n to denote the numbers of rows and columns of $\mathbf{A}$, respectively, and we assume without loss of generality $m \geqslant n$ throughout.

$\mathbf{A}$	Input matrix
$\mathbf{W}$	Left low rank factor
$\mathbf{H}$	Right low rank factor
m	Number of rows of input matrix
n	Number of columns of input matrix
k	Low rank
$\mathbf{M}_i$	ith row block of matrix $\mathbf{M}$
$\mathbf{M}^i$	ith column block of matrix $\mathbf{M}$
$\mathbf{M}_{ij}$	(i, j)th subblock of $\mathbf{M}$
p	Number of parallel processes
p_r	Number of rows in processor grid
p_c	Number of columns in processor grid

Table 1: Notation

2.2 Communication model

To analyze our algorithms, we use the α-β-γ model of distributed-memory parallel computation. In this model, interprocessor communication occurs in the form of messages sent between two processors across a bidirectional link (we assume a fully connected network). We model the cost of a message of size n words as $\alpha + n\beta$, where α is the per-message latency cost and β is the per-word bandwidth cost. Each processor can compute floating point operations (flops) on data that resides in its local memory; γ is the per-flop computation cost. With this communication model, we can predict the performance of an algorithm in terms of the number of flops it performs as well as the number of words and messages it communicates. For simplicity, we will ignore the possibilities of overlapping computation with communication in our analysis. For more details on the α-β-γ model, see [2, 25].

2.3 MPI collectives

Point-to-point messages can be organized into collective communication operations that involve more than two processors. MPI provides an interface to the most commonly used collectives like broadcast, reduce, and gather, as the algorithms for these collectives can be optimized for particular network topologies and processor characteristics. The algorithms we consider use the all-gather, reduce-scatter, and all-reduce collectives, so we review them here, along with their costs. Our analysis assumes optimal collective algorithms are used (see [2, 25]), though our implementation relies on the underlying MPI implementation.

At the start of an all-gather collective, each of p processors owns data of size n/p. After the all-gather, each processor owns a copy of the entire data of size n. The cost of an all-gather is $\alpha \cdot \log p + \beta \cdot \frac{p-1}{p} n$. At the start of a reduce-scatter collective, each processor owns data of size n. After the reduce-scatter, each processor owns a subset of the sum over all data, which is of size n/p. (Note that the reduction can be computed with other associative operators besides addition.) The cost of an reduce-scatter is $\alpha \cdot \log p + (\beta + \gamma) \cdot \frac{p-1}{p} n$. At the start of an all-reduce collective, each processor owns data of size n. After the all-reduce, each processor owns a copy of the sum over all data, which is also of size n. The cost of an all-reduce is $2\alpha \cdot \log p + (2\beta + \gamma) \cdot \frac{p-1}{p} n$. Note that the costs of each of the collectives are zero when $p = 1$.

3. Related Work

In the data mining and machine learning literature there is an overlap between low rank approximations and matrix factorizations due to the nature of applications. Despite its name, non-negative matrix "factorization" is really a low rank approximation.

The recent distributed NMF algorithms in the literature are [6, 8, 18, 19, 29]. Liu et al. propose running Multiplicative Update (MU) for KL divergence, squared loss, and "exponential" loss functions [19]. Matrix multiplication, element-wise multiplication, and element-wise division are the building blocks of the MU algorithm. The authors discuss performing these matrix operations effectively in Hadoop for sparse matrices. Using similar approaches, Liao et al. implement an open source Hadoop based MU algorithm and study its scalability on large-scale biological data sets [18]. Also, Yin, Gao, and Zhang present a scalable NMF that can perform frequent updates, which aim to use the most recently updated data [29]. Gemmula et al. propose a *Generic algorithm* that works on different loss functions, often involving the distributed computation of the gradient [8]. According to the authors, the formulation presented in the paper can also be extended to handle non-negative constraints. Similarly Faloutsos et al. propose a distributed, scalable method for decomposing matrices, tensors, and coupled data sets through stochastic gradient descent on a variety of objective functions [6]. The authors also provide an implementation that can enforce non-negative constraints on the factor matrices.

We note that Spark [30] is a popular big-data processing infrastructure that is is generally more efficient for iterative algorithms such as NMF than Hadoop, as it maintains data in memory and avoids file system I/O. Although Spark has collaborative filtering libraries such as MLlib [21], which use matrix factorization and can impose non-negativity constraints, none of them implement pure NMF, and so we do not have a direct comparison against NMF running on Spark. The problem of collaborative filtering is different from NMF because non-nonzero entries are treated as missing values rather than zeroes, and therefore different computations are performed at each iteration.

Apart from distributed NMF algorithms using Hadoop, there are also implementations of the MU algorithm in a distributed memory setting using X10 [9] and on a GPU [20].

4. Foundations

In this section, we will introduce the Alternating-Updating NMF (AU-NMF) framework, multiple methods for solving NMF and details on ANLS-BPP (Alternating Non-negative Least Squares - Block Principal Pivoting). We also present a straightforward approach to parallelization of the framework.

4.1 Alternating-Updating NMF Algorithms

NMF algorithms take a non-negative input matrix $\mathbf{A} \in \mathbb{R}_+^{m\times n}$ and a low rank k and determine two non-negative low rank factors $\mathbf{W} \in \mathbb{R}_+^{m\times k}$ and $\mathbf{H} \in \mathbb{R}_+^{k\times n}$ such that $\mathbf{A} \approx \mathbf{WH}$. We define Alternating-Updating NMF algorithms as those that alternate between updating $\mathbf{W}$ for a given $\mathbf{H}$ and updating $\mathbf{H}$ for a given $\mathbf{W}$. In the context of our parallel framework, we restrict attention to the class of NMF algorithms that use the Gram matrix associated with a factor matrix and the product of the input data matrix $\mathbf{A}$ with the corresponding factor matrix, as we show in Algorithm 1.

The specifics of lines 3 and 4 depend on the NMF algorithm. In the block coordinate descent framework where two blocks are the unknown factors $\mathbf{W}$ and $\mathbf{H}$, we solve the following subproblems, which have a unique solution for a full rank $\mathbf{H}$ and $\mathbf{W}$:

Algorithm 1 $[\mathbf{W}, \mathbf{H}]$ = AU-NMF(A, k)

```
Require: $\mathbf{A}$ is an $m \times n$ matrix, $k$ is rank of approximation
1: Initialize $\mathbf{H}$ with a non-negative matrix in $\mathbb{R}_+^{n\times k}$.
2: while stopping criteria not satisfied do
3:     Update $\mathbf{W}$ using $\mathbf{HH}^T$ and $\mathbf{AH}^T$
4:     Update $\mathbf{H}$ using $\mathbf{W}^T\mathbf{W}$ and $\mathbf{W}^T\mathbf{A}$
5: end while
```

$$\begin{aligned} \mathbf{W} &\leftarrow \underset{\tilde{\mathbf{W}}\geqslant 0}{\operatorname{argmin}} \left\|\mathbf{A} - \tilde{\mathbf{W}}\mathbf{H}\right\|_F, \\ \mathbf{H} &\leftarrow \underset{\tilde{\mathbf{H}}\geqslant 0}{\operatorname{argmin}} \left\|\mathbf{A} - \mathbf{W}\tilde{\mathbf{H}}\right\|_F. \end{aligned} \tag{2}$$

Since each subproblem involves nonnegative least squares, this two-block BCD method is also called the Alternating Non-negative Least Squares (ANLS) method [15]. Block Principal Pivoting (BPP), discussed more in detail at Section 4.2, is an algorithm that solves these NLS subproblems. In the context of the AU-NMF algorithm, this ANLS method *maximally* reduces the overall NMF objective function value by finding the optimal solution for given $\mathbf{H}$ and $\mathbf{W}$ in lines 3 and 4 respectively.

There are other popular NMF algorithms that update the factor matrices alternatively without maximally reducing the objective function value each time, in the same sense as in ANLS. These updates do not necessarily solve each of the subproblems (2) to optimality but simply improve the overall objective function (1). Such methods include Multiplicative Update (MU) [24] and Hierarchical Alternating Least Squares (HALS) [3], which was also independently proposed as Rank-one Residual Iteration (RRI) [10]. To show how these methods can fit into the AU-NMF framework, we discuss them in more detail.

In the case of HALS/RRI, individual columns of $\mathbf{W}$ and rows of $\mathbf{H}$ are updated with all other entries in the factor matrices fixed. This approach is a block coordinate descent method with $2k$ blocks, set to minimize the function

$$f(\mathbf{w}^1, \cdots, \mathbf{w}^k, \mathbf{h}_1, \cdots, \mathbf{h}_k) = \left\|\mathbf{A} - \sum_{i=1}^{k} \mathbf{w}^i\mathbf{h}_i\right\|_F, \tag{3}$$

where $\mathbf{w}^i$ is the ith column of $\mathbf{W}$ and $\mathbf{h}_i$ is the ith row of $\mathbf{H}$. The update rules can be written in closed form:

$$\begin{aligned} \mathbf{w}^i &\leftarrow \left[\mathbf{w}^i + \frac{(\mathbf{AH}^T)^i - \mathbf{W}(\mathbf{HH}^T)^i}{(\mathbf{HH}^T)_{ii}}\right]_+, \\ \mathbf{h}_i &\leftarrow \left[\mathbf{h}_i + \frac{(\mathbf{W}^T\mathbf{A})_i - (\mathbf{W}^T\mathbf{W})_i\mathbf{H}}{(\mathbf{W}^T\mathbf{W})_{ii}}\right]_+. \end{aligned} \tag{4}$$

Note that the columns of $\mathbf{W}$ and rows of $\mathbf{H}$ are updated in order, so that the most up-to-date values are always used, and these $2k$ updates can be done in an arbitrary order. However, if all the $\mathbf{W}$ updates are done before $\mathbf{H}$ (or vice-versa), the method falls into the AU-NMF framework. After computing the matrices $\mathbf{HH}^T$, $\mathbf{AH}^T$, $\mathbf{W}^T\mathbf{W}$, and $\mathbf{W}^T\mathbf{A}$, the extra computation is $2(m+n)k^2$ flops for updating both $\mathbf{W}$ and $\mathbf{H}$.

In the case of MU, individual entries of $\mathbf{W}$ and $\mathbf{H}$ are updated with all other entries fixed. In this case, the update rules are

$$\begin{aligned} w_{ij} &\leftarrow w_{ij}\frac{(\mathbf{AH}^T)_{ij}}{(\mathbf{WHH}^T)_{ij}}, \\ h_{ij} &\leftarrow h_{ij}\frac{(\mathbf{W}^T\mathbf{A})_{ij}}{(\mathbf{W}^T\mathbf{WH})_{ij}}. \end{aligned} \tag{5}$$

Instead of performing these $(m+n)k$ in an arbitrary order, if all of $\mathbf{W}$ is updated before $\mathbf{H}$ (or vice-versa), this method also follows the AU-NMF framework. The extra cost of computing $\mathbf{W}(\mathbf{H}\mathbf{H}^T)$ and $(\mathbf{W}^T\mathbf{W})\mathbf{H}$ is $2(m+n)k^2$ flops to perform updates for all entries of $\mathbf{W}$ and $\mathbf{H}$.

The convergence properties of these different algorithms are discussed in detail by Kim, He and Park [15]. We emphasize here that both HALS/RRI and MU require computing Gram matrices and matrix products of the input matrix and each factor matrix. Therefore, if the update ordering follows the convention of updating all of $\mathbf{W}$ followed by all of $\mathbf{H}$, both methods fit into the AU-NMF framework. Our proposed parallel algorithm (presented in Section 5) can be extended to these methods (or any other AU-NMF method) with only a change in local computation.

4.2 Block Principal Pivoting

In this paper, we focus on and use the BPP method [14] to solve the NLS problem, as it is the fastest algorithm (in terms of number of iterations). As argued in Section 4.1, we note that many NMF algorithms, including MU and HALS, can be used within our parallel frameworks (Algorithms 2 and 3).

BPP is the state-of-the-art method for solving the NLS subproblems in Eq. (2). The main subroutine of BPP is the single right-hand side NLS problem

$$\min_{\mathbf{x}\geqslant 0} \|\mathbf{C}\mathbf{x}-\mathbf{b}\|_2. \tag{6}$$

The Karush-Kuhn-Tucker (KKT) optimality conditions for Eq. (6) are as follows

$$\mathbf{y} = \mathbf{C}^T\mathbf{C}\mathbf{x} - \mathbf{C}^T\mathbf{b} \tag{7a}$$
$$\mathbf{y} \geqslant 0 \tag{7b}$$
$$\mathbf{x} \geqslant 0 \tag{7c}$$
$$x_i y_i = 0 \quad \forall i. \tag{7d}$$

The KKT conditions (7) states that at optimality, the support sets (i.e., the non-zero elements) of $\mathbf{x}$ and $\mathbf{y}$ are complementary to each other. Therefore, Eq. (7) is an instance of the *Linear Complementarity Problem* (LCP) which arises frequently in quadratic programming. When $k \ll \min(m,n)$, active-set and active-set-like methods are very suitable because most computations involve matrices of sizes $m\times k, n\times k$, and $k\times k$ which are small and easy to handle.

If we knew which indices correspond to nonzero values in the optimal solution, then computing the solution is an unconstrained least squares problem on these indices. In the optimal solution, call the set of indices i such that $x_i = 0$ the active set, and let the remaining indices be the passive set. The BPP algorithm works to find this final active set and passive set. It greedily swaps indices between the intermediate active and passive sets until finding a partition that satisfies the KKT condition. In the partition of the optimal solution, the values of the indices that belong to the active set will take zero. The values of the indices that belong to the passive set are determined by solving the unconstrained least squares problem restricted to the passive set. Kim, He and Park [14], discuss the BPP algorithm in further detail. We use the notation

$$\mathbf{X} \leftarrow \text{SolveBPP}(\mathbf{C}^T\mathbf{C}, \mathbf{C}^T\mathbf{B})$$

to define the (local) function for using BPP to solve Eq. (6) for every column of $\mathbf{X}$. We define $C_{\text{BPP}}(k,c)$ as the cost of SolveBPP, given the $k\times k$ matrix $\mathbf{C}^T\mathbf{C}$ and $k\times c$ matrix $\mathbf{C}^T\mathbf{B}$. SolveBPP mainly involves solving least squares problems over the intermediate passive sets. Our implementation uses the normal equations to solve the unconstrained least squares problems because the normal equations matrices have been pre-computed in order to check the KKT condition. However, more numerically stable methods such as QR decomposition can also be used.

Algorithm 2 $[\mathbf{W},\mathbf{H}]$ = Naive-Parallel-NMF$(\mathbf{A},k)$

```
Require: $\mathbf{A}$ is an $m\times n$ matrix distributed both row-wise and
    column-wise across $p$ processors, $k$ is rank of approximation
Require: Local matrices: $\mathbf{A}_i$ is $m/p\times n$, $\mathbf{A}^i$ is $m\times n/p$, $\mathbf{W}_i$ is $m/p\times k$,
    $\mathbf{H}^i$ is $k\times n/p$
1: $p_i$ initializes $\mathbf{H}^i$
2: while stopping criteria not satisfied do
       /* Compute W given H */
3:     collect $\mathbf{H}$ on each processor using all-gather
4:     $p_i$ computes $\mathbf{W}_i \leftarrow$ SolveBPP$(\mathbf{H}\mathbf{H}^T, \mathbf{A}_i\mathbf{H}^T)$
       /* Compute H given W */
5:     collect $\mathbf{W}$ on each processor using all-gather
6:     $p_i$ computes $(\mathbf{H}^i)^T \leftarrow$ SolveBPP$(\mathbf{W}^T\mathbf{W}, (\mathbf{W}^T\mathbf{A}^i)^T)$
7: end while
Ensure: $\mathbf{W},\mathbf{H} \approx \operatorname{argmin}_{\tilde{\mathbf{W}}\geqslant 0,\tilde{\mathbf{H}}\geqslant 0} \|\mathbf{A}-\tilde{\mathbf{W}}\tilde{\mathbf{H}}\|$
Ensure: $\mathbf{W}$ is an $m\times k$ matrix distributed row-wise across pro-
    cessors, $\mathbf{H}$ is a $k\times n$ matrix distributed column-wise across
    processors
```

4.3 Naive Parallel NMF Algorithm

In this section we present a naive parallelization of NMF algorithms [5]. Each NLS problem with multiple right-hand sides can be parallelized on the observation that the problems for multiple right-hand sides are independent from each other. That is, we can solve several instances of Eq. (6) independently for different $\mathbf{b}$ where $\mathbf{C}$ is fixed, which implies that we can optimize row blocks of $\mathbf{W}$ and column blocks of $\mathbf{H}$ in parallel.

Algorithm 2 presents a straightforward approach to setting up the independent subproblems. Let us divide $\mathbf{W}$ into row blocks $\mathbf{W}_1,\ldots,\mathbf{W}_p$ and $\mathbf{H}$ into column blocks $\mathbf{H}^1,\ldots,\mathbf{H}^p$. We then double-partition the data matrix $\mathbf{A}$ accordingly into row blocks $\mathbf{A}_1,\ldots,\mathbf{A}_p$ and column blocks $\mathbf{A}^1,\ldots,\mathbf{A}^p$ so that processor i owns both $\mathbf{A}_i$ and $\mathbf{A}^i$ (see Figure 1). With these partitions of the data and the variables, one can implement any ANLS algorithm in parallel, with only one communication step for each solve.

The computation cost of Algorithm 2 depends on the local NLS algorithm. For comparison with our proposed algorithm, we assume each processor uses BPP to solve the local NLS problems. Thus, the computation at line 4 consists of computing $\mathbf{A}^i\mathbf{H}^T$, $\mathbf{H}\mathbf{H}^T$, and solving NLS given the normal equations formulation of rank k for m/p columns. Likewise, the computation at line 6 consists of computing $\mathbf{W}^T\mathbf{A}_i$, $\mathbf{W}^T\mathbf{W}$, and solving NLS for n/p columns. In the dense case, this amounts to $4mnk/p+(m+n)k^2+C_{\text{BPP}}((m+n)/p,k)$ flops. In the sparse case, processor i performs $2(\text{nnz}(\mathbf{A}_i)+\text{nnz}(\mathbf{A}^i))k$ flops to compute $\mathbf{A}^i\mathbf{H}^T$ and $\mathbf{W}^T\mathbf{A}_i$ instead of $4mnk/p$.

The communication cost of the all-gathers at lines 3 and 5, based on the expression given in Section 2.3 is $\alpha\cdot 2\log p+\beta\cdot(m+n)k$. The local memory requirement includes storing each processor's part of matrices $\mathbf{A}$, $\mathbf{W}$, and $\mathbf{H}$. In the case of dense $\mathbf{A}$, this is $2mn/p+(m+n)k/p$ words, as $\mathbf{A}$ is stored twice; in the sparse case, processor i requires $\text{nnz}(\mathbf{A}_i)+\text{nnz}(\mathbf{A}^i)$ words for the input matrix and $(m+n)k/p$ words for the output factor matrices. Local memory is also required for storing temporary matrices $\mathbf{W}$ and $\mathbf{H}$ of size $(m+n)k$ words.

We summarize the algorithmic costs of Algorithm 2 in Table 2. This naive algorithm [5] has three main drawbacks: (1) it requires storing two copies of the data matrix (one in row distribution and one in column distribution) and both full factor matrices locally, (2) it does not parallelize the computation of $\mathbf{H}\mathbf{H}^T$ and $\mathbf{W}^T\mathbf{W}$ (each processor computes it redundantly), and (3) as we will see in Section 5, it communicates more data than necessary.

Algorithm	Flops	Words	Messages	Memory
Naive-Parallel-NMF	$4\frac{mnk}{p} + (m+n)k^2 + C_{\text{BPP}}\left(\frac{m+n}{p}, k\right)$	$O((m+n)k)$	$O(\log p)$	$O\left(\frac{mn}{p} + (m+n)k\right)$
HPC-NMF ($m/p \geqslant n$)	$4\frac{mnk}{p} + \frac{(m+n)k^2}{p} + C_{\text{BPP}}\left(\frac{m+n}{p}, k\right)$	$O(nk)$	$O(\log p)$	$O\left(\frac{mn}{p} + \frac{mk}{p} + nk\right)$
HPC-NMF ($m/p < n$)	$4\frac{mnk}{p} + \frac{(m+n)k^2}{p} + C_{\text{BPP}}\left(\frac{m+n}{p}, k\right)$	$O\left(\sqrt{\frac{mnk^2}{p}}\right)$	$O(\log p)$	$O\left(\frac{mn}{p} + \sqrt{\frac{mnk^2}{p}}\right)$
Lower Bound	–	$\Omega\left(\min\left\{\sqrt{\frac{mnk^2}{p}}, nk\right\}\right)$	$\Omega(\log p)$	$\frac{mn}{p} + \frac{(m+n)k}{p}$

Table 2: Leading order algorithmic costs for Naive-Parallel-NMF and HPC-NMF (per iteration). Note that the computation and memory costs assume the data matrix **A** is dense, but the communication costs (words and messages) apply to both dense and sparse cases.

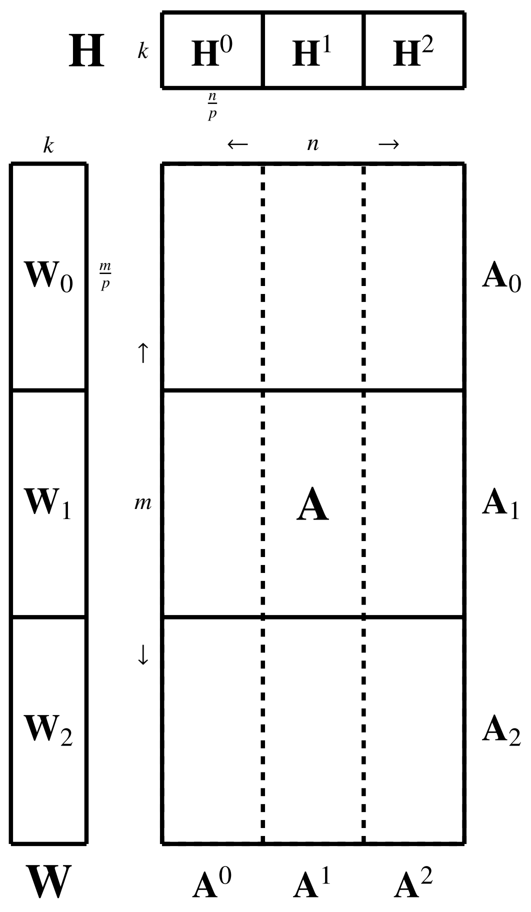


Figure 1: Distribution of matrices for Naive-Parallel-NMF (Algorithm 2), for $p = 3$. Note that $\mathbf{A}_i$ is $m/p \times n$, $\mathbf{A}^i$ is $m \times n/p$, $\mathbf{W}_i$ is $m/p_r \times k$, and $\mathbf{H}^i$ is $k \times n/p$.

5. High Performance Parallel NMF

We present our proposed algorithm, HPC-NMF, as Algorithm 3. The main ideas of the algorithm are to (1) exploit the independence of NLS problems for rows of **W** and columns of **H** and (2) use communication-optimal matrix multiplication algorithms to set up the NLS problems. The naive approach (Algorithm 2) shares the first property, by parallelizing over rows of **W** and columns of **H**, but it uses parallel matrix multiplication algorithms that communicate more data than necessary. The central intuition for communication-efficient parallel algorithms for computing $\mathbf{HH}^T$, $\mathbf{AH}^T$, $\mathbf{W}^T\mathbf{W}$, and $\mathbf{W}^T\mathbf{A}$ comes from a classification proposed by Demmel et al. [4]. They consider three cases, depending on the relative sizes of the dimensions of the matrices and the number of processors; the four multiplies for NMF fall into either the "one large dimension" or "two large dimensions" cases. HPC-NMF uses a careful data distribution in order to use a communication-optimal algorithm for each of the matrix multiplications, while at the same time exploiting the parallelism in the NLS problems.

The algorithm uses a 2D distribution of the data matrix **A** across a $p_r \times p_c$ grid of processors (with $p = p_r p_c$), as shown in Figure 2. Algorithm 3 performs an alternating method in parallel with a per-iteration bandwidth cost of $O\left(\min\left\{\sqrt{mnk^2/p}, nk\right\}\right)$ words, latency cost of $O(\log p)$ messages, and load-balanced computation (up to the sparsity pattern of **A** and convergence rates of local BPP computations).

To minimize the communication cost and local memory requirements, in the typical case p_r and p_c are chosen so that $m/p_r \approx n/p_c \approx \sqrt{mn/p}$, in which case the bandwidth cost is $O\left(\sqrt{mnk^2/p}\right)$. If the matrix is very tall and skinny, i.e., $m/p > n$, then we choose $p_r = p$ and $p_c = 1$. In this case, the distribution of the data matrix is 1D, and the bandwidth cost is $O(nk)$ words.

The matrix distributions for Algorithm 3 are given in Figure 2; we use a 2D distribution of **A** and 1D distributions of **W** and **H**. Recall from Table 1 that $\mathbf{M}_i$ and $\mathbf{M}^i$ denote row and column blocks of **M**, respectively. Thus, the notation $(\mathbf{W}_i)_j$ denotes the jth row block within the ith row block of **W**. Lines 3–8 compute **W** for a fixed **H**, and lines 9–14 compute **H** for a fixed **W**; note that the computations and communication patterns for the two alternating iterations are analogous.

In the rest of this section, we derive the per-iteration computation and communication costs, as well as the local memory requirements. We also argue the communication-optimality of the algorithm in the dense case. Table 2 summarizes the results of this section and compares them to Naive-Parallel-NMF.

Computation Cost Local matrix computations occur at lines 3, 6, 9, and 12. In the case that **A** is dense, each processor performs

$$\frac{n}{p}k^2 + 2\frac{m}{p_r}\frac{n}{p_c}k + \frac{m}{p}k^2 + 2\frac{m}{p_r}\frac{n}{p_c}k = 4\frac{mnk}{p} + \frac{(m+n)k^2}{p}$$

flops. In the case that **A** is sparse, processor (i, j) performs $(m+n)k^2/p$ flops in computing $\mathbf{U}_{ij}$ and $\mathbf{X}_{ij}$, and $4\text{nnz}(\mathbf{A}_{ij})k$ flops in computing $\mathbf{V}_{ij}$ and $\mathbf{Y}_{ij}$. Local non-negative least squares problems occur at lines 8 and 14. In each case, the symmetric positive semi-definite matrix is $k \times k$ and the number of columns/rows of length k to be computed are m/p and n/p, respectively. These costs together require $C_{\text{BPP}}(k, (m+n)/p)$ flops. There are computation costs associated with the all-reduce and reduce-scatter collectives, both those contribute only to lower order terms.

Communication Cost Communication occurs during six collective operations (lines 4, 5, 7, 10, 11, and 13). We use the cost expressions presented in Section 2.3 for these collectives. The communication cost of the all-reduces (lines 4 and 10) is $\alpha \cdot 4 \log p +$

```
Algorithm 3 $[\mathbf{W},\mathbf{H}]$ = HPC-NMF($\mathbf{A}, k$)
Require: $\mathbf{A}$ is an $m \times n$ matrix distributed across a $p_r \times p_c$ grid of
    processors, $k$ is rank of approximation
Require: Local matrices: $\mathbf{A}_{ij}$ is $m/p_r \times n/p_c$, $\mathbf{W}_i$ is $m/p_r \times k$, $(\mathbf{W}_i)_j$
    is $m/p \times k$, $\mathbf{H}_j$ is $k \times n/p_c$, and $(\mathbf{H}_j)_i$ is $k \times n/p$
 1: $p_{ij}$ initializes $(\mathbf{H}_j)_i$
 2: while stopping criteria not satisfied do
        /* Compute W given H */
 3:     $p_{ij}$ computes $\mathbf{U}_{ij} = (\mathbf{H}_j)_i (\mathbf{H}_j)_i^T$
 4:     compute $\mathbf{HH}^T = \sum_{i,j} \mathbf{U}_{ij}$ using all-reduce across all procs
                                          ▷ $\mathbf{HH}^T$ is $k \times k$ and symmetric
 5:     $p_{ij}$ collects $\mathbf{H}_j$ using all-gather across proc columns
 6:     $p_{ij}$ computes $\mathbf{V}_{ij} = \mathbf{A}_{ij} \mathbf{H}_j^T$
                                          ▷ $\mathbf{V}_{ij}$ is $m/p_r \times k$
 7:     compute $(\mathbf{AH}^T)_i = \sum_j \mathbf{V}_{ij}$ using reduce-scatter across proc
    row to achieve row-wise distribution of $(\mathbf{AH}^T)_i$
                                          ▷ $p_{ij}$ owns $m/p \times k$ submatrix $((\mathbf{AH}^T)_i)_j$
 8:     $p_{ij}$ computes $(\mathbf{W}_i)_j \leftarrow$ SolveBPP($\mathbf{HH}^T, ((\mathbf{AH}^T)_i)_j$)
        /* Compute H given W */
 9:     $p_{ij}$ computes $\mathbf{X}_{ij} = (\mathbf{W}_i)_j{}^T (\mathbf{W}_i)_j$
10:     compute $\mathbf{W}^T\mathbf{W} = \sum_{i,j} \mathbf{X}_{ij}$ using all-reduce across all procs
                                          ▷ $\mathbf{W}^T\mathbf{W}$ is $k \times k$ and symmetric
11:     $p_{ij}$ collects $\mathbf{W}_i$ using all-gather across proc rows
12:     $p_{ij}$ computes $\mathbf{Y}_{ij} = \mathbf{W}_i{}^T \mathbf{A}_{ij}$
                                          ▷ $\mathbf{Y}_{ij}$ is $k \times n/p_c$
13:     compute $(\mathbf{W}^T\mathbf{A})^j = \sum_i \mathbf{Y}_{ij}$ using reduce-scatter across proc
    columns to achieve column-wise distribution of $(\mathbf{W}^T\mathbf{A})^j$
                                          ▷ $p_{ij}$ owns $k \times n/p$ submatrix $((\mathbf{W}^T\mathbf{A})^j)^i$
14:     $p_{ij}$ computes $((\mathbf{H}^j)^i)^T \leftarrow$ SolveBPP($\mathbf{W}^T\mathbf{W}, (((\mathbf{W}^T\mathbf{A})^j)^i)^T$)
15: end while
Ensure: $\mathbf{W},\mathbf{H} \approx \operatorname{argmin}_{\tilde{\mathbf{W}}\geqslant 0, \tilde{\mathbf{H}}\geqslant 0} \|\mathbf{A} - \tilde{\mathbf{W}}\tilde{\mathbf{H}}\|$
Ensure: $\mathbf{W}$ is an $m \times k$ matrix distributed row-wise across pro-
    cessors, $\mathbf{H}$ is a $k \times n$ matrix distributed column-wise across
    processors
```

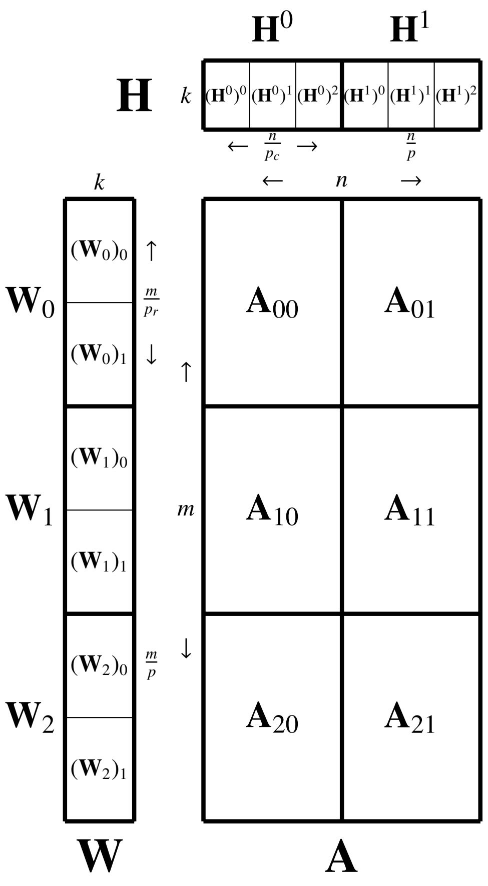


Figure 2: Distribution of matrices for HPC-NMF (Algorithm 3), for $p_r = 3$ and $p_c = 2$. Note that $\mathbf{A}_{ij}$ is $m/p_r \times m/p_c$, $\mathbf{W}_i$ is $m/p_r \times k$, $(\mathbf{W}_i)_j$ is $m/p \times k$, $\mathbf{H}_j$ is $k \times n/p_c$, and $(\mathbf{H}_j)_i$ is $k \times n/p$.

$\beta \cdot 2k^2$; the cost of the two all-gathers (lines 5 and 11) is $\alpha \cdot \log p + \beta \cdot ((p_r-1)nk/p + (p_c-1)mk/p)$; and the cost of the two reduce-scatters (lines 7 and 13) is $\alpha \cdot \log p + \beta \cdot ((p_c-1)mk/p + (p_r-1)nk/p)$.

In the case that $m/p < n$, we choose $p_r = \sqrt{np/m} > 1$ and $p_c = \sqrt{mp/n} > 1$, and these communication costs simplify to $\alpha \cdot O(\log p) + \beta \cdot O(mk/p_r + nk/p_c + k^2) = \alpha \cdot O(\log p) + \beta \cdot O(\sqrt{mnk^2/p} + k^2)$. In the case that $m/p \geqslant n$, we choose $p_c = 1$, and the costs simplify to $\alpha \cdot O(\log p) + \beta \cdot O(nk)$.

Memory Requirements The local memory requirement includes storing each processor's part of matrices $\mathbf{A}$, $\mathbf{W}$, and $\mathbf{H}$. In the case of dense $\mathbf{A}$, this is $mn/p + (m+n)k/p$ words; in the sparse case, processor (i, j) requires $\mathrm{nnz}(\mathbf{A}_{ij})$ words for the input matrix and $(m+n)k/p$ words for the output factor matrices. Local memory is also required for storing temporary matrices $\mathbf{W}_j$, $\mathbf{H}_i$, $\mathbf{V}_{ij}$, and $\mathbf{Y}_{ij}$, of size $2mk/p_r + 2nk/p_c$) words.

In the dense case, assuming $k < n/p_c$ and $k < m/p_r$, the local memory requirement is no more than a constant times the size of the original data. For the optimal choices of p_r and p_c, this assumption simplifies to $k < \max\left\{\sqrt{mn/p}, m/p\right\}$.

We note that if the temporary memory requirements become prohibitive, the computation of $((\mathbf{AH}^T)_i)_j$ and $((\mathbf{W}^T\mathbf{A})_j)_i$ via all-gathers and reduce-scatters can be blocked, decreasing the local memory requirements at the expense of greater latency costs. While this case is plausible for sparse $\mathbf{A}$, we did not encounter local memory issues in our experiments.

Communication Optimality In the case that $\mathbf{A}$ is dense, Algorithm 3 provably minimizes communication costs. Theorem 5.1 establishes the bandwidth cost lower bound for any algorithm that computes $\mathbf{W}^T\mathbf{A}$ or $\mathbf{AH}^T$ each iteration. A latency lower bound of $\Omega(\log p)$ exists in our communication model for any algorithm that aggregates global information [2], and for NMF, this global aggregation is necessary in each iteration. Based on the costs derived above, HPC-NMF is communication optimal under the assumption $k < \sqrt{mn/p}$, matching these lower bounds to within constant factors.

Theorem 5.1 ([4]). *Let $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{W} \in \mathbb{R}^{m\times k}$, and $\mathbf{H} \in \mathbb{R}^{k\times n}$ be dense matrices, with $k < n \leqslant m$. If $k < \sqrt{mn/p}$, then any distributed-memory parallel algorithm on p processors that load balances the matrix distributions and computes $\mathbf{W}^T\mathbf{A}$ and/or $\mathbf{AH}^T$ must communicate at least $\Omega(\min\{\sqrt{mnk^2/p}, nk\})$ words along its critical path.*

Proof The proof follows directly from [4, Section II.B]. Each matrix multiplication $\mathbf{W}^T\mathbf{A}$ and $\mathbf{AH}^T$ has dimensions $k < n \leqslant m$, so the assumption $k < \sqrt{mn/p}$ ensures that neither multiplication has "3 large dimensions." Thus, the communication lower bound is either $\Omega(\sqrt{mnk^2/p})$ in the case of $p > m/n$ (or "2 large dimensions"), or $\Omega(nk)$, in the case of $p < m/n$ (or "1 large dimension"). If $p < m/n$, then $nk < \sqrt{mnk^2/p}$, so the lower bound can be written as $\Omega(\min\{\sqrt{mnk^2/p}, nk\})$.

We note that the communication costs of Algorithm 3 are the same for dense and sparse data matrices (the data matrix itself is never communicated). In the case that $\mathbf{A}$ is sparse, this communication lower bound does not necessarily apply, as the required data movement depends on the sparsity pattern of $\mathbf{A}$. Thus, we cannot make claims of optimality in the sparse case (for general $\mathbf{A}$). The communication lower bounds for $\mathbf{W}^T\mathbf{A}$ and/or $\mathbf{A}\mathbf{H}^T$ (where $\mathbf{A}$ is sparse) can be expressed in terms of hypergraphs that encode the sparsity structure of $\mathbf{A}$ [1]. Indeed, hypergraph partitioners have been used to reduce communication and achieve load balance for a similar problem: computing a low-rank representation of a sparse tensor (without non-negativity constraints on the factors) [12].

6. Experiments

In this section, we describe our implementation of HPC-NMF and evaluate its performance. We identify a few synthetic and real world data sets to experiment with HPC-NMF as well as Naive-Parallel-NMF, comparing performance and exploring scaling behavior.

In the data mining and machine learning community, there has been a large interest in using Hadoop for large scale implementation. Hadoop requires disk I/O and is designed for processing gigantic text files. Many of the real world data sets that are available for research are large scale sparse internet text data, recommender systems, social networks, etc. Towards this end, there has been interest towards Hadoop implementations of matrix factorization algorithms [8, 18, 19]. However, the use of NMF extends beyond sparse internet data and is also applicable for dense real world data such as video, images, etc. Hence in order to keep our implementation applicable to wider audience, we choose to use MPI for our distributed implementation. Apart from the application point of view, we use an MPI/C++ implementation for other technical advantages that are necessary for scientific applications such as (1) the ability to leverage recent hardware improvements, (2) effective communication between nodes, and (3) the availability of numerically stable and efficient BLAS and LAPACK routines.

6.1 Experimental Setup

6.1.1 Data Sets

We used sparse and dense matrices that are either synthetically generated or from real world applications. We will explain the data sets in this section.

- Dense Synthetic Matrix (*DSyn*): We generate a uniform random matrix of size 172,800 × 115,200 and add random Gaussian noise. The dimensions of this matrix is chosen such that it is uniformly distributable across processes. Every process will have its own prime seed that is different from other processes to generate the input random matrix $\mathbf{A}$.
- Sparse Synthetic Matrix (*SSyn*): We generate a random sparse Erdős-Rényi matrix of the dimension 172,800 × 115,200 with density of 0.001. That is, every entry is nonzero with probability 0.001.
- Dense Real World Matrix (*Video*): NMF can be performed in the video data for background subtraction to detect moving objects. The low rank matrix $\hat{\mathbf{A}} = \mathbf{W}\mathbf{H}^T$ represents background and the error matrix $\mathbf{A} - \hat{\mathbf{A}}$ represents moving objects. Detecting moving objects has many real-world applications such as traffic estimation [7], security monitoring, etc. In the case of detecting moving objects, only the last minute or two of video is taken from the live video camera. The algorithm to incrementally adjust the NMF based on the new streaming video is presented in [15]. To simulate this scenario, we collected a video in a busy intersection of the Georgia Tech campus at 20 frames per second for two minutes. We then reshaped the matrix such that every RGB frame is a column of our matrix, so that the matrix is dense with dimensions 1,013,400 × 2400.
- Sparse Real World Matrix *Webbase* : This data set is a very large, directed sparse graph with nearly 1 million nodes (1,000,005) and 3.1 million edges (3,105,536), which was first reported by Williams et al. [27]. The NMF output of this directed graph helps us understand clusters in graphs.

The size of both real world data sets were adjusted to the nearest dimension for uniformly distributing the matrix.

6.1.2 Machine

We conducted our experiments on "Edison" at the National Energy Research Scientific Computing Center. Edison is a Cray XC30 supercomputer with a total of 5,576 compute nodes, where each node has dual-socket 12-core Intel Ivy Bridge processors. Each of the 24 cores has a clock rate of 2.4 GHz (translating to a peak floating point rate of 460.8 Gflops/node) and private 64KB L1 and 256KB L2 caches; each of the two sockets has a (shared) 30MB L3 cache; each node has 64 GB of memory. Edison uses a Cray "Aries" interconnect that has a dragonfly topology. Because our experiments use a relatively small number of nodes, we consider the local connectivity: a "blade" comprises 4 nodes and a router, and sets of 16 blades' routers are fully connected via a circuit board backplane (within a "chassis"). Our experiments do not exceed 64 nodes, so we can assume a very efficient, fully connected network.

6.1.3 Software

Our objective of the implementation is using open source software as much as possible to promote reproducibility and reuse of our code. The entire C++ code was developed using the matrix library Armadillo [23]. In Armadillo, the elements of the dense matrix are stored in column major order and the sparse matrices in Compressed Sparse Column (CSC) format. For dense BLAS and LAPACK operations, we linked Armadillo with OpenBLAS [28]. We use Armadillo's own implementation of sparse matrix-dense matrix multiplication, the default GNU C++ Compiler and MPI library on Edison.

6.1.4 Initialization and Stopping Criteria

To ensure fair comparison among algorithms, the same random seed was used across different methods appropriately. That is, the initial random matrix $\mathbf{H}$ was generated with the same random seed when testing with different algorithms (note that $\mathbf{W}$ need not be initialized). This ensures that all the algorithms perform the same computations (up to roundoff errors), though the only computation with a running time that is sensitive to matrix values is the local NNLS solve via BPP.

In this paper, we used number of iterations as the stopping criteria for all the algorithms. For fair comparison, all the algorithms in the paper were executed for 10 iterations.

6.2 Algorithms

For each of our data sets, we benchmark and compare three algorithms: (1) Algorithm 2, (2) Algorithm 3 with $p_r = p$ and $p_c = 1$ (1D processor grid), and (3) Algorithm 3 with $p_r \approx p_c \approx \sqrt{p}$ (2D processor grid). We choose these three algorithms to confirm the following conclusions from the analysis of Section 5: the performance of a naive parallelization of Naive-Parallel-NMF (Algorithm 2) will be severely limited by communication overheads, and the correct choice of processor grid within Algorithm 3 is necessary to optimize performance. To demonstrate the latter conclusion, we choose the two extreme choices of processor grids and test some data sets where a 1D processor grid is optimal (e.g., the Video matrix) and some where a squarish 2D grid is optimal (e.g., the Webbase matrix).

While we would like to compare against other high-performance NMF algorithms in the literature, the only other distributed-memory implementations of which we're aware are implemented using Hadoop and are designed only for sparse matrices [18], [19], [8], [29] and [6]. We stress that Hadoop is not designed for high performance computing of iterative numerical algorithms, requiring disk I/O between steps, so a run time comparison between a Hadoop implementation and a C++/MPI implementation is not a fair comparison of parallel algorithms. A qualitative example of differences in run time is that a Hadoop implementation of the MU algorithm on a large sparse matrix of dimension $2^{17} \times 2^{16}$ with 2×10^8 nonzeros (with k=8) takes on the order of 50 minutes per iteration [19], while our implementation takes a second per iteration for the synthetic data set (which is an order of magnitude larger in terms of rows, columns, and nonzeros) running on only 24 nodes.

6.3 Time Breakdown

To differentiate the computation and communication costs among the algorithms, we present the time breakdown among the various tasks within the algorithms for both performance experiments. For Algorithm 3, there are three local computation tasks and three communication tasks to compute each of the factor matrices:

- **MM**, computing a matrix multiplication with the local data matrix and one of the factor matrices;
- **NLS**, solving the set of NLS problems using BPP;
- **Gram**, computing the local contribution to the Gram matrix;
- **All-Gather**, to compute the global matrix multiplication;
- **Reduce-Scatter**, to compute the global matrix multiplication;
- **All-Reduce**, to compute the global Gram matrix.

In our results, we do not distinguish the costs of these tasks for **W** and **H** separately; we report their sum, though we note that we do not always expect balance between the two contributions for each task. Algorithm 2 performs all of these tasks except Reduce-Scatter and All-Reduce; all of its communication is in All-Gather.

6.4 Algorithmic Comparison

Our first set of experiments is designed primarily to compare the three parallel implementations. For each data set, we fix the number of processors to be 600 and vary the rank k of the desired factorization. Because most of the computation (except for NLS) and bandwidth costs are linear in k (except for the All-Reduce), we expect linear performance curves for each algorithm individually.

The left side of Figure 3 shows the results of this experiment for all four data sets. The first conclusion we draw is that HPC-NMF with a 2D processor grid performs significantly better than the Naive-Parallel-NMF; the largest speedup is 4.4×, for the sparse synthetic data and $k = 10$ (a particularly communication bound problem). Also, as predicted, the 2D processor grid outperforms the 1D processor grid on the squarish matrices. While we expect the 1D processor grid to outperform the 2D grid for the tall-and-skinny Video matrix, their performances are comparable; this is because both algorithms are computation bound, as we see from Figure 3g, so the difference in communication is negligible.

The second conclusion we can draw is that the scaling with k tends to be close to linear, with an exception in the case of the Webbase matrix. We see from Figure 3e that this problem spends much of its time in NLS, which does not scale linearly with k. Note that for a fixed problem, the size of the local NLS problem remains the same across algorithms. Thus, we expect similar timing results and observe that to be true for most cases.

We can also compare HPC-NMF with a 1D processor grid with Naive-Parallel-NMF for squarish matrices (SSyn, DSyn, and Webbase). Our analysis does not predict a significant difference in communication costs of these two approaches (when $m \approx n$), and we see from the data that Naive-Parallel-NMF outperforms HPC-NMF for two of the three matrices (but the opposite is true for DSyn). The main differences appear in the All-Gather versus Reduce-Scatter communication costs and the local MM (all of which are involved in the $\mathbf{W}^T\mathbf{A}$ computation). In all three cases, our proposed 2D processor grid (with optimal choice of $m/p_r \approx n/p_c$) performs better than both alternatives.

6.5 Strong Scalability

The goal of our second set of experiments is to demonstrate the strong scalability of each of the algorithms. For each data set, we fix the rank k to be 50 and vary the number of processors (this is a strong-scaling experiment because the size of the data set is fixed). We run our experiments on $\{24, 96, 216, 384, 600\}$ processors/cores, which translates to $\{1, 4, 9, 16, 25\}$ nodes. The dense matrices are too large for 1 or 4 nodes, so we benchmark only on $\{216, 384, 600\}$ cores in those cases.

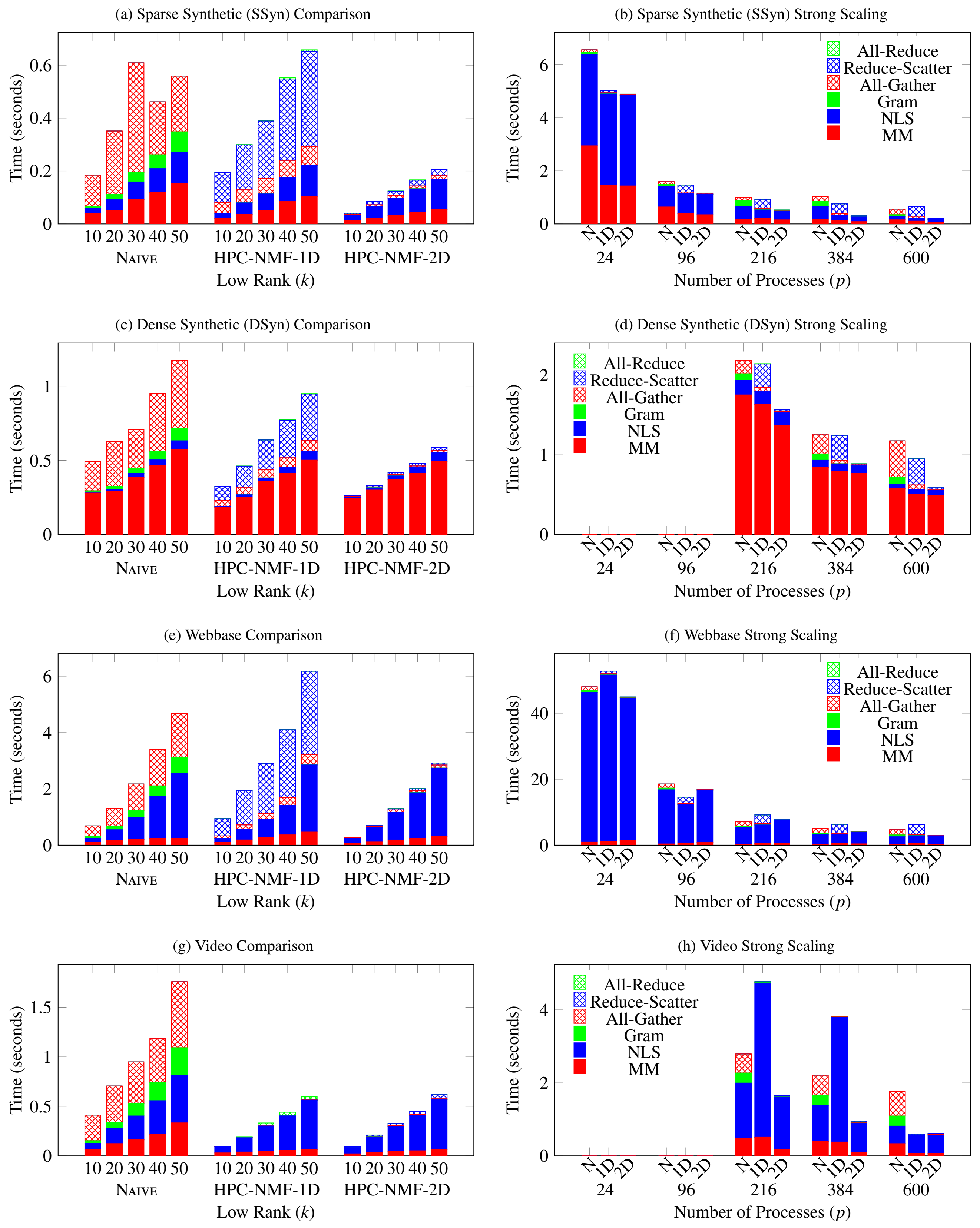
The right side of Figure 3 shows the scaling results for all four data sets, and Table 3 gives the overall per-iteration time for each algorithm, number of processors, and data set. We first consider the HPC-NMF algorithm with a 2D processor grid: comparing the performance results on 25 nodes (600 cores) to the 1 node (24 cores), we see nearly perfect parallel speedups. The parallel speedups are 23× for SSyn and 28× for the Webbase matrix. We believe the superlinear speedup of the Webbase matrix is a result of the running time being dominated by NLS; with more processors, the local NLS problem is smaller and more likely to fit in smaller levels of cache, providing better performance. For the dense matrices, the speedup of HPC-NMF on 25 nodes over 9 nodes is 2.7× for DSyn and 2.8× for Video, which are also nearly linear.

In the case of the Naive-Parallel-NMF algorithm, we do see parallel speedups, but they are not linear. For the sparse data, we see parallel speedups of 10× and 11× with a 25× increase in number of processors. For the dense data, we observe speedups of 1.6× and 1.8× with a 2.8× increase in the number of processors. The main reason for not achieving perfect scaling is the unnecessary communication overheads.

6.6 Weak Scalability

Our third set of experiments shows the weak scalability of each of the algorithms. We consider only the synthetic data sets so that we can flexibly scale the dimensions of the data matrix. Again, we run our experiments on $\{24, 96, 216, 384, 600\}$ processors/cores. We fix the input data size per processor in this scaling experiment: mn/p is the same across all experiments (dense and sparse). The data matrix dimensions range from 57600×38400 on 24 cores up to 288000×192000 on 600 cores; for HPC-NMF with a 2D processor grid, the local matrix is always 9600×9600. The dimensions are chosen so that this experiment matches the strong-scaling experiment on 216 processors. Figure 4 shows our results.

We emphasize that while this experiment fixes the amount of input matrix data per processor, it does not fix the amount of factor matrix data per processor (which decreases as we scale up). Likewise, it fixes the number of MM flops performed by each processor but not the number of NLS flops; the latter also decreases as we scale up. Thus, if communication were free, we would expect the overall time to decrease as we scale to more processors, at a rate that depends on the relative time spent in MM and NLS. This behavior generally holds true in Figure 4: in the dense case, since most of the time is in MM, the time generally holds steady as the



,

Figure 3: Comparison and strong-scaling experiments on sparse and dense data sets for three algorithms: Naive (N), HPC-NMF-1D (1D), HPC-NMF-2D (2D). Plots on the left vary the low rank k for fixed $p = 600$, and plots on the right vary the number of processes (cores) p for fixed $k = 50$. The reported time is the average over 10 iterations.

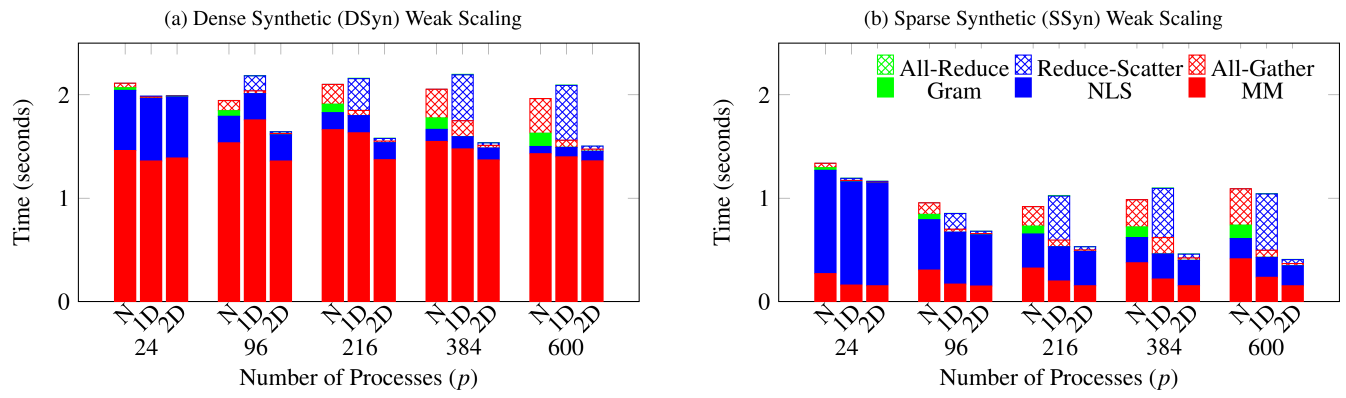


Figure 4: Weak-scaling experiments on dense (left) and sparse (right) synthetic data sets for three algorithms: Naive (N), HPC-NMF-1D (1D), HPC-NMF-2D (2D). The ratio mn/p is fixed across all data points (dense and sparse), with data matrix dimensions ranging from 57600×38400 on 24 cores up to 288000×192000 on 600 cores. The reported time is the average over 10 iterations.

	Naive-Parallel-NMF				HPC-NMF-1D				HPC-NMF-2D			
Cores	**DSyn**	**SSyn**	**Video**	**Webbase**	**DSyn**	**SSyn**	**Video**	**Webbase**	**DSyn**	**SSyn**	**Video**	**Webbase**
24		6.5632		48.0256		5.0821		52.8549		4.8427		84.6286
96		1.5929		18.5507		1.4836		14.5873		1.1147		16.6966
216	2.1819	0.6027	2.7899	7.1274	2.1548	0.9488	4.7928	9.2730	1.5283	0.4816	1.6106	7.4799
384	1.2594	0.6466	2.2106	5.1431	1.2559	0.7695	3.8295	6.4740	0.8620	0.2661	0.8963	4.0630
600	1.1745	0.5592	1.7583	4.6825	0.9685	0.6666	0.5994	6.2751	0.5519	0.1683	0.5699	2.7376

Table 3: Average per-iteration running times (in seconds) of parallel NMF algorithms for $k = 50$.

number of processors increases, while in the sparse case, more time is spent in NLS and times decrease from left to right.

We also point out that we used the same matrix dimensions in the dense and sparse cases; because the communication does not depend on the input matrix sparsity, we see that the communication costs are same (for each algorithm and number of processors). The main difference in running time comes from MM, which is much cheaper in the sparse case.

In the case of HPC-NMF-2D, the weak scaling is nearly perfect as the time spent in communication is negligible. This is explained by the theory (see Table 2): if mn/p is fixed, then the bandwidth cost $O(\sqrt{mnk^2/p})$ is also fixed, so we expect HPC-NMF-2D to scale well to much larger numbers of processors. In the case of Naive-Parallel-NMF and HPC-NMF-1D, we see that communication costs increase as we scale up. Again, Table 2 shows that the bandwidth costs of those algorithms increase as we scale up, so we don't expect those to scale as well in this case. We note that this is only one form of weak scaling; for example, if we were to fix the quantity m/p, then we would expect HPC-NMF-1D to scale well (though Naive-Parallel-NMF would not). The best overall speedup we observe from this experiment is in the sparse case on 600 processors: HPC-NMF-2D is 2.7× faster than Naive-Parallel-NMF.

7. Conclusion

In this paper, we propose a high-performance distributed-memory parallel algorithm that computes an NMF by iteratively solving alternating non-negative least squares (ANLS) subproblems. We carefully designed a parallel algorithm which avoids communication overheads and scales well to modest numbers of cores.

For the data sets on which we experimented, we showed that an efficient implementation of a naive parallel algorithm spends much of its time in interprocessor communication. In the case of HPC-NMF, the problems remain computation bound on up to 600 processors, typically spending most of the time in local matrix multiplication or NLS solves.

We focus in this work on BPP, because it has been shown to reduce overall running time in the sequential case by requiring fewer iterations [14]. Because much of the time per iteration of HPC-NMF is spent on local NLS, we believe further empirical exploration is necessary to understand the proposed HPC-NMF's advantages for other AU-NMF algorithms such as MU and HALS. We note that if we use the MU or HALS approach for determining low rank factors, the relative cost of interprocessor communication will grow, making the communication efficiency of our algorithm more important.

In future work, we would like to extend HPC-NMF algorithm to dense and sparse tensors, computing the CANDECOMP/PARAFAC decomposition in parallel with non-negativity constraints on the factor matrices. We would also like to explore more intelligent distributions of sparse matrices: while our 2D distribution is based on evenly dividing rows and columns, it does not necessarily load balance the nonzeros of the matrix, which can lead to load imbalance in MM. We are interested in using graph and hypergraph partitioning techniques to load balance the memory and computation while at the same time reducing communication costs as much as possible. Finally, we have not yet reached the limits of the scalability of HPC-NMF; we would like to expand our benchmarks to larger numbers of nodes on the same size data sets to study performance behavior when communication costs completely dominate the running time.

Acknowledgments

This research was supported in part by an appointment to the Sandia National Laboratories Truman Fellowship in National Security Science and Engineering, sponsored by Sandia Corporation (a wholly owned subsidiary of Lockheed Martin Corporation) as Operator of Sandia National Laboratories under its U.S. Department of Energy Contract No. DE-AC04-94AL85000.

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Partial funding for this work was also provided by AFOSR Grant FA9550-13-1-0100, National Science Foundation (NSF) grants IIS-1348152 and ACI-1338745, Defense Advanced Research Projects Agency (DARPA) XDATA program grant FA8750-12-2-0309. We also thank NSF for the travel grant to present this work in the conference through the grant CCF-1552229. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the USDOE, NERSC, AFOSR, NSF or DARPA.

References

[1] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz. Brief announcement: Hypergraph partitioning for parallel sparse matrix-matrix multiplication. In *Proceedings of SPAA*, pages 86–88, 2015. URL http://doi.acm.org/10.1145/2755573.2755613.

[2] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007. URL http://dx.doi.org/10.1002/cpe.1206.

[3] A. Cichocki, R. Zdunek, A. H. Phan, and S.-i. Amari. *Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation*. Wiley, 2009.

[4] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *Proceedings of IPDPS*, pages 261–272, 2013. URL http://dx.doi.org/10.1109/IPDPS.2013.80.

[5] J. P. Fairbanks, R. Kannan, H. Park, and D. A. Bader. Behavioral clusters in dynamic graphs. *Parallel Computing*, 47:38–50, 2015. URL http://dx.doi.org/10.1016/j.parco.2015.03.002.

[6] C. Faloutsos, A. Beutel, E. P. Xing, E. E. Papalexakis, A. Kumar, and P. P. Talukdar. Flexi-FaCT: Scalable flexible factorization of coupled tensors on Hadoop. In *Proceedings of the SDM*, pages 109–117, 2014. URL http://epubs.siam.org/doi/abs/10.1137/1.9781611973440.13.

[7] R. Fujimoto, A. Guin, M. Hunter, H. Park, G. Kanitkar, R. Kannan, M. Milholen, S. Neal, and P. Pecher. A dynamic data driven application system for vehicle tracking. *Procedia Computer Science*, 29: 1203–1215, 2014. URL http://dx.doi.org/10.1016/j.procs.2014.05.108.

[8] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the KDD*, pages 69–77. ACM, 2011. URL http://dx.doi.org/10.1145/2020408.2020426.

[9] D. Grove, J. Milthorpe, and O. Tardieu. Supporting array programming in X10. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 38:38–38:43, 2014. URL http://doi.acm.org/10.1145/2627373.2627380.

[10] N.-D. Ho, P. V. Dooren, and V. D. Blondel. Descent methods for nonnegative matrix factorization. *CoRR*, abs/0801.3199, 2008.

[11] P. O. Hoyer. Non-negative matrix factorization with sparseness constraints. *JMLR*, 5:1457–1469, 2004. URL www.jmlr.org/papers/volume5/hoyer04a/hoyer04a.pdf.

[12] O. Kaya and B. Uçar. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of SC*, pages 77:1–77:11. ACM, 2015. URL http://doi.acm.org/10.1145/2807591.2807624.

[13] H. Kim and H. Park. Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis. *Bioinformatics*, 23(12):1495–1502, 2007. URL http://dx.doi.org/10.1093/bioinformatics/btm134.

[14] J. Kim and H. Park. Fast nonnegative matrix factorization: An active-set-like method and comparisons. *SIAM Journal on Scientific Computing*, 33(6):3261–3281, 2011. URL http://dx.doi.org/10.1137/110821172.

[15] J. Kim, Y. He, and H. Park. Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework. *Journal of Global Optimization*, 58(2):285–319, 2014. URL http://dx.doi.org/10.1007/s10898-013-0035-4.

[16] D. Kuang, C. Ding, and H. Park. Symmetric nonnegative matrix factorization for graph clustering. In *Proceedings of SDM*, pages 106–117, 2012. URL http://epubs.siam.org/doi/pdf/10.1137/1.9781611972825.10.

[17] D. Kuang, S. Yun, and H. Park. SymNMF: nonnegative low-rank approximation of a similarity matrix for graph clustering. *Journal of Global Optimization*, pages 1–30, 2013. URL http://dx.doi.org/10.1007/s10898-014-0247-2.

[18] R. Liao, Y. Zhang, J. Guan, and S. Zhou. CloudNMF: A MapReduce implementation of nonnegative matrix factorization for large-scale biological datasets. *Genomics, proteomics & bioinformatics*, 12(1): 48–51, 2014. URL http://dx.doi.org/10.1016/j.gpb.2013.06.001.

[19] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on MapReduce. In *Proceedings of the WWW*, pages 681–690. ACM, 2010. URL http://dx.doi.org/10.1145/1772690.1772760.

[20] E. Mejía-Roa, D. Tabas-Madrid, J. Setoain, C. García, F. Tirado, and A. Pascual-Montano. NMF-mGPU: non-negative matrix factorization on multi-GPU systems. *BMC bioinformatics*, 16(1):43, 2015. URL http://dx.doi.org/10.1186/s12859-015-0485-4.

[21] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine Learning in Apache Spark, May 2015. URL http://arxiv.org/abs/1505.06807.

[22] V. P. Pauca, F. Shahnaz, M. W. Berry, and R. J. Plemmons. Text mining using nonnegative matrix factorizations. In *Proceedings of SDM*, 2004.

[23] C. Sanderson. Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, 2010. URL http://arma.sourceforge.net/armadillo_nicta_2010.pdf.

[24] D. Seung and L. Lee. Algorithms for non-negative matrix factorization. *NIPS*, 13:556–562, 2001.

[25] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005. URL http://hpc.sagepub.com/content/19/1/49.abstract.

[26] Y.-X. Wang and Y.-J. Zhang. Nonnegative matrix factorization: A comprehensive review. *TKDE*, 25(6):1336–1353, June 2013. URL http://dx.doi.org/10.1109/TKDE.2012.51.

[27] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009.

[28] Z. Xianyi. Openblas, Last Accessed 03-Dec-2015. URL http://www.openblas.net.

[29] J. Yin, L. Gao, and Z. Zhang. Scalable nonnegative matrix factorization with block-wise updates. In *Machine Learning and Knowledge Discovery in Databases*, volume 8726 of *LNCS*, pages 337–352, 2014. URL http://dx.doi.org/10.1007/978-3-662-44845-8_22.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10. USENIX Association, 2010. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.